ABSTRACT

Title of dissertation: PARALLEL ALGORITHMS FOR PROCESSING MASSIVE TEXTS AND GRAPHS

Hamed Saleh, Doctor of Philosophy, 2023

Dissertation directed by: Professor MohammadTaghi Hajiaghayi
University of Maryland Department of Computer Science

Designing parallel algorithms with sublinear space is an inevitable scenario when the input data 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 domain vary from social network graphs to DNA sequences. However, to address any problem subject to this restriction efficiently, we need alternative models of computation as the traditional RAM model is no longer an option. The main focus of our research is studying classical algorithms on texts and graphs in the *Massively Parallel Computation* model.

During the last decade, the *Massively Parallel Computation* (MPC) model attracted a considerable amount of attention. The MPC model was originally proposed to provide a theoretical foundation to algorithms implemented on modern large scale data processing frameworks such as MapReduce, Hadoop, and Spark. The key idea behind this model, and also aforementioned frameworks, is to use many machines to compensate for the shortage of space in individual machines. In this model, the data is distributed among a set of machines each with a sublinear memory, and the process is consisted of 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. We propose constant rounds MPC algorithms for *Edge Coloring*, *Pattern Matching*, constructing *Suffix Trees*, *Longest Common Substring*, and *Longest Palindrome Substring*.

Recently, the *Adaptive Massively Parallel Computation* (AMPC) model was introduced as an extension of MPC. Despite the significant progress in designing highly efficient MPC algorithms, the community is facing an impenetrable barrier due to a hardness conjecture that *Graph Connectivity* requires at least a logarithmic number of rounds. Inspired by this limitation, the AMPC model adds distributed hash-tables to the setup where each machine can adaptively read from them during each round. This tweak not only leads to a constant-round *Graph Connectivity* algorithm and a plethora of logarithmic improvements in many graph problems, but it is also practical thanks to the recent developments in hardware infrastructure and technologies such as RDMA and eRPC. We explore the limitations of this model by introducing a general *Tree Contraction* framework which translates well-studied applications in the context of PRAM into constant-round AMPC algorithms, and solving the *Minimum Cut* problem in sublogarithmic rounds.

PARALLEL ALGORITHMS FOR PROCESSING MASSIVE TEXTS AND GRAPHS

by

Hamed Saleh

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2023

Advisory Committee:
Professor MohammadTaghi Hajiaghayi, Chair/Advisor
Professor Alexander Barg
Professor David Mount
Professor Barna Saha
Professor Ming Lin
Professor Soheil Feizi

Preface

When designing sequential algorithms, we often take it for granted that we have a random access to any part of input at any time. Furthermore, we assume that accessing any part of input takes unit amount of time. There is some truth behind this assumption since our computers are designed to work at certain time steps, called clocks, and the CPU is supposed to wait until the current memory read query succeed before moving on to the next instruction. However, any seasoned programmer knows that reading a number of consecutive bytes in memory is much faster than reading the same number of bytes that are scattered through the memory. There are inherent irregular complexities in computer hardwares, and the programmers utilize their intuition and the vague memories of Computer Architecture course to optimize the running time of their code.

The algorithm designers, on the other hand, need to rely on rigorous mathematical foundations to avoid unnecessary complexities in their calculations. Algorithms must not depend on how the computer hardwares are built currently, and the best an algorithm designer can do to catch up with the recent developments in computing paradigms is to use new models of computation. And once in a while, a group of computer scientists, including Computer Architecture experts, Algorithm Designers, and Programmers, gather together to define a new model that represents the computational power of the current computer systems which was not explainable with the existing models. The creation of MPC, short for *Massively Parallel Computation*, loosely followed the same story. It may not be the best approach to analyze parallel algorithms, but it fills the void for a simple and almost-practical model that we need for understanding the parallelizability of algorithms.

Dedication

To Mahsa Amini and the brave people of Iran who are fighting for their freedom and honor relentlessly.

Acknowledgments

I want to thank my advisor Mohammad Hajiaghayi who always supported me through these years. I am grateful to other members of my advisory committee Alexander Barg, David Mount, Barna Saha, Ming Lin, and Soheil Feizi. Many results in this dissertation are part of joint research with multiple co-authors, namely Soheil Behnezhad, Debarati Das, Mahsa Derakhshan, Jacob Gilbert, MohammadTaghi HajiAghayi, Marina Knittel, Tomasz Kociumaka, Jan Olkowski, Barna Saha, Saeed Seddighin, Hsin-Hao Su, and Xiaouri Sun.

Table of Contents

Chapter 1: Introduction

We start by introducing the concept of *Massively Parallel Computation*, or MPC for short. The origins of this model dates back to 2010 when Karloff et al. [95] introduced the MRC model as an abstraction of MapReduce framework. Despite notational differences, the MRC model has the same computation power as MPC. Notably, the MRC model remains faithful to the MapReduce framework by explicitly considering map, shuffle, and reduce stages. The main contribution of Karloff et al. [95] was a reduction from any PRAM algorithm to an MRC algorithm with the same number of rounds, through which the vast literature of parallel PRAM algorithms is automatically translated to counterpart MRC algorithms. Many important techniques and building blocks such as Filtering [102] and Sorting [76] were soon discovered for more relaxed variants of MRC. Through a line of works by Goodrich et al [76], Andoni et al [6], and Beame et al [22] the current formulation of MPC crystallized and it gradually became the de-facto standard model for parallel/distributed computing.

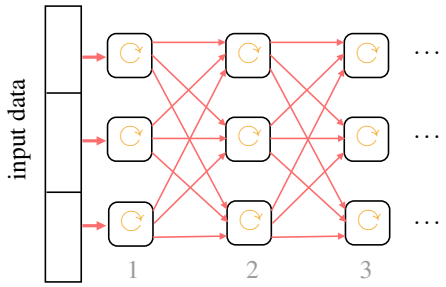
The main incentive for designing algorithms in the MPC model is that it captures the computation power of modern parallel/distributed frameworks such as MapReduce, Hadoop, Spark, and Flume. Basically, many MPC algorithms can be implemented in the mentioned frameworks. A common feature of these frameworks is that both data and computation are scattered through a pool of machines with limited memory. The memory of each machine S must be sublinear in the input size, which is a sequence of N words, so let $\mathsf{S} = O(N^{\varepsilon})$ for some $0 < \varepsilon < 1$. The input is initially distributed, maybe randomly, among

$\mathsf{M} = O(N^{1-\varepsilon})$ machines, and ideally we use asymptotically the same number of machines during the algorithm process.

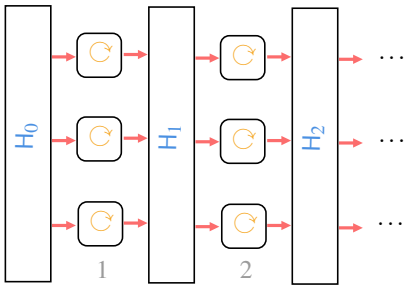
An MPC algorithm consists of several rounds. During each round, every machine performs a polynomial time computation on its local data. At the end of each round, machines can communicate with each other. This communication is all-to-all and the only constraint is that the total amount of incoming/outgoing words to/from each machine must be limited by $O(\mathsf{S})$. We aim to minimize the number of rounds since in practice the main bottleneck is the communication phase. Translating PRAM algorithms to MPC gives us polylogarithmic number of rounds for many problems. However, for some problems we can design sublogarithmic MPC algorithms which is not possible in PRAM. Many graph problems such as maximal matching, maximum matching approximation, maximal independent set, vertex coloring, edge coloring, etc. have sublogarithmic MPC algorithms. On the other side of spectrum, Dynamic programming problems such as Longest Common Substring and Edit Distance have very efficient (sometimes $O(1)$ rounds) algorithms. Other problems considered in this manuscript are Minimum Cut, Suffix Tree, Tree Contraction, and String Matching.

Other Variants of MPC: In literature other variants of the MPC model is considered. In the original MRC model, the upper bound on both space per machine and number of machines is $O(n^{1-\varepsilon})$ for some constant $0 < \varepsilon < 1$. Many results solved in this memory regime were further improved to tighter bounds on memory. The superlinear local memory regime of $\mathsf{S} = O(n^{1+\varepsilon})$, or $\mathsf{S} = O(n^{1+\varepsilon} + m)$ for some graph problems, was first considered by Lattenzi et al [102] in which they showed $O(1)$ round MPC algorithms for maximal matching, maximal independent set, etc. using the Filtering technique. The next line of work considered linear local memory $O(n)$ for mostly graph problems in which the total memory is $O(n+m)$, which can be up to $O(n^2)$. Maximal matching and maximal inde-

pendent set were solved in $O(\log\log n)$ rounds in this linear memory regime [31, 49]. The most general regime of memory in MPC is the sublinear memory of $\mathsf{S} = O(N^{\varepsilon})$ for any $0 < \varepsilon < 1$ where the number of machines is equal to $\mathsf{M} = O(N^{1-\varepsilon})$. Here N shows the size of input so that $N = O(n+m)$ for graph problems.



(a) In MPC, the input data is initially distributed among the machines (usually at random). At each round machines perform local computations, and at the end of a round any machine can send messages to any other machine.



(b) In AMPC, we have distributed hash tables $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2, \ldots$ in addition to machines. Every machine at round i has adaptive read access to H_{i-1} during local computations, and at the end of round i writes to H_i.

Figure 1.1: A schematic comparison of MPC and AMPC models using three machines. Each column of rounded rectangles corresponds to the set of machines in a specific round (the round number is indicated below each column of machines with gray color). The yellow circular arrows represent computations on local data, and the red arrows represent the communications. Note that distributed hash tables in the AMPC model are sufficient for simulating MPC algorithms. We can store the input data in H_0, and $\mathsf{H}_1, \mathsf{H}_2, \ldots$ are used for machine-machine communications.

The power of MPC on graph problems in many cases is limited due to a hardness conjecture called "1cycle-2cycle". According to this conjecture, distinguishing between a cycle of length n and two cycles of length $n/2$ needs at least $\Omega(\log n)$ rounds in MPC. In essence, "1cycle-2cycle" is a simpler version of the Connectivity problem, meaning that if we aim to solve Connectivity in sublogarithmic rounds, first we need to solve "1cycle-2cycle" in sublogarithmic rounds. Since many global graph properties require Connectivity, this conjecture gives a lower bound for other graph problems as well.

AMPC: The AMPC model, introduced by Behnezhad et. al [29], is an extension of the standard MPC model with additional access to a *distributed hash table*. In MPC, data is initially distributed across machines and then computation proceeds in rounds where machines execute local computations and then are able to share small messages with each other before the next round of computation. A distributed hash table stores a collection of key-value pairs which are accessible from every machine, and it is required that both key and value have a constant size. Each machine can adaptively query a bounded sequence of keys from a centralized distributed hash table during each round, and write a bounded number of key-value pairs to a distinct distributed hash table which is accessible to all machines in the next round. The distributed hash tables can also be utilized as the means of communication between the machines, which is implicitly handled in the MPC model, as well as a place to store the initial input of the problem. It is straight-forward to see how every MPC algorithm can be implemented within the same guarantees for the round-complexity and memory requirements in the AMPC model.

Formally speaking, in the AMPC model, there are $\mathsf{M} = O(n^{1-\varepsilon})$ machines each with **sublinear** local space $\mathsf{S} = O(n^{\varepsilon})$ for some constant $0 < \varepsilon < 1$. In addition, there exist a collection of distributed hash tables $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2, \ldots$, where H_0 contains the initial input. The process consists of several rounds. During round i, each machine is allowed to make at most $O(\mathsf{S})$ read queries from H_{i-1} and to write at most $O(\mathsf{S})$ key-value pairs to H_i. Meanwhile, the machines are allowed to perform an arbitrary amount of computation locally. Therefore, it is possible for machines to decide what to query next after observing the result of previous queries. In this sense, the queries in this model are **adaptive**.

1.1 A Brief History of Parallel Models

Many algorithms that we use on a daily basis are sequential. The *Random Access Machine* model, or RAM [1] for short, was introduced by Cook and Reckhow [48] to act as a basis for analyzing sequential algorithms that could be implemented on the computers at the time. This model is a relatively realistic machine model to compensate for the shortcomings of older and somewhat unrealistic mathematical models such as Turing Machines in analyzing time and memory complexity. Roughly speaking, there is an infinite array of registers in this model, which resembles Random Access Memory, and a single processor can read from and write to any register in unit time. The **memory** complexity is measured by the number of registers we use throughout the algorithm, and the **time** complexity is proportional to the total units of time we spend to calculate the output.

The distinguishing factor between sequential and parallel models is the multiplicity of processors. The *Parallel Random Access Machine* model, or PRAM, incorporates an array of M processors $p_1, p_2, \ldots, p_{\mathsf{M}}$ that simultaneously work on a **shared memory**. Similar to RAM, the memory is simulated by an infinite array of registers, and the memory complexity is measured by the number of registers the algorithm uses. This memory, however, is shared among all processors. To avoid race conditions in case of concurrent reads and writes, different policies are established leading to numerous variants of PRAM in the literature. Based on whether reads and writes are exclusive or concurrent, i.e., whether each register can be accessed by only one or multiple processors at a single unit of time, PRAM variants are grouped into four categories:

- EREW (Exclusive Read, Exclusive Write): Each register is constrained to be read and be written only by one processor at a single unit of time.

[1] Not to be confused with Random Access Memory

- CREW (Concurrent Read, Exclusive Write): No two processors can write to the same register, but any number of processors can read from the same register.
- CRCW (Concurrent Read, Concurrent Write): There are no constraints on memory access, and any number of processors can read from or write to a single register at the same time.[2]
- ERCW (Exclusive Read, Concurrent Write): Defined just for the sake of completeness.

One can verify that any EREW algorithm is also a CRCW algorithm, but the reverse is not necessarily true. Thus, we conclude that CRCW PRAM model is a stronger model than EREW PRAM since the same problem potentially admits more efficient algorithms in terms of time and memory complexity in CRCW PRAM. Regardless of the memory access policy, a PRAM algorithm reads the input from the shared memory at the beginning and writes the output to the shared memory when it finishes the computations. At one unit of time, which we also call **rounds** alternatively, processors synchronously perform either memory read, memory write, or basic arithmetic computations using their constant number of local registers.

The running time of a PRAM algorithm is defined as the number of rounds it takes to compute the output. It is worth noting that the concept of running time in parallel algorithms is subtly different from sequential algorithms. The running time of parallel algorithms, also referred to as parallel-time in literature, is the time spent while all processors work together to compute the output, which could be much smaller the size of input unlike sequential algorithms whose running time is lower-bounded by the size of input. Another

[2]Since the value of a register where multiple processors attempt to write it with different values is ambiguous, various strategies such as "common" and "priority" are considered to determine the value of such registers. In the former, all processors are obliged to write the same value to the register, and in the latter, processors are assigned priorities and the value written by the processor with the highest priority overwrites others.

perspective is to look at PRAM algorithms as circuits where each operation is simulated by a logical gate, and communications are modeled by wires. One can observe that the running time equals the depth of the circuit, whereas the running time of the same algorithm simulated sequentially in RAM equals the size of the circuit. This means that the concept of running time is not comparable between PRAM and RAM, which motivates us to define another factor called **work** in context of PRAM as the number of rounds multiplied by memory. Note that the running time of the sequential version of a PRAM algorithm simulated in RAM is comparable to the work.

For a long time, PRAM was considered the standard model for analyzing parallel algorithms. Although constant-round CRCW PRAM algorithms exist at the cost of blowup in the number of processors, possibly quadratic number of processors, we usually face a logarithmic round barrier in PRAM if we restrict ourselves to linear number of processors. Miller and Reif Tree Contraction [113] is an example of logarithmic round PRAM framework for solving problems on trees. Even basic problems such as finding the sum of n integers also require $\Omega(\log n)$ rounds in PRAM. Another drawback of PRAM is practicality. It is possible to produce specialized hardware capable of simulating PRAM algorithms, but our typical computers are not designed for this purpose.

The crux of MapReduce framework was taking advantage of parallelism without dependence on any special hardware. Clusters made of thousands of typical computers were able to process large-scale datasets using the new easy-to-use software framework which guaranteed fault-tolerance and fast communication. [51] It was successful in restricting the flexibility of user while preserving the computational power of a large cluster of computer working in parallel. The data is represented in sequences of $\langle \mathsf{key}, \mathsf{value} \rangle$ pairs, and the process consists of synchronized repetitions of three stages: **map**, **shuffle**, and **reduce**. The user provides two functions for map and reduce stages, and the shuffle stage is implicitly handled by the framework. In the map stage each $\langle \mathsf{key}, \mathsf{value} \rangle$ pair is converted to a new

pair with a possibly different format, the shuffle stage aggregates pairs with the same key into the same machine, and the reduce stage processes the set of pairs with the same key and produces a new list of pairs.

Some years after the introduction of MapReduce, it was realized that a large amount of natural large-scale data processing needs can be translated into the language of map and reduce functions, and consequently be implemented in the framework. This lead to formalizing the MapReduce computational power as a mathematical machine model for analyzing parallel algorithms. Karloff et al. [95] introduced MRC model as an abstract model based off MapReduce computing paradigm. In this model, M machines $\mu_1, \mu_2, \ldots, \mu_{\mathsf{M}}$ work in parallel in synchronous rounds that accomplish map, shuffle, and reduce stages of MapReduce. Each machine μ_i is a bounded-memory RAM instance consisted of a processor p_i and an array of S registers which we call **local memory**. The input is initially partitioned into chunks of size $O(\mathsf{S})$ and each chunk is stored in the local memory of one machine. In some cases, we assume the input elements are randomly distributed into different machines but it is straight-forward to see that this can be done in one round by mapping each input pair to a new pair with a key randomly chosen from $[1, \mathsf{M}]$.

1.2 Summary of Contributions

Probably the most effective tool for solving problems in MPC is careful **partitioning** of the input into smaller subproblems. By choosing the proper partitioning scheme we could reduce the size of problem, solve the subproblems sequentially in each machine, and aggregate the results to compute the final output. Although the overall process seems like a standard divide-and-conquer approach many subtleties arise while completing the big picture. In this section, we provide a summary of contributions through the lens of partitioning techniques we used in our publications.

1.2.1 Graph Partitioning

Given a graph $G = (V,E)$ made of n vertices and m edges with no further properties, we have limited options for partitioning. In literature, random vertex partitioning and random edge partitioning are studied extensively. The randomness plays an important part especially in vertex partitioning. We can observe that if we randomly partition vertices into a number of groups, the number of edges within each partition is concentrated due to Chernoff Bound. This allows us to confidently send the induced subgraph of each partition to a machine without worrying about memory blowup since we can bound the maximum number of edges in each subgraph with high probability. Variants of this argument appear in many algorithms which are based on vertex partitioning. Behnezhad et al [31], for example, utilized this argument in their vertex partitioning approach for solving maximal matching in $O(\log\log n)$ rounds of linear-memory MPC.

Another powerful tool in graph MPC algorithms is **sparsification**. Based on the nature of the problem we can select a subset of important edges and discard the others, while guaranteeing an approximation of the result is preserved. In conjunction with a suitable partitioning, sparsification plays an important role in designing MPC algorithms. The fil-

tering method of Lattenzi et al [102] uses problem-dependent sparsification along with random edge partitioning of graph to solve problems such as Minimum Spanning Tree, Maximal Matching, Approximate Weighted Matching, and Minimum Cut in superlinear memory regime of MPC, i.e., $\mathsf{S} = \Omega(n^{1+\varepsilon})$. The idea is to partition the edges of a graph randomly among the machines, and discarding unimportant edges by solving the problem locally on each subgraph.

In Appendix A, the problem of Edge Coloring in MPC is considered. The partitioning approach used in this paper is a combination of vertex and edge partitioning. To partition the edges of the input graph, we first partition the vertices randomly into k groups. Then, the induced subgraph of each $\binom{k}{2}$ pairs of these groups is sent to a separate machine. This partitioning scheme has structural properties which we utilize in solving the Edge Coloring problem.

1.2.2 Tree Partitioning

Trees lie in the intersection of graphs and sequences. We have more flexibility when working with trees, and at the same time we face more challenges as we aim to solve harder problems. There are numerous ways to partition the trees, but we still face a logarithmic barrier in MPC since usually we need to integrate a global property of a tree and parallel tree traversal approaches require logarithmic rounds. Bateni et al [20] introduce a general approach for solving Dynamic Programming formulations defined on trees. They devise a low-depth decomposition of trees by recursively finding a centroid, a vertex whose removal divides the tree into balanced subgraphs. The recursion has $O(\log n)$ levels, and therefore in $O(\log n)$ rounds of MPC we could traverse the tree. Tree contraction is another approach for traversing a tree in $O(\log n)$ levels introduced by Miller and Reif [113] for PRAM. A formal definition can be found in Subsection C.2.1, but intuitively we alternate between

compressing long paths and removing all the leaves, and one can show that after $O(\log n)$ repetitions we end up with a single vertex. Again, $O(\log n)$ rounds of MPC are necessary for simulating Tree Contractions.

In Appendix C, we study the Tree Contraction method for solving tree problems in AMPC. Even though AMPC is much more powerful than MPC, classical tree traversals still require logarithmic number of rounds in AMPC. We introduce two partitioning methods that together lead to a constant-round AMPC algorithm for Tree Contraction. First, pre-order decomposition partitions vertices of a degree-bounded tree into almost equal-sized chunks based on their index in the pre-order traversal. By contracting connected components induced in each partition, we end up with a smaller tree that after removing all the leaves the number of vertices drops by a factor of $O(\mathsf{S}) = O(n^{\varepsilon})$. Second, we devise a generalization of Tree Contraction method by relaxing the COMPRESS stage to contract every vertex with degree at most $O(n^{\varepsilon})$. The main property of the generalized formulation of Tree Contraction is that the depth of recursion is decreased from $O(\log n)$ to $(1/\varepsilon)$. By combining these two partitioning method, more precisely using the first one as a subroutine of the second one, we end up with a tree traversal method capable of solving problems solvable by Tree Contraction approach in $O(1/\varepsilon^3)$ rounds of AMPC.

The partitioning method used in Appendix B for solving cut problems in AMPC is a generalization of low-depth tree decomposition. Intuitively, we find the heavy-light decomposition of a Minimum Spanning Tree and replace each heavy path with a binary tree. Using the previous results regarding trees in AMPC, we can find this $O(\log^2 n)$-level decomposition of trees in constant rounds of AMPC. This partitioning still has logarithmic depth, and solving the min cut recursively similar to previous is still not an option for us. Therefore, we decouple the processing of different levels of this decomposition and solve them in parallel. This will lead to a sublogarithmic algorithm finding minimum cut of a graph.

1.2.3 Sequence Partitioning

For partitioning a sequence the most straight-forward approach may be dividing it into consecutive blocks that could fit in the memory of a single machine. Since the only kind of sequences that we studied in this manuscript are strings, here we focus only on strings, sequences of characters from an alphabet Σ. Appendix D studies the problem of String Matching in presence of different types of wildcard characters. When there are no wildcards, partitioning the string into blocks of equal size is enough for solving the problem in constant rounds of MPC. We complement the block partitioning by computing the hash of every prefix in each block. This data structure facilitates the process of querying the hash of any substring of our string. The general form of such block-based data structures is formalized in Section E.1.3.

In Appendix E, we aim to find the suffix tree of a given string in constant rounds of MPC. At the core of our algorithm is LCPQ oracle which finds the LCP (Longest Common Prefix) of two substrings in parallel. In making of LCPQ oracle, we used a distinct type of partitioning in addition to block-based data structure. To approximate the length of LCP up to $O(n^{\varepsilon})$ additive error, we utilize Modular Partitioning. More details can be found in Section E.1.3. Roughly speaking, we first find the hash of each substring of length $O(n^{\varepsilon})$, and then partition them based on the modulo of the index of start in division by $O(n^{\varepsilon})$. This partitioning scheme gives us the ability to quickly estimate the length of LCP, and then we again use block-based data structures to find the precise length.

Part I

Graph Problems

Appendix A: Edge Coloring

A.1 Introduction

Given a graph $G(V,E)$, an edge coloring of G is an assignment of "colors" to the edges in E such that no two incident edges receive the same color. The goal is to find an edge coloring that uses few colors. Edge coloring is among the most fundamental graph problems and has been studied in various models of computation, especially in distributed and parallel settings.

Denoting the maximum degree in the graph by Δ, it is easy to see that Δ colors are necessary in any proper edge coloring. On the other hand, Vizing's celebrated theorem asserts that $\Delta+1$ colors are always sufficient [133]. While determining whether a graph can be Δ colored is NP-hard, a $\Delta+1$ coloring can be found in polynomial time [10, 61]. These algorithms are, however, highly sequential. As a result, in restricted settings, it is standard to consider more relaxed variants of the problem where more colors are allowed [5, 16, 60, 72, 73, 82, 90, 100, 104, 107, 119].

In this paper, we study edge coloring in large-scale graph settings. Specifically, we focus on the *Massively Parallel Computations* (MPC) model and the *Graph Streaming* model.

A.1.1 Massively Parallel Computation

The Model. The MPC model [22, 76, 95] is a popular abstraction of modern parallel frameworks such as MapReduce, Hadoop, Spark, etc. In this model, there are N machines, each with a space of S words[1] that all run in parallel. The input, which in our case is the edge-set of graph $G(V,E)$, is initially distributed among the machines arbitrarily. Afterwards, the system proceeds in synchronous rounds wherein the machines can perform any arbitrary local computation on their data and can also send messages to other machines. The messages are then delivered at the start of the next round so long as the total messages sent and received by each machine is $O(S)$ for local machine space S. The main parameters of interest are S and the round-complexity of the algorithm, i.e., the number of rounds it takes until the algorithm stops. Furthermore, the total available space over all machines should ideally be linear in the input size, i.e., $S \cdot N = O(|E|)$.

Related Work in MPC. We have seen a plethora of results on graph problems ever since the formalization of MPC. The studied problems include matching and vertex cover [4, 11, 26, 31, 36, 49, 65, 102], maximal independent set [26, 36, 65, 83], vertex coloring [12, 65, 83, 121, 122], as well as graph connectivity and related problems [6, 7, 18, 28, 91]. (This is by no means a complete list of the prior works.)

We have a good understanding of the complexity of vertex coloring in the MPC model, especially if the local space is near linear in n: Assadi et al. [12] gave a remarkable algorithm that using $\widetilde{O}(n)$ space per machine, finds a $(\Delta+1)$ vertex coloring in a constant number of rounds. The algorithm is based on a sparsification idea that reduces the number of edges from m to $O(n\log^2 n)$. But this algorithm alone cannot be used for coloring the edges, even if we consider the more relaxed $(2\Delta-1)$ edge coloring problem which is

[1] Throughout the paper, the stated space bounds are in the number of words that each denotes $O(\log n)$ bits.

equivalent to $(\Delta+1)$ vertex coloring on the line graph. The reason is that the line-graph has $O(m)$ vertices where here m is the number of edges in the original graph. Therefore even after the sparisification step, we have $\widetilde{O}(m)$ vertices in the graph which is much larger than the local space available in the machines.

Not much work has been done on the edge coloring problem in the MPC model. The only exception is the algorithm of Harvey *et al.* [83] which roughly works by random partitioning the *edges*, and then coloring each partition in a different machine using a sequential $(\Delta+1)$ edge coloring algorithm. The choice of the number of partitions leads to a trade-off between the number of colors used and the space per machine required. The main shortcoming of this idea, however, is that if one desires a $\Delta+\widetilde{O}(\Delta^{1-\Omega(1)})$ edge coloring, then a strongly super linear local space of $n\Delta^{\Omega(1)}$ is required.

Our main MPC result is the following algorithm which uses a more efficient partitioning. The key difference is that we use a *vertex* partitioning as opposed to the algorithm of Harvey *et al.* which partitions the edges.

Result 1 (Theorem A.2.1)**.** There exists an MPC algorithm that using $O(n)$ space per machine and $O(m)$ total space, returns a $\Delta+\widetilde{O}(\Delta^{3/4})$ edge coloring in $O(1)$ rounds.

The algorithm exhibits a tradeoff between the space and the number of colors (see Theorem A.2.1) and can be made more space-efficient as the maximum degree gets larger. For instance, if $\Delta > n^{\varepsilon}$ for any constant $\varepsilon > 0$, it requires a strictly sublinear space of $n^{1-\Omega(1)}$ to return a $\Delta+o(\Delta)$ edge coloring in $O(1)$ rounds. This is somewhat surprising since all previous non-trivial algorithms in the strictly sublinear regime of MPC require $\omega(1)$ rounds.

Our algorithm can also be implemented in $O(1)$ rounds of *Congested Clique*, leading to a $\Delta+\widetilde{O}(\Delta^{3/4})$ edge coloring there. Prior to our work, no sublogarithmic round Congested Clique algorithm was known even for $(2\Delta-1)$ edge coloring.

A.1.2 Streaming

The Model. In the standard graph streaming model, the edges of a graph arrive one by one and the algorithm has a space that is much smaller than the total number of edges. A particularly important choice of space is $\widetilde{O}(n)$—which is also known as the *semi-streaming* model [58]—so that the algorithm has enough space to store the vertices but not the edges. For edge coloring, the output is as large as the input, thus, we cannot hope to be able to store the output and report it in bulk at the end. For this, we consider a standard twist on the streaming model where the output is also reported in a streaming fashion. This model is referred to in the literature as the "W-streaming" model [53, 71]. We particularly focus on one-pass algorithms.

Designing one-pass W-streaming algorithms is particularly challenging since the algorithm cannot "remember" all the choices made so far (e.g., the reported edge colors). Therefore, even the sequential greedy algorithm for $(2\Delta-1)$ edge coloring, which iterates over the edges in an arbitrary order an assigns an available to each color upon visiting it, cannot be implemented since we are not aware of the colors used incident to an edge.

Our first result is to show that a natural algorithm w.h.p.[2] provides an $O(\Delta)$ edge coloring if the edges arrive in a random-order.

Result 2 (Theorem A.3.2)**.** If the edges arrive in a random-order, there is a one-pass $\widetilde{O}(n)$ space W-streaming edge coloring algorithm that always returns a valid edge coloring and w.h.p. uses $(2e+o(1))\Delta \approx 5.44\Delta$ colors.

If the edges arrive in an arbitrary order, we give another algorithm that requires more colors.

[2]Throughout, we use "w.h.p." to abbreviate "with high probability" implying probability at least $1-1/\operatorname{poly}(n)$.

Result 3 (Theorem A.3.3)**.** For any arbitrary arrival of edges, there is a one-pass $\widetilde{O}(n)$ space W-streaming edge coloring algorithm that succeeds w.h.p. and uses $O(\Delta^2)$ colors.

These are, to our knowledge, the first streaming algorithms for edge coloring.

A.2 The MPC Algorithm

In this section, we consider the edge coloring problem in the MPC model. Our main result in this section is an algorithm that achieves the following:

Theorem A.2.1. *For any parameter k (possibly dependent on Δ) such that $n/k \gg \log n$, there exists an MPC algorithm with $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ space per machine and $O(m)$ total space that w.h.p. returns a $\Delta + O(\sqrt{k\Delta \log n})$ edge coloring in $O(1)$ rounds.*

By setting $k = \sqrt{\Delta} + \log n$, the space required per machine will be $O(n)$ and the number of colors would be $\Delta + \widetilde{O}(\Delta^{3/4})$. Using a reduction from [25], this also leads to an $O(1)$ round Congested Clique algorithm using the same number of colors.

Corollary A.2.2. *There exists a randomized MPC algorithm with $O(n)$ local space, as well as a Congested Clique algorithm, that both w.h.p. find a $\Delta + \widetilde{O}(\Delta^{3/4})$ edge coloring in $O(1)$ rounds.*

Moreover, assuming that $\Delta = n^{\Omega(1)}$, by setting $k = \Delta^{0.5+\varepsilon}$ for a small enough constant $\varepsilon \in (0,1)$, we get the following $O(1)$ round algorithm which requires $n^{1-\Omega(1)}$ machine space, which is notably strictly sublinear in n:

Corollary A.2.3. *If $\Delta = n^{\Omega(1)}$, there exists a randomized MPC algorithm with $O(n/\Delta^{2\varepsilon}) = n^{1-\Omega(1)}$ space per machine and $O(m)$ total space that w.h.p. returns a $\Delta + \widetilde{O}(\Delta^{0.75+\varepsilon/2})$ edge coloring in $O(1)$ rounds.*

The Idea Behind the Algorithm. The first step in the algorithm is a random partitioning of the *vertex set* into k groups, $V_1, \ldots, V_k$. We then introduce one subgraph for each vertex subset, called $G_1, \ldots, G_k$, and one subgraph for every pair of groups which we denote as $G_{1,2}, \ldots, G_{1,k}, \ldots, G_{k-1,k}$. Any such G_i is simply the induced subgraph of G on V_i. Moreover, any such $G_{i,j}$ is the subgraph on vertices $V_i \cup V_j$, with edges with one point in V_i and the other in V_j.

The general idea is to assign different *palettes*, i.e., subsets of colors, to different subgraphs so that the palettes assigned to any two neighboring subgraphs (i.e., those that share a vertex) are completely disjoint. A key insight to prevent this from blowing up the number of colors, is that since any two edges from $G_{i,j}$ and $G_{i',j'}$ with $i \neq i'$ and $j \neq j'$ cannot share endpoints by definition, it is safe to use the same color palette for them.

To assign these color palettes, we consider a complete k-vertex graph with each vertex v_i in it corresponding to partition V_i and each edge (v_i, v_j) in it corresponding to the subgraph $G_{i,j}$. We then find a k edge coloring of this complete graph, which exists by Vizing's theorem since maximum degree in it is $k-1$. This edge coloring can actually be constructed extremely efficiently using merely the edges' endpoint IDs. Thereafter, we map each of these k colors to a color palette. By carefully choosing k and the number of colors in each palette, we ensure that: (1) The total number of colors required is close to Δ. (2) Each subgraph $G_{i,j}$ can be properly edge-colored with those colors in its palette. (3) Each subgraph fits the memory of a single machine so that we can put it in whole there and run the sequential edge coloring algorithm on it.

The algorithm outlined above is formalized as Algorithm 1. We start by proving certain bounds on subgraphs' size and degrees.

Claim A.2.4. *W.h.p., every subgraph of type G_i or $G_{i,j}$ has maximum degree $\frac{\Delta}{k} + O(\sqrt{\Delta \log \frac{n}{k}})$ and has at most $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ edges.*

Algorithm 1: An MPC algorithm for edge coloring.

Parameter: k.;
Output: An edge coloring of a given graph $G = (V,E)$ with maximum degree Δ using $\Psi := \Delta + d\sqrt{k\Delta \log n}$ colors for some large enough constant d.
Independently and u.a.r. partition V into k subsets $V_1, \ldots, V_k$.;
For every $i \in [k]$, let G_i be the induced subgraph of G on V_i.;
For every $i, j \in [k]$ with $i \neq j$, let $G_{i,j}$ be the subgraph of G including an edge $e \in E$ iff one end-point of e is in V_i and the other is in V_j.;
Partition $[\Psi]$ into $k+1$ disjoint subsets $C_1, \ldots, C_k, C'$, which we call *color palettes*, in an arbitrarily way such that each palette has exactly $\frac{\Psi}{k+1}$ colors.;
for *each graph G_i in parallel* **do**
    Color G_i sequentially in a single machine with palette C'.;
end
// In what follows, we implicitly construct a k edge coloring of a complete k-vertex graph K_k and assign palette C_α to subgraph $G_{i,j}$ where α is the color of edge (i,j) in K_k.
for *each graph $G_{i,j}$ in parallel* **do**
    Color $G_{i,j}$ sequentially in a machine with palette C_α where $\alpha = ((i+j) \bmod k) + 1$.;
end

Proof. Let us start with bounding the degree of an arbitrary vertex $v \in V_i$ in subgraph G_i. The degree of vertex v in G_i is precisely the number of its neighbors that are assigned to partition V_i. Since there are k partitions, the expected degree of v in G_i is $\deg_G(v)/k \leq \Delta/k$. Furthermore, since the assignment of vertices to the partitions is done independently and uniformly at random, by a simple application of Chernoff bound, v's degree in G_i should be highly concentrated around its mean. Namely, with probability at least $1 - n^{-2}$, it holds that $\deg_{G_i}(v) \leq \frac{\Delta}{k} + O(\sqrt{\Delta \log n/k})$. Now, a union bound over the n vertices in the graph, proves that the degree of all vertices in their partitions should be at most $\frac{\Delta}{k} + O(\sqrt{\Delta \log n/k})$ with probability $1 - 1/n$.

Bounding vertex degrees in subgraphs of type $G_{i,j}$ also follows from essentially the same argument. The only difference is that we have to union bound over $n \cdot k$ choices, as we would like to bound the degree of any vertex v with say $v \in V_i$ in k subgraphs $G_{i,1}, \ldots, G_{i,k}$.

Nonetheless, since $k \leq n$, there are still $\text{poly}(n)$ many choices to union bound over. Thus, by changing the constants in the lower terms of the concentration bound, we can achieve the same high probability result.

Finally, we focus on the number of edges in each of the subgraphs. Each partition V_i has n/k vertices in expectation since the n vertices are partitioned into k groups independently and uniformly at random. A simple application of Chernoff and union bounds, implies that the number of vertices in each partition V_i is at most $O(\frac{n}{k})$ w.h.p., so long as $n/k \gg \log n$, which is the case. Since the number of edges in each partition is less than the number of vertices times max degree, combined with the aforementioned bounds on the max degree, we can bound the number of edges in G_i and $G_{i,j}$ for any i and j by

$$O\left(\frac{n}{k}\right) \cdot O\left(\frac{\Delta}{k} + \sqrt{\frac{\Delta}{k}\log n}\right) = O\left(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\frac{\Delta}{k}\log n}\right),$$

which is the claimed bound. □

Next, observe that we use palettes $C_1, \ldots, C_{k+1}, C'$, each of size $\frac{\Psi}{k+1}$ to color the subgraphs. We need to argue that the maximum degree in each subgraph is at most $\frac{\Psi}{k+1} - 1$ to be able to argue that using Vizing's theorem in one machine, we can color any of the subgraphs with the assigned palettes. This can indeed be easily guaranteed if the constant d is large enough:

Observation A.2.1. *If constant d in Algorithm 1 is large enough, then maximum degree of every graph is at most $\frac{\Psi}{k+1} - 1$, w.h.p.*

Proof. We have $\Psi = \Delta + d\sqrt{k\Delta\log n}$ in Algorithm 1, therefore:

$$\frac{\Psi}{k+1} = \frac{\Delta}{k+1} + \frac{d\sqrt{k\Delta\log n}}{k+1} = \frac{\Delta}{k} + \Theta(\sqrt{\Delta\log n/k}),$$

where the hidden constants in the second term of the last equation can be made arbitrarily

large depending on the choice of constant d. On the other hand, recall from Claim A.2.4 that the maximum degree in any of the subgraphs is also at most $\frac{\Delta}{k} + O(\sqrt{\Delta \log n/k})$. Thus, the palette sizes are sufficient to color the subgraphs if d is a large enough constant. □

We are now ready to prove the algorithm's correctness.

Lemma A.2.5. *Algorithm 1 returns a proper edge coloring of G using $\Delta + O(\sqrt{k\Delta \log n})$ colors.*

Proof. The algorithm clearly uses $\Psi = \Delta + O(\sqrt{k\Delta \log n})$ colors, it remains to argue that the returned edge coloring is proper. Each subgraph (of type G_i or $G_{i,j}$) is sent to a single machine and edge-colored there using the palette that it is assigned to. Since by Observation A.2.1, each palette has at least $\Delta' + 1$ colors for Δ' being the max degree in the subgraphs, there will be no conflicts in the colors associated to the edges within a partition. We only need to argue that two edges e and f sharing a vertex v that belong to two different subgraphs are not assigned the same color. Note that all subgraphs of type G_i are vertex disjoint and all receive the special color palette C', thus there cannot be any conflict there. To complete the proof, it suffices to prove that any two subgraphs $G_{i,j}$ and $G_{i',j'}$ that share a vertex receive different palettes. Note that in this case, either $i = i'$ or $j = j'$ by the partitioning. Assume w.l.o.g. that $i = i'$ and thus $j \neq j'$. Based on Algorithm 1 for $G_{i,j}$ and $G_{i',j'}$ to be assigned the same color palette, it should hold that

$$((i+j) \bmod k) + 1 = ((i'+j') \bmod k) + 1.$$

Since $i = i'$, this would imply that $(j \bmod k) = (j' \bmod k)$, though this would not be possible given that both j and j' are in $[k]$ and that $j \neq j'$. Therefore, any two subgraphs that share a vertex receive different palettes and thus there cannot be any conflicts, completing the proof. □

Next, we turn to prove the space bounds.

Lemma A.2.6 (Implementation and Space Complexity)**.** *Algorithm 1 can be implemented with total space $O(m)$ and space per machine of $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ w.h.p.*

Proof. We start with an implementation that uses the specified space per machine but can be wasteful in terms of the total space, then describe how we can overcome this problem and also achieve an optimal total space of $O(m)$.

We can use $k + \binom{k}{2}$ machines, each with a space of size $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ to assign colors to the edges in parallel. The first $m_1, \ldots, m_k$ machines will be used for edge coloring on $G_1, G_2, \ldots, G_k$ respectively. The other $m_{k+1}, \ldots, m_{k+\binom{k}{2}}$ machines will be used for edge coloring on the $G_{i,j}$ graphs. Lemma A.2.4 already guarantees that each subgraph has size $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ w.h.p., and thus fits the memory of a single machine.

In the implementation discussed above, since the machines use $\widetilde{O}(n\Delta/k^2)$ space and there are $O(k^2)$ machines, the total memory can be $\widetilde{O}(n\Delta)$ which may be much larger than $O(m)$. This is because we allocate $O(n\Delta/k^2)$ space to each machine regardless of how much data it actually received. Though, observe that each edge of the graph belongs to exactly one of the subgraphs, i.e., the machines together only handle a total of $O(m)$ data. So we must consolidate into fewer machines. We do this by putting multiple subgraphs in each machine.

We start by recalling a sorting primitive in the MPC model which was proved in [76]. Basically, if there are N items to be sorted and the space per machine is $N^{\Omega(1)}$, then the algorithm of [76] sorts these items into the machines within $O(1)$ rounds. To use this primitive, we first label each edge $e = (u, v)$ of the graph by its subgraph name (e.g. G_i or $G_{i,j}$) which can be determined solely based on the end-points of the edge. After that, we sort the edges based on these labels. This way, all the edges inside each subgraph can be sent to the same machine within $O(1)$ rounds while also ensuring that the total required

space remains $O(m)$. □

The algorithm for Theorem A.2.1 was formalized as Algorithm 1. We showed in Lemma A.2.5 that the algorithm correctly finds an edge coloring of the graph with the claimed number of colors. We also showed in Lemma A.2.6 that the algorithm can be implemented with $O(m)$ total space and $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ space per machine. This completes the proof of Theorem A.2.1.

A.3 Streaming Algorithms

We start in Section A.3.1 by describing our streaming algorithm and its analysis when the arrival order is random. Then in Section A.3.2, we give another algorithm for adversarial order streams.

A.3.1 Random Edge Arrival Setting

In this section, we give a streaming algorithm for $O(\Delta)$ edge coloring using $\widetilde{O}(n)$ space where the edges come in a random stream. That is, a permutation over the edges is chosen uniformly at random and then the edges arrive according to this permutation.

We first note that if $\Delta = O(\log n)$ then the problem is trivial as we can store the whole graph and then report a $\Delta + 1$ edge coloring (even without knowledge of Δ). As such, we assume $\Delta = \omega(\log n)$.

The algorithm — formalized as Algorithm 2 — maintains a counter c_v for each vertex v. At any point during the algorithm, this counter c_v basically denotes the highest color number used for the edges incident to v so far, plus 1. Therefore, upon arrival of an edge (u,v), it is safe to color this edge with $\max(c_u, c_v)$ as all edges incident to u and v have a color that is strictly smaller than this. Then, we increase the counters of both v and u to $\max(c_u, c_v) + 1$. It is not hard to see that the solution is always a valid coloring, in

the remainder of this section, we mainly focus on the number of colors required by this algorithm and show that w.h.p., it is only $O(\Delta)$ for random arrivals.

Algorithm 2: Edge coloring for random streams.

Result: A feasible coloring $\mathscr{C} : E \to [\Psi]$ for a given graph $G = (V,E)$ with maximum degree Δ in a random stream

```
c_v ← 0  ∀v ∈ V;
while (u, v) is read from stream do
    C(u, v) ← max(c_u, c_v);
    c_u, c_v ← C(u, v) + 1;
end
```

We start by noting that this algorithm can actually be extremely bad if the order is adversarial. To see this, consider a path of size n. In an adversarial stream where the edges arrive in the order of the path, Algorithm 2 uses as many as $n-1$ colors while the maximum degree is only 2! It is easy to see why this example is very unlikely to occur in random order streams: For a fixed path, it is very unlikely that the edges are randomly ordered in this very specific way.

To make this intuition rigorous for general graphs, we first prove the following crucial lemma which gives us the correct parameter to bound.

Lemma A.3.1. *Let Ψ be the size of the longest monotone (in the order of arrival) path in the line-graph of G. Then Algorithm 2 uses exactly Ψ colors.*

Proof. Take a monotone path $v_1, v_2, \ldots, v_\Psi$ in the line-graph of G and let $e_1, e_2, \ldots, e_\Psi$ be the edges of the original graph that correspond to these vertices respectively, i.e., e_1 arrives before e_2 which arrives before e_3 and so on. Since for any i, v_i and v_{i+1} are neighbors in the line-graph, then e_i and e_{i+1} should share an end-point v. This means that at the time of arrival of e_{i+1}, we have $c_v \geq \mathscr{C}(e_i) + 1$ which in turn, implies $\mathscr{C}(e_\Psi) > \mathscr{C}(e_{\Psi-1}) > \ldots > \mathscr{C}(e_1)$. Therefore, $\mathscr{C}(e_\Psi) \geq \Psi$.

On the other hand, suppose that there is an edge $e_1 = (u, v)$ for which $\mathscr{C}(e_1) = \Psi$ in Algorithm 2. This means that at least one of c_u or c_v equals Ψ when e_1 arrives, say c_u w.l.o.g. Let e_2 be the last edge incident to u that has arrived before e_1. It should hold that $\mathscr{C}(e_2) = \Psi - 1$. Using the same argument, for each $1 < i \leq \Psi$, we can find a neighboring edge e_i such that $\mathscr{C}(e_i) = \mathscr{C}(e_{i-1}) - 1$. This way, we end up with a sequence $e_1, \ldots, e_\Psi$ of edges, the path corresponding to this sequence in the line graph will be a monotone path of length Ψ, completing the proof. □

Theorem A.3.2. *There is a streaming edge coloring algorithm that for any graph $G = (V, E)$ uses at most $(2e + \varepsilon)\Delta \approx 5.44\Delta$ colors w.h.p. for any constant $\varepsilon > 0$ given that the edges in E arrive in a random order.*

Proof. We first prove that Algorithm 2 gives us a feasible coloring of graph G. Consider two edges $e_1 = (u, v)$ and $e_2 = (u, v')$ incident to vertex u such that e_1 appears earlier than e_2 in the stream. For any edge e we represent by $\mathscr{C}(e)$ the color assigned to that by the algorithm. After the algorithm colors e_1 with $\mathscr{C}(e_1)$, it sets c_u to $\mathscr{C}(e_1) + 1$. Thus, c_u is at least $\mathscr{C}(e_1) + 1$ when e_2 arrives and $\mathscr{C}(e_2) \geq \mathscr{C}(e_1) + 1$ consequently. Therefore, $\mathscr{C}(e_2) > \mathscr{C}(e_1)$ for any pair of edges incident to a common vertex, and $\mathscr{C}$ is a feasible coloring.

Next, for some constant α that we fix later, we show that the probability that an edge is assigned a color number at least $\alpha\Delta$ is at most n^{-c} for some constant $c \geq 2$, implying via a union bound over all the edges that indeed w.h.p., $\Psi \leq \alpha\Delta$.

We showed in Lemma A.3.1 that if the number of colors Ψ used is $\alpha\Delta$, then there should exist a monotone path in the line-graph with size at least $\alpha\Delta$. Let $e_0, e_2, \ldots, e_{\alpha\Delta}$ be the corresponding edges to this path. Thus, it suffices to bound the probability of this event. Let Π denote the set of all such paths in the line graph. For a specific path $\pi \in \Pi$, the probability that it is monotone is $1/(\alpha\Delta)!$. Call this event X_π. On the other hand, we can

upper bound the number of such paths by $(2\Delta)^{\alpha\Delta}$, i.e., $|\Pi| \leq (2\Delta)^{\alpha\Delta}$. This follows from the fact that each path should start from the corresponding vertex to e_0 in the line-graph, and that maximum degree in the line graph is $2\Delta - 2$ (which is the upper bound on the number of neighboring edges to each edge). Thus:

$$\mathbf{Pr}[\mathscr{C}(e_0) \geq \alpha\Delta] = \mathbf{Pr}[\bigvee_{\pi\in\Pi} X_\pi] \leq \sum_{\pi\in\Pi} \mathbf{Pr}[X_\pi = 1] \leq \frac{(2\Delta)^{\alpha\Delta}}{(\alpha\Delta)!},$$

where the last inequality is obtained by replacing $\mathbf{Pr}[X_\pi = 1]$ and $|\Pi|$ by the aforementioned bounds. Taking the logarithm of each side of the inequality, we get

$$\begin{aligned}
\ln(\mathbf{Pr}[\mathscr{C}(e_0) \geq \alpha\Delta]) &\leq \alpha\Delta\ln(2\Delta) - \ln((\alpha\Delta)!) \\
&\leq \alpha\Delta\ln(2\Delta) - ((\alpha\Delta + 1/2)\ln(\alpha\Delta) - \alpha\Delta) && \text{(A.1)} \\
&= \alpha\Delta\ln(2e/\alpha) - 1/2\ln(\alpha\Delta) && \text{(A.2)} \\
&\leq \alpha\Delta\ln(2e/\alpha). && \text{(A.3)}
\end{aligned}$$

To obtain (A.1), we use Stirling's approximation of factorials to lower-bound $\ln((\alpha\Delta)!)$. Finally, we rearranged terms to imply (A.2). By plugging in $\alpha = 2e(1+\varepsilon)$, we get

$$\begin{aligned}
\ln(\mathbf{Pr}[\mathscr{C}(e_0) \geq 2e(1+\varepsilon)\Delta]) &\leq 2e(1+\varepsilon)\Delta\ln\left(\frac{1}{1+\varepsilon}\right) \\
&= -2e(1+\varepsilon)\ln(1+\varepsilon)\Delta \\
&\leq -2e(1+\varepsilon)\ln(1+\varepsilon)\frac{c}{2e(1+\varepsilon)\ln(1+\varepsilon)}\ln(n) && \text{(A.4)} \\
&= -c\ln(n)
\end{aligned}$$

Since $\Delta = \omega(\log(n))$, we have $\Delta > c'\ln(n)$ for any constant c'. Inequality (A.4) follows from setting $c' = c/(2e(1+\varepsilon)\ln(1+\varepsilon))$ in $\Delta > c'\ln(n)$, where c is the constant for which

Clique Size	100	200	300	400	500	600	700	800	900	1000
Colors Used	3.363Δ	3.563Δ	3.665Δ	3.717Δ	3.756Δ	3.787Δ	3.815Δ	3.838Δ	3.849Δ	3.863Δ

Table A.1: The number of colors used by Algorithm 2 on cliques averaged over 100 trials.

we want to show the probability is upper-bounded by n^{-c}. Hence,

$$\mathbf{Pr}[\mathscr{C}(e_0) \geq 2e(1+\varepsilon)\Delta] \leq n^{-c}.$$

Thus, Algorithm 2 returns a feasible coloring of the input graph G using at most $2e(1+\varepsilon)\Delta$ colors, for any constant $\varepsilon > 0$ w.h.p. if the edges arrive in a random order. □

To further evaluate the performance of Algorithm 2, we implemented and ran it for cliques of different size. The result of this experiment is provided in Table A.1. The numbers are obtained by running the experiment 100 times and taking the average number of colors used. As it can be observed from Table A.1, for cliques of size 100 to 1000, the number of colors used by the algorithm is in range $[3.3\Delta, 3.9\Delta]$ and it slightly increases by the size of the graph. Our analysis, however, shows that it should never exceed 5.44Δ.

A.3.2 Adversarial Edge Arrival Setting

In this section, we turn to arbitrary (i.e., adversarial) arrivals of the edges. We assume that the adversary is *oblivious*, i.e., the order of the edges is determined before the algorithm starts to operate so that the adversary cannot abuse the random bits used by the algorithm. Having this assumption, we give a randomized algorithm that w.h.p., outputs a valid edge coloring of the graph using $O(\Delta^2)$ colors while using $\tilde{O}(n)$ space. The algorithm is formalized as Algorithm 3. We note that this algorithm, as stated, requires knowledge of Δ. However we later show that we can get rid of this assumption. Overall, we get the following result:

Algorithm 3: Edge coloring in the adversarial order

Result: A feasible coloring for a given graph $G = (V, E)$ with maximum degree Δ

for *any vertex* $v \in V$ **do**
  $r_v \leftarrow$ a sequence of $\log(n)$ independent random bits.
  for *any* $i \in [\log n]$ **do**
    $c_{v,i} \leftarrow 0$
  end
end
for *any edge* $e = (u, v)$ *in the stream* **do**
  Let i be the smallest index for which $r_{v,i} \neq r_{u,i}$.
  if $\Delta 2^{-i} > \log n$ **then**
    if $r_{u,i} = 1$ **then**
      Assign color $(c_{u,i}, c_{v,i}, i)$ to e.
    else
      Assign color $(c_{v,i}, c_{u,i}, i)$ to e.
    end
    Increase both $c_{v,i}$ and $c_{u,i}$ by one.
  else
    Store edge e.
  end
end
Color the stored edges using a new set of colors.

Theorem A.3.3. *Given a graph G with maximum degree* Δ*, there exists a one pass streaming algorithm, that outputs a valid edge coloring of the G using* $O(\Delta^2)$ *colors w.h.p., using* $\widetilde{O}(n)$ *memory.*

Consider two vertices v and u and their string of random bits r_v and r_u defined in the algorihtm. Let $d_{u,v}$ be the smallest index i where $r_{u,i} \neq r_{v,i}$. Upon arrival of an edge $e = (u, v)$, we first find $i := d_{u,v}$. If $\Delta 2^{-i} > \log n$, we color the edge immediately. Otherwise, we store it. We will show that all the stored edges fit in the memory thus after reading all the stream we can color them with a palette of at most $\Delta + 1$ new colors. In the algorithm, for any vertex v and any $i \in [\log n]$, we define a counter $c_{u,i}$. If $\Delta 2^{-i} > \log n$ for any edge e, then we immediately assign e a color which is represented by a tuple $(c_{u,i}, c_{v,i}, i)$. Then, we increase counters $c_{u,i}$ and $c_{v,i}$. Note that we say two colors are the same if all three

elements of them are equal. We first show that this gives us a valid coloring, which means it does not assign the same color to two edges adjacent to the same vertex. We use proof by contradiction. Assume that our algorithm assigns the same color to edges $e_1 = (u, v_1)$ and $e_2 = (u, v_2)$ adjacent to vertex u. None of them can be from the stored edges since we color them using a new palette. This means that $d_{u,v_1} = d_{u,v_2}$. Let us denote it by i. Without loss of generality, we assume that $r_{u,i} = 1$ and that in the input stream e_1 arrives before e_2. Note that the first element of the colors (which are tuples) assigned to these edges is the value of counter $c_{u,i}$ when they arrive. However, the algorithm increases $c_{u,i}$ by one after arrival of e_1 thus the colors assigned to e_1 and e_2 cannot be the same.

Now, it suffices to show that the total number of colors used by the algorithm is $O(\Delta^2)$. Given a vertex v, and a number $l \in [\log n]$ let us compute an upper-bound for counter $c_{v,i}$. Let N_v be the set of neighbors of this vertex and let $N_{v,i}$ be the set of neighbors like u where $d_{v,u} = i$. We know that $c_{v,i} = |N_{v,i}|$, thus given any vertex v and $i \in [\log(n)]$, we need to find a bound for $|N_{v,i}|$. Given any edge $e = (v, u)$ the probability of e being in set $N_{v,i}$ is 2^{-i} which means $\mathbb{E}[|N_{v,i}|] = \deg(v)2^{-i}$ where $\deg(v)$ is the degree of vertex v in the input graph.

Using a simple application of the Chernoff bound, for any vertex v, we get:

$$\mathbf{Pr}\Big[|N_{v,i}| \geq \deg(v)2^{-i} + O\big(\sqrt{\deg(v)2^{-i}\log n}\big)\Big] \leq \frac{1}{n^c}.$$

Setting c to be a large enough constant, one can use union bound and show that w.h.p., for any vertex v and $i \in [\log n]$ where $\deg(v)2^{-i} \geq \log n$, we have $|N_{v,i}| \leq O(\deg(v)2^{-i})$.

Having this, we conclude that for any $i \in [\log n]$, where $\Delta 2^{-i} > \log n$, the number of colors used by the algorithm whose third element is i is at most $O(\Delta^2 2^{-2i})$ since the first and the second element of the color can get at most $O(\Delta 2^{-i})$ different values. Therefore, the total number of colors used for any such i is at most $O\big(\sum_{i\in[\log n]} \Delta^2 2^{-2i}\big) = O(\Delta^2)$. We

will also show that the stored edges fit in the memory and thus we can color them using $O(\Delta)$ new colors. As a result the total number of colors used is $O(\Delta^2)$.

To give an upper-bound for the number of stored edges we first show that the expected number of stored edges for each vertex is $O(\log n)$. Let $j := \log(\frac{\Delta}{\log n})$. Recall that we store an edge (u,v) when $\Delta 2^{-d_{u,v}} < \log n$. Thus the expected number of stored edges adjacent to a single vertex v is at most

$$\sum_{j \le i \le \log n} d_v 2^{-i} \le \sum_{j \le i \le \log n} \Delta 2^{-i} \le \sum_{j \le i \le \log n} \log(n) 2^{-i+j} = O(\log n).$$

To get the last equation we use the fact that $\Delta 2^{-j} \le \log n$. By a similar argument that we used above (using Chernoff and Union bounds), with a high probability the total number of stored edges is $O(n \log n)$ which can be stored in the memory. Therefore the proof of this theorem is completed.

Knwoledge of Δ. As written, our algorithm depends on the knowledge of Δ because we must check $\Delta 2^{-i} > \log n$. We can get rid of this condition by keeping track of the degree $\deg_v^H$ of a vertex in the subgraph H we have seen so far, and then computing the max degree $\deg_{max}^H$. This only requires an additional $O(n)$ space. Thereafter, instead of checking if $\Delta 2^{-i} > \log n$, we check if $\deg_{max}^H 2^{-i} > \log n$. Whenever $\deg_{max}^H$ increases, we iterate over all stored edges and recompute whether or not $\deg_{max}^H 2^{-i} > \log n$. If so, we color the edge and remove it from the buffer, else we keep it. It is easy to see that this will not exceed the space bounds because at any timestep, we can assume the input graph was H in the first place. Then its max degree is $\Delta_H = \deg_{max}^H$, and we can apply the same argument for the space bounds as before, but using Δ_H instead of Δ. All other parts of the proof still hold. Therefore our algorithm does not require knowledge of Δ.

Finally, we remark that if one allows more space, then one can modify Algorithm 3 to

use fewer number of colors. Though we focused only on the $\widetilde{O}(n)$ memory regime.

A.4 Open Problems

We believe the most notable future direction is to improve the number of colors used in our streaming algorithms. Specifically, our streaming algorithm for adversarial arrivals requires $O(\Delta^2)$ colors. A major open question is whether this can be improved to $O(\Delta)$ while also keeping the memory near-linear in n. Also for random arrival streams, we showed that Algorithm 2 achieves a 5.44Δ coloring and showed, experimentally, that it uses at least 3.86Δ colors. A particularly interesting open question is whether there is an algorithm that uses arbitrarily close to 2Δ colors using $\widetilde{O}(n)$ space in random arrival streams.

Appendix B: Minimum Cut

B.1 Introduction

Massively Parallel Computation (MPC) – introduced by Karloff et al. [95] in 2010 – is an abstract model that captures the capabilities of the modern parallel/distributed frameworks widely used in practice such as MapReduce [51], Hadoop [135], Flume [40], and Spark [137]. MPC has been at the forefront of the research on parallel algorithms in the past decade, and it is now known as the de facto standard computation model for the analysis of parallel algorithms.

In this paper, we focus on sublogarithmic-round algorithms for the Min Cut problem in the *Adaptive Massively Parallel Computation* (AMPC) model, which is a recent extension of MPC. In both MPC and AMPC, the input data is far larger than the memory of a single machine, and thus an input of size N is initially distributed across a collection of M machines. In the MPC model, the algorithm executes in several synchronous rounds, in which each machine executes local computations isolated from other machines, and the machines can only communicate at the end of a round. The total size of incoming/outgoing messages for each machine is limited by local memory constraints. We are interested in *fully-scalable* algorithms in which every machine is allocated a local memory of size $O(\mathsf{N}^{\varepsilon})$ for any constant $0 < \varepsilon < 1$. Moreover, we can often improve the round complexity[1]

[1]The number of rounds is a main complexity of interest since in practice the bottleneck is often the communication phase.

of the massively parallel algorithms by allowing a super-linear total memory $O(\mathsf{N}^{1+\varepsilon})$, for example, the filtering technique of Lattenzi et al. [102] in MPC or the maximal matching algorithm of Behnezhad et al. [30] in AMPC. So we are primarily interested in algorithms with $\widetilde{O}(\mathsf{N})$ total memory, and therefore we assume there are $\mathsf{M} = \widetilde{O}(\mathsf{N}^{1-\varepsilon})$ machines.

Recent developments in the hardware infrastructure and new technologies such as RDMA [54], eRPC, and Farm [55] allow for high-throughput, low-latency communication among machines in data centers, such that remote volatile memory accesses are becoming faster than accessing local persistent storage. The concept of a shared remote memory is in particular useful when machines need to query data adaptively – i.e., deciding what to query next based on the previously queried data – which requires a communication round per query in the MPC model. Behnezhad et al. [29] incorporates this RDMA-like paradigm of remote memory access into the MPC model and introduces AMPC. In the new model, the machines can *adaptively* query from a *distributed hash table*, or a shared read-only memory, during each round. Machines are only allowed to write to shared memory at the end of each round. There is also empirical evidence that AMPC algorithms for several problems – including maximal independent set, maximal matching, and connectivity – obtain significant speedups in running time compared to state-of-the-art MPC algorithms [30]. This fact, which stems from the meaningful drop in the number of communication rounds, verifies the practical power of the AMPC model.

In this paper, we provide the first AMPC-specific algorithms for the Min Cut problem. The Min Cut of a given graph $G = (V,E)$ is the minimum number of outgoing edges, $\delta(S)$, among every subset of vertices $S \subseteq V$. The celebrated result of Karger and Stein [94] solves Min Cut by recursively contracting edges in random order. Specifically, it runs two instances of the contraction process with different seeds in parallel. Each instance is run in parallel until the graph size is reduced by a factor of $\frac{1}{\sqrt{2}}$, at which point each instance recurses (thereby creating a parallel split again). They return the minimum of the two

returned cuts. The algorithm itself is mainly inspired by another result of Karger [93] for finding the Min Cut using graph contractions. We also extend our approach to the Min k-Cut problem, in which we are given a graph $G = (V,E)$ and an integer k and we want to find a decomposition of V into k subsets $V_1, V_2, \ldots, V_k$ so that $\sum_{i=1}^{k} \delta(V_i)$ is minimized. We utilize the greedy algorithm of Saran and Vazirani [129] which gives an $O(2-\frac{2}{k})$-approximation of the Min k-Cut. Gomuri and Hu give an alternative algorithm with the same approximation guarantee with additional features [74].

We study the Min Cut and Min k-Cut problems in the AMPC model. We give $O(\log\log n)$-round AMPC algorithms for a $(2+\varepsilon)$-approximation of Min Cut and a $(4+\varepsilon)$-approximation of Min k-Cut.

B.1.1 Adaptive Massively Parallel Computation (AMPC)

Massively Parallel Computation (MPC) and Adaptive Massively Parallel Computation (AMPC) both sprung out of an interest in formalizing a theoretical model for the famous MapReduce programming framework. The most common problems in MPC and AMPC are on graph inputs, and since our paper only considers graph problems, we define these two models in terms of problems on graphs. Say a graph has n vertices and m edges.

In standard MPC [6, 76, 95, 102], we are given a collection of M machines and are allowed to compute the solution to a problem in parallel. As we have already discussed, MPC computation occurs in synchronous rounds, each consisting of local polynomial-time computation and ending with machine-machine communication where all messages sent to and from a machine must fit within its local memory. Fully-scalable algorithms, the strongest memory regime in MPC, require the local memory to be constrained by $O(n^{\varepsilon})$ for any given $0 < \varepsilon < 1$. Additionally, we are primarily interested in algorithms that require at most $O(\log n)$ rounds. However, often sublogarithmic – i.e., $O(\sqrt{\log n})$ or $O(\log\log n)$ –

round complexity is much more desirable. In most cases, the total space must be at most $\widetilde{O}(n+m)$, though sometimes we allow slightly superlinear total space.

AMPC extends MPC to add functionality while remaining implementable on modern hardware. Formally, in the AMPC model, we are given a set of distributed hash tables $\mathscr{H}_0, \ldots, \mathscr{H}_k$ for each of the k rounds of computation. These hash tables are each limited in size by the total space of the model (i.e., $\widetilde{O}(n+m)$). As in MPC, we are given a number of machines and computation proceeds in rounds. In each round, local computations occur and then messages are sent between machines. The distinction in AMPC is that during the local computations, machines are allowed simultaneous read access to the hash table for that round (i.e., $\mathscr{H}_{i-1}$ for round i) and during the messaging phase of the round, they are allowed to write data to the next hash table, $\mathscr{H}_i$. The power of the AMPC model over the MPC model is that, at the beginning of a round, the machines do not need to choose all the data they will access during the round. Instead, they can dynamically access the data stored in the hash table over the course of the local computation, thus potentially selecting data based on its own local computation.

It is not too hard to see that AMPC is a strictly stronger model than MPC. In fact, it was formally shown that all MPC algorithms can be implemented in AMPC with the same round and space complexities [29].

B.1.2 Our Contributions and Methods

This work is the first to study the Adaptive Massively Parallel Computation (AMPC) model for Min Cut problems on graphs. We mainly focus on the standard single Min Cut problem, although we also propose an approximation algorithm for the Min k-Cut problem. Our main result for the Min Cut problem is a $2+\varepsilon$ approximate algorithm that uses sublogarithmic $O(\log\log n)$ rounds.

Theorem B.1.1. *There is an $O(\log\log n)$-round AMPC algorithm that uses $\widetilde{O}(n+m)$ total memory and $\widetilde{O}(n^{\varepsilon})$ memory per machine which finds a $(2+\varepsilon)$-approximation of Min Cut with high probability.*

Note that this is a vast improvement over the current state-of-the art algorithms in MPC by Ghaffari and Nowicki [68], which achieves the same $2+\varepsilon$ approximation in $O(\log n \log\log n)$ rounds. Both our algorithm and that of Ghaffari and Nowicki use Karger's methods as a general structure for finding the Min Cut. Using this method, the goal is to recursively execute random graph contractions. From the results of Karger, the contraction process either finds a singleton cut that is a $2+\varepsilon$ approximation or preserves a specific Min Cut with probability dependent on the depth of recursion. To leverage this result, at each step of the recursion process, we find the best singleton cut on the existing graph. Once the graph is small enough, the problem can be solved efficiently. Out of all the singleton solutions found during this process and the final Min Cut on the small graph, we simply select the best cut. This is a $2+\varepsilon$ approximate Min Cut with high probability.

To implement this approach in a distributed model, both methods assign random weights to the edges of the input graph and find a minimum spanning tree (MST). Greedily, selecting edges in order of decreasing weight, we contract the graph along the current edge. This process is equivalent to the same greedy random contraction process on the original graph. This step, already, currently requires at least $\Omega(\log n)$ rounds in MPC, but the flexibility of the AMPC model allows us to achieve this step in a constant number of rounds.

It remains to show how can one find the best singleton cuts at each level of recursion. In order to do this, we employ a *low-depth tree decomposition* on the minimum spanning tree until it becomes a set of separated vertices. On top of this recursive divide-and-conquer process, we design a process to compute and remember the best singleton cut.

The high level idea of recursively partitioning the tree and applying a process on top of

that to find the best singleton cut is the same in both our paper and Ghaffari and Nowicki's paper [68]. However, the processes used to do this in MPC do not yield simple improvements in AMPC. Rather, we must use entirely novel techniques that leverage adaptivity to get truly sublogarithmic results. In fact, this must be done in constant rounds to achieve our results, whereas Ghaffari and Nowicki do this in $O(\log n)$ rounds. In order to create a tree decomposition, we consider maximal paths of heavy edges (i.e., edges that go from a parent to its child with the largest subtree). These paths are replaced by binary trees whose leaves are the path and the root connects to the path's parent. Consider labeling the resulting vertices in the graph with their depth. For each internal node in one of these binary trees, which was *not* a vertex in the original tree, we select a specific descendant leaf in the binary tree expansion of the path to send its depth to. The final value a vertex receives is then what we call the "label", which measures at what level of recursion the tree splits at that vertex. An entire labeling of the tree encodes an entire tree decomposition. This is done in constant AMPC rounds.

To compute the singleton cuts at each level, we assign to each singleton cut formed during the contraction process a vertex that has the lowest label. We show that such vertices are well-defined, i.e. there is only one vertex with the lowest label within vertices on the same side of a singleton cut. Because removing vertices of labels lower than i partitions the tree into disjoint subtrees such that each subtree contains at most one vertex with label i, we are able to calculate minimal singleton cuts corresponding to these vertices with label i in parallel in a constant number of AMPC rounds. Since, we constructed the low-depth decomposition such that the range of labels has size $O(\log^2 n)$, thus, by increasing the total memory, we can perform these computations for all different lables in a constant number of AMPC rounds. For more details, we defer to Section B.4.

We then show how this work can be leveraged to achieve efficient results for approximate Min k-Cut, generalizing the results from Saran and Vazirani [129]. At a high level, we

start by computing a Min Cut. Then we add the edges of the cut to a set D. In all following $k-1$ iterations, we calculate the Min Cut on the graph without edges in D, and add the new cut edges to D for the next iteration. The set of the first k cuts we compute is our k-cut.

Compared to Saran's and Vazirani's technique, our method uses an *approximate* Min Cut rather than an exact Min Cut on each splitting step. This requires adapted analysis of this general approach. We employ the structure of Gomory-Hu trees (see [74]) for this purpose and show the following result:

Theorem B.1.2. *Algorithm APX-SPLIT is an* $(4+\varepsilon)$ *approximation of the Min k-Cut. Furthermore, it can be implemented in the AMPC model with* $\mathrm{O}(n^{\varepsilon})$ *memory per machine in* $\mathrm{O}(k \log\log n)$ *rounds and* $\mathrm{O}(m)$ *total memory.*

Therefore, for small values of k, we can achieve efficent algorithms for $4+\varepsilon$ approximate Min k-Cut in AMPC. Note that there are no existing results in the MPC model, however our methods applied to the work of Ghaffari and Nowicki [68] yield:

Corollary B.1.3. *There is an algorithm that achieves a* $(4+\varepsilon)$ *approximation of the Min k-Cut with high probability that can be implemented in the MPC model with* $\mathrm{O}(n^{\varepsilon})$ *memory per machine in* $\mathrm{O}(k \log n \log\log n)$ *rounds and* $\mathrm{O}(m)$ *total memory.*

Note there is still a logarithmic-in-n improvement in the round complexity in AMPC over MPC no matter the value of k. Due to space constraints both these result are presented in Section B.5 in the appendix.

B.2 Minimum Cut in AMPC

Karger and Stein [94] proposed a foundational edge contraction strategy for solving Min Cut:

- Create two copies of G, and independently on each, contract edges in a random order until there are at most $\frac{n}{\sqrt{2}}$ vertices.
- Recursively solve the problem on each contracted copy until they have constant size.
- Return the minimum of the cuts found on both copies.

Lemma B.2.1 ([94])**.** *The contraction process executed to the point where there are only $\frac{n}{t}$ vertices left preserves any fixed minimum cut with probability $\Omega\left(\frac{1}{t^2}\right)$.*

According to Lemma B.2.1, naively contracting random edges until there are only two vertices remaining preserves at least one minimum cut with probability $\Omega\left(\frac{1}{n^2}\right)$. Thus, we need to repeat the naive contraction process at least $O\left(n^2 \log n\right)$ times so that we have a high *probability of success*, i.e., preserving a minimum cut. However, Karger and Stein [94] show that their recursive strategy succeeds with probability $\Omega\left(\frac{1}{\log n}\right)$. In turn, running $O\left(\log^2 n\right)$ instances of the recursive strategy is enough to find a minimum cut with high probability.

Roughly speaking, the choice of $t = \sqrt{2}$ as the inverse of the branching factor assures that a minimum cut is preserved with probability $\frac{1}{t^2} = \frac{1}{2}$ throughout the contractions in each copy. Hence, the probability of success, say $P(n)$, for n vertices is bounded by:

$$P(n) \geq 1 - \left(1 - \frac{1}{2} \cdot P\left(\frac{n}{\sqrt{2}}\right)\right)^2 \tag{B.1}$$

Note that the random contractions in two copies are assumed to be independent, and the probability of success for each copy is at least $\frac{1}{2} \cdot P\left(\frac{n}{\sqrt{2}}\right)$ since we recurse on the resulting contracted graph with $\frac{n}{\sqrt{2}}$ vertices. Inequality (B.1) implies that at the k-th level of recursion (counting from the bottom), the probability of success is $\Omega\left(\frac{1}{k}\right)$, and in particular $\Omega\left(\frac{1}{\log n}\right)$ at the root of recursion. [94].

Let us now give some high-level insight into Ghaffari's and Nowicki's approach. Ghaffari and Nowicki [68] observed that if we only desire a $(2+\varepsilon)$ approximate cut, we can use a better bound for the probability of preserving a minimum cut, or alternatively, the success probability.

Lemma B.2.2 ([68, 94])**.** *On an n-vertex graph G, let C be a minimum cut with weight* λ*. Fix an arbitrary* $\varepsilon \in (0,1)$*. The described random contraction process that contracts G down to* $\frac{n}{t}$ *vertices either at some step creates a singleton cut of size at most* $(2+\varepsilon)\lambda$ *or preserves C - i.e., it does not contract any of its edges - with probability at least* $\frac{1}{t^{1-\varepsilon/3}}$*.*

A *singleton cut* is a partitioning of graph vertices so that there is only one vertex on one side, i.e., $\delta(S)$ so that $|S| = 1$. Assuming that one is able to verify whether a singleton cut of a small size has been formed during the contraction process, they show that this greater probability of success can boost the recursive process. In short, consider the k-th level of recursion, where level 0 corresponds to the bottom level. Let $\frac{n}{t_k}$ be the size of a single recursive instance at level k, and denote by s_k the total number of instances on this level. For all k, they ensure $s_k = t_k^{1-\varepsilon/3}$.

Now, let $x_k^{1-\varepsilon/3}$ be the branching factor on level k. That is, the recursion produces $x_k^{1-\varepsilon/3}$ copies of the instance at level k, and on each of them independently contracts edges in a random order until the number of vertices is bigger than $\frac{n}{t_k} \cdot \frac{1}{x_k}$. If we have an algorithm that is able to *track* whether a small singleton cut appeared in each of these random processes, we either get a singleton cut that $(2+\varepsilon)$ approximates a minimum cut or a minimum cut is preserved with probability $x_k^{1-\varepsilon/3}$. Since we made $x_k^{1-\varepsilon/3}$ copies, by a similar argument as in Karger's approach, we get that, in the latter case, the probability of preserving a minimum cut is $\Omega\left(\frac{1}{k}\right)$.

Finally, observe that on the k-th level of recursion, the most costly operation is copying a k-th level instance $x_k^{1-\varepsilon/3}$ times in order to contract edges in each of these instances.

Since the instance has size $\frac{n}{t_k}$ and we have s_k instances, processing these tasks in parallel requires $\frac{n}{t_k} \cdot s_k \cdot x_k^{1-\varepsilon/3}$ space. If one want to fit this in $O(n)$ space, then it must be that $x_k \leq t_k^{(\varepsilon/3)/(1-\varepsilon/3)}$. Anyway, we get that the number of contractions we can make on k-th level is polynomial in the number of contractions we made on higher levels, and if the recurrence is solved, then it follows that it will be $O(\log\log n)$ levels until we reach a graph of a constant size.

Ghaffari and Nowicki [68], use Lemma B.2.2 and the above boosting scheme to show an $O(\log\log n \cdot \log n)$-round MPC algorithm for Min Cut. The main non-trivial part of their algorithm involves tracking the smallest singleton cut on each recursion level, which they do in $O(\log n)$ rounds because of the divide and conquer nature of their approach. Effectively, they assign all edges random and unique edge weights, and contract all uncontracted edges in decreasing order by edge weight. It can then be shown that all that needs to be done is to compute the MST of this graph and contract these edges accordingly (all other edges will be automatically contracted when another edge is contracted). We reduce the number of rounds for singleton cut tracking down to $O(1)$ rounds in the AMPC model. We aim to prove the following theorem.

Theorem B.1.1. There is an $O(\log\log n)$-round AMPC algorithm that uses $\widetilde{O}(n+m)$ total memory and $\widetilde{O}(n^{\varepsilon})$ memory per machine which finds a $(2+\varepsilon)$-approximation of Min Cut with high probability.

To track singleton cuts, the first step is to find a *low depth decomposition* of the current MST. At a high level, a low depth decomposition of a tree is a labeling of its vertices with values 1 through d, where d is the depth. This label must satisfy the following: for every level $i \in [d]$, the connected components induced on vertices with label at least i must contain at most one vertex for each i. This defines a recursive splitting process: starting at depth 1, there must be at most one vertex v with the minimum label, so we

can split the tree into multiple parts by removing v. Then we simply recurse on each connected component, considering the next set of labels, and knowing the process will always split each connected component once at a time. This is the general idea captured by both this and previous works. However, in order to increase the efficiency of this step, we require a new decomposition structure (see Definition B.1) and new methods for finding the decomposition. Notice that it is always true that at each level, each connected component contains at most one vertex at the next level.

In Section B.3, we show how to find a low depth decomposition with depth $O(\log^2 n)$ in AMPC in $O(1/\varepsilon)$ rounds (Lemma B.3.1) with $O(n^\varepsilon)$ space per machine. Roughly speaking, we create a heavy-light decomposition of the MST, where we store "heavy paths" consisting of edges connecting vertices to their children with the largest number of descendants and isolated "light nodes". We replace each heavy path with a complete binary tree whose leaves contain the vertices in the path, which gives us an efficient structure to obtain our labeling. This yields our low depth decomposition.

In the next step, we compute the size of of every singleton cut S that is created during the process. Note that the contractions are inherently sequential and the number of contractions we need to make at step k is $x_k \in O(n)$. However, each singleton cut is a connected component on the MST containing a specific edge e, whose contraction – in the increasing order of contracting MST edges – results in subset S, if we only allow the edges that have a smaller weight than e. We partition these connected components based on the vertex in the cut with the lowest level in the heavy-light decomposition of the MST. This way, we can compute every level of the low depth decomposition in parallel with only an $O(\log^2 n)$ blowup in total memory. In Section B.4, we show that we can track every singleton cut in the contraction process in $O(1)$ AMPC rounds. A high level pseudocode of the main algorithm is given in Algorithm 4.

Note that MinSingletonCut (Algorithm 6) is introduced in Section B.4 and it leverages

Algorithm 4: AMPC-MinCut
(An algorithm that calculates $(2+\varepsilon)$ approximation of Min Cut in G. The novel part is underlined.)

Data: A graph $G = (V(G), E(G))$, a parameter k.
Result: $(2+\varepsilon)$ approximation of Min Cut.

if $|G| \in n^{\varepsilon}$ **then**
  return *Min Cut of G calculated on a single machine*
end
Let $\widehat{G_1}$, ..., $\widehat{G_k}$ be copies of $G_1, \ldots, G_k$ with assigned random weight on edges (independently for each copy);
In parallel for all $i \in [k]$, $S_i \leftarrow \mathsf{MinSingletonCut}(\widehat{G_i})$;
In parallel for all $i \in [k]$, $G_i \leftarrow$ copy of $\widehat{G_i}$ after first k contractions;
In parallel, $C_i \leftarrow \mathsf{AMPC\text{-}MinCut}(G_i)$;
return $\min(S_1, \ldots, S_k, C_1, \ldots, C_k)$;

LowDepthDecomp (Algorithm 5) from Section B.3.

B.3 Generalized Low Depth Tree Decomposition

This section and the next address our algorithmic formulation and analysis. Note that all omitted proofs are deferred to the Appendix.

In order to efficiently compute the singleton cuts in parallel, we first need to compute an efficient decomposition of the MST. The low depth tree decomposition Ghaffari and Nowicki [68] introduce is a very specific decomposition with i levels such that at each level, the tree is split into two trees who share a root and no other vertices and are at least $1/3$ the size of the original graph. Unfortunately, it is unclear how to calculate this precise decomposition efficiently in AMPC. To work around this, we introduce a more generalized version of the low depth tree decomposition, show that it can be computed in AMPC, and later show that we can leverage this to obtain our Min Cut algorithm.

Definition B.1. A **generalized low depth tree decomposition** of some tree T is a labeling $\ell : V(T) \to [h]$ of vertices with **levels** for decomposition height $h \in O(\log^2 n)$ such that for

each level i, the connected components induced on $T^i = \{v \in T : \ell(v) \geq i\}$ have at most one vertex labeled i each.

To see what this means, consider the first level. We start by removing the sole vertex at the minimum level, which leaves a forest. For each tree in the forest, we must then have at most one vertex in the next level. We then remove the next level vertices (if they exist), and are left with a forest that is either the same or more refined. This process continues until the final, isolated vertices are labeled with the final level. Notice that we are guaranteed that at any level i, each connected component has at most one level i vertex that it will remove. The completion time of this process depends on the height of the decomposition, which in our case is $O(\log^2 n)$. We will, of course, make this more efficient in Section B.4.

It is not that hard to see that Ghaffari and Nowicki's low depth tree decomposition is a specific example of generalized tree decomposition with depth $O(\log n)$. They put a single vertex in the first level and then simply recurse on the two trees in the remaining forest. Note that they require additional properties of this decomposition to obtain their result, specifically that each new component has size at least $\frac{1}{3}$ of the original component, but we will see later that these are not necessary for finding the singleton cuts.

Like Ghaffari and Nowicki in MPC, we prove this can be computed efficiently in, instead, AMPC.

Lemma B.3.1. *Computing a generalized low depth tree decomposition of an n-vertex tree can be done in $O(1/\varepsilon)$ AMPC rounds with $O(n^\varepsilon)$ memory per machine and $O(n \log^2 n)$ total memory.*

The rest of this section is dedicated to proving Lemma B.3.1. The formal and complete algorithm is shown in Algorithm 5 and further details and definitions can be found later in this section. At a high level, our algorithm proceeds as follows:

1. Root the tree and orient the edges [line 2].

2. Contract heavy paths in a *heavy-light decomposition* of T into *meta vertices* to construct a *meta tree*, T_M [lines 3 to 5].

3. For each meta vertex, create a *binarized path*, a binary tree whose leaves are the vertices in the heavy path, in order. Expanding meta vertices in this manner yields our *expanded meta tree* [lines 7 to 11].

4. Label each vertex according to properties of the expanded meta tree. For all new vertices (i.e., vertices created in step 3) v, label v with the depth of the highest vertex u in the same meta node such that v is the leftmost leaf descending from the rightmost child of u in the binarized path of the meta node [lines 13 to 15].

Each of these steps correspond to the following subsections. For instance, step 1 corresponds to Section B.3.1. All relevant terminology related to these steps are additionally found in the corresponding subsections. Lemma B.3.1 is proven at the end of the final subsection.

B.3.1 Rooting the Tree

Like in Ghaffari and Nowicki, the first thing we need to do in line 2 of Algorithm 5 is compute an orientation of the edges. Fortunately this, along with rooting the tree, can be done quickly in AMPC by the results of Behnezhad et al. [29] in their Theorem 7.

Lemma B.3.2 (Behnezhad et al. [29])**.** *Given a forest F on n vertices, the trees in F can be rooted and edges can be oriented in $O(1/\varepsilon)$ AMPC rounds w.h.p. using $O(n^{\varepsilon})$ local memory and $O(n \log n)$ total space w.h.p.*

Here, w.h.p. means "with high probability." This completes the first step of our algorithm.

Algorithm 5: LowDepthDecomp
(Computing a generalized low depth tree decomposition of an input tree in AMPC)

Data: A tree $T = (V(T), E(T))$.
Result: A mapping $\ell : V(T) \to \mathbb{N}$ of tree vertices to levels.

1 Initialize $\ell : V(T) \to \mathbb{N}$;
2 Root and orient T;
3 Let $T_H = (V(T), \{e \in E(T) : e \text{ is heavy}\}$;
4 Let $\mathscr{P}$ be the connected components of T_H;
5 Let $T_M = (\mathscr{P}, \{(P_1, P_2) : P_1, P_2 \in \mathscr{P}, \exists (u_1, u_2) \in V(P_1) \times V(P_2)$ such that $(u_1, u_2) \in E(T)\})$;
6 **for** $v \in T_M$ *of heavy path* P_v *in parallel* **do**
7 Let $V(T_v)$ be a vertex set of size $2|P_v| - 1$ with associated indices $1, \ldots, 2|P_v| - 1$, denoted by i_u;
8 Let $T_v = (V(T_v), \{(u, p_u) : i_{p_u} = \lfloor i_u/2 \rfloor\})$;
9 Pre-order traverse T and sort P_v accordingly;
10 Pre-order traverse T_v and let L be its sorted leaves;
11 For all $i \in [|P_v|]$, map $P_v[i]$ to $L[i]$;
12 **for** $u \in V(T_v)$ **do**
13 Find path P^u to the root of the expanded meta-tree;
14 Let $u' \in V(T_v) \cap P^u$ be such that u is the leftmost descendant of u''s right child (otherwise $u' = u$);
15 Label $\ell(u) = d(u')$;
16 **end**
17 **end**
18 **return** ℓ *limited to the original vertices in* T;

B.3.2 Meta Tree Construction

We also leverage Ghaffari and Nowicki's notion of heavy-light decompositions for our AMPC algorithm, which can be found from lines 3 through 5 in Algorithm 5. This process allows us to quickly decompose the tree into a set of disjoint paths of *heavy edges*, which are defined as follows (note that our definition slightly deviates from Ghaffari and Nowicki [68], where the heavy edge must extend to the child with the largest subtree without requiring this subtree to be that large, though it is the same as the definition used by Sleator and Tarjan [130]):

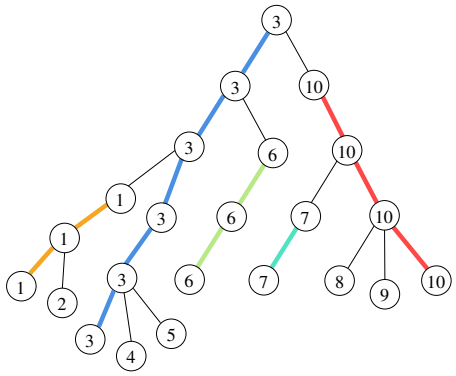


Figure B.1: The heavy-light decomposition of an example tree.

Definition B.2 (Sleator and Tarjan [130])**.** Given a tree T and a vertex $v \in T$, let $\{u_i\}_{i \in k}$ be the set of children of v where the subtree rooted at u_1 is the largest out of all u_i. If there is no strictly largest subtree, we arbitrarily choose exactly one of the children with a largest subtree. Then (u_1, v) is a **heavy edge** and (u_i, v) is a **light edge** for all $1 < i \leq k$.

Then the definition of a heavy path follows quite simply.

Definition B.3 (Ghaffari and Nowicki [68])**.** Given a tree T, a **heavy path** is a maximal length path consisting only of heavy edges in T.

Ghaffari and Nowicki then make the observation that the number of light edges and heavy paths is highly limited in a tree. This comes from a simple counting argument, where if you consider the path from root r to some vertex v, any time you cross a light edge, the size of the current subtree is reduced by at least a factor of 2. This holds even with our different notion of heavy edges since subtrees rooted at children of light edges are still much smaller compared to the subtree rooted at the parent vertex. This bounds the number of light edges between r and v, where each pair of light edges are separated by at most one heavy path, and therefore it also bounds the number of heavy paths.

Observation B.3.1 (Ghaffari and Nowicki [68])**.** *Consider a tree T oriented towards root r. For each vertex v, there are only $O(\log n)$ light edges and only $O(\log n)$ heavy paths on*

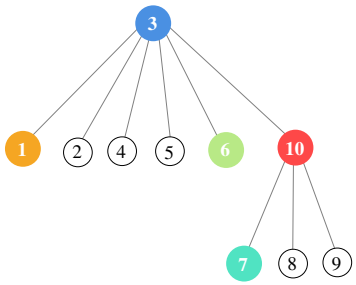


Figure B.2: The *meta-tree* of the same tree from Figure B.1 is demonstrated in this figure.

the path from v to r.

Using the definition of heavy edge from Sleator and Tarjan [130] instead of from Ghaffari and Nowicki, we get an additional nice property. This is because in our definition, every internal vertex has one descending heavy edge to one child.

Observation B.3.2 (Sleator and Tarjan [130])**.** *Given a tree T and an internal vertex $v \in T$, v must be on exactly one heavy path. For a leaf $\ell \in T$, ℓ must be on at most one heavy path.*

Our first goal is to compute what we call a *meta tree*. This is a decomposition of our tree that will allow us to effectively handle heavy edges. It is quite analogous to Ghaffari and Nowicki's notion of the heavy-light decomposition, which partitions the tree into heavy and light edges.

Definition B.4. Given a tree T, the **meta tree** of T, denoted T_M, comes from contracting all the heavy paths in T. We call the vertices of T **original vertices** and the vertices of T_M **meta vertices**.

Note that contracting all heavy paths simultaneously is valid because, by Observation B.3.2, all heavy paths must be disjoint. Additionally, all internal meta vertices are contracted heavy paths (as opposed to original vertices), again by Observation B.3.2. We note that in AMPC, since connectivity is easy, it is additionally quite easy to contract the heavy paths of T into single vertices.

Lemma B.3.3. *Given a tree T, the meta tree T_M can be computed, rooted, and oriented in AMPC in $O(1/\varepsilon)$ rounds with $O(n^{\varepsilon})$ memory per machine and $O(n\log^2 n)$ total space w.h.p.*

Proof. First off, we know subtree size can be computed in $O(1/\varepsilon)$ low-memory AMPC rounds on trees as shown by Behnezhad et al. [29], and the child of a vertex v with minimum subtree can then be found in $O(1/\varepsilon)$ rounds by dividing the children amongst machines and iteratively finding the smallest. Next, consider removing all light edges from T to create a forest F. Run Behnezhad et al.'s AMPC connectivity algorithm [30], which satisfies the round and space constraints, to identify the components and contract them. Add the light edges back in to connect contracted nodes. This clearly is T_M. Additionally, as before, we run Behnezhad et al.'s [29] AMPC algorithm for orienting the tree. This too falls within the constraints. □

This completes the second step of our decomposition algorithm.

B.3.3 Expanding Meta Vertices

In order to label the vertices, we need a way to handle the heavy paths corresponding to each meta vertex. Let $v \in T_M$ be a meta vertex, and P_v be the heavy path of original vertices in T corresponding to v. Note that we have no stronger bound on the length of a heavy path than $O(n)$. Therefore, a recursive partitioning, or labeling of vertices that has polylogarithmic depth must be able to cleverly divide heavy paths. We can do this with a new data structure.

Definition B.5. Given some path P, a **binarized path** is an almost complete binary tree T with $|P|$ leaves where there is a one-to-one mapping between P and the leaves of T such that the pre-order traversal of P and T limited to its leaves agree.

By "agree", we mean that if a vertex $v \in P$ comes before a vertex $u \in P$ in the pre-order traversal of P, then it also does in the pre-order traversal of T. To characterize this tree, we make a quick observation:

Observation B.3.3. *An almost complete binary tree on n leaves has* $2n-1$ *vertices,* $\lfloor \log_2 n \rfloor + 1$ *max depth, and every layer is full except the last, which has* $2n - 2^{\lfloor \log_2 n \rfloor + 1}$ *vertices.*

Additionally, we can find a relationship between the ancestry of triplets in P based off of the order of the three vertices. While this is not required for expanding meta vertices, it is a property of the binarized path that will be useful when we label vertices later.

Observation B.3.4. *Given a binarized path T of a path P, for any* $u, u', u'' \in P$ *that appear in that order (or reversed), if v is the lowest common ancestor of u and* u' *and* v' *is the lowest common ancestor of u and* u''*, then* v' *is an ancestor of v or* $v' = v$*.*

Proof. Consider such a $u, u', u'' \in P_v$ that appear in this order (or reversed), and let v and v' be the lowest common ancestors of u and u', and u and u'' respectively. Consider traversing from u up the tree from child to parent, and let p be the current vertex. We start with $p = u$ and thus the leaf set of the subtree rooted at p is $L_p = \{u\}$. As we traverse upwards, we add sets of leaves to L_p that are contiguous in P_v. Additionally, one vertex is directly adjacent to a vertex from L_p in P_v because P_v was mapped to the leaves of T_v according to the pre-order traversal. Therefore, L_p is a contiguous chunk of P_v. Thus, when u'' gets added to the subtree (i.e., when $p = v'$) u' must be in L_p too, either because it was added previously or it is being added at the same time. In the former case, v must have happened before v' and thus v' is an ancestor of v, and in the latter case, $v = v'$. □

To create the tree, we do the following for every $v \in T_M$:

1. Create an almost complete binary tree T_v with $|P_v|$ leaves, linking children to parents and noting if a vertex is a left or right child [lines 7 and 8].

2. Do a pre-order traversal of T_v and P_v and map the vertices in P_v to the leaves of T_v such that the pre-order traversal of P_v and of T_v limited to its leaves agree. [lines 9 to 11].

Next, it is pretty direct to see that the produced tree is a binarized path.

Observation B.3.5. *The process described above produces a binarized path T_v of P_v for all v.*

We prove that this can be done in the proper constraints.

Lemma B.3.4. *The heavy paths of a tree can be converted into binarized paths in $O(1/\varepsilon)$ AMPC rounds with $O(n^{\varepsilon})$ local memory and $O(n \log n)$ total space w.h.p.*

Proof. Correctness of the process described in this section is seen in Observation B.3.5. Thus we simply need to show how to implement it in AMPC. For the first step, we must construct a generic almost complete binary tree with $|P_v|$ leaves. Call this tree T_v. By Observation B.3.3, this has $2|P_v| - 1$ vertices, $\lfloor \log_2 |P_v| \rfloor + 1$ max depth, and each layer is full except the last which has $2|P_v| - 2^{\lfloor \log_2 |P_v| \rfloor + 1}$ vertices. Thus, it is fairly simple to, in parallel, create the set of all vertices in the tree and then connect each vertex to its parent. Each vertex can be given an index: a unique identifier for vertices numbered $1, \ldots, 2|P_v| - 1$. This is going to represent the order of the vertices in a breadth-first traversal of the tree. For a vertex with index $1 < i < 2|P_v|$, its parent's index j can be computed as $j = \lfloor i/2 \rfloor$. Whether or not a vertex is a left or right child is simply determined by the parity of its pre-order index.

Note that the size of the tree is $O(|P_v|)$ but each individual processor computation (i.e., computing the size of the tree, and then having each index connect itself to its parent) can be done in constant space and 2 rounds. Thus in 2 rounds, we can create such a tree. Note that we have to construct these trees in parallel, but it is not hard to see that this will only require $O(n \log n)$ total space which can be divided appropriately amongst machines.

For the second step, we can use Behnezhad et al.'s [29] algorithm for pre-order numbering with $O(1/\varepsilon)$ AMPC rounds w.h.p. using $O(n^{\varepsilon})$ local memory and $O(n)$ total space. Let L be a list of the leaves of T_v in pre-order. To map P_v to the leaves, one can simply do a direct map between P_v and L in one round. □

B.3.4 Labeling Vertices

Our next goal is to label the vertices with the level they should be split on. Consider, hypothetically, expanding the meta tree T_M such that every heavy path for a meta vertex v is replaced with its binarized path (which is an almost complete binary tree) T_v, and the tree continues at the leaves corresponding to the nodes in the heavy path. Note that only some vertices in the hypothetical tree correspond to vertices in the original tree T. Specifically, the internal nodes of each component subtree T_v are not vertices in T, but the leaves correspond exactly to the vertices in T.

Ultimately, for a vertex $u \in T$ in meta vertex v, let u' be the vertex in T_v such that u is the leftmost leaf-descendant of the right child of u' in T_{u_M} (or if this doesn't exist, $u' = u$). Then we will label $\ell(u) = d(u')$ where d is the depth in the expanded meta tree. Following this, our vertex labeling process will be as follows for each $v \in T_M$ and $u \in T_v$:

1. Vertex u finds the path P^u from u to the root of T_M, assuming the meta vertices are expanded [line 13].

2. Let u' be the highest vertex in T_v such that u is the leftmost descendant of the right child of u. If there is no such vertex, let $u' = u$ [line 14].

3. Label u with the depth (assuming roots have depth 1) of u' in the expanded T_M [line 15].

We start by making a quick observation that comes directly from Observations B.3.1

and B.3.3.

Observation B.3.6. *The max depth of T_M with meta nodes expanded ("the expanded T_M") into binary trees is $O(\log^2 n)$.*

This will be greatly helpful in showing the efficiency of our algorithm. We now show that this final part can be implemented efficiently, which is sufficient to prove our main lemma.

Lemma B.3.5. *The process described above finds a generalized low depth tree decomposition of original tree T of height $h \in O(\log^2 n)$ in 1 round with $O(n^\varepsilon)$ local memory and $O(n \log^2 n)$ total space.*

Proof. First, note that P^u can be stored entirely on one machine by Observation B.3.6, and additionally, since both T_M and T_v for all $v \in M$ is oriented, it is quite simple to adaptively query the path from u to the root in one round within the space constraints. Assuming the orientations also labels if the vertex is a left or right child, u' can be found simply by searching the path. Finally, the depth of u', which is the label of u, can also be found quite simply given access to all of P^u.

It is quite simple to show the height is bounded by $O(\log^2 n)$: all labels are depths in the expanded T_M and Observation B.3.6 bounds the max depth. All that is left is to show the connected components induced on $T^i = \{v \in T : \ell(v) \geq i\}$ contain exactly one vertex with label i each.

We show this by induction on the level. At the 1st level, we should only partition the graph once. Let u be a vertex labeled 1 and P^u be its path. For u to be given depth 1, it must have received its label from the root r_M of the expanded T_M, since we are counting depth starting at 1. Thus, it must be the leftmost descendant of the right child of r_M. This is clearly unique, thus u is unique. Therefore, there is exactly one vertex at the 1st level.

Consider a connected component C in $T^i = \{v \in T : \ell(v) \geq i\}$ for some level $i \in h$. Let the "neighborhood" $N(C)$ be all the vertices in $T \setminus C$ that are adjacent to some vertex in C. We will first show that for the largest level j of a vertex in $N(C)$, there is exactly one vertex in $N(C)$ of level j. Note $j < i$, otherwise a vertex of level j in $N(C)$ would actually be included in the component C.

Assume for contradiction there are at least two such vertices, $u, v \in N(C)$ with level j. Let u' and v' be their respective neighbors in C. Since C is a connected component, there is a path P from u' to v' containing only vertices in C. All of these vertices have level i or higher by the definition of C. Tacking on u and v to the start and end of P respectively, there is a path from u to v consisting of vertices $\{u, v\} \cup P$. Since u and v are at level j, that means every vertex in this path has level j or higher. Therefore, u and v must have been in the same connected component C' in the earlier level T^j. By induction, that component must have had only one vertex at level j. This is a contradiction. Thus $N(C)$ must have exactly one vertex in level j.

Let $v \in N(C)$ be the vertex at level j, and let u be its neighbor in C (note there can only be one since T is a tree). We consider three cases.

Case 1 u is a child of v and they are not on the same heavy path. Let u_M and v_M be the meta vertices in T_M containing u and v respectively, u' and v' be the corresponding nodes found in step 2 for u and v (whose depths are the labels of u and v), and r_{u_M} be the root of the binarized path for u_M. Clearly, u_M is a child of v_M since u is a child of v and $u_M \neq v_M$. Moreover, r_{u_M} must be the child of v in the expanded T_M. Since v' is an ancestor of v, that means its depth in the expanded T_M satisfies $d(v') < d(r_{u_M})$. Since this defines the label of v', $\ell(v) \leq d(r_{u_M}) - 1$.

Consider any leaf $l \in T_{u_M}$. Its label is the depth of some vertex $w \in T_{u_M}$, which must be deeper than the root r_{u_M}. Thus $\ell(l) \geq d(r_{u_M}) \geq \ell(v) + 1$, then implying all of T_{u_M} is in

C. Additionally, since only one $l \in T_{u_M}$ is labeled by r_{u_M}, i.e. $\ell(l) = d(r_{u_M})$, it must have a unique (and smallest) label out of all vertices in P_{u_M}. It turns out this $\ell(l)$ will be our j.

Now, consider l's placement in the original tree T. It is on a heavy path containing u, and it has the smallest label on the heavy path. Vertex u is also directly adjacent to v. Thus in level i when we consider the tree induced on T^i, it must be in the same component as u since the path from l to u is contained in T^i. Additionally, since T is a tree, the only vertex in $N(C)$ above the component itself is v. Therefore, all of C is a descendant of v, and thus they must also be descendants of u. Thus, in T_M, they must have been in meta vertices at the depth of or below u_M. By a similar logic as before, their label must be strictly greater than l's label. This implies that $\ell(l) = j$, and it is the only vertex in T_{u_M} with such a label.

Case 2 u is the parent of v and they are not on the same heavy path. Using u_M and v_M as before and with the same logic as before but with reversed roles, we find that $\ell(v) > \ell(u)$ so $\ell(u) < i$. This contradicts that u is in T^i, and thus this case is impossible. Note that this also shows that C cannot contain a vertex in an ancestor u_M of v_M in T_M, because by connectivity, this would imply that there is some u that is the parent of the root of T_{v_M}. A similar argument will hold.

Case 3 u and v are on the same heavy path corresponding to meta vertex u_M whose binarized path T_{u_M} has root r_{u_M}. Let $P \subseteq P_{u_M}$ be the connected subpath of this heavy path that contains u if we remove all vertices of level $i-1$ or lower. Obviously, $P = C \cap P_{u_M} \subseteq C$. Assume for contradiction that two vertices $p, p' \in P$ have level i, so $\ell(p) = \ell(p') = i$. Let $a = lca_M(p, p')$ be the least common ancestor of p and p' in T_{u_M}. Because a is the least common ancestor, one of its children must contain p and the other p' in its subtree. Without loss of generality, assume p is a descendant of the left child and p' is a descendant of the right child.

Let w and w' be the (possibly internal) vertices of T_{u_M} such that p and p' are labeled with their expanded meta tree depths in step 3 respectively ($\ell(p) = d(w)$ and $\ell(p') = d(w')$). This means $i = d(w) = d(w')$. Also, notice that w is an ancestor of p and w' is an ancestor of p', so they are on the respective paths from p and p' to the root of T_{u_M}, call it r_{u_M}. Additionally, a is on both paths, and specifically the paths must merge at a. Since w and w' only pass their depth to exactly one leaf of T_{u_M} each, they must be distinct. Therefore, they cannot be at depth $d_{u_M}(a)$ or lower, else they would both be a. Thus, w and w' are strict descendants of a, so $d(w), d(w') > d(a)$, meaning that $\ell(p), \ell(p') > d(a)$.

Let $\overline{p}$ be the leftmost leaf descendant of the right child of a. Then $\overline{p}$ was labeled by a, thus $\ell(\overline{p}) = d(a)$. Therefore, $\ell(p), \ell(p') > \ell(\overline{p})$. Since p and p' are at level i, $\ell(\overline{p}) < i$. Additionally, note that by the structure of T_{u_M}, $\overline{p}$ is necessarily located between p and p' in the leaf set, thus it must come between them in the pre-order traversal of T_{u_M} by Observation B.3.4. Since T_{u_M} is the binarized path of P_{u_M}, this also implies $\overline{p}$ comes between p and p' in P_{u_M}, and by extension P. Note, however, that $P \subseteq C \subseteq T^i$, and thus its vertices must all have level i or greater. This contradicts that $\ell(\overline{p}) < i$. Thus, it must be the case that there is at most vertex in P that has level i.

Finally, all that needs to be shown is that for every vertex $c \in C \setminus P$, $\ell(c) > i$. For any such c, let c^* be its lowest ancestor in P_{u_M}. Since $\ell(c^*) \geq i$, we can make the same argument as Case 1 to show that c must have label $i+1$ or larger. □

Proof of Lemma B.3.1. By combining Lemmas B.3.2, B.3.3, B.3.4, and B.3.5, we have our result. □

B.4 Calculating the smallest singleton cut

In this section, we show a $O(1/\varepsilon)$ round AMPC algorithm that executes a series of contractions and outputs the size of the smallest singleton cut that appeared during the

contraction process. That is we prove the following result.

Theorem B.4.1. *There exists an AMPC algorithm that given a graph G with unique weights on edges calculates the minimum singleton cut that appears during the contraction process in $O(1/\varepsilon)$ rounds using $O(n^{\varepsilon})$ local memory and $O((n+m)\log^2 n)$ total space.*

B.4.1 Contraction process

We view the contraction process of a weighted graph $G = (V, E, w : E \to [n^3])$ as a sequential process in which we iterate over multiple timesteps 0 to n^3. For a given time i, we contract the edge e having $w(e) = i$ to a single vertex. Let $G_0, \ldots, G_{n^3}$ be the sequence of graphs created in the process, where G_0 denotes the graph before any contraction and G_{n^3} denotes the graph after all contractions. Via a quick comparison to Kruskal's algorithm, it is clear that the edges whose contraction changed the topology of the graph must belong to the minimum spanning tree of the weighted graph G (since weights are unique, the MST is unique as well). Let $T = (V, E_T, w : E_T \to [n^3])$ be the minimum spanning tree of G.

From the previous observation, it is enough to consider only contracting edges from tree T, which we will focus on in the rest of this section. It will also be convenient visualize vertices as simply being grouped instead of fully contracted.

Definition B.6. A **bag** of vertex v at *time* $t \in [n^3]$, which we denote $\mathsf{bag}(v,t)$, is the set of vertices that can be reached from v using only edges of tree T of weight at most t. We denote $\mathsf{nbr\_bag}(v,t)$ for set of **neighbors of a bag**, that is set of these vertices u that do not belong to the bag and there exists an edge connecting u and any vertex of the bag of weight greater than t. The **degree of a bag**, denoted $\Delta\mathsf{bag}(v,t)$, is the size of the set $\mathsf{nbr\_bag}(v,t)$.

If we proceed with our edge contraction process, where an edge with weight t is contracted at time t, then $\mathsf{bag}(v,t)$ is the set of all vertices that have been contracted with v at

time t. The value $\Delta\mathsf{bag}(v,t)$ is simply the degree of the vertex that corresponds to contracted vertices. Therefore, the following simple observations holds.

Observation B.4.1. *The value of the minimum singleton cut in the contraction process of the weighted graph G is equal to the following:*

$$\min_{v\in V, t\in[n^3]} \Delta\mathsf{bag}(v,t).$$

B.4.2 Simulating tree contractions with low depth decomposition

By Observation B.4.1 our goal is to calculate the value of

$$\min_{v\in V, t\in[n^3]} \Delta\mathsf{bag}(v,t).$$

To find this, we could calculate the value $\min_{t\in[n^3]} \Delta\mathsf{bag}(v,t)$ for every vertex v independently in parallel. However, this would require a minimum of $\Omega(n\cdot(n+m))$ total space, which roughly corresponds to replicating the whole graphs for each independent instance. There are two key observations that will allow us to reduce the space complexity. First, bags are determined solely from the topology of tree T. Second, for larger t, it is likely the case that $\mathsf{bag}(u,t) = \mathsf{bag}(v,t)$, so we would like to remove this redundant computation. Therefore, we will exploit tree properties and the low depth decomposition to partition the work and avoid redundancy.

Let $\ell : V \to [h], h \in O(\log^2 n)$ be the labeling from the generalized low depth decomposition of tree T (see Definition B.1). Let us asses to each bag a uniquely chosen vertex.

Definition B.7. The **leader** of a bag, denoted $\mathsf{bagLeader}(v,t)$, is the vertex u with the smallest label $\ell(u)$ among all vertices from $\mathsf{bag}(v,t)$. We define a number $\mathsf{ldr\_time}(v)$ to be

the greatest number $0 \leq t' \leq n^3$ such that $\mathsf{bagLeader}(v,t') = v$.

Let us first argue the correctness of the above definitions.

Lemma B.4.2. *The leader of every bag can be determined uniquely. Also, for every vertex $v \in V$ it holds: the number $\mathsf{ldr\_time}(v)$ exists, $\mathsf{ldr\_time}(v) \geq 0$, and for every $0 \leq t' \leq \mathsf{ldr\_time}(v)$ we have that $\mathsf{bagLeader}(v,t') = v$.*

Proof. Consider $\mathsf{bag}(v,t)$ for some $v \in V$, $t \in [n^3]$. Because $v \in \mathsf{bag}(v,t)$, this bag cannot be empty. Let u be the vertex with the smallest label among all vertices from $\mathsf{bag}(v,t)$. Assume for contradiction that there exists another vertex $u' \in \mathsf{bag}(v,t)$ such that $\ell(u) = \ell(u')$. Observe, that there exists only one path between u and u' in tree T and all vertices of this path must be contained in $\mathsf{bag}(v,t)$ since $\mathsf{bag}(v,t)$ always forms a connected component when viewed as a subtree of T, c.f. Definition B.6. From the properties of the low depth decomposition, we get that there is a vertex z on the path with label smaller than both $\ell(u)$ and $\ell(u')$ which contradicts with the choice of u. This also proves the uniqueness of the leader. Let us now argue about $\mathsf{ldr\_time}(v)$. First, for every $v \in V$ we have that $bag(v,0) = v$ as there is no edge with weight smaller than 1. This already proves two first properties. Second, it holds: $\mathsf{bag}(v,0) \subseteq \mathsf{bag}(v,1), \ldots, \mathsf{bag}(v,n^3)$ as a bag defined for larger time can expand more edges. Therefore, if v is the leader of $\mathsf{bag}(v,t)$ for some $0 \leq t \leq n^3$, it has to be the leader of any subset of this bag. □

Using the fact that each bag has exactly one leader, we can reformulate the expression $\min_{v \in V, t \in [n^3]} \Delta\mathsf{bag}(v,t)$ as follows

$$\min_{v \in V, t \in [n^3]} \Delta\mathsf{bag}(v,t) = \min_{v \in V} \min_{0 \leq t \leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t).$$

We then will distribute the work needed to calculate the right-hand side of the above equality by requiring each vertex to calculate the minimal degree among bags for which it is the

leader:

$$\min_{0\le t\le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t).$$

Let i be a number in $\left[\lceil log^2 n\rceil\right]$. Let L_i (the ith level) be the set of vertices $v \in V$ with low depth decomposition label $\ell(v) = i$, and $L_{\le i}$ be that with label $\ell(v) \le i$ (for convenience we assume that $L_{\le 0} = \emptyset$). Let T^i be the tree T with $L_{\le i-1}$ removed. The following observation, derived from the fact that a bag is a connected subgraph of T and the leader has lowest value $\ell(\cdot)$, relates bag location to the topology of the low depth decomposition.

Observation B.4.2. *For every $i \in \left[\lceil log^2 n\rceil\right]$, $v \in L_i$, and $0 \le t \le \mathsf{ldr\_time}(v)$, the set $\mathsf{bag}(v,t)$ belongs to a single connected component of graph T^i. For any two $u, v \in L_i$, sets $\mathsf{bag}(u, \mathsf{ldr\_time}(u))$ and $\mathsf{bag}(v, \mathsf{ldr\_time}(v))$ belong to different components of graph T^i.*

Recall, that we wanted to calculate the value

$$\min_{0\le t\le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t)$$

for every $v \in V$, which we rewrote as

$$\min_{v\in V} \min_{0\le t\le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t).$$

Grouping by vertices in the same layers, we get

$$\min_{v\in V} \min_{0\le t\le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t)$$
$$= \min_{i\in\left[\lceil log^2 n\rceil\right]} \min_{v\in L_i} \min_{0\le t\le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t).$$

By Observations B.4.2, we can hope that computing the value

$$\min_{v \in L_i} \min_{0 \le t \le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t),$$

can be done in parallel without exceeding global memory limit of $O(m \log^2 n)$, since for different $v \in L_i$, their bags up to time $\mathsf{ldr\_time}(v)$ belong to different components of T^i, thus we might avoid redundant work. The details of computing this value are presented in the next section. Let us now formalize the progress so far.

Lemma B.4.3. *Given a tree T and a graph $G = (V, E, w : E \to [n^3])$ as an input, calculating the value $\min_{v \in V, t \in [n^3]} \Delta\mathsf{bag}(v,t)$ can be reduced to $O(\log^2 n)$ instances of calculating values*

$$\min_{v \in L_i} \min_{0 \le t \le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t),$$

for $i \in \left[\lceil log^2 n \rceil\right]$. The reduction can be implemented in AMPC with $O(1/\varepsilon)$ rounds, $O((n+m)\log^2 n)$ total space, and $O(n^\varepsilon)$ local memory.

Proof. The correctness follows from the above discussion. For the implementation, the generalized low depth decomposition of T can be determined in $O(1/\varepsilon)$ rounds with $O(n \log^2 n)$ total space by Lemma B.3.1. Consider now $O(\log^2)$ tuples of format (T, ℓ, E, L_i). Preparing them requires $O((n+m)\log^2 n)$ total space and the above discussion shows that the value

$$\min_{v \in L_i} \min_{0 \le t \le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t)$$

for every $i \in \left[\lceil log^2 n \rceil\right]$ can be computed from the tuple (T, ℓ, E, L_i), thus the lemma follows.

□

B.4.3 Resolving the problem for vertices on the same level.

Following Lemma B.4.3, we fix $i \in \left[\lceil log^2 n \rceil\right]$ and set L_i. We calculate:

$$\min_{v \in L_i} \min_{0 \leq t \leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t).$$

In this appraoch, we will frequently query the minimum value over a path in a tree, thus the following result is helpful.

Theorem B.4.4 (Behnezhad et al. [31])**.** *Consider a rooted, weighted tree T, the heavy-light decomposition of this tree together with an RMQ data structure that supports queries on heavy paths can be computed in $O(1/\varepsilon)$ AMPC rounds using $O(n^{\varepsilon})$ local memory and $O(n \log n)$ total space. If the aforementioned data structures are precomputed, then obtaining a minimum value on a path of a tree can be calculated with $O(\log n)$ queries to global memory.*

We will also make use of the following theorem.

Theorem B.4.5 (Behnezhad et al. [28])**.** *For a given sequence of integer number S of length n, computing the minimum prefix sum over all prefix sums can be done in $O(1/\varepsilon)$ AMPC rounds using $O(n^{\varepsilon})$ local memory and $O(n \log n)$ total space.*

Finally, we show that the construction of the low depth decomposition provided in Section B.3 gives easy access to edges that connect vertices of higher labels with vertices of smaller labels.

Lemma B.4.6. *For any connected component C^i in T^i, there are at most 2 tree edges between C^i and $V \setminus T^i$ according to the low depth decomposition ℓ given in Lemma B.3.5. Moreover, both edges can be calculated in $O(1/\varepsilon)$ AMPC rounds with $O(n^{\varepsilon})$ memory per machine and $O(n \log^2 n)$ total memory.*

Proof. Let v be a vertex of this component with lowest value $\ell(v)$. By the construction of the low depth decomposition, the component C_i consists of some connected part P_1 of binarized path P to which v belongs and all binarized paths incident to the part F that contain vertices with larger values ℓ, denote this set P_2. Note, that each vertex $v \in P_2$ has no edged to the part of graph $V \setminus T^i$. Thus, only vertices from part P_1 can be connected with vertices of smaller labels than i. However, since P_1 forms a path, and we consider only tree edges, then there can be at most two such edges.

The above proof instructs also how to compute these two edges. First, the vertex v can determined in $O(1/\varepsilon)$ rounds, since it requires computing max function only over labels of vertices belonging to C_i. Assume, that for each vertex of a tree, we store not only its value $\ell(v)$, but also its position and the length of the binarized path to which it belongs. Then the first vertex to the left with label smaller than v and the first vertex to the right with label smaller than v on v's binarized path can be computed in constant time in local memory of a single machine, since their positions in the binarized path are functions of only the length of the path and the position of v. Assumed, that in the global memory all binarized paths are stored, then the corresponding edges can be found in $O(1)$ queries to the global memory. □

Let us now turn to the proper part of this subsection. First, we show how to compute values $\mathsf{ldr\_time}(v)$ for all $v \in L_i$.

Lemma B.4.7. *Given a tuple (T, ℓ, E, L_i) for tree T, low depth decomposition ℓ, set of weighted edges E, and levels L_i for some $i \in [\lceil \log^2 n \rceil]$, there exists an AMPC algorithm that calculates the value $\mathsf{ldr\_time}(v)$ for every $v \in L_i$, in $O(1/\varepsilon)$ rounds using $O(n^\varepsilon)$ local memory and $O((n+m)\log^2 n)$ global memory.*

Proof. Consider vertex $v \in L_i$. Vertex v ceases to be the leader of a bag at the first time t when its bag is contracted with another bag containing at least one vertex of the set

$L_{\leq i-1}$. According to the tree contraction process, time t is equal to the largest weight of tree edges between v's connected component in graph T^i and the set of vertices $L_{\leq i-1}$. By Lemma B.4.6, these edges can be extracted with at most $O(\log^2 n)$ queries to the low depth decomposition structure. We then simply find the minimum. Thus, all values $\mathsf{ldr\_time}(v)$ for vertices from L_i can be computed in constant number of rounds assumed the conditions stated in the lemma. □

We can assume that values $\mathsf{ldr\_time}(v) \in L_i$ are known. We would like to efficiently compute

$$\min_{0 \leq t \leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t),$$

for each $v \in L_i$. For this, we make the following observation.

Lemma B.4.8. *Consider an edge $(x,y) =: e \in E$ and a vertex $v \in L_i$. All possible values $0 \leq t' \leq \mathsf{ldr\_time}(v)$ at which e belongs to set $\mathsf{nbr\_bag}(v,t')$ form a consecutive (possible empty) interval of integers $[a_e, b_e] \subseteq [0, \ldots, \mathsf{ldr\_time}(v)]$, called also a **time interval** with respect to v.*

Proof. The lemma follows immediately from the fact that $\mathsf{bag}(v,0) \subseteq \mathsf{bag}(v,1) \subseteq \ldots \subseteq \mathsf{bag}(v,n^3)$. □

Additionally, the following observation shows, given edge time intervals, how to derive $\min_{0 \leq t \leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t)$ and clarifies the purpose of time intervals.

Observation B.4.3. *Fix a vertex $v \in V$ and consider time intervals $[a_e, b_e]$ with respect to v, for all $e \in E$. Denote the set of all intervals containing value x by I_x. Then, computing the value*

$$\min_{0 \leq t \leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t),$$

is equivalent to computing the minimum over all values $|I_x|$ for x in the range $[0, \mathsf{ldr\_time}(v)]$.

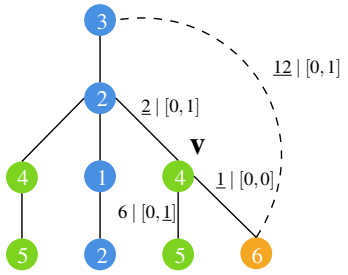


Figure B.3: A sample structure of an MST tree. Firm edges are *tree* edges, while dotted are *non-tree* edges. The number inside vertices denote their levels. Different colors symbolize different binarized paths. The numbers underlined are times of contraction of corresponding edges. Next to these number the time intervals of these edges with respect to vertex v are given. Since $\mathsf{ldr\_time}(v) = 2$, thus all these intervals are contained in $[0,2]$.

Since this task is 'linear', it can be computed efficiently in AMPC. We now discuss how to compute the intervals for all edges in E.

Lemma B.4.9. *Given a tuple (T, ℓ, E, L_i), there exists an AMPC algorithm that for every vertex $v \in L_i$ and every edge $e \in E$ calculates the maximal, non-empty time interval $[e_a, e_b] \subseteq [0, \ldots, \mathsf{ldr\_time}(v)]$ of e with respect to v. The algorithm works in $O(1/\varepsilon)$ rounds, uses $O(n^{\varepsilon})$ local memory and $O((n+m)\log^2 n)$ global memory.*

Proof. The algorithm starts by removing vertices $L_{\leq i-1}$ with all edges adjacent to them from tree T which gives us T^i. Given decomposition ℓ, this can be done in $O(1)$ rounds. By definition B.1, vertices $L_i = \{v_1, \ldots, v_q\}$ belong to different trees. Next, the algorithm roots these trees that contain vertices from L_i in $v_1, \ldots, v_q$ and calculates heavy-light decompositions of each tree together with an RMQ structure on heavy paths. By Theorem B.4.4, this can be done in $O(1/\varepsilon)$ within our memory constraints.

Let us now fix an edge $(x,y) =: e \in E$. Importantly, we consider here all edges of the graph G, not only tree edges E_T. Let $r_x \in \{\perp, v_1, \ldots, v_q\}$ be the root of this tree in T^i to which the vertex x belongs. If the vertex x does not belong to any tree, that is $x \in L_{\leq i-1}$ since these are the vertices that have been removed, we write $r_x = \perp$. Let $\mathsf{mw}(x)$ be the minimum weight over edges of path that connects vertex x with vertex r_x in graph T^i.

Observe, that unless $r_x = \perp$ this value is well defined as T^i is a collection of tree and there is exactly one path connecting is x and r_x in this graph. We extend the above definitions on y in the natural way.

By Theorem B.4.4, computing $r_x, r_y, \mathsf{mw}(x), \mathsf{mw}(y)$ takes $O(\log n)$ queries to the memory for a single edge. Therefore, we can compute these values for all edges $e \in E$ in $O(1)$ round under the conditions assumed in this lemma.

Observe that edge $e = (x,y)$ can have non-empty time intervals only with vertices r_x and r_y. Any other vertex from L_i belongs to a different connected component in graph T_i and therefore its bag cannot contain x nor y while the vertex is the leader of its bag. Thus, all that is left to show is how $\mathsf{mw}(x)$ and $\mathsf{mw}(y)$ can help determine the time intervals in which edge e belongs to $\mathsf{nbr\_bag}(r_x)$ and $\mathsf{nbr\_bag}(r_y)$. We consider the following cases.

Case 1. $r_x = \perp, r_y = \perp$. In this case, edge (x,y) has no effect on degrees of bags of vertices r_x and r_y at any time. The algorithm skips such edges.

Case 2. $r_x = \perp, r_y \neq \perp$, (or symmetrically $r_x \neq \perp, r_y = \perp$). Since T^i is a subset of the minimum spanning tree T, thus the first time when vertex x belongs to r_x's bag is the time $\mathsf{mw}(x)$. Now, y starts to belong to r_x's bag either at the time being equal to the maximal weight on the path between r_x and y. Observer however, that this path has to contain vertices that does not belong to T^i and therefore the maximal weight has to be greater than $\mathsf{ldr\_time}(r_x)$. What follows the correct interval in this case is:

$$[\mathsf{mw}(x), \mathsf{ldr\_time}(r_x)],$$

or an empty interval if $\mathsf{mw}(x) > \mathsf{ldr\_time}(r_x)$.

Case 3. $r_x \neq \perp, r_y \neq \perp$. We distinguish two sub-cases:

Subcase a) $r_x \neq r_y$. Since the path between r_x and r_y does not belong to T^i we can

proceed analogously to the *Case* 2.. The correct interval for vertex x is

$$[\mathsf{mw}(x), \mathsf{ldr\_time}(r_x)],$$

or an empty interval if $\mathsf{mw}(x) > \mathsf{ldr\_time}(r_x)$, while for vertex y it is

$$[\mathsf{mw}(y), \mathsf{ldr\_time}(r_y)],$$

or an empty interval if $\mathsf{mw}(y) > \mathsf{ldr\_time}(r_y)$

Subcase b) $r_x = r_y$. Since T^i is a subgraph of the minimum spanning tree T, we have that $\min(\mathsf{mw}(x), \mathsf{mw}(y))$ is the first time when at least one of x and y belongs to r_x's bag, while the first time when both belong to r_x's bag is $\max(\mathsf{mw}(x), \mathsf{mw}(y))$. Thus, the proper time interval for this edge:

$$[\min(\mathsf{mw}(x), \mathsf{mw}(y)), \max(\mathsf{mw}(x), \mathsf{mw}(y))] \cap [1, \ldots, \mathsf{ldr\_time}(r_x)]$$

We obtain that for every edge $e \in E$ all non-empty time intervals in which this edge belong to $\mathsf{nbr\_bag}$ of some vertex v can be computed in $O(\log(n))$ queries to the memory. Therefore, computing these values for all edges can be done in constant number of rounds assumed $O(m \log n)$ total memory. □

Implementing Observation B.4.3 is purely technical.

Lemma B.4.10. *There exist an AMPC algorithm that given a set of integer intervals* $I = \{[p_1, k_1], \ldots, [p_n, k_n]\}$, $\forall_{i \in [n]} [p_i, k_i] \subseteq [0, R]$ *finds the minimal number of intersecting intervals in* $O(1/\varepsilon)$ *rounds using* $O(n^{\varepsilon})$ *local memory and* $O(n \log^2 n)$ *total memory.*

Proof. First, the algorithm sorts the set $\{p_1, k_1, \ldots, p_n, k_n\}$ of all endpoints of these intervals in non-increasing order (ties are resolved with priority for endpoints p_i) obtaining a

sequence S. Consider assigning to every endpoint $p_i, i \in [n]$ from sequence S value $+1$ and to every endpoint $k_i, i \in [n]$ value -1. This operation leads to a sequence S' of pairs of format (endpoint, value). Finally, let S'' be a sequence constructed from S' in which all consecutive pairs that have the same first coordinate are compressed to a single pair in which the first coordinate is preserved and the second is the sum of second coordinates of contracted pairs. It can be observed that finding the minimal prefix sum of sequence made from second coordinates of pairs in S'' is equivalent to the minimal number of intersecting intervals. The construction of sequence S'' requires only sorting and contracting consecutive pairs which can be implemented in $O(1/\varepsilon)$ rounds in AMPC with the memory constrains stated in the lemma. To find the minimal prefix sum we use Theorem B.4.5 which completes the proof. □

The above discussion is summarized in the following lemma.

Lemma B.4.11. *There exists an AMPC algorithm that given a tuple (T, ℓ, E, L_i) calculates the value*

$$\min_{v \in L_i} \min_{0 \le t \le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t)$$

in $O(1/\varepsilon)$ rounds using $O(n^\varepsilon)$ local memory and $O((n+m)\log^2 n)$ total memory.

Proof. Using Lemma B.4.7 we are able to calculate value ldr_time for every $v \in L_i$ in constant number of rounds. By Lemma B.4.9 we can calculate time all non-empty time intervals for every $e \in E$ and every $v \in L_i$. This requires $O(m\log^2 n)$ total memory. Each time interval $[a,b]$ can be assigned a vertex v with respect to whom it was calculated. Then, we group time intervals with respect to vertices from L_i they were calculated. This can be done in a single round with $O(m\log^2 n)$ global memory since there are only $O(m)$ non-empty time intervals. Finally, Lemma B.4.10 guarantees that we can compute, for every $v \in L_i$, the minimum number of intersecting intervals in $O(1/\varepsilon)$ rounds with total memory

proportional to the number of these intervals. Therefore, assumed $O(m\log^2 n)$ global memory we can extend the last computation to a parallel computation for $v \in L_i$ while preserving the round complexity. By Observation B.4.3 this is equivalent to calculating

$$\min_{0\leq t\leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t),$$

for every $v \in L_i$. Since the minimum of the above values over $v \in L_i$ can be computed in a single round, the lemma is proven. □

B.4.4 The final algorithm.

We are now able to prove Theorem B.4.1 and present the final algorithm, SmallestSingletonCut, that calculates the smallest singleton cut that appears in the contraction process of G. The pseudcode can be found in Figure 6, while the proof of correctness is below.

Algorithm 6: SmallestSingletonCut

Data: Graph $G = (V,E,w : V \to [n^3])$.
Result: Size of the smallest singleton cut.
1 Compute the minimum spanning tree T of G;
2 Compute the low depth decomposition D_T of T;
3 Prepare $O(\log^2 n)$ tuples $(T,D_T,E,L_i), i \in \left[\lceil log^2 n\rceil\right]$;
4 **foreach** *tuple* (T,D_T,E,L_i) **do**
5 | Compute: $\mathsf{lc}_i \leftarrow \min_{v\in L_i} \min_{0\leq t\leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t)$;
6 **end**
7 **return** $\min(\mathsf{lc}_1,\ldots,\mathsf{lc}_{\left[\lceil log^2 n\rceil\right]})$;

Proof of Theorem B.4.1. The correctness follows from Observation B.4.1 and Lemmas B.4.3 and B.4.11. Also the implementation details of lines $3-7$ are discussed in the these two lemmas. To calculate minimum spanning tree in line 1 we use Lemma B.3.2 while the implementation of the low depth decomposition from the next line is given by Lemma B.3.1.

□

B.5 AMPC algorithm for approximated minimum k-cut

In this section, we show that given an algorithm that calculates $2+\varepsilon$ approximation of a min cut, one can construct $4+\varepsilon$ approximation of minimum k-cut. Consider the following greedy algorithm, called APX-SPLIT, that extends the classic result of Saran and Vazirani [129]. The algorithm works sequentially. In each iteration, it extends the approximated solution with the smallest non-trivial approximation of the minimum cut of the graph available at a given moment. Being precise, assume that at the beginning i-th iteration the algorithm has split the graph G into ℓ_i connected components (we start with $c_1 = 1$). Then the algorithm calculates $(2+\varepsilon)$-approximation of the minimum cut in each connected component and enlarges the solution by the smallest of these cuts, thereby increasing the number of components by at least 1. The algorithm ends after the first iteration such that the number of connected components after this iteration is at least k. The pseudocode of the algorithm is given in the Algorithm 7.

Algorithm 7: APX-SPLIT
(A greedy algorithm computing an approximation of the minimum k-cut in AMPC)

Data: A graph $G = (V(G), E(G))$ and a parameter k.
Result: A $(4+\varepsilon)$-approximation of minimum k-cut.

```
Initialize D → ∅;
while G' := (V(G), E(G) \ ⋃_{d∈D} d) has less than k connected components do
    Let C_1, ..., C_l be the set connected of components of G';
    d_i^* ← AMPC-MinCut(C_i) for all i ∈ [l];
    j ← arg min_{i∈[l]} weight(d_i);
    Add d_j^* to D;
end
return set of cuts D;
```

We will show by generalizing the idea of Saran and Vazirani that the aforementioned greedy algorithm is $(4+\varepsilon)$-approximation minimum k-cut.

Theorem B.1.2. Algorithm APX-SPLIT is an $(4+\varepsilon)$ approximation of the Min k-Cut. Furthermore, it can be implemented in the AMPC model with $O(n^{\varepsilon})$ memory per machine in $O(k \log\log n)$ rounds and $O(m)$ total memory.

We define $comps(c_1 \cup \ldots \cup c_k)$ the number of components of the graph G after removing all edges from the set $c_1 \cup \ldots \cup c_k$.

Proof. The standard line of proof, proposed in [129] for the case whe exact minimum cut is used at each splitting step, is to compare the cut selected by the APX-SPLIT algorithm to the approximated minimum k-cut obtained from the Gomory-Hu tree. The main difficulty is that in our case, we use *only* $(2+\varepsilon)$ approximation in each splitting step. This makes our proof different and novel compared to [129]. Let us set a Gomory-Hu tree $H = (V(H) = V(G), E(G))$ of the graph G. The Gomory-Hu tree is defined as follows.

Definition B.8 (Gomory and Hu [74])**.** Consider an arbitrary graph G. A weighted tree $H = (V(H), E(H))$ with the set of vertices being equal $V(G)$ is called a Gomory-Hu tree of G, if for every pair of different vertices $s,t \in V(G)$ the minimum weight on the path between s and t in the tree H is equal to the minimum s-t cut in graph G. The existence and construction of Gomory-Hu trees was shown in [74].

Let us order edges of the tree H (or equivalently cuts in the G graph) with respect to non-decreasing weights. Denote $b_1^*, \ldots, b_{l-1}^*$ the sequence of the first $l \leq k-1$ edges (cuts equivalently) from this order such that corresponding cuts split G graph into at least k connected components. Let $b_1, \ldots, b_{k-1}$ be a dual sequence of cuts corresponding to that sequence of edges $b_1^*, \ldots, b_{l-1}^*$, with this addition that we put each cut b_i this number of times it increases the number of connected components in G and possibly cut some suffix of such generated sequence to obtain exactly $k-1$ cuts. For such construction we have the following.

Observation B.5.1 (Saran and Vazirani [129]). *The sequence of cuts $b_1, \ldots, b_{k-1}$ satisfies:*

1. *the sequence $|b_1|, \ldots, |b_{k-1}|$ is non-decreasing,*
2. $\forall_{i \in [k-1]} comps(b_1 \cup \ldots \cup b_i) > i$.

Saran and Vazirani also proved that such selected (and possibly refactored) sequence of cuts is $(2 - \frac{2}{k})$-approximation of the minimum k-cut.

Theorem B.5.1 (Saran and Vazirani [129]). *The cut $\bigcup_{i \in [k-1]} b_i$ is $(2 - \frac{2}{k})$ approximation of the minimum k-cut of G.*

Having established the crucial properties of Gomory-Hu trees and corresponding cuts we can proceed to the proof of correctness of the APX-SPLIT algorithm.

Let $d_1, \ldots, d_m$ be the successive cuts selected by the APX-SPLIT algorithm. Note that with each new cut, at least one new component appears in the graph thus $m \leq k-1$. We will show that the sum of these cuts' sizes is not greater than the sum of sizes of cuts $b_1, \ldots, b_{k-1}$. Let $\#c_1, \ldots, \#c_m$ be a sequence of numbers where $\#c_i := comps(d_1 \cup \ldots \cup d_i)$. We will show by induction that

$$\forall_{j \in [m]} |d_1 \cup \ldots d_j| \leq (2+\varepsilon)|b_1 \cup \ldots \cup b_{\min(k-1, \#c_j)}|,$$

.

The idea behind the induction step defined in the previous line can be explained as follows: inclusion of cuts from d_1 to d_i are at least $(2+\varepsilon)$ approximation of cut generated by inclusion of cuts of such prefix of sequence $b_1, \ldots, b_k$ that split G on $\#c_i$ for connected components.

For the basis of induction we see that in the first step of the APX-SPLIT algorithm chooses $(2+\varepsilon)$-approximation of the smallest cut in the whole graph G. The b_1 is an s-t cut therefore we have $|d_1| \leq (2+\varepsilon)|b_1|$ which implies that $|d_1| \leq (2+\varepsilon)|b_1 \cup \ldots b_{\min(k-1, \#c_1)}|$.

Now consider $i \in [m-1]$. Since $i < m$, we observe that $\#c_i < k$. Otherwise the algorithm APX-SPLIT would have executed only $m-1$ iterations instead of m. Consider cuts $b_1, \ldots, b_{\#c_i+1}$. From the Observation B.5.1 we have that $comps(b_1 \cup \ldots \cup b_{\#c(i)+1}) > \#c(i)$. On the other hand $comps(d_1 \cup \ldots d_i) = \#c_i$. Since $comps(b_1 \cup \ldots \cup b_{\#c_i+1}) > comps(d_1 \cup \ldots d_i)$ then there must be a cut b_j, $j \in [\#c_i + 1]$ that is not covered by the first i cuts from the sequence $d_1, \ldots, d_m$. Namely, we can choose j such that $b_j \nsubseteq d_1 \cup \ldots \cup d_i$. Moreover, since b_j is an s-t cut in the graph G (with all edges included), thus it must split at least one connected component of the graph $G = (V, E \setminus (d_1 \cup \ldots d_i))$ into two non-empty parts. Thus this cut is considered in the $i+1$-th iteration of the APX-SPLIT algorithm, which implies that $|d_{i+1}| \leq (2+\varepsilon)|b_j| \leq (2+\varepsilon)|b_{\#c_i+1}|$. Since $\#c_i + 1 \leq \#c_{i+1}$, we conclude that $|d_1 \cup \ldots d_{i+1}| \leq (2+\varepsilon)|b_1 \cup \ldots \cup b_{\min(k-1, \#c_{i+1})}|$, which proves the inductive step. Now, we see from Theorem B.1.2 that the solution of $b_1, \ldots, b_{k-1}$ is an $(2 - \frac{2}{k})$-approximation of the minimum k-cut. Thus the solution $d_1 \ldots d_{k-1}$ is $(2+\varepsilon)(2-\frac{2}{k}) =$ approximation of the minimum k-cut. This proves the correctness of the algorithm.

It remains to be noted that a single iteration of the algorithm can be performed in $\mathrm{O}(\log \log n)$ rounds in the AMPC model with $\mathrm{O}(n^{\varepsilon})$ memory per machine and in total memory $\mathrm{O}(m)$. The dominant operation is the calculation of a $(2+\varepsilon)$ approximation of the minimum cut in each of the components. Its performance is analyzed in Theorem B.1.1. The calculation of the smallest of all approximated cuts corresponding to different components is a standard operation and can be performed in $O(1)$ rounds. Also, Behnezhad et. al in [30] showed that the number of components of a graph can be determined in $\mathrm{O}(1)$ rounds in AMPC with $\mathrm{O}(n^{\varepsilon})$ memory per machine and $\mathrm{O}(m)$ total memory. This completes the performance analysis of the algorithm. □

Appendix C: Tree Contraction

C.1 Introduction

In this paper, we study and extend Miller and Reif's fundamental FOCS'85 [113–115] $O(\log n)$-round parallel *tree contraction* method. Tree contraction is a process involving iterated contraction on graph components for efficient computation of problems on trees (see Section C.1.2). Their work leverages PRAM, a model of computation in which a large number of processors operate synchronously under a single clock and are able to randomly access a large shared memory. In PRAM, tree contractions require n processors. Though the initial study of tree contractions was in the CRCW (concurrent read from and write to shared memory) PRAM model, this was later extended to the stricter EREW (exclusive read from and write to shared memory) PRAM model [52] as well, and then to work-optimal parallel algorithms with $O(n/\log n)$ processors [63]. Since then, a number of additional works have also built on top of Miller and Reif's tree contraction algorithm [1, 47, 70]. Tree-based computations have a breadth of applications, including natural graph problems like matching and bisection on trees, as well as problems that can be formulated on tree-like structures including expression simplification.

The tree contraction method in particular is an extremely broad technique that can be applied to many problems on trees. Miller and Reif [114] initially motivated their work by showing it can be used to evaluate arithmetic expressions. They additionally studied a number of other applications [115], using tree contractions to construct the first poly-

logarithmic round algorithm for tree isomorphism and maximal subtree isomorphism of unbounded degrees, compute the 3-connected components of a graph, find planar embeddings of graphs, and compute list-rankings. An incredible amount of research has been conducted to further extend the use of tree contractions for online evaluation of arithmetic circuits [112], finding planar graph separators [64], approximating treewidth [33], and much more [14, 75, 77, 89, 111, 120]. This work extends classic tree contractions to the adaptive massively parallel setting.

The importance of large-scale data processing has spurred a large interest in the study of massively parallel computing in recent years. Notably, the *Massively Parallel Computation* (MPC) model has been studied extensively in the theory community for a range of applications [4, 6–8, 11–13, 18, 20, 27, 28, 31, 35, 49, 65, 79, 83, 101, 118, 128, 136], many with a particular focus on graph problems. MPC is famous for being an abstraction of MapReduce [95], a popular and practical programming framework that has influenced other parallel frameworks including Spark [137], Hadoop [135], and Flume [40]. At a high level, in MPC, data is distributed across a range of low-memory machines which execute local computations in rounds. At the end of each round, machines are allowed to communicate using messages that do not exceed their local space constraints. In the most challenging space-constrained version of MPC, we restrict machines to $O(n^{\varepsilon})$ local space for a constant $0 < \varepsilon < 1$ and $\widetilde{O}(n+m)$ total space (for graphs with m edges, or just $\widetilde{O}(n)$ otherwise).

The computation bottleneck in practical implementations of massively parallel algorithms is often the amount of communication. Thus, work in MPC often focuses on *round complexity*, or the number of rounds, which should be $O(\log n)$ at a baseline. More ambitious research often strives for sublogarithmic or even constant round complexity, though this often requires very careful methods. Among others, a specific family of graph problems known as *Locally Checkable Labeling* (LCL) problems – which includes vertex col-

oring, 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 [11, 12, 24, 31, 49, 66, 69]. Another consists of DP problems on sequences including edit distance [35] and longest common subsequence [81], as well as pattern matching [80]. The round complexity of aforementioned MPC algorithms can be interpreted as the parallelization limit of the corresponding problems.

While MPC is generally an extremely efficient model, it is theoretically limited by the widely believed 1-vs-2Cycle conjecture [67], which poses that distinguishing between a graph that is a single n-cycle and a graph that is two $n/2$-cycles requires $\Omega(\log n)$ rounds in the low-memory MPC model. This has been shown to imply lower bounds on MPC round complexity for a number of other problems, including connectivity [28], matching [67, 118], clustering [136], and more [8, 67, 101]. To combat these conjectured bounds, Behnezhad et al. [29] developed a stronger and practically-motivated extension of MPC, called *Adaptive Massively Parallel Computing* (AMPC). AMPC was inspired by two results showing that adding distributed hash tables to the MPC model yields more efficient algorithms for finding connected components [97] and creating hierarchical clusterings [18]. AMPC models exactly this: it builds on top of MPC by allowing in-round access to a distributed read-only hash table of size $O(n+m)$. See Section C.1.1 for a formal definition.

In their foundational work, Behnezhad et al. [29] design AMPC algorithms that outperform the MPC state-of-the-art on a number of problems. This includes solving minimum spanning tree and 2-edge connectivity in $\log\log_{m/n}(n)$ AMPC rounds (outperforming $O(\log n)$ and $O(\log D \log\log_{m/n} n)$ MPC rounds respectively), and solving maximal independent set, 2-Cycle, and forest connectivity in $O(1)$ AMPC rounds (outperforming $\widetilde{O}(\sqrt{\log n})$, $O(\log n)$, and $O(\log D \log\log_{m/n} n)$ MPC rounds respectively). Perhaps most notably, however, they proved that the 1-vs-2Cycle conjecture does not apply to AMPC by finding an algorithm to solve connectivity in $O(\log\log_{m/n} n)$ rounds. This was later

improved to be $O(1/\varepsilon)$ by Behnezhad et al. [30], who additionally found improved algorithms for AMPC minimum spanning forest and maximum matching. Charikar, Ma, and Tan [42] recently show that connectivity in the AMPC model requires $\Omega(1/\varepsilon)$ rounds unconditionally, and thus the connectivity result of Behnezhad et al. [30] is indeed tight. In a subsequent work, Behnezhad [23] shows an $O(1/\varepsilon)$-round algorithm for the maximal matching problem in AMPC.

A notable drawback of the current work in AMPC is that there is no generalized framework for solving multiple problems of a certain class. Such methods are important for providing a deeper understanding of how the strength of AMPC can be leveraged to beat MPC in general problems, and often leads to solutions for entirely different problems. Studying Miller and Reif [114]'s tree contraction algorithm in the context of AMPC provides exactly this benefit. We get a generalized technique for solving problems on trees, which can be extended to a range of applications.

Recently, Bateni et al. [19] introduced a generalized method for solving "polylog-expressible" and "linear-expressible" dynamic programs on trees in the MPC model. This was heavily inspired by tree contractions, and also is a significant inspiration to our work. Specifically, their method solves minimum bisection, minimum k-spanning tree, maximum weighted matching, and a large number of other problems in $O(\log n)$ rounds. We extend these methods, as well as the original tree contraction methods, to the AMPC model to create more general techniques that solve many problems in $O_\varepsilon(1)$ rounds.

C.1.1 The AMPC Model

The AMPC model, introduced by Behnezhad et. al [29], is an extension of the standard MPC model with additional access to a *distributed hash table*. In MPC, data is initially distributed across machines and then computation proceeds in rounds where machines ex-

ecute local computations and then are able to share small messages with each other before the next round of computation. A distributed hash table stores a collection of key-value pairs which are accessible from every machine, and it is required that both key and value have a constant size. Each machine can adaptively query a bounded sequence of keys from a centralized distributed hash table during each round, and write a bounded number of key-value pairs to a distinct distributed hash table which is accessible to all machines in the next round. The distributed hash tables can also be utilized as the means of communication between the machines, which is implicitly handled in the MPC model, as well as a place to store the initial input of the problem. It is straight-forward to see how every MPC algorithm can be implemented within the same guarantees for the round-complexity and memory requirements in the AMPC model.

Definition C.1. Consider a given graph on n vertices and m edges. In the **AMPC model**, there are M machines each with **sublinear** local space $\mathsf{S} = O(n^{\varepsilon})$ for some constant $0 < \varepsilon < 1$, and the total memory of machines is bounded by $\widetilde{O}(n+m)$. In addition, there exist a collection of distributed hash tables $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2, \ldots$, where H_0 contains the initial input.

The process consists of several rounds. During round i, each machine is allowed to make at most $O(\mathsf{S})$ read queries from H_{i-1} and to write at most $O(\mathsf{S})$ key-value pairs to H_i. Meanwhile, the machines are allowed to perform an arbitrary amount of computation locally. Therefore, it is possible for machines to decide what to query next after observing the result of previous queries. In this sense, the queries in this model are **adaptive**.

C.1.2 Our Contributions

The goal of this paper is to present a framework for solving various problems on trees with constant-round algorithms in AMPC. This is a general strategy, where we intelligently shrink the tree iteratively via a *decomposition* and *contraction* process. Specifically, we

follow Miller and Reif's [114] two-stage process, where we first *compress* each connected component in our decomposition, and then *rake* the leaves by contracting all leaves of the same parent together. We repeat until we are left with a single vertex, from which we can extract a solution. To retrieve the solution when the output corresponds to many vertices in the tree (i.e., maximum matching istead of maximum matching value), we can undo the contractions in reverse order and populate the output as we gradually reconstruct the original tree.

The decomposition strategy must be constructed very carefully such that we do not lose too much information to solve the original problem and each connected component must fit on a single machine with $O(n^{\varepsilon})$ local memory. To compress, we require oracle access to a black-box function, a *connected contracting function*, which can efficiently contract a connected component into a vertex while also retaining enough information to solve the original problem. To rake leaves, we require oracle access to another block-box function, a *sibling contracting function*, which executes the same thing but on a set of leaves that share a parent. These two black-box functions are problem specific (e.g., we need a different set of functions for maximum matching and maximum independent set). In this paper,

we only require contracting functions to accept n^{ε} vertices as the input subgraphs, and we always run these black-box functions locally on a single machine. Thus, we can compress any arbitrary collection of disjoint components of size at most n^{ε} in $O(1)$ AMPC rounds. See Section C.2.1 for formal definitions.

This general strategy actually works on a special class of structures, called *degree-weighted trees* (defined in §C.2). Effectively, these are trees $T = (V, E, W)$ with a multi-dimensional weight function where $W(v) \in \{0,1\}^{\widetilde{O}(\deg(v))}$ stores a vector of bits proportional in size to the degree of the vertex $v \in V$. When we use our contracting functions, we use W to store data about the set of vertices we are contracting. This is what allows our algorithms to retain enough information to construct a solution to the entire tree T when

we contract sets of vertices. Note that the degree of the surviving vertex after contraction could be much smaller than the total degree of the original set of vertices.

Our first algorithm works on trees with bounded degree, more precisely, trees with maximum degree at most n^ε. The reason this is easier is because when an internal connected component is contracted, we often need to encode the output of the subproblem at the root (e.g., the maximum weighted matching on the rooted subtree) in terms of the children of this component post-contraction. In high degree graphs, it may have many children after being contracted, and therefore require a large encoding (i.e., one larger than $O(n^\varepsilon)$) and thus not fit on one machine.

In this algorithm, we find that if the degree is bounded by n^ε and we compress sufficiently small components, then the algorithm works out much more smoothly. The underlying technique that allows us to contract the tree into a single vertex in $O(1/\varepsilon)$ iterations is a decomposition of vertices based on their preorder numbering. The surprising fact is that each group in this decomposition contains at most one non-leaf vertex after contracting connected components. Thus, an additional single rake stage is sufficient to collapse any tree with n vertices to a tree with at most $n^{1-\varepsilon}$ vertices in a single iteration. However, we need $O(1/\varepsilon)$ AMPC rounds at the beginning of each iteration to find the decomposition associated with the resulting tree after contractions performed in the previous iteration. This becomes $O(1/\varepsilon^2)$ AMPC rounds across all iterations. See Section C.3.1 for the proofs and more details.

This is a nice independent result, proving a slightly more efficient $O(1/\varepsilon^2)$-round algorithm on degree bounded trees. Additionally, many problems on larger degree trees can be represented by lower degree graphs. For example, both the original Miller and Reif [113] tree contraction and the Betani et al. [19] framework consider only problems in which we can replace each high degree vertex by a balanced binary tree, reducing the tree-based computation on general trees to a slightly different computation on binary trees. Equally

notably, it is an important subroutine in our main algorithm.

Theorem C.1.1. *Consider a degree-weighted tree $T = (V, E, W)$ and a problem P. Given a connected contracting function on T with respect to P, one can compute $P(T)$ in $O(1/\varepsilon^2)$ AMPC rounds with $O(n^\varepsilon)$ memory per machine and $\widetilde{O}(n)$ total memory if $\deg(v) \leq n^\varepsilon$ for every vertex $v \in V$.*

Remark. It may be tempting to suggest that in most natural problems the input tree can be transformed into a tree with degree bounded by n^ε. However, we briefly pose the MedianParent problem, where leaves are given values and parents are defined recursively as the median of their children. By transforming the tree to make it degree bounded, we lose necessary information to find the median value among the children of a high degree vertex.

Next, we move onto our main result: a generalized tree contraction algorithm that works on any input tree with arbitrary structure. Building on top of Theorem C.1.1, we can create a natural extension of tree contractions. Recall that the black-box contracting functions encode the data associated with a contracted vertex in terms of its children post-contraction. Thus, allowing high degree vertices introduces difficulties working with contracting functions. In particular, it is not possible to store the weight vector $W(v)$ of a high degree vertex v inside the local memory of a single machine. The power of this algorithm is its ability to implement COMPRESS and RAKE for n^ε-tree-contractions in $O(1/\varepsilon^3)$ rounds.

The most significant novelty of our main algorithm is the handling of high degree vertices. To do this, we first handle all maximal connected components of low degree vertices using the algorithm from Theorem C.1.1 as a black-box. This compresses each such component into one vertex without needing to handle high degree vertices. By contracting these components, we obtain a special tree called *Big-Small-tree* (defined formally in §C.3.2) which exhibits nice structural properties. Since the low degree components are maximal, the degree of each vertex in every other layer is at least n^ε, implying that a large fraction of

the vertices in a Big-Small-tree must be leaves. Hence, after a single rake stage, the number of high degree vertices drops by a factor of n^{ε}.

In order to rake the leaves of high degree vertices, we have to carefully apply our sibling contracting functions in a way that can be implemented efficiently in AMPC. Unlike Theorem C.1.1 in which having access to a connected contracting function is sufficient, here we also require a sibling contracting function. Consider a star tree with its center at the root. Without a sibling contracting function, we are able to contract at most $O(n^{\varepsilon})$ vertices in each round since the components we pass to the contracting functions must be disjoint. But having access to a sibling contracting function, we can rake up to $O(n)$ leaf children of a high degree vertex in $O(1/\varepsilon)$ rounds. For more details about the algorithm and proofs see Section C.3.2.

Theorem C.1.2. *Consider a degree-weighted tree $T = (V,E,W)$ and a problem P. Given a connected contracting function and a sibling contracting function on T with respect to P, one can compute $P(T)$ in $O(1/\varepsilon^3)$ AMPC rounds with $O(n^{\varepsilon})$ memory per machine and $\widetilde{O}(n)$ total memory.*

Theorem C.1.1 and Theorem C.1.2 give us general tools that have the power to create efficient AMPC algorithms for any problem that admits a connected contracting function and a sibling contracting function. Intuitively, they reduce constant-round parallel algorithms for a specific problem on trees to designing black-box contracting functions that are sequential. We should be careful in designing contracting functions to make sure that the amount of data stored in the surviving vertex does not asymptotically exceed its degree in the contracted tree. Also note that a connected contracting function works with unknown values that depend on the result of other components.

Satisfying these conditions is a factor that limits the extent of problems that can be solved using our framework. For example, the framework of Bateni et. al [19] works

on a wider range of problems on trees since their algorithm, roughly speaking, tolerates exponential growth of weight vectors using a careful decomposition of tree. Indeed, they achieve these benefits at the cost of an inherent requirement for at least $O(\log n)$ rounds due to the divide-and-conquer nature of their algorithm. However, their framework comes short on addressing problems such as MedianParent (defined in Remark) that are not reducible to binary trees. Nonetheless, we show several techniques for designing contracting functions that satisfy these conditions, in particular:

1. In Section C.3.3, we prove a general approach for designing a connected contracting function and a sibling contracting function given a PRAM algorithm based on the original Miller and Reif [113] tree contraction. We do this by observing that in almost every conventional application of Miller and Reif's framework, the length of data stored at each vertex remains constant throughout the algorithm.

2. Storing a minimal tree representation of a connected component contracted into v in the weight vector $W(v)$ enables us to simplify a recursive function defined on the subtree rooted at v in terms of yet-unknown values of its children, while keeping the length of $W(v)$ asymptotically proportional to $deg(v)$. For instance, our maximum weighted matching algorithm (See Section 4.1 of the full version for more details) uses this approach.

Ultimately, this is a highly efficient generalization of the powerful tree contraction algorithm. To illustrate the versatility of our framework, we show that it gives us efficient AMPC algorithms for many important applications of frameworks such as Miller and Reif [114]'s and Bateni et al. [19]'s by constructing sequential black-box contracting functions. In doing so, we utilize a diverse set of techniques, including the ones mentioned above, that are of independent interest and can be applied to a broad range of problems on

trees. The proof Theorem C.1.3 and more details about each application can be found in the full version.

Theorem C.1.3. *Algorithms 8 and 9 can solve, among other applications, dynamic expression evaluation, tree isomorphism testing, maximal matching, and maximal independent set in $O(1/\varepsilon^2)$ AMPC rounds, and maximum weighted matching and maximum weighted independent set in $O(1/\varepsilon^3)$ AMPC rounds. All algorithms use $O(n^\varepsilon)$ memory per machine and $\widetilde{O}(n)$ total memory.*

C.1.3 Paper Outline

The work presented in this paper is a constant-round generalized technique for solving a large number of graph theoretic problems on trees in the AMPC model. In Section C.2, we go over some notable definitions and conventions we will be using throughout the paper. This includes the introduction of a generalized weighted tree, a formalization of the general tree contraction process, the definition of contracting functions, and a discussion of a tree decomposition method we call the *preorder decomposition*. In the Section C.3, we go over our main results, algorithms, and proofs. The first result (§C.3.1) is an algorithm for executing a tree contraction-like process which solves the same problems on trees of bounded maximum degree. The second result (§C.3.2) utilizes the first result as well as additional novel techniques to implement generalized tree contractions. We additionally show (§C.3.3) that our algorithms can also implement Miller and Reif's standard notion of tree contractions, and (§C.3.4) we show how to efficiently reconstruct a solution on the entire graph by reversing the tree contracting process.

C.2 Preliminaries

In this work, we are interested in solving problems on trees $T = (V, E)$ where $|V| = n$. Our algorithms iteratively transform T by contracting components in an intelligent way that: (1) components can be stored on a single machine, (2) the number of iterations required to contract T to a single vertex is small, and (3) at each step of the process, we still have enough information to solve the initial problem on T.

To achieve (3), we must retain some information about an original component after we contract it. For instance, consider computing all maximal subtree sizes. For a connected component S with $r = \text{lca}(S)$[1], the contracted vertex v_S of S might encode $|S|$ and a list of its leaves (when viewing S as a tree itself). It is not difficult to see that this would be sufficient knowledge to compute all maximal subtree sizes for the rest of the vertices in T without considering all individual vertices in S. Data such as this is encoded as a multi-dimensional weight function which maps vertices to binary vectors. We will specifically consider trees where the dimensionality of the weight function is bounded by the degree of the vertex.

We note that in this paper, when we refer to the degree of a vertex in a rooted tree, we ignore parents. Therefore, $\deg(v)$ is the number of children a vertex has.

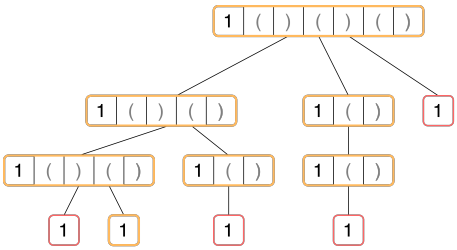
Definition C.2. A **degree-weighted tree** is a tree $T = (V, E, W)$ with vertex set V, edge set E, and vertex weight vector function W such that for all $v \in V$, $W(v) \in \{0,1\}^{\widetilde{O}(\deg(v))}$.[2]

Notationally, we let $w(v) = \dim(W(v)) = \widetilde{O}(\deg(v))$ be the length of the weight vectors. Additionally, note that a tree $T = (V, E)$ is a degree-weighted tree where $W(v) = \emptyset$ for all $v \in V$.

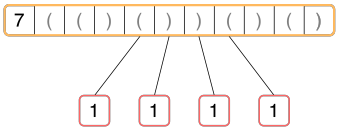
In order to implement our algorithm, we also require specific *contracting functions* whose properties allow us to achieve the desired result (§C.2.1). In addition, we will intro-

[1] lca is the least common ancestor function.

[2] $\widetilde{O}(f(n)) = O(f(n)\log n)$.



(a) Each vertex in the degree weighted tree T stores the size of its subtree, which is 1 initially and the structure of the subtree between each vertex and its children using parenthesis notation which is simply a star for every vertex at the beginning. In the parenthesis notation, we traverse the tree according to the preorder numbering and put an '(' whenever we go down from a parent to a child, and a ')' whenever we go up from a child to a parent.



(b) In the contracted degree weighted tree T', the structure of the yellow subgraph is recorded in the weight vector of the root, a tree with 4 leaves (equal to the degree of root) which is not a star. In addition, the size of the contracted subgraph is stored in the weight vector of the root.

Figure C.1: A degree weighted tree $T = (V, E, W)$ with $|V| = 11$. In Subfigure C.1a, we have a degree weighted tree with $|W_v| \leq 4deg(v)$. We contract the subgraph with 7 vertices depicted by yellow in Subfigure C.1a using a connected contracting function (Defined in Definition C.4). The resulting degree weighted tree T' is depicted in Subfigure C.1b. Note that the length of weight vectors in proportional to the degree of each vertex even after the contraction.

duce a specific tree decomposition method, called a *preorder decomposition*, that we will efficiently implement and leverage in our final algorithms (§C.2.2).

C.2.1 Tree Contractions and Contracting Functions

Our algorithms provide highly efficient generalizations to Miller and Reif's [114] tree contraction algorithms. At a high level, their framework provides the means to compute a global property with respect to a given tree in $O(\log n)$ *phases*. In each *phase*, there are two *stages*:

- COMPRESS stage: Contract around half of the vertices with degree 1 into their parent.
- RAKE stage: Contract all the leaves (vertices with degree 0) into their parent.

Repeated application of COMPRESS and RAKE alternatively results in a tree which has only one vertex. Intuitively, the COMPRESS stage aims to shorten the long *chains*, maximal connected sequences of vertices whose degree is equal to 1, and the RAKE stage cleans up the leaves. Both stages are necessary in order to guarantee that $O(\log n)$ phases are enough to end up with a single remaining vertex [114].

In the original variant, every odd-indexed vertex of each *chain* is contracted in a COMPRESS stage. In some randomized variants, each vertex is selected with probability $1/2$ independently, and an independent set of the selected vertices is contracted. In such variants, contracting two consecutive vertices in a chain is avoided in order to efficiently implement the tree contraction in the PRAM model. However, this restriction is not imposed in the AMPC model, and hence we consider a more *relaxed variant* of the COMPRESS stage where each maximal chain is contracted into a single vertex.

We introduce a more generalized version of tree contraction called α*-tree-contractions*. Here, the RAKE stage is the same as before, but in the COMPRESS stage, every maximal subgraph containing only vertices with degree less than α is contracted into a single vertex.

Definition C.3. In an α-tree-contraction of a tree $T = (V,E)$, we repeat two stages in a number of phases until the whole tree is contracted into a single vertex:

- COMPRESS stage: Contract every maximal connected component S containing only vertices with degree less than α, i.e., $deg(v) < \alpha \ \ \forall v \in S$, into a single vertex S'.
- RAKE stage: Contract all the leaves into their parent.

Notice that the relaxed variant of Miller and Reif's COMPRESS stage is the special case when $\alpha = 2$. Our goal will be to implement efficient α-tree-contractions where $\alpha = n^{\varepsilon}$.

In order to implement COMPRESS and RAKE, we need fundamental tools for contracting a single set of vertices into each other. We call these *contracting functions*. In the

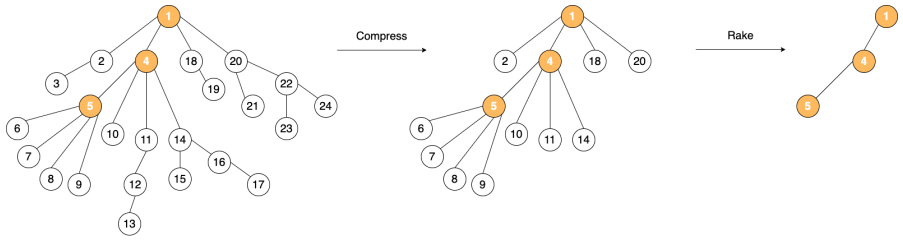


Figure C.2: An example phase in α-tree-contraction for $\alpha = 4$. In the leftmost tree the initial tree is depicted, and the vertices are numbered from 1 to n in the preorder ordering. In the middle tree, we performed a COMPRESS stage to get a tree with 12 vertices. Next, we RAKE the leaves to end up with a tree with 3 vertices depicted on the right.

COMPRESS stage, we must contract connected components. In the RAKE stage, we must contract leaves with the same parent into a single vertex. These functions run locally on small sets of vertices.

Definition C.4. Let P be some problem on degree-weighted trees such that for some degree-weighted tree T, $P(T)$ is the solution to the problem on T. A **contracting function** on T with respect to P is a function f that replaces a set of vertices in T with a single vertex and incident edges to form a degree-weighted tree T' such that $P(T) = P(T')$[3]. There are two types:

1. f is a **connected contracting function** if f contracts[4] connected components into a single vertex of T.

2. f is a **sibling contracting function** if f is defined on sets of leaf siblings (i.e., leaves that share a parent p) of T, and the new vertex is a leaf child of p.

Since the output of the contracting function is a degree-weighted tree, it implicitly must create a weight $W(v)$ for any newly contracted vertex v.

[3]With some nuance, it depends on the format of the problem. For instance, when computing the value of the maximum independent set, the single values $P(T)$ and $P(T')$ should be the same. When computing the maximum independent set itself, uncontracted vertices must have the same membership in the set, and contracted vertices represent their roots.

[4]Consider a connected component S with a set of external neighbors $N(S) = \{v \in V \setminus S : \exists u \in S(v,u) \in E\}$. Then contracting S means replacing S with a single vertex with neighborhood $N(S)$.

C.2.2 Preorder Decomposition

A *preorder decomposition* (formally defined shortly) is a strategy for decomposing trees into a disjoint union of (possibly not connected) vertex groups. In this paper, we will show that the preorder decomposition exhibits a number of nice properties (see §C.3) that will be necessary for our tree contraction algorithms. Ultimately, we wish to find a decomposition of vertices $V_1, V_2, \ldots, V_k \subseteq V$ of a given tree $T = (V, E)$ ($\cup_{i=1}^{k} V_i = V$ and $V_i \cap V_j = \phi \ \ \forall i, j : i \neq j$) so that for all $i \in [k]$, after contracting each connected component contained in the same vertex group, the maximum degree is bounded by some given λ. Obviously, this won't be generally possible (i.e., consider a large star), but we will show that this holds when the maximum degree of the input tree is bounded as well.

The preorder decomposition is depicted in Figure C.3a. Number the vertices by their index in the preorder traversal of tree T, i.e., vertices are numbered $1, 2, \ldots, n$ where vertex i is the i-th vertex that is visited in the preorder traversal starting from vertex 1 as root. In a preorder decomposition of T, each group V_i consists of a consecutive set of vertices in the preorder numbering of the vertices. More precisely, let l_i denote the index of the vertex $v \in V_i$ with the largest index, and assume $l_0 = 0$ for consistency. In a preorder decomposition, group V_i consists of vertices $l_{i-1} + 1, l_{i-1} + 2, \ldots, l_i$.

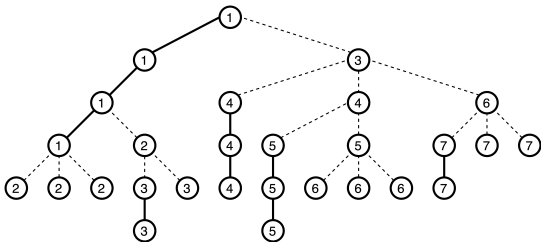
Definition C.5. Given a tree $T = (V, E)$, a *"preorder decomposition"* $V_1, V_2, \ldots, V_k$ of T is defined by a vector $l \in \mathbb{Z}^{k+1}$, such that $0 = l_0 < l_1 < \ldots < l_k = n$, as $V_i = \{l_{i-1} + 1, l_{i-1} + 2, \ldots, l_i\} \ \ \forall i \in [k]$. See Subfigure C.3a for an example.

Assume we want each V_i in our preorder decomposition to satisfy $\sum_{v \in V_i} \deg(v) \leq \lambda$ for some λ. As long as $\deg(v) \leq \lambda$ for all $v \in V$, we can greedily construct components $V_1, \ldots, V_k$ according to the preorder traversal, only stopping when the next vertex violates the constraint. Since $\sum_{v \in V} \deg(v) \leq n$, it is not hard to see that this will result in $O(n/\lambda)$ groups that satisfy the degree sum constraint.

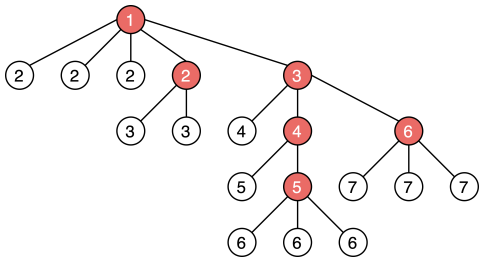
Observation C.2.1. *Consider a given tree $T = (V,E)$. For any parameter λ such that $\deg(v) \leq \lambda$ for all $v \in V$, there is a preorder decomposition $V_1, V_2, \ldots, V_k$ such that $\forall i \in [k]$, $\sum_{v \in V_i} \deg(v) \leq \lambda$, and $k = O(n/\lambda)$.*

The *dependency tree* $T' = (V', E')$, as seen in Figure C.3b of a decomposition is useful notion for understanding the structure of the resulting graph. In T', vertices represent connected components within groups, and there is an edge between vertices if one contains a vertex that is a parent of a vertex in the other. This represents our contraction process and will be useful for bounding the size of the graph after each step.

Definition C.6. Given a tree $T = (V,E)$ and a decomposition of vertices $V_1, V_2, \ldots, V_k$, the **dependency tree** $T' = (V', E')$ of T under this decomposition is constructed by contracting each connected component $C_{i,j}$ for all $j \in [c_i]$ in each group V_i. We call a component contracted to a leaf in T' an **independent** component, and a component contracted to a non-leaf vertex in T' a **dependent** component.



(a) An example preorder decomposition of T into $V_1, V_2, \ldots, V_7$ with $\lambda = 8$. Edges within any F_i are depicted bold, and edges belonging to no F_i are depicted dashed.



(b) Dependency tree T', created by contracting connected components of every F_i. Each red vertex represents a dependent component, and each white vertex represents an independent component.

Figure C.3: In Subfigure (a), a preorder decomposition of a given tree T is demonstrated. Based on this preorder decomposition, we define a dependency tree T' so that each connected component S in each forest F_i is contracted into a single vertex S'. This dependency tree T' is demonstrated in Subfigure (b). It is easy to observe that the contracted components are maximal components which are connected using bold edges in T, and each edge in T' corresponds to a dashed edge in T.

C.3 Constant-round Tree Contractions in AMPC

The main results of this paper are two new algorithms. The first algorithm applies α-tree-contraction-*like* methods in order to solve problems on trees where the degrees are bounded by n^ε. Though this algorithm is similar in inspiration to the notion of tree contractions, it is not a true α-tree-contraction method.

Theorem C.1.1. *Consider a degree-weighted tree $T = (V,E,W)$ and a problem P. Given a connected contracting function on T with respect to P, one can compute $P(T)$ in $O(1/\varepsilon^2)$ AMPC rounds with $O(n^\varepsilon)$ memory per machine and $\widetilde{O}(n)$ total memory if $\deg(v) \leq n^\varepsilon$ for every vertex $v \in V$.*

This algorithm provides us with two benefits: (1) it is a standalone result that is quite powerful in its own right and (2) it is leveraged in our main algorithm for Theorem C.1.2. The only differences between this result and our main result for generalized tree contractions is that we require $\deg(v) \leq n^\varepsilon$, but it runs in $O(1/\varepsilon^2)$ rounds, as opposed to $O(1/\varepsilon^3)$ rounds. Thus, if the input tree has degree bounded by n^ε, then clearly the precondition is satisfied. Additionally, if the tree can be decomposed into a tree with bounded degree such that we can still solve the problem on the decomposed tree, this result applies as well.

Our general results are quite similar, with a slightly worse round complexity, but with the ability to solve the problem on all trees. Notably, it is a true α-tree-contraction algorithm.

Theorem C.1.2. *Consider a degree-weighted tree $T = (V,E,W)$ and a problem P. Given a connected contracting function and a sibling contracting function on T with respect to P, one can compute $P(T)$ in $O(1/\varepsilon^3)$ AMPC rounds with $O(n^\varepsilon)$ memory per machine and $\widetilde{O}(n)$ total memory.*

In this section, we introduce both algorithms and prove both theorems.

C.3.1 Contractions on Degree-Bounded Trees

We now provide an $O(1/\varepsilon^2)$-round AMPC algorithm with local space $O(n^\varepsilon)$ for solving any problem P on a degree-weighted tree $T = (V, E, W)$ with bounded degree $\deg(v) \leq n^\varepsilon$ for all $v \in V$ given a *connected contracting function* for P. The method, which we call BoundedTreeContract, can be seen in Algorithm 8.

Much like an α-tree-contraction algorithm, it can be divided into a COMPRESS and RAKE stage. In the COMPRESS stage, instead of compressing the whole maximal components that consist of low-degree vertices as required for α-tree-contractions, we partition the vertices into groups using a preorder decomposition and bounding the group size by n^ε. In the RAKE stage, since the degree is bounded by n^ε, all leaves who are children of the same vertex can fit on one machine. Thus each sibling contraction that must occur can be computed entirely locally. If we include the parent of the siblings, we can simply apply COMPRESS's connected contracting function on the children. This is why we do not need a sibling contracting function.

Let $T_0 = T$ be the input tree. For every iteration $i \in [O(1/\varepsilon)]$: (1) find a preorder decomposition $V_1, \ldots V_k$ of T_{i-1}, (2) contract each connected component in the preorder decomposition, and (3) put each maximal set of leaf-siblings (i.e., leaves that share a parent) in one machine and contract them into their parent. We sometimes refer to these maximal sets of leaf-siblings by *leaf-stars*. After sufficiently many iterations, this should reduce the problem to a single vertex, and we can simply solve the problem on the vertex.

Notice that we can view the first and second steps as the COMPRESS stage except that we limit each component such that the sum of the degrees in each component is at most n^ε. Since the size of the vector $W(v)$ is $w(v) = \widetilde{O}(\deg(v)) = \widetilde{O}(n^\varepsilon)$, we can store an entire component (in its current, compressed state) in a single machine, thus making the second step distributable. The third step can be viewed as a RAKE function which, as we stated,

Algorithm 8: BoundedTreeContract
(Computing the solution $P(T)$ of a problem P on degree-weighted tree T with max degree n^ε using connected contracting function $\mathscr{C}$)

Data: Degree-weighted tree $T = (V, E, W)$ with degree bounded by n^ε and a connected contracting function $\mathscr{C}$.
Result: The problem output $P(T)$.

$T_0 \leftarrow T$;
for $i \leftarrow 1$ **to** $l = O(1/\varepsilon)$ **do**
  Let $\mathscr{O}$ be a preordering of V_{i-1};
  Find a *preorder decomposition* $V_1, V_2, \ldots, V_k$ of T_{i-1} with $\lambda = n^\varepsilon$ using $\mathscr{O}$;
  Let $\mathscr{S}_{i-1}$ be the set of all connected components in V_i for all $i \in [k]$;
  Let T'_{i-1} be the result of contracting $\mathscr{C}(S_{i-1,j})$ for all $S_{i-1,j} \in \mathscr{S}_{i-1}$;
  Let $\mathscr{L}_{i-1}$ be the set of all maximal leaf-stars (containing their parent) in T'_{i-1};
  Let T_i be the result of contracting $\mathscr{C}(L_{i-1,j})$ for all $L_{i-1,j} \in \mathscr{L}_{i-1}$;
end
return $\mathscr{C}(T_l)$;

can be handled on one machine per contraction using the connected contracting function.

In order to get $O(1/\varepsilon^2)$ rounds, we first would like to show that the number of phases is bounded by $O(1/\varepsilon)$. To prove this, we show that there will be at most one non-leaf node after we contract the components in each group. In other words, the dependency tree resulting from the preorder decomposition has at most one non-leaf node per group in the decomposition. This is a necessary property of decomposing the tree based off the preorder traversal. To see why this is true, consider a connected component in a partition. If it is *not* the last connected component (i.e., it does not contain the partition's last vertex according to the preorder numbering), then after contracting, it cannot have any children.

Lemma C.3.1. *The dependency tree $T' = (V', E')$ of a preorder decomposition $V_1, V_2, \ldots, V_k$ of tree $T = (V, E)$ contains at most 1 non-leaf vertex per group for a total of at most k non-leaf vertices. In other words, there are at most k dependent connected components in $\cup_{i \in [k]} F_i$.*

Proof. Each group V_i induces a forest F_i on tree T, and recall that each F_i is consisted

of multiple connected components $C_{i,1}, C_{i,2}, \ldots, C_{i,c_i}$, where c_i is the number of connected components of F_i. Assume w.l.o.g. component C_{i,c_i} is the component which contains vertex l_i, the vertex with the largest index in V_i. We show that every connected component in F_i except C_{i,c_i} is independent, and thus Lemma C.3.1 statement is implied. See in Subfigure C.3b that there is at most 1 dependent component, red vertices in T', for each group V_i. Also note that C_{i,c_i}, the only possibly dependent component in F_i, is always the last component if we sort the components based on their starting index since $l_i \in C_{i,c_i}$ and each $C_{i,j}$ contains a consecutive set of vertices.

Assume for contradiction that there exists $C_{i,j}$ for some $i \in [k], j \in [c_i - 1]$, a non-last component in group V_i, such that $C_{i,j}$ is a dependent component, or equivalently $C'_{i,j}$ is not a leaf in T'. Since $C_{i,j}$ is a dependent component, there is a vertex $v \in C_{i,j}$ which has a child outside of V_i. Let u be the first such child of v in the pre-order traversal, and thus $u \in V_j$ for some $j > i$. Consider a vertex $w \in V_i$ that comes after v in the pre-order traversal. Then, since u, and thus V_j, comes after v and V_i in the pre-order traversal, u must come after w in the pre-order traversal. Since w is between v and u in the pre-order traversal, and u is a child of v, the only option is for w to be a descendant of v. Then the path from w to v consists of w, w's parent $p(w)$, $p(w)$'s parent, and so on until we reach v. Since a parent always comes before a child in a pre-order traversal, all the intermediate vertices on the path from w to v come between w and v in the pre-order traversal, so they must all be in V_i. This means w is in $C_{i,j}$ since w is connected to v in F_i. Since any vertex after v in V_i must be in $C_{i,j}$, $C_{i,j}$ must be the last connected component, i.e., $j = c_i$. This implies that the only possibly dependent connected component of F_i is C_{i,c_i}, and all other $C_{i,j}$'s for $j \in [c_i - 1]$ are independent. □

Lemma C.3.1 nicely fits with our result from Observation C.2.1 to bound the total number of phases BoundedContract requires. In addition, we can show how to implement

each phase to bound the complexity of our algorithm. Note that we are assuming that our component contracting function is defined to always yield a degree-weighted tree. We only need to show that the degrees stay bounded throughout the algorithm.

Proof of Theorem C.1.1. In each phase of this algorithm, the only modifications to the graph that occur are applications of the connected contracting functions to connected components of the tree. Since these are assumed to preserve $P(T)$ and we simply solve $P(T_l)$ for the final tree T_l, correctness of the output is obvious.

An important invariant in this algorithm is the $O(n^\varepsilon)$ bound on the degree of vertices throughout the algorithm. At the beginning, we know that the degrees are bounded according as it is promised in the input. We show that this bound on the maximum degree of the tree is invariant by proving the degree of vertices are still bounded after a single contraction.

Recall that we use preorder decomposition with $\lambda = n^\varepsilon$ to find the connected components we need to contract in the COMPRESS stage. According to definition, the total degree of each group in our decomposition is bounded by λ. After we contract a component S, the degree of the contracted vertex v^S never exceeds the sum of the degree of all vertices in S since every child of v^S is a child of exactly one of the vertices in S. Thus, the degree of v^S is bounded by $\lambda = n^\varepsilon$. In RAKE stage, we contract a number of sibling leaves into their common parent. In this case, the degree of the parent only decreases and the bound still holds.

We now focus on round and space complexities. A preordering can be computed using the preorder traversal algorithm from Behnezhad et al. [29], which executed in $O(1/\varepsilon)$ rounds with $O(n^\varepsilon)$ local space and $\widetilde{O}(n)$ total space w.h.p.[5] This completes step 1. In steps 2 and 3, the contracting functions are applied in parallel for a total of $O(1)$ rounds

[5]This means with probability at least $1/poly(n)$

(based off our assumption about any given contracting functions) within the same space constraints. Thus, all phases require $O(1)$ rounds except the first, which is $O(1/\varepsilon)$ rounds, and satisfy the space constraints of our theorem.

Now we must count the phases. Lemma C.3.1 tells us that for every group, we only have one non-leaf component in the dependency graph after each step 2. In step 3, we then "RAKE" all leaves into their parents. This means that the remaining number of vertices after step 3 is equal to the number of non-leaf vertices in the dependency graph after step 2, which is $k = n^{\varepsilon}$. Observation C.2.1 tells us that the resulting graph size is then $O(n/n^{\varepsilon}) = O(n^{1-\varepsilon})$. Therefore, in order to get a graph where $|T_l| = 1$, we require $O(1/\varepsilon)$ phases. Combining this with the complexity of each phase yields the desired result. □

C.3.2 Generalized α-Tree-Contractions

In the rest of this section we prove our main result: a generalized tree contraction algorithm, Algorithm 9. Building on top of Theorem C.1.1, we can create a natural extension of tree contractions. Recall from §C.2 that in the COMPRESS stage, we must contract maximal connected components containing only vertices v with degree $d(v) < \alpha$. Conveniently, by Theorem C.1.1, Algorithm 8 achieves precisely this. Therefore, to implement tree contractions, we simply need to:

1. Identify maximal connected components of low degree (Algorithm 9, line 3), which can be done in $O(1/\varepsilon)$ rounds by Behnezhad et al. [30].

2. Use our previous algorithm to execute the COMPRESS stage on each component (Algorithm 9, line 5), which can be done by Algorithm 8 in $O(1/\varepsilon^2)$ rounds.

3. Apply a function that can execute the RAKE stage (Algorithm 9, lines 7 through 14).

To satisfy the third step, we use a *sibling contracting function* (Definition C.4), which

can contract leaf-siblings of the same parent into a single leaf. Since a vertex might have up to n children, to do this in parallel, we may have to group siblings into n^{ε}-sized groups and repeatedly contract until we reach one leaf. Assuming sibling contractions are locally performed inside machines, this will then take $O(1/\varepsilon)$ AMPC rounds.

Algorithm 9: TreeContract
(Computing the solution $P(T)$ of a problem P on degree-weighted tree T using a connected contracting function $\mathscr{C}$ and a sibling contracting function $\mathscr{R}$)

Data: Degree-weighted tree $T = (V,E,W)$, a connected contracting function $\mathscr{C}$, and a sibling contracting function $\mathscr{R}$.
Result: The problem output $P(T)$.

$T_0 \leftarrow T$;
for $i \leftarrow 1$ **to** $l = O(1/\varepsilon)$ **do**
  Let $\mathscr{S}_{i-1} \leftarrow \mathsf{Connectivity}(T_{i-1} \setminus \{v \in V : deg(v) > n^{\varepsilon}\})$;
  Let $\mathscr{K}_{i-1} \leftarrow$ Components in $\mathscr{S}_{i-1,j}$ which represent a leaf in T'_{i-1};
  Contract each $S_{i-1,j} \in \mathscr{K}_{i-1}$ into $S'_{i-1,j}$ by applying $\mathsf{BoundedTreeContract}(S_j, \mathscr{C})$;
  Let $\mathscr{L}_{i-1}$ be the set of all maximal leaf-stars (excluding their parent) in T'_{i-1};
  for $L_{i-1,0} = \{v_1, v_2, \ldots, v_k\} \in \mathscr{L}_{i-1}$ **do**
    for $j \leftarrow 1$ **to** $1/\varepsilon$ **do**
      Split $L_{i-1,j-1}$ into $k/n^{j\varepsilon}$ parts $L_{i-1,j,1}, \ldots, L_{i-1,j,k/n^{j\varepsilon}}$ each of size n^{ε};
      Contract each $L_{i-1,j,z}$ into $L'_{i-1,j,z}$ by applying $\mathscr{R}(L_{i-1,j,z})$;
      Let $L_{i-1,j} \leftarrow \{L'_{i-1,j,1}, \ldots, L'_{i-1,j,k/n^{j\varepsilon}}\}$;
    end
    Contract $L_{i-1,1/\varepsilon}$ by applying $\mathscr{R}(L_{i-1,1/\varepsilon})$;
  end
  Let $T_i \leftarrow T'_{i-1}$;
end
return $\mathscr{C}(T_l)$;

We can show that this requires $O(1/\varepsilon)$ phases to execute, and each phase takes $O(1/\varepsilon^2)$ rounds to compute due to Theorem C.1.1 and our previous argument for RAKE by sibling contraction. Thus we achieve the following result:

Theorem C.1.2. *Consider a degree-weighted tree $T = (V,E,W)$ and a problem P. Given a connected contracting function and a sibling contracting function on T with respect to*

P, one can compute $P(T)$ in $O(1/\varepsilon^3)$ AMPC rounds with $O(n^\varepsilon)$ memory per machine and $\widetilde{O}(n)$ total memory.

Recall the definition of α-tree-contraction (Definition C.3) from §C.2.1. First, we prove Lemma C.3.2 to bound the number of phases in α-tree-contraction.

Lemma C.3.2. *For any $\alpha \geq 2$, the number of α-tree-contraction phases until we have a constant number of vertices is bounded by $O(\log_\alpha(n))$.*

To show Lemma C.3.2 we will introduce a few definitions. The first definition we use is a useful way to represent the resulting tree after each COMPRESS stage. Before stating the definition, recall the *Dependency Tree T'* of a tree T from Definition C.6.

Definition C.7. An **α-Big-Small Tree** T' is the *dependency tree* of a tree T with weighted vertices if it is a minor of T constructed by contracting all components of T made up of low vertices v with $deg(v) < \alpha$ (i.e., the connected components of T if we were to simply remove all vertices u with $deg(v) \geq \alpha$) into a single node.

We call a node v in T' with $deg(v) > \alpha$ in T a **big node**. All other nodes in T', which really represent contracted components of small vertices in T, are called **small components**.

Note that a small component may not be small in itself, but it can be broken down into smaller vertices in T. It is not hard to see the following simple property. This simply comes from the fact that maximal components of small degree vertices are compressed into a single small component, thus no two small components can be adjacent.

Observation C.3.1. *No small component in an α-Big-Small tree can be the parent of another small component.*

Consider our dependency tree T' based off a tree T that has been compressed. Obviously, T' is a minor of T constructed as described for α-Big-Small Tree because the weight

of a vertex equals its number of children (by the assumption of Lemma C.3.2). Note a small component refers to the compressed components, and a big node refers to nodes that were left uncompressed.

To show Lemma C.3.2, we start by proving that the ratio of leaves to nodes in T' is large. Since RAKE removes all of these leaves, this shows that T gets significantly smaller at each step. Showing that the graph shrinks sufficiently at each phase will ultimately give us that the algorithm terminates in a small number of phases.

Lemma C.3.3. *Let T_i' be the tree at the end of phase i. Then the fraction of nodes that are leaves in T_i' is at least $\alpha/(\alpha+4)$ as long as $w(v)$ is equal to the number of children of v for all $v \in T_i'$ and $\alpha \geq 2$.*

Proof. For our tree T_i', we will call the number of nodes n, the number of leaves ℓ, and the number of big nodes b. We want to show that $\ell > n\alpha/(\alpha+4)$. We induct on b. When $b = 0$, we can have one small component in our tree, but no others can be added by Observation C.3.1. Then $n = \ell = 1$, so $\ell > n\alpha/(\alpha+4)$.

Now consider T_i' has some arbitrary b number of big nodes. Since T_i' is a tree, there must be some big node v that has no big node descendants. Since all of its children must be small components and they cannot have big node descendants transitively, then Observation C.3.1 tells us each child of v is a leaf. Note that since v is a big node, it must have weight $w(v) > \alpha$, which also means it must have at least α children (who are all leaves) by the assumption that $w(v)$ is equal to the number of children.

Consider trimming T_i' on the edge just above v. The size of this new graph is now $n^* = n - w(v) - 1$. It also has exactly one less big node than T_i'. Therefore, inductively, we know the number of leaves in this new graph is at least $\ell^* \geq \frac{\alpha}{\alpha+4} n^* = \frac{\alpha}{\alpha+4}(n - w(v) - 1)$. Compare this to the original tree T_i'. When we replace v in the graph, we remove up to one leaf (the parent p of v, if p was a leaf when we cut v), but we add $w(v)$ new leaves. This

means the number of leaves in T_i' is:

$$\begin{aligned}
\ell =& \ell^* - 1 + w(v) \\
=& \frac{\alpha}{\alpha+4}(n - w(v) - 1) - 1 + w(v) \\
=& \frac{\alpha}{\alpha+4}n - \frac{\alpha}{\alpha+4}w(v) - \frac{\alpha}{\alpha+4} - 1 + w(v) \\
=& \frac{\alpha}{\alpha+4}n + \frac{4}{\alpha+4}w(v) - \frac{2\alpha+4}{\alpha+4} \\
>& \frac{\alpha}{\alpha+4}n + \frac{4}{\alpha+4}\alpha - \frac{2\alpha+4}{\alpha+4} && (1) \\
\geq& \frac{\alpha}{\alpha+4}n && (2)
\end{aligned}$$

Where in line (1) we use that $w(v) > \alpha$ and in line (2) we use that $\alpha \geq 2$. □

Now we can prove our lemma.

Proof of Lemma C.3.2. To show this, we will prove that the number of nodes from the start of one COMPRESS to the next is reduced significantly. Consider T_i as the tree before the ith COMPRESS and T_i' as the tree just after. Let T_{i+1} be the tree just before the $i+1$st COMPRESS, and let n_i be the number of nodes in T_i, n_i' be the number of nodes in T_i', and n_{i+1} be the number of nodes in T_{i+1}. Since T_i' is a minor of T_i, it must have at most the same number of vertices as T_i, so $n_i' \leq n_i$. Since T_{i+1} is formed by applying RAKE to T_i', then it must have the number of nodes in T_i' minus the number of leaves in T_i' (ℓ_i'). Therefore:

$$n_{i+1} = n_i' - \ell_i' \leq n_i' - \frac{\alpha}{\alpha+4}n_i' = \frac{4}{\alpha+4}n_i' \leq \frac{4}{\alpha+4}n_i$$

Where we apply both Lemma C.3.3 that says $\ell_i' \geq \frac{\alpha}{\alpha+4}n_i'$ and the fact that we just showed that $n_i' \leq n_i$. This shows that from the start of one compress phase to another, the number of vertices reduces by a factor of $\frac{4}{\alpha+4}$. Therefore, to get to a constant number of vertices, we require $\log_{\frac{\alpha+4}{4}}(n) = O(\log_\alpha(n))$ phases.

□

Now we are ready to prove our main theorem.

Theorem C.1.2. *Consider a degree-weighted tree $T = (V,E,W)$ and a problem P. Given a connected contracting function and a sibling contracting function on T with respect to P, one can compute $P(T)$ in $O(1/\varepsilon^3)$ AMPC rounds with $O(n^\varepsilon)$ memory per machine and $\widetilde{O}(n)$ total memory.*

Proof. We will show that our Algorithm 9 achieves this result. Lemma C.3.2 shows that there will be only at most $O(1/\varepsilon)$ phases. In each phase i, we start by running a connectivity algorithm to find maximally connected components of bounded degree, which takes $O(1/\varepsilon)$ time. Let $\mathscr{K}_{i-1}$ be the set of connected components which are leaves in T'_{i-1}. Then for each component $S_{i-1,j} \in \mathscr{K}_{i-1}$, we run BoundedTreeContract (Algorithm 8) in parallel using only our connected contracting function $\mathscr{C}$. Since the total degree of vertices over all members of $\mathscr{K}_{i-1}$ is not larger than $|T_{i-1}|$ and the amount of memory required for storing a degree-weighted trees is not larger than the total degree, the total number of machines is bounded above by $O(n^{1-\varepsilon})$. By definition, the maximum degree of any $S_{i-1,j}$ is n^ε. By Theorem C.1.1, each instance of BoundedTreeContract requires $O(1/\varepsilon^2)$ rounds, $O(|S_{i-1,j}|^\varepsilon)$ local memory and $\widetilde{O}(|S_{i-1,j}|)$ total memory. As $|S_{i-1,j}| \leq |T_{i-1}|$ (we know $|T_0| = n$ and it only decreases over time), we only require at most $O(n^\varepsilon)$ memory per machine. Since Since the total degree of vertices over all members of $\mathscr{K}_{i-1}$ is not larger than $|T_{i-1}|$, the total memory required is only $\widetilde{O}(|T_{i-1}|)) = \widetilde{O}(n)$. This is within the desired total memory constraints.

Finally, $\mathscr{R}$ is given to us as a sibling contractor. Consider the RAKE stage in our algorithm. We distribute machines across maximal leaf-stars. For any leaf-star with $n^\varepsilon \leq deg(v) \leq kn^\varepsilon$ for some (possibly not constant) k, we will allocate k machines to that vertex. Since again the number of vertices is bounded above by n, this requires only $O(n^{1-\varepsilon})$

machines. On each machine, we allocate up to n^ε leaf-children to contract into each other. We can then contract siblings into single vertices using $\mathscr{R}$. Since there are at most n children for a single vertex, it takes at most $O(1/\varepsilon)$ rounds to contract all siblings into each other. Then, finally, we can use $\mathscr{C}$ to compress the single child into its parent, which takes constant time.

Therefore, we have $O(1/\varepsilon)$ phases which require $O(1/\varepsilon^2)$ rounds each, so the total number of rounds is at most $O(1/\varepsilon^3)$. We have also showed that throughout the algorithm, we maintain $O(n^\varepsilon)$ memory per machine and $\widetilde{O}(n)$ total memory. This concludes the proof.

□

C.3.3 Simulating 2-tree-contraction in $O(1)$ AMPC rounds

Due to Theorem C.1.2, we can compute any $P(T)$ on trees as long as we are provided with a connected contracting function and a sibling contracting function with respect to P. A natural question that arises is the following: for which class of problem P there exists black-box contracting functions? We argue that many problems P for which we have a 2-tree-contraction algorithm can also be computed in $O(1/\varepsilon^3)$ AMPC rounds using n^ε-tree-contraction.

In many problems which are efficiently implementable in the Miller and Reif [114] *Tree Contraction* framework, we are given $\mathscr{C}$ and $\mathscr{R}$ contracting functions, for COMPRESS and RAKE stages respectively, which contract only one node: either a leaf in case of RAKE or a vertex with only one child in case of COMPRESS. Let us call this kind of contracting functions *unary contracting functions* and denote them by $\mathscr{C}^1$ and $\mathscr{R}^1$. This is a key point of original variants of Tree Contraction which contract odd-indexed vertices, or contract a maximal independent set of randomly selected vertices. Working efficiently regardless of using only *unary* contracting functions is the reason Tree Contraction was considered a

fundamental framework for designing parallel algorithms on trees in more restricted models such as PRAM. For example, in the EREW variant of PRAM, an $O(\log(n))$ rounds tree contraction requires to use only *unary* contracting functions $\mathscr{R}^1$ and $\mathscr{C}^1$. More generally, we define i-ary contracting functions as follows.

Definition C.8. An *"i-ary contracting function"*, denoted by $\mathscr{C}^i$ or $\mathscr{R}^i$, is a *contracting function* which admits a subset $S = \{v_1, v_2, \ldots, v_k\}$ of at most $i+1$ vertices at a time such that $\sum_{j=1}^{k} deg(v_j) = O(i)$. A special case of i-ary contracting functions, are *"unary contracting functions"*, denoted by $\mathscr{C}^1$ or $\mathscr{R}^1$, which contract only one vertex at a time.

However, in the the AMPC model, we can contract the chains more efficiently, and thus we are allowed to utilize more relaxed variants of COMPRESS stage. Furthermore, as we show in Theorem C.3.4, designing *unary* contracting functions $\mathscr{C}^1$ and $\mathscr{R}^1$ is not easier than designing *i-ary* contracting functions $\mathscr{C}^i$ and $\mathscr{R}^i$ in the AMPC model. We show this by reducing $\mathscr{C}^i$ and $\mathscr{R}^i$ to $\mathscr{C}^1$ and $\mathscr{R}^1$ in $O(1)$ rounds for any $i = O(n^\varepsilon)$. In other words, the restrictions of PRAM model, which requires $\mathscr{C}^1$ and $\mathscr{R}^1$ exclusively, enables us to directly translate a vast literature of problems solved using tree contraction to efficient AMPC algorithms for the same problem given $\mathscr{C}^1$ and $\mathscr{R}^1$.

As we have shown in Theorem C.1.2, it is possible to solve any problem $P(T)$ in $O(1/\varepsilon^3)$ AMPC rounds given a connected contracting function $\mathscr{C}$ and a sibling contracting function $\mathscr{R}$, where both are n^ε-ary contracting with respect to P. In what follows, we demonstrate the construction of n^ε-ary contracting functions given a unary connected contracting function $\mathscr{C}^1$ and a sibling contracting function $\mathscr{R}^1$.

Theorem C.3.4. *Given a* unary connected contracting function $\mathscr{C}^1$ *and a* unary sibling contracting function $\mathscr{R}^1$ *with respect to a problem P defined on trees, one can build an* i-ary connected contracting function $\mathscr{C}^i$ *and an* i-ary sibling contracting function $\mathscr{R}^i$ *with respect to P and both* $\mathscr{C}^i$ *and* $\mathscr{R}^i$ *run in one AMPC rounds as long as* $i = O(n^\varepsilon)$.

Proof. First, we present an algorithm for $\mathscr{C}^i$. We are given a connected subtree induced by $S = \{v_1, v_2, \ldots, v_k\}$ of T so that $\sum_{j=1}^{k} deg_T(v_j) = O(i)$. Since $i = O(n^\varepsilon)$, the whole subtree fits into the memory of a single machine. Some of the leaves of this subtree are *known*, meaning that they are a leaf also in T, and others are *unknown*, meaning that they have children outside S. Let $\mathscr{U} = \{u_1, u_2, \ldots, u_l\}$ be the set of the children of *unknown* leaves as well as the children of non-leaf nodes which are outside of S. Ultimately, we want to compress the data already stored on the vertices of S into a memory of $\widetilde{O}(l)$ as the degree of v_1 in the contracted tree T' will be $l+1$, and thus $deg_{T'}(v_1) = l+1$.

The $W(v_1)$ of each contracted vertex v_1 is a weighted-degreee tree structure $T^{\mathscr{C}}(v_1)$ whose leaves are the children of v_1 in T', and there is no vertex with exactly one child in $T^{\mathscr{C}}(v_1)$. Thus, the number of vertices in $T^{\mathscr{C}}(v_1)$ is bounded by $O(l) = O(deg_{T'}(v_1))$. In addition, we are guaranteed that the total size of vectors on each vertex of $T^{\mathscr{C}}(v_1)$ is bounded by $|T^{\mathscr{C}}(v_1)|$ since $\mathscr{C}^1$ and $\mathscr{R}^1$ are unary contracting functions. Therefore, we assume each $W(v_j)$ for each vertex $v_j \in S$ has stored a tree structure of size $\widetilde{O}(deg_T(v_j))$. We concatenate all these trees to get an initial $T^{\mathscr{C}}(v_1)$ whose size is bounded by $\sum_{j=1}^{k} \widetilde{O}(deg_T(v_j)) = \widetilde{O}(n^\varepsilon)$.

We run a 2-tree-contraction-*like* algorithm locally on $T^{\mathscr{C}}(v_1)$ using $\mathscr{C}^1$ and $\mathscr{R}^1$. Note that we can only rake the known leaves since the data of unknown leaves depend on their children. We repeating COMPRESS and RAKE stages until there is no known leaf or a vertex with one child remain in $T^{\mathscr{C}}(v_1)$. Then, according to Lemma C.3.2 for $\alpha = 2$, the number of remaining vertices in $T^{\mathscr{C}}(v_1)$ is bounded by $O(l)$. We store the final $T^{\mathscr{C}}(v_1)$ in T' which requires a memory of $\widetilde{O}(l) = \widetilde{O}(w_{T'}(v_1))$. Hence, $\mathscr{C}^i$ satisfies the size-constraint on the weight vectors of the resulting weighted-degree tree.

Finally, we present an algorithm for $\mathscr{R}^i$ which is more straight-forward compared to that of $\mathscr{C}^i$. We are given a leaf-star $S = \{v_1, v_2, \ldots, v_k\}$ of T so that $\sum_{j=1}^{k} deg_T(v_j) = O(i)$. This implies that there are at most $O(n^\varepsilon)$ vertices in S as long as $i = O(n^\varepsilon)$, and we can fit the whole S into a memory of a single machine. To simulate $\mathscr{R}^i$, we only need to $k-1$

times apply $\mathscr{R}^1$ on $S_j = \{v_1, v_{j+1}\}$ at the j-th iteration. Note that every $v_j \in S$ is a leaf in T, so the data stored in $W(v_j)$ is just $\widetilde{O}(1)$ bits and not a tree structure. Theorem C.3.4 statement is implied. □

C.3.4 Reconstructing the Tree for Linear-sized Output Problems

Consider a problem $P(T)$ whose output size is also linear in the size of input n. For instance, in maximum weighted matching we need to find the matching itself. Up to this point, in all of our algorithms, we assume the output of function $P(T)$ is of constant-sized. We simply contract the tree through some iterations until it collapses into a single vertex, and we do not need to remember anything about a vertex which is contracted as a member of a connected component or as a member of a leaf-star.

In this section, we present a general approach for retrieving the linear-sized solution in a natural scenario, where we need to retrieve a recursively-defined weight vector $P(v)$ of constant size for each vertex $v \in V$. In the special case of maximum weighted matching which can be formulated as a dynamic programming problem, $P(v)$ contains the final value of different DP values with respect to the subtree rooted at v[6].

Roughly speaking, our reconstruction algorithm is based on storing the information about components we contracted throughout the algorithm in an auxiliary memory of size $O(n)$. It is easy to observe that if we store the degree-weighted subtree of every connected component or leaf-star that we contract during the algorithm we need at most $O(n)$ addition total memory. Note that during each application of black-box contracting functions, we remove at least one vertex from the tree and each vertex except root is removed exactly once when the algorithm terminates. Namely, for every phase i we need to store $\mathscr{S}_i$ and $\mathscr{L}_i$ in Algorithm 8, and $\mathscr{K}_i$ and every $L_{i,j,z}$ in Algorithm 9 (In addition to the data stored by

[6]Note that retrieving $P(v)$ for each vertex v still does not give us the optimum matching and a problem-specific post-processing step is required to retrieve the actual matching

each black-box application of Algorithm 8). Since we have adaptive access to these subsets in AMPC, it is sufficient to index them by the id of the surviving vertex of each subset.

The full reconstruction algorithm starts after the main contraction algorithm finishes. We only need to store the information about contracted subsets during the running time of the contraction algorithm. Next, we iterate over the phases of the algorithm in reverse order, i.e., $i = \{1/\varepsilon, 1/\varepsilon - 1, \ldots, 1\}$, and undo the contractions that were performed during phase i. Let C_v be a connected contracted component rooted at v, and $\{w_1, w_2, \ldots, w_k\}$ be children of v post-contraction.

Whenever we undo a connected contraction like C_v, we replace v with the whole structure of C_v including $W(u)$ for every $u \in C_v, u \neq v$. Then we populate the $P(u)$ for every $u \in C_v, u \neq v$. During the contraction algorithm $P(w_j)$ is not known for any j. However, during the reconstruction we know $P(w_j)$ for every $1 \leq j \leq k$ since these vertices are contracted in a later phase than the phase we contract C_v. Hence, we have already populated $P(w_j)$ and we can use these values to locally populate $P(u)$ for every $u \in C_v$. Undoing the sibling contracting functions in much simpler since their values do not depend on other vertices nor the value of other vertices depend on their value. We populate $P(u)$ for every $u \in L$, where L is a leaf-star, based on the already constant-sized weight vectors $W(u)$.

C.4 Conclusion

This paper introduces some of the first generalized techniques for solving various problems in the AMPC model. Specifically, we show that Miller and Reif's [113] $O(\log n)$-time PRAM tree contraction algorithm can be efficiently extended to a constant-round low-memory AMPC algorithm. This implies $O(1/\varepsilon^2)$-round algorithms for expression evaluation, tree isomorphism testing, maximal matching on trees, and maximal independent set on trees. It additionally implies $O(1/\varepsilon^3)$-round algorithms for maximum weighted

matching and maximum weighted independent set on trees. However, we expect these algorithms to have much broader applications to tree-based problems, as did the original work by Miller and Reif.

It remains to be seen precisely which of extensions of the PRAM algorithm apply to the AMPC model. Many of them require computational overhead beyond the black-box application of the tree contraction process (which our algorithm can directly and efficiently simulate), therefore the extension of those applications to this work is highly non-trivial. Of notable interest is the application of tree contractions to graphs of bounded treewidth, where Bodlaender and Hagerup [34] showed how to construct low-width tree decompositions using PRAM tree contractions. If AMPC tree contractions can also solve this and additionally solve tree contractions on graphs of bounded tree-width, then our work can be notably generalized. Another potential course of research is the exploration of problems such as MedianParent, which *cannot* be simplified to problems on trees with bounded tree width. It is an open question if these, too can be solved in $O(1/\varepsilon^2)$ rounds.

Part II

String Problems

Appendix D: String Matching

D.1 Introduction

The string matching problem with wildcards, or *pattern matching*, seeks to identify pieces of a text that adhere to a certain structure called the *pattern*. Pattern matching is one of the most applied problems in computer science. Examples range from simple batch applications such as Awk, Sed, and Diff to very sophisticated applications such as anti-virus tools, database queries, web browsers, personal firewalls, search engines, social networks, etc. The astonishing growth of data on the internet as well as personal computers emboldens the need for fast and scalable pattern matching algorithms.

In theory too, pattern matching is a well-studied and central problem. The simplest variant of pattern matching, namely *string matching*, dates back to 1960s. In this problem, two strings T and P are given as input and the goal is to find all substrings of T that are identical to P. The celebrated algorithm of Knuth, Morris, and Pratt [98] (KMP) deterministically solves the problem in linear time. Since then, attention has been given to many variants and generalizations of pattern matching [2, 3, 17, 32, 39, 43, 44, 56, 84–86, 98, 103, 105, 109, 116, 117, 123–127, 131]. Natural generalizations of string matching are when either the text or the pattern is a tree instead of a string [56, 84, 127, 131] or when the pattern has a more sophisticated structure that allows for '?', '+', '*', or in general any regular expression [17, 39, 86, 103]. Also, different computational systems have been considered in the literature: from sequential algorithms [17, 39, 56, 84, 86, 98, 103, 127, 131],

to quantum algorithms [32, 109, 116, 126], to distributed settings [3, 44, 85], to the streaming setting [2, 43, 123], to PRAM [105, 117, 124, 125], etc.

An obvious application of pattern matching is in anti-virus softwares. In this case, a malware is represented with a pattern and a code or data is assumed to be infected if it contains the pattern. In the simplest case, the pattern only consists of ascii characters. However, it happens in practice that malwares allow for slight modifications. That is, parts of the pattern code are subject to change. This can be captured by introducing wildcards to pattern matching. More precisely, each element of the pattern is either an ascii code or a special character '?' which stands for a wildcard. The special character is allowed to match with any character of the text.

Indeed the desirable property of the '?' case is that the length of the pattern is always fixed. However, one may even go beyond this setting and consider the cases where the pattern may match to pieces of the text with variant lengths. Two classic ways to incorporate this into the model is to consider two special characters '+' and '*'. The former allows for arbitrary repetitions of a single character and the latter allows for arbitrary repetitions of any combination of characters. For example, as an application of pattern matching in bioinformatics, we might be looking for a set of gene patterns in a DNA sequence. Obviously, these pattern are not necessarily located consecutively in the DNA sequence, and one might utilize '*' wildcard to address this problem.

In practice, these problems are formulated around huge data sets. For instance, a human DNA encompasses roughly a Gigabyte of information, and an anti-virus scans Gigabytes (if not Petabyets) of data on a daily basis. Thus, the underlying algorithm has to be scalable, fast, and memory efficient. A natural approach to obtain such algorithms is parallel computation. Motivated by such needs, the massively parallel computation (MPC) model [6, 22, 76, 95] has been introduced to understand the power and limitations of parallel algorithms. It first proposed by Karloff et al. [95] as a theoretical model to embrace Map

Reduce algorithms, a class of powerful parallel algorithms not compatible with previously defined models for parallel computation. Recent developments in the MPC model have made it a cornerstone for obtaining massively parallel algorithms.

While in the previous parallel settings such as the PRAM model, usually an $O(\log n)$ factor in the round complexity is inevitable, MPC allows for sublogarithmic round complexity [49, 95, 102]. Karloff et al. [95] also compared this model to PRAM, and showed that for a large portion of PRAM algorithms, there exists an MPC algorithm with the same number of rounds. In this model, each machine has unlimited access to its memory, however, two machines can only interact in between two rounds. Thus, a central parameter in this setting is the round complexity of algorithm since network communication is the typical main bottleneck in practice. The ultimate goal is developing constant-round algorithms, which are highly desirable in practice.

The MPC model. In this paper, we assume that the input size is bounded by $O(n)$, and we have M machines of each with a memory of S. In the MPC model [6, 22, 76, 95], we assume the number of machines and the local memory size on each machine is asymptotically smaller than the input size. Therefore, we fix an $0 < x < 1$ and bound the memory of each machine by $\widetilde{O}(n^{1-x})$. Also, our goal is to have near linear total memory and therefore we bound the number of machines by $\widetilde{O}(n^{x})$. An MPC algorithm runs in a number of rounds. In every round, every machine makes some local computation on its data. No communication between machines is allowed during a round. Between two rounds, machines are allowed to communicate so long as each machine receives no more communication than its memory. Any data that is outputted from a machine must be computed locally from the data residing on the machine and initially the input data is distributed across the machines.

In this work we give MPC algorithms for different variants of the pattern matching problem. For the regular string matching and also '?' and '+' wildcard problems, our

algorithms are tight in terms of running time, memory per machine, and round complexity. Both '?' and '+' wildcard problems are reduced to fast convolution at the end, and make use of the fact that FFT could computed in $O(1)$ MPC rounds with near-linear total running time and total memory. Also, for the case of '*' wildcard we present nontrivial MPC algorithms for two special cases that mostly tend to happen in practice. However, the round complexity of these two cases is $O(\log(n))$, and the general case problem is not addressed in this paper.

D.1.1 Our Results and Techniques

Throughout this paper, we denote the text by T and the pattern by P. Also, we denote the set of characters by Σ.

We begin, as a warm-up, in Section D.3 by giving a simple MPC algorithm that solves string matching in 2 rounds. The basic idea behind our algorithm is to cleverly construct hash values for the substrings of the text and the pattern. In other words, we construct an MPC data structure that enables us to answer the following query in a single MPC round:

Given indices i and j of the text, what is the hash value for the substring of the text starting from position i and ending at position j?.

Indeed, after the construction of such a data structure, one can solve the problem in a single round by making a single query for every position of the text. This gives us a linear time MPC algorithm that solves string matching in constant rounds.

Result 4 (Theorem D.3.1)**.** There exists an MPC algorithm that solves string matching in constant rounds. The total memory and the total running time of the algorithm are linear.

For the case of wildcard '?', the hashing algorithm is no longer useful. It is easy to see that since special '?' characters can be matched with any character of the alphabet, no

hashing strategy can identify the matches. However, a more sophisticated coding strategy enables us to find the occurrences of the pattern in the text. Assume for simplicity that $m = |\Sigma|$ is the size of the alphabet and we randomly assign a number $1 \leq \mathsf{mp}_c \leq m$ to each character c of the alphabet. Moreover, we assume that all the numbers are unique that is for two characters c and c' we have $\mathsf{mp}_c = \mathsf{mp}_{c'}$ if and only if $c = c'$. Now, construct a vector $T^\dagger$ of size $2|T|$ such that $T^\dagger_{2i-1} = \mathsf{mp}_{T_i}$ and $T^\dagger_{2i} = 1/\mathsf{mp}_{T_i}$ for any $1 \leq i \leq |T|$. Also, we construct a vector $P^\dagger$ of size $2|P|$ similarly, expect that $P^\dagger_{2i-1} = P^\dagger_{2i} = 0$ if the i'th character of P is '?'. Let nz_P be the number of the normal characters ('?' excluded) of the pattern. It follows from the construction of $T^\dagger$ and $P^\dagger$ that if P matches with a position i of the text, then we have:

$$T^\dagger[2i-1, 2i+2(|P|-1)].P^\dagger = \mathsf{nz}_P$$

where $T^\dagger[2i-1, 2i+2(|P|-1)]$ is a sub-vector of A only containing indices $2i-1$ through $2i+2(|P|-1)$. Moreover, it is showed in [59] that the vice versa also holds. That is if $T^\dagger[2i-1, 2i+2(|P|-1)].P^\dagger = \mathsf{nz}_P$ for some i then pattern P matches position i. This reduces the problem of pattern matching to the computation of dot products which is known to admit a linear time solution using fast Fourier transform (FFT) [38].

Result 5 (Theorem D.4.2)**.** There exists an MPC algorithm that computes FFT in constant rounds. The total memory and the total running time of the algorithm are $\widetilde{O}(n)$.

Corollary D.4.3 gives us an efficient MPC algorithm for the wildcard setting. While the reduction from wildcard matching to FFT is a known technique [46] the fact that FFT is computable in $O(1)$ MPC rounds leads to efficient MPC algorithm for a plethora of problems. FFT is used in various combinatorial problems such as knapsack [21], 3-sum [41], subset-sum [99], tree-sparsity [15], tree-separability [21], necklace-alignment [37], etc.

The case of '+' and '*' wildcards are technically more involved. In the case of repetition, special characters '+' may appear in the pattern. These special characters allow

for arbitrary repetitions of a single character. For instance, a pattern "a+bb+" matches all strings of the form $\mathsf{a}^x.\mathsf{b}^y$ such that $x \geq 1$ and $y \geq 2$. Indeed this case is more challenging as the pattern may match with substrings of the text with different lengths.

To tackle this problem, we first compress both the text and the pattern into two strings T° and P° using the run-length encoding method. In the compressed versions of the strings, we essentially avoid repetitions and simply write the numbers of repetitions after each character. For instance, a text "aabcccddad" is compressed into "a[2]b[1]c[3]d[2]a[1]d[1]" and a pattern "ab+ccc+" is compressed into "a[1]b[1+]c[3+]". With this technique, we are able to break the problem into two parts. The first subproblem only incorporates the characters which is basically the conventional string matching. The second subproblem only incorporates repetitions. More precisely, in the second subproblem, we are given a vector A of n integer numbers and a vector B of m entries in the form of either i or $i+$. An entry of i matches only with indices of A with value i but an entry of $i+$ matches with any index of A with a value at least i. Since we already know how to solve string matching efficiently, in order to solve the repetition case, we need to find a solution for the latter subproblem.

To solve this subproblem, we use an algorithm due to Cole and Hariharan [46]. They showed the subset matching problem could be solved in near-linear time. The definition of the subset matching problem is as follows.

Problem D.5.2 (restated). *Given T, a vector of n subsets of the alphabet Σ, and P, a vector of m subsets in the same format as T, find all occurrences of P in T. P is occurred at position i in T if for every $1 \leq j \leq |P|$, $P_j \subseteq T_{i+j-1}$.*

We can reduce our problem to an instance of the subset matching problem by replacing every T_i with $\{1+, 2+, \ldots, T_i+\} \cup \{T_i\}$, and keep S intact, i.e., replacing each S_i with $\{S_i\}$.

This way, if there is a match, each S_i has to be included in the respective T_j. There is an algorithm for this problem with $\widetilde{O}(s)$ running time [46], where s shows the total size of all subsets in T and P. The running time is good enough for our algorithm to be near linear since s is at most twice the number of characters in the original text and pattern. Each T_i is the compressed version of T_i consecutive repetitions of a same character in T. We also show that the subset matching problem could be implemented in a constant number of MPC rounds, and thus a constant-round MPC algorithm for string matching with '+' wildcard is implied.

Result 6 (Theorem D.5.4)**.** There exists an MPC algorithm that solves string matching with '+' wildcard in constant rounds. The total memory and the total running time of the algorithm are $\widetilde{O}(n)$.

Despite positive results for '?' and '+' wildcards, we do not know whether a poly-logarithmic round MPC algorithm exists for '*' wildcard in the most general case or not. This wildcard character matches any substring of arbitrary size in the text. We can consider a pattern consisted of '*' and Σ characters as a sequence of subpatterns, maximal substrings not having any '*', with a '*' between each consecutive pair. In the sequential settings, we can solve the problem by iterating over the subpatterns, and find the next matching position in the text for each one. If we successfully find a match for every subpattern in order, we end up with a substring of T matching P. Otherwise, it is easy to observe that T does not match the pattern P. To find the next matching position, we can perform a KMP algorithm on T and all the subpatterns one at a time, and make a transition to the next subpattern whenever we find a match, which is still linear in $n+m$, as the total size of the subpatterns is limited by m. The algorithm is provided with more details in Observation D.6.1.

However, things are not as easy in the MPC model, because each subpattern can happen virtually anywhere in the text, and furthermore the number of subpatterns could be as large

as $O(m)$. Thus, intuitively, it is impossible to know which subpatterns match each location of T with a linear total memory since the total size of this data could be $\Omega(nm)$, and also we cannot transfer this data in poly-logarithmic number of MPC rounds. Nonetheless, we provide two examples of how we can overcome this restriction by adding a constraint on the input. In both the cases, a near-linear $O(\log(n))$-round MPC algorithm exists for solving '*' wildcard problem.

1. When the whole pattern fits in a single machine, $m = O(n^{1-x})$, then the number of subpatterns is limited by $O(n^{1-x})$. Thus, each machine could access each subpattern, and finds out if an interval of subpatterns happens in its part of input. At the end, we merge these pieces of information using dynamic programming to obtain the result.

   Result 7 (theorem D.6.1)**.** Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{'*'}\}^m$ for $m = O(n^{1-x})$, there is an MPC algorithm to find the solve the string matching problem in $O(\log n)$ rounds using $O(n^x)$ machines.

2. When no subpattern is a prefix of another, then the number of different subpatterns matching each starting position is limited by 1. Exploiting this property, we can find out which subpattern matches each starting position, if any. Next, we create a graph with positions in T as vertices, and we put an edge from a position i which matches a subpattern P_k, to the minimum position j matching subpattern P_{k+1}, which is also located after $i + |P_k| - 1$. This way, '*' wildcard string matching reduces to a graph connectivity problem; finding whether there is path from a position matching the first subpattern to a position matching the last subpattern. Therefore, we will have a $O(\log(n))$-round MPC algorithm for '*' wildcard problem using the standard graph connectivity results in MPC [95].

Problem		Theorem	Rounds	Total Runtime	Nodes
$P \in \Sigma^m$		D.3.1	$O(1)$	$O(n)$	$O(n^x)$
$P \in \{\Sigma \cup \text{`?'}\}^m$		D.4.3	$O(1)$	$\widetilde{O}(n)$	$O(n^x)$
$P \in \{\Sigma \cup \text{`+'}\}^m$		D.5.4	$O(1)$	$\widetilde{O}(n)$	$O(n^x)$
$P \in \{\Sigma \cup \text{`*'}\}^m$	$m = O(n^{1-x})$	D.6.1	$O(\log(n))$	$O(n)$	$O(n^x)$
	no prefix	D.6.2	$O(\log(n))$	$O(n)$	$O(n^x)$

Table D.1: Overview of results

Result 8 (Theorem D.6.2)**.** Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{`*'}\}^m$ such that subpatterns are not a prefix of each other, there is an MPC algorithm to find the solve the string matching problem in $O(\log n)$ rounds using $O(n^x)$ machines.

D.2 Preliminaries

In the pattern matching problem, we have two strings T and P of length n and m over an alphabet Σ. In the string matching problem, the first string T is a text, and the second string P is the pattern we are looking for in T. For a string s, we denote by $s[l,r]$ the substring of s from l to r, i.e., $s[l,r] = \langle s_l, s_{l+1}, \ldots, s_r \rangle$. Given two strings T and P, we are looking for all occurrences of the pattern P as a substring of T. In other words, we are looking for all $i \in [1, n-m+1]$ such that $T[i, i+m-1] = P$. In this case, we say substring $T[i, i+m-1]$ matches pattern P.

Problem D.2.1. Given two strings $T \in \Sigma^n$ and $P \in \Sigma^m$, we want to find all occurrences of the string P (pattern) as a substring of string T (text).

Problem D.2.1 has been vastly studied in the literature, and there are a number of solutions that solve the problem in linear time. [98] However, the main focus of this paper is to design massively parallel algorithms for the pattern matching problem. We consider MPC

as our parallel computation model, as it is a general framework capturing state-of-the-art parallel computing frameworks such as Hadoop MapReduce and Apache Spark.

We extend the problem of string matching adding wildcard characters to the pattern. A wildcard character is a special character $\phi \notin \Sigma$ in pattern that is not required to match by the same character in T. For example, wildcard character '?' can be matched by any arbitrary character in T. For instance, let T and P be "abracadabra" and "a?a" respectively. Then, pattern P occurs at $i = 4$ and $i = 6$ since $P \overset{?}{=} T[4,6]$ and $P \overset{?}{=} T[6,8]$. Notation $\overset{\phi}{=}$ denoted the equality of two strings regarding the wildcard character ϕ.

We consider three kinds of wildcard characters in this paper:

1. Character replacement wildcard '?': Any character can match character replacement wildcard.

2. Character repetition wildcard '+': If character repeat wildcard appears immediately after a character c in the pattern, then any number of repetition for character c is accepted.

3. String replacement wildcard '*': A string replacement wildcard can be matched with any string of arbitrary length.

D.2.1 Hashing

To boost comparing the pattern p with the substrings of string s, we can compare the fingerprints or hashes instead. A hash of string s, denoted by $h(s)$, is a single number such that $h(s) \neq h(s')$ implies $s \neq s'$ always, and $h(s) = h(s')$ implies $s = s'$ with a high probability. Consider a naive hash function h_r such that $h_r(s) = \sum_{i=1}^{|s|} s_i \mod r$, where r is an arbitrary number. Note that we can label the characters in Σ with a permutation of size $|\Sigma|$, thereby, considering a numerical value for each character to simplify the formulas.

This hash function does not guarantee a high probability of $s = s'$ in the case of $h_r(s) = h_r(s')$, because this hash function is not locally sensitive, i.e., modifying two consecutive characters in s in a way that the first one reduced by one and the other one increased by one gives us the same hash.

However, simple hash function h_r has a desirable property that is rolling. A rolling hash function h allows us to achieve $h(s[l+1, r+1])$ from $h(s[l,r])$ in $O(1)$ time, i.e., $h_r(s[l+1, r+1]) = h_r(s[l,r]) - s_l + s_{r+1} \mod r$. Using a rolling hash function, one can imagine a Monte-Carlo randomized algorithm for string matching problem by first computing $h(p)$ and $h(s_{1,m})$ in $O(m)$ time, and then building $h(s[i+1, i+m])$ from $h(s[i, i+m-1])$ for i from 1 to $n-m$ in $O(n)$ time. Afterwards, we can compare the hash of each substring with $h(p)$ in $O(1)$ time. Therefore, we desire a locally sensitive rolling hash function. Karp and Robbin [96] provided such rolling hash functions that guarantee a bounded probability of hash collision. An example of such hash function is

$$h_{r,b}(s) = \sum_{i=1}^{|s|} s_i b^{|s|-i} \mod r$$

The difference between this hash function and the previous one is local sensitivity. It's not likely anymore to end up with the same hash applying few modifications if we chose r to be a prime number or simply chose such r and b to be relatively prime numbers. Besides, the hash collision probability of an ideal hash function is r^{-1}, and no matter how we minimize the correlation of the hashes of similar strings, we cannot achieve a smaller probability. By the way, $h_{r,b}$ with relatively prime r and b achieve a probability in that this correlation is almost negligible. Therefore, it suffices to pick a large enough r to guarantee high probability. More precisely, we want $\bigcup_{i=1}^{n-m+1} \mathbf{Pr}[P \neq T[i, i+m-1] \wedge h(P) = h(T[i, i+m-1])]$ be at most $1/n$. Thus,

$$\bigcup_{i=1}^{n-m+1} \mathbf{Pr}[P \neq T[i,i+m-1] \wedge h(P) = h(T[i,i+m-1])] \leq \frac{1}{n} \Rightarrow$$
$$\sum_{i=1}^{n-m+1} \mathbf{Pr}[P \neq T[i,i+m-1] \wedge h(P) = h(T[i,i+m-1])] \leq \frac{1}{n} \Rightarrow$$
$$O(n)\frac{1}{r} \leq \frac{1}{n} \Rightarrow r \geq O(n^2) \Rightarrow \log(r) \geq O(\log(n))$$

Therefore, the number of bits required to store hashes should be logarithmic in the size of the text which is a fairly reasonable assumption. Alternatively, we can reduce the probability substantially by comparing hashes based on multiple values of r. Hence, we can safely assume that using locally such sensitive hash functions, we can guarantee a high probability of avoiding hash collisions.

Fact D.2.1. Given two strings $T \in \Sigma^n$ and $P \in \Sigma^m$, using locally sensitive rolling hash functions one can find all occurrences of the string P (pattern) as a substring of string S (text) in $O(n+m)$ time with a high probability.

In addition to rolling property, we can consider a more general property for $h_{r,b}$ namely partially decomposability. A hash function h is partially decomposable if it is possible to calculate in $O(1)$ time $h(s[l,r])$ from the hash of the prefixes of s, that is $h(s[1,i])$ for all $i \in [1,|s|]$. In addition, it should be possible to calculate the hash of the concatenation of two strings in $O(1)$ time. For example, $h_{r,b}(s[l,r]) = (h_{r,b}(s[1,r]) - h_{r,b}(s[1,l-1])b^{r-l+1}) \mod r$, and $h_{r,b}(s+s') = (h_{r,b}(s)b^{|s'|} + h_{r,b}(s')) \mod r$, where $s+s'$ means the concatenation of s and s'.

Throughout this paper, we refer to the suitable hash function for an instance of string matching problem by h and ignore the internal complications of the hash functions.

D.3 String matching without wildcard

In this section, we provide an algorithm for the string matching problem with no wildcards; Given strings T and P, we wish to find all the starting points in T that match P. The round complexity of our algorithm, which is the most important factor in MPC model, is constant. Furthermore, the algorithm is tight in a sense that the total running time and memory is linear. For the more restricted cases, we enhance our algorithm to work in even less rounds.

The algorithm consists of three stages. We discuss the different stages of the algorithm with details in the following. The main goal of the algorithm is to compare the hash of each length m substring of string T with the hash of string P, and therefore, find an occurrence if they are equal. We consider the following properties, namely partially decomposability, for our hash function h,

- Merging the hashes of two strings s and s', to find the hash of their concatenation $s+s'$, can be done in $O(1)$.
- Querying the hash of a substring $s[l,r]$ can be done in $O(1)$, with $O(|s|)$ preprocessing time.

Consider strings T and P are partitioned into several blocks of size S, each fit into a single machine. We denote these blocks by $T^1, T^2, \ldots, T^{n/\mathsf{S}}$ and $P^1, P^2, \ldots, P^{m/\mathsf{S}}$. It is easy to solve the problem when P is small relative to S. We can search for P in each block independently by maintaining the partial hash for each prefix of the block. The caveat of this solution is that some substrings lie in two machines. We can fix this issue by feeding the initial string chunks with overlap. If the length of the overlaps is greater than m, it is guaranteed than each candidate substring is contained as a whole in one of the initial string chunks. We call this method "Double Covering". Utilizing this method, we can propose an

algorithm which leads to Observation D.3.1. Therefore, the main catch of the algorithm is to deal with the matching when string P spans multiple blocks.

Observation D.3.1. *Given $T \in \Sigma^n$ and $P \in \Sigma^m$ where $m = O(n^{1-x})$, there is an MPC algorithm for string matching problem with no wildcards in* 1 *MPC round using $O(n^x)$ machines in linear total running time.*

Proof. We provide for each machine, which holds T^i, the next m characters in the T, i.e., $T[i\mathsf{S}+1 : i\mathsf{S}+m]$, which is viable since $m = O(\mathsf{S})$. Therefore, we can compute the hash of each substring starting in T^i as well as P in each machine. See Algorithm 10 for more details. □

Algorithm 10: StringMatching(a)

Data: two array T and P, where $m = O(\mathsf{S})$
Result: function $f : \{1,2,\ldots,n-m+1\} \to \{0,1\}$ such that $f(i) = 1$ iff $T[i : i+m-1] = P$.

$f(i) \leftarrow 0 \;\; \forall 1 \le i \le n-m+1$;
send a copy of P as well as $T[i\mathsf{S}+1 : i\mathsf{S}+m]$ to the i-th machine, which holds T^i, truncate if $i\mathsf{S}+m$ surpasses n.
Run in parallel: **for** $1 \le i \le n/\mathsf{S}$ **do**
    append $T[i\mathsf{S}+1 : i\mathsf{S}+m]$ to T^i;
    for $1 \le j \le \mathsf{S}$ **do**
        if $h(P) = h(T^i[j : j+m-1])$ **then**
            $f((i-1)\mathsf{S}+j) \leftarrow 1$;

Theorem D.3.1. *For any $x < 1/2$, given $T \in \Sigma^n$ and $P \in \Sigma^m$, there is an MPC algorithm for string matching problem with no wildcards in $O(1)$ MPC rounds using $O(n^x)$ machines in linear total running time.*

Intuitively, we can handle larger patterns by transferring the partial hash of each ending point to the machine which the respective starting point lies in. In addition, we compute the hash of each k first blocks, and by which, we can find the hash of each interval of blocks.

Note that we need to assume $x < 1/2$, so that memory of a single machine be capable of storing $O(1)$ information regarding each machine. The algorithm is outlined in Algorithm 11.

Proof. Assume there are $(n+m)/\mathsf{S}+2$ machines as following:

- machines $a_1, a_2, \ldots, a_{n/\mathsf{S}}$ which T^i is stored in a_i.
- machines $b_1, b_2, \ldots, b_{m/\mathsf{S}}$ which P^i is stored in b_i.
- machines c and d which are used for aggregation purposes.

At the first round, we compute the hash of each block T^i and P^i and send them to machines c and d respectively. Furthermore, we compute the partial hashes of each prefix of each block T^i, that is $h(T^i[1,j])$ for $1 \leq j \leq \mathsf{S}$. The calculated hash of each prefix should be sent to the machine containing the respective starting point. Therefore, $T^i[1,j]$ should be sent to the node containing starting point $(i-1)\mathsf{S}+j-m+1$. The total communication overhead of this round is $O(n+m)$.

In the next round, we aggregate the calculated hashes in the previous round. In node d, we calculate the hash of string P by merging the hashes of $T^1, T^2, \ldots, T^{m/\mathsf{S}}$. The value of $h(P)$ should be sent to all a_i for $1 \leq i \leq n/\mathsf{S}$. Those nodes are supposed to compare $h(P)$ with the hash of each starting point lying in their block to find the matches in the last round. Furthermore, in node c, the hash of each first k blocks of T should be calculated, i.e., $h(T^1+T^2+\ldots+T^i)$ for all $1 \leq i \leq n/\mathsf{S}$. Each machine a_i needs to compute the hash of a sequence of consecutive blocks of T that lie between each starting point in T^i and its respective end point. Therefore, each machine should receive up to $O(1)$ of these hashes, because the set of respective ending points spans at most 2 blocks.

In the final round, we have all the data required for finding the matches with starting point inside block T^i at node a_i. It suffices to find the hash of each suffix of T^i, i.e.,

$h(T^i[l,\mathsf{S}])$, in which l is the candidate starting point. Consider T^k is the block where the ending point respective to l lies inside, and r is the index of the ending point inside T^k. Then, using the hash of the sequence of block between T^i and T^k, and $h(T^k[1,r])$ (both the hashes has been sent to a_i in the previous steps) one can decide whether the starting point l in block T^i is a match or not. □

Algorithm 11: StringMatching(b)

Data: two array T and P
Result: function $f : \{1,2,\ldots,n-m+1\} \to \{0,1\}$ such that $f(i) = 1$ iff $T[i : i+m-1] = P$.

$f(i) \leftarrow 0 \;\; \forall 1 \le i \le n-m+1$;
Run in parallel: **for** $1 \le i \le m/\mathsf{S}$ **do**
    send $h(P^i)$ to machine d;
Run in parallel: **for** $1 \le i \le n/\mathsf{S}$ **do**
    send $h(T^i)$ to machine c;
    for $1 \le j \le \mathsf{S}$ **do**
        send $h(T^i[1,j])$ to the machine a_k, where the corresponding starting point, i.e., $(i-1)\mathsf{S}+j-m+1$, lies in T^k;
send $h(P) \leftarrow \mathsf{merge}(h(P^1), h(P^2), \ldots, h(P^{m/\mathsf{S}}))$ to each a_i for $1 \le i \le n/\mathsf{S}$;
for $1 \le i \le n/\mathsf{S}$ **do**
    $h(T^1+T^2+\ldots+T^i) \leftarrow \mathsf{merge}(h(T^1+T^2+\ldots+T^{i-1}), h(T^i))$;
    send $h(T^1+T^2+\ldots+T^i)$ to each machine a_j that needs it in the next round;
Run in parallel: **for** $1 \le i \le n/\mathsf{S}$ **do**
    for $1 \le l \le \mathsf{S}$ **do**
        $s \leftarrow (i-1)\mathsf{S}+l$;
        let k be the index of the machine where $s+m-1$ lies in T^k;
        $r \leftarrow s+m-1-(k-1)\mathsf{S}$;
        $\eta = h(T^{i+1}+T^{i+2}+\ldots+T^{k-1})$;
        $h(T[s,s+m-1]) \leftarrow \mathsf{merge}(h(T^i[l,\mathsf{S}]), \eta, h(T^k[1,r]))$;
        if $h(T[s,s+m-1]) = h(P)$ **then**
            $f(s) \leftarrow 1$;

D.4 Character Replace Wildcard '?'

We start this section summarizing the algorithm for string matching with '?' in sequential settings, which require Fast Fourier Transform.

Theorem D.4.1 (Proven in [59]). *Given strings $T \in \Sigma^n$ and $P \in (\Sigma \cup \{?\})^m$, it is possible to find all the substrings of T that match with pattern P in $\widetilde{O}(n+m)$.*

The idea behind Theorem D.4.1 is taking advantage of fast multiplication algorithms, e.g. using Fast Fourier Transform, to find the occurrences of P which contains '?' wildcard characters. Fischer and Paterson [59] proposed the first algorithm for string matching with '?' wildcard. The main idea is to replace each character $c \in \Sigma$ in string T or P with two consecutive numbers mp_c and $1/\mathsf{mp}_c$ and each '?' with two consecutive zeros. If we compute the convolution of T and the reverse of P, we can ensure a match if the convoluted value with respect to a substring of T equals the number non-wildcard characters in P, aka nz_P. This procedure requires a non-negligible precision in float arithmetic operations that adds a $\log(|\Sigma|)$ factor to the order of the algorithm. In the subsequent works, the dependencies on the size of alphabet has been eliminated. [45, 88, 92]

As stated in Theorem D.4.1, we can solve string matching with '?' wildcards in $O(n\log(n))$ using convolution. Convolution can be computed in $O(n\log(n))$ by applying Fast Fourier Transform on both the arrays, performing a point-wise product, and then applying inverse FFT on the result. Therefore, we should implement FFT in a constant round MPC algorithm in order to solve pattern matching with wildcard '?'. Inverse FFT is also possible with the similar approach.

Given an array $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, we want to find the Discrete Fourier Transform of array A. Without loss of generality, we can assume $n = 2^k$, for some k, to avoid the unnecessary complication of prime factor FFT algorithms; it is possible to right-pad by

zeroes otherwise. By Fast Fourier Transform, applying radix-2 Cooley-Tukey algorithm for example, one can compute the Discrete Fourier Transform of A in $O(n\log(n))$ time, which is defined as:

$$a_k^* = \sum_{j=0}^{n-1} a_j \cdot e^{-2\pi i jk/n} \tag{D.1}$$

Where $A^* = \langle a_0^*, a_1^*, \ldots, a_{n-1}^* \rangle$ is the DFT of A. We interchangeably use the alternative notation $W_n = e^{-2\pi i/n}$, by which Equality D.1 becomes

$$a_k^* = \sum_{j=0}^{n-1} a_j \cdot W_n^{jk} \tag{D.2}$$

We state the following theorem regarding the FFT in the MPC model.

Theorem D.4.2. *For any $x \leq 1/2$, a collection of $O(n^x)$ machines each with a memory of size $O(n^{1-x})$ can solve the problem of finding the Discrete Fourier Transform of A with $O(1)$ number of rounds in MPC model. The total running time equals $O(n\log(n))$, which is tight.*

Roughly speaking, our algorithm is an adaptation of Cooley-Tukey algorithm in the MPC model. In Subsection D.4.1, we justify Theorem D.4.2 by giving an overview of the algorithm, and then we show how it results in a $O(1)$-round algorithm for computing the DFT of an array in the MPC model.

Corollary D.4.3. *Given strings $T \in \Sigma^n$ and $P \in (\Sigma \cup \{?\})^m$, it is possible to find all the substrings of T that match with pattern P with a $O(1)$-round MPC algorithm with total runtime and memory of $\widetilde{O}(n+m)$.*

Proof. Build vectors $T^\dagger$ and $P^\dagger$ as following:

$$T^\dagger = \langle \mathsf{mp}_{T_1}, \mathsf{mp}_{T_1}^{-1}, \mathsf{mp}_{T_2}, \mathsf{mp}_{T_2}^{-1}, \ldots, \mathsf{mp}_{T_n}, \mathsf{mp}_{T_n}^{-1} \rangle$$
$$P^\dagger = \langle \mathsf{mp}_{P_1}, \mathsf{mp}_{P_1}^{-1}, \mathsf{mp}_{P_2}, \mathsf{mp}_{P_2}^{-1}, \ldots, \mathsf{mp}_{P_n}, \mathsf{mp}_{P_n}^{-1} \rangle$$

Note that mp_c for all characters $c \in \Sigma$ has a positive integer value, and mp_c^{-1} equals $1/\mathsf{mp}_c$. However, let $\mathsf{mp}_? = \mathsf{mp}_?^{-1} = 0$ for wildcard character '?'. For example, for text "abracadabra" and pattern "a?a", considering mp_c equals the index of each letter in the English alphabet, we will have

$$T^\dagger = \langle 1, \frac{1}{1}, 2, \frac{1}{2}, 18, \frac{1}{18}, 1, \frac{1}{1}, 3, \frac{1}{3}, 1, \frac{1}{1}, 4, \frac{1}{4}, 1, \frac{1}{1}, 2, \frac{1}{2}, 18, \frac{1}{18}, 1, \frac{1}{1} \rangle$$
$$P^\dagger = \langle 1, \frac{1}{1}, 0, 0, 1, \frac{1}{1} \rangle$$

Let $\mathscr{C} = T^\dagger \circledast \mathsf{rev}(P^\dagger)$, where operator $\circledast$ shows the convolution of its two operands. We can observe that for all $1 \leq i \leq n-m+1$ we have

$$\mathscr{C}_{2i+2m-1} = \sum_{j=1}^{2m} T^\dagger_{2(i-1)+j} \cdot P^\dagger_j \tag{D.3}$$

It is clear that if $T[i, i+m-1]$ matches pattern P, then $\mathscr{C}_{2i+2m-1} = 2\mathsf{nz}_P$ since each the wildcard characters add up to 0, and for any other character c, $\mathsf{mp}_c \times 1/\mathsf{mp}_c + \mathsf{mp}_c \times 1/\mathsf{mp}_c = 2$. According to [59], the other side of this expression also holds, i.e., $T[i, i+m-1]$ matches pattern P if $\mathscr{C}_{2i+2m-1} = 2\mathsf{nz}_P$. Therefore, to find that whether a substring $T[i, i+m-1]$ matches pattern P or not, it suffices to

1. Compute $\mathscr{C} = T^{\dagger} \circledast \mathsf{rev}(P^{\dagger})$ utilizing FFT, which can be implemented in $O(1)$ rounds in MPC model as stated in Theorem D.4.2.

2. For all $1 \leq i \leq n-m+1$, substring $T[i, i+m-1]$ matches pattern P iff $\mathscr{C}_{2i+2m-1} = 2\mathsf{nz}_P$.

□

D.4.1 $O(1)$-round algorithm for FFT

To find the $O(1)$ round algorithm in MPC model, we first review how one can find the Discrete Fourier Transform in $O(n\log(n))$ time. The following recurrence helps us to solve the problem using a divide and conquer algorithm. [50] Let $\Psi[2k] = \langle a_0, a_2, \ldots, a_{n-2}\rangle$, i.e., an array of the even-indexed elements of A, and $\Psi[2k+1] = \langle a_1, a_3, \ldots, a_{n-1}\rangle$, an array of the odd-indexed elements of A likewise. If $\Psi[2k]^* = \langle \psi[2k]^*_0, \psi[2k]^*_1, \ldots, \psi[2k]^*_{n/2-1}\rangle$ and $\Psi[2k+1]^* = \langle \psi[2k+1]^*_0, \psi[2k+1]^*_1, \ldots, \psi[2k+1]^*_{n/2-1}\rangle$ show the DFT of $\Psi[2k]$ and $\Psi[2k+1]$ respectively, then for all $0 \leq j < n/2$

$$\begin{cases} a^*_j & = \psi[2k]^*_j + W_n^j \psi[2k+1]^*_j \\ a^*_{j+n/2} & = \psi[2k]^*_j - W_n^j \psi[2k+1]^*_j \end{cases} \tag{D.4}$$

Intuitively, the recurrence decomposes A into two smaller arrays of size $n/2$, $\Psi[2k]$ and $\Psi[2k+1]$, and represents the DFT of A based on $\Psi[2k]^*$ and $\Psi[2k+1]^*$. Accordingly, we can solve two smaller DFT problems, each with size $n/2$, and then merge them in $O(n)$ time to find A^*. Therefore, we can find A^* in $O(n\log(n))$ time since the depth of the recurrence is $O(\log(n))$. We cannot find a constant round MPC algorithm exclusively using this recurrence. To extend the recursion, we can define $\Psi[pk+q]$ as the following,

$$\Psi[pk+q] = \langle a_q, a_{p+q}, \ldots, a_{(l-1)p+q} \rangle \quad \forall\, 0 \le q < p \le n$$

Where $l = \lfloor \frac{n-1-q}{p} \rfloor + 1$ is the number of the indices whose reminder is q in division by p. $\Psi[pk+q]$ contains all the elements of A whose index is q modulo p. Note that the notation of $\Psi[pk+q]$ assumes a universal quantification on all $0 \le k \le n/p$. We also show the DFT of $\Psi[pk+q]$ by

$$\Psi[pk+q]^* = \langle \psi[pk+q]_0^*, \psi[pk+q]_1^*, \ldots, \psi[pk+q]_{l-1}^* \rangle$$

We can observe that for all $0 \le b \le \log_2(n)$ and $0 \le q < 2^{b+1}$, assuming $p = 2^b$, the following counterpart of Recurrence D.4 holds for all $0 \le j < l$, where l is the length of $\Psi[2^b k+q]$ which equals $\log(n) - b$:

$$\begin{cases} \psi[2^b k+q]_j^* & = \psi[2^{b+1}k+q]_j^* + W_l^j \psi[2^{b+1}k+2^b+q]_j^* \\ \psi[2^b k+q]_{j+l/2}^* & = \psi[2^{b+1}k+q]_j^* - W_l^j \psi[2^{b+1}k+2^b+q]_j^* \end{cases} \tag{D.5}$$

Since $\Psi[2^{b+1}k+q]$ and $\Psi[2^{b+1}k+2^b+q]$ decompose $\Psi[2^b k+q]$ into the even-indexed and odd-indexed elements, Recurrence D.5 is implied by plugging in $\Psi[2^b k+q]$ as A in Recurrence D.4. The following Lemmas (D.4.4 and D.4.5) point out two properties regarding FFT in the MPC model. These properties immediately give us an $O(\log(n))$-round algorithm for the FFT problem.

Lemma D.4.4. *For all* $\log(\mathsf{M}) \le b \le \log(n)$ *and* $0 \le q < 2^b$*, the value of* $\Psi[2^b k+q]^*$ *could be computed in a single machine with memory of* S.

Proof. If all of the entries of $\Psi[2^b k+q]$ exist in the memory of a single machine, which is viable since $b \ge \log(\mathsf{M})$, then we can compute $\Psi[2^b k+q]^*$ as following:

$$\Psi[2^b k+q]^* = \mathsf{fft}(\Psi[2^b k+q])$$

□

In many implementations of Fast Fourier Transform, bit-reversal technique is used to facilitate the algorithm from various perspectives, to achieve an in-place FFT algorithm for example. Bit-reversal technique reorders the initial array based on the reverse binary representation of the indices; Considering $(10100)_2$ as the sort key for a_5, $5 = (00101)_2$, when $n = 32$ for example. In other words, bit reversal technique permutes the elements of A by a permutation π such that $\pi(i) = \mathsf{rev}(i)$, where $\mathsf{rev}(i)$ is the reverse binary presentation of i. Applying bit reversal technique guarantees the locality of data, especially in the first $\log(\mathsf{S})$ stages.

Lemma D.4.5. *The bit-reversal operation makes a permutation of input entries which if we it split into 2^b continuous chunks of equal size, then for all $0 \leq b \leq \log(n)$, $\Psi[2^b k + \mathsf{rev}(q)]$ would the q-th chunk.*

Proof. Notice that the bit-reversal permutation is actually sorting the entries based on the reverse binary representation of the index of each entry. Therefore, all indices with the same low b bits form a continuous interval of size $2^{\log(n)-b}$ in the bit-reversal permutation. Thus, $\Psi[2^b k + \mathsf{rev}(q)]$ which all of its members have the same low b bits, $\mathsf{rev}(q)$. Moreover, it is the q-th chunk in the bit-reversal permutation, because $\mathsf{rev}(q)$ ranks q-th in the bit-reversal ordering of all non-negative integers less than 2^b. □

Definition D.1. For all $0 \leq q < \mathsf{M}$, Let $\Phi_q = \Psi[\mathsf{M}k + \mathsf{rev}(q)]$. ($\Phi_q$ shows the input of the q-th machine after performing the bit-reversal permutation, according to Lemma D.4.5) We also show the DFT of Φ_q by Φ_q^+, instead of Φ_q^*. The j-th element of Φ_q (Φ_q^+) is denoted by $\phi_{q,j}$. ($\phi_{q,j}^+$)

Utilizing Lemma D.4.4 and Lemma D.4.5, we can achieve a $O(\log(n))$-round MPC algorithm which merges all Φ_q^+s in $\log(\mathsf{M})$ steps. At the first step, we apply bit-reversal permutation in order to gather each Φ_q in a single machine. Applying bit-reversal permutation does not have any special communication overhead, as we are just transferring the elements. Then we compute Φ_q^+ for all $q \in \mathbb{Z}_m$. Afterwards, in the l-th step, for $0 \leq l < \log(m)$, we can merge $2^{\log(m)-l}$ blocks of size $2^{\log(\mathsf{S})+l}$ using equation D.5 in pairs.

It seems that $\log(n)$ MPC rounds is required because we need to merge Φ_q^+s which are distributed in various machines, and according to Equality D.1, each a_k^* depends on every Φ_q^+ for $q \in \mathbb{Z}_m$. However, we can exploit the nice properties of the graph of dependencies of a_i^*s to Φ_q^+s, which is also called the *Butterfly Graph*, to decompose these dependencies efficiently. An example of this dependencies graph is demonstrated in Figure D.1. We can concentrate these dependencies in single machines by gathering $\phi_{q,j}^+$s with the same j. Figure D.1 demonstrates why it suffices to have $\phi_{q,j}^+$s with the same j stored in a single machine. In this figure, S and M are equal to 4, and the entries that should be stored in the same machine are colored with the same color. At the first step, the entries are distributed in bit-reversal permutation, and continuous chunks are stored in each machine. However, in the next step, we can see for example a_5^*, which is colored green, depends only on the green entries, which all have the same $j = 1$.

Definition D.2. Let $\widetilde{\phi}_{q,j}^+ = W_n^{qj}\phi_{q,j}^+$. These coefficients are called twiddle factors, which allows us to express a_k^* based on the FFT of $\widetilde{\phi}_{q,j}^+$s. Let Φ_j^* be the DFT of vector $\langle \widetilde{\phi}_{0,j}^+, \widetilde{\phi}_{1,j}^+, \ldots, \widetilde{\phi}_{\mathsf{M}-1,j}^+ \rangle$. We will show in Lemma D.4.6 that $a_{q\mathsf{S}+j}^*$ equals $\phi_{j,q}^*$, which is the j-th element of Φ_j^*.

Lemma D.4.6. $a_{q\mathsf{S}+j}^* = \phi_{j,q}^*$.

Proof. We know that a_k^* equals a weighted sum of all a_j where weights are the n-th roots of unity to the power of k. i.e.,

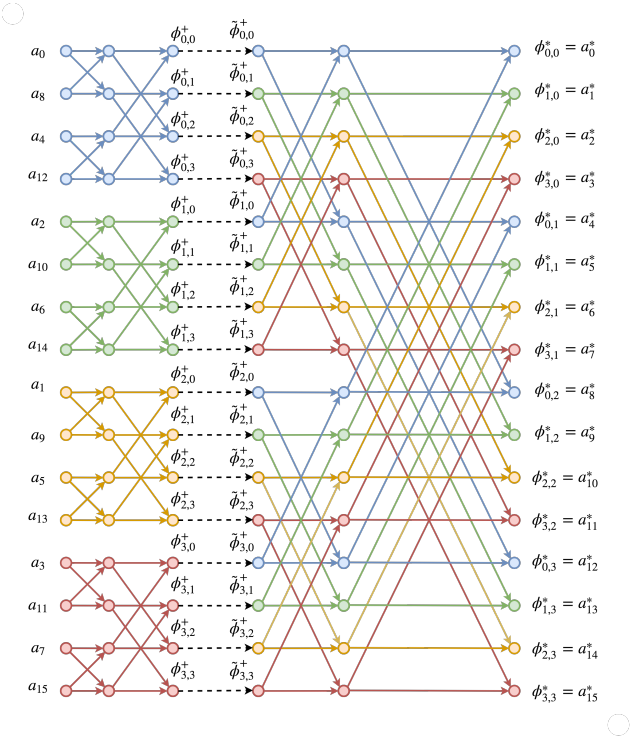


Figure D.1: The dependency graph of the FFT recurrence in MPC model. Different colors represent different machines, in a setting with 4 machines with memory of 4 each. It could be observed that the dependency graph in each machine at the first stage is isomorphic to the one of the second stage.

$$a_k^* = \sum_{j=0}^{n-1} W_n^{jk} a_j$$

We break down the summation into M smaller summations of size S such that each smaller summation represent one Φ_i for some $0 \leq i \leq \mathsf{M}-1$. Moreover, we show k by $q\mathsf{S}+j$, where $0 \leq j \leq \mathsf{S}-1$. Hence, we can show the following.

$$a^*_{q\mathsf{S}+j} = \sum_{z=0}^{\mathsf{M}-1}\sum_{l=0}^{\mathsf{S}-1} W_n^{(l\mathsf{M}+z)(q\mathsf{S}+j)} a_{l\mathsf{M}+z} \tag{D.6}$$

$$= \sum_{z=0}^{\mathsf{M}-1}\sum_{l=0}^{\mathsf{S}-1} W_n^{l\mathsf{M}q\mathsf{S}} W_n^{l\mathsf{M}j} W_n^{zq\mathsf{S}} W_n^{zj} a_{l\mathsf{M}+z} \tag{D.7}$$

$$= \sum_{z=0}^{\mathsf{M}-1}\sum_{l=0}^{\mathsf{S}-1} W_{\mathsf{S}}^{lj} W_{\mathsf{M}}^{zq} W_n^{zj} a_{l\mathsf{M}+z} \tag{D.8}$$

$$= \sum_{z=0}^{\mathsf{M}-1} W_{\mathsf{M}}^{zq} W_n^{zj} \sum_{l=0}^{\mathsf{S}-1} W_{\mathsf{S}}^{lj} a_{l\mathsf{M}+z} \tag{D.9}$$

$$= \sum_{z=0}^{\mathsf{M}-1} W_{\mathsf{M}}^{zq} W_n^{zj} \phi^+_{z,j} \tag{D.10}$$

$$= \sum_{z=0}^{\mathsf{M}-1} W_{\mathsf{M}}^{zq} \widetilde{\phi}^+_{z,j} = \phi^*_{j,q} \tag{D.11}$$

Equality D.8 is implied from the fact that $W_{kn}^k = W_n$, and therefore, $W_n^{l\mathsf{M}p\mathsf{S}} = W_n^{lpn} = W_1^{lp} = 1$, $W_n^{l\mathsf{M}j} = W_{\mathsf{S}}^{lj}$, and $W_n^{zp\mathsf{S}} = W_{\mathsf{M}}^{zp}$. □

Utilizing Lemma D.4.6, we can provide an algorithm which solves the FFT problem in $O(1)$ MPC rounds, because we already know how to compute $\phi^*_{j,q}$s. Algorithm 12 shows the algorithm in details. It can be observed that our algorithm is analogous to Cooley-Tukey algorithm with radix n^x.

Theorem D.4.2. At the first step of the algorithm, we apply the bit-reversal permutation in the input, thereby grouping the members of Φ_q in each of M machines. Applying a permutation imposes a communication overhead of $O(\mathsf{S})$ for each machine. Now, the q-th machine can compute Φ^+_q and $\widetilde{\Phi}^+_j$ standalone in $\widetilde{O}(n)$ time as following:

- $\Phi^+_q = \mathsf{fft}(\Phi_q)$
- $\widetilde{\Phi}^+_{q,j} = W_n^{qj} \Phi^+_{q,j}$

Algorithm 12: mpc-fft(b)

Data: an array a.

Result: array a^* containing the DFT of a.

permute a such that members of Φ_q be together in a single machine for all $q \in \mathbb{Z}_{\mathsf{M}}$.

Run in parallel: **for** $q \in \mathbb{Z}_{\mathsf{M}}$ **do**

  $\Phi_q^+ \leftarrow \mathsf{fft}(\Phi_q)$;

  $\widetilde{\phi}_{q,j}^+ \leftarrow W_n^{qj}\phi_{q,j}^+ \quad \forall\, j \in \mathbb{Z}_{\mathsf{S}}$;

distribute $\widetilde{\phi}_{q,j}^+$ into different machines such that entries with same j be in a same machine.

Run in parallel: **for** $j \in \mathbb{Z}_{\mathsf{S}}$ **do**

  $\Phi_j^* \leftarrow \mathsf{fft}(\langle \widetilde{\phi}_{0,j}^+, \widetilde{\phi}_{1,j}^+, \ldots, \widetilde{\phi}_{\mathsf{M}-1,j}^+ \rangle)$;

permute all $\phi_{j,q}^*$ $(= a_{q\mathsf{S}+j}^*)$ in a way that each of them retain its correct position.

In the next round, we distribute $\widetilde{\Phi}_{q,j}^+$s with the same j in the same machines. Note that $\widetilde{\Phi}_{q,j}^+$s with the same j fit into a single machine since $x \leq 1/2$. We even might need to fit multiple groups of $\widetilde{\Phi}_{q,j}^+$s into a single machine, which causes no problem. By the way, we can compute Φ_j^*s in this stage in $\widetilde{O}(n)$ time. Afterwards it suffices restore appropriate a_k^*s to their original position, where for $k = qS + j$, $a_k^* = \phi_{j,q}^*$. □

Observation D.4.1. *Although Algorithm 12 requires* 3 *rounds of communication for computing DFT in the MPC model, the convolution of two arrays can be computed in* 4 *communication rounds.*

Proof. Since we need to apply FFT twice for computing the convolution, we can find the convolution in 6 communication rounds. However, since the first round applies a permutation on the elements, and the last round applies the inverse permutation, we can ignore the last round of computing the FFT, along with the first round of computing the inverse FFT. The point-wise product operation which is needed two be done between two FFTs allows us to do so, as it is independent of the order of the arrays. □

D.5 Character Repetition Wildcard '+'

The character repetition wildcard, shown by '+', allows expressing "an arbitrary number of a single character" within the pattern. An occurrence of '+' which is immediately after a character c, matches arbitrary repetition of character c. For example, text "bookkeeper" has a substring that matches pattern "oo+k+ee+", but neither of its substrings won't match pattern "oo+kee+". In Subsection D.5.1, we reduce pattern matching with wildcard '+' to greater-than matching in $O(1)$ MPC rounds. The greater-than matching problem is defined with further details in Problem D.5.1. Afterwards, we show a reduction of greater-than matching from subset matching problem, explained in D.5.2. Then, by showing subset matching could be implemented in $O(1)$ MPC rounds, we propose an $O(1)$-round MPC algorithm for string matching with wildcard '+' in Theorem D.5.4.

D.5.1 Relation to greater-than matching

Problem D.5.1. *Greater-than matching:* Given two arrays T and P, of length n and m respectively, find all indices $1 \leq i \leq n-m+1$ such that the continuous subarray of T starting from i and of length m is element-wise greater than P, i.e., $T_{i+j-1} \geq P_j \;\; \forall 1 \leq j \leq m$.

To reduce pattern matching with wildcard '+' to greater-than matching, we perform run-length encoding D.3 on both the text and the pattern. This way, we have a sequence of letters each along with a number showing the number of its repetitions, or a lower-bound restricting the number of repetitions in case we have a '+' wildcard. Subsequently, we can solve the problem for letters and numbers separately, and merge the result afterwards. Note that pattern P matches a substring of T if and only if the both the respective letters and the respective repetition restrictions match. We also show we can find the run-length encoding

of a string in $O(1)$ MPC rounds in Observation D.5.1.

Definition D.3. For an arbitrary string s, let s° be the run-length encoding of s, computed in the following way:

- Ignoring '+' characters, decompose string s into maximal blocks consisting of the same character representing by pairs $\langle c_i, \mathsf{cnt}_i \rangle$ which show a block of cnt_i repetitions of character c_i.
- If a '+' character is located immediately after a block or within a block, that block becomes a wildcard block, and represented as $\langle c_i, \mathsf{cnt}_i+ \rangle$, where cnt_i is still the number of the occurrences of character c_i in the block.
- s° equals the list of these pairs $\langle c_i, \mathsf{cnt}_i \rangle$ or $\langle c_i, \mathsf{cnt}_i+ \rangle$, concatenated in a way that preserves the original ordering of the string.

For example, for $T =$ "bookkeeper" and $P =$ "o+o+k+ee+p",

$$T^\circ = \langle \langle \mathsf{b}, 1 \rangle, \langle \mathsf{o}, 2 \rangle, \langle \mathsf{k}, 2 \rangle, \langle \mathsf{e}, 2 \rangle, \langle \mathsf{p}, 1 \rangle, \langle \mathsf{e}, 1 \rangle, \langle \mathsf{r}, 1 \rangle \rangle$$
$$P^\circ = \langle \langle \mathsf{o}, 2+ \rangle, \langle \mathsf{k}, 1+ \rangle, \langle \mathsf{e}, 2+ \rangle, \langle \mathsf{p}, 1 \rangle \rangle$$

We alternatively show the compressed string as a string, for example,

- $T^\circ =$ "b[1]o[2]k[2]e[2]p[1]e[1]r[1]".
- $P^\circ =$ "o[2+]k[1+]e[2+]p[1]".

Observation D.5.1. *We can perform run-length encoding in $O(1)$ MPC rounds.*

Proof. Suppose string s is partitioned among M machines such that i'th machine contains s_i for $1 \leq i \leq \mathsf{M}$. Run-length encoding of each s_i could be computed separately inside each machine. Let

$$s_i^\circ = \langle \langle c_{i,1}, cnt_{i,1} \rangle, \langle c_{i,2}, cnt_{i,2} \rangle, \ldots, \langle c_{i,l_i}, cnt_{i,l_i} \rangle \rangle$$

be the run-length encoding of s_i, where l_i is its length. Then in the next round, we can merge $\langle c_{i,l_i}, cnt_{i,l_i} \rangle$ and $\langle c_{i+1,1}, cnt_{i+1,1} \rangle$ if $c_{i,l_i} = c_{i+1,1}$ for all $1 \leq i \leq \mathsf{M}-1$, and we will end up with s° if we concatenate all s_i°s. □

As we mentioned before, in order to reduce from greater-than matching, it is possible to divide the problem into two parts: matching the letters, and ensuring whether the repetition constraints are hold, and solve each part separately. The former is a simple string matching problem, but the latter requires could be solved with greater-than matching. Formally, we need to find all indices $1 \leq i \leq n-m+1$ such that for every $1 \leq j \leq m$, the following constraints holds:

1. $T^\circ_{i+j-1} = \langle c_j, x \rangle$ for some $x \geq \mathsf{cnt}_j$ if $P^\circ_j = \langle c_j, \mathsf{cnt}_j+ \rangle$.
2. $T^\circ_{i+j-1} = \langle c_j, \mathsf{cnt}_j \rangle$ if $P^\circ_j = \langle c_j, \mathsf{cnt}_j \rangle$ and $2 \leq j \leq m-1$.
3. $T^\circ_{i+j-1} = \langle c_j, x \rangle$ for some $x \geq \mathsf{cnt}_j$ if $P^\circ_j = \langle c_j, \mathsf{cnt}_j \rangle$ and $j \in \{1, m\}$.

Constraint 3 needs to be considered to allow the substring in the original text T starts and ends in the middle of a block of c_1 or c_m letters. By considering only those substrings which their letters match, we can get rid of letter constraints. Also to ensure constraint 2, we can perform a wildcard '?' pattern matching by replacing each cnt_j+ and also cnt_1 and cnt_m by a '?' wildcard, and keep only the numbers that constraint 2 checks. Thus, if a substring $T^\circ_{i,i+m-1}$ matches according to this wildcard '?' pattern matching, what remains is a greater-than matching, as we only need to check constraints 1 and 3, and we can replace numbers we already checked for constraint 2 by some 0's.

Observation D.5.2. *We can reduce pattern matching with wildcard '+' from greater-than matching in $O(1)$ MPC rounds.*

Using Observation D.5.1, it only remains to perform $O(1)$ wildcard '?' matchings to obtain an instance of greater-than matching that already satisfies constraint 2, as well as letter constraints. Thus, Observation D.5.2 is implied.

D.5.2 Reduction from subset matching

Problem D.5.2. *Subset Matching:* Given T, a vector of n subsets of the alphabet Σ, and P, a vector of m subsets in the same format as T, find all occurrences of P in T. P is occurred at position i in T if for every $1 \leq j \leq |P|$, $P_j \subseteq T_{i+j-1}$.

The subset matching problem is a variation of pattern matching where pattern matches text if each of the pattern entries is a subset of the respective entries in text. Keeping this in mind, we use subset matching to solve greater-than matching in $O(1)$ MPC rounds, and thereby achieving a $O(1)$-round solution for the problem of pattern matching with '+' wildcard character using the subset matching algorithm proposed by Cole and Hariharan [46].

Observation D.5.3. *We can reduce greater-than matching to subset matching in $O(1)$ MPC rounds with total runtime and total memory of $O(Q)$, where Q is the sum of all T_i's, i.e. $Q = \sum_{i=1}^{m} T_i$.*

The idea behind Observation D.5.3 is to have a subset $\{0,1,2,\ldots,T_i\}$ instead of each entry of T_i, and a subset $\{P_i\}$ instead of each P_i. This way if T matches P in a position $1 \leq i \leq n-m+1$, then we have $P_j \in \{0,1,2,\ldots,T_{i+j-1}\}$ since $P_j \leq T_{i+j-1}$ for all $1 \leq j \leq m$. This way, we can solve pattern matching with wildcard '+' in $O(1)$ MPC rounds if subset matching could be solved in $O(1)$ MPC rounds. In the following, Lemma D.5.3 shows it is possible to solve subset matching w.h.p in a constant number of MPC rounds. The algorithm utilizes $O(log(n))$ invocations of wildcard '?' matching.

Lemma D.5.3. *We can solve subset matching in $O(1)$ MPC rounds with total runtime and total memory of $\widetilde{O}(Q)$, where Q is the sum of the size all T_i's, i.e. $Q = \sum_{i=1}^{n} |T_i|$.*

Proof. First, consider solving an instance of the greater than matching problem in $O(1)$ MPC rounds. We partition the entries inside all T^i's into Σ sequences $T^{i,1}, T^{i,2}, \ldots, T^{i,\Sigma}$ inside each machine, where $T^{i,j} = \{k \mid j \in T_{iS+k}\}$. We just create a subset of these sets which are not empty, i.e., $\mathscr{T}_i = \{T^{i,j} \mid |T^{i,j}| > 0\}$, so that $\mathscr{T}_i$ fits inside the memory of each machine. $\mathscr{P}_i$ also can be defined similarly. We can sort all the union of all $\mathscr{T}_i$'s and $\mathscr{P}_i$'s in a non-decreasing order of j and breaking ties using i in $O(1)$ MPC rounds [76].

This way, we end up with a sparse wildcard matching for each character in Σ, because we want to check in which substrings of T each occurrence of character $j \in \Sigma$ in P is contained in the corresponding T_i. We can put a '1' instead of each occurrence of j in P, a '?' instead of each j in T, and a '0' in all other places since we are considering a greater than matching instance. Using the similar algorithm as in [46], we can easily reduce each instance of sparse wildcard matching to $O(\log(k))$ normal wildcard matching with size of $O(k)$ in $O(1)$ MPC rounds, where k is the number of non-zero entries. Using Corollary D.4.3, we can give a $O(1)$ round MPC algorithm for subset matching when the input is an instance of greater than matching. We can easily extend these techniques to solve subset matching algorithm in $O(1)$ MPC rounds, as Cole and Hariharan [46] showed it is analogous to sparse wildcard matching. □

Theorem D.5.4. *There exists an MPC algorithm that solves string matching with '+' wildcard in constant rounds. The total memory and the total running time of the algorithm are $\widetilde{O}(n)$.*

Proof. We can also simplify the algorithm for pattern matching with wildcard '+', by exploiting subset matching flexibility. We can also use subset matching to verify constraint 2 of greater-than matching reduction (instead of wildcard '?' pattern matching), as well as

letter constraint (instead of regular string matching with no wildcard). We define T' and P' as follows:

$$T'_i = \bigcup_{j=1}^{cnt_i} \{\langle c_i, j+\rangle\} \cup \{\langle c_i, cnt_i\rangle\} \qquad \text{if } T_i^\circ = \langle c_i, cnt_i\rangle \tag{D.12}$$

$$P'_i = \{\langle c_i, cnt_i\rangle\} \qquad \text{if } P_i^\circ = \langle c_i, cnt_i\rangle \tag{D.13}$$

$$P'_i = \{\langle c_i, cnt_i+\rangle\} \qquad \text{if } P_i^\circ = \langle c_i, cnt_i+\rangle \tag{D.14}$$

It could easily be observed that subset matching of T' and P' is equivalent to pattern matching with wildcard '+' of T and P, and also this simpler reduction is straight-forward to implement in $O(1)$ MPC rounds. In addition, the total runtime and memory of this subset matching which is $\widetilde{O}(Q)$ is equal to $\widetilde{O}(n)$ in the original input. Note that T' is resulted form T° whose sum of its numbers, that each of them shows the repetitions of the respective letter, equals n. □

D.6 String Replace Wildcard '*'

In this section we consider the string matching problem with wildcard '*'.

Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{'*'}\}^m$, we say a substring of p is a subpattern if it is a maximal substring not containing '*'. We present the following results in this section:

1. A sequential algorithm for string matching with wildcard '*' in time $O(n+m)$.

2. An MPC algorithm for string matching with wildcard '*' in $O(\log n)$ rounds using $O(n^x)$ machines if the length of pattern p is at most $O(n^{1-x})$.

3. An MPC algorithm for string matching with wildcard '*' in $O(\log n)$ rounds using $O(n^x)$ machines if all the subpatterns of p are not prefix of each other.

D.6.1 Linear time sequential algorithm

Observation D.6.1. *Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{`*'}\}^m$, there is a sequential algorithm to decide if s matches with pattern p in time $\tilde{O}(n+m)$.*

Let the subpatterns of p to be $P_1, P_2, \ldots, P_w$. Our sequential algorithm is StringMatchingWithStar(a).

Algorithm 13: StringMatchingWithStar(a)

Data: $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{`*'}\}^m$.
Result: Yes or No.

```
Set i ← 1;
for j = 1, 2, ..., w and i ≤ n do
    Run KMP for the string s[i, n] and pattern P_j;
    If KMP fails, then Return No;
    Let i' be the position satisfying s[i', i' + |P_j| − 1] = P_j obtained by KMP;
    Set i ← i' + |P_j|;
Return Yes.
```

D.6.2 MPC algorithm for small subpattern

Theorem D.6.1. *Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{`*'}\}^m$ for $m = O(n^{1-x})$, there is an MPC algorithm to find the solve the string matching problem in $O(\log n)$ rounds using $O(n^x)$ machines.*

We assume string s is partitioned into $s_1, s_2, \ldots, s_t$ for some $t = O(n^x)$ such that every s_i has length at most $O(n^{1-x})$. We say string s is an exact matching of p if there is a partition of s into $|p|$ (possibly empty) substrings such that if $p[i]$ is not '*', then i-th substring is same to $p[i]$.

Given indices $i, j \in [t]$ of string s and position k of pattern p, let $f(i, j, k)$ be the largest position of p such that the concatenation of $s_i, s_{i+1}, \ldots, s_j$ matches pattern $p[k, f(i, j, k)]$.

To illustrate our idea, we need the following definitions:

1. $g(k)$ for position k of p: the largest position which is smaller than or equal to k such that $p[k]$ is '*'.

2. $h(k)$ for position k of p such that $p[k] \neq$ '*': the smallest integer r such that $p[g(k)+1, k-r]$ is a prefix of $p[g(k)+1, k]$.

Consider the following equation for an arbitrary $i \leq i' < j$ (let $\beta = f(i, i', k)$)

- if $p[\beta] =$ '*':

$$f(i,j,k) = f(i'+1, j, \beta)$$

- if $p[\beta] \neq$ '*':

$$f(i,j,k) = \max \begin{cases} f(i'+1, j, g(\beta)), \\ \max_{0 \leq \ell \leq \lfloor \frac{\beta - h(\beta)}{g(\beta)} \rfloor} \{f(i'+1, j, \beta - \ell \cdot h(\beta))\} \end{cases} \tag{D.15}$$

Our algorithm is StringMatchingWithStar(b).

of D.6.1. We show that Eq. D.15 correctly computes $f(i,j,k)$ for an arbitrary $i \leq i' < j$. Then the theorem follows from the description of StringMatchingWithStar(b), Eq. D.15 and Observation D.6.1.

Let α be the largest position of p such that there is an exact matching of the concatenation of $s_i, \ldots, s_j$ and $p[k, \alpha]$. If $f(i, i', \cdot)$ and $f(i'+1, j, \cdot)$ are correct, and $f(i,j,k)$ is computed by Eq. D.15, then $f(i,j,k) \leq \alpha$, since Eq. D.15 implies a feasible exact matching.

Now we show that $f(i,j,k) \geq \alpha$ if $f(i,j,k)$ is computed by Eq. D.15. There is an exact matching of the concatenation of $s_i, \ldots, s_j$ and $p[k, \alpha]$. Let γ be the position of pattern p such that the last symbol of $s_{i'}$ is matched to $p[\gamma]$. By the definition of function f, $\gamma \leq \beta$.

Consider the case of $p[\gamma] =$'*' or $p[\gamma] \neq$'*' but $p[\gamma] =$'*'. We have $g(\beta) \geq \gamma$ and $f(i'+1, j, \gamma) = \alpha$. By the monotone property of f, we have $f(i,j,k) \geq f(i'+1, j, g(\beta)) \geq f(i'+1, j, \gamma) = \alpha$.

Consider the case of $p[\gamma] \neq$'*' and $p[\beta] \neq$'*'. If γ and β are in different subpatterns, then using above argument, we have $f(i,j,k) \geq \alpha$. Otherwise, $p[h(\beta)+1, \gamma]$ is a suffix of $p[h(\beta)+1, \beta]$, which implies that there is a non-negative integer ℓ such that $\gamma = \beta - \ell \cdot h(\beta)$. Hence, $f(i,j,k) \geq \alpha$. □

Algorithm 14: StringMatchingWithStar(b)

Data: two array s and p.
Result: Yes or No.

1 Distribute $s_1, s_2, \ldots, s_t$ to distinct machines, and distribute p to every machine;
2 Compute $f(i,i,k)$ for all the k on the machine containing s_i by algorithm StringMatchingWithStar(a) **in parallel**;
3 **for** $m = 1, 2, \ldots, \lceil \log_2 t \rceil$ **do**
4 For every $i \geq 1$, put $f(i, i+2^{m-1}-1, k)$ and $f(i+2^{m-1}, i+2^m-1, k)$ into same machine for all the k **in parallel**;
5 For every $i \geq 1$, compute $f(i, i+2^m-1, k)$ for all the k by Equation D.15 with $j = i+2^m-1$ and $i' = i+2^{m-1}-1$ **in parallel**;
6 Return Yes if $f(1,t,1) = w$, otherwise return No.

D.6.3 MPC algorithm for non-prefix subpatterns

Theorem D.6.2. *Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup$ '*'$\}^m$ such that subpatterns are not a prefix of each other, there is an MPC algorithm to find the solve the string matching problem in $O(\log n)$ rounds using $O(n^x)$ machines.*

Proof. We show that algorithm StringMatchingWithStar(c) solves the problem.

We first prove the correctness of the algorithm. Since all the subpatterns are not a prefix of each other, for every position i of string s, there is at most one subpattern P_u such that P_u is a prefix of $s[i,n]$. On the other hand, if $s[i,j]$ is not a prefix of any subpattern, then

$h(s[i,j])$ does not equal to any of the hash value obtained in Step 1, otherwise, $h(s[i,j])$ is equal to some hash value obtained in Step 1. Hence, for every $i \in [n]$, the "for" loop of Step 2 finds the subpattern P_u such that P_u is a prefix of $s[i,n]$ by binary search if P_u exists.

If string s matches pattern p, then any set of positions $a_1, a_2, \ldots, a_u$ with the following two conditions

1. P_i is a prefix of $s[a_i, n]$ for every $i \in [w]$.

2. $a_i + |P_i| \leq a_{i+1}$ for every $i \in [w-1]$.

corresponds to a matching between s and p. Hence, string s matches pattern p if and only if v_0 and v_{m+1} are connected in the constructed graph of Step 15 and 16.

Now we consider the number of MPC rounds required. Using the argument of Section D.6.2, computing hash of all the prefixes of every subpattern or a set of n substrings of s needs constant MPC rounds. Hence Step 1 needs constant rounds, and the "for" loop of Step 2 needs $O(\log n)$ rounds. Step 15 naturally needs a single round. Step 16 can be done in $O(\log n)$ rounds by sorting all the $(i, f(i))$ pairs according to the f function values and selecting j such that $j \geq i + |P_{f(i)}|$ and $f(j) = f(i) + 1$ for all the pairs $(i, f(i))$. The graph connectivity needs $O(\log n)$ rounds. □

Algorithm 15: StringMatchingWithStar(c)

Data: two array s and p.
Result: Yes or No.

For each subpattern P_i **in parallel** compute the hash value of every prefix of p_i;
for *each position i of string s* ***in parallel*** **do**
    Set $j = 0$ and $k = n$ initially;
    while $j < k$ **do**
        Let $\ell = \lceil (j+k)/2 \rceil$;
        Compute the hash $h(s[i, i+\ell])$;
        if *there is a hash obtained in step 2 same to* $h(s[i, i+\ell])$ **then**
            Set $j \leftarrow \ell$;
        else
            Set $k \leftarrow \ell - 1$;
    if *there is a subpattern* p_u *same to* $s[i, i+j]$ **then**
        Set $f(i) \leftarrow u$;
    else
        Set $f(i) \rightarrow 0$;
Construct an empty graph **in parallel** with vertices $v_0, v_1, \ldots, v_{n+1}$. Add edge (v_0, v_s) and (v_t, v_{n+1}) where s is the smallest integer such that $f(s) = 1$, t is the largest integer such that $f(t) = w$;
For every $i \in [n]$ such that $f(i) > 0$, **in parallel** add edge (v_i, v_j) to the graph where j is the smallest integer such that $j \geq i + |P_{f(i)}|$ and $f(j) = f(i) + 1$;
Run graph connectivity algorithm on the graph constructed. return Yes if v_0 and v_{n+1} are in the same connected component of the graph, otherwise return No ;

Appendix E: Suffix Tree

E.1 Introduction

Perhaps the earliest questions that were studied in computer science are 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, and LPS 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 tools for solving the location-sensitive problems. These algorithms 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 and well-studied problem in the computer science literature. The concept was first introduced by Weiner [134], and McCreight (JACM, 1976) [110] showed that it is possible to create a *compressed* suffix tree of total size $O(n)$. Later, Ukkonen [132] 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* [78]. Two particular

applications of interest [62] that we discuss at length in this paper are:

1 Longest Palindrome Substring (LPS), and

2 Longest Common Substrings (LCS).

Designing algorithms which are sublinear in space is crucial when the input data 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 social network graphs to DNA sequences, 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 which 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 [95], and enhanced in subsequent works [6, 22, 76], 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 $0 < \varepsilon < 1$, each machine has $O(n^{1-\varepsilon})$ 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 $O(n^{1-\varepsilon})$ is larger than the number of machines $O(n^{\varepsilon})$, and so, we restrict ε to $0 < \varepsilon \leq .5$. This restriction is well-motivated for the MPC model since often in practice, 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. 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 with tight $O(n)$ total memory for LPS and LCS by applying a technical analysis of string periodicity in each problem.

In one of the final sections of the paper, we also discuss an improved algorithm for *suffix tree* construction in the stronger *Adaptive* MPC (AMPC) model recently proposed in [29]. In the AMPC model, 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].

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 [11, 12, 24, 27, 31, 49, 66, 69]. 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 [80] 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 E.1.3). Furthermore, our methods in this paper can be applied to the string matching problem (without wildcards) as an application of suffix trees [110].

PRAM solutions to these problems also are heavily related to our work. Apostolico, Breslauer and Galil [9] 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 [87] build a suffix tree from a simpler data structure, the suffix array, in PRAM in tight $\tilde{O}(n)$ total memory with $O(\log n)$

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

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 [106] which solves the *longest common subsequence* problem and [57] which solves suffix tree construction for binary strings, both in the PRAM model as well.

E.1.1 Problem Definition

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

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* $s[i:n)$ $(0 \leq i < n)$, and the concatenation of the labels in the path from the root to this leaf results in suffix $s[i:n)$. See Figure E.1 for an example for string $s =$ "bobocel". A *compressed* suffix tree of string s is a minimal suffix tree that does not have any vertex with degree 2, with an exception for the root, and thus the total number of vertices is bounded by $O(n)$. In addition, the total length of the labels on the edges of a compressed suffix tree is bounded by $O(n)$. We can create a *compressed* suffix tree for any given string s with $O(n)$ running time [110], and in $O(n)$ total time even if the characters are accessible one by one

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

– and the suffix tree is maintained after every incoming character [132]. 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'.

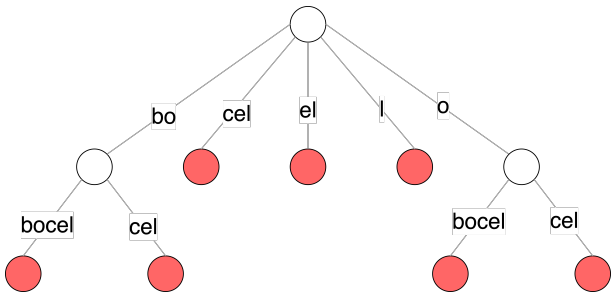


Figure E.1: An example of *suffix tree* for example string "bobocel".

Since the implementation of *compressed* suffix trees is complicated, Manber and Meyers [108] 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 $\{0,1,2,\ldots,n-1\}$. 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.

One of the primitive tools in solving classical string problems is the **longest common prefix** (LCP) query. For two input strings of lenght 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, [108] uses an array containing $\mathsf{LCP}_{s,s}(\pi_i, \pi_{i+1})$ for every $0 \leq i < n-1$ 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 $s'[j:n]$.

Definition E.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.

E.1.2 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. To build a suffix tree, it is necessary to do some sort of lexicographic comparisons on pairs of suffixes to find their placement in the trie. Anytime we need to sort suffixes of a given string, we will call the LCPQ Oracle.

We utilize an LCPQ oracle to give algorithms for constructing suffix array and suffix tree in $O(1/\varepsilon)$ rounds of MPC using $\widetilde{O}(n^{1+\varepsilon})$ total memory. Then, we study location-

sensitive problems *Longest Palindrome Substring* (LPS) and *Longest Common Substring* (LCS). We show direct reductions from LPS and LCS to suffix tree in MPC where we show a novel approach to traversing the suffix tree since we cannot easily perform traditional tree searches, viz. breadth-first search and depth-first search. Moreover, we present memory-efficient algorithms for LPS and LCS relying on an analysis of the *periodicity* of strings and *compressed* LCPQ oracles in which we save memory when storing the LCPQ oracle by only storing information relevant to the queries we will need to answer.

Problem	Model	Source	Rounds	Total Memory
LCPQ Oracle (with k queries)	MPC	Theorem E.1.1	$O(1)$	$\widetilde{O}(n^{1+\varepsilon}+k)$
		Corollary E.1.2	$O(1)$	$\widetilde{O}(n+k+\min(n,k)\cdot n^{\varepsilon})$
		Lemma E.4.2	$O(\log\log n)$	$\widetilde{O}(n+k)$
Suffix Array/Tree	MPC	Theorem E.1.3,E.1.5	$O(1/\varepsilon)$	$\tilde{O}(n^{1+\varepsilon})$
LPS	MPC	Theorem E.1.8	$O(1)$	$\widetilde{O}(n)$
LCS	MPC	Theorem E.1.9	$O(1)$	$\widetilde{O}(n^{1+\varepsilon})$
Suffix Tree	AMPC	Theorem E.1.10	$O(1)$	$\widetilde{O}(n)$

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

Problem	Model	Source	Rounds	Notes
LPS	PRAM	[9]	$O(\log n)$	$O(n^2)$ memory/work
String Matching	MPC	[80]	$O(1)$	$O(n)$ total memory
Suffix Array/Tree	PRAM	[87]	$O(\log(n))$	$\tilde{O}(n)$ memory/work

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

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 $O(n^{1-\varepsilon})$-size local memory, and furthermore, a naive approach to computing the longest common prefix by comparing substrings of length $O(n^{1-\varepsilon})$ sequentially could require $O(n^{\varepsilon})$ rounds for long suffixes. By hashing every substring of size $n^{1-\varepsilon}$, 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 $s[i, i+n^{\varepsilon}]$, $s[i+n^{\varepsilon}, i+2n^{\varepsilon}]$, 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 $O(n^{\varepsilon})$ 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 $n^{1-\varepsilon}$, in order to compare blocks directly we only need $O(n^{2\varepsilon})$ 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 1.3.1 since it is common to all of our MPC algorithms. The resulting data structure can answer a collection of $k = O(n^{1+\varepsilon})$ arbitrary LCP queries $\mathscr{Q} = \{q_1, q_2, \ldots, q_k\}$ in $O(1)$ rounds.

Since our LCPQ oracle construction relies on hashing, our algorithms are necessarily randomized and must succed *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 $O(n^{1-\varepsilon})$ must be larger than $O(n^{\varepsilon})$ in order to find the hashes of blocks of length n^{ε}. Therefore, as discussed in the introduction, we bound ε by $0 < \varepsilon \leq .5$ to make sure local memory $O(n^{1-\varepsilon})$ is always larger than the number of machines $O(n^{\varepsilon})$.

Theorem E.1.1. *For $\varepsilon \in (0, .5]$, there is an $O(1)$-round MPC algorithm, with $\widetilde{O}(n^{1+\varepsilon})$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ 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 $k = O(n^{1+\varepsilon})$ queries, $\mathscr{Q} = \{q_1, q_2, \ldots, q_k\}$, in $O(1)$ rounds.*

We show in Section E.4 that for any value of k which is either $O(n^{1-\varepsilon})$ or $\Omega(n^{1+\varepsilon})$, 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 $k = \omega(n^{1-\varepsilon}) \cap o(n^{1+\varepsilon})$, we present a compressed LCPQ oracle with $O(n + k + \min(n,k) \cdot n^{\varepsilon})$ total memory. Intuitively, since the bottleneck is $\min(n,k) \cdot n^{\varepsilon}$ 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 $O(n^{\varepsilon})$ multiplicative factor of each other); we introduce a different approach utilizing *modular partitioning* to show that solving the LCP queries with

[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.

a total of $O(n+k)$ memory is achievable if we allow $O(\log\log n)$ rounds of communication in MPC (Lemma E.4.2). 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 $n^{\varepsilon/2}$, $n^{\varepsilon/4}$, 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 E.2 or Table E.3 to find more precise details about the performance of a *compressed* LCPQ oracle.

Corollary E.1.2. *For $\varepsilon \in (0,.5]$, there is an $O(1)$-round MPC algorithm $\widetilde{O}(n^{1-\varepsilon})$ memory per machine which initializes an* LCPQ **oracle** *in $O(1)$ rounds w.h.p., and then processes a collection of k queries, $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$, in $O(1)$ rounds. The total memory used by this algorithm is $\widetilde{O}(n+k+\min(n,k)\cdot n^{\varepsilon})$.*

Massively Parallel Suffix Tree

One of our main contributions is an $O(1/\varepsilon)$-round MPC algorithm for building the *suffix tree* of a given string with $O(n^{1+\varepsilon})$ total memory. The LCPQ oracle is central to suffix tree construction and its applications in the MPC model. But first, we give an $O(1/\varepsilon)$-round algorithm for the *suffix array* problem using $O(n^{1+\varepsilon})$ total memory. 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 $\tilde{O}(n^{1+\varepsilon})$ similar to the one discussed in [76]. 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 $\widetilde{O}(n^{1-\frac{i\varepsilon}{2}})$ with high probability. Thus, after $O(1/\varepsilon)$ rounds, we end up with $\widetilde{O}(n^{\frac{\varepsilon}{2}})$-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.

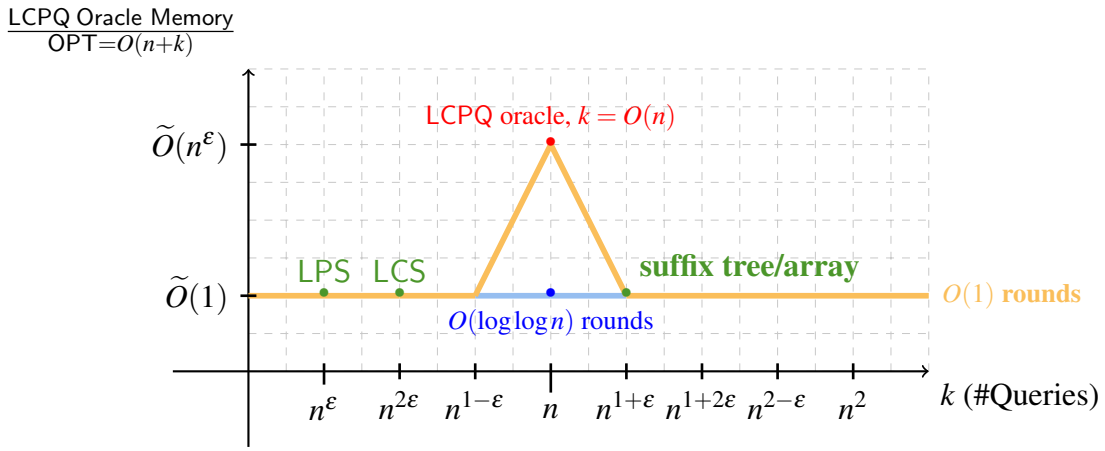


Figure E.2: This plot illustrates the ratio of the memory blowup in a *compressed* LCPQ oracle for different ranges of k where k denotes the number of LCP queries in a batch. The vertical axis shows the ratio of memory blowup of the LCPQ oracle given query set $\mathcal{Q}$ of size k compared to the 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 E.5), LCS (Section E.6), and suffix tree (Section E.3) – 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.

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 E.1.3. *For $\varepsilon \in (0,.5]$, there is an $O(1/\varepsilon)$-round MPC algorithm for computing the* **suffix array** *of a given string s with $\widetilde{O}(n^{1+\varepsilon})$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ memory per processor w.h.p..*

When converting a suffix array to a suffix tree, we follow the main ideas of [87]. One important observation of [87] is that the order of suffixes in the suffix array corresponds to the order of leaves in the suffix tree. Furthermore, they show that internal nodes of the suffix tree correspond to longest common prefixes of a pair of neighboring suffixes in the suffix array. Therefore, we will store these longest common prefixes between neighbor suffixes in an auxiliary array that we call the LCP *array*. To populate the LCP array, we send $O(n)$ queries to the LCPQ oracle to find the LCP of each pair of neighboring suffixes of the suffix array. So, by Theorem E.1.1 we can find the LCP array of s using $\widetilde{O}(n^{1+\varepsilon})$ memory in $O(1)$ rounds of MPC with high probability.

Corollary E.1.4. *For $\varepsilon \in (0,.5]$, given a string s and its suffix array π, there is an algorithm for computing the* LCP ***array*** *of s, w.h.p., with $\widetilde{O}(n^{1+\varepsilon})$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ memory per machine.*

We next construct the suffix tree from the suffix array and the LCP array in $O(1/\varepsilon)$ rounds and $O(n^{1+\varepsilon})$ memory, again mainly following the ideas of [87]. As discussed above, each leaf of the tree corresponds to a unique suffix of the input string, and each internal node represents the longest common prefix between two subsequent suffixes in the suffix array. Therefore, each internal node is already given to us for free from the LCP array, we just need to find the edges between the nodes.

For a suffix i and its corresponding leaf in the suffix tree, we may find its parent by finding the nearest pair of neighboring suffixes with a smaller LCP than that of i and $i+$

1. Since we are only checking the LCP of pairs of neighboring suffixes, finding parents with this method can easily be done with the LCP array. In fact, we observe that any consecutive sequence of suffixes containing i and $i+1$ such that no pair of neighboring suffixes have a larger LCP than $\mathsf{LCP}_{s,s}(\pi_i, \pi_{i+1})$ must in fact share a common prefix. For any such sequence, all the leaves corresponding to the suffixes of this sequence must be descendants of some common internal node corresponding to the nearest smaller LCP of the sequence, which we call the *nearest smallest neighbor*. Unfortunately, it may be possible that some internal nodes are actually the parents of more than two children depending on the prefix this node represents, so it is possible that several entries of the LCP array correspond to the same internal node. To find a unique internal node from the LCP array without any repetitions, we find the *leftmost* suffix that shares a common prefix with a sequence of suffixes which provides a unique internal node (see [87] for further discussion).

Finding the nearest smallest neighbors and leftmost suffixes for a given suffix i may require communication between many machines containing different blocks of the input string. In particular, if an internal node is the ancestor of $O(n)$ suffixes, the nearest smallest neighbor and the leftmost suffix for this internal node may be very far away in the suffix array from some of its descendant suffixes. To avoid unnecessary communication, we utilize a novel block-based data structure to find the exact blocks containing the nearest smallest neighbor and leftmost suffixes without needing to check up to $O(n^{\varepsilon})$ blocks first. We first find the minimum LCP array value for blocks of size $O(n^{1-\varepsilon})$ of the LCP array. If the minimum LCP of a block is smaller than the LCP of a pair of adjacent suffixes, we know this block is a candidate to find the nearest smallest neighbor and leftmost suffixes for this pair. Each pair of suffixes will need to communicate with at most two other machines to find the nearest smallest neighbor, checking the first block with a smaller minimum LCP array value to the left of the given pair of suffixes and the first block with a smaller minimum LCP array value to the right. And similarly, once we have the nearest smallest neighbor values

for each pair of neighboring suffixes, we only need a constant number of communications to determine the *leftmost* suffix pair. Therefore, we can limit communication needed between machines to calculate the edges of the suffix tree to $O(1)$ rounds by using a novel block-based data structure containing minimum LCP array values for each block, which completes the suffix tree.

Theorem E.1.5. *For $\varepsilon \in (0, .5]$, there is an $O(1/\varepsilon)$-round MPC algorithm for computing the* **suffix tree** *of a given string s with $\widetilde{O}(n^{1+\varepsilon})$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ memory per processor with high probability.*

After making the suffix array and the suffix tree, we show reductions to the two applications discussed above, LPS and LCS. Folklore solutions utilizing a suffix tree typically require a tree search such as depth-first search or bread-firth search to find specific nodes within the tree. Unfortunately, since a suffix tree may have depth $O(n)$ and breadth $O(n)$, we must find a different way to answer suffix tree queries without searching or else our communication rounds may be as large $O(n^{\varepsilon})$. We first solve this problem for the LCS application. We will build the suffix tree on $s\#s'$ for input strings s and s'. Recall that each internal node of the suffix tree corresponds to a longest common prefix of two suffixes of the string $s\#s'$. Therefore, we wish to find the lowest internal node of the suffix tree that is an ancestor of a suffix from s and a suffix from s'. The depth of this node represents the length of the longest common substring of s and s'. In MPC, since we cannot do a traditional tree search in constant rounds, we first construct a novel auxiliary array that mediates as an oracle to answer $O(n)$ queries in $O(1)$ rounds about whether or not internal nodes have descendants from both input strings. This array is constructed using similar techniques to the nearest smaller neighbor. We find the range of leaves that are children of a given internal node by comparing LCP array values, and using a block-based data structure, we check if the internal node is indeed the parent of a suffix from each input string in

$O(1)$ rounds.

Corollary E.1.6. (LCS using Suffix Tree). *For $\varepsilon \in (0, .5]$, there is an $O(1/\varepsilon)$-round MPC algorithm for computing the* LCS *of two input strings s, s' with $\widetilde{O}(n^{1+\varepsilon})$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ memory per processor with high probability.*

The LPS solution using suffix tree follows similarly to the LCS solution and uses a modified approach to the typical folklore reduction to suffix tree as well. Instead of building the suffix tree on two input strings, we build the suffix tree on the input string and its reverse string. Then, we want to find the lowest internal node of the suffix tree that is an ancestor of a suffix and its corresponding reverse suffix in the reverse string. Similarly to the modified LCS reduction, we build an auxiliary array that allows us to efficiently check if each internal node is indeed the ancestor of such a suffix and its reverse. With this array, we are able to then investigate each internal node in parallel to find the lowest one and output the longest palindrome of the input string.

Corollary E.1.7. (LPS using Suffix Tree). *For $\varepsilon \in (0, .5]$, there is an $O(1/\varepsilon)$-round MPC algorithm for computing the* LPS *of an input strings s with $\widetilde{O}(n^{1+\varepsilon})$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ memory per processor with high probability.*

LPS, LCS, and the Periodicity Properties

In Section E.5, we present an improved algorithm for LPS over the previous suffix tree reduction that uses linear total memory and a constant amount of rounds for communication. 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 E.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 [9] to analyze the periodicity of a string with multiple palindromes complemented by the novel compressed LCPQ oracle (see Section E.4) to find the maximum length palindrome per block using tight memory and constant rounds, as described in Theorem E.1.8. The periodicity property of blocks with more than one LPS candidates helps us to reduce k, the number of LCP queries.

Theorem E.1.8. *For $\varepsilon \in (0, .5]$, there is an $O(1)$-round MPC algorithm that solves* **Longest Palindrome Substring (**LPS**)** *with $\widetilde{O}(n)$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ memory per processor, w.h.p.*

Similarly, in Section E.6 we present an improved algorithm for LCS over the initial suffix tree reduction that uses $O(1)$ rounds. The algorithm uses $O(n^{1+\varepsilon})$ 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 $\mathsf{LCP}(q) \geq 3n^{1-\varepsilon}$ 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 $3n^{1-\varepsilon}$. We use *pairwise block machines* to compute the longest common substring with length smaller than $3n^{1-\varepsilon}$ 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 same period string is bounded by $O(n^{2\varepsilon})$. 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 $O(n^{2\varepsilon})$ 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 determine the candidate pair with maximum LCP in this pair of blocks. Thus, there are a collection of $O(n^{2\varepsilon})$ candidate pairs which the maximum LCP among these candidate pairs is equal to the LCS of two strings s and s'.

Theorem E.1.9. *For $\varepsilon \in (0, .5]$, there is an $O(1)$-round MPC algorithm that solves* **Longest Common Substring (**LCS**)** *with $\widetilde{O}(n^{1+\varepsilon})$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ memory per processor, w.h.p.*

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 $O(n)$ total memory and in $O(1/(1-\varepsilon))$ 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 $\varepsilon \leq .5$, which stemmed from modular partitioning and the MPC versions of the block-based data structures.

Theorem E.1.10. (Suffix Tree in AMPC). *For $\varepsilon \in (0, 1]$, 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 $\widetilde{O}(n^{1-\varepsilon})$ memory per processor with high probability.*

[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.

We do not give the statements and their proofs, but utilizing the fully-functional suffix tree to solve LPS and LCS in AMPC easily follows using the reductions shown for MPC.

E.1.3 MPC Implementation Details

In the rest of 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 Sections E.2 and E.4, 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 Section E.3, as well as in the memory-optimal algorithms for LPS and LCS in Sections E.5 and E.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 the core, 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 prime number p.[5]

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

Block-based Data Structures

The input string s is initially partitioned into n^{ε} consecutive *blocks* of size $O(n^{1-\varepsilon})$, denoted by $b_0, b_1, \ldots, b_{n^{\varepsilon}-1}$, so that each block fits into the memory of a single machine.

[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*.

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 E.2 (Blocks b_α and Block Machines μ_α)**.** There is a collection of n^ε machines $\{\mu_0, \mu_1, \ldots, \mu_{n^\varepsilon-1}\}$ referred to as ***block machines*** for string s *(*and $\{\mu'_0, \mu'_1, \ldots, \mu'_{n^\varepsilon-1}\}$ 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 $b_\alpha = s[\alpha n^{1-\varepsilon} : (\alpha+1)n^{1-\varepsilon})$.

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-sized information about every block b_α, say $f(b_\alpha)$[6], so that $\{f(b_0), f(b_1), \ldots, f(b_{n^\varepsilon-1})\}$ can be copied and fit inside the memory of a single machine as long as $\varepsilon \leq 0.5$. Furthermore, we use a *segment tree* inside each machine μ_α which computes $f(b_\alpha[l:r])$ for a total of $O(n^{1-\varepsilon})$ 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 (E.1), the corresponding function merge function g equals:

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

[6]Function f can also be defined on any arbitrary array with length n instead of the string itself.

Modular Partitioning

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

Definition E.3. There are a collection of n^ε machines $\{\lambda_0, \lambda_1, \ldots, \lambda_{n^\varepsilon-1}\}$ for string s *(*and $\{\lambda'_0, \lambda'_1, \ldots, \lambda'_{n^\varepsilon-1}\}$ for string s'*)*, where λ_x *(*and λ'_x*)* contains the hash of substrings of length $O(n^\varepsilon)$ starting at indices i so that $i \bmod n^\varepsilon = x$, and they are called ***mod machines***.

Weighted Load Balancing

Our machine and data structures are often required to answer large amounts of queries in parallel, i.e. $O(n)$ or $O(n^{1+\varepsilon})$ queries. We first partition a query set into groups so that all queries that must be routed to the same machine to be answered 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

[7] Note that we have a local access to the next block, i.e., $b_{\alpha+1}$, inside μ_α.

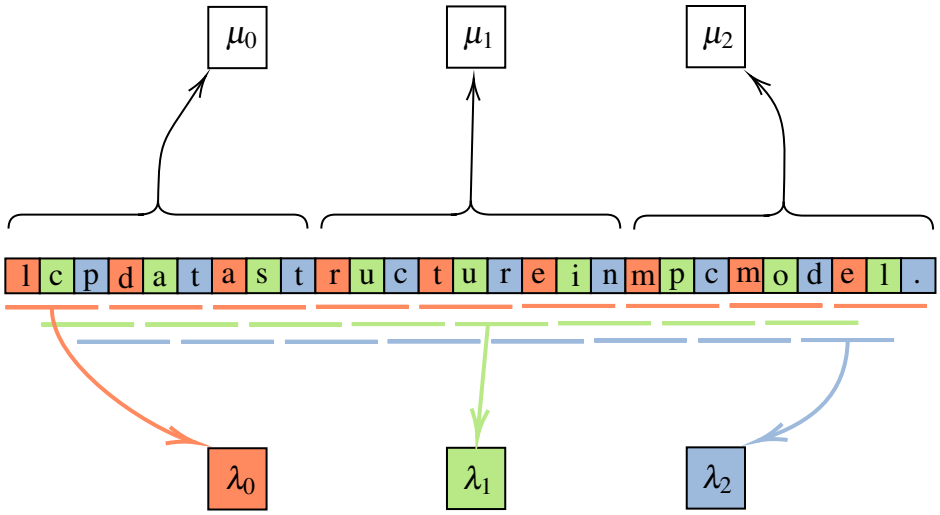


Figure E.3: The modular partitioning is illustrated in this figure. In this example, $n = 27$ and $\varepsilon = 1/3$, which means $n^{\varepsilon} = 3$ and $n^{1-\varepsilon} = 9$. There are 3 *block machines* μ_0, μ_1, and μ_2 which store the substrings of s corresponding to blocks b_0, b_1, and b_2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

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 $\mathcal{Q}$ as an example where each query $q \in \mathcal{Q}$ 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 $\mathcal{Q}$ into $n^{2\varepsilon}$ 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 $i \bmod n^{\varepsilon} = x$ and $j \bmod n^{\varepsilon} = y$. We may assign small groups, ones with size $O(n^{1-\varepsilon})$, to one of $\lambda_{x,y}$ machines without violating the memory constraints. However, sometimes a large number of queries are concentrated in specific groups. The input queries might even consist of a single group of size $O(n^{1+\varepsilon})$. To fix this issue, we replicate the data stored in each machine based on the number of queries assigned to it. See Algorithm 16 for more details. In Algorithm 16, the data with total size $O(n^{1+\varepsilon})$ is partitioned into $n^{2\varepsilon}$ groups $P_{x,y}$, and we aim to distribute the queries among a collection of $O(n^{2\varepsilon})$ machines

$\xi_{x,y}$ with possible repetitions[8].

Algorithm 16: WeightedLoadBalancing($\mathscr{S} = \{\xi_1, \xi_2, ...\xi_m\}, \{P_1, P_2, ...P_m\}$)

for $\xi_x \in \mathscr{S}$ **do**
  $c_x \leftarrow \lceil \frac{|P_x|}{n^{1-\varepsilon}} \rceil$;
  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 $1 \leq z \leq c_x$;
end
return $\{c_1, c_2, ..., c_m\}$;

E.1.4 Outline of the Paper

In Section E.2, we introduce the LCPQ oracles and show that they can handle a batch of LCP queries using $O(n^{1+\varepsilon})$ total memory. In Section E.3, we build suffix array and suffix tree using LCPQ oracles in $O(1/\varepsilon)$ rounds. Moreover, we show constant-round reductions from LCS and LPS to suffix trees in MPC using block-based data structures. In Section E.5 and Section E.6, we give MPC algorithms respectively for LPS and LCS with linear total memory using the periodicity properties of strings. But first, we introduce compressed variants of LCPQ oracles in Section E.4 with tight memory for different ranges of k, the number of queries. In Section E.5, we improve the total memory and the number of rounds of the LPS problem to $O(n)$ and $O(1)$ respectively using compressed LCPQ. In Section E.6, we first demonstrate an $O(1)$-round MPC algorithm for solving LCS using $O(n^{1+\varepsilon})$ total memory. We improve the total memory of LCS to $O(n)$ for the case that the solution size is larger than $O(n^{2\varepsilon})$. Then, we exploit the optimizations of the LCPQ oracles to give an

[8] In the definition Algorithm 16, 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.

$O(n)$ memory for LCS in $O(\log\log n)$ rounds of MPC. Finally, we show a memory-optimal and constant round algorithm for finding suffix arrays in the AMPC model in Section E.7.

E.2 Longest Common Prefix Query Oracle

A fundamental component of our later algorithms is a data structure for simultaneously processing a query batch consisting of $O(n^{1+\varepsilon})$ arbitrary *Longest Common Prefix* (LCP) queries in $O(1)$ rounds of the MPC model. Recall the definition of LCP queries from Definition E.1. An LCP query $q = (i, j)$ returns the length of the longest common prefix of the suffixes of two given strings s and s' starting at indices i and j respectively, denoted by $\mathsf{LCP}_{s,s'}(i, j)$. Along with the input strings s and s', we are also given a collection of LCP queries $\mathscr{Q} = \{q_1, q_2, \ldots, q_k\}$ where $k = O(n^{1+\varepsilon})$.

Recall the definitions of *block machines* and *mod machines* from respectively Definitions E.2 and E.3. In order to use the data stored in a collection of mod machines $\lambda_0, \lambda_1, \ldots, \lambda_{n^\varepsilon-1}, \lambda'_0, \lambda'_1, \ldots, \lambda'_{n^\varepsilon-1}$ to estimate the length of the LCP queries, it is tempting to allocate a machine for each pair of remainders. In other words, if we allocate $O(n^{2\varepsilon})$ machines $\lambda_{x,y}$ for every pair $0 \leq x, y < n^\varepsilon$ so that it contains the data stored in λ_x as well as the data stored in λ'_y, we do not violate the memory constraints. However, we only need an $O(n^\varepsilon)$-sized subset $\Lambda \subseteq \{\lambda_{x,y}\}$ of mod machines. We can retrieve the required information from the following subset[9]:

$$\Lambda = \{\lambda_{0,0}, \lambda_{0,1}, \ldots, \lambda_{0,n^\varepsilon-1}, \lambda_{1,0}, \ldots, \lambda_{n^\varepsilon-1,0}\}.$$

The sketch of the algorithm is to only allocate $\lambda_{x,y}$ when there is a query $q = (i, j)$ where $i \bmod n^\varepsilon = x$ and $j \bmod n^\varepsilon = y$. In each λ_x we don't need to keep the hash of every

[9] We will need this observation later in Section E.4 to optimize the required memory for a *compressed* LCPQ oracle.

substring with length n^ε and starting index $i \bmod n^\varepsilon = x$. We can only keep the substrings with the length equal to the difference between two consecutive existing first coordinate of queries, i.e.,

$$D_{x,y} = \{d \mid \exists\, 1 \le j \le |Q_{x,y}| \quad \text{s.t. } d = i_{j+1} - i_j\} \tag{E.2}$$

where $i_1, i_2, \ldots, i_{|Q_{x,y}|}$ is the sorted sequence of the first coordinate of every (i, j) in $Q_{x,y}$. If we only keep the substrings with a length included in $D_{x,y}$, we have $O(n^{2\gamma})$ total memory, where the worst cost happens when all queries belong to $Q_{0,0}$ for example. We do another similar step for block machines $\mu_{\alpha,\beta}$ in the second stage of processing LCP queries.

The total memory used by all $\lambda_{x,y}$ machines is therefore bounded by $O(n^{1+\varepsilon})$. For a given query $q = (i, j)$, so that $i \bmod n^\varepsilon = x$ and $j \bmod n^\varepsilon = y$, we use machine $\lambda_{x,y}$ to find the longest common prefix of $s[i : n]$ and $s'[j : n]$ whose length is a multiple of n^ε. Each machine $\lambda_{x,y}$ is able to find the hash of every substring with a length multiple of n^ε which start at indices with remainder x in s or with remainder y in s' when dividing by n^ε. Thus, we only need to redirect every query to a target machine $\lambda_{x,y}$ in order to finish the first phase. Similarly, we create $\mu_{\alpha,\beta}$ for every $0 \le \alpha, \beta < n^\varepsilon$ that contains the data stored in μ_α and μ'_β, see Figure E.4 for an example.

Definition E.4. There are two collections of $n^{2\varepsilon}$ machines, $\{\mu_{\alpha,\beta} \mid 0 \le \alpha, \beta < n^\varepsilon\}$ and $\{\lambda'_{x,y} \mid 0 \le x, y < n^\varepsilon\}$, so that:

- $\mu_{\alpha,\beta}$ contains the substring of the α-th block of s and β-th block of s', respectively b_α and b'_β, and is called a ***pairwise block machine***.
- Similarly, $\lambda_{x,y}$ contains the hash of substrings of length $O(n^\varepsilon)$ starting at indices i in s so that $i \bmod n^\varepsilon = x$, and indices j in s' so that $j \bmod n^\varepsilon = y$, and is called a ***pairwise mod machine***.

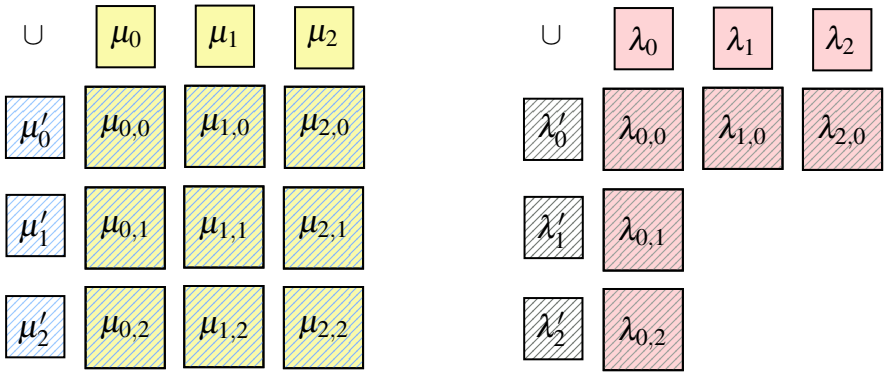


Figure E.4: An illustration of *pairwise block machines* $\mu_{\alpha,\beta}$ and *pairwise mod machines* $\lambda_{x,y}$ for the same example. Pairwise block machines and pairwise mod machines have $O(n^{2\varepsilon})$ and $O(n^{\varepsilon})$ members respectively (9 and 5 members in this example), where in the former for every $0 \leq \alpha, \beta < n^{\varepsilon}$, $\mu_{\alpha,\beta}$ stores the union of the data of μ_{α} and μ'_{β}, i.e., $\mu_{\alpha,\beta} = \mu_{\alpha} \cup \mu'_{\beta}$. Similarly, we have $\lambda_{x,y} = \lambda_x \cup \lambda'_y$ for every $0 \leq x, y < n^{\varepsilon}$ such that $\min(x, y) = 0$.

We initialize a pairwise block-based data structure which fits into $O(n^{2\varepsilon})$ machines each with $\widetilde{O}(n^{1-\varepsilon})$ local memory by utilizing *Modular Partitioning* and *Weighted Load Balancing*. Next, we utilize the data stored in our data structure to find in parallel the length of every LCP query from $\mathscr{Q}$ using $O(1)$ rounds. Theorem E.1.1 formally states our result about initializing and using the LCP data structure in the MPC model.

Theorem E.1.1. *For $\varepsilon \in (0, .5]$, there is an $O(1)$-round MPC algorithm, with $\widetilde{O}(n^{1+\varepsilon})$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ 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 $k = O(n^{1+\varepsilon})$ queries, $\mathscr{Q} = \{q_1, q_2, \ldots, q_k\}$, in $O(1)$ rounds.*

The data structure processes the queries in two phases:

1 It first computes an estimated length of each query $q = (i, j)$. More precisely, it finds the longest common prefix starting at i and j whose length is a multiple of n^{ε}. In other words,

$$\max\{u \geq 0 \mid s[i : i + un^{\varepsilon}] = s'[j : j + un^{\varepsilon}]\}$$

2 Then, it finds the exact length of each query by gathering the queries that end in block b_α of s and block b'_β of s' into the same machine so that all such queries can be answered locally.

Algorithm 17: LCPInit(s, s')

```
for 0 ≤ x < n^ε do
    for 1 ≤ u < n^{1−ε} do
        Compute hash(s[(u−1)n^ε + x : un^ε + x]) in μ_{⌊u/n^{1−2ε}⌋} and send it to λ_x;
        Compute hash(s'[(u−1)n^ε + x : un^ε + x]) in μ'_{⌊u/n^{1−2ε}⌋} and send it to λ'_x;
    Replicate the data stored in mod machines λ_x and λ'_x into pairwise mod
     machines λ_{x,y} and λ_{y,x} respectively for all 0 ≤ y < n^ε;
    Replicate the data stored in block machines μ_α and μ'_α into pairwise block
     machines μ_{α,β} and μ_{β,α} respectively for all 0 ≤ β < n^ε;
```

Now that we have shown how to initialize the LCP data structure, which is formally summarized in Algorithm 17, we are ready to describe an $O(1)$-round MPC algorithm to answer a batch of queries $\mathscr{Q}$ of size $O(n^{1+\varepsilon})$. In the rest of this section, we prove Theorem E.1.1.

of Theorem E.1.1. We first partition $\mathscr{Q}$ into $n^{2\varepsilon}$ groups $Q_{x,y}$ based on the remainders of dividing both coordinate of each query by n^ε. Each group $Q_{x,y}$ contains all the queries $q = (i, j)$ such that $i \bmod n^\varepsilon = x$ and $j \bmod n^\varepsilon = y$. We may assign small groups, ones with size $O(n^{1-\varepsilon})$, to one of $\lambda_{x,y}$ machines without violating the memory constraints. However, sometimes a large number of queries are concentrated in specific groups. The input queries might even consist of a single group of size $O(n^{1+\varepsilon})$. We utilize Algorithm 16 (WeightedLoadBalancing) to balance the number of replicas of each machine $\lambda_{x,y}$ based on the number of queries assigned to the same machine, i.e., $|Q_{x,y}|$.

We partition the queries and redirect them into the replicas of $\lambda_{x,y}$ created by Algorithm 16. Then, for a batch of queries $Q^{(z)}_{x,y}$ of size $O(n^{1-\varepsilon})$, we use the local memory of a

replica of $\lambda_{x,y}$ to find an estimate of the length of each query. See line (6) of Algorithm 18 for more details. We compute the longest common prefix of query $q = (i, j)$ whose length is a multiple of n^{ε}. In terms of a variable u^*, the longest common prefix for the query (i, j) found so far has length equal to $u^* n^{\varepsilon}$. We also have $n^{2\varepsilon}$ initially empty bags $F_{\alpha,\beta}$ for all $0 \leq \alpha, \beta < n^{\varepsilon}$, one for each pair of blocks. We use the estimated length to find out the pair of blocks (b_α, b'_β) on which the LCP query $q = (i, j)$ ends, and we put it in the respective bag $F_{\alpha,\beta}$. We use Algorithm 16 again, to balance the number of replicas of each $\mu_{\alpha,\beta}$ based on the size of the bags $F_{\alpha,\beta}$, which represent the load, and to redirect the queries,.

Once every query is assigned to a replica $\mu^{(z)}_{\alpha,\beta}$, we start the second phase of processing queries. In this phase, our goal is to find the exact length of every query, and each query $q = (i, j)$ is already redirected to a machine $\mu^{(z)}_{\alpha,\beta}$ where b_α and b'_β are respectively blocks of s and s' on which the LCP of $q = (i, j)$ ends. Thus, the data stored in $\mu^{(z)}_{\alpha,\beta}$ is sufficient to find the exact LCP of $q = (i, j)$. Let $i' = i \bmod n^{1-\varepsilon}$ and $j' = j \bmod n^{1-\varepsilon}$ be indices of respectively b_α and b'_β with the same offset i and j have in their own blocks. Let d be the minimum of $\{i', j'\}$. We find the longest common prefix of $b_\alpha[i' - d : n^{1-\varepsilon}]$ and $b'_\beta[j' - d : n^{1-\varepsilon}]$, and we denote by ℓ the length of this common prefix. We show that the length of query $q = (i, j)$ is equal to $\ell + \max\{\alpha n^{1-\varepsilon} - i, \beta n^{1-\varepsilon} - j\}$. See Algorithm 18 for more details. □

E.3 Suffix Array and Suffix Tree

We wish to find the *suffix tree* of a given string s with length n in the MPC model with $O(1/\varepsilon)$ rounds and $\widetilde{O}(n^{1-\varepsilon})$ memory per machine for some $0 < \varepsilon \leq 0.5$. In addition, the total memory needs to be $\widetilde{O}(n^{1+\varepsilon})$. First, we attempt to build the **suffix array** of a given string s in a constant number of rounds in MPC. The *suffix array* of a given string s is just a permutation $\pi : [n] \to [n]$ over the suffixes of s so that they are sorted lexicographically

Algorithm 18: LCPBatchQueries$(s, s', \mathscr{Q})$

$F_{\alpha,\beta} \leftarrow \emptyset$ for all $0 \le \alpha, \beta < n^{\varepsilon}$;
$Q_{x,y} \leftarrow \{q = (i,j) \in \mathscr{Q} \mid x = i \bmod n^{\varepsilon} \wedge y = j \bmod n^{\varepsilon}\} \quad \forall 0 \le x, y < n^{\varepsilon}$;
$c_{x,y} \leftarrow \mathsf{WeightedLoadBalancing}(\{\lambda_{x,y} \mid 0 \le x, y < n^{\varepsilon}\}, \{Q_{x,y} \mid 0 \le x, y < n^{\varepsilon}\})$;
for $0 \le x, y < n^{\varepsilon}, 1 \le z \le c_{x,y}$ **do**
    // in machine $\lambda_{x,y}^{(z)}$
    for $q = (i,j) \in Q_{x,y}^{(z)}$ **do**
        $u^* \leftarrow \max\{u \mid s[i : i + un^{\varepsilon}] = s'[j : j + un^{\varepsilon}]\}$;
        Add q to $F_{\lfloor \frac{i+u^* n^{\varepsilon}}{n^{1-\varepsilon}} \rfloor, \lfloor \frac{j+u^* n^{\varepsilon}}{n^{1-\varepsilon}} \rfloor}$;
$c'_{\alpha,\beta} \leftarrow \mathsf{WeightedLoadBalancing}(\{\mu_{\alpha,\beta} \mid 0 \le \alpha, \beta < n^{\varepsilon}\}, \{F_{\alpha,\beta} \mid 0 \le \alpha, \beta < n^{\varepsilon}\})$;
for $0 \le \alpha, \beta < n^{\varepsilon}, 1 \le z \le c'_{\alpha,\beta}$ **do**
    // in machine $\mu_{\alpha,\beta}^{(z)}$
    for $q = (i,j) \in F_{\alpha,\beta}^{(z)}$ **do**
        $i' \leftarrow i \bmod n^{1-\varepsilon}$;
        $j' \leftarrow j \bmod n^{1-\varepsilon}$;
        $d \leftarrow \min\{i', j'\}$;
        $\ell \leftarrow \max\{\ell \mid b_{\alpha}[i' - d : i' - d + \ell] = b'_{\beta}[j' - d : j' - d + \ell]\}$;
        $\mathsf{LCP}(i,j) \leftarrow \ell + \max\{\alpha n^{1-\varepsilon} - i, \beta n^{1-\varepsilon} - j\}$;

based on π. We employ a quick-sort-like approach, similar to [76], which sorts an array in $O(1/\varepsilon)$ rounds in MPC. Note that we need to make comparison queries carefully since we can't answer each of them in $O(1)$ time as opposed to the sorting of an array of integers.

Theorem E.1.3. *For $\varepsilon \in (0, .5]$, there is an $O(1/\varepsilon)$-round MPC algorithm for computing the* **suffix array** *of a given string s with $\widetilde{O}(n^{1+\varepsilon})$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ memory per processor w.h.p..*

Proof. In the first step $z = 1$, we uniformly randomly select $n^{\varepsilon/2}$ pivots for each remaining bag in L_z, and therefore, we need to perform a total of $O(n^{1+\varepsilon/2})$ lexicographical comparison of two suffixes of s in the first step. There are a total of $\ell = (2/\varepsilon) - 1$ steps to finish the sorting, and at the end of the z-th step ($1 \le z \le \ell$), we have $O(n^{z\varepsilon/2})$ disjoint subsequences with expected size $O(n^{1-z\varepsilon/2})$, which is concentrated, and then each of them needs to be

Algorithm 19: SuffixArray(s)

```
Let ℓ ← ⌈2/ε⌉ − 1;
Let L_0, L_1, L_2, ..., L_ℓ be a collection of lexicographically ordered bags of sets;
Add {1, 2, ..., n} to L_0;
D_{s,s} ← LCPInit(s,s);
π ← {(i, i) | 0 ≤ i < n};
for 1 ≤ z ≤ ℓ do
    for 1 ≤ t ≤ |L_z| do
        Let U_t = {u_{t,1}, u_{t,2}, ..., u_{t,k}} be the t-th set in L_z ordered by the
          lexicographical order;
        if z = ℓ then
            // We have k = O(n^{ε/2}).
            SmallChunkSort(t, U_t, π)
        else
            PivotFiltering(t, U_t, O(n^{ε/2}))
        end
    end
end
Relabel π so that the relative order of π_i's is preserved, and the range of π is [n];
```

sorted recursively. See Algorithm 20 for more details. PivotFiltering(t, U_t, V) assumes U_t is the t-th unsorted chunk, in the relative ordering of the chunks, and decomposes U_t into V smaller chunks based on $V-1$ pivots so that the smaller chunks are relatively sorted, and will be processed in the next step.

In each step $1 \leq z < 2/\varepsilon$, we again choose $n^{\varepsilon/2}$ random pivot for each of those subsequences, but still the total number of comparisons is bounded by $O(n^{\varepsilon/2} \cdot n^{z\varepsilon/2} \cdot n^{1-z\varepsilon}/2) = O(n^{1+\varepsilon/2})$. We first sort the randomly selected pivots using Algorithm 21 since $V = O(n^{\varepsilon/2})$, which we explain shortly. We group the remaining elements of a bag using the result of their comparison with the $O(n^{\varepsilon/2})$ sorted pivots of the same bag.

Finally, in the last step $z = \ell$, we have at most $O(n^{\varepsilon/2})$ suffixes in each bag, and we can query every pair of suffixes in every bag to create a collection Q of $O(n^{1+\varepsilon/2} \cdot n^{\varepsilon/2}) = O(n^{1+\varepsilon})$ queries. We locally sort the small number, $O(n^{\varepsilon/2})$ of suffixes by comparing each

Algorithm 20: PivotFiltering$(t, U_t = \{u_{t,1}, u_{t,2}, \ldots, u_{t,k}\}, V)$

Let p_0 and p_V be dummy elements which are by definition lexicographically smaller and lexicographically greater respectively than any other suffix;
Let $p'_1, p'_2 \leq \ldots \leq p'_{V-1}$ be $V-1$ uniformly random indices from $[k]$;
// Sorting the pivots
Let $\pi' \leftarrow \{(i,i) \mid 1 \leq i < V\}$;
SmallChunkSort$(0, \{p'_1, p'_2, \ldots, p'_{V-1}\}, \pi')$;
Let $p_{\pi'_i} \leftarrow p'_i$ for all $1 \leq i < V$;
// Decompose U_t based on $O(V)$ pivots into $v_0, v_1, \ldots, v_{V-1}$.
$\kappa_{i,j} \leftarrow D_{s,s}.\mathsf{query}(p_i, u_{t,j})$ for every $0 \leq i \leq V$ and $1 \leq j \leq k$;
Let $v_i \leftarrow \left\{ u_{t,j} \in U_t \mid \begin{cases} s[p_i + \kappa_{i,j}] \leq s[u_{t,j} + \kappa_{i,j}], \text{and} \\ s[p_{i+1} + \kappa_{i+1,j}] > s[u_{t,j} + \kappa_{i+1,j}] \end{cases} \right\}$ for every $0 \leq i < V$;
Add $v_0, v_1, \ldots, v_{V-1}$ to L_{z+1} so that v_i has index $(t-1)n^{\varepsilon/2} + i$ in L_{z+1};

pair of them together and use it as an oracle for sorting. See Algorithm 21 for more details. According to Theorem E.1.1, we can handle all these queries in $O(1)$ round and $O(n^{1+\varepsilon})$ total memory using our LCPQ oracle. Knowing the result of the collection of queries $\mathscr{Q}$, we can compare the respective characters and compare two suffixes lexicographically. Ultimately, we sort each bag of size $O(n^{\varepsilon/2})$ using SmallChunkSort subroutine, and the resulting function π is sorted lexicographically but exceeds the range limits. The last step is to relabel π so that the range of the function is indeed $[0,n)$ and also π is a valid permutation.

Algorithm 21: SmallChunkSort$(t, U_t = \{u_{t,1}, u_{t,2}, \ldots, u_{t,k}\}, \pi)$ s.t $k = O(n^{\varepsilon/2})$

$\kappa_{i,j} \leftarrow D_{s,s}.\mathsf{query}(u_{t,i}, u_{t,j}) \quad \forall (i,j) \leq [k] \times [k]$;
$\eta_{i,j} \leftarrow \begin{cases} 1 & \text{if } s[u_{t,i} + \kappa_{i,j}] \leq s[u_{t,j} + \kappa_{i,j}] \\ 0 & \text{if } s[u_{t,i} + \kappa_{i,j}] > s[u_{t,j} + \kappa_{i,j}] \end{cases}$;
Sort U_t locally using $\eta_{i,j}$'s as a comparison oracle to get $W_t = \{w_{t,1}, w_{t,2}, \ldots, w_{t,k}\}$;
Let $\pi_{w_{t,i}} \leftarrow tn + i$ for every $1 \leq i \leq k$;

□

E.3.1 Converting Suffix Array to Suffix Tree

Now we are ready to use the constructed suffix array to build a suffix tree. Let $\pi : [n] \to [n]$ be the permutation returned by the suffix array algorithm. Then $s_{\pi_i} s_{\pi_i+1} \dots s_{n-1}$ is the i-th substring of s lexicographically. We start by finding the LCP of each pair of adjacent suffixes in the suffix array, which we store in an array denoted LCP^{π}, e.g. $\mathsf{LCP}^{\pi}[i] = \mathsf{LCP}_{s,s}(\pi_i, \pi_{i+1})$. We populate the n indices of LCP^{π} using our LCPQ oracle from Section E.2. Once the LCP^{π} array is made, we utilize the ideas from Iliopoulos and Rytter [87] complemented by a block-based data structure on the minimum values of the LCP^{π} array across each block to build the internal nodes of the suffix tree in constant rounds and $\tilde{O}(n^{1+\varepsilon})$ total memory.

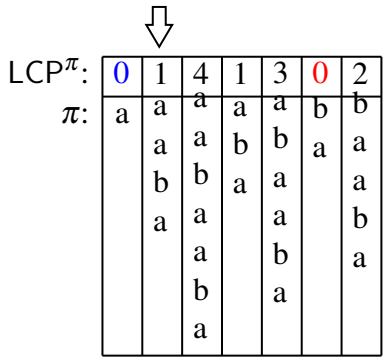


Figure E.5: The above figure is an example of the suffix array π and LCP^{π} array for string "aabaaba". Furthermore, we look at π_1 as an example, the second suffix in the suffix array denoted by the arrow. The nearest left suffix and nearest right suffixes with a smaller LCP^{π} value are the suffixes "a" and "ba", as denoted by the blue and red colorings. The indices of these suffixes will therefore be $\mathscr{L}[\pi_1]$ and $\mathscr{R}[\pi_1]$, respectively.

To describe the details of our suffix tree algorithm, we review the solution of [87] to address the difficulties we need to confront when implementing their approach in MPC. First, they observe that each non-leaf node in a suffix tree is the longest common ancestor (LCA) of two consecutive suffixes in the suffix array. They therefore start by finding the LCP of each pair of consecutive suffixes in the suffix array, which in turn gives the depth

of the LCA of the same pair of vertices in the suffix tree. As we mentioned before we can find the LCP of pairs of suffixes using our LCPQ oracle. Following the methods of [87], we need to compute two more arrays. The first array is a nearest smallest neighbor array SN, which is defined as:

$$\begin{cases} \mathsf{SN}[i] = & \begin{cases} \mathscr{L}[i], & \text{if } \mathsf{LCP}^{\pi}[\mathscr{L}[i]] \geq \mathsf{LCP}^{\pi}[\mathscr{R}[i]] \\ \mathscr{R}[i] & \text{o.w.} \end{cases} \\ \mathscr{L}[i] = & \max_{j<i} \mathsf{LCP}^{\pi}[j] < \mathsf{LCP}^{\pi}[i], \\ \mathscr{R}[i] = & \min_{j>i} \mathsf{LCP}^{\pi}[j] < \mathsf{LCP}^{\pi}[i]. \end{cases}$$

Intuitively, $\mathsf{SN}[i]$ will correspond to the parent of suffixes i and $i-1$ since it is the nearest pair of suffixes that have a shorter LCP^{π} value than $\mathsf{LCP}^{\pi}[i]$. See Figure E.5 for an example of the LCP^{π} array and finding $\mathscr{L}$ and $\mathscr{R}$ values for an index. To break the ties between multiple pairs of suffixes that have the same LCP, we will also compute an array named $\mathsf{Leftmost}[i]$ which is the leftmost sibling of the LCA of suffixes i and $i-1$, defined formally as:

$$\mathsf{Leftmost}[i] = \begin{cases} 0 & \text{if } \mathsf{LCP}^{\pi}[i] = 0 \\ \min_{j \in [1,i]} \{\mathsf{LCP}^{\pi}[j] = \mathsf{LCP}^{\pi}[i], j \geq \mathsf{SN}[i]\} & \text{o.w.} \end{cases}$$

By doing this, [87] observed that the parent of each non-leaf vertex i may be correctly and uniquely identified by $j = \mathsf{Leftmost}[\mathsf{SN}[i]]$ with label $(j, \mathsf{LCP}^{\pi}[j])$. It is not hard to find the father of leaves as well using the same two arrays Leftmost and SN. The following lemma from [87] summarizes the relationships between these arrays and the suffix tree:

Lemma E.3.1 (Lemma 1, [87])**.** *Given internal node* $(k_1, LCP^{\pi}[k_1])$ *and leaf node* (k_2) *where* k_1, k_2 *are the indices of leaf nodes corresponding to the suffixes* $s[k_1 : n), s[k_2 : n)$

respectively:

1 *The parent of* $(k_1, \mathsf{LCP}^\pi[k_1])$ *is* $(j, \mathsf{LCP}^\pi[j])$ *for* $j = \mathsf{Leftmost}[\mathsf{SN}[k_1]]$.

2 *If* $\mathsf{LCP}^\pi[k_2] \leq \mathsf{LCP}^\pi[k_2+1]$ *then the parent of* (k_2) *is* $(j, \mathsf{LCP}^\pi[j])$ *for* $j = \mathsf{Leftmost}[k_2]$. *Else the parent of* (k_2) *is* $(k_2+1, \mathsf{LCP}^\pi[k_2+1])$.

Now, to compute the SN and Leftmost arrays in MPC, we will first find and store LCP^π using the compressed LCPQ oracle. Specifically, on all block processors μ_α, $0 \leq \alpha < n^\varepsilon$, the processor with block $b_\alpha = s[in^{1-\varepsilon} : (i+1)n^{1-\varepsilon}]$ as defined in E.2, we will store $\mathsf{LCP}^\pi[j]$ for any index j such that $s_j \in b_\alpha$. Additionally, on every processor μ_α, we find the minimum over all stored LCP^π values of the previous step, which we denote as

$$m_i := \min_{j \in [in^{1-\varepsilon}, (i+1)n^{1-\varepsilon})} (\mathsf{LCP}^\pi[j]).$$

We will distribute these minimum to all processors so that any processor μ_α has a block-based data structure M such that $M[i] = m_i$. We then easily find $\mathscr{L}[k]$ and $\mathscr{R}[k]$ by finding the nearest left and right blocks b_ℓ and b_r such that $m_\ell, m_r < \mathsf{LCP}[k]$ and then send queries to find the exact value of $\mathscr{L}[k], \mathscr{R}[k]$ to processors μ_ℓ, μ_r, respectively, of all indices k where b_ℓ, b_r are the nearest blocks with a smaller longest common prefix value than $\mathsf{LCP}[k]$. If $\Omega(n^{1-\varepsilon})$ queries are sent to a single processor, we will duplicate the processor and distribute the queries evenly using Algorithm 16. Since there is only one query per index to find SN, there is $O(n)$ total queries, which means we only need constant rounds of communication for this step. A similar set of steps is used to then build the Leftmost array for block b^i on processor μ_i. If $\mathsf{LCP}^\pi[k] = 0$ for an index k, then $\mathsf{Leftmost}[k]$ is trivially 0. Otherwise, we check the minimum LCP^π values of each neighboring blocks to the left of b^i to see if any have an equal value to $\mathsf{LCP}^\pi[k]$. If there is an equal minimum, then we query the processor with that block to find the exact value of $\mathsf{Leftmost}[k]$; if there is not an equal minimum,

we query μ_ℓ again since it is the first block with a smaller minimum since either that block may have an index j such that $\mathsf{LCP}^\pi[j] = \mathsf{LCP}^\pi[k]$ or there is no such index with an equal longest common prefix for a $j \geq \mathsf{SN}[i]$ as required by the definition of Leftmost. In this case, $\mathsf{Leftmost}[k] = k$. Again, since we have n indices that require a single processor query, we can use Algorithm 16 to build the Leftmost array in constant rounds. We formally give the suffix tree construction by Algorithm 12 below.

Algorithm 22: SuffixTree(π, s)

1 Distribute blocks of the suffix array from π such that processor μ_α has $\pi_{\alpha n^\varepsilon}, \pi_{\alpha n^\varepsilon+1}, ..., \pi_{(\alpha+1)n^\varepsilon-1}$ for all $0 \leq \alpha < n^\varepsilon$;
2 $D_{s,s} \leftarrow \mathsf{LCPInit}(s, s)$;
3 $\mathsf{LCP}^\pi[j] \leftarrow D_{s,s}.\mathsf{query}(\pi_j, \pi_{j-1})$ for all $0 \leq j < n$;
4 $m_\alpha \leftarrow \min_{j \in [\alpha n^{1-\varepsilon}, (\alpha+1)n^{1-\varepsilon}]}(\mathsf{LCP}^\pi[j])$ for all $0 \leq \alpha < n^\varepsilon$;
5 $\mathscr{L}, \mathscr{R} \leftarrow$Build$\mathscr{L}\mathscr{R}(\mathsf{LCP}^\pi, \{m_\alpha | 0 \leq \alpha < n^\varepsilon\})$;
6 $P \leftarrow \{(k, \mathscr{L}[k], \mathscr{R}[k], \mathsf{LCP}^\pi[\mathscr{L}[k]], \mathsf{LCP}^\pi[\mathscr{R}[k]]) | 0 \leq k < n\}$;
7 $\mathsf{SN} \leftarrow \mathsf{BuildSN}(P)$;
8 $\mathsf{Leftmost} \leftarrow \mathsf{BuildLeftmost}(\mathsf{LCP}^\pi, \mathsf{SN})$;
9 **for** $0 \leq k < n$ **do**
10 Add leaf node (k) to suffix tree;
11 **if** $k ==$ *Leftmost*$[k]$ **then**
12 Add internal node $(k, \mathsf{LCP}^\pi[k])$ to suffix tree;

Using Algorithm 12 and all its subroutines, we then can prove our main result.

Theorem E.1.5. *For $\varepsilon \in (0, .5]$, there is an $O(1/\varepsilon)$-round MPC algorithm for computing the* **suffix tree** *of a given string s with $\widetilde{O}(n^{1+\varepsilon})$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ memory per processor with high probability.*

Proof. We initialize the LCPQ oracle for s with itself in step 2 and then perform $O(n)$ queries to find the LCP between all pairs of consecutive suffixes in the suffix array in step 3. By Theorem E.1.1, the LCP initialization and $O(n)$ LCP queries can be performed in $O(1)$ rounds and $\tilde{O}(n^{1+\varepsilon})$ total memory. Each processor μ_α will be in charge of storing the $O(n^{1-\varepsilon})$ values of $\mathsf{LCP}^\pi[j]$ for all indices j in its block of the suffix array. In step 4,

we use these LCP^{π} values to find the minimum LCP of two consecutive suffixes per suffix array block, which can be done locally on each block machine and then sent to all other machines in a single round since each of the $n^{\varepsilon} < n^{1-\varepsilon}$ machines only sends a single value to all other machines. Next, we call a subroutine described in the following paragraph to fill in the $\mathscr{L}$ and $\mathscr{R}$ as defined previously.

The below algorithm, Algorithm 28, finds $\mathscr{L}$ and $\mathscr{R}$ in $O(n)$ total space and $O(1)$ rounds. First, we will discuss building the $\mathscr{L}$ array in steps 3-8 and 20-22, where $\mathscr{L}[k]$ contains the index of the nearest pair of consecutive suffixes in the suffix array to the left of k that has a smaller LCP than that $\mathsf{LCP}^{\pi}[k]$. The $\mathscr{R}$ array has the same steps but mirrored to find the nearest pair of suffixes with a smaller LCP to the right of k and can be seen in steps 9-14 and 23-24. First, for each index k, we check all LCP^{π} values for an LCP smaller than $\mathsf{LCP}^{\pi}[k]$ within the block containing k only for indices to the left of k. This is performed locally on a single processor. If $\mathscr{L}[k]$ is not found in the same block as k, we then check the minimum LCP^{π} values of neighboring blocks to the left of the block containing k using our block-based data structure until we find the nearest left block with a minimum LCP^{π} smaller than $\mathsf{LCP}^{\pi}[k]$. Since the minimum LCP^{π} value of this block is smaller than $\mathsf{LCP}^{\pi}[k]$, we are guaranteed to find $\mathscr{L}[k]$ at this block. We add requests to a request set so that we can query these left blocks for $\mathscr{L}[k]$ for all indices k in parallel. Before actually sending the queries, we use $\mathsf{WeightedLoadBalancing}$ to make sure each processor can handle its requests and copy any processor when needed. Since each index k for $0 \leq k < n$ only has a single query to find $\mathscr{L}$, then there are only $O(n)$ queries total to find the $\mathscr{L}$ and $\mathscr{R}$ arrays. Therefore, steps 17-25 only requires $O(1)$ rounds and $O(\tilde{n^{1-\varepsilon}})$ space per processor after $\mathsf{WeightedLoadBalancing}$. To handle each $\mathscr{L}$ query in steps 20-22, a processor is given the index k and its $\mathsf{LCP}^{\pi}[k]$ and can compare the $\mathsf{LCP}^{\pi}[k]$ locally to its own values of LCP until it finds the maximum index in its suffix array block with a smaller LCP than $\mathsf{LCP}^{\pi}[k]$. The maximum index with a smaller LCP is also the rightmost index,

i.e. we find $\mathscr{L}[k]$ which is the closest index to k with a smaller LCP than $\mathsf{LCP}^\pi[k]$. As mentioned previously, it is easy to check that the steps to find $\mathscr{R}[k]$ mirror those to find $\mathscr{L}[k]$ and require the same space and number of rounds.

Algorithm 23: Build$\mathscr{L}\mathscr{R}(\mathsf{LCP}^\pi, \{m_\alpha | 0 \le \alpha < n^\varepsilon\})$

$F_\alpha \leftarrow \emptyset$ for $0 \le \alpha < n^\varepsilon$;
for $0 \le \alpha < n^\varepsilon$ **do**
  // in machine μ_α
  for $k \in [\alpha n^{1-\varepsilon}, (\alpha+1)n^{1-\varepsilon}]$ **do**
    if $\exists k' \in [\alpha n^{1-\varepsilon}, k)$ *s.t.* $LCP^\pi[k'] < LCP^\pi[k]$ **then**
      $\mathscr{L}[k] \leftarrow \max_{k' \in [\alpha n^{1-\varepsilon}, k)} \{k' : \mathsf{LCP}^\pi[k'] < \mathsf{LCP}^\pi[k]\}$;
    else
      $\ell \leftarrow \max_{\beta < \alpha} \{\beta : m_\beta < m_\alpha\}$;
      Add $(k, \mathsf{LCP}^\pi[k])$ to F_ℓ;
    end
    if $\exists k' \in (k, (\alpha+1)n^{1-\varepsilon})$ *s.t.* $LCP^\pi[k'] < LCP^\pi[k]$ **then**
      $\mathscr{R}[k] \leftarrow \min_{k' \in (k, (\alpha+1)n^{1-\varepsilon})} \{k' : \mathsf{LCP}^\pi[k'] < \mathsf{LCP}^\pi[k]\}$;
    else
      $r \leftarrow \min_{\beta > \alpha} \{\beta : m_\beta < m_\alpha\}$;
      Add $(k, \mathsf{LCP}^\pi[k])$ to F_r;
    end
  end
end
$\{c_\alpha \mid 0 \le \alpha < n^\varepsilon\} \leftarrow \mathsf{WeightedLoadBalancing}(\{\mu_\alpha | 0 \le \alpha < n^\varepsilon\}, \{F_\alpha | 0 \le \alpha < n^\varepsilon\})$;
for $0 \le \alpha < n^\varepsilon, 1 \le z \le c_\alpha$ **do**
  // in machine $\mu_\alpha^{(z)}$
  for $(k, LCP^\pi[k]) \in F_\alpha^{(z)}$ **do**
    if $k > (\alpha+1)n^{1-\varepsilon}$ **then**
      $\mathscr{L}[k] \leftarrow \max_{k' \in [\alpha n^{1-\varepsilon}, k)} \{k' : \mathsf{LCP}^\pi[k'] < \mathsf{LCP}^\pi[k]\}$;
    else
      $\mathscr{R}[k] \leftarrow \min_{k' \in (k, (\alpha+1)n^{1-\varepsilon})} \{k' : \mathsf{LCP}^\pi[k'] < \mathsf{LCP}^\pi[k]\}$;
    end
  end
end
Return $\mathscr{L}, \mathscr{R}$;

Using the $\mathscr{L}$, $\mathscr{R}$, and LCP^π arrays, we can build the SN array according to its definition.

We call another subroutine Algorithm 8 to build the SN array with these inputs in steps 7-8 of the suffix tree construction. Each processor μ_α can find and store the block of SN for the suffixes in its block of the suffix array. Therefore, this step trivially runs in constant rounds, $O(n^{1-\varepsilon})$ space per machine, and $O(n)$ total memory.

Algorithm 24: BuildSN($P = \{(k, \mathscr{L}[k], \mathscr{R}[k], \mathsf{LCP}^\pi[\mathscr{L}[k]], \mathsf{LCP}^\pi[\mathscr{R}[k]]) | 0 \leq k < n\}$)

```
1 for (k, L[k], R[k], LCP^π[L[k]], LCP^π[R[k]]) ∈ P do
2 |   if LCP^π[L[k]] ≥ LCP^π[R[k]] then
3 |   |   SN[k] ← L[k];
4 |   else
5 |   |   SN[k] ← R[k];
6 |   end
7 end
8 return SN;
```

After building the SN array, we can construct the final array before making the suffix tree, the Leftmost array. In step 9, we call the BuildLeftmost subroutine, described below, which builds Leftmost according to its definition using similar steps to the Build$\mathscr{L}\mathscr{R}$ subroutine.

Algorithm 15 begins in steps 3 and 4 for a suffix index k by setting any index with $\mathsf{LCP}^\pi[k] = 0$ to 0, which is the correct value by the definition of the Leftmost array. If $\mathsf{LCP}^\pi[k] \neq 0$, let b_i be the block containing k of size $n^{1-\varepsilon}$, i.e. $i = \lceil \frac{k}{n^{1-\varepsilon}} \rceil$. Then, in steps 5 and 6, the algorithm searches for the farthest block b_j from b_i of length $n^{1-\varepsilon}$ with a minimum LCP^π value equal to $\mathsf{LCP}^\pi[k]$ between b_k and the block containing $\mathsf{SN}[k]$. If $\mathsf{SN}[k]$ is to the right of k, no such block exists and we know that $\mathsf{Leftmost}[k] = k$ by definition which is fulfilled in step 9. If $\mathsf{SN}[k]$ is to the left of $\mathsf{LCP}^\pi[k]$, we know that the block containing $\mathsf{SN}[k]$ is the farthest block to the left of b_i with a smaller minimum LCP^π value than $\mathsf{LCP}[k]$. Thus, if we do not find such a block b_j with a minimum LCP^π equal to $\mathsf{LCP}[k]$, we know no such block exists between b_i and the block containing $\mathsf{SN}[k]$ and

again in step 9 we set Leftmost$[k]$ to k. If we find such a block b_j, then in steps 6 and 7 we add a query to be handled later to find Leftmost on block b_j in processor μ_j^π. Step 10 calls AdjustRepetitions2 to make sure each processor only needs to handle $O(n^{1-\varepsilon})$ queries out of the total $O(n)$ queries, and then steps 13 and 14 find Leftmost$[k]$ according to its definition by finding the leftmost index on block b_j for index k with an LCP^π value equal to $\mathsf{LCP}^\pi[k]$. Since each index k only requires a single query, AdjustRepetitions2 will successfully allow all $O(n)$ potential queries to be handled in $O(1)$ rounds, total space $O(n)$, and $O(n^{1-\varepsilon})$ memory per processor. Thus, BuildLeftmost will build the leftmost array in the desired space and rounds.

Algorithm 25: BuildLeftmost($\mathsf{LCP}^\pi, \mathsf{SN}$)

1 $F_\alpha \leftarrow \emptyset$ for $0 \le \alpha < n^\varepsilon$;
2 **for** $0 \le \alpha < n^\varepsilon$ **do**
  // in machine μ_α
3 **for** $k \in [\alpha n^{1-\varepsilon}, (\alpha+1)n^{1-\varepsilon})$ **do**
4 **if** $LCP^\pi[k] == 0$ **then**
5 Leftmost$[k] \leftarrow 0$;
6 **else if** $\exists \beta, \lceil \frac{SN[k]}{n^{1-\varepsilon}} \rceil \le \beta < \lceil \frac{k}{n^{1-\varepsilon}} \rceil$ *s.t.* $m_\beta = LCP^\pi[k]$ **then**
7 $\beta' \leftarrow \min_{\beta \in [\lceil \frac{\mathsf{SN}[k]}{n^{1-\varepsilon}} \rceil, \lceil \frac{k}{n^{1-\varepsilon}} \rceil)}$;
8 Add $(k, \mathsf{LCP}^\pi[k])$ to $F_{\beta'}$;
9 **else**
10 Leftmost$[k] \leftarrow k$;
11 $\{c_\alpha \mid 0 \le \alpha < n^\varepsilon\} \leftarrow$ WeightedLoadBalancing$(\{\mu_\alpha | 0 \le \alpha < n^\varepsilon\}, \{F_\alpha | 0 \le \alpha < n^\varepsilon\})$;
12 **for** $0 \le \alpha < n^\varepsilon, 1 \le z \le c_\alpha$ **do**
  // in machine $\mu_\alpha^{(z)}$
13 **for** $(k, LCP^\pi[k]) \in F_\alpha^{(z)}$ **do**
14 Leftmost$[k] \leftarrow \min_{j \in [\alpha n^{1-\varepsilon}, (\alpha+1)n^{1-\varepsilon}]} \{\mathsf{LCP}^\pi[j] = \mathsf{LCP}^\pi[k]\}$;
15 return Leftmost;

With the addition of the SN and Leftmost arrays, we may build the suffix tree according to [87]. We simply add internal nodes for every unique Leftmost suffix. Edges are detailed in Lemma E.3.1 and are implicitly already provided to us using the LCP^π, SN, and Leftmost

arrays.

□

Corollary E.3.2. *Given a string s, its suffix array π, and its LCP^{π} array, we can construct the suffix tree of s in MPC with $O(1)$ rounds and $O(n)$ total memory.*

Proof. This corollary is a natural result of the previous proof. Note that to build a suffix tree from π and LCP^{π} in MPC, we only need to run the $\mathsf{Build}\mathscr{L}\mathscr{R}$, $\mathsf{BuildSN}$, and $\mathsf{BuildLeftmost}$ subroutines. All three of these algorithms use $\mathsf{WeightedLoadBalancing}$ to run in $O(1)$ rounds and $O(n)$ total memory. □

E.3.2 LCS with Suffix Tree in MPC

We will now use our suffix tree structure augmented with the $\mathsf{LCP}^{\pi}, \mathscr{L}, \mathscr{R}, \mathsf{SN}, \mathsf{Leftmost}$, and suffix arrays to solve the LCS and LPS applications in $O(1/\varepsilon)$ rounds and $\widetilde{O}(n^{1+\varepsilon})$ total memory. Our algorithms for these problems utilizing the suffix tree in MPC are different than prior sequential solutions due to an inability to perform tree searches in constant rounds. To overcome this issue, we look at each internal node of the suffix tree and find its descendants using the ordering of the suffix array and block-based data structures. We will formally state this idea for the LCS application first in this section.

Given two input strings s and s' for the LCS problem, we first follow the steps of a typical suffix tree solution and concatenate the strings with a special character such as \$ to separate the strings. For example, given input strings $s =$ "apple" and $s' =$ "banana", we build the suffix tree on concatenated string "apple\$banana". Once we have our suffix tree, we then search the internal nodes of the tree in parallel for the LCS; note there may be at most $O(n)$ internal nodes, one per suffix. Any internal node represents the longest common ancestor between at least two suffixes in the suffix tree. If any internal node is the ancestor of suffixes from both strings s and s', then it corresponds to a common substring between s

and s', and we therefore want to find the internal node corresponding to the longest common substring. However, to check if an internal node is an ancestor of a suffix from each input string typically requires a search of the tree, e.g. a depth-first search. Unfortunately, since the suffix tree can have a depth of $O(n)$, a depth-first search could require up to $O(n^{\varepsilon})$ rounds of communication, one per processor. We avoid a depth-first search entirely by using the underlying suffix array and the additional suffix information we computed when converting suffix array to suffix tree. Given an internal node $(k, \mathsf{LCP}^{\pi}[k])$ of the suffix tree, we observe that the range of indices in the suffix array that are descendants of this internal node is $(\mathscr{L}[k], \mathscr{R}[k])$.

Observation E.3.1. *Given internal node $v = (k, \mathsf{LCP}^{\pi}[k])$, a leaf node $u = (j)$ is a descendant of v if and only if $\mathscr{L}[k] \leq j < \mathscr{R}[k]$.*

The previous statement follows since any suffix with index k' within this range has $\mathsf{LCP}^{\pi}[k'] > \mathsf{LCP}^{\pi}[k]$, which implies that the k'th suffix indeed contains the prefix of length $\mathsf{LCP}^{\pi}[k]$ corresponding to the internal node $(k, \mathsf{LCP}^{\pi}[k])$. Note that only one side is inclusive because the $\mathsf{LCP}^{\pi}[k]$ is defined as the longest common prefix between π_k and π_{k-1}.

By keeping track of whether or not each block of the suffix array has a suffix from each string, we can easily find the lowest internal node on each processor μ_{α} that has a leaf descendant from each input string and then take the lowest internal node across all processors to find the longest common substring. To this end, we will simply construct an additional array $\mathscr{B}$ that has a value of 1 if an internal node of the suffix tree has a child from both strings to complete the algorithm.

With the above algorithm that relies on our suffix tree construction, we have the following corollary:

Corollary E.1.6. (LCS using Suffix Tree). *For $\varepsilon \in (0, .5]$, there is an $O(1/\varepsilon)$-round MPC algorithm for computing the* LCS *of two input strings s, s' with $\widetilde{O}(n^{1+\varepsilon})$ total memory and*

Algorithm 26: $\mathsf{LCS}(s = \{s_0 s_1 ... s_{n-1}\}, s' = \{s'_0 s'_1 ... s'_{n-1}\})$

Let $t = \{s_0 s_1 ... s_{n-1} \$ s'_0 s'_1 ... s'_{n-1}\}$;
$\pi \leftarrow \mathsf{SuffixArray}(t)$;
$\mathsf{SuffixTree}(\pi, t)$;
$\mathsf{Build}\mathscr{B}()$;
$mx_\alpha, mxk_\alpha \leftarrow 0$ for $0 \leq \alpha < n^\varepsilon$;
for *internal nodes* $(k, LCP^\pi[k])$ **do**
    // On processor $\mu_{\lfloor \frac{k}{n^{1-\varepsilon}} \rfloor}$
    $\alpha \leftarrow \lfloor \frac{k}{n^{1-\varepsilon}} \rfloor$;
    if $\mathscr{B}[k] == 1$ *AND* $LCP^\pi[k] > mx_\alpha$ **then**
        $mx_\alpha \leftarrow \mathsf{LCP}^\pi[k]$;
        $mxk_\alpha \leftarrow k$;
    end
end
Return mx_α, mxk_α such that $mx_\alpha = \max_{0 \leq beta < n^\varepsilon}\{mx_\beta\}$;

$\widetilde{O}(n^{1-\varepsilon})$ *memory per processor with high probability.*

Note that the construction of $\mathscr{B}$ is written below in Algorithm 27. The construction of $\mathscr{B}$ is long but straightforward. First, a block-based data structure is created and stored on all block machines that stores whether or not a block has a suffix from s and $\bar{s}$. Then, for each internal node $(k, \mathsf{LCP}^\pi[k])$, we simply check if any blocks in the range of blocks between $\mathscr{L}[k], \mathscr{R}[k]$ have a suffix from s and $\bar{s}$ using the block-based data structure. If not, we check the specific blocks containing $\mathscr{L}[k]$ and $\mathscr{R}[k]$ as a last chance and set $\mathscr{B}[k]$ accordingly.

E.3.3 LPS with Suffix Tree in MPC

For our second suffix tree application, we solve the LPS problem in $O(1/\varepsilon)$ rounds and $\widetilde{O}(n^{1+\varepsilon})$ total memory. Once again, we use the suffix tree structure augmented with the same arrays created during the suffix tree construction. Instead of building a suffix tree on two concatenated input strings as in LCS, we build the suffix tree on the single

Algorithm 27: Build$\mathscr{B}()$

for $0 \leq \alpha < n^{\varepsilon}$ **do**
  // On processor μ_{α}
  $h_{\alpha}, \overline{h_{\alpha}} \leftarrow 0$;
  if $\exists k \in (\alpha n^{1-\varepsilon}, (\alpha+1)n^{1-\varepsilon})$ *such that* $\$ \in s[\pi_k : n)$ **then**
    $h_{\alpha} \leftarrow 1$;
  end
  if $\exists k \in (\alpha n^{1-\varepsilon}, (\alpha+1)n^{1-\varepsilon})$ *such that* $\$ \notin s[\pi_k : n)$ **then**
    $\overline{h_{\alpha}} \leftarrow 1$;
  end
  Send $h_{\alpha}, \overline{h_{\alpha}}$ to all processors μ_{β} for $0 \leq \beta < n^{\varepsilon}$;
end
$F_{\alpha} \leftarrow \emptyset$ for $0 \leq \alpha < n^{\varepsilon}$;
for *internal nodes* $(k, LCP^{\pi}[k])$ **do**
  // On processor $\mu_{\lfloor \frac{k}{n^{1-\varepsilon}} \rfloor}$
  $\alpha \leftarrow \lfloor \frac{k}{n^{1-\varepsilon}} \rfloor$;
  $f_k, \overline{f_k} \leftarrow 0$;
  if $\exists \beta, \lfloor \frac{\mathscr{L}[k]}{n^{1-\varepsilon}} \rfloor < \beta < \lfloor \frac{\mathscr{R}[k]}{n^{1-\varepsilon}} \rfloor$ *s.t.* $h_{\beta} = 1$ **then**
    $f_k \leftarrow 1$;
  end
  if $\exists \beta, \lfloor \frac{\mathscr{L}[k]}{n^{1-\varepsilon}} \rfloor < \beta < \lfloor \frac{\mathscr{R}[k]}{n^{1-\varepsilon}} \rfloor$ *s.t.* $\overline{h_{\beta}} = 1$ **then**
    $\overline{f_k} \leftarrow 1$;
  end
  Add $(k, \mathscr{L}[k], f_k, \overline{f_k})$ to $F_{\lfloor \frac{\mathscr{L}[k]}{n^{1-\varepsilon}} \rfloor}$;
  Add $(k, \mathscr{R}[k], f_k, \overline{f_k})$ to $F_{\lfloor \frac{\mathscr{R}[k]}{n^{1-\varepsilon}} \rfloor}$;
end
$\{c_{\alpha} \mid 0 \leq \alpha < n^{\varepsilon}\} \leftarrow$ WeightedLoadBalancing$(\{\mu_{\alpha} | 0 \leq \alpha < n^{\varepsilon}\}, \{F_{\alpha} | 0 \leq \alpha < n^{\varepsilon}\})$;
for $0 \leq \alpha < n^{\varepsilon}, 1 \leq z \leq c_{\alpha}$ **do**
  // in machine $\mu_{\alpha}^{(z)}$
  for $(k, o, f_k, f'_k) \in F_{\alpha}^{(z)}$ **do**
    if $\exists j$ *s.t.* $\$ \in s[\pi_j : n)$ *for* $\alpha n^{1-\varepsilon} \leq j < o$ *OR* $o \leq j < (\alpha+1)n^{1-\varepsilon}$ **then**
      $f_k \leftarrow 1$;
    end
    if $\exists j$ *s.t.* $\$ \notin s[\pi_j : n)$ *for* $\alpha n^{1-\varepsilon} \leq j < o$ *OR* $o \leq j < (\alpha+1)n^{1-\varepsilon}$ **then**
      $\overline{f_k} \leftarrow 1$;
    end
    Send $(f_k, \overline{f_k})$ to $\mu_{\lfloor \frac{k}{n^{1-\varepsilon}} \rfloor}$;
  end
end
for *internal nodes* $(k, LCP^{\pi}[k])$ **do**
  // On processor $\mu^{\pi}_{\lfloor \frac{k}{n^{1-\varepsilon}} \rfloor}$
  $\mathscr{B}[k] = 0$;
  if $f_k = 1$ *AND* $\overline{f_k} = 1$ **then**

input string s concatenated with the reverse of the input string $\bar{s}$. Any internal node that is an ancestor of both a suffix from s starting at index i and a suffix from $\bar{s}$ starting at $n-i$ corresponds to a palindrome since it is a common prefix of the substring to the left and right of index i in s. Unfortunately, as in the LCS suffix tree solution, we cannot perform efficient tree searches in constant rounds and small memory to find the lowest internal node in a sequential fashion. Instead, we will once again search all internal nodes of the suffix tree in parallel with an additional array $\mathscr{P}$ that we must construct that will answer queries for whether or not an internal node corresponds to a palindrome of s or not, i.e. is the ancestor of a suffix and its correlated reverse suffix.

Algorithm 28: LPS($s = \{s_1 s_2 ... s_n\}$

Let $t = \{s_1 s_2 ... s_n \$ s_n s_{n-1} ... s_1\}$;
$\pi \leftarrow$ SuffixArray(t);
SuffixTree(π, t);
Build$\mathscr{P}$();
$mx_i, mxk_i \leftarrow 0$ for $0 \leq i < n^{\varepsilon}$;
for *internal nodes* $(k, LCP^{\pi}[k])$ **do**
    // On processor $\mu^{\pi}_{\lfloor \frac{k}{n^{1-\varepsilon}} \rfloor}$
    $i \leftarrow \lfloor \frac{k}{n^{1-\varepsilon}} \rfloor$;
    if $\mathscr{P}[k] == 1$ *AND* $LCP^{\pi}[k] > mx_i$ **then**
        $mx_i \leftarrow \mathsf{LCP}^{\pi}[k]$;
        $mxk_i \leftarrow k$;
    end
end
Return mx_i, mxk_i such that $mx_i = \max_{0 \leq j < n^{\varepsilon}} \{mx_j\}$;

With Algorithm 28, which relies on our suffix tree construction and subroutine Algorithm 29, we have the following corollary:

Corollary E.1.7. (LPS using Suffix Tree). *For $\varepsilon \in (0, .5]$, there is an $O(1/\varepsilon)$-round MPC algorithm for computing the* LPS *of an input strings s with $\widetilde{O}(n^{1+\varepsilon})$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ memory per processor with high probability.*

Note that to build $\mathscr{P}$ we follow a similar set of steps in Algorithm 29 to the construction of $\mathscr{B}$ for LCS. For an internal node $(k, \mathsf{LCP}^{\pi}[k])$, instead of checking for a suffix descendant of both strings between $\mathscr{L}[k]$ and $\mathscr{R}[k]$, we simply check for a suffix and its reverse suffix between $\mathscr{L}[k]$ and $\mathscr{R}[k]$.

Algorithm 29: Build$\mathscr{P}()$

$\mathscr{P}[k] \leftarrow 0$ for $0 \leq k < n$;
for $0 \leq \alpha < n^{\varepsilon}$ **do**
    // On processor μ_{α}
    $\ell_{\alpha} \leftarrow \max\{j | \exists k \in [\alpha n^{1-\varepsilon}, (\alpha+1)n^{1-\varepsilon})$ s.t. $j < k$ and $\pi_j = n - \pi_k\}$;
    $r_{\alpha} \leftarrow \min\{j | \exists k \in [\alpha n^{1-\varepsilon}, (\alpha+1)n^{1-\varepsilon})$ s.t. $j > k$ and $\pi_j = n - \pi_k\}$;
    Send ℓ_{α} and r_{α} to all processors;
$F_j \leftarrow \emptyset$ for $0 \leq j < n^{\varepsilon}$;
for *internal nodes* $(k, \mathsf{LCP}^{\pi}[k])$ **do**
    // On processor $\mu_{\lfloor \frac{k}{n^{1-\varepsilon}} \rfloor}$
    $\alpha \leftarrow \lfloor \frac{k}{n^{1-\varepsilon}} \rfloor$;
    $\alpha_{\ell} \leftarrow \lfloor \frac{\mathscr{L}[k]}{n^{1-\varepsilon}} \rfloor$;
    $\alpha_r \leftarrow \lfloor \frac{\mathscr{R}[k]}{n^{1-\varepsilon}} \rfloor$;
    if $\exists \beta \in (\alpha_{\ell}, \alpha_r)$ *s.t.* $\mathscr{L}[k] \leq \ell_{\beta} \leq \mathscr{R}[k]$ *or* $\mathscr{L}[k] \leq r_{\beta} \leq \mathscr{R}[k]$ **then**
        $\mathscr{P}[k] \leftarrow 1$;
    Add $(k, \mathscr{L}[k], \mathscr{R}[k])$ to $F_{\alpha_{\ell}}, F_{\alpha_r}$;
$\{c_{\alpha} \mid 0 \leq \alpha < n^{\varepsilon}\} \leftarrow$ WeightedLoadBalancing$(\{\mu_{\alpha}^{\pi} | 0 \leq \alpha < n^{\varepsilon}\}, \{F_{\alpha} | 0 \leq \alpha < n^{\varepsilon}\})$;
for $0 \leq \alpha < n^{\varepsilon}, 1 \leq z \leq \alpha$ **do**
    // in machine $\mu_{\alpha}^{(z)}$
    for $(k, \mathscr{L}[k], \mathscr{R}[k]) \in F_{\alpha}^{(z)}$ **do**
        if $\exists j \in [\alpha n^{1-\varepsilon}, (\alpha+1)n^{1-\varepsilon}]$ *s.t.* $\pi_j = n - \pi_k$ *and* $\mathscr{L}[k] < j, k < \mathscr{R}[k]$ **then**
            $\mathscr{P}[k] \leftarrow 1$;

E.4 Compressed LCPQ Oracle

In this section, we study instances of the LCPQ oracle in which there are a smaller collection of queries $\mathscr{Q} = \{q_1, q_2, \ldots, q_k\}$, for example $k = O(n)$ as opposed to $O(n^{1+\varepsilon})$

which we considered in the previous part. 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 the values of k, we can can 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 E.3 as well as in Figure E.2.

In Lemma E.4.1, we present an alternative approach for solving a batch of k LCP queries. The total memory used by this approach equals $\widetilde{O}(n+kn^{\varepsilon})$ instead of $\widetilde{O}(n^{1+\varepsilon}+k)$ total memory of Theorem E.1.1. This implies a constant round MPC algorithm with total memory $\widetilde{O}(n+k+\min(n,k)\cdot n^{\varepsilon})$ if we use Theorem E.1.1 when $k=\Omega(n)$ and Lemma E.4.1 when $k=O(n)$. Thus, the algorithm uses $\widetilde{O}(n+k+\min(n,k)\cdot n^{\varepsilon})$ total memory since it needs to store the strings and queries.

Corollary E.1.2. *For $\varepsilon \in (0,.5]$, there is an $O(1)$-round MPC algorithm $\widetilde{O}(n^{1-\varepsilon})$ memory per machine which initializes an* LCPQ **oracle** *in $O(1)$ rounds w.h.p., and then processes a collection of k queries, $\mathcal{Q}=\{q_1,q_2,\ldots,q_k\}$, in $O(1)$ rounds. The total memory used by this algorithm is $\widetilde{O}(n+k+\min(n,k)\cdot n^{\varepsilon})$.*

However, for the values of k in the range from $\omega(n^{1-\varepsilon})$ to $o(n^{1+\varepsilon})$, we still get a sub-optimal total memory of $O\big(n+k+\min(n,k)\cdot n^{\varepsilon}\big)=\omega(n+k)$. 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 Lemma E.4.2 for more details). Our memory-optimal algorithms for different ranges of k are demonstrated in Table E.3. We start by showing the algorithm with $\widetilde{O}(n+k\cdot n^{1+\varepsilon})$ memory and a constant number of rounds that handles a batch of k queries.

#Queries	Source	Rounds	Total Memory
$k = O(n^{1-\varepsilon})$	Lemma E.4.1	$O(1)$	$O(n + k \cdot n^{\varepsilon}) = O(n)$
$k = \omega(n^{1-\varepsilon})$, and $k = o(n^{1+\varepsilon})$	Lemma E.4.1	$O(1)$	$O(n + k + \min(n,k) \cdot n^{\varepsilon})$
	Lemma E.4.2	$O(\log\log n)$	$O(n+k)$
$k = \Omega(n^{1+\varepsilon})$	Theorem E.1.1	$O(1)$	$O(n^{1+\varepsilon} + k) = O(k)$

Table E.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.

Lemma E.4.1. *For $0 < \varepsilon < 1/2$, there is an $O(1)$-round MPC algorithm, with $\widetilde{O}(n + k \cdot n^{1+\varepsilon})$ total memory and $\widetilde{O}(n^{\varepsilon})$ memory per machine, which initiates the* LCPQ **Oracle** *in $O(1)$ rounds w.h.p., and then computes a collection of k queries, $\mathscr{Q} = \{q_1, q_2, \ldots, q_k\}$, in $O(1)$ rounds.*

Proof. Recall from our observation at the start of Section E.2, we can assume the length of the queries in $\mathscr{Q}$ is bounded by $O(n^{\varepsilon})$. We need a total memory of $O(n + k \cdot n^{\varepsilon})$ to store every substring of length $O(n^{\varepsilon})$ starting at endpoints i and j (in s and s' respectively) for every query $q = (i, j) \in \mathscr{Q}$. Once we gathered the required information, we solve the LCP queries locally by comparing the characters of $s\big[i : \min(i + n^{\varepsilon}, n)\big)$ and $s'0\big[j : \min(j + n^{\varepsilon}, n)\big)$ one by one. □

Lemma E.4.2. *For $0 < \varepsilon < 1/2$, there is an $O(\log\log n)$-round MPC algorithm, with $\widetilde{O}(n + k)$ total memory and $\widetilde{O}(n^{\varepsilon})$ memory per machine, which uses the* LCPQ **Oracle** *to compute a collection of k queries, $\mathscr{Q} = \{q_1, q_2, \ldots, q_k\}$, w.h.p.*

Proof. Again, we know from observation from Section E.2 that the solution of every query $q = (i, j) \in \mathscr{Q}$ is bounded by $O(n^{\varepsilon})$. Let $\mathscr{L}$ denote the guarantee on the maximum length of the current queries. We use an iterative approach that at each step reduces the current guarantee on the length of LCP queries from $O(\mathscr{L})$ to $O(\sqrt{\mathscr{L}})$. It is clear that after $O(\log\log n)$ iterations, the guarantee on the size of LCP queries collapses to $O(1)$. We use the ideas of Lemma E.4.1 to solve the problem using $O(n+k)$ at that point to solve the problem locally.

Recall the *modular partitioning* technique from Section E.1.3. To reduce the current guarantee from $\mathscr{L}$ to $O(\sqrt{\mathscr{L}})$ at each iteration, we perform a modular partitioning modulo $O(\sqrt{\mathscr{L}})$. Since the total number of characters on s and s' which are within substrings of length $O(\mathscr{L})$ starting from i or j for any query $q = (i, j)$ is bounded by $O(n)$, we need to query the hash of a total of at most $O(n)$ substrings either to find out that $\mathsf{LCP}_{s,s'}(i, j) \leq \sqrt{\mathscr{L}}$ or to reduce the guarantee to $O(\sqrt{\mathscr{L}})$. We use a *block-based* data structure such that $f\big(s[l : r)\big) = \mathsf{hash}\big(s[l : r)\big)$ to handle the interval hash queries. Utilizing the computed modular partitioning, we estimate the length of each LCP with an additive error of $O(\sqrt{\mathscr{L}})$. This imply that we can reduce our guarantee on the maximum LCP among all queries in $\mathscr{Q}$ to $O(\sqrt{\mathscr{L}})$ by replacing each query $q = (i, j)$ by $q' = (i', j') = (i + \kappa, j + \kappa)$, where

$$\kappa = \left\lfloor \frac{\mathsf{LCP}_{s,s'}(i, j)}{\sqrt{\mathscr{L}}} \right\rfloor \cdot \sqrt{\mathscr{L}}.$$

□

E.5 Longest Palindrome in MPC

In this section, we give an improved solution to emphLongest Palindrome Search (LPS). Given a string $s = s_0 s_1 ... s_{n-1}$, let $\overline{s} = s_{n-1} ... s_1 s_0$ 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 $c = \overline{c}$. 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 33 below, will use $O(n^{\varepsilon})$ processors each with $\widetilde{O}(n^{1-\varepsilon})$ memory for $\varepsilon < .5$ 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 $O(n^{1-\varepsilon})$ 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 [9], will be split into several steps. At the start of our main algorithm (Algorithm 33), we break the input string s into blocks b_α of size $n^{1-\varepsilon}$ where $n = |s|, 0 \leq \alpha < n^{\varepsilon}$. We then utilize the following algorithm, Algorithm 30, 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 $\overline{s}$ and return the longest palindrome.

The first algorithm we formally write, Algorithm 30, 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 30 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 $2n^{1-\varepsilon}-1$ 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 $2^\varepsilon - 1$, 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 30 correctly finds minimum length period strings in space $O(n^{1-\varepsilon})$ given blocks of size $n^{1-\varepsilon}$.

Lemma E.5.1. *On input $t = b_{\alpha-1}b_\alpha b_{\alpha+1}$ with $|b_{\alpha-1}| = |b_\alpha| = |b_{\alpha+1}| = n^{1-\varepsilon}$, Algorithm 30 will return w,C where C contains all indices of b_α that are centers of palindromes of length at least $2n^{1-\varepsilon}+1$ and w is the minimum length period string if $|C| \geq 2$.*

Proof. On input $t = b_{\alpha-1}b_\alpha b_{\alpha+1}$ with $|b_{\alpha-1}| = |b_\alpha| = n^{1-\varepsilon}$, any index j such that $n^\varepsilon \leq j < 2n^\varepsilon$ implies that $t[j : j+1) \in b_\alpha$. Thus, any palindrome candidate index added to C in step 5 of Algorithm 30 will be an index of b_α. Furthermore, index j is added to C if and only if $t[j-n^\varepsilon : j) = t[j+n^\varepsilon, j)$, i.e. j is a palindrome center with palindrome length $2n^\varepsilon + 1$.

From Lemma 3.3 of [9], $|C| \geq 2$ implies that block b_α is periodic, and for any $j,k \in C, j < k$, there is a period of length $l = 2(k-j)+1$. Thus, the minimum period length will be the GCD of all period lengths for any index pair of C, which we denote with l_0 in step

Algorithm 30: LPSCandidate($t = b_{\alpha-1}b^{\alpha}b^{\alpha+1}$)

Initialize the list of LPS candidate indices to the empty set, $C \leftarrow \emptyset$;
// Find palindrome center candidates in $b\alpha$
for $n^{\varepsilon} \leq j < 2n^{\varepsilon}$ **do**
    if $t[j-n^{\varepsilon} : j) = t[j+n^{\varepsilon} : j)$ **then**
        $C \leftarrow C \cup \{j\}$;
if $|C| \geq 2$ **then**
    // Calculate the minimum period length of t
    For any $j,k \in C$ such that $j < k$, $\ell(j,k) := 2(k-j)+1$;
    $\ell_0 \leftarrow gcd(\{l(j,k)-1 | j,k \in C, j < k\})+1$;
else
    $l_0 \leftarrow 2n^{\varepsilon}+1$;
if $|C| \geq 1$ **then**
    // Find the minimum period w of block b_{α}
    Let the period string of b_{α} be $w := t[n^{\varepsilon}, n^{\varepsilon}+l_0]$;
else
    Find the maximum length palindrome centered at j in b_{α} using any sequential palindrome algorithm, store j in $C[0]$;
    $w \leftarrow -1$;
Return w, C;

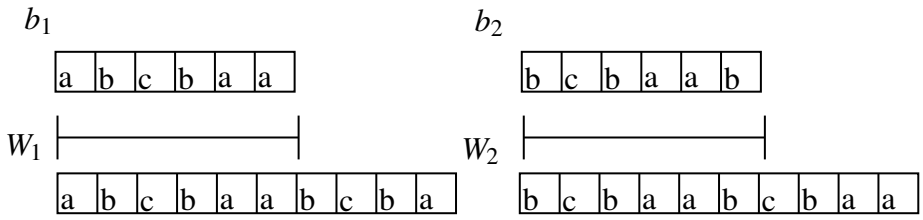


Figure E.6: An example of two consecutive string blocks b_1 and b_2. Block b_1 has period $w_1 =$ "*abcba*", which is used to build W_1 as well as W_2, a rotation of $W1$, used to check that b_2 follows the same period.

9 of Algorithm 30. Since each candidiate $j \in C$ is the center of a palindrome spanning the entire block b_α, the period string is repeated across the entire block and so we can take the first ℓ_0 characters as the period of the block. □

Lemma E.5.2. *On input $t = b_{\alpha-1}b_\alpha b_{\alpha+1}$ with $|b_{\alpha-1}| = |b_\alpha| = |b_{\alpha+1}| = n^{1-\varepsilon}$, Algorithm 30 uses $O(n^{1-\varepsilon})$ memory.*

Proof. By Lemma E.5.1, Algorithm 30 only considers candidates within block b_α of size $O(n^{1-\varepsilon})$, so storing palindrome centers requires at most space $O(n^{1-\varepsilon})$. Calculating the period and storing the period string of a single block of size $O(n^{1-\varepsilon})$ similarly requires space $O(n^{1-\varepsilon})$. □

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 $\lceil \frac{n^{1-\varepsilon}}{|w|} \rceil$ 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 E.6 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.

Algorithm 31: LPSPeriodRange($w_\alpha, H = [\mathsf{hash}(b_0), \mathsf{hash}(b_1), ..., \mathsf{hash}(b_{n^\varepsilon - 1})]$)

1 $W_\alpha \leftarrow \lceil \frac{n^{1-\varepsilon}}{|w_\alpha|} \rceil \cdot w_\alpha$;
2 $\ell, r \leftarrow -1$;
3 **for** $\beta = 0$ *to* $n^\varepsilon - 1$ **do**
4 **if** $\ell == -1$ *and* $\alpha - \beta \geq 0$ **then**
5 $W_\beta \leftarrow \text{rotate}(W_\alpha, -\beta \cdot (|W_\alpha| - n^{1-\varepsilon}))$;
6 **if** $H[\alpha - \beta] \neq W_\beta[0 : n^\varepsilon)$ **then**
7 $\ell = \alpha - \beta$;
8 **if** $r == -1$ *and* $\alpha + \beta < n^\varepsilon$ **then**
9 $W_\beta \leftarrow \text{rotate}(W_\alpha, \beta \cdot (|W_\alpha| - n^{1-\varepsilon}))$;
10 **if** $H[\alpha + \beta] \neq W_\beta[0 : n^\varepsilon)$ **then**
11 $r = \alpha + \beta$;
12 return ℓ, r;

Lemma E.5.3. *Given the period string w_α of periodic block b_α and an array of block string hashes, Algorithm 31 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 $\widetilde{O}(n^{1-\varepsilon})$ w.h.p..*

Proof. The correctness of Algorithm 31 to find the period range (ℓ, r) relies on the claim that $W_\beta[0 : n^\varepsilon)$ as constructed in steps 5 and 11 matches blocks $b_{\alpha-\beta}$ and $b_{\alpha+\beta}$, respectively, if these blocks are part of the period range with period w_α. We will prove this claim by induction on index j. For $j = 0$, this is trivial since we are given that block b_α is already periodic with period w_α.

Now let β be an index such that the claim holds true; we will prove that the claim holds for $\beta + 1$, first for the W_β constructed in step 11. Since the claim holds for β, we know that $W_\beta[0 : n^\varepsilon) = b_{\alpha+\beta}$. Furthermore, W_β is just a rotation of W_α, a periodic string with period w_α as constructed in step 1. Therefore, string W_β contains a fragment of a period that must be the start of the next block $b_{\alpha+\beta+1}$ if the following block is in the period range. Thus W_β rotated to the right by $|W_\beta| - n^\varepsilon$ characters will correctly match the remaining fragment to the next block as well as make sure the rest of the block is periodic since W_β is a rotation

of period block string W_α. Therefore,

$$\begin{aligned} W_{\beta+1} &= rotate(W_\beta, (|W_\beta| - n^{1-\varepsilon})) \\ &= rotate(W_\beta, (|W_\alpha| - n^{1-\varepsilon})) \\ &= rotate(W_\alpha, \beta \cdot (|W_\alpha| - n^{1-\varepsilon}) + |W_\alpha| - n^{1-\varepsilon})) \\ &= rotate(W_\alpha, (\beta+1) \cdot (|W_\alpha| - n^{1-\varepsilon})) \end{aligned}$$

which is exactly the string we set $W_{\beta+1}$ to be in step 11. The same arguments hold for W_β constructed in step 5 but with left rotations since we are checking the reverse direction from β to $\beta - 1$. Thus, the claim holds.

From this claim, we can see that by checking that each block b_β surrounding b_α matches W_β, we will find some index β_1 such that block $b_{\alpha-\beta_1}$ does not match W_{β_1} in step 6 and some index β_2 such that block $b_{\alpha+\beta_2}$ that does not match W_{β_2} in step 12, which are the nearest left and right blocks that fall outside of the period range. We then set our range indices accordingly in steps 7 and 12 so that $\ell = \alpha - \beta_1$ and $r = \alpha + \beta_2$.

Finally, Algorithm 31 requires $O(n^{1-\varepsilon})$ space to store W_α and W_β. Furthermore, Algorithm 31 must also store its input array H, which contains the hash of each block. Since we have that $\varepsilon < .5$ in our parallel setting, $|H| = n^\varepsilon < n^{1-\varepsilon}$, and so Algorithm 31 only requires $n^{1-\varepsilon}$ space. □

Lemma E.5.4. *Given period w of block b_α and period range ℓ, r on processor μ_α, Algorithm 32 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 $\widetilde{O}(n^{1-\varepsilon})$ and $O(1)$ rounds w.h.p.*

Proof. The main idea of Algorithm 32 is to iteratively check that the hash of the period w matches the hash of period-length chunks of the blocks b_ℓ and b_r. This idea itself seen

Algorithm 32: LPSExactPeriodRange(α, w, ℓ, r)

1 $a_r \leftarrow |w| - (r-\alpha)(\lceil \frac{n^{1-\varepsilon}}{|w_\alpha|} \rceil \cdot w_\alpha - n^{1-\varepsilon}) \mod (|w|)$;
2 $a_\ell \leftarrow |w| - (\ell-\alpha+1)(\lceil \frac{n^{1-\varepsilon}}{|w_i|} \rceil \cdot w_\alpha - n^{1-\varepsilon}) \mod (|w|)$;
3 $h_w \leftarrow \mathsf{hash}(w)$;
4 Send $a_r, h_w, |w|$ to μ_r;
5 Send $a_\ell, h_w, |w|$ to μ_ℓ;
6 **for** $j = 0$ *to* $\lfloor \frac{n^{1-\varepsilon}}{|w|} \rfloor$ *on processor* μ_r **do**
7 $t \leftarrow b_{r-1} b_r b_{r+1}$;
8 $k \leftarrow j \cdot |w| + (n^{1-\varepsilon} - a_r)$;
9 **if** $h_w \neq \mathsf{hash}(t[k : k+|w|))$ **then**
10 $o_r \leftarrow k$;
11 **for** $j = 0$ *to* $\lfloor \frac{n^{1-\varepsilon}}{|w|} \rfloor$ *on processor* μ_ℓ **do**
12 $t \leftarrow b_{\ell-1} b_\ell b_{\ell+1}$;
13 $k \leftarrow (2n^{1-\varepsilon} + a_\ell) - j \cdot |w|$;
14 **if** $h_w \neq \mathsf{hash}(t[k-|w| : k))$ **then**
15 $o_\ell \leftarrow k - w$;
16 Return o_ℓ, o_r;

in steps 6-12 and 13-19 is quite simple; the main difficulty is aligning the period with the blocks. Since the block lengths are $n^{1-\varepsilon}$, it is not guaranteed that the start of the period will align with the start of a block. Thus, in steps 1 and 2 we calculate the number of characters before the start of each block until the beginning of a full period.

When calculating value a_r in step 1, we follow a similar argument as in the proof of Lemma E.5.3. The misalignment of block b_α and period block W_α built in Algorithm 31 consisting of repeated period w is initially just $\lceil \frac{n^{1-\varepsilon}}{|w|} \rceil - n^{1-\varepsilon}$. Therefore, each additional block extends the misalignment by another factor of $\lceil \frac{n^{1-\varepsilon}}{|w|} \rceil - n^{1-\varepsilon}$. Since the period range is the range of the period w repeated consecutively, misalignment can only be at most the size of the period before another starts; thus, we mod our misalignment by $|w|$. To calculate the length of the period fragment at the beginning of block b_r, we therefore have $(r-\alpha)(\lceil \frac{n^{1-\varepsilon}}{|w|} \rceil - n^{1-\varepsilon})$. Lastly, since we want to make sure we check for full periods and

the beginning of block b_r is a period fragment, we subtract the length calculated up until now from $|w|$ to find out how many characters before b_r we must check to find a full period, and store this number in a_r. Step 8 on processor μ_r finds where a full period should start just before block b_r using a_r and steps 9 and 10 find the leftmost index that does not begin a full period $|w|$ and store it in o_r. By a symmetric argument, step 2 finds the number of characters after block b_ℓ until a full period and stores this in a_ℓ. Steps 15-17 then use a_ℓ to find the rightmost index in b_ℓ that does not begin a full period $|w|$ and stores this index in o_ℓ.

The number of rounds needed for the algorithm is only 2 since process μ_α must send and receive once each to processors μ_ℓ and μ_r to send integers $a_r, a_\ell, h_w, |w|$ and receive o_ℓ, o_r. Additionally, there is only $O(n^{1-\varepsilon})$ space needed per processor since the size of the string blocks t will be $O(n^{1-\varepsilon})$ and all auxiliary variables created to find o_ℓ and o_r trivially only require a constant amount of space. □

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 30. Each block then finds its surrounding period range using Algorithms 31 and 32. 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 E.1.8. *For $\varepsilon \in (0, .5]$, there is an $O(1)$-round MPC algorithm that solves* **Longest Palindrome Substring (LPS)** *with $\widetilde{O}(n)$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ memory per processor, w.h.p.*

Algorithm 33: LongestPalindromeSearch($s = \{s_0 s_\alpha ... s_{n-1}\}$)

1 Partition s into n^ε blocks of length $n^{1-\varepsilon}$ where b_α denotes the αth block;
2 Distribute blocks $b_{\alpha-1}, b_\alpha, b_{\alpha+1}$ to processor μ_α for $0 \leq \alpha < n^\varepsilon$;
3 Distribute $H = (\mathsf{hash}(b_1), \mathsf{hash}(b_2), ..., \mathsf{hash}(b_{n^\varepsilon - 1}))$ to all processors;
4 $D \leftarrow \mathsf{LCPinit}(s, \bar{s})$;
5 **for** *processors* $\mu_\alpha, 0 \leq \alpha < n^\varepsilon$ *in parallel* **do**
6 $\quad w_\alpha, C_\alpha \leftarrow \mathsf{LPSCandidate}(b^{\alpha-1} b^\alpha b^{\alpha+1})$;
7 $\quad \ell_\alpha, r_\alpha \leftarrow \mathsf{LPSPeriodRange}(w_\alpha, H)$;
8 $\quad o_{\ell_\alpha}, o_{r_\alpha} \leftarrow \mathsf{LPSExactPeriodRange}(\alpha, w_\alpha, \ell_\alpha, r_\alpha)$;
9 $\quad c_\alpha \leftarrow \frac{1}{2}\left((r_\alpha \cdot n^{1-\varepsilon} + o_r) - ((\ell_\alpha + 1) \cdot n^{1-\varepsilon} - o_{\ell_\alpha})\right) + ((\ell_\alpha + 1) \cdot n^{1-\varepsilon} - o_{\ell_\alpha})$;
10 $\quad$ **if** $s_{c_\alpha} \in b_\alpha$ **then**
11 $\quad\quad d_\alpha \leftarrow 2 \cdot D.\mathsf{query}(s_{c_\alpha}, n - s_{c_\alpha}) + 1$;
12 $d_{max} = \max_{\alpha, 1 \leq \alpha \leq n^\varepsilon}\{d_\alpha\}$;
13 Return (c_α, d_α) such that $d_\alpha = d_{max}$;

Proof. For the correctness of Algorithm 33 we parse the algorithm one step at a time. First, each process μ_α receives its blocks $b_{\alpha-1}, b_\alpha, b_{\alpha+1}$ of length $n^{1-\varepsilon}$ and an array of hashes of each block. Second, the LCPQ oracle is setup on s and its reverse string $\bar{s}$. In parallel, by Lemma E.5.1, step 6 finds the largest palindrome candidate centers in block b_α and the period w_α of this block. In step 7, by Lemma E.5.3, the range of block indices (ℓ_α, r_α) is found such that each block in this range matches a periodic string with period w_i. In step 8, by Lemma E.5.4, the exact indices $o_{\ell_\alpha}, o_{r_\alpha}$ that denote the exact characters in b_{ℓ_α} and b_{r_α} where the period ends, specifically no more full periods continue to the left of o_{ℓ_α} nor to the right of o_{r_α}. At this point, we know that any maximum length palindrome must be at the center of this period range and spans at least the entire period range length. If there was another maximum length palindrome within this period range that was not the center, then it could not be longer than the period range since we know that one side would have a longer periodic section than the other. Therefore, we can consider only the very center of the period range for a maximum length palindrome. This center is therefore exactly halfway between the $(n^{1-\varepsilon} - o_{\ell_\alpha})$-th character in block b_{ℓ_α} and the o_{r_α}-th character

in block b_{r_α}. Since each block has length $n^{1-\varepsilon}$, the first character of block b_{ℓ_α} has index $\ell_\alpha \cdot n^{1-\varepsilon}$ in s and the first character of block b_{r_α} is at index $r_\alpha \cdot n^{1-\varepsilon}$. Thus, the length of the period range will be

$$(r_\alpha \cdot n^{1-\varepsilon} + o_r) - ((\ell_\alpha + 1) \cdot n^{1-\varepsilon} - o_{\ell_\alpha})$$

and in step 9, we add half this length to the leftmost index of the period range, $((\ell_\alpha + 1) \cdot n^{1-\varepsilon} - o_{\ell_\alpha})$, to find the center of the period range c_α. In step 11, we use the LCPQ oracle to find the length of the palindrome by finding the length of the longest common prefix of the palindrome center in s with the palindrome center in $\bar{s}$, which is equivalent to finding the length of characters to the right of c_α in s that match the characters to the left of c_α, which is equivalent to the characters to the right of $n - c_\alpha$.

The number of rounds needed in steps 1-4 to distribute the blocks, hashes, and to set up the LCPQ oracle is just $O(1)$ since each processor only receives $O(n^{1-\varepsilon})$ information, which is the size of each processor's memory. Then, in steps 6 and 7, LPS_Candidate are performed locally on a single processor. By Lemma E.5.4, Step 8 only requires $O(1)$ rounds and can be performed in parallel by each processor since each processor only sends a constant amount of information to two other processors. Step 11 requires $O(1)$ rounds as discussed in Section E.2 and Section E.4. Finally, finding the max palindrome across the entire string in step 14 can be done in 1 round since there is at most n^ε candidate indices, one per block, and there is $O(n^{1-\varepsilon})$ memory per processor which is larger than n^ε since $\varepsilon < .5$.

The memory per processor is trivially $O(n^{1-\varepsilon})$ since each processor only gets 3 blocks of size $O(n^{1-\varepsilon})$ and by Lemma's E.5.2, E.5.3, and E.5.4. Since we use $O(n^\varepsilon)$ processors to store each block of the string, build the LCPQ oracle (see Corollary E.1.2), and find the maximum length palindrome across the entire string, we therefore use $O(n^\varepsilon \cdot n^{1-\varepsilon}) = O(n)$

memory total.

□

E.6 Longest Common Substring in MPC

In this section, we show a constant-round MPC algorithm for finding the *Longest Common Substring* (LCS) of two strings s and s'. Without loss of generality, we assume both strings have length n. A substring $s[i:i+l)$ of string s is a consecutive interval of characters $s_i, s_{i+1}, \ldots, s_{i+l-1}$. Recall the definition of LCP queries from Section E.2. $\mathsf{LCP}_{s,s'}(i,j)$ is the longest common prefix of two suffixes $s[i:n)$ and $s'[j:n)$. Thus,

$$\mathsf{LCS}_{s,s'} = \max_{i,j\in[n]} \mathsf{LCP}_{s,s'}(i,j) \tag{E.3}$$

Assume both strings are partitioned into n^{ε} consecutive blocks of size $O(n^{1-\varepsilon})$, denoted by $b_1, b_2, \ldots, b_{n^{\varepsilon}}$ for s and $b'_1, b'_2, \ldots, b'_{n^{\varepsilon}}$ for s', so that each block fits into the memory of a single machine. We denote the machine which stored b_α and b'_β by μ_α and $\overline{\mu}_\beta$ respectively. Our high-level approach is to find the longest common substring whose starting points fall in a fixed pair of blocks, say b_α and b'_β. Let us denote this value by $\mathsf{LCS}_{s,s'}(b_\alpha, b'_\beta)$, i.e, $\mathsf{LCS}_{s,s'}(b_\alpha, b'_\beta) = \max_{i\in b_\alpha, j\in b'_\beta} \mathsf{LCP}_{s,s'}(i,j)$. Then, the final answer can be computed as the maximum of this partial solutions for each pair of blocks, i.e.,

$$\mathsf{LCS}_{s,s'} = \max_{\alpha,\beta\in[\mathsf{M}]} \mathsf{LCS}_{s,s'}(b_\alpha, b'_\beta) \tag{E.4}$$

Thus, our focus in this section is to efficiently compute $\mathsf{LCS}_{s,s'}(b_\alpha, b'_\beta)$. We may omit subscripts in LCS and LCP functions whenever s and s' are implied from the context. We

also need that this process requires only $O(1)$ machines, so that we can compute $O(n^{2\varepsilon})$ different values in parallel and still have $O(n^{1+\varepsilon})$ total memory.

Definition E.5 (Candidate Pair)**.** For a pair of blocks b_α and b'_β, we call a pair of indices $p = (i, j)$, where $i \in b_\alpha$ and $j \in b'_\beta$, a *"candidate pair"* if $\mathsf{LCP}(i, j) \geq 3\mathsf{S}$. The *"offset"* of a candidate pair $p = (i, j)$, denoted by $\delta(p)$, is defined as $i - j$.

We can find a set of candidate pairs by transferring $O(1)$ blocks into a single machine and performing an exhaustive search. Let $P_{\alpha,\beta} = \{p_1, p_2, \ldots, p_k\}$ be the set of candidate pairs belonging to the pair of blocks b_α and b'_β. If there are multiple candidate pairs with the same offset, we only keep the one with the smallest i. We show that when there are multiple candidate pairs, $k > 1$, the subsequent blocks $b_{\alpha+1}$ and $b'_{\beta+1}$ are periodic. We will later use this periodicity property in our algorithm. Roughly speaking, assume we have two candidate pairs p_1 and p_2. This means that for every $i \in b_{\alpha+1}$,

$$s_i = s'_{i-\delta(p_1)} = s_{i-\delta(p_1)+\delta(p_2)}$$

This intuitively means that $b_{\alpha+1}$ is periodic with period length $\delta(p_1) - \delta(p_2)$. However, since $|\delta(p_1) - \delta(p_2)|$ might be as large as $2n^{1-\varepsilon}$, we prove the periodicity properties for the concatenation of the next two blocks $b_{\alpha+1}b_{\alpha+2}$. In the following lemma, we formally show that the period lengths of both $b_{\alpha+1}b_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$ are equal to $\gcd\{\delta(p_1) - \delta(p_2) \mid p_1, p_2 \in P_{\alpha,\beta}\}$.

Lemma E.6.1. *The period length of both $b_{\alpha+1}b'_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$ is equal to* $\gcd\{\delta(p_1) - \delta(p_2) \mid p_1, p_2 \in P_{\alpha,\beta}\}$.

Proof. Recall that $P_{\alpha,\beta}$ is the set of candidate pairs $(i, j) \in b_\alpha \times b'_\beta$ so that $\mathsf{LCP}(i, j) \geq$

$3n^{1-\varepsilon}$. This implies that for any $p^\circ = (i^\circ, j^\circ) \in b_{\alpha+1}b_{\alpha+2} \times b'_{\beta+1}b'_{\beta+2}$, we have

$$s_{i^\circ} = s'_{j^\circ} \quad \text{if } \exists\, p = (i,j) \in P_{\alpha,\beta} \quad \text{s.t. } \delta(p) = \delta(p^\circ) \tag{E.5}$$

We can further expand the domain of variable p° in equation E.5 by requiring either $i^\circ \in b_{\alpha+1}b_{\alpha+2}$ or $j^\circ \in b'_{\beta+1}b'_{\beta+2}$. As long as there is a candidate pair p with the same offset as p°, then we have either

$$\begin{cases} i^\circ \in b_{\alpha+1}b_{\alpha+2}\text{:} & i \le (\alpha+1)n^{1-\varepsilon} \le i^\circ \Rightarrow j \le j^\circ \\ j^\circ \in b'_{\beta+1}b'_{\beta+2}\text{:} & j \le (\beta+1)n^{1-\varepsilon} \le j^\circ \Rightarrow i \le i^\circ \end{cases}$$

Thus, if at least one of the indices in a pair p° is within the two subsequent blocks, i.e., $i^\circ \in b_{\alpha+1}b_{\alpha+2}$, and there is a candidate pair in $P_{\alpha,\beta}$ with the same offset, then $s_{i^\circ} = s'_{j^\circ}$. We exploit this fact to demonstrate that the length of the period string $w_{\alpha,\beta}$ of subsequent blocks, $b_{\alpha+1}b_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$, divides the offset difference of every pair of candidate pairs. More precisely, we show that the subsequent blocks are periodic with length $\delta(p_1) - \delta(p_2)$, which means that this length is a multiple of the minimal period length, i.e., $|w_{\alpha,\beta}|$.

$$(\delta(p_1) - \delta(p_2)) \bmod |w_{\alpha,\beta}| = 0 \quad \forall\, p_1, p_2 \in P_{\alpha,\beta} \tag{E.6}$$

To show that b_α and b'_β are periodic with length $\delta(p_1) - \delta(p_2)$, we use the same idea we intuitively mentioned before. Note that the period range $\delta(p_1) - \delta(p_2)$ is bounded by $2n^{1-\varepsilon}$. Consider two indices $i, j \in b_{\alpha+1}b_{\alpha+2}$, so that $i - j = \delta(p_1) - \delta(p_2)$. Since i is within the two subsequent blocks, then $s_i = s'_{i-\delta(p_1)}$. In addition, since j is also within the

subsequent block, then $s_j = s'_{j-\delta(p_2)}$. However,

$$i - \delta(p_1) = (i-j) - (\delta(p_1) - j) = \delta(p_1) - \delta(p_2) - \delta(p_1) + j = j - \delta(p_2)$$

Hence, $s_i = s_j$ since at least one index in each pair is within the periodic region. Now that we have shown that the period length of b_α divides the offset difference of every $p_1, p_2 \in P_{\alpha,\beta}$, it is straight-forward to observe that the period length of b_α is equal to the GCD of all the offset differences in every pair of candidate pairs, i.e., the right hand side of Equation E.7. We can prove the same thing for b'_β similarly.

$$|w_{\alpha,\beta}| = \gcd\{\delta(p_1) - \delta(p_2) \mid p_1, p_2 \in P_{\alpha,\beta}\} \tag{E.7}$$

□

As mentioned before, we aim to compute $\mathsf{LCS}(b_\alpha, b'_\beta)$ for every pair of blocks b_α and b'_β in parallel. We reduced this problem to a collection of $O(n^2)$ LCP queries according to Equation (E.3). Even by excluding every query which is not a candidate pair, we might have to make $\Omega(n^2)$ LCP queries, while an LCPQ oracle from Section E.2 is capable of handling at most $O(n^{1+\varepsilon})$ LCP queries in $O(1)$ rounds without blowing up the memory. To overcome this issue, we use the periodicity properties of the subsequent blocks $b_{\alpha+1}b_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$ to limit the number of LCP queries we need to make for each pair of blocks to $O(1)$. At the beginning, we build three instances of the LCPQ oracle $\mathscr{D}_{s,s}$, $\mathscr{D}_{s,s'}$, and $\mathscr{D}_{s',s'}$. The algorithm consists of three stages:

1 Find the set of candidate pairs, $P_{\alpha,\beta}$, by an exhaustive search on the local data of $\mu_{\alpha,\beta}$. If $|P_{\alpha,\beta}| = O(1)$, we make a separate LCP query to $\mathscr{D}_{s,s'}$ for each candidate pair and we are done. Otherwise, the subsequent blocks $b_{\alpha+1}b_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$ are periodic. Let $w_{\alpha,\beta}$ be the period string where $|w_{\alpha,\beta}| = \gcd\{\delta(p_1) - \delta(p_2) \mid p_1, p_2 \in$

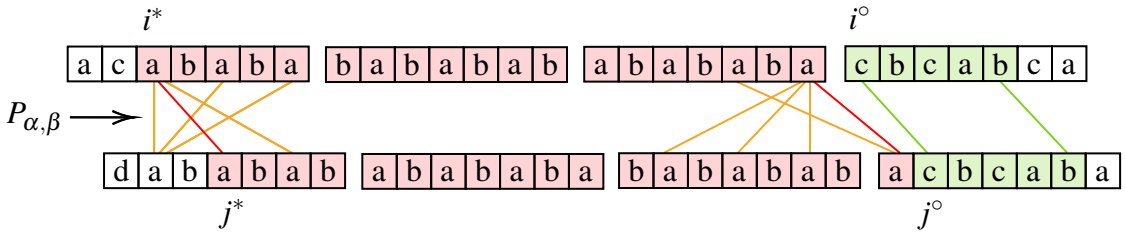


Figure E.7: Candidate pairs, first inconsistent character, and the extension beyond periodic part

$P_{\alpha,\beta}\}$ according to Lemma E.6.1.

2 Next, we find the boundaries of the periodic pattern. The periodic pattern might repeat after the subsequent blocks $b_{\alpha+1}b_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$ and span several blocks. Let $i^\circ \in \alpha^\circ$ and $j^\circ \in \beta^\circ$ be the first characters in s and s' which are inconsistent with the periodic pattern. We call i° and j° the *first inconsistent characters*. To find the exact location of i° and j°, we utilize the LCPQ oracle we build for the strings and themselves, $\mathscr{D}_{s,s}$ and $\mathscr{D}_{s',s'}$. The first inconsistent characters in s and s' are respectively located at:

$$i^\circ = \alpha n^{1-\varepsilon} + |w_{\alpha,\beta}| + \mathscr{D}_{s,s}.\mathsf{query}(\alpha n^{1-\varepsilon}, \alpha n^{1-\varepsilon} + |w_{\alpha,\beta}|)$$
$$j^\circ = \beta n^{1-\varepsilon} + |w_{\alpha,\beta}| + \mathscr{D}_{s',s'}.\mathsf{query}(\beta n^{1-\varepsilon}, \beta n^{1-\varepsilon} + |w_{\alpha,\beta}|)$$

3 If there exists a candidate pair $p^* = (i^*, j^*) \in P_{\alpha,\beta}$ so that $\delta(p^*) = i^\circ - j^\circ$, then $\mathsf{LCP}(i^*, j^*)$ may extend beyond the boundaries of the periodic pattern. For this specific candidate pair, we make an LCP query to $\mathscr{D}_{s,s'}$. For all other candidate pairs in $P_{\alpha,\beta}$, we claim that knowing the location of i° and j° is sufficient for computing $\mathsf{LCP}(i, j)$.

Algorithm 34 demonstrates the algorithm in more details. In what follows, we prove that the described algorithm correctly finds the *Longest Common Substring* of two strings s and s', $\mathsf{LCS}_{s,s'}$, and the MPC memory constraints are not violated during $O(1)$ rounds.

Algorithm 34: $\mathsf{LCS}(s = \{s_0 s_1 ... s_{n-1}\}, s' = \{s'_0 s'_1 ... s'_{n-1}\},)$

1 $\mathscr{D}_{s,s}, \mathscr{D}_{s,s'}, \mathscr{D}_{s',s'} \leftarrow \mathsf{LCPInit}(s,s), \mathsf{LCPInit}(s,s'), \mathsf{LCPInit}(s',s')$;
2 **for** *all processors* $\mu_{\alpha,\beta}$, $0 \le \alpha, \beta < n^{\varepsilon}$, *in parallel* **do**
3 $\quad P_{\alpha,\beta} \leftarrow \left\{(i,j) \in b_\alpha \times b'_\beta \mid \begin{cases} s[i : i+3\mathsf{S}) = s'[j : j+3\mathsf{S}) \\ s[i-1 : i-1+3\mathsf{S}) \neq s'[j-1 : j-1+3\mathsf{S}) \end{cases}\right\}$;
4 $\quad w_{\alpha,\beta} \leftarrow b_{\alpha+1} b_{\alpha+2}[1 : \gcd\{\delta(p_1) - \delta(p_2) \mid p_1, p_2 \in P_{\alpha,\beta}\}]$;
$\quad$ // Find the boundaries of the periodic region
5 $\quad i^\circ \leftarrow (\alpha+1)n^{1-\varepsilon} + \mathscr{D}_{s,s}.\mathsf{query}((\alpha+1)n^{1-\varepsilon}, (\alpha+1)n^{1-\varepsilon} + |w_{\alpha,\beta}|)$;
6 $\quad j^\circ \leftarrow (\beta+1)n^{1-\varepsilon} + \mathscr{D}_{s',s'}.\mathsf{query}((\beta+1)n^{1-\varepsilon}, (\beta+1)n^{1-\varepsilon} + |w_{\alpha,\beta}|)$;
7 $\quad p^\circ \leftarrow (i^\circ, j^\circ)$;
8 $\quad S_{\alpha,\beta} \leftarrow \max_{(i,j) \in P_{\alpha,\beta}} \min\{i^\circ - i, j^\circ - j\}$;
$\quad$ // Find the extension beyond the periodic region
9 $\quad$ **if** *there is any* $p^* = (i^*, j^*) \in P_{\alpha,\beta}$ *such that* $\delta(p^*) = \delta(p^\circ)$ **then**
10 $\quad\quad S_{\alpha,\beta} \leftarrow \max\{S_{\alpha,\beta}, \mathscr{D}_{s,s'}.\mathsf{query}(i^*, j^*)\}$;
11 $\quad$ **end**
12 **end**

Theorem E.1.9. *For* $\varepsilon \in (0, .5]$*, there is an* $O(1)$*-round MPC algorithm that solves* **Longest Common Substring (**LCS**)** *with* $\widetilde{O}(n^{1+\varepsilon})$ *total memory and* $\widetilde{O}(n^{1-\varepsilon})$ *memory per processor, w.h.p.*

Proof. The algorithm starts by building the set of candidate pairs using an exhaustive search on each $\mu_{\alpha,\beta}$ in parallel. There are a collection of $O(n^{2\varepsilon})$ machines $\{\mu_{\alpha,\beta} \mid 0 \le \alpha, \beta < n^{\varepsilon}\}$ each with memory $O(n^{1-\varepsilon})$, which are initialized with the appropriate data in the beginning. Recall the construction of $\mu_{\alpha,\beta}$ machine from Section E.2. We allow each machine to have access to 3 subsequent blocks in each string additionally, i.e., $\{b_{\alpha+1}, b_{\alpha+2}, b_{\alpha+3}, b'_{\beta+1}, b'_{\beta+2}, b'_{\beta+3}\}$, so that we can locally compute the set of candidate pairs $P_{\alpha,\beta}$ in each machine $\mu_{\alpha,\beta}$ (Line 3 of Algorithm 34 which follows the definition of *candidate pairs* from Definition E.5). Note that if there are several candidate pairs with the same offset, we keep the one with smallest i. Therefore, the total size of $P_{\alpha,\beta}$ is bounded by $O(n^{1-\varepsilon})$.

The next step is to reduce the number of potential candidates in each machine to $O(1)$, which may count up to $O(n^{1-\varepsilon})$ and in that case we need to make $O(n^{1-\varepsilon})$ queries per machine if we naively query each candidate pair.[10] If the number of candidate pairs is already $O(1)$, we can query each candidate pair separately using an LCPQ oracle. As mentioned before we initiate 3 instances of LCPQ oracle as the following which fit the memory limits.

$$\begin{cases} \mathscr{D}_{s,s} \leftarrow \mathsf{LCPInit}(s,s) \\ \mathscr{D}_{s,s'} \leftarrow \mathsf{LCPInit}(s,s') \\ \mathscr{D}_{s',s'} \leftarrow \mathsf{LCPInit}(s',s') \end{cases}$$

We use $\mathscr{D}_{s,s}$ and $\mathscr{D}_{s',s'}$ for finding the borders of the periodic pattern in s and s' respectively, and $\mathscr{D}_{s,s'}$ allows us to query the LCP of an individual candidate pair. According to Lemma E.6.1, the subsequent blocks $b_{\alpha+1}b_{\alpha+2}$ and $b'_{\beta+1}b'_{\beta+2}$ are periodic with period length

$$|w_{\alpha,\beta}| = \gcd\{\delta(p_1) - \delta(p_2) \mid p_1, p_2 \in P_{\alpha,\beta}\}.$$

Once we know the period length $|w_{\alpha,\beta}|$, we query $\mathscr{D}_{s,s}$ to find where the periodic pattern ends in s, aka *the first inconsistent character* or i°. We can pick any two pair of indices with distance $|w_{\alpha,\beta}|$ which both belong to $b_{\alpha+1}b_{\alpha+2}$, for example, $(\alpha+1)n^{1-\varepsilon}$ and $(\alpha+1)n^{1-\varepsilon}+|w_{\alpha,\beta}|$. Similarly, we find j° by querying $\mathscr{D}_{s',s'}$, and let $p^\circ = (i^\circ, j^\circ)$. The LCP of every candidate pair except one needs to end at either i° or j°.

Note that a character within the periodic range never matches with a first inconsistent character. Therefore, no candidate pair with an offset not equal to $\delta(p^\circ)$ can extend beyond p°. On the other hand, we know that every candidate pair $p = (i, j)$ has an LCP which is at

[10] The LCPQ oracle actually supports $O(n^{1-\varepsilon})$ queries per machine, however we minimize the number of required queries anyways for more flexibility.

least $\min\{i^\circ - i, j^\circ - j\}$ since the periodic pattern continues up to p°. Thus, we don't even need to query $\mathscr{D}_{s,s'}$ for almost all of the candidate pairs. However, if there is a candidate pair $p^* = (i^*, j^*)$ so that $\delta(p^*) = \delta(p^\circ)$, it may go beyond p° as long as the two strings have the same inconsistent characters, i.e, $s_{i^\circ+1} = s'_{j^\circ+1}, s_{i^\circ+2} = s'_{j^\circ+2}, \ldots$. Only for this candidate pair, which is unique since we keep only one candidate if there were several with the same offset initially, we make a query to $\mathscr{D}_{s,s'}$. Since we already have computed the LCP of every pair $i, j \in b_\alpha \times b'_\beta$, we return the maximum LCP as the partial solution of this pair of blocks $\mathsf{LCS}_{s,s'}(b_\alpha, b'_\beta)$.

The total number of queries we make to LCPQ oracles is bounded by $O(n^{2\varepsilon})$, since there are only $O(1)$ LCP queries for each pair of blocks. Therefore, the total memory we need for LCP data structures is bounded by $O(n^{1+\varepsilon})$. □

E.7 Suffix Tree in AMPC

We will now show that in the AMPC model, we are able to improve the total memory of the suffix tree construction from MPC to $\tilde{O}(n)$ memory. 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 $\mathsf{hash}(s[i:n))$. 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.

Algorithm 35: LCP2Queries(i, j, o, k)

```
1 if hash(s[i : i+o)) = hash(s[j : j+o)) then
2     if k = 1 or hash(s[i : i+o+1)) ≠ hash(s[j : j+o+1)) then
3         return o;
4     o ← o + ⌈(n−j)/2^k⌉;
5     LCP2Queries(i, j, o, k+1);
6 else
7     o ← o − ⌈(n−j)/2^k⌉;
8     LCP2Queries(i, j, o, k+1);
```

Lemma E.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 $o = n - j$ as input, Algorithm 35 computes the LCP of two indices in s in $O(1)$ rounds and $O(\log^2 n)$ total space with high probability.*

Proof. Algorithm 35 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 35 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 $\mathsf{LCP}(i, j) = 1$, 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 $\mathsf{LCP}(i, j) > 0$, which is easily verifiable without the need for hashing.

Assume that for the kth recursive call to Algorithm 35, the algorithm is correct. Consider the $(k+1)$th call. If $\mathsf{LCP}(i,j) \geq o$, then we find in step 1 that $\mathsf{hash}(s[i:i+o])$=$\mathsf{hash}(s[j:j+o])$ 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 $\mathsf{LCP}(i,j) < o$, 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 $\mathsf{LCP}(i,j) = o$, then $\mathsf{hash}(s[i:i+o+1)) \neq \mathsf{hash}(s[j:j+o+1))$ and we still stop at step 3 correctly and output $\mathsf{LCP}(i,j) = 0$.

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, $n^\varepsilon > \log(n)$, then Algorithm 35 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 E.7.1, we have reduced the memory requirements of the LCPQ oracle construction from $O(n^{1+\varepsilon})$ in MPC to $\tilde{O}(n)$ in AMPC with $O(1)$ rounds. We are also able to update the $\mathsf{PivotFiltering}$ parameters from Section E.3 in AMPC to select $n^{1-\varepsilon}$ 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 use Algorithm 36 to 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, by Corollary E.3.2, we can construct the suffix tree in the same space and rounds. Thus, we have our main AMPC result:

Theorem E.1.10. (Suffix Tree in AMPC). *For $\varepsilon \in (0,1]$, there is an $O(1)$-round AMPC*

Algorithm 36: SuffixArray(s)

```
Let ℓ ← ⌈2/ε⌉ − 1;
Let L_0, L_1, L_2, ..., L_ℓ be a collection of lexicographically ordered bags of sets;
Add {1, 2, ..., n} to L_0;
D_{s,s} ← LCPInit(s,s);
π ← {(i, i) | 0 ≤ i < n};
for 1 ≤ z ≤ ℓ do
    for 1 ≤ t ≤ |L_z| do
        Let U_t = {u_{t,1}, u_{t,2}, ..., u_{t,k}} be the t-th set in L_z ordered by the
          lexicographical order;
        if z = ℓ then
            // We have k = O(n^{ε/2}).
            SmallChunkSort(t, U_t, π)
        else
            PivotFiltering(t, U_t, O(n^{(1−ε)/2}))
        end
    end
end
Relabel π so that the relative order of π_i's is preserved, and the range of π is [n];
```

algorithm for computing the **suffix tree** *of a given string s with $\tilde{O}(n)$ total memory and $\widetilde{O}(n^{1-\varepsilon})$ memory per processor with high probability.*

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Shan Leung Maverick Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *SODA*, pages 531–540, 2004.

[2] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 147–160. ACM, 2008.

[3] Reaz Ahmed and Raouf Boutaba. Distributed pattern matching: a key to flexible and efficient p2p search. *IEEE Journal on selected areas in communications*, 25(1), 2007.

[4] Kook Jin Ahn and Sudipto Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. In *SPAA*, pages 202–211, 2015.

[5] Noga Alon, László Babai, and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *J. Algorithms*, 7(4):567–583, December 1986.

[6] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *STOC*, pages 574–583, 2014.

[7] Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *FOCS*, pages 674–685, 2018.

[8] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Log diameter rounds algorithms for 2-vertex and 2-edge connectivity. In *Automata, Languages, and Programming*, pages 14:1–14:16, 2019.

[9] Alberto Apostolico, Dany Breslauer, and Zvi Galil. 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*, pages 497–506. Springer, 1994.

[10] Eshrat Arjomandi. An efficient algorithm for colouring the edges of a graph with Δ+ 1 colours. *INFOR: Information Systems and Operational Research*, 20(2):82–101, 1982.

[11] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab S. Mirrokni, and Cliff Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In *SODA*, pages 1616–1635, 2019.

[12] Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear algorithms for $\delta + 1$ vertex coloring. In *SODA*, pages 767–786, 2019.

[13] Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively parallel algorithms for finding well-connected components in sparse graphs. In *ACM SIGACT*, pages 461–470, 2019.

[14] Mikhail J. Atallah, S. Rao Kosaraju, Lawrence L. Larmore, Gary L. Miller, and Shang-Hua Teng. Constructing trees in parallel. In *SPAA*, 1989.

[15] Arturs Backurs, Piotr Indyk, and Ludwig Schmidt. Better approximations for tree sparsity in nearly-linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2215–2229. SIAM, 2017.

[16] Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The Locality of Distributed Symmetry Breaking. *J. ACM*, 63(3):20:1–20:45, June 2016.

[17] Joshua Baron, Karim El Defrawy, Kirill Minkovich, Rafail Ostrovsky, and Eric Tressler. 5pm: Secure pattern matching. In *International Conference on Security and Cryptography for Networks*, pages 222–240. Springer, 2012.

[18] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab S. Mirrokni. Affinity clustering: Hierarchical clustering at scale. In *NeurIPS*, pages 6864–6874, 2017.

[19] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Vahab S. Mirrokni. Brief announcement: Mapreduce algorithms for massive trees. In *Automata, Languages, and Programming*, pages 162:1–162:4, 2018.

[20] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Vahab S. Mirrokni. Massively parallel dynamic programming on trees. *CoRR*, 2018.

[21] MohammadHossein Bateni, MohammadTaghi Hajiaghayi, Saeed Seddighin, and Cliff Stein. Fast algorithms for knapsack via convolution and prediction. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1269–1282. ACM, 2018.

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

[23] Soheil Behnezhad. Time-optimal sublinear algorithms for matching and vertex cover. *arXiv preprint arXiv:2106.02942*, 2021.

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

[25] Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Brief Announcement: Semi-MapReduce Meets Congested Clique. *CoRR*, abs/1802.10297, 2018.

[26] Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Richard M. Karp. Massively parallel symmetry breaking on sparse graphs: MIS and maximal matching. *CoRR*, abs/1807.06701, 2018.

[27] Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Marina Knittel, and Hamed Saleh. Streaming and massively parallel algorithms for edge coloring. In *ESA*, pages 15:1–15:14, 2019.

[28] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab S. Mirrokni. Near-optimal massively parallel graph connectivity. In *FOCS*, pages 1615–1636, 2019.

[29] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, Vahab S. Mirrokni, and Warren Schudy. Massively parallel computation via remote memory access. In Christian Scheideler and Petra Berenbrink, editors, *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, pages 59–68. ACM, 2019.

[30] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, Vahab S. Mirrokni, and Warren Schudy. Parallel graph algorithms in constant adaptive rounds: Theory meets practice. *Proc. VLDB Endow.*, 13(13):3588–3602, 2020.

[31] Soheil Behnezhad, MohammadTaghi Hajiaghayi, and David G. Harris. Exponentially faster massively parallel maximal matching. In *FOCS*, pages 1637–1649, 2019.

[32] Charles H Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and weaknesses of quantum computing. *SIAM journal on Computing*, 26(5):1510–1523, 1997.

[33] Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM J. Comput.*, pages 317–378, 2016.

[34] Hans L. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. In Zoltán Fülöp and Ferenc Gécseg, editors, *Automata, Languages and Programming, 22nd International Colloquium, ICALP95, Szeged, Hungary, July 10-14, 1995, Proceedings*, volume 944 of *Lecture Notes in Computer Science*, pages 268–279. Springer, 1995.

[35] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi Hajiaghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and mapreduce. In *SODA 2018*, 2018.

[36] Sebastian Brandt, Manuela Fischer, and Jara Uitto. Matching and MIS for uniformly sparse graphs in the low-memory MPC model. *CoRR*, abs/1807.05374, 2018.

[37] David Bremner, Timothy M Chan, Erik D Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, Mihai Patraşcu, and Perouz Taslakian. Necklaces, convolutions, and x+y. *Algorithmica*, 69(2):294–314, 2014.

[38] E Oran Brigham and E Oran Brigham. *The fast Fourier transform and its applications*, volume 448. prentice Hall Englewood Cliffs, NJ, 1988.

[39] Benjamin C Brodie, David E Taylor, and Ron K Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *ACM SIGARCH computer architecture news*, 34(2):191–202, 2006.

[40] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In Benjamin G. Zorn and Alexander Aiken, editors, *ACM SIGPLAN*, pages 363–375, 2010.

[41] Timothy M Chan and Moshe Lewenstein. Clustered integer 3sum via additive combinatorics. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 31–40. ACM, 2015.

[42] Moses Charikar, Weiyun Ma, and Li-Yang Tan. Unconditional lower bounds for adaptive massively parallel computation. In *SPAA*, pages 141–151, 2020.

[43] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 455–466. ACM, 2005.

[44] Christopher R Clark and David E Schimmel. Scalable pattern matching for high speed networks. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 249–257. IEEE, 2004.

[45] Peter Clifford and Raphaël Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53–54, 2007.

[46] Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 592–601. ACM, 2002.

[47] Richard Cole and Uzi Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, pages 329–346, 1988.

[48] Stephen A Cook and Robert A Reckhow. Time-bounded random access machines. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 73–80, 1972.

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

[50] Gordon Charles Danielson and Cornelius Lanczos. Some improvements in practical fourier analysis and their application to x-ray scattering from liquids. *Journal of the Franklin Institute*, 233(4):365–380, 1942.

[51] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[52] Eliezer Dekel, Simeon Ntafos, and Shie-Tung Peng. Parallel tree techniques and code optimization. In Filia Makedon, Kurt Mehlhorn, T. Papatheodorou, and P. Spirakis, editors, *VLTSI Algorithms and Architectures*, pages 205–216, 1986.

[53] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Trading off space for passes in graph streaming problems. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 714–723, 2006.

[54] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. Rdma reads: To use or not to use? *IEEE Data Eng. Bull.*, 40(1):3–14, 2017.

[55] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In {*USENIX*} *({NSDI})*, pages 401–414, 2014.

[56] Jean-François Dufayard, Laurent Duret, Simon Penel, Manolo Gouy, François Rechenmann, and Guy Perrière. Tree pattern matching in phylogenetic trees: automatic search for orthologs or paralogs in homologous gene sequence databases. *Bioinformatics*, 21(11):2596–2603, 2005.

[57] Martin Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In Friedhelm Meyer auf der Heide and Burkhard Monien, editors, *Automata, Languages and Programming, 23rd International Colloquium, ICALP96, Paderborn, Germany, 8-12 July 1996, Proceedings*, volume 1099 of *Lecture Notes in Computer Science*, pages 550–561. Springer, 1996.

[58] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, pages 531–543, 2004.

[59] Michael J Fischer and Michael S Paterson. String-matching and other products. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1974.

[60] Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 625–634, 2016.

[61] H. N. Gabow, T. Nishizeki, O. Kariv, D. Leven, , and O. Terada. Algorithms for edgecoloring graphs. Technical report, Tohoku University, 1985.

[62] François Le Gall and Saeed Seddighin. Quantum meets fine-grained complexity: Sublinear time quantum algorithms for string problems. *CoRR*, abs/2010.12122, 2020.

[63] H. Gazit, Gary L. Miller, and ShangHua Teng. Optimal tree contraction in an EREW model. In *Concurrent Computations: Algorithms, Architecture and Technology*, pages 139–156, 1988.

[64] Hillel Gazit and Gary L. Miller. A parallel algorithm for finding a separator in planar graphs. In *FOCS*, pages 238–248, 1987.

[65] Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrovic, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for mis, matching, and vertex cover. In *ACM SIGACT*, pages 129–138, 2018.

[66] Mohsen Ghaffari, Christoph Grunau, and Ce Jin. Improved mpc algorithms for mis, matching, and coloring on trees and beyond. *arXiv preprint arXiv:2002.09610*, 2020.

[67] Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. In *FOCS*, pages 1650–1663, 2019.

[68] Mohsen Ghaffari and Krzysztof Nowicki. Massively parallel algorithms for minimum cut. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 119–128, 2020.

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

[70] Alan Gibbons and Wojciech Rytter. Optimal parallel algorithms for dynamic expression evaluation and context-free recognition. *Inf. Comput.*, page 32–45, 1989.

[71] Christian Glazik, Jan Schiemann, and Anand Srivastav. Finding euler tours in one pass in the w-streaming model with o(n log(n)) RAM. *CoRR*, abs/1710.04091, 2017.

[72] A. Goldberg, S. Plotkin, and G. Shannon. Parallel Symmetry-breaking in Sparse Graphs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 315–324, New York, NY, USA, 1987. ACM.

[73] Andrew V. Goldberg and Serge A. Plotkin. Parallel $(\Delta + 1)$-coloring of Constant-degree Graphs. *Inf. Process. Lett.*, 25(4):241–245, June 1987.

[74] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

[75] Michael T. Goodrich and S. Rao Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. *J. ACM*, pages 331–361, 1996.

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

[77] Martin Grohe and Oleg Verbitsky. Testing graph isomorphism in parallel by playing a game. In *Automata, Languages and Programming*, pages 3–14, 2006.

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

[79] MohammadTaghi Hajiaghayi and Marina Knittel. Matching affinity clustering: Improved hierarchical clustering at scale with guarantees. In *AAMAS*, pages 1864–1866, 2020.

[80] MohammadTaghi Hajiaghayi, Hamed Saleh, Saeed Seddighin, and Xiaorui Sun. String matching with wildcards in the massively parallel computation model. In *SPAA*, pages 275–284, 2021.

[81] MohammadTaghi Hajiaghayi, Saeed Seddighin, and Xiaorui Sun. Massively parallel approximation algorithms for edit distance and longest common subsequence. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1654–1672. SIAM, 2019.

[82] David G Harris, Johannes Schneider, and Hsin-Hao Su. Distributed $(\Delta + 1)$-coloring in sublogarithmic rounds. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 465–478. ACM, 2016.

[83] Nicholas JA Harvey, Christopher Liaw, and Paul Liu. Greedy and Local Ratio Algorithms in the MapReduce Model. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2018*.

[84] Christoph M Hoffmann and Michael J O'Donnell. Pattern matching in trees. *Journal of the ACM (JACM)*, 29(1):68–95, 1982.

[85] Jan Holub, Costas S Iliopoulos, Bořivoj Melichar, and Laurent Mouchard. Distributed pattern matching using finite automata. *Journal of Automata, Languages and Combinatorics*, 6(2):191–204, 2001.

[86] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for xml. In *ACM SIGPLAN Notices*, volume 36, pages 67–80. ACM, 2001.

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

[88] Piotr Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, pages 166–173. IEEE, 1998.

[89] Artur Jez and Markus Lohrey. Approximation of smallest linear tree grammar. *Inf. Comput.*, pages 215–251, 2016.

[90] Öjvind Johansson. Simple distributed *Delta*+1-coloring of graphs. *Inf. Process. Lett.*, 70(5):229–232, 1999.

[91] Tomasz Jurdzinski and Krzysztof Nowicki. MST in $O(1)$ rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2620–2632, 2018.

[92] Adam Kalai. Efficient pattern-matching with don't cares. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 655–656. Society for Industrial and Applied Mathematics, 2002.

[93] David R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 21–30. ACM/SIAM, 1993.

[94] David R Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4):601–640, 1996.

[95] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. Society for Industrial and Applied Mathematics, 2010.

[96] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[97] Raimondas Kiveris, Silvio Lattanzi, Vahab S. Mirrokni, Vibhor Rastogi, and Sergei Vassilvitskii. Connected components in mapreduce and beyond. In *ACM Symposium on Cloud Computing*, pages 18:1–18:13, 2014.

[98] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[99] Konstantinos Koiliaris and Chao Xu. A faster pseudopolynomial time algorithm for subset sum. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1062–1072. SIAM, 2017.

[100] Fabian Kuhn and Rogert Wattenhofer. On the Complexity of Distributed Graph Coloring. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 7–15, New York, NY, USA, 2006. ACM.

[101] Jakub Lacki, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Walking randomly, massively, and efficiently. In *ACM SIGACT*, pages 364–377, 2020.

[102] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 85–94. ACM, 2011.

[103] Moshe Lewenstein, Yakov Nekrich, and Jeffrey Scott Vitter. Space-efficient string indexing for wildcard pattern matching. *arXiv preprint arXiv:1401.0625*, 2014.

[104] Nathan Linial. Locality in Distributed Graph Algorithms. *SIAM J. Comput.*, 21(1):193–201, February 1992.

[105] Yu Liu, Longjiang Guo, Jinbao Li, Meirui Ren, and Keqin Li. Parallel algorithms for approximate string matching with k mismatches on cuda. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2414–2422. IEEE, 2012.

[106] Mi Lu and Hua Lin. Parallel algorithms for the longest common subsequence problem. *IEEE Trans. Parallel Distributed Syst.*, 5(8):835–848, 1994.

[107] M Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 1–10, New York, NY, USA, 1985. ACM.

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

[109] P Mateus and Y Omar. Quantum pattern matching. *arXiv preprint quant-ph/0508237*, 2005.

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

[111] Gary L. Miller and Vijaya Ramachandran. A new graph triconnectivity algorithm and its parallelization. In *STOC*, pages 335–344, 1987.

[112] Gary L. Miller, Vijaya Ramachandran, and Erich Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Comput.*, pages 687–695, 1988.

[113] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *FOCS*, pages 478–489, 1985.

[114] Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. *Adv. Comput. Res.*, pages 47–72, 1989.

[115] Gary L. Miller and John H. Reif. Parallel tree contraction, part 2: Further applications. *SIAM J. Comput.*, pages 1128–1147, 1991.

[116] Ashley Montanaro. Quantum pattern matching fast on average. *Algorithmica*, 77(1):16–39, 2017.

[117] Koji Nakano. Efficient implementations of the approximate string matching on the memory machine models. In *Networking and Computing (ICNC), 2012 Third International Conference on*, pages 233–239. IEEE, 2012.

[118] Danupon Nanongkai and Michele Scquizzato. Equivalence classes and conditional hardness in massively parallel fcomputations. In *PODS*, pages 33:1–33:16, 2019.

[119] Alessandro Panconesi and Aravind Srinivasan. Improved Distributed Algorithms for Coloring and Network Decomposition Problems. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 581–592, New York, NY, USA, 1992. ACM.

[120] Dimitrios Papadopoulos, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Practical authenticated pattern matching with optimal proof size. *Proc. VLDB Endow.*, pages 750–761, 2015.

[121] Merav Parter. $(\Delta+1)$ Coloring in the Congested Clique Model. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 160:1–160:14, 2018.

[122] Merav Parter and Hsin-Hao Su. Randomized $(\Delta+1)$-Coloring in $O(\log^* \Delta)$ Congested Clique Rounds. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 39:1–39:18, 2018.

[123] Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*, pages 315–323. IEEE, 2009.

[124] S Viswanadha Raju and A Vinayababu. Optimal parallel algorithm for string matching on mesh network structure. *International Journal of Applied Mathematical Sciences*, 3(2):167–175, 2006.

[125] VS Raju and A Vinayababu. Parallel algorithms for string matching problem on single and two-dimensional reconfigurable pipelined bus systems. *Journal of Computer Science*, 3(9):754–759, 2007.

[126] H Ramesh and V Vinay. String matching in o (n+ m) quantum time. *Journal of Discrete Algorithms*, 1(1):103–110, 2003.

[127] RAMAKRISHNAN Ramesh and IV Ramakrishnan. Nonlinear pattern matching in trees. *Journal of the ACM (JACM)*, 39(2):295–316, 1992.

[128] Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and circuits: (on lower bounds for modern parallel computation). In *SPAA*, pages 1–12, 2016.

[129] Huzur Saran and Vijay V. Vazirani. Finding k cuts within twice the optimal. *SIAM J. Comput.*, 24(1):101–108, 1995.

[130] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*, pages 114–122. ACM, 1981.

[131] Jean-Marc Steyaert and Philippe Flajolet. Patterns and pattern-matching in trees: an analysis. *Information and Control*, 58(1-3):19–58, 1983.

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

[133] Vadim G Vizing. On an estimate of the chromatic class of a p-graph. *Discret Analiz*, 3:25–30, 1964.

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

[135] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[136] Grigory Yaroslavtsev and Adithya Vadapalli. Massively parallel algorithms and hardness for single-linkage clustering under ℓ_p distances. In *ICML*, pages 5596–5605, 2018.

[137] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, pages 56–65, 2016.