MapReduce Algorithms For Massive Trees

MohammadHossein Bateni[1], Soheil Behnezhad[2], Mahsa Derakhshan[2], MohammadTaghi Hajiaghayi[2], and Vahab Mirrokni[1]

1 **Google Research, New York.**
`{bateni, mirrokni}@google.com`

2 **Department of Computer Science, University of Maryland.**
`{soheil, mahsa, hajiagha}@cs.umd.edu`

Abstract

Solving large-scale graph problems is a fundamental task in many real-world applications, and it is an increasingly important problem in data analysis. For this reason, in the past few years several big-data algorithms for graph problems have been proposed. Despite this large effort, for many classic graph problems we still do not know how to design algorithms that use only a sublinear number of machines and space in the input size. Specifically when the input graph is large and sparse, which is indeed the case for many real-world graphs, it becomes impossible to store and access all the vertices in one machine — something that is often taken for granted in designing algorithms for massive graphs. The theoretical model that we consider is the *Massively Parallel Communications* (MPC) model which is a popular theoretical model of MapReduce-like systems.

Dynamic programming is a powerful algorithmic technique that, in the context of graph problems, is specifically useful for trees – an important special case of sparse graphs. However, these algorithms are often inherently sequential. In this paper, we give an algorithmic framework to adopt a large family of such dynamic programs on MPC. We start by introducing two classes of dynamic programming problems, namely "(poly log)-expressible" and "linear-expressible" problems. We show that both classes can be solved efficiently using a sublinear number of machines and a sublinear memory per machine. To achieve this result, we introduce a series of techniques that can be plugged together. To illustrate the generality of our framework, we implement in $\tilde{O}(\log n)$ rounds of MPC, the dynamic programming solution of fundamental problems such minimum bisection, k-spanning tree, maximum independent set, longest path, etc., when the input graph is a tree.

To further motivate the importance of trees, we would like to mention the line of work in embedding general graphs into trees, which allow for approximation algorithms of a large family of *connectivity* and *cut* graph problems via trees and their parallel implementations.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases Big Data Algorithms, MapReduce, Graph Problems, Dynamic Programming

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.

1 Introduction

With the inevitable growth of the size of datasets to analyze, the rapid advance of distributed computing infrastructure and platforms (such as MapReduce, Spark [48], Hadoop [46], Flume [18], etc.), and more importantly the availability of such infrastructure to medium- and even small-scale enterprises via services at Amazon Cloud and Google Cloud, the need for developing better distributed algorithms is felt far and wide nowadays. The past decade has seen a lot of progress in studying important computer science problems in the large-scale setting, which led to either adapting the sequential algorithms to distributed settings or at times designing from scratch distributed algorithms for these problems [3, 6, 16, 21, 17].

Despite this trend, we still have limited theoretical understanding of the status of several fundamental problems when it comes to designing large-scale algorithms. In fact, even simple and widely used techniques such as the greedy approach or dynamic programming seem to suffer from an inherent sequentiality that makes them difficult to adapt in parallel or distributed settings on the aforementioned platforms. Finding methods to run generic greedy algorithms or dynamic programming algorithms on MapReduce, for instance, has broad applications. This is the main goal of this paper.

Model. We consider the most restrictive model, called Massively Parallel Communication or $\mathcal{MPC}$, among previously studied MapReduce-like models [31, 26, 13], which is "arguably the most popular" one [29]. Let n denote the input size and let m denote the number of available machines which is given in the input.[1] At each round, every machine can use a space of size $s = \tilde{O}(n/m)$ and run an algorithm that is preferably linear time (but at most polynomial time) in the size of its memory.[2] Machines may only communicate between the rounds, and no machine can receive or send more data than its memory.

Recently, Im, Moseley, and Sun [29] initiated a principled framework for simulating sequential dynamic programming solutions of Optimal Binary Search Tree, Longest Increasing Subsequence, and Weighted Interval Selection problems on the MapReduce setting. This is quite an exciting development, however, it is not clear whether similar ideas can be extended to other problems and in particular to natural and well-known graph-theoretic problems. It has in fact already been observed [34, 3] that many graph problems can be solved if the input graph is dense (i.e., the number of edges is superlinear in the number of nodes). In contrast, sparse graphs are much harder to handle. To see this, recall that the machines are assumed to have sublinear (in the input size) memory. This implies that if the input graph is sparse, e.g., if $|E| = O(|V|)$, we are unable to access all the vertices of the graph in one machine. A widely accepted conjecture in the literature is that computing any non-trivial property (such as connectivity) of general (i.e., not necessarily dense) graphs requires $\Omega(\log n)$ rounds of $\mathcal{MPC}$ (see e.g., [47] for a formal statement of the conjecture). In fact, we show that even for the seemingly easy special case of trees, designing efficient distributed algorithms requires involved approaches.

To further emphasize the importance of sparse graphs in designing distributed algorithms, note that most of the real-world massive graphs are indeed sparse. As an example, the social network graph of Facebook has billions of vertices (i.e., users) whereas the average degree of each vertex (i.e., the number of connections of each user) is approximately 190 [43].

In this paper, we give an algorithmic framework that could be used to simulate many natural dynamic programs on trees. Indeed we formulate the properties that make a dynamic program (on trees) amenable to our techniques. These properties, we show, are natural and are satisfied by many known algorithms for fundamental optimization problems. To illustrate the generality of our framework, we design $\tilde{O}(\log n)$ round algorithms for famous graph problems on trees, such as, minimum bisection, minimum k-spanning tree, maximum

[1] We assume there exists a small constant $0 < \delta < 1$ for which the number of machines is always greater than n^δ (otherwise there is no point in designing a distributed algorithm).

[2] We assume that the available space on each machine is more than the number of machines (i.e., $m \leq s$ and hence $m = O(\sqrt{n})$). It is argued in [6] that this is a realistic assumption since each machine needs to have at least the index of all the other machines to be able to communicate with them. Also, as argued in [29], in some cases, it is natural to assume the total memory is $\tilde{O}(n^{1+\epsilon})$ for a small $0 < \epsilon < 1$.

weighted matching, longest path, minimum vertex cover, maximum independent set, facility location, k-center, etc.

Being able to solve a problem on trees is a useful and versatile technique with far-reaching consequences in solving or approximating problems on general graphs. Due to the line of work [9, 10, 4, 22, 39] in embedding general graphs into trees, we can embed graph metrics into trees in such a way that all distances are preserved up to a factor of $O(\log n)$ on average. As a result, often a factor of $\mathrm{O}(\log n)$ appears in approximation algorithms of connectivity problems when we go from trees to general graphs. Räcke [39] extended embedding into trees for cut problems as well. Later the results for both connectivity and cut problems were extended to subtrees of the original graph [5, 14, 19]. Blelloch, Gupta, and Tangwongsan [14] show how the probabilistic tree embeddings can be computed in parallel. These algorithms, however, are yet to be adopted to the model that we consider and are beyond the scope of this paper which focuses on systematically adopting dynamic programs on trees.

1.1 Related Work

Though so far several models for MapReduce have been introduced (see, e.g., [23, 6, 25, 26, 30, 38, 41]), Karloff, Suri, and Vassilvitskii [31] were the first to present a refined and simple theoretical model for MapReduce. In their model, Karloff et al. [31] extract only key features of MapReduce and bypass several message-passing systems and parameters. This model has been extended since then [13, 6, 41] (in this paper we mainly use the further refined model by Andoni, Nikolov, Onak, and Yaroslavtsev [6]) and several algorithms both in theory and practice have been developed in these settings, often using sketching [27, 30], coresets [12, 36] and sample-and-prune [33] techniques. Examples of such algorithms include k-means and k-center clustering [8, 12, 28], general submodular function optimization [20, 33, 37] and query optimization [13].

Thanks to their common application to data mining tasks, distributed algorithms on graphs have received a lot of attention. In [34] Lattanzi, Moseley, Suri and Vassilvitskii, propose algorithms for several problems on dense graphs. Subsequently other authors study graph problems in the MapReduce model [3, 2, 7, 15, 35] but almost all the solutions apply only to dense graphs. In contrast in this paper we present a framework to solve optimization problems on sparse graphs using strictly sublinear space.

Another related area of research is the design of parallel algorithms. Parallel algorithms, instead of distributing the work-load into different machines, assume a shared memory is accessible to several *processors* of the same machine. Roughly speaking, MapReduce models are more powerful in the sense that they combine parallelism with sequentiality. That is, the internal computation of the machines is free and we only optimize the rounds of communication. In this work, we use the internal computation of the machines in a crucial way to design a unified and general framework to implement a large family of sequential dynamic programs on $\mathcal{MPC}$. We are not aware of any similar framework to parallelize dynamic programs in purely parallel settings.

1.2 Organization

We start with an overview of our results and techniques in Section 2. We give a formalization of dynamic programming classes in Section 3. Then in Section 4 we show how to solve a particular class of problems which we call (poly log)-expressible problems. In Section 5 we show how to solve a more general class of problems, namely, linear-expressible problems. Due to space limitations, we leave most of the detailed proofs and examples to the appendix.

2 Main Results & Techniques

We introduce a class of dynamic programming problems which we call f-expressible problems. Here, f is a function and we get classes such as $(\mathrm{poly}\log)$-expressible problems or linear-expressible problems. Roughly speaking, f is proportional to the amount of data that each node of the tree stores in the dynamic program. Thus, linear-expressible problems are generally harder to solve than $(\mathrm{poly}\log)$-expressible problems. (See Section 3 for the formal definition of these classes.)

$(\mathrm{poly}\log)$-Expressible Problems. Many natural problems can be shown to be $(\mathrm{poly}\log)$-expressible. For example, the following graph problems are all $(\mathrm{poly}\log)$-expressible if defined on trees: maximum (weighted) matching, vertex cover, maximum independent set, dominating set, longest path, etc. Intuitively, the dynamic programming solution of each of these problems, for any vertex v, computes at most a constant number of values. Our first result is to show that every $(\mathrm{poly}\log)$-expressible problem can be efficiently solved in $\mathcal{MPC}$. As a corollary of that, all the aforementioned problems can be solved efficiently on trees using the optimal total space of $\tilde{O}(n)$.

▶ **Theorem 1.** *For any given m, there exists an algorithm to solve any $(\mathrm{poly}\log)$-expressible problem in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability each machine uses a space of size at most $\tilde{O}(n/m)$ and runs an algorithm that is linear in its input size.*

Proof sketch. The first problem in solving dynamic programs on trees is that there is no guarantee on the depth of the tree. If the given tree has only logarithmic depth one can obtain a logarithmic round algorithm by simulating a bottom-up dynamic program in parallel, where nodes at the same level are handled in the same round simultaneously. This is reminiscent of certain parallel algorithms whose number of rounds depends on the diameter of the graph.

Unfortunately the input tree might be quite unbalanced, with superlogarithmic depth. An extreme case is a path of length n. In this case we can partition the path into equal pieces (despite not knowing a priori the depth of each particular node), handling each piece independently, and then stitching the results together. Complications arise, because the subproblems are not completely independent. Things become more nuanced when the input tree is not simply a path.

To resolve this issue, we adopt a celebrated *tree contraction* method of parallel computing to our model. Roughly speaking, the algorithm decomposes the tree into pieces of size at most $\tilde{O}(m)$ (i.e., we can fit each component completely on one machine), with small interdependence. Omitting minor technical details, the latter property allows us to almost independently solve the subproblems on different machines. This results in a partial solution that is significantly smaller than the whole subtree; therefore, we can send all these partial solutions to a master machine in the next round and merge them.

Linear-Expressible Problems. Although many natural problems are indeed $(\mathrm{poly}\log)$-expressible, there are instances that are not. Consider for example the *minimum bisection problem*. In this problem, the goal is to assign two colors (blue and red) to the vertices in such a way that minimizes the total weight of the edges between blue vertices and red vertices while half of the vertices are colored red and half of the vertices are colored blue. In the natural dynamic programming solution of this problem, for a subtree T of size n_T, we store

$\mathrm{O}(n_T)$ different values. That is, for any $i \in [n_T]$, the dynamic program stores the weight of the optimal coloring that assigns blue to i vertices and red to the rest of $n_T - i$ vertices. This problem is not necessarily (poly log)-expressible unless we find another problem specific dynamic programming solution for it. However, it can be shown that minimum bisection, as well as many other natural problems, including k-spanning-tree, k-center, k-median, etc., are linear-expressible.

It is notoriously more difficult to solve linear-expressible problems using a sublinear number of machines and a sublinear memory per machine. However, we show that it is still possible to obtain the same result using a more involved algorithm.

▶ **Theorem 2** (Main Result). *For any given m, there exists an algorithm to solve any linear-expressible problem that is splittable in $\tilde{\mathrm{O}}(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability each machine uses a space of size $\tilde{\mathrm{O}}(n^{4/3}/m)$, and runs an algorithm that is polynomial in its space size for some $0 < \epsilon < 1$.*

Proof sketch. Recall that linear-expressibility implies that the dynamic programming data on each node, can be as large as the size of its subtree (i.e., even up to $\mathrm{O}(n)$). Therefore, even by using the tree decomposition technique, the partial solution that is computed for each component of the tree can be linear in its size. This means that the idea of sending all these partial data to one master machine, which worked for (poly log)-expressible problems, does not work here since when aggregated, they take as much space as the original input. Therefore we have to distribute the merging step among the machines.

Assume for now that each component that is obtained by the tree decomposition algorithm is contracted into a node and call this *contracted tree*. The tree decomposition algorithm has no guarantee on the depth of the contracted tree and it can be super-logarithmic; therefore a simple bottom-up merging procedure does not work. However, it is guaranteed that the contracted tree itself (i.e., when the components are contracted) can be stored in one machine. Using this, we send the contracted tree to a machine and design a *merging schedule* that informs each component about the round at which it has to be merged with each of its neighbours. The merging schedule ensures that after $\mathrm{O}(\log n)$ phases, all the components are merged together. The merging schedule also guarantees that the number of neighbours of the components, after any number of merging phases, remains constant. This is essential to allow (almost) independent merging for many linear-expressible problems such as minimum bisection.

Observe that after a few rounds, the merged components grow to have up to $\Omega(n)$ nodes and even the partial data of one component cannot be stored in one machine. Therefore, even merging the partial data of two components has to be distributed among the machines. For this to be possible, we use a *splitting* technique of independent interest that requires a further *splittability* property on linear-expressible problems. Indeed we show that the aforementioned linear-expressible problems, such as minimum bisection and k-spanning tree, have the splittability property, and therefore Theorem 2 implies they can also be solved in $\tilde{\mathrm{O}}(\log n)$ rounds of $\mathcal{MPC}$.

3 Dynamic Programming Classes

To understand the complexity of different dynamic programs in the $\mathcal{MPC}$ model, we first attempt to classify these problems. The goal of this section is to introduce a class of problems which we call f-expressible problems.

We say a problem P *is defined on trees*, if the input to P is a rooted tree $\mathcal{T}$, where each vertex v of $\mathcal{T}$ may contain data of size up to $\text{poly}(\log n)$, denoted by `v.data`.[3]

Dynamic programming is effective when the solution to the problem can be computed recursively. On trees, the recursion is usually defined on the nodes of the tree, where each node computes its data using the data provided by its children. Let $\mathcal{D}_P$ denote a dynamic program that solves a problem P that is defined on trees. Also let $P_{\mathcal{T}}$ be an instance of P and denote its input by $\mathcal{T}$. We use $\mathcal{D}_P(\mathcal{T})$ to denote the solution of $P_{\mathcal{T}}$ and use $\mathcal{D}_P(\text{v})$ to denote the dynamic data that $\mathcal{D}_P$ computes for a vertex v of $\mathcal{T}$. The first property that we define is "binary adaptability".

Binary Adaptability. To illustrate when this property holds, consider any vertex v of an input tree and a given ordering of its children $(u_1, u_2, \ldots, u_k)$ and let $\mathcal{D}(u)$ denote the dynamic data of vertex u. As long as the operation to compute the dynamic data of v could be viewed as an aggregation of binary operations over the dynamic data that is provided by its children in the given order, i.e., $\mathcal{D}(v) = f_1(\mathcal{D}(u_1), f_2(\mathcal{D}(u_2), \ldots))$, this property is satisfied. (For example, $\max\{\mathcal{D}(u_1), \mathcal{D}(u_2), \mathcal{D}(u_3)\}$ could be viewed as $\max\{\mathcal{D}(u_1), \max\{\mathcal{D}(u_2), \mathcal{D}(u_3)\}\}$). Most of the natural dynamic programs satisfy this property.

To formally state this property, we first define "binary extensions" of trees and then define a problem to be binary adaptable if there exists a dynamic programming solution that given any binary extension of an input, calculates the solution of the original tree.

▶ **Definition 3** (Binary Extension). A binary tree $\mathcal{T}^b$ is a binary extension of a tree $\mathcal{T}$ if there is a one-to-one (and not necessarily surjective) mapping $f : V(\mathcal{T}) \to V(\mathcal{T}^b)$ such that $f(v)$ is an ancestor of $f(u)$ in $\mathcal{T}^b$ if v is an ancestor of u in $\mathcal{T}$. We assume the data of each vertex v in T is also stored in its equivalent vertex, $f(v)$ in $\mathcal{T}^b$.

Intuitively, a binary extension of a tree $\mathcal{T}$ is a binary tree that is obtained by adding auxiliary vertices to $\mathcal{T}$ in such a way that ensures any ancestor of each vertex remains its ancestor.

▶ **Definition 4** (Binary Adaptability). Let P be a problem that is defined on trees. Problem P is *binary adaptable* if there exists a dynamic program $\mathcal{D}_P$ where for any instance $P_{\mathcal{T}}$ of P with $\mathcal{T}$ as its input tree and for any binary extension $\mathcal{T}^b$ of $\mathcal{T}$, the output of $\mathcal{D}_P(\mathcal{T}^b)$ is a solution of $P_{\mathcal{T}}$. We say $\mathcal{D}_P$ is a binary adapted dynamic program for P.

We are now ready to define f-expressiveness. We will mainly consider $(\text{poly}\log)$-expressible and linear-expressible problems in this paper, however the definition is general.

f-Expressiveness. This property is defined on the problems that are binary adaptable. Roughly speaking, it specifies how large the data that we store for each subtree should be to be able to merge subtrees efficiently.

▶ **Definition 5** (f-Expressiveness). Let P be a binary adaptable problem and let $\mathcal{D}_P$ be its binary adapted dynamic program. Moreover, let $\mathcal{T}^b$ be a binary extension of a given input to P. For a function f, we say P is f-expressible if the following conditions hold.

1. The dynamic data of any given vertex v of $\mathcal{T}^b$ has size at most $\tilde{O}(f(n_v))$ where n_v denotes the number of descendants of v.

[3] Consider weighted trees as an example of the problems that define an additional data on the nodes (in case the edges are weighted, each vertex stores the weight of its edge to its parent).

2. There exist two algorithms $\mathcal{C}$ (compressor), and $\mathcal{M}$ (merger) with the following properties. Consider any arbitrary connected subtree T of $\mathcal{T}^b$ and let v denote the root vertex of T. If for at most a constant number of the leaves of T the dynamic data is unknown (call them the unknown leaves of T), algorithm $\mathcal{C}$ returns a partial data of size at most $\tilde{O}(f(n_T))$ for T (without knowing the dynamic data of the unknown leaves) in time $\text{poly}(f(n_T))$ such that if this partial data and the dynamic data of the unknown leaves are given to $\mathcal{M}$, it returns $\mathcal{D}_P(\text{v})$ in time $\text{poly}(f(n_T))$.
3. For two disjoint subtrees T_1 and T_2 that are connected to each other with one edge, and for subtree $T = T_1 + T_2$, if there are at most a constant number of the leaves of T that are unknown, then $\mathcal{M}(\mathcal{C}(T_1), \mathcal{C}(T_2))$ returns $\mathcal{C}(T)$ in time $\text{poly}(f(n_T))$.

We remark that graph problems such as maximum (weighted) matching, vertex cover, maximum independent set, dominating set, longest path, etc., when defined on trees, are all $(\text{poly}\log)$-expressible. Also, problems such as minimum bisection, k-spanning tree, k-center, and k-median are linear-expressible on trees.

4 Warm-Up: Solving $(\text{poly}\log)$-Expressible Problems

The first problem in solving dynamic programs on trees is that there is no guarantee on the depth of the tree. If the given tree has only logarithmic depth one can obtain a logarithmic round algorithm by simulating a bottom-up dynamic program in parallel, where nodes at the same level are handled in the same round simultaneously. Unfortunately the input tree might be quite unbalanced, with superlogarithmic depth.

To resolve this issue, we adopt a celebrated *tree contraction* method of parallel computing to our model. Roughly speaking, the algorithm decomposes the tree into pieces of bounded size, with small interdependence. The latter property allows us to almost independently solve the subproblems. To remain self-contained, we describe the tree decomposition algorithm and its proof in Appendix D. The following theorem is an informal restatement of the main result of that section.

Theorem 30. (Informal restatement) There exists a randomized algorithm that converts any given tree to a binary version of it and then decomposes the binary tree into connected subgraphs of bounded size such that each subgraph can be stored in one machine and is connected to at most 3 other subgraphs. For any given m, the algorithm uses m machines and with high probability terminates in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$. Each machine uses a memory of size at most $\tilde{O}(n/m)$ and runs an algorithm that is linear in its memory size.

To motivate the need to use an involved decomposition algorithm, we first give hints on why trivial approaches do not work. A natural (trivial) algorithm is to choose a set of randomly selected *joint* edges (or vertices) and divide the tree based on them. Note that the main difficulty is that we need to bound the size of the largest component with a high probability. It could be shown that this trivial algorithm, even for the simple case of full binary trees fails. More precisely, it could be shown that the size of the component that contains the root is of size $\Omega(n)$ (instead of $\tilde{O}(n/m)$) with a high probability.

Overview of Tree Decomposition. At first, the input tree is converted into a binary tree. To do this, the algorithm first computes the degree of each vertex and then adds the needed *auxiliary* vertices to it. After that, roughly, the challenge is that the vertices have little knowledge about their status in the tree (e.g., they do not know their depth) and all the decomposition happens using the local information that is accessible to the vertices. To do this, the algorithm proceeds in *phases* (not rounds). In each phase, the algorithm *selects* a subset of vertices based on a set of local rules such as their degree, status of their parent, etc. Next, each vertex is *merged* to its closest selected ancestor. Interestingly, it can be proved that in fact these local rules guarantee that the size of the maximum component is bounded by $\tilde{O}(n/m)$ with high probability.

After decomposing the tree into pieces that each can be stored in one machine, we can solve (poly log)-expressible problems as follows. We first use the compressor algorithm that is guaranteed to exist for (poly log)-expressible problems and store a data of size at most $\operatorname{poly}\log(n)$ bits for each component. Then in the next step, since there are at most $\tilde{O}(m)$ components, we can send the partial data of all components into one machine. Then in only one round, we merge all these partial data sequentially in one machine; obtaining the following result.

Theorem 1. For any given m, there exists an algorithm to solve any (poly log)-expressible problem in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability each machine uses a space of size $\tilde{O}(n/m)$ and runs an algorithm that is linear in its input size.

For space limits, we defer the formal proof of this theorem and examples of (poly log)-expressible problems to Appendix A.

5 Solving Linear-Expressible Problems

5.1 The Splittability Property

The main goal of this section is to define a property called "splittability" for linear-expressible problems, that indeed holds for the aforementioned linear-expressible problems and prove, by using a total memory of $\tilde{O}(n^{4/3})$, we are able to parallelize linear-expressible problems that are splittable in $\tilde{O}(\log n)$ rounds. The main reason behind the need to define this extra property for linear-expressible problems, is that in contrast to (poly log)-expressible problems, the partial data of the subtrees in linear-expressible problems might have a relatively large size, hence when merging two subtrees, we are not able to access their partial data in one machine. As a result, we need to be able to distribute the merging step among different machines. It is worth mentioning that we prove the total memory of $\tilde{O}(n^{4/3})$ is in some sense tight, unless we limit the number of machines to be too small, which defeats the purpose of MapReduce, which is to have maximum possible parallelization.

Intuitively, a problem instance P is splittable if it is possible to represent it with a set of n *objects* such that the output of the problem depends only on the pair-wise relations of these objects. In the DP context, the splittability property is useful for linear-expressible problems. The splittability property ensures that these partial data can be effectively merged using the splitting technique.

▶ **Definition 6** (Splittability). Consider a linear-expressible problem P and a binary adapted dynamic program $\mathcal{D}_P$ for it. We know since P is a linear-expressible problem, for any subtree T of $\mathcal{T}^b$ with n_T vertices, a compressor function returns a partial data of size at most $\tilde{O}(n_T)$ if at most a constant number of the leaves of T are unknown. Denote this partial data by $\mathcal{C}(T)$. We say problem P is splittable, if for any given connected subtree T' of $\mathcal{T}^b$, the partial data $\mathcal{C}(T')$ can be represented as a vector $\mathcal{C}(\vec{T'})$ of at most $\tilde{O}(n_{T'})$ elements such that for any connected subtrees T_1 and T_2 of $\mathcal{T}^b$ that are connected with an edge (with T being $T_1 + T_2$) the following condition holds. There exist two algorithms $\mathcal{S}$ (sub unifier) and $\mathcal{U}$ (unifier) such that for any consecutive partitioning $(P_1, P_2, \dots, P_{k_1})$ of $\mathcal{C}(\vec{T_1})$ and any consecutive partitioning $(Q_1, Q_2, \dots, Q_{k_2})$ of $\mathcal{C}(\vec{T_2})$,

1. Algorithm $\mathcal{S}(P_i, Q_j)$ returns a vector of size at most $|P_i| + |Q_j|$ in time $\text{poly}(|P_i| + |Q_j|)$ such that each element of this vector "affects" at most one element of $\mathcal{C}(\vec{T})$.
2. Given the elements in $\mathcal{S}(P_i, Q_j)$ that "affect" an element a in $\mathcal{C}(\vec{T})$ for every $1 \leq i \leq k_1$ and $1 \leq j \leq k_2$, $\mathcal{U}$ computes the value of element a using linear space and polynomial time in input size.

The linear-expressible problems such as minimum bisection, k-spanning tree, etc., that we have mentioned before are all indeed splittable. For the proof and concrete examples of unifier and sub unifier algorithms see Appendix B.

5.2 Solving Splittable Linear-Expressible Problems

We show with a slightly more total space (the space on each machine is still sublinear) it is possible to design parallel algorithms in $\tilde{O}(\log n)$ rounds for linear-expressible problems if they are splittable.

Theorem 2. For any given m, there exists an algorithm to solve any linear-expressible problem defined on trees that is also splittable, in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability each machine uses a space of size $\tilde{O}(n^{4/3}/m)$, and runs an algorithm that is polynomial in its input size.

As a corollary of Theorem 2, we give logarithmic round algorithms for minimum bisection, and k-spanning tree problems using an overall space of $\tilde{O}(n^{1+\epsilon})$ for some constant $\epsilon < 1$.

Our algorithm, similar to the algorithm given for (poly log)-expressible problems starts with decomposing a binary extension of the input tree. However, note that for linear-expressible problems, we cannot store the data of all components in one machine since the space limit will be violated. (The size of the total data would be roughly the same as the size of input.) Therefore we merge the components in multiple rounds. Unfortunately, the components tree could be unbalanced. To overcome this difficulty, we first partition the components tree. This partitioning of the components tree, specifies which components at which rounds, should be merged together. The merging process cannot be done in one machine either. Mainly because the partial data of even one component could be very large. Using the splittability property we distribute the data of the components among the machines and merge them in parallel.

We first prove the partitioning lemma and then prove Theorem 2.

▶ **Definition 7** (border vertices). For any subtree T of a given tree $\mathcal{T}$, the border vertices are the set of vertices in $V(T)$ that are connected with an edge of tree $\mathcal{T}$ to a vertex in $V(\mathcal{T}) - V(T)$.

▶ **Lemma 8.** *There exists a linear time algorithm that given a binary tree $\mathcal{T}$ with $n_{\mathcal{T}}$ vertices, and a set of border vertices $S_{\mathcal{T}} \in V(\mathcal{T})$ with size at most 3, finds a partitioning $P_{\mathcal{T}} = \langle H_1, H_2, \ldots, H_c\rangle$ of $\mathcal{T}$, such that $|V(H_i)| \leq \frac{2n_{\mathcal{T}}}{3}$ for any $i \leq c$, and the total number of border vertices of any H_i based on $(P_{\mathcal{T}}, S_{\mathcal{T}})$ is at most 3.*

We defer the proof of Lemma 8 to the appendix. We are now ready to prove Theorem 2.

Proof of Theorem 2. Let $\mathcal{T}$ denote the input tree. Since any linear-expressible problem is binary adaptable, there exists a dynamic program $\mathcal{D}_P$ where for the instance $P_{\mathcal{T}}$ of P with $\mathcal{T}$ as its input tree and for any binary extension $\mathcal{T}^b$ of $\mathcal{T}$, the output of $\mathcal{D}_P(\mathcal{T}^b)$ is a solution of $P_{\mathcal{T}}$. We convert $\mathcal{T}$ to its binary extension $\mathcal{T}^b$ in O(1) round of $\mathcal{MPC}$ such that with high probability each machine uses space $\tilde{O}(n/m)$ (see Lemma 26 in the appendix). Since the output of $\mathcal{D}_P$ is a solution of $P_{\mathcal{T}}$, it suffices to solve $\mathcal{D}_P$.

The first stage in this algorithm is decomposing tree $\mathcal{T}^b$, which by Theorem 30 is possible in $\tilde{O}(\log n)$ round of $\mathcal{MPC}$ such that with high probability each machine uses $\tilde{O}(n/m)$ time and space. Let $C(\mathcal{T}^b)$ denote the set of components in decomposition of $\mathcal{T}^b$ by Definition 29, and let $\mathcal{T}^c$ denote the rooted component tree that is given by contracting all components in $C(\mathcal{T}^b)$. Roughly speaking, we partition $\mathcal{T}^c$ to at most three subtrees T_1, T_2, and T_3, such that $\max_{i=1}^3 |\mathcal{T}_i| \leq \frac{2}{3}|\mathcal{T}^c|$, and $\max_{i=1}^3 |B_i| \leq 3$, where B_i denotes the set of border vertices in $\mathcal{T}_i$. We find the partial dynamic data for these partitions recursively and we merge them to find the dynamic data for $\mathcal{T}^c$. Algorithm 1 is the overview of this algorithm.

Algorithm 1

```
1: procedure SolveLinearExpressible($T$)
2:     Decompose $\mathcal{T}^b$ and let $C(\mathcal{T}^b)$ denote the set of components in decomposition of $\mathcal{T}^b$.
3:     Let $\mathcal{T}^c$ be the component tree that is given by contracting all components in $C(\mathcal{T}^b)$ .
4:     GetDynamicData($\mathcal{T}^c$, $\emptyset$)
5: procedure GetDynamicData($\mathcal{T}$, $B$)          ▷ $B$ is the set of border vertices in $V(\mathcal{T})$
6:     if $|\mathcal{T}| = 1$ then
7:         Based on $B$, compute and return the dynamic data for $T$.
8:     else
9:         Partition $\mathcal{T}$ to partitions $T_1, T_2$, and $T_3$, such that (1) $\max_{i=1}^3 |\mathcal{T}_i| \leq \frac{2}{3}|\mathcal{T}|$, (2) $\max_{i=1}^3 |B_i| \leq 3$ where $B_i$ denotes the set of border vertices in $\mathcal{T}_i$.
10:        for every $i$ in $\{1,2,3\}$ do
11:            $D_i \leftarrow$ GetDynamicData($\mathcal{T}_i$, $B_i$)
12:        Compute the dynamic data for $\mathcal{T}$, by merging $D_1, D_2$, and $D_3$, and return it.
```

Partitioning stage. The second stage of our algorithm partitions $\mathcal{T}^c$ using the algorithm given in Lemma 8. Let n_c denote the number of vertices of tree $\mathcal{T}^c$. Starting from $\mathcal{T}^c$, in each step, we partition the remaining subtrees by removing one edge from each and repeat this until we have n_c disjoint vertices. Since each sub-tree is split into sub-trees of a constant fraction size of it, this stage halts after $O(\log n_c)$ steps, so the overall running time of this stage is $\tilde{O}(n_c) \leq \tilde{O}(m)$. At the end of this stage, for each edge e of $\mathcal{T}^c$, we store in which step it was deleted and use this data to design an algorithm to merge the dynamic data of the sub-trees and finally find an optimal solution for $\mathcal{T}^d$. More precisely, after this stage, the edges of $\mathcal{T}^c$ are partitioned into subsets $E_1, E_2, \ldots, E_{2S}$ where $S = O(\log n)$ is the number of steps this stage takes, and for any $i \in [S]$, E_{2i-1} and E_{2i} are respectively the set of edges deleted by Algorithm 3 and Algorithm 4 in the $(S-i+1)$-th step of this stage. We store $E_1, E_2, \ldots, E_{2S}$ in any machine in $\mathcal{M}$ without violating the memory limits since the total number of edges of $\mathcal{T}^c$ is at most $\tilde{O}(m)$

The third stage of the algorithm is the merging stage that runs exactly in the reverse order of the previous stage. Starting from single vertices of $\mathcal{T}^c$, which are equivalent to components in $C(\mathcal{T}^b)$, in the first round, the algorithm continues to merge the dynamic data of two subtrees if there exists an edge $e \in E_1$ that connects them. Note that the size of any component in $C(\mathcal{T}^b)$ is at most $\tilde{O}(n/m)$, so it is possible to find its partial dynamic data in one machine.

Let vector $D(T)$ denote the dynamic data (It is the partial dynamic data if there is any unknown data in T.) for any subtree T. Since the maximum number of border vertices for any partition is a constant, by linear-expressible (Definition 5) we can store the dynamic data for any partition in linear space to the size of the partition. Moreover by the same definition, if $T = T_1 + T_2$ it is possible to construct the dynamic data for T by merging the dynamic data for T_1 and T_2. Note that to update the dynamic data for any sub-tree T, we have to update $O(|V(T)|)$ different values, and to update each of them, it is possible that we need more than$\mathrm{O}(n/m)$ space. This unfortunately cannot be done in one machine since it violates the memory limit of $O(n^{1+\epsilon})$ on each machine. To handle this issue we use the following method to merge the dynamic data of two components.

Merging stage. This stage of algorithm takes $2R$ steps. Intuitively, for any $1 \leq r \leq 2R$ and for any edge in E_r that is between two partitions T_i^r and T_j^r, in the first round of step r we assign any machine a portion of dynamic data of T_i^r and T_j^r. This machine is responsible to merge these two portions using the function $\mathcal{S}$ (sub unifier)in Definition 6. Let T^{r+1} be the subtree which is the result of merging subtrees T_i^r and T_j^r in step r, and let $D(T_i^r)[b]$ denote the b-th element of vector $D(T_i^r)$. More precisely, for any two positive integers $k_1, k_2 \leq \lceil\sqrt{m}\rceil$ dynamic data $D(T_i^r)[(k_1-1)\alpha_i : k_1\alpha_i]$ and $D(T_j^r)[(k_2-1)\alpha_j : k_2\alpha_j]$ is assigned to machine μ with index $k_1\sqrt{m} + k_2$, where for any i, $\alpha_i = |T_i^r|/\sqrt{m}$. (For any subtree T and any two positive integers a, b such that $a < b$ $D(T)[a : b]$ denotes the sub-vector $(D(T)[a], \dots, D(T)[b])$.) This machine is responsible to merge the given portion of dynamic data and generate $D^\mu(T)$, such that $|D^\mu(T)| = \mathrm{O}(D(T_i^r) + D(T_j^r))$, and any element in $D^\mu(T)$ affects at most one element in $D(T)$. This is possible by function $\mathcal{S}$. Therefore to compute $D(T)[a]$ for any valid a in one round of $\mathcal{MPC}$ it suffices if for every $\mu \in \mathcal{M}$ we have the element in $D^\mu[T]$ that affects $D(T)[a]$ (if there exists any) in the same machine, and then use the function $\mathcal{U}$ (unifier). To achieve this for any T^{r+1} in the set of partitions of $\mathcal{T}^b$ in round r and for any $a \leq |V(T^{r+1})|$ we assign each $D(T^{r+1})[a]$ a unique index in $[|V(\mathcal{T}^b)|]$ generated based on T^{r+1} and a. Using this index and a hash function h chosen uniformly at random from a family of $\log n-$universal hash functions, we distribute pairs of (T^{r+1}, a) over all m machines, and any machine μ sends at most one element of $D_c^\mu(T^{r+1})$ that affects $D(T^{r+1})[a]$ to the machine μ' iff pair (T^{r+1}, a) is assigned to machine μ' using hash function h. Therefore, all the data needed to compute $D(T^{r+1})[a]$, is gathered in one machine, and in the second round of r-th step, we run function $\mathcal{U}$ on all the machines to compute the dynamic data of T^{r+1}.

Time and space analysis. By Lemma 26 and Theorem 30 it is possible to convert any given tree to a binary extension of it, and decompose the binary extension in $\tilde{\mathrm{O}}(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability, each machine uses a memory of size at most $\tilde{\mathrm{O}}(n^{1+\epsilon}/m)$ and runs an algorithm that is linear in its memory size. Since the size of the component tree $\mathcal{T}^c$ is $\mathrm{O}(m)$, and the partitioning stage takes $S = \mathrm{O}(\log n)$ steps, by 8 it is possible to do the partitioning stage of the algorithm using $\tilde{\mathrm{O}}(n^{1+\epsilon}/m)$ time and space in $\mathrm{O}(1)$ round. (Note that $\tilde{\mathrm{O}}(m) \leq \tilde{\mathrm{O}}(n^{1+\epsilon}/m)$.) In addition, the merging stage of the algorithm takes $S = \mathrm{O}(\log n)$ round, and the space needed on each machine is $\tilde{\mathrm{O}}(n^{1+\epsilon}/m)$. Since in round r each vertex in $\mathcal{T}^b$ is at most in one partition, the total size of the partitions in one round is $\mathrm{O}(n)$. Note that, the dynamic data of each partition is linear to its size. Therefore when in the first round of any step $r \in [2R]$ of merging we assign each machine $\mathrm{O}(1/\sqrt{m})$ portion of the dynamic data of each partition, the overall data assigned to each machine is $\mathrm{O}(n/\sqrt{m})$. We set $\epsilon = \frac{1}{3}$. Since $m < n^{(1+\epsilon)/2}$ it is easy to see that $\mathrm{O}(n/\sqrt{m}) \leq \mathrm{O}(n^{1+\epsilon}/m)$, so computing function $\mathcal{S}$in each round takes $\mathrm{O}(n^{1+\epsilon}/m)$ space and $\mathrm{poly}(n^{1+\epsilon}/m)$ time in any machine. Moreover, with high probability the data assigned to each machine in the second round of any step r is $\tilde{\mathrm{O}}(n^{1+\epsilon}/m)$ since we distribute any pair of partition T in step r and $a \leq |T|$ using a hash function h chosen uniformly at random from a family of $\log n-$universal hash functions. The maximum data that any machine receives for any pair (T, a) that is assigned to it by hash function h is at most $\mathrm{O}(m)$ and the overall data assigned to all the machines is $\mathrm{O}(n\sqrt{m})$. Note that, $\mathrm{O}(n\sqrt{m}) \leq \mathrm{O}(n^{1+\epsilon})$. Therefore by Lemma 21 with high probability the maximum load over all the machines is $\tilde{\mathrm{O}}(n^{1+\epsilon}/m)$, so each machine takes $\tilde{\mathrm{O}}(n^{1+\epsilon}/m)$ space and $\mathrm{poly}(n^{1+\epsilon}/m)$ time to compute unifier for all the pairs (T, a) that are assigned to it in any round. ◀

We further show, in details, how these techniques could be used to solve the minimum bisection problem on trees. For space limitations, we defer the discussion about minimum bisection to appendix.

Acknowledgements. We would like to thank Silvio Lattanzi for fruitful discussions and his helpful comments.

References

1 Karl Abrahamson, Norm Dadoun, David G. Kirkpatrick, and T Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, 1989.

2 Kook Jin Ahn and Sudipto Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 202–211. ACM, 2015.

3 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 459–467, 2012. URL: `http://portal.acm.org/citation.cfm?id=2095156&CFID=63838676&CFTOKEN=79617016`.

4 Noga Alon, Richard M Karp, David Peleg, and Douglas West. A graph-theoretic game and its application to the k-server problem. *SIAM Journal on Computing*, 24(1):78–100, 1995.

5 Reid Andersen and Uriel Feige. Interchanging distance and capacity in probabilistic mappings. *arXiv preprint arXiv:0907.3631*, 2009.

6 Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pages 574–583. ACM, 2014.

7 Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. Densest subgraph in streaming and mapreduce. *Proceedings of the VLDB Endowment*, 5(5):454–465, 2012.

8 Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633, 2012.

9 Yair Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*, pages 184–193. IEEE, 1996.

10 Yair Bartal. On approximating arbitrary metrices by tree metrics. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 161–168. ACM, 1998.

11 MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and vahab Mirrokni. Affinity clustering: Hierarchical clustering at scale. 2017.

12 MohammadHossein Bateni, Aditya Bhaskara, Silvio Lattanzi, and Vahab Mirrokni. Distributed balanced clustering via mapping coresets. In *Advances in Neural Information Processing Systems*, pages 2591–2599, 2014.

13 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 273–284. ACM, 2013.

14 Guy E Blelloch, Anupam Gupta, and Kanat Tangwongsan. Parallel probabilistic tree embeddings, k-median, and buy-at-bulk network design. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 205–213. ACM, 2012.

15 Flavio Chierichetti, Nilesh N. Dalvi, and Ravi Kumar. Correlation clustering in mapreduce. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 641–650, 2014. URL: `http://doi.acm.org/10.1145/2623330.2623743`, `doi:10.1145/2623330.2623743`.

16 Rajesh Chitnis, Graham Cormode, Hossein Esfandiari, MohammadTaghi Hajiaghayi, Andrew McGregor, Morteza Monemizadeh, and Sofya Vorotnikova. Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1326–1344, 2016. URL: `http://dx.doi.org/10.1137/1.9781611974331.ch92`, `doi:10.1137/1.9781611974331.ch92`.

17 Rajesh Hemant Chitnis, Graham Cormode, Mohammad Taghi Hajiaghayi, and Morteza Monemizadeh. Parameterized streaming: Maximal matching and vertex cover. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1234–1251, 2015. URL: `http://dx.doi.org/10.1137/1.9781611973730.82`, `doi:10.1137/1.9781611973730.82`.

18 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

19 Michael Elkin, Yuval Emek, Daniel A Spielman, and Shang-Hua Teng. Lower-stretch spanning trees. *SIAM Journal on Computing*, 38(2):608–628, 2008.

20 Alina Ene and Huy L Nguyen. Random coordinate descent methods for minimizing decomposable submodular functions. In *ICML*, pages 787–795, 2015.

21 Hossein Esfandiari, Mohammad Taghi Hajiaghayi, Vahid Liaghat, Morteza Monemizadeh, and Krzysztof Onak. Streaming algorithms for estimating the matching size in planar graphs and beyond. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1217–1233, 2015. URL: `http://dx.doi.org/10.1137/1.9781611973730.81`, `doi:10.1137/1.9781611973730.81`.

22 Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 448–455. ACM, 2003.

23 Jon Feldman, S Muthukrishnan, Anastasios Sidiropoulos, Cliff Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Transactions on Algorithms (TALG)*, 6(4):66, 2010.

24 Hillel Gazit, Gary L Miller, and Shang-Hua Teng. Optimal tree contraction in the erew model. In *Concurrent Computations*, pages 139–156. Springer, 1988.

25 Ashish Goel and Kamesh Munagala. Complexity measures for map-reduce, and comparison to parallel computing. *arXiv preprint arXiv:1211.6526*, 2012.

26 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.

27 Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE transactions on knowledge and data engineering*, 15(3):515–528, 2003.

28 Sungjin Im and Benjamin Moseley. Brief announcement: Fast and better distributed mapreduce algorithms for k-center clustering. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 65–67. ACM, 2015.

29 Sungjin Im, Benjamin Moseley, and Xiaorui Sun. Efficient massively parallel methods for dynamic programming. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*. ACM, 2017.

30 Daniel M Kane, Jelani Nelson, and David P Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 41–52. ACM, 2010.

31 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.

32 Valerie King, Chung Keung Poon, Vijaya Ramachandran, and Santanu Sinha. An optimal erew pram algorithm for minimum spanning tree verification. *Information Processing Letters*, 62(3):153–159, 1997.

33 Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. Fast greedy algorithms in mapreduce and streaming. *ACM Transactions on Parallel Computing*, 2(3):14, 2015.

34 Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 85–94, 2011. URL: `http://doi.acm.org/10.1145/1989493.1989505`, `doi:10.1145/1989493.1989505`.

35 Brendan Lucier, Joel Oren, and Yaron Singer. Influence at scale: Distributed computation of complex contagion in networks. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 735–744, 2015. URL: `http://doi.acm.org/10.1145/2783258.2783334`, `doi:10.1145/2783258.2783334`.

36 Vahab S. Mirrokni and Morteza Zadimoghaddam. Randomized composable core-sets for distributed submodular maximization. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 153–162, 2015. URL: `http://doi.acm.org/10.1145/2746539.2746624`, `doi:10.1145/2746539.2746624`.

37 Baharan Mirzasoleiman, Amin Karbasi, Rik Sarkar, and Andreas Krause. Distributed submodular maximization: Identifying representative elements in massive data. In *Advances in Neural Information Processing Systems*, pages 2049–2057, 2013.

38 Andrea Pietracaprina, Geppino Pucci, Matteo Riondato, Francesco Silvestri, and Eli Upfal. Space-round tradeoffs for mapreduce computations. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 235–244. ACM, 2012.

39 Harald Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 255–264. ACM, 2008.

40 Margaret Reid-Miller, Gary L Miller, and Francesmary Modugno. List ranking and parallel tree contraction. *Synthesis of Parallel Algorithms*, pages 115–194, 1993.

41 Tim Roughgarden, Sergei Vassilvitskii, and Joshua R Wang. Shuffles and circuits:(on lower bounds for modern parallel computation). In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 1–12. ACM, 2016.

42 Jeanette P Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff–hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.

43 Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

44 Salil P Vadhan et al. Pseudorandomness. *Foundations and Trends® in Theoretical Computer Science*, 7(1–3):1–336, 2012.

45 Mark N Wegman and J Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences*, 22(3):265–279, 1981.

46 Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

47 Grigory Yaroslavtsev and Adithya Vadapalli. Massively parallel algorithms and hardness for single-linkage clustering under l_p-distances. *arXiv preprint arXiv:1710.01431*, 2017.

48 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, 2010. URL: `http://dl.acm.org/citation.cfm?id=1863103.1863113`.

A (poly log)-Expressible Problems

The goal of this section is to prove and give examples of Theorem 1. Let us first re-state the theorem.

Theorem 1. For any given m, there exists an algorithm to solve any (poly log)-expressible problem in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability each machine uses a space of size $\tilde{O}(n/m)$ and runs an algorithm that is linear in its input size.

Proof. Assume we are given a (poly log)-expressible problem P. We first convert the given tree into a binary extension of it using Lemma 26 in $O(1)$ rounds and then decompose the binary extension using Theorem 30 in $\tilde{O}(\log n)$ rounds. Note that since any component has at most two children (Claim 34), there are at most 2 leaves in any component that is unknown. Therefore we can use the compressor algorithm that is guaranteed to exist since P is assumed to be (poly log)-expressible and store a data of size at most poly $\log(n)$ for each component. Then in the next step, since there are at most $\tilde{O}(m)$ components, we can send the partial data of all components into one machine. Then in only one round, we are able to merge all this partial data since the merger algorithm (that is guaranteed to exist since P is (poly log)-expressible) takes polylogarithmic time and space to calculate the answer of the dynamic programming for each of the components and it suffices to start from the leave components and compute the value of the dynamic programming for each of them one by one (all of this is done in one step because we have the data of all the components in one machine). ◀

We first formally prove why Maximum Weighted Matching is a (poly log)-expressible problem and therefore as a corollary, it can be solved in $\tilde{O}(\log n)$ rounds.

Maximum Weighted Matching. An edge-weighted tree $\mathcal{T}$ is given in the input. (Let w_e denote the weight of an edge e.) The problem is to find a sub-graph M of $\mathcal{T}$ such that the degree of each vertex in M is at most one and the weight of M (which is $\sum_{e \in M} w_e$) is maximized.

At first, we explain the sequential DP that given any binary extension $\mathcal{T}^b$ (see Definition 24) of an input tree $\mathcal{T}$, finds a maximum matching of $\mathcal{T}$ and then prove this DP is (poly log)-expressible.

Define a subset M^b of the edges of $\mathcal{T}^b$ to be an *extended matching* if it has the following properties. 1. For any original vertex in $\mathcal{T}^b$, at most one of its edges is in M^b. 2. For any auxiliary vertex v of $\mathcal{T}^b$, at most one of the edges between v and its children is in M^b. 3. The edge between an auxiliary vertex v and its parent is in M^b if and only if an edge between v and one of its children is in M^b.

Consider an edge e between a vertex u and its parent v in $\mathcal{T}^b$. If u is an auxiliary vertex, we define w_e to be 0 and if v is an original vertex, we define w_e to be the weight of the edge between v' and its parent where v' is the vertex in $\mathcal{T}$ that is equivalent to v. Furthermore, we define the weight of an extended matching M^b of $\mathcal{T}^b$ to be $\sum_{e \in M^b} w_e$.

Let M^b be an extended matching of $\mathcal{T}^b$ and let M be a matching of $\mathcal{T}^b$; we say M^b and M are equivalent if for any edge $(u, v) \in M$, every edge between the equivalent vertices of v and u in $\mathcal{T}^b$ is in M^b and vice versa. Let $v', u' \in V(\mathcal{T}^b)$ denote the equivalent vertices of $v, u \in V(\mathcal{T})$ respectively. If v is the parent of u in $\mathcal{T}$, all the other vertices in the path between v' and u' are auxiliary by definition of binary extensions. This means all the edges in this path, except for the edge between u' and its parent (which is equal to $w_{u,v}$), have weight 0. Therefore the weight of any extended matching of $\mathcal{T}^b$ is equal to its equivalent matching of $\mathcal{T}$. Hence, to find the maximum weighted matching of $\mathcal{T}$, one can find the extended matching of $\mathcal{T}^b$ with the maximum weight.

Now, we give a sequential DP that finds the extended matching of $\mathcal{T}^b$ with the maximum weight. To do so, for any vertex v of $\mathcal{T}^b$, define $C(v)$ to be the maximum weight of an extended matching of the subtree of v where at least one edge of v to its children is part of the extended matching (if no such extended matching exists for v set $C(v)$ to be $-\infty$. Also define $C'(v)$ to be the maximum weight of an extended matching of the subtree of v where no edge of v to its children is part of the extended matching.

The key observation in updating $C(v)$ and $C'(v)$ for a vertex v is that if v is connected to one of its original children u, then u should not be connected to its children; if v is

To update $C(v)$ and $C'(v)$, if v has no children, then $C(v) = -\infty$ and $C'(v) = 0$, also if v has only one child u, then $C(v) = w_{(v,u)}$ and $C'(v) = 0$. W.l.o.g. we consider the following three cases for the case where v has two children u_1 and u_2.

- If u_1 and u_2 are both original vertices, then

$$C(v) = \max\{w_{v,u_1} + C'(u_1) + C(u_2), w_{v,u_2} + C'(u_2) + C(u_1)\},$$

$$C'(v) = \max\{C(u_1), C'(u_1)\} + \max\{C(u_2), C'(u_2)\}.$$

- If u_1 and u_2 are both auxiliary vertices, then

$$C(v) = \max\{w_{v,u_1} + C(u_1) + C'(u_2), w_{v,u_2} + C(u_2) + C'(u_1)\},$$

$$C'(v) = \max\{C'(u_1), C'(u_2)\}.$$

- If u_1 is auxiliary and u_2 is an original vertex, then

$$C(v) = \max\Big\{w_{v,u_1} + C(u_1) + \max\{C'(u_2), C(u_2)\}, w_{v,u_2}, +C'(u_2) + C'(u_1)\}\Big\},$$

$$C'(v) = C'(u_1) + \max\{C(u_2), C'(u_2)\}.$$

Now we prove that the (poly log)-expressiveness property holds for the proposed DP. Consider a subtree T of $\mathcal{T}^b$ and let u_1 and u_2 denote the leaves of T with the unknown DP values. We apply the proposed DP rules and calculate the DP values as functions of $C(u_1)$, $C'(u_2)$, $C(u_2)$, and $C'(u_2)$ instead of plain numbers. We claim that for any vertex v of T there exist functions $f_v, f'_v : \{0,1\}^4 \to \mathbb{Z}$ such that

$$C(v) = \max_{a_i \in \{0,1\}} \{a_0 C(u_1) + a_1 C'(u_1) + a_2 C(u_2) + a_3 C'(u_2) + f_v(a_0, a_1, a_2, a_3)\},$$

and

$$C'(v) = \max_{a_i \in \{0,1\}} \{a_0 C(u_1) + a_1 C'(u_1) + a_2 C(u_2) + a_3 C'(u_2) + f'_v(a_0, a_1, a_2, a_3)\}.$$

Therefore it suffices to store the values of functions f'_r and f_r for the root vertex r of T to be able to evaluate $C(v)$ and $C'(v)$. Since f_r and f'_r take only 16 different input combinations, there are only $O(1)$ different output values to be stored.

▶ **Corollary 9.** *For any given m, there exists an algorithm to solve the Maximum Weight Matching problem on trees in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability each machine uses a space of size $\tilde{O}(n/m)$ and runs an algorithm that is linear in its input size.*

Other problems. The dynamic programming solutions of Maximum Independent Set, Minimum Vertex Cover, Longest Path and many other problems are similar to the proposed DP for Maximum Weighted Matching. For example, for the Maximum Independent Set problem, we only need to keep two dynamic data, whether the root vertex of a subtree is part of the independent set in its subtree or not and for each of these cases what is the size of the maximum independent set in the subtree. Roughly speaking, the only major difference in the DP is that we have to change max to sum.

▶ **Corollary 10.** *For any given m, there exists algorithms to solve the Maximum Independent Set, Minimum Vertex Cover, Longest Path and Dominating Set problems on trees in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability each machine uses a space of size $\tilde{O}(n/m)$ and runs an algorithm that is linear in its input size.*

B Linear-Expressible Problems

We continue the discussion about linear-expressible problems by giving a detailed proof that the minimum bisection problem can be solved in logarithmic rounds.

B.1 Minimum Bisection

▶ **Theorem 11.** *There exists an algorithm to solve bisection in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability each machine uses $\tilde{O}(n^{4/3}/\mathcal{M})$ space, and takes $O(n^2/m)$ time in each round of the algorithm.*

Proof. While converting $\mathcal{T}$ to binary version of it, for any auxiliary vertex a we set the weight of the edge between a and its parent to be ∞. Note that in any optimal solution for the new tree, the infinity weights enforce all auxiliary vertices to be in the same bisection set that their parents are, thus any solution for the original tree will have the same cost for the binary tree as long as we make sure the auxiliary vertices are in the same set that their parents are (otherwise the cost will be ∞). Recall that in the bisection's solution the number of vertices of the two sets should be equal, hence we have to keep track of the auxiliary vertices that we added to the tree and make sure we do not count them.

The dynamic data $D_c(T_i)[b]$, for any partition (sub-tree) T_i, and any possible coloring c of its border vertices into red and blue sets, and any number $b \leq |V(T_i)|$, is the minimum cost of partitioning T_i such that the partitions match what is specified in c and there are exactly b non-auxiliary vertices in the blue set. (Recall that the color of auxiliary vertices will have to be the same as their parents or the cost would be ∞). Note that we fix the exact coloring of border vertices by c to make it possible to merge the partial dynamic data for two components. Two components are only connected through their border vertices and if we know how the border vertices are colored, we will know if in a solution we have to add the weight of the edges between border vertices to the cost or not. Note that there are at most 4 border vertices in any partition so there are constant number of possible cases for c. This implies that bisection is linear-expressible.

If we define the unifier ($\mathcal{U}$) and the sub unifier ($\mathcal{S}$) functions for the dynamic data stored for each partition we obtain that bisection is also splittable (Definition 6). Let T be the result of merging T_i and T_j, and let set $D(T')[b]$ denote the set of all $D_c(T')[b]$ for any possible coloring c of border vertices in $V(T')$, and any partition T'. The function $\mathcal{S}$ for any consecutive portion of $D(T_i)$ from element $D(T_i)[a_i]$ to $D_c(T_i)[b_i]$ denoted by $D(T_i)[a_i : b_i]$, and any consecutive portion of $D(T_j)$ denoted by $D(T_j)[a_j : b_j]$ is as follows: for any pair k_i and k_j such that $a_i \leq k_i \leq b_i$, $a_j \leq k_j \leq b_j$, the value of pair $(D(T_i)[k_i], D(T_j)[k_j])$ is only related to $D(T)[k_i + k_j]$. Therefore, for any possible coloring c of border vertices of T, and any number x such that $a_i + a_j \leq x \leq b_i + b_j$, $\mathcal{S}$ outputs a value for $D_c(T)[x]$ which is the maximum value for $D_c(T)[x]$ based on the input given to this function. It is easy to see that $\mathcal{U}$ for any c is the maximum of all its inputs that match with the color of border vertices c, which is a function polynomial to its input.

At the end, $\max_c D_c(\mathcal{T}^d)[\frac{n'}{2}]$, where n' is the number of non-auxiliary vertices of $\mathcal{T}$, is the solution we return.

Therefore, by Theorem 2 since bisection is linear- and splittable, there exists an algorithm to solve bisection in $\tilde{O}(\log n)$ round of $\mathcal{MPC}$ using m machines, such that with high probability each machine uses $\tilde{O}(n^{1+\epsilon}/\mathcal{M})$ space, and takes $O(n^2/m)$ time in each round of the algorithm, for $\epsilon = \frac{1}{3}$. ◀

Extensions. It could be shown that with very similar techniques, the "k-maximum spanning tree" problem on a tree could also be solved. In this problem a weighted tree is given and the goal is to find the maximum weighted connected subgraph of the given tree with exactly k vertices. When we convert this tree to a binary tree we only need to make sure that we mark auxiliary vertices such that we do not disconnect them from their parents in the algorithm. Moreover, instead of considering all the possible coloring of the border vertices in bisection, in the dynamic for k-maximum spanning tree, we need to consider all the cases for the connectivity of the border vertices with the maximum weighted tree in the partition.

Furthermore, we expect that this idea can be extended to several other problems, including facility location problem, minimum diameter, k partitioning, k-median, and k-center, because they have very similar dynamic programming solutions when the input graph is a tree. We refer the detailed discussion to the full version of the paper.

C Further Applications

C.1 Minimum Spanning Tree on Sparse Graphs

Finding the minimum spanning tree is a fundamental problem. In fact, it is one of the main problems that Karloff et al. [31] focus on, when introducing the initial theoretical model of MapReduce. In particular, they give a constant round algorithm to find the MST of dense graphs. Moreover finding MST in metric spaces has been widely studied by [47, 6]. The total memory that they use is linear to the input size, however they give approximation algorithms for finding MST in limited metric spaces with limited dimensions. As it was noted before, obtaining constant round algorithms for general graphs is very unlikely.

In this section, we show how our framework could be used in designing algorithms for finding MST in general graphs. We first give an algorithm for finding exact MST in metric spaces, with arbitrary number of dimensions. We also give an $\tilde{O}(\log^2 n)$ round MST algorithm for sparse graphs. We would like to note that, there already exist efficient PRAM algorithms for minimum spanning tree problem [32] in sparse graphs. Coupling them with the simulation methods of [31, 26] one can obtain distributed algorithms with almost the same round complexity. However, our main purpose in this section, is to show how our framework could be applied to sparse graphs.

▶ **Theorem 12.** *Let V denote a set of vectors in d dimensional space and let f denote a function that determines the distance between any two vectors u and v where $f(u,v)$ is computable in $O(d)$ space. Given any m where $m \leq n^{2/3}$, there exists an algorithm to find MST of V in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability, each machine uses a space of size at most $\tilde{O}(d \times n^{4/3}/m)$.*

Proof. Let G denote the metric graph of vector set V. any vector in V represents a vertex in G, and the weight of an edge between any two vertex u and v is $f(u, v)$. We partition the vertices into $\sqrt{m}$ partitions, each of which containing $n/\sqrt{m}$ vertices. Let $p_1, \ldots, p_{\sqrt{m}}$ denote the partitions. For any i and j, where $1 \leq i < j \leq \sqrt{m}$, we put all the vertices in p_i and p_j in one machine. Note that each partition is in exactly $\sqrt{m} - 1$ machines. Let $M_{i,j}$ denote the machine that contains p_i and p_j. In this machine we find the MST between vertices in $p_i \cup p_j$, and we denote it by $T_{i,j}$. By Lemma 13 any edge e that is in MST of G is also in the MST between the vertices of at least one of the machines. The size of the complete graph between the vertices in one machine is $\mathrm{O}((n^{4/3}/m)^2)$ so it does not fit in a single machine. To overcome this, in each machine we generate edges sequentially. We update the MST after generating any edge, and remove any edge that is not in the MST.

In the first round, the algorithm in any machine $M_{i,j}$ is as follows: at the beginning $T_{i,j}$ is empty. For any pair of vertices u and v in $p_i \cup p_j$, we compute $f(u, v)$ in $O(d)$ space which is the wight of the edge $e_{i,j}$. We then add $e_{u,v}$ to $T_{i,j}$. If there was a loop in $T_{i,j}$ after adding this edge, we update $T_{i,j}$ by removing the edge with the maximum weight in this loop. Then we remove any edge that is not in $T_{i,j}$. To run this algorithm in one machine we do not need space more than $\mathrm{O}(d \times n^{4/3}/m)$, since we just store the edges of the MST. The size of the MST in each machine is at most equal to the size of the vertices in that machine which is $\mathrm{O}(d \times n^{4/3}/m)$.

After the first round of the algorithm there are $\mathrm{O}(n^{4/3}/m)$ edges in each machine, which is $\mathrm{O}(n^{4/3})$ in total. So, the problem of finding the MST of G is now reduced to finding an MST in a graph that has n vertices and $\mathrm{O}(n^{4/3})$ edges. Since there exists an algorithm that finds the MST of such a graph in $\tilde{\mathrm{O}}(\log n^{4/3}) = \tilde{\mathrm{O}}(\log n)$ rounds using m machines and $\tilde{\mathrm{O}}(n^{4/3}/m)$ space per machine, the number of rounds that we need to find the MST of G is $\tilde{\mathrm{O}}(\log n)$ as well. ◀

▶ **Lemma 13.** *[11] Let M denote the MST of $H(V', E')$ which is a subgraph of graph $G = (V, E)$. If $e \in E'$ and $e \notin M$, then e is not in the MST of G.*

Next, we give give an $\tilde{\mathrm{O}}(\log^2 n)$ round MST algorithm for sparse graphs to show how our algorithm works on sparse graphs.

▶ **Theorem 14.** *For any given m, there exists an algorithm to find the minimum spanning tree of a given graph in $\tilde{\mathrm{O}}(\log^2 n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability each machine uses a space of size $\tilde{\mathrm{O}}(n/m)$ and runs an algorithm that is linear in its input size.*

Proof sketch. We first explain the Borůvka's famous MST algorithm, then show how to implement it as a distributed algorithm using our framework. Borůvka's algorithm initially starts with n *components*, where each component contains one of the vertices of the graph. Defining the *distance* between two components to be the minimum edge between them, Borůvka's algorithm in each step merges each component to its closest component simultaneously. It could be shown that the size of smallest component is doubled in each step, hence roughly after $\mathrm{O}(\log n)$ rounds, all components are merged. The edges through which we merge these components will be the MST of the input graph.

We call each round of Borůvka a super-round. Then show how each of the super-rounds could be implemented in $\tilde{\mathrm{O}}(\log n)$ rounds of $\mathcal{MPC}$. Combining them together, we achieve a $\tilde{\mathrm{O}}(\log^2 n)$ round algorithm for MST.

Let us focus on a super round. Consider a graph that maps each component to a vertex and contains a directed edge from vertex v to u if the corresponding component of vertex u

is the closest component to the corresponding component of v. Assuming that no two edges of the initial graph have the same weight (which could be achieved by simply perturbing all the edge weights), the structure of this graph would be as follows. Each vertex has exactly one outgoing edge and the maximum length of a directed cycle is 2. To see this, note that if there is a cycle of size more than 2, there has to be at least two edges with the same weight. A cycle of length two indicates two components that are each other's nearest neighbors. By contracting cycles of length two, we obtain a collection of trees. In fact, each cycle of length two becomes the root of a tree. At the end of the super-round, the Borůvka's algorithm merges all the components that belong to the same tree together.

The idea is to use our framework to efficiently merge these trees. While merging the trees, we also have to update the distances between the contracted trees for the next super round. To do this, in the first step we make sure that all the vertices that belong to the same tree *know* each other by assigning the same *color* (e.g., the id of the root) to them. Note that the diameter of these trees might be arbitrarily large. Hence we need to incorporate our framework to handle this in $\tilde{O}(\log n)$ rounds. After all the vertices are assigned the color of the tree that they belong to, we find for each tree its minimum edge to other trees. Assume there are k different trees. Using a $(\text{poly}\log)$-expressible bottom-up dynamic program, for each node of a tree, we keep a list of $k-1$ different edges, where each one denotes the minimum weight edge to another tree. After this list is completely merged, we can contract the trees and remove the edges that are no longer required.

Each super step takes $\tilde{O}(\log n)$ rounds. Hence the total running time of the algorithm is $\tilde{O}(\log^2 n)$. ◀

C.2 Further Applications of the Splitting Technique

Assume that a set V of vectors in a d-dimensional space is given in the input and a distance function $f : V \times V \to \mathbb{R}_+$ determines the distances between the input vectors. In the closest points problem, our goal is to find the pair of vectors that are closest to each other. Moreover, in the MST in metric spaces, our goal is to find the minimum-weight tree that connects all these vectors. Our goal is to come up with exact (i.e., not approximate) solutions for these problems.

Before explaining the actual algorithm, we need to mention that in the literature, with certain assumptions on the distance function or the dimension, efficient $\mathcal{MPC}$ algorithms are known for finding the MST in metric spaces [6, 47]. However, our goal here is to design a general algorithm without any assumption on the dimension or the distance function. There are two ways to consider general distance functions: (1) the value of f for any two vectors is specified in the input; (2) each machine has access to an oracle that can compute the distance between any two given vectors. The former case increases the input size quadratically which simplifies the solution. For instance, finding MST in metric spaces becomes the same as finding the MST in dense (or in this case, complete) graphs for which efficient constant round $\mathcal{MPC}$ algorithms are known [31, 34, 11]. Our focus, therefore, is on the latter case.

The difficulty with the oracle-based model is that the input size is of size $d \cdot |V|$ and thus we cannot fit all the vectors in one machine if we want the memory to be sublinear in the input size. On the other hand, all possible pairs of vectors have to be in the same machine at some point during the algorithm, or otherwise, we would not be able to determine the distance between them.

Let $n := d \cdot |V|$ denote the input size. To focus on the general idea, assume at first, that the number of machines m, is $\lceil n^{2/3} \rceil$ (instead of being given in the input) and that each machine has space $s = \tilde{O}(n^{2/3})$. We first randomly *partition* the vectors into $k = \lceil n^{1/3} \rceil$

groups $\{V_1, V_2, \dots, V_k\}$. As an application of Chernoff bound, the size of groups is bounded by $\tilde{O}(n^{2/3})$ with high probability. In the next step, for any $i \in [k]$ and $j \in [k]$ we put V_i and V_j in one machine. Note that the number of machines, $m = \lceil n^{2/3} \rceil$, is more than $\binom{k}{2}$ and thus we have enough machines to put each group pair in one machine.

To solve the closest points problem, in the second round (after distributing the group pairs into machines) each machine computes the minimum distance between the points that are stored in it and all machines send this minimum distance to a single machine μ, then in the third round, machine μ reports the overall minimum distance. Let us denote by $S := s \cdot m$ the overall space on all machines. For the mentioned algorithm, S is $n^{4/3}$. Also, by setting $k = \sqrt{m}$, we can get the same guarantee on S as long as $m \leq n^{2/3}$. This way, the space of each machine would be of size $\tilde{O}(n/\sqrt{m})$ and therefore the overall space that the algorithm uses would be of size $\tilde{O}(n\sqrt{m}) = \tilde{O}(n^{4/3})$.

▶ Remark. For any given m where $m \leq n^{2/3}$, there exists an algorithm to solve the oracle-based closest points problem in 3 rounds of $\mathcal{MPC}$ using m machines, such that with high probability, each machine uses a space of size at most $\tilde{O}(n^{4/3}/m)$.

But can we get any better bounds on S? Obviously if there are very few machines with large memories we can improve this bound. However, the goal in distributed settings is to parallelize the work-load into as many machines as possible. More precisely, for a given overall space S, we want to able to have as many as $m = \lfloor \sqrt{S} \rfloor$ machines, each with memory $s \leq S/m = O(\sqrt{S})$. (Recall that in $\mathcal{MPC}$ we assume $m \leq s$, therefore $m^2 \leq s \cdot m = S$ and thus $m \leq \sqrt{S}$. Meaning that $m = \lfloor \sqrt{S} \rfloor$ is the maximum number of machines one can hope for.) With this natural assumption, we get the following result implying that the $n^{4/3}$ overall space is in some sense tight.

▶ **Lemma 15.** *If $m = \lfloor \sqrt{S} \rfloor$, any algorithm in $\mathcal{MPC}$ that solves the oracle-based closest points problem requires at least $\Omega(n^{4/3})$ overall space.*

We defer the proof of the following theorem about finding MST in metric spaces to the forthcoming sections. However, using the "splitting" idea, we can first *sparsify* the complete distance graph such that we can randomly distribute its edges in the machines and then solve the MST problem on a sparse graph.

Theorem 12. (restated) Let V denote a set of vectors in d dimensional space and let f denote a function that determines the distance between any two vectors u and v where $f(u, v)$ is computable in $O(d)$ space. Given any m where $m \leq n^{2/3}$, there exists an algorithm to find MST of V in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability, each machine uses a space of size at most $\tilde{O}(d \cdot n^{4/3}/m)$.

D Tree Decomposition on $\mathcal{MPC}$

The goal of this section is to give a distributed algorithm to decompose a given tree $\mathcal{T}$ with n vertices into $O(m)$ "components" of size at most $\tilde{O}(n/m)$, such that each component has at most a constant number of "outer edges" to the other components. A very similar problem called tree contraction has been widely studied in the PRAM setting. There is a series of works on this problem [40] [1] [24]. There are very nice ideas in these papers. However, due to the restrictions of the PRAM model, the algorithms presented for tree contraction are very complicated. In this section we present a much simpler approach for decomposing a tree.

Overview. To motivate the need to design an involved decomposition algorithm, we first give hints on why trivial approaches do not work. A possible (trivial) algorithm is to choose a set of randomly selected *joint* edges (or vertices) and divide the tree based on them. Note that the main difficulty is that we need to bound the size of the largest component with a high probability. It could be shown that this trivial algorithm, even for the simple case of full binary trees fails. More precisely, it could be shown that the size of the component that contains the root is of size $\Omega(n)$ (instead of $\tilde{O}(n/m)$) with a high probability. In general, random selection does not perform well in partitioning the tree.

In contrast, our algorithm is as follows. At the very first, we convert the tree into a binary tree. To do this, we first have to compute the degree of each vertex and then add *auxiliary* vertices to the graph. After that, our algorithm proceeds in *iterations* (not rounds). In each iteration, we merge the vertices of the tree based on some local rules and after $\tilde{O}(\log n)$ iterations, we achieve the desired decomposition. More precisely, in each iteration, we *select* a set of vertices based on a set of local rules (e.g., the degree of a vertex, status of its parent, etc.) and *merge* each vertex to its closest selected ancestor. We prove in fact these local rules guarantee that the size of the maximum component is bounded by $\tilde{O}(n/m)$ with high probability.

The input, as mentioned before, is a rooted tree $\mathcal{T}$, with n vertices numbered from 1 to n (we call these numbers the indexes of the vertices).[4] Each machine initially receives a subset of size $\tilde{O}(n/m)$ of the vertices of $\mathcal{T}$ where each vertex object v contains the index of v, denoted by v.`index`, and the index of its parent, denoted by v.`parent`.

In Section D.1 we show how universal hash functions allow us to efficiently distribute and access objects in our model. In Section D.2 we introduce an algorithm to convert the given tree into a "binary-extension" of it. Finally, in Section D.3 we provide an algorithm to decompose the binary extension.

D.1 Load Balancing via Universal Hash Families

We start with the definition of universal hash families, that was first proposed by [45].

▶ **Definition 16.** A family of hash functions $H = \{h : A \to B\}$ is k-universal if for any hash function h that is chosen uniformly at random from H and for any k distinct keys $(a_1, a_2, \dots, a_k) \in A^k$, random variables $h(a_1), h(a_2), \dots, h(a_k)$ are independent and uniformly distributed in B. A k-universal hash function is a hash function that is chosen uniformly at random from a k-universal hash family.

The following lemma shows that we can generate a $(\log m)$-universal hash function in constant rounds and store it on all the machines. Roughly speaking, the idea is to generate a small set of random coefficients on one machine and share it with all other machines and then use these coefficients to evaluate the hash function for any given input.

▶ **Lemma 17.** *For any given $k \in \mathbb{Z}^+$ where $k \le \mathrm{poly}(n)$, there exists an algorithm that runs in $O(1)$ rounds of $\mathcal{MPC}$ and generates the same hash function $h : [1, k] \to \mathcal{M}$ on all machines where function h is chosen uniformly at random from a $(\log m)$-universal hash family, and calculating $h(.)$ in a machine takes $O(\log n)$ time and space.*

We mainly use the universal hash functions to distribute objects into machines. Assuming that an object a has an integer index denoted by a.`index`, we store object a in machine $h(a.\texttt{index})$ where h is the universal hash function stored on all machines. This also allows other machines to know in which machine an object with a given index is stored.

[4] The assumption that the vertices are numbered from 1 to n is just for the simplicity of presentation. Our algorithms can be adopted to any arbitrary naming of the vertices.

We claim even if the objects to be distributed by a $(\log m)$-universal hash function have different sizes (in terms of the memory that they take to be stored), the maximum load on a machine will not be too much with high probability. To that end, we give the following definitions and then formally state the claim.

▶ **Definition 18.** A set $A = \{a_1, a_2, \dots, a_k\}$ is a weighted set if each element $a_i \in A$ has an associated non-negative integer weight denoted by $w(a_i)$. For any subset B of A, we extend the notion of w such that $w(B) = \sum_{a \in B} w(a)$.

▶ **Definition 19.** A weighted set $A = \{a_1, a_2, \dots, a_k\}$ is a *distributable* set if $w(A)$ is $\mathrm{O}(n)$ and the maximum weight among its elements (i.e., $\max_{a_i \in A} w(a_i)$) is $\tilde{\mathrm{O}}(n/m)$.

▶ **Definition 20.** We call a hash function $h : A \to \mathcal{M}$ a *distributer* if A is a weighted set and $\mathcal{M}$ is the set of all machines. We define the *load* of h for a machine m to be $l_h(m) = \sum_{a \in A : h(a) = m} w(a)$. Moreover we define the maximum load of h to be $\max_{m \in \mathcal{M}} l_h(m)$.

▶ **Lemma 21.** *Let $h : A \to \mathcal{M}$ be a hash function chosen uniformly at random from a $(\log m)$-universal hash function where A is a distributable set and $\mathcal{M}$ is the set of machines (i.e., h is a distributer). The maximum load of h is $\tilde{\mathrm{O}}(n/m)$ with probability at least $1 - (\frac{2^{1/\delta} e}{\delta \log n})^{\delta \log n}$ where $m \geq n^{\delta}$.*

The full proof of Lemma 21 is given in the proofs section. The general idea is to first partition the elements of A into $\mathrm{O}(n/m)$ subsets $S_1, S_2, \dots, S_k$ of size $\mathrm{O}(m)$ such that the elements are grouped based on their weights (i.e., S_1 contains the m elements of A with the lowest weights, S_2 contains the m elements in $A - S_1$ with the lowest weights, and so on). Then we show that the maximum load of the objects in any set S_i is $\tilde{\mathrm{O}}(w_i)$ where w_i is the weight of the object in S_i with the maximum weight. Finally using the fact that the objects are grouped based on their weights, we show that $\sum_{i \in [k]} w_i$, which is the total load is $\tilde{\mathrm{O}}(n/m)$.

A technique that we use in multiple parts of the algorithm is to distribute the vertices of $\mathcal{T}$ among the machines using a $(\log m)$-universal hash function.

▶ **Definition 22.** Let T be a given tree and let $h : \{1, \dots, |V(T)|\} \to \mathcal{M}$ be a mapping of the vertex indexes of T to the machines. We say T is distributed by h if any vertex v of T is stored in machine $h(v.\texttt{index})$.

As a corollary of Lemma 17:

▶ **Corollary 23.** *One can distribute a tree T by a hash function h that is chosen uniformly at random from a $(\log m)$-universal hash family in $\mathrm{O}(1)$ rounds in such a way that each machine can evaluate h in $\mathrm{O}(\log n)$ time and space.*

D.2 Conversion of $\mathcal{T}$ to a Binary Tree

The first step towards finding a decomposition of $\mathcal{T}$ with the desired properties is to convert it into a binary tree $\mathcal{T}^b$ that preserves the important characteristics of the original tree (e.g., the ancestors of each vertex must remain to be its ancestors in the binary tree too).

The definition of an *extension* of a tree is as follows. By the end of this section, we prove it is possible to find a binary extension of $\mathcal{T}$ in constant rounds.

▶ **Definition 24.** A rooted tree $\mathcal{T}'$ is an extension of a given rooted tree $\mathcal{T}$ if $|V(\mathcal{T}')| = \mathrm{O}(|V(\mathcal{T})|)$ and there exists a mapping function $f : V(\mathcal{T}) \to V(\mathcal{T}')$ such that for any $v \in V(\mathcal{T})$, $v.\texttt{index} = f(v).\texttt{index}$ and if u is an ancestor of v in $\mathcal{T}$, $f(u)$ is also an ancestor of $f(v)$ in $\mathcal{T}'$.

The following lemma proves it is possible to find the degree of all vertices in constant rounds.

▶ **Lemma 25.** *There exists a randomized algorithm to find the degree of each vertex of a given rooted tree $\mathcal{T}$ in $O(1)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability, each machine uses a space of size at most $\tilde{O}(n/m)$ and runs an algorithm that is linear in its memory size.*

Proof. The first step is to distribute the vertices of $\mathcal{T}$ using a $(\log m)$-universal hash function h. By Corollary 23 this takes $O(1)$ rounds. Define the local degree of a vertex v on a machine μ to be the the number of children of v that are stored in μ and denote it by $d_{\mu,v}$. We know there are at most $\tilde{O}(n/m)$ vertices with a non-zero local degree in each machine. Hence in one round, every machine can calculate and store the local degree of every such vertex. In the communication phase, every machine μ, for any vertex v such that $d_{\mu,v} > 0$, sends $d_{\mu,v}$ to machine $h(v)$. Then in the next round, for any vertex v, machine $h(v)$ will receive the local degree of v on all other machines and can calculate its total degree.

We claim with high probability, no machine receives more than $\tilde{O}(n/m)$ data after the communication phase.

Define the weight of a vertex v, denoted by $w(v)$, to be $\min\{m, \deg(v)\}$. The weight of a vertex v is an upper bound on the size of communication that machine $h(v)$ receives for vertex v. To see this, observe that a machine sends the local degree of v to $h(v)$ only if this local degree is non-zero, therefore the total communication size for v cannot exceed its degree. On the other hand, the total number of machines is m, hence machine $h(v)$ cannot receive more than m different local degrees for vertex v.

We first prove that $V(\mathcal{T})$ is a distributable set (Definition 19) based on weight function w. To see this, note that first, by definition of w, the maximum value that w gets is m, which indeed is $\tilde{O}(n/m)$ (recall that in the model we assumed m is less than the space on each machine and hence $m = \tilde{O}(n/m)$), and second, the total weight of all vertices is $O(n)$ since $\sum_{v \in V(\mathcal{T})} \deg(v) = 2n - 2$.

By Lemma 21, since $V(\mathcal{T})$ is a distributable set based on w and since h is chosen from a $(\log m)$-independent hash family, the maximum load of h, i.e., the maximum communication size of any machine is $\tilde{O}(n/m)$ with high probability. ◀

We are now ready to give a binary extension of $\mathcal{T}$.

▶ **Lemma 26.** *There exists a randomized algorithm to convert a given rooted tree $\mathcal{T}$ to a binary tree $\mathcal{T}^b$, an extension of $\mathcal{T}$, in $O(1)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability, each machine uses a memory of size at most $\tilde{O}(n/m)$ and runs an algorithm that is linear in its memory size.*

Proof. Our algorithm for converting $\mathcal{T}$ to a binary extension of it done in two phases. In the first phase, we convert $\mathcal{T}$ to an extension of it, $\mathcal{T}^d$, with a maximum degree of $\tilde{O}(n/m)$. Then in the next phase, we convert $\mathcal{T}^d$ to a binary extension.

▶ **Claim 27.** *There exists a randomized algorithm to convert a given rooted tree $\mathcal{T}$ to a tree $\mathcal{T}^d$, with maximum degree $\tilde{O}(n/m)$ that is an extension of $\mathcal{T}$ in $O(1)$ rounds of $\mathcal{MPC}$ such that with high probability, each machine uses a memory of size at most $\tilde{O}(n/m)$ and runs an algorithm that is linear in its memory size.*

The detailed proof of Claim 27 and the pseudo-code to implement it is given in the proofs section. Intuitively, after calculating the degree of each vertex, we send the index of all vertices with degree more than $O(n/m)$ to all machines (call them high degree vertices).

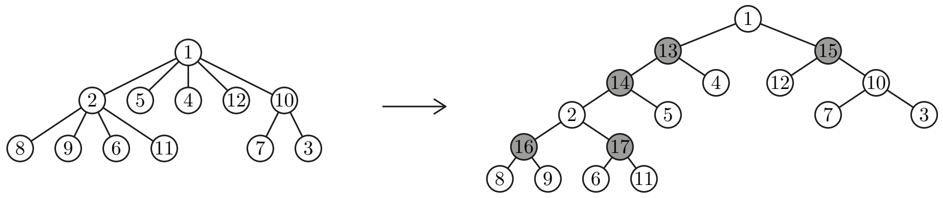

Figure 1 An example of converting a tree to a binary extension of it. The gray nodes denote the added auxiliary vertices and the numbers within the nodes denote their indexes.

This is possible because there are at most $\mathrm{O}(m)$ high degree vertices and m is assumed to be less than the space of each machine. To any high degree vertex v, in the next step, we add a set of at most $\mathrm{O}(n/m)$ auxiliary children and set the parent of any previous child of v to be one of these auxiliary vertices that is chosen uniformly at random. The detailed proof of why no machine violates its memory size during the process and how we assign indexes to the auxiliary vertices is explained in the proofs section.

The next phase is to convert this bounded degree tree to a binary tree.

▶ **Claim 28.** *There exists a randomized algorithm to convert a tree $\mathcal{T}^d$, with maximum degree $\tilde{\mathrm{O}}(n/m)$, to a binary extension of it $\mathcal{T}^b$, in $\mathrm{O}(1)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability, each machine uses a memory of size at most $\tilde{\mathrm{O}}(n/m)$ and runs an algorithm that is linear in its memory size.*

Again, the detailed proof of Claim 28 and a pseudo-code for implementing it in the desired setting is given in the proofs section. Roughly speaking, since the degree of each vertex is at most $\tilde{\mathrm{O}}(n/m)$, we can store all the children of any vertex in the same machine (the parent may be stored in another machine). Having all the children of a vertex in the same machine allows us to add auxiliary vertices and locally reduce the number of children of each vertex to at most 2. Figure 1 illustrates how we convert the tree to its binary extension, the details of how exactly the algorithm works is described in the proofs section..

By Claim 27 and Claim 28, for any given tree, there exists an algorithm to construct a binary extension of it in $\mathrm{O}(1)$ round of $\mathcal{MPC}$, which with high probability uses $\tilde{\mathrm{O}}(n/m)$ time and space. ◀

D.3 Decomposing a Binary Tree

We start with the formal definition of decompositions.

▶ **Definition 29.** We define a decomposition of a binary tree $\mathcal{T}^b$ to be a set of $\mathrm{O}(m)$ components, where each component contains a subset of the vertices of $\mathcal{T}^b$ that are connected, and each vertex of $\mathcal{T}^b$ is in exactly one component. A component c_i, in addition to a subset of $V(\mathcal{T}^b)$ (which we denote by $V(c_i)$) and their "inner edges", i.e., the edges between the vertices in $V(c_i)$, contains their "outer edges" (i.e., the edges between a vertex in $V(c_i)$ and a vertex in another component) too. The data stored in each component, including the vertices and their inner and outer edges should be of size up to $\tilde{\mathrm{O}}(n/m)$.

Since the vertices in any component c_i are connected, there is exactly one vertex in c_i that its parent is not in c_i, we call this vertex the root vertex of c_i and denote it by c_i.`root`. We also define the index of a component c_i to be equal to the index of its root; i.e., c_i.`index` $=$ c_i.`root.index` (recall that we assume every vertex of $\mathcal{T}^b$ has a unique index). Moreover, by contracting a component we mean contracting all of its vertices and only keeping their outer edges to other components. Since the vertices in the same component have to be connected by definition, the result of contraction will be a rooted tree. We may use this fact and refer to the components as nodes and even say a component c_i is the parent (child, resp.) of c_j if after contraction, c_i is indeed the parent (child, resp.) of c_j.

▶ **Theorem 30.** *There exists a randomized algorithm to find a decomposition of a given binary tree $\mathcal{T}^b$ with n vertices using m machines, such that with high probability the algorithm terminates in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$, and each machine uses a memory of size at most $\tilde{O}(n/m)$ and runs an algorithm that is linear in its memory size.*

Algorithm 2 gives a sequential view of the decomposition algorithm we use. We first prove some properties of the algorithm, and then using these properties, we prove it can actually be implemented in the desired setting.

Algorithm 2 The sequential algorithm for finding a valid $\mathcal{MPC}$ decomposition of a given binary tree $\mathcal{T}^b$.

```
Require: A binary tree $\mathcal{T}^b$.
 1: $C \leftarrow \emptyset$                              ▷ $C$ will contain all components.
 2: for each vertex $v$ in $\mathcal{T}^b$ do
 3:     Insert a component containing only $v$ to $C$.
 4: $F \leftarrow \emptyset$                              ▷ $F$ will contain the completed components.
 5: while $|C| > 14m$ do
 6:     $S \leftarrow \emptyset$                          ▷ $S$ will contain the selected components.
 7:     for each component $c \in C - F$ do
 8:         if $c$ is the root component then
 9:             Add $c$ to $S$.
10:         else if $c$ has exactly two children components then
11:             Add $c$ to $S$.
12:         else if the parent component of $c$ is in $F$ then
13:             Add $c$ to $S$.
14:         else if $c$ has exactly one child component then
15:             Add $c$ to $S$ with probability 0.5.
16:     for each component $c \in C - F - S$ do
17:         $d \leftarrow$ Find the closest ancestor of $c$ that is in $S$.
18:         Merge $c$ into $d$.
19:     for every node $c$ in $C - F$ do
20:         if $c$ contains at least $n/m$ vertices of $\mathcal{T}^b$ then
21:             Add $c$ to $F$.
```

The algorithm starts with n components where each component contains only one vertex of $\mathcal{T}^b$. Then it merges them in several iterations until there are only $O(m)$ components left. At the end of each iteration, if the size of a component is more than n/m, we mark it as a completed component and never merge it with any other component. This enables us to prove that the components' size does not exceed $\tilde{O}(n/m)$. In the merging process, at each iteration, we "select" some of the components and merge any unselected component to its closest selected ancestor. We select a component if at least one of the following conditions hold for it. 1. It is the root component. 2. It has exactly two children. 3. Its parent component is marked as complete. In addition to these conditions, if a component has exactly one child, we select it with an independent probability of 0.5. Figure 2 illustrates the iterations of Algorithm 2.

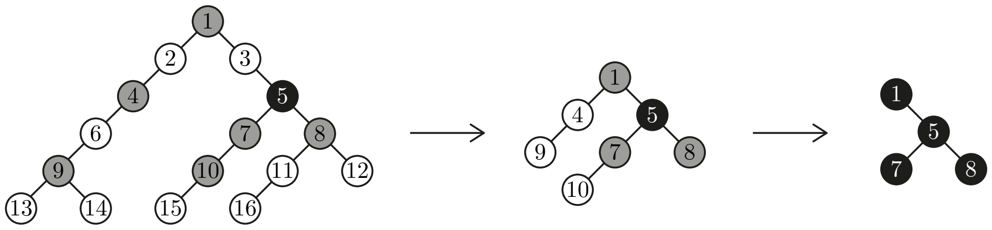

Figure 2 An example of running Algorithm 2. Each node denotes a component. A gray node denotes a selected component and a black node denotes a completed component.

The intuition behind the selecting conditions is as follows. The first condition ensures every unselected component has at least one selected ancestor to be merged to. The second condition ensures no component will have more than two children after the merging step. (To see this, let c_1 be a component with two children, c_2 and c_3. Also let c_3 have two children, c_4 and c_5. If only c_3 is not selected and is merged to c_1, then the resulting component will have three children c_2, c_4 and c_5.) The third condition ensures the path of a component to its closest selected ancestor does not contain any completed component. This condition is important to keep the components connected. Finally, randomly selecting some of the components with exactly one child ensures the size of no component exceeds $\tilde{O}(n/m)$. (Otherwise, a long chain of components may be merged to one component in one iteration.) We also prove at most a constant fraction of the components are selected at each iteration. This proves the total number of iterations, which directly impacts the number of rounds in the parallel implementation, is logarithmic in the number of vertices.

D.3.1 Properties of Algorithm 2

In this section, we prove some properties of Algorithm 2. These properties will be useful in designing and analyzing the parallel implementation of the algorithm.

Many of these properties are defined on the iterations of the while loop at line 5 of Algorithm 2. Any time we say an iteration of Algorithm 2, we refer to an iteration of this while loop. We start with the following definition.

▶ **Definition 31.** We denote the total number of iterations of of Algorithm 2 by I. Moreover, for any $i \in [I]$, we use C_i and F_i to respectively denote the value of variables C (which contains all components) and F (which contains the completed components) at the start of the i-th iteration. Analogously, we use S_i to denote the selected components (S) at the end of the i-th iteration. We also define T_i to be the rooted component tree that is given by contracting all components in C_i (e.g., $T_1 = \mathcal{T}^b$).

▶ **Claim 32.** *For any $i \in [I-1]$, each component $c \in C_{i+1}$ is obtained by merging a connected subset of the components in C_i.*

Proof. Observe that if in a step a component v is the closest selected ancestor of component u, it is also the closest selected ancestor of every other component in the path from u to v (denote the set of these components by P_u). We claim if u is merged to v, every component in P_u will also be merged to v in the same step. To see this, note that if there exists a component in P_u that is in F_i or S_i, the closest selected ancestor of u would not be v. Hence all of these components will be merged to v and the resulting component will be obtained by merging a connected subset of the components in C_i. ◀

The following is a rather general fact that is useful in proving some of the properties of Algorithm 2.

▶ **Fact 33.** *For a given binary tree T, let $k_i(T)$ denote the number of vertices of T with exactly i children. Then $k_0(T) = k_2(T) + 1$.*

We claim the components' tree, in all iterations of the while loop, is a binary tree.

▶ **Claim 34.** *For any $i \in [I]$, the component tree T_i is a binary tree.*

Proof. We use induction to prove this property. We know by Definition 31, that $T_1 = \mathcal{T}^b$ and since $\mathcal{T}^b$ is a binary tree, the condition holds for the base case T_1.

Assuming that T_{i-1} is a binary tree, we prove T_i is also a binary tree. Fix any arbitrary component c in C_i, we prove it cannot have more than two children. Let $U = \{c_1, \ldots, c_k\}$ be the components in C_{i-1} that were merged with each other to create c. For any component $c_j \in U$ let $\text{child}_U(c_j)$ denote the number of children of c_j that are in U and let $\text{child}(c_j)$ denote the total number of children of c_j (no matter if they are in U or not). For any child of c, there is a component in U with a child that is outside of U, therefore to prove c cannot have more than two children it suffices to prove

$$\sum_{c_j \in U} \big(\text{child}(c_j) - \text{child}_U(c_j)\big) \leq 2. \tag{1}$$

We first prove $\sum_{c_j \in U} \text{child}(c_j) \leq |U| + 1$. Note that at most one of the components in U has two children in T_{i-1} since all vertices with two children are selected in line 11 of Algorithm 2 and if two components are merged at least one of them is not selected (line 16 of Algorithm 2). Therefore other than one vertex in U which might have two children in C_{i-1}, others have at most one children. Hence $\sum_{c_j \in U} \text{child}(c_j) \leq |U| + 1$.

Next, we prove $\sum_{c_j \in U} \text{child}_U(c_j) \geq |U| - 1$. We know by Claim 32 that the components in U are connected to each other. Therefore there must be at least $|U| - 1$ edges within them and each such edge indicates a parent-child relation in U. Hence $\sum_{c_j \in U} \text{child}_U(c_j) \geq |U| - 1$.

Combining $\sum_{c_j \in U} \text{child}(c_j) \leq |U|+1$ and $\sum_{c_j \in U} \text{child}_U(c_j) \geq |U|-1$ we obtain Equation 1 always holds and therefore T_i is a binary tree. ◀

The following property bounds the number of unselected ancestors of a component that is going to be merged to its closest selected ancestor.

▶ **Claim 35.** *With probability at least $1 - 1/n$, for any given i and c_i, where $i \in [I]$ and $c_i \in C_i - F_i - S_i$, there are at most $2 \log n$ components in the path from c_i to its closest selected ancestor.*

Proof. We first show with probability $1 - 1/n^2$, there are at most $2 \log n$ components in the path from c to its closest selected ancestor. To see this, let P_c denote the set of ancestors of c that have distance at most $2 \log n$ to c. We prove with probability $1 - 1/n^2$ at least one component in P_c is selected. Since, all the children of any component in F_i is in S_i, If $P_c \cap F_i \neq \emptyset$, then $P_c \cap S_i \neq \emptyset$. Otherwise, any component in P_c is selected independently with probability at least $\frac{1}{2}$ and the probability that $P_c \cap S_i \neq \emptyset$ is $1 - 2^{-2 \log n} = 1 - 1/n^2$.

In addition we prove that $\Sigma_{i \in [I]} |C_i - F_i - S_i| \leq n$, so that we can use union bound over all the components in $\cup_{i \in [I]} C_i - F_i - S_i$. To prove this it suffices to prove that any vertex $v \in V(\mathcal{T})$ is at most the root vertex of one of the components in $\cup_{i \in [I]} C_i - F_i - S_i$. To prove this, let j denote the first iteration that v is the root vertex of a component in $C_j - F_j - S_j$. In this iteration we merge this component with one of its ancestors, so it can not be root vertex of any other component in $\cup_{i\in[I],i>j} C_i - F_i - S_i$. Therefore using union bound over all the components in $\cup_{i\in[I]} C_i - F_i - S_i$, we obtain that with probability at least $1 - 1/n$ for any given i and c_i, where $i \in [I]$ and $c_i \in C_i - F_i - S_i$, there are at most $2\log n$ components in the path from c_i to its closest selected ancestor. ◀

The following property implies that w.h.p. at each round at most a constant fraction of the incomplete components are selected.

▶ **Claim 36.** *There exists a constant number $0 < c < 1$ such that for any $i \in [K]$, with probability at least $1 - e^{-n/24}$, $|S_i| \leq c \cdot |C_i - F_i|$.*

Proof. Let $K_j(C_i)$ denote the set of components in C_i with exactly j children. Since each component in $K_1(C_i)$ is selected independently with probability $\frac{1}{2}$, by Chernoff bound, in round i,

$$\Pr[|K_1(C_i) \cap S_i| > \frac{3}{4}|K_1(C_i)|] \leq e^{-n/24}.$$

The components that are selected in each round are as follows. With probability at least $1 - e^{-n/24}$ we choose less than three forths of the components in $K_1(C_i)$, the root component, all the components in $K_j(C_i)$, and at most two components for any component in F_i. Therefore, with probability at least $1 - e^{-n/24}$,

$$|S_i| < 1 + |K_2(C_i)| + \frac{3}{4}|K_1(C_i)| + 2|F_i|. \tag{2}$$

By Fact 33, $|K_0(C_i)|= |K_2(C_i)| + 1$, and by Lemma 34, T_i is a binary tree, so $C_i = K_0(C_i) + K_1(C_i) + K_2(C_i)$. Therefore,

$$1 + |K_2(C_i)| + \frac{3}{4}|K_1(C_i)| < \frac{1}{2}(|K_2(C_i)| + |K_0(C_i)| + 1) + \frac{3}{4}|K_1(C_i)| \leq \frac{3}{4}|C_i|. \tag{3}$$

By equations 2 and 3 with probability at least $1 - e^{-n/24}$, $|S_i| \leq \frac{3}{4}|C_i| + 2|F_i|$. Therefore to complete the proof, it suffices to prove that there exists a constant number $0 < c < 1$ such that equation

$$\frac{3}{4}|C_i| + 2|F_i| \leq c \cdot |C_i - F_i|,$$

which is equivalent to

$$\frac{2-c}{c-3/4} \leq \frac{|C_i|}{|F_i|}$$

holds. Since the size of any component in F_i is at least n/m, $|F_i| \leq m$. In addition, by line 3 of Algorithm 2, $|C_i| > 14m$, therefore $\frac{|C_i|}{|F_i|} \geq 14$. It is easy to see that the equation

$$\frac{2-c}{c-3/4} \leq 14 \leq \frac{|C_i|}{|F_i|}$$

holds for $c = 5/6$, which is a constant number in (0, 1). ◀

The following property is a direct corollary of Claim 36.

▶ **Corollary 37.** *With high probability the while loop at line 5 of Algorithm 2, has at most $\mathrm{O}(\log n)$ iterations.*

We are now ready to prove each component has at most $\tilde{\mathrm{O}}(n/m)$ vertices.

▶ **Claim 38.** *The size of each components at any iteration of Algorithm 2 is bounded by $\tilde{\mathrm{O}}(n/m)$ with high probability.*

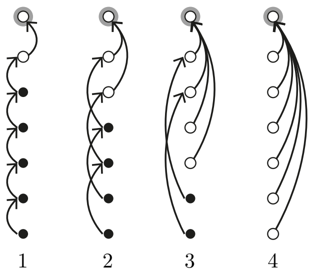

Figure 3 Intuition behind finding the closest selected ancestor in $O(\log\log n)$ rounds. The details are explained in the text.

Proof. By Claim 32 any component $c_i \in C_i$ is obtained by merging a connected subset of the components in C_{i-1}. Let $X_i \subset C_{i-1}$ denote this subset, and let $T(X_i)$ denote the component tree given by contracting all components in X_i. We first prove that $|X_i| = O(\log n)$.

By Claim 34, $T(X_i)$ is a binary tree, and by line 16 of Algorithm 2 $X_i - T(X_i).root \subset C_{i-1} - F_{i-1} - S_{i-1}$. Note that, any component in $C_{i-1} - F_{i-1} - S_{i-1}$ has at most 1 other component as a child, and $T(X_i).root$ has at most 2 children. In addition, by Claim 35, with high probability for any $c \in X_i - T(X_i)$.root the distance between c and $T(X_i)$.root is $2\log n$. Therefore, with high probability $|X_i| \leq 4\log n + 1$, which is $O(\log n)$.

In addition, note that $X_i \subset C_{i-1} - F_{i-1}$, and any component in $C_{i-1} - F_{i-1}$ has size at most $O(n/m)$, hence with high probability $|c_i| = \Sigma_{c\in X_i}|c| = \tilde{O}(n/m)$. ◀

D.3.2 Parallel Implementation of Algorithm 2

Algorithm 2 is a sequential implementation of our decomposition algorithm. To implement it in the desired model, each iteration of the while loop at line 5 of Algorithm 2 will be completed in several rounds. In fact, we show that with high probability each iteration can be completed in $O(\log\log n)$ rounds and that there are at most $O(\log n)$ iterations. This means the algorithm will be completed in $O(\log n \log\log n)$ rounds with high probability.

The sets C_i, F_i and S_i (Definition 31) may have up to $O(n)$ elements. This indicates that they cannot be stored in one machine and we have to distribute them among different machines. To that end, we use C_i^μ, F_i^μ, and S_i^μ to respectively denote the subsets of C_i, F_i and S_i that are stored in machine μ in the i-th iteration (recall that each iteration is not mapped to exactly one round and may contain several rounds).

In the first round, as analogous to lines 1-4 of Algorithm 2, each machine μ for any vertex v that is stored in it, creates a component that only contains v and stores it in C_1^μ. Clearly $\cup_{\mu\in\mathcal{M}} C_1^\mu = C_1$. Line 5 of Algorithm 2 however, requires to know the total number of components in all machines. To access this, each machine at the end of each iteration, shares the number components that it contains with all other machines.

Lines 6-15 of Algorithm 2, where some components are selected and added to S, can be locally implemented on each machine and over only the components that are stored in it. The only issue is that in line 12, we need to know whether the parent of a component, which may not be in the same machine, is a completed component or not. To handle this, we already know by Claim 34 that each component will be the parent of at most two other components at any iteration. Hence for any component, at the end of each iteration, we can notify the machines that contain its children whether it is marked as completed or not without violating the communication size limit.

In lines 16-18 of Algorithm 2, each component is merged to its closest selected ancestor

(CSA). This step requires several rounds in the parallel setting since the ancestors of a component may be stored in other machines. Let d denote the maximum distance of a component to its CSA. By iteratively querying the ancestors of the components we can find the CSA of all components in $\mathrm{O}(d)$ rounds. This algorithm can be improved to work in $\mathrm{O}(\log d)$ rounds. Consequently, since by Claim 35 we know with high probability that $d = \mathrm{O}(\log n)$, it takes at most $\mathrm{O}(\log\log n)$ rounds to find the CSA of all components.

At each round, for any component c, we use $F(c)$ to point to the furthest ancestor of c to which we have searched for a selected component. (Figure 3 illustrates how the algorithm updates $F(.)$; each component in Figure 3 is shown by a node and a directed edge from node x to node y indicates $F(x)$ points to y.) We initially set $F(c)$ to be c.`parent` for any component c (column 1 in Figure 3). In each round, each machine for any component c that is stored in it, sends a query to the machine containing the component c' where c'.`index`$=F(c)$ and in the next round receives a response containing $F(c')$; then updates $F(c)$ to be $F(c')$. If at any time, $F(c)$ points to a selected component, that is the CSA of component c and we do not update $F(c)$ anymore. (The white nodes in Figure 3 show the components that the algorithm has found their CSA.) The distance of any component c to its furthest explored ancestor $F(c)$ is exponentially increasing at each round until it points to a selected component and hence it takes $\mathrm{O}(\log d)$ rounds to find the closest selected ancestors of all components.

After finding the CSA of all components, we move any component that is not selected or finished to the machine that contains its CSA. Then in the next round, all components that have to be merged together will be in the same machine and can be merged.

Lines 19-21 of Algorithm 2 can be locally done on each machine and only over the components that are stored in it.

More details of the parallel implementation, and hence, the formal proof of Theorem 30 is left to the appendix.

E Omitted Proofs

Proof of Lemma 2

Proof. The main idea of the algorithm is to first partition $\mathcal{T}$ into two sub-trees $\mathcal{T}_1$ and $\mathcal{T}_2$ such that each of them has at most $\frac{2n_{\mathcal{T}}}{3}$ vertices. Unfortunately, in case all inherited border vertices of $\mathcal{T}$ lie within one of the sub-trees, we may end up having 4 border vertices in one of the sub-trees since the partitioning step creates one new border vertex in each of the sub-trees. To handle this case, we again partition the sub-tree with 4 border vertices into two sub-trees such that they will not have more than 3 border vertices at the end.

Let $D[v]$ denote the number of descendants of vertex v (including itself), and let $B[v]$ denote the number of border vertices among these descendants. It is easy to see that there exists linear time bottom-up dynamic algorithms to calculate these two vectors.

In the first step, we use Algorithm 3 to find an edge and claim removing it partitions $\mathcal{T}$ into two sub-trees $\mathcal{T}_1$ and $\mathcal{T}_2$, where $\max(|V(\mathcal{T}_1)|, |V(\mathcal{T}_2)|) \leq \frac{2n_{\mathcal{T}}}{3}$.

To prove the returned edge $e = (v, u)$ is a desired one, it suffices to prove $\frac{n_{\mathcal{T}}}{3} \leq D[u] \leq \frac{2n_{\mathcal{T}}}{3}$ since this yields that the size of the other partition is also less than or equal to $\frac{2n_{\mathcal{T}}}{3}$. The condition at Line 7 of Algorithm 3 ensures: (i) $D[u] \leq \frac{2n_{\mathcal{T}}}{3}$, and (ii) $D[v] > \frac{2n_{\mathcal{T}}}{3}$. The former inequality is a direct result of the condition; the latter also holds since otherwise v (the parent of u) would not have been updated. Using the second inequality and since $u = C_D[v]$ (i.e., the child of v with maximum number of descendants) we can obtain $D[u] \geq \frac{D[v]-1}{2} \geq \frac{n_{\mathcal{T}}}{3}$ as desired.

Algorithm 3

```
1: procedure FirstCut($\mathcal{T}$)
2:     $D \leftarrow$ Descendants($\mathcal{T}$)
3:     for any vertex $v$ of $T$ do
4:         $C_D[v] \leftarrow$ the child $u$ of $v$ with maximum $D[u]$
5:     $n_{\mathcal{T}} \leftarrow D[\mathcal{T}.root]$
6:     $v \leftarrow \mathcal{T}.root$
7:     while $D[C_D[v]] > \frac{2n_{\mathcal{T}}}{3}$ do
8:         $v \leftarrow C_D[v]$
9:     return edge $(v, C_D[v])$
```

After partitioning $\mathcal{T}$ we may end up having one partition with more than 3 border vertices (exactly 4) since removing an edge can add up to one new border vertex to each partition. In this case, let $\mathcal{T}_1$ denote the sub-tree with 4 border vertices. We run Algorithm 4 to partition $\mathcal{T}_1$ into two sub-trees with less than 4 border-vertices. ◀

Algorithm 4

```
1: procedure SecondCut($\mathcal{T}$)
2:     $B \leftarrow$ BorderVertices($\mathcal{T}$)
3:     for any vertex $v$ of $\mathcal{T}$ do
4:         $C_B[v] \leftarrow$ the child $u$ of $v$ with maximum $B[u]$
5:     $v \leftarrow \mathcal{T}.root$
6:     while $B[C_B[v]] > 2$ do
7:         $v \leftarrow C_B[v]$
8:     return $edge(v, C_B[v])$
```

Proof of Lemma 17

Proof. We use the following proposition by [44].

▶ **Proposition 39** (Corollary 3.34 in [44]). *For any $d, p, k \in \mathbb{Z}^+$, there is a family of k-wise independent hash functions $H = \{h : [d] \to \{0, 1, \ldots, 2^p - 1\}\}$, from which choosing a random function h takes $k \cdot \max\{\log d, p\}$ random bits, and evaluating any function $h \in H$ takes time* $\mathrm{poly}(\log d, p, k)$.

Recall that we assume the number of machines, m, is a power of two in the model. Therefore we can use Proposition 39 to choose the hash function $h : [k] \to \mathcal{M}$ with only $\max\{\log k, \log m\}$ random bits which we can generate on one machine and share with all other machines. The total communication of this machine would be $\mathrm{O}(m(\log k + \log m))$ which is $\tilde{\mathrm{O}}(n/m)$ since $m = \mathrm{O}(\sqrt{n})$ and k is $\mathrm{poly}(n)$. ◀

Proof of Lemma 21

To prove Lemma 21 we use a known result on the balls and bins problem by [42]. We start with the definition of the balls and bins problem.

▶ **Definition 40** (Balls and bins). In an instance of balls and bins problem there are n balls and n bins. Each ball is placed into one of the bins that is chosen uniformly at random. Let random variable X_i with range $\{1, .., n\}$ denote the bin that the i-th ball is assigned to. In this problem, the load of an arbitrary bin $b \in [n]$ is the number of balls assigned to this bin, which is equal to $\Sigma_{i=1}^{n} X_i = b$.

The following claim is based on the result of [42].

▶ **Claim 41.** *Let l_i denote load of the i-th bin in an instance of balls and bins problem in which random variables $X_1, \ldots, X_n$ are ($\log n$)-wise independent, then* $\Pr[\max_i^n l_i > 2\log n] \leq (\frac{2e}{\log n})^{\log n}$.

Proof. By Theorem 6-(V) in [42] we infer that, given an instance of balls and bins problem in which random variables $X_1, \ldots, X_n$ are ($\log n$)-wise independent, for any r and η such that $r \geq 1 + \eta + \log n$, and $k \geq \eta$ The following equation holds:

$$\Pr[l_i > r] \leq \frac{e^\eta}{(1+\eta)^{1+\eta}}$$

.

Using this result, by setting $\eta = \log n - 1$ and $r = 2\log n$ in the mentioned instance of balls and bins problem, for any $i \in [n]$, $\Pr[l_i > 2\log n] \leq (\frac{e}{\log n})^{\log n}$. Given this fact, by taking union bound over all n bins we achieve the following result:

$$\Pr[\max_i^n l_i > 2\log n] \leq n \cdot \Pr[l_i > 2\log n] = 2^{\log n} \cdot (\frac{e}{\log n})^{\log n} = (\frac{2e}{\log n})^{\log n}. \quad ◀$$

We are now ready to prove Lemma 21.

Proof of Lemma 21. Our starting point is Lemma 41. We represent the elements of A as balls and the machines as bins. However, there are two main problems. First, $|A|$ may be much larger than the number of machines (m), while Lemma 41 expects the number of balls to be equal to the number of bins. Second, the balls are unweighted in Lemma 41 whereas the elements of A are weighted.

To resolve the above issues, we first partition the elements of A into $\lceil n/m \rceil$ ($= \alpha$) subsets $S_1, S_2, \ldots, S_\alpha$ of size m (except the last subset S_α which might have fewer elements if n/m is not an integer) such that the elements are grouped based on their weights. More precisely, S_1 contains the m elements of A with the lowest weights, S_2 contains the m elements in $A - S_1$ with the lowest weights, and so on.

▶ **Claim 42.** *For $m = n^\delta$, with probability at least $1 - (\frac{2^{1/\delta}e}{\delta \log n})^{\delta \log n}$ there is no set S_i where h maps more than $\mathrm{O}(\log m)$ elements of S_i to the same machine.*

Proof. Since h is chosen from a ($\log m$)-universal hash function, and since each S_i has at most m elements, the machines that the elements in S_i are assigned to are ($\log m$)-wise independent. Therefore, by fixing a set S_i, we can use Lemma 41 to prove at most $\mathrm{O}(\log m)$ elements of S_i are assigned to the same machine with probability $1 - (2e/\log m)^{\log m}$. By taking union bound over the failure probability of all sets we obtain that with probability at least

$$1 - \frac{n}{m} * (\frac{2e}{logm})^{\log m} = 1 - n(\frac{e}{\log m})^{\log m} = 1 - (\frac{2^{1/\delta}e}{\delta \log n})^{\delta \log n}$$

there is no set S_i where h maps more than $\mathrm{O}(\log m)$ elements of S_i to the same machine. ◀

Let $w_i = \max_{a \in S_i} w(a)$ denote the weight of the element in S_i with the maximum weight. Also let $l_h^i(\mu)$ denote the load of h on machine μ from the elements in set S_i (i.e., $l_h^i(\mu) = \sum_{a \in S_i : h(a) = \mu} w(a)$). By Claim 42, with high probability, from any set S_i, at most

$\mathrm{O}(\log m)$ elements are mapped to the same machine. Also since the maximum weight of the elements in S_i is w_i we know with high probability that $l_h^i(\mu) \leq \mathrm{O}(\log m \cdot w_i)$ for any i. We know by Definition 20 that $l_h(\mu) = \sum_{i=1}^{\alpha} l_h^i(\mu)$, therefore

$$l_h(\mu) = l_h^1(\mu) + \ldots + l_h^\alpha(\mu) \leq \mathrm{O}(\log m \cdot (w_1 + \ldots w_\alpha)). \tag{4}$$

Therefore the maximum load of h is $\tilde{\mathrm{O}}(w_1 + w_2 + \ldots + w_\alpha)$. To complete our proof, it suffices to prove $w_1 + w_2 + \ldots + w_\alpha = O(n/m)$. To see this, note that $w(a) \geq w_{i-1}$ for any a in S_i or otherwise a would have been in S_{i-1} instead of S_i. Therefore,

$$\sum_{a \in S_i} w(a) \geq |S_i| \cdot w_{i-1} = m \cdot w_{i-1} \qquad \forall i : 1 \leq i < \alpha, \tag{5}$$

which means,

$$m(w_1 + w_2 + \ldots + w_{\alpha-1}) \leq \sum_{i=2}^{\alpha} \sum_{a \in S_i} w(a). \tag{6}$$

On the other hand since the total weight of the elements in A is $\mathrm{O}(n)$ by Definition 19, we obtain

$$m(w_1 + w_2 + \ldots + w_{\alpha-1}) \leq \mathrm{O}(n), \tag{7}$$

and therefore $\sum_{i=1}^{\alpha-1} w_i \leq \mathrm{O}(n/m)$. Finally since by Definition 19 we know the maximum weight in A, and consequently w_α is not more than n/m, we prove $\sum_{i=1}^{\alpha} w_i \leq \mathrm{O}(n/m)$ as desired. ◀

Proof of Claim 27

Proof. Algorithm 5 is the implementation of such an algorithm in $\mathcal{MPC}$. Before running this algorithm, we distribute all the vertices in $V(\mathcal{T})$ over $\mathcal{M}$ machines independently and uniformly at random. To construct $\mathcal{T}^d$ we first add $|V(\mathcal{T})|$ vertices to it with the same set of indexes as the vertices in $V(\mathcal{T})$. These vertices do not have any parent yet, so we assign parents to them later in the algorithm. We will also add some auxiliary vertices to this tree.

Set Δ to be $\lceil \frac{n}{m} \rceil$. For any vertex $v \in \mathcal{T}$, let C_v denote the set of children of v in tree $\mathcal{T}$, and let v^d denote the vertex equivalent to v in $\mathcal{T}^d$, which has the same index as v. If $\Delta \geq |C_v|$ for any vertex $u \in C_v$, we set the parent of vertex u^d as v^d. Otherwise, if $\Delta < |C_v|$, we add a set of $\frac{|C_v|}{\Delta}$ auxiliary vertices to $V(\mathcal{T}^d)$, denoted by A_v, such that v^d is the parent of all the vertices in A_v, and for any vertex $u \in C_v$, we choose one of the auxiliary vertices in A_v independently and uniformly at random as the parent of the vertex u^d. This guarantees that, for any two vertex u and v that v is an ancestor of u in $\mathcal{T}$, v^d is also an ancestor of u^d. In addition $|V(\mathcal{T}^d)| = \mathrm{O}(|V(\mathcal{T})|)$, so $\mathcal{T}^d$ is an extension of T. Note that $\frac{|C_v|}{\Delta} \leq m \leq \mathrm{O}(n/m) = \mathrm{O}(\Delta \log n)$, therefore for any vertex v of T, the degree of v^d is at most $\mathrm{O}(\Delta \log n)$. Moreover, for any two vertices u and v such that $|C_v| > \Delta$ and $u \in A_v$, it is easy to see that by Claim 41 with high probability the maximum number of vertices from C_v that choose u as the parent for their equivalent vertex in T^d, is at most $\mathrm{O}(\Delta \log n)$ since any vertex in C_v choses a vertex in A_v independently and uniformly at random. Therefore, the maximum degree in $\mathcal{T}^d$ is $\mathrm{O}(\Delta \log n)$.

To implement this algorithm in $\mathcal{MPC}$, each machine in $\mathcal{M}$ needs to have the list of all the vertices in $\mathcal{T}$ with degree more than Δ alongside with the number of children each one's equivalent vertex will have in tree T^d . Let K_μ denote the set of such vertices of

$\mathcal{T}$ in any machine $\mu \in \mathcal{M}$, and let S_μ be a dictionary with the set of keys K_μ, such that $S_\mu(v)$ is the number of children that vertex v^d will have in tree $\mathcal{T}^d$ for any vertex $v \in K_\mu$. We assume that each machine has the degree of all the vertices that are stored in it. By Lemma 25 we can achieve this in $O(1)$ rounds of $\mathcal{MPC}$. In the first round of the algorithm each machine constructs and sends this dictionary to all machines. In the next round, each machine construct a dictionary S which is the union of all the dictionaries received in round 1, and its own dictionary. (Since the dictionaries have distinct set of keys the union of them is well-defined.) For any vertex $v \in V(\mathcal{T})$ we construct v^d in the same machine as v, and using S, we set its parent. In addition, if the degree of v in tree $\mathcal{T}$ is more than Δ, we add $\frac{S(v)}{\Delta}$ auxiliary vertices as the children of v^d in this machine. It is easy to see that the data in dictionary S, suffices to assign a unique index to any auxiliary vertex. After this algorithm terminates each vertex of $V(\mathcal{T}^d)$ is stored in exactly one machine.

To prove that during the algorithm, with high probability no machine violates the memory limit of $\tilde{O}(n/m)$, it suffices to prove that with high probability the number of vertices with degree greater than Δ in any machine $\mu \in \mathcal{M}$ is $O(\log n)$, since it yields that the size of the message each machine sends in round 1 of the algorithm is $O(\log n)$, and hence we have $\tilde{O}(m) \leq \tilde{O}(n/m)$, the overall message passing for each machine does not violate the memory limit. Moreover, it indicates that the total number of vertices added to each machine is at most $O(\log n) \cdot \frac{n}{\Delta} \leq \tilde{O}(n/m)$. To prove this claim, we use two facts. First, before running the algorithm we uniformly distribute the vertices over all the machines, and second, the total number of the vertices with degree at least Δ in a tree is at most $O(\frac{n}{\Delta}) \leq O(m)$. So, by Claim 41 with high probability there are at most $O(\log n)$ vertices with degree at least Δ in any machine $\mu \in \mathcal{M}$. ◀

Proof of Claim 28

Proof. Algorithm 6 is the implementation of an algorithm to convert $\mathcal{T}^d$ to $\mathcal{T}^b$ in $\mathcal{MPC}$. To construct $\mathcal{T}^b$ we first add $|V(\mathcal{T}^d)|$ vertices to it with the same set of indexes as the vertices in $V(\mathcal{T}^d)$. These vertices do not have any parent yet, so we assign parents to them later in the algorithm. We will also add some auxiliary vertices to this tree. Let C_v denote the set of children of vertex v, for any $v \in V(\mathcal{T}^d)$, and let v^b denote its equivalent vertex in $\mathcal{T}^b$. The overall idea in this algorithm is as follows: for any vertex with more than two children we construct a binary tree t_v with root v^b such that for any $u \in C_v$, u^b is a leave in this tree, and $|V(t_v)| = 2|C_v| - 1$. Then we add the vertices of t_v excluding the root to binary tree $\mathcal{T}^b$ with the same parent they have in t_v. In addition, for any v with less than 3 children we set the parent of its children's equivalent vertices in $\mathcal{T}^b$ as v^b.

To implement this algorithm in $\mathcal{MPC}$, we need to have all the children of any vertex in one machine. To achieve this, we first distribute all the vertices in $V(\mathcal{T}^d)$ over m machines using a hash function $h : [|V(\mathcal{T}^b)|] \rightarrow [m]$ which is chosen uniformly at random from a family of $\log n$-universal hash functions. By Corollary 23 it is possible to distribute $V(\mathcal{T}^d)$ by hash function h in $O(1)$ round of $\mathcal{MPC}$, and using $O(\log m)$ time and space on each machine in such a way that each machine can evaluate h in $O(\log m)$ time and space. Let V_μ denote the set of vertices that is assigned to an arbitrary machine μ by this distribution. For any vertex $v \in V_\mu$ let p_v denote v's parent, and let μ_v be the machine that $p_v \in V_{\mu_v}$. Since evaluating h in each machine takes $O(\log m)$ time and space, It is possible to find μ_v for all $v \in V_\mu$ using $\tilde{O}(n/m)$ time and space in machine μ. We send any vertex v as a message to the machine μ_v. In this way all the children of any vertex are sent to the same machine. In addition since $|C_v| \leq \tilde{O}(n/m)$, by Lemma 21 with high probability the total size of the messages received

Algorithm 5 An algorithm to convert a given tree $\mathcal{T}$ to a bounded degree tree $\mathcal{T}^d$, an extension of $\mathcal{T}$

```
Require: given a number $\Delta$ this algorithm converts a tree $\mathcal{T}$ whose vertices are distributed
    over $m$ machines to an extension of it, $\mathcal{T}^d$ with maximum degree $O(\Delta \log n)$
 1: Let $\mu$ denote the machine in which this instance of algorithm is running, and let $V_\mu$
    denote the set of vertices of $\mathcal{T}$ that are stored in this machine.
 2: round 1:
 3:     Let $S_\mu$ be a dictionary with the set of keys $K_\mu = \{v.\texttt{index} | \deg(v) > \Delta, v \in V_\mu\}$.
 4:     For any $v$, such that $v.\texttt{index} \in K_\mu$, $S_\mu(v.\texttt{index}) = \lceil \frac{|\deg(v)|}{\Delta} \rceil$.
 5:     Send $S_\mu$ and $K_\mu$ to all the other machines.
 6: end
 7: round 2:
 8:     $K \leftarrow \cup_{i \in \mathcal{M}} K_i$
 9:     $S \leftarrow \cup_{i \in \mathcal{M}} S_i$          ▷ Since $K_i$s are distinct, for any $i \in K$ S[i] has exactly one value
10:     Let $D$ be a dictionary on the same set of keys as $S$, while for any $i \in K$, $D[i] =$
    $n + \Sigma_{j \in S, j < i} S[j]$.
11:     for every $v$ in $V_\mu$ do
12:         if $v.\texttt{parent} \in K$ then
13:             Choose $x$ uniformly at random from $[S(v.\texttt{index})]$.
14:             $v.\texttt{parent} \leftarrow D[v.\texttt{parent}] + x$          ▷ $D[v.\texttt{parent}] + x$ is a unique index in the
    range $[n, 2n]$ assigned to the $x$-th child of vertex v in $\mathcal{T}^d$
15:     for every $v$ in $K_\mu$ do
16:         for every $j$ in $[S[v]]$ do
17:             Add Vertex$(v, D[v] + j)$ to $V_\mu$ .
18: end
```

by any machine μ which is equal to $\Sigma_{v in V_\mu} |C_v|$ is $\tilde{O}(n/m)$.

For any vertex $v \in V(\mathcal{T}^d)$ where $|C_v| > 2$, we add some auxiliary vertices to $\mathcal{T}^b$, which does not have any equivalent vertex in $V(\mathcal{T}^d)$, so we need to assign an index to them. To make sure that these assigned indexes are unique, any machine μ computes the number of all the auxiliary vertices generated in this machine, which is equal to$\Sigma_{v \in V_\mu} \max(|C_v| - 2, 0)$, and send it to all the other machines with index greater than $\mu.\texttt{index}$. It is easy to see that using this received data each machine is able to generate unique indexes for any auxiliary vertex in it. Figure 1 represents how this algorithm works. The gray vertices in this figure are the auxiliary vertices added to the tree. ◀

Algorithm 6 An algorithm to convert a given bounded degree tree $\mathcal{T}^d$ to a binary extension of it, $\mathcal{T}^d$

```
Require: In the beginning of the algorithm vertices of the tree $\mathcal{T}^d$ with maximum degree
    $O(\Delta \log n)$ are distributed over $m$ machines using a hash function $h$. This algorithm
    converts $\mathcal{T}^d$ to a binary tree $\mathcal{T}^b$, which is an extension of $\mathcal{T}^d$.
 1: Let $\mu$ denote the machine in which this instance of algorithm is running, and let $V_\mu$
    denote the set of vertices of $\mathcal{T}^d$ that are stored in this machine.
 2: round 1:
 3:     for every $v$ in $V_m$ do
 4:         $\mu_d = h(v.\texttt{parent})$
 5:         Send $(v.\texttt{parent}, v)$ to machine $\mu_d$.        ▷ Send it even if $\mu_d = \mu$.
 6:         Delete v from $V_\mu$.
 7: end
 8: round 2:
 9:     Let $R$ denote the set of all the messages received in the previous round.
10:     $C_v \leftarrow \{u | (v, u) \in R\}$        ▷ $C_v$ is the set of all the children of vertex $v$
11:     Send $\Sigma_{v \in V_\mu} \max(|C_v| - 2, 0)$ to any machine $\mu'$ such that $\mu'.\texttt{index} > \mu.\texttt{index}$.
12: end
13: round 3:
14:     Let s denote the summation of all the messages received in the previous round.
15:     $d \leftarrow s + 1$
16:     for every $v$ in $V_\mu$ do
17:         Construct a binary tree $b_v$ with root $v$, leaves $C_v$, and $|C_v| - 2$ inner vertices
    indexed by $d, \ldots, d + |C_v| - 2$.  ▷ Any vertex in a tree that is not a root, nor a leave is
    considered as an inner vertex. If $|C_v| \leq 2$ there is no inner vertex.
18:         $d \leftarrow d + \max(0, |C_v| - 2)$
19:         for every vertex $u$ in $b_v$ excluding $v$ do
20:             Let $u$.par denote the index of $u$'s parent in the tree $b_v$.
21:             Add Vertex($u$.par, $u.\texttt{index}$) to $V_m$.
22: end
```

Algorithm 7 This algorithm runs on any machine μ in $\mathcal{MPC}$ to decompose a binary tree $\mathcal{T}^b$

```
Require: A binary tree $\mathcal{T}^b$.
1: Before this algorithm begins, we distribute vertices of $\mathcal{T}^b$ using a hash function $h$ choosing
   uniformly from a family of $\log n$-universal hash functions.
2: round 1:
3:     $C^\mu \leftarrow \emptyset$                ▷ $C^\mu$ will contain all components in machine $\mu$.
4:     for each vertex $v$ in $\mathcal{T}^b$ do
5:         Insert a component containing only $v$ to $C^\mu$.
6:     $F^\mu \leftarrow \emptyset$          ▷ $F^\mu$ will contain the completed components in machine $\mu$.
7:     $F \leftarrow \emptyset$ ▷ $F$ will contain the index of all the completed components in all the machines
8: end
9: repeat
10:     begin round
11:         $S^\mu \leftarrow \emptyset$          ▷ $S^\mu$ will contain the selected components in machine $\mu$.
12:         for each component $c \in C^\mu - F^\mu$ do
13:             if $c$ is the root component then
14:                 Add $c$ to $S^\mu$.
15:             else if $c$ has exactly two children components then
16:                 Add $c$ to $S^\mu$.
17:             else if the index of the parent component of $c$ is in $F$ then
18:                 Add $c$ to $S^\mu$.
19:             else if $c$ has exactly one child component then
20:                 Add $c$ to $S^\mu$ with probability 0.5.
21:     end round
22:     FCSA$(C_i^\mu, F_i^\mu, S_i^\mu)$
23:     begin round
24:         for every $c$ in $C_i^\mu - F_i^\mu - S_i^\mu$ do
25:             Send message $(c, F(c))$ to the machine containing $F(c)$.
26:             delete $c$ from $C_i^\mu$.
27:     end round
28:     begin round
29:         for every node $c$ in $C^\mu - F^\mu$ do
30:             For any message $(c', c)$ received in the previous round, merge component $c'$
   with $c$.
31:             if $c$ contains at least $n/m$ vertices of $\mathcal{T}^b$ then
32:                 Add $c$ to $F^\mu$.
33:                 Add $c$.index to $F$.
34:                 Send message ($c$.index) to all the other machines.
35:     end round
36:     begin round
37:         for every component $c$ that $c$.index received in the previous round do
38:             Add $c$.index to $F$.
39:     end round
40: until $\sum_{i \in \mathcal{M}} |C^i| > 14m$
```

E.1 Parallel Implementation of The Decomposition

Algorithm 8 Finding the closest selected ancestor of all components in machine μ.

```
Require: During the process of finding closest selected ancestor, all the machines run this
    algorithm. For any component $c$ that is not selected or completed at the end of this
    algorithm it has access to the index of its closest selected ancestor, denoted by $F(c)$.
 1: procedure FCSA($C_i^\mu, F_i^\mu, S_i^\mu$)
 2:     begin round
 3:         for any $c \in C^\mu - F^\mu - S^\mu$ do
 4:             $F(c) \leftarrow c$.parent
 5:     end round
 6:     while we receive requests or want to send requests do
 7:         begin round
 8:             Let $Q^\mu$ and $R^\mu$ respectively denote the set of queries and responses that are
    sent to machine $\mu$.
 9:             for each response $\{q : r\} \in R^\mu$ do
10:                 $F(c) \leftarrow r$ where $c$ is the component in machine $\mu$ for which $F(c) = q$.
11:             for each component $c \in C^\mu - F^\mu - S^\mu$ do
12:                 if $F(c)$ is not selected then
13:                     Send query $F$(c) to the machine containing component with index $F(c)$.
14:             for each query $q \in Q^\mu$ do
15:                 Send response $F(q)$ to the machine containing component with index $q$
16:         end round
```

Proof of Theorem 30. Algorithm 7 is the implementation of Algorithm 2 in $\mathcal{MPC}$. by Definition 29 a decomposition of a binary tree $\mathcal{T}^b$ is a set C of components with some properties that we mention each of them and prove that the decomposition given by Algorithm7 satisfies them. Let $C = \cup_{\mu\in\mathcal{M}} C^\mu$ where for any machine μ, C^μ is the components stored in machine μ when this algorithm terminates.

1. $|C| = \mathrm{O}(m)$: by line 40 in Algorithm 7, the number of components when the algorithm terminates is less than $14m = \mathrm{O}(m)$.
2. Each component contains a subset of the vertices of $\mathcal{T}^b$ that are connected: by Claim 32 the decomposition given by Algorithm 7 satisfies this property.
3. Each vertex of $\mathcal{T}^b$ is in exactly one component: it is easy to see that Algorithm 7 satisfies this property.
4. The data stored in each component is of size up to $\tilde{\mathrm{O}}(n/m)$: by Claim 38, the size of each component is at most $\tilde{\mathrm{O}}(n/m)$. Moreover by Claim 34 the component tree is always a binary tree. Therefore the data stored for each component which includes its vertices and the index of adjacent components in component tree is $\tilde{\mathrm{O}}(n/m)$.

By Corollary 37, with high probability this algorithm terminates in $\tilde{\mathrm{O}}(\log n)$ rounds of $\mathcal{MPC}$. We also prove that during this algorithm no machine uses memory more than $\tilde{\mathrm{O}}(n/m)$. At the beginning of Algorithm 7, we distribute vertices of $\mathcal{T}^b$ using a hash function h choosing uniformly at random from a family of $\log n$-universal hash functions. Let $R_i \in V(\mathcal{T}^b)$ denotes the root vertices of the components in $\cup_{\mu\in\mathcal{M}} C_i^\mu$ in i-th iteration of the algorithm. In addition, for any $\mu \in \mathcal{M}$, let R_i^μ denote the set of vertices in R_i that is assigned to machine μ by hash function h at the beginning of algorithm. At the end of any iteration i, the component with root $r \in R_i^\mu$ is in machine μ. By claim 38, the size of each component is at most

$\tilde{O}(n/m)$. Moreover the total size of the components is $O(n)$. Therefore by Lemma 21, with high probability the maximum load at the end of each iteration on any machine is at most $\tilde{O}(n/m)$. We are allowed to use this Lemma here since the decomposition is each iteration is independent from the distribution of vertices at the beginning of the algorithm. To sum up, this implies that there exists a randomized algorithm to find a decomposition of a given binary tree $\mathcal{T}^b$ such that with high probability the algorithm terminates in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$, and each machine uses a memory of size at most $\tilde{O}(n/m)$ and runs an algorithm that is linear in its memory size. ◀