Adaptive Massively Parallel Constant-Round Tree Contraction

MohammadTaghi Hajiaghayi ⊠

University of Maryland, College Park, MD, USA

University of Maryland, College Park, MD, USA

University of Maryland, College Park, MD, USA

Hsin-Hao Su ⊠

Boston College, MA, USA

Abstract -

Miller and Reif's FOCS'85 [49] classic and fundamental tree contraction algorithm is a broadly applicable technique for the parallel solution of a large number of tree problems. Additionally it is also used as an algorithmic design technique for a large number of parallel graph algorithms. In all previously explored models of computation, however, tree contractions have only been achieved in $\Omega(\log n)$ rounds of parallel run time. In this work, we not only introduce a generalized tree contraction method but also show it can be computed highly efficiently in $O(1/\epsilon^3)$ rounds in the Adaptive Massively Parallel Computing (AMPC) setting, where each machine has $O(n^\epsilon)$ local memory for some $0 < \epsilon < 1$. AMPC is a practical extension of Massively Parallel Computing (MPC) which utilizes distributed hash tables [10, 16, 44]. In general, MPC is an abstract model for MapReduce, Hadoop, Spark, and Flume which are currently widely used across industry and has been studied extensively in the theory community in recent years. Last but not least, we show that our results extend to multiple problems on trees, including but not limited to maximum and maximal matching, maximum and maximal independent set, tree isomorphism testing, and more.

2012 ACM Subject Classification Theory of computation \rightarrow Massively parallel algorithms

Keywords and phrases Adaptive Massively Parallel Computation, Tree Contraction, Matching, Independent Set, Tree Isomorphism

Digital Object Identifier 10.4230/LIPIcs.ITCS.2022.83

Related Version Full Version: https://arxiv.org/abs/2111.01904

Funding MohammadTaghi Hajiaghayi: Supported by the NSF BIGDATA Grant No. 1546108, NSF SPX Grant No. 1822738, and NSF AF Grant No. 2114269.

Marina Knittel: Supported by the NSF BIGDATA Grant No. 1546108, NSF SPX Grant No. 1822738, ARCS Endowment Award, and Ann G. Wylie Fellowship.

Hsin-Hao Su: Supported by NSF Grant No. CCF-2008422.

1 Introduction

In this paper, we study and extend Miller and Reif's fundamental FOCS'85 [48-50] $O(\log n)$ round parallel tree contraction method. Tree contraction is a process involving iterated
contraction on graph components for efficient computation of problems on trees (see Section 1.2). Their work leverages PRAM, a model of computation in which a large number of
processors operate synchronously under a single clock and are able to randomly access a large
shared memory. In PRAM, tree contractions require n processors. Though the initial study
of tree contractions was in the CRCW (concurrent read from and write to shared memory)
PRAM model, this was later extended to the stricter EREW (exclusive read from and write

© MohammadTaghi Hajiaghayi, Marina Knittel, Hamed Saleh, and Hsin-Hao Su; licensed under Creative Commons License CC-BY 4.0

13th Innovations in Theoretical Computer Science Conference (ITCS 2022).

Editor: Mark Braverman; Article No. 83; pp. 83:1–83:23

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to shared memory) PRAM model [27] as well, and then to work-optimal parallel algorithms with $O(n/\log n)$ processors [29]. Since then, a number of additional works have also built on top of Miller and Reif's tree contraction algorithm [1,25,35]. Tree-based computations have a breadth of applications, including natural graph problems like matching and bisection on trees, as well as problems that can be formulated on tree-like structures including expression simplification.

The tree contraction method in particular is an extremely broad technique that can be applied to many problems on trees. Miller and Reif [49] initially motivated their work by showing it can be used to evaluate arithmetic expressions. They additionally studied a number of other applications [50], using tree contractions to construct the first polylogarithmic round algorithm for tree isomorphism and maximal subtree isomorphism of unbounded degrees, compute the 3-connected components of a graph, find planar embeddings of graphs, and compute list-rankings. An incredible amount of research has been conducted to further extend the use of tree contractions for online evaluation of arithmetic circuits [47], finding planar graph separators [30], approximating treewidth [20], and much more [9,36,37,42,46,52]. This work extends classic tree contractions to the adaptive massively parallel setting.

The importance of large-scale data processing has spurred a large interest in the study of massively parallel computing in recent years. Notably, the Massively Parallel Computation (MPC) model has been studied extensively in the theory community for a range of applications [2–8, 10, 12, 15, 17, 19, 22, 26, 31, 38, 41, 45, 51, 53, 54], many with a particular focus on graph problems. MPC is famous for being an abstraction of MapReduce [43], a popular and practical programming framework that has influenced other parallel frameworks including Spark [55], Hadoop [28], and Flume [23]. At a high level, in MPC, data is distributed across a range of low-memory machines which execute local computations in rounds. At the end of each round, machines are allowed to communicate using messages that do not exceed their local space constraints. In the most challenging space-constrained version of MPC, we restrict machines to $O(n^{\epsilon})$ local space for a constant $0 < \epsilon < 1$ and $\widetilde{O}(n+m)$ total space (for graphs with m edges, or just $\widetilde{O}(n)$ otherwise).

The computation bottleneck in practical implementations of massively parallel algorithms is often the amount of communication. Thus, work in MPC often focuses on round complexity, or the number of rounds, which should be $O(\log n)$ at a baseline. More ambitious research often strives for sublogarithmic or even constant round complexity, though this often requires very careful methods. Among others, a specific family of graph problems known as Locally Checkable Labeling (LCL) problems – which includes vertex coloring, edge coloring, maximal independent set, and maximal matching to name a few – admit highly efficient MPC algorithms, and have been heavily studied during recent years [6,7,14,19,26,32,34]. Another consists of DP problems on sequences including edit distance [22] and longest common subsequence [40], as well as pattern matching [39]. The round complexity of aforementioned MPC algorithms can be interpreted as the parallelization limit of the corresponding problems.

While MPC is generally an extremely efficient model, it is theoretically limited by the widely believed 1-vs-2Cycle conjecture [33], which poses that distinguishing between a graph that is a single n-cycle and a graph that is two n/2-cycles requires $\Omega(\log n)$ rounds in the low-memory MPC model. This has been shown to imply lower bounds on MPC round complexity for a number of other problems, including connectivity [17], matching [33,51], clustering [54], and more [5,33,45]. To combat these conjectured bounds, Behnezhad et al. [16] developed a stronger and practically-motivated extension of MPC, called Adaptive Massively Parallel Computing (AMPC). AMPC was inspired by two results showing that adding distributed hash tables to the MPC model yields more efficient algorithms for finding

connected components [44] and creating hierarchical clusterings [10]. AMPC models exactly this: it builds on top of MPC by allowing in-round access to a distributed read-only hash table of size O(n+m). See Section 1.1 for a formal definition.

In their foundational work, Behnezhad et al. [16] design AMPC algorithms that outperform the MPC state-of-the-art on a number of problems. This includes solving minimum spanning tree and 2-edge connectivity in $\log\log_{m/n}(n)$ AMPC rounds (outperforming $O(\log n)$ and $O(\log D\log\log_{m/n}n)$ MPC rounds respectively), and solving maximal independent set, 2-Cycle, and forest connectivity in O(1) AMPC rounds (outperforming $O(\sqrt{\log n})$, $O(\log n)$, and $O(\log D\log\log_{m/n}n)$ MPC rounds respectively). Perhaps most notably, however, they proved that the 1-vs-2Cycle conjecture does not apply to AMPC by finding an algorithm to solve connectivity in $O(\log\log_{m/n}n)$ rounds. This was later improved to be $O(1/\epsilon)$ by Behnezhad et al. [18], who additionally found improved algorithms for AMPC minimum spanning forest and maximum matching. Charikar, Ma, and Tan [24] recently show that connectivity in the AMPC model requires $O(1/\epsilon)$ rounds unconditionally, and thus the connectivity result of Behnezhad et al. [18] is indeed tight. In a subsequent work, Behnezhad [13] shows an $O(1/\epsilon)$ -round algorithm for the maximal matching problem in AMPC.

A notable drawback of the current work in AMPC is that there is no generalized framework for solving multiple problems of a certain class. Such methods are important for providing a deeper understanding of how the strength of AMPC can be leveraged to beat MPC in general problems, and often leads to solutions for entirely different problems. Studying Miller and Reif [49]'s tree contraction algorithm in the context of AMPC provides exactly this benefit. We get a generalized technique for solving problems on trees, which can be extended to a range of applications.

Recently, Bateni et al. [11] introduced a generalized method for solving "polylog-expressible" and "linear-expressible" dynamic programs on trees in the MPC model. This was heavily inspired by tree contractions, and also is a significant inspiration to our work. Specifically, their method solves minimum bisection, minimum k-spanning tree, maximum weighted matching, and a large number of other problems in $O(\log n)$ rounds. We extend these methods, as well as the original tree contraction methods, to the AMPC model to create more general techniques that solve many problems in $O_{\epsilon}(1)$ rounds.

1.1 The AMPC Model

The AMPC model, introduced by Behnezhad et. al [16], is an extension of the standard MPC model with additional access to a distributed hash table. In MPC, data is initially distributed across machines and then computation proceeds in rounds where machines execute local computations and then are able to share small messages with each other before the next round of computation. A distributed hash table stores a collection of key-value pairs which are accessible from every machine, and it is required that both key and value have a constant size. Each machine can adaptively query a bounded sequence of keys from a centralized distributed hash table during each round, and write a bounded number of key-value pairs to a distinct distributed hash table which is accessible to all machines in the next round. The distributed hash tables can also be utilized as the means of communication between the machines, which is implicitly handled in the MPC model, as well as a place to store the initial input of the problem. It is straight-forward to see how every MPC algorithm can be implemented within the same guarantees for the round-complexity and memory requirements in the AMPC model.

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

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

1.2 Our Contributions

The goal of this paper is to present a framework for solving various problems on trees with constant-round algorithms in AMPC. This is a general strategy, where we intelligently shrink the tree iteratively via a decomposition and contraction process. Specifically, we follow Miller and Reif's [49] two-stage process, where we first compress each connected component in our decomposition, and then rake the leaves by contracting all leaves of the same parent together. We repeat until we are left with a single vertex, from which we can extract a solution. To retrieve the solution when the output corresponds to many vertices in the tree (i.e., maximum matching istead of maximum matching value), we can undo the contractions in reverse order and populate the output as we gradually reconstruct the original tree.

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

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

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

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

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

This is a nice independent result, proving a slightly more efficient $O(1/\epsilon^2)$ -round algorithm on degree bounded trees. Additionally, many problems on larger degree trees can be represented by lower degree graphs. For example, both the original Miller and Reif [48] tree contraction and the Betani et al. [11] framework consider only problems in which we can replace each high degree vertex by a balanced binary tree, reducing the tree-based computation on general trees to a slightly different computation on binary trees. Equally notably, it is an important subroutine in our main algorithm.

- ▶ Theorem 2. Consider a degree-weighted tree T = (V, E, W) and a problem P. Given a connected contracting function on T with respect to P, one can compute P(T) in $O(1/\epsilon^2)$ AMPC rounds with $O(n^{\epsilon})$ memory per machine and $\widetilde{O}(n)$ total memory if $\deg(v) \leq n^{\epsilon}$ for every vertex $v \in V$.
- ▶ Remark 3. It may be tempting to suggest that in most natural problems the input tree can be transformed into a tree with degree bounded by n^{ϵ} . However, we briefly pose the MedianParent problem, where leaves are given values and parents are defined recursively as the median of their children. By transforming the tree to make it degree bounded, we lose necessary information to find the median value among the children of a high degree vertex.

Next, we move onto our main result: a generalized tree contraction algorithm that works on any input tree with arbitrary structure. Building on top of Theorem 2, we can create a natural extension of tree contractions. Recall that the black-box contracting functions encode the data associated with a contracted vertex in terms of its children post-contraction. Thus, allowing high degree vertices introduces difficulties working with contracting functions. In particular, it is not possible to store the weight vector W(v) of a high degree vertex v inside the local memory of a single machine. The power of this algorithm is its ability to implement Compress and Rake for n^{ϵ} -tree-contractions in $O(1/\epsilon^3)$ rounds.

The most significant novelty of our main algorithm is the handling of high degree vertices. To do this, we first handle all maximal connected components of low degree vertices using the algorithm from Theorem 2 as a black-box. This compresses each such component into one vertex without needing to handle high degree vertices. By contracting these components, we obtain a special tree called Big-Small-tree (defined formally in §3.2) which exhibits nice structural properties. Since the low degree components are maximal, the degree of each vertex in every other layer is at least n^{ϵ} , implying that a large fraction of the vertices in a Big-Small-tree must be leaves. Hence, after a single rake stage, the number of high degree vertices drops by a factor of n^{ϵ} .

In order to rake the leaves of high degree vertices, we have to carefully apply our sibling contracting functions in a way that can be implemented efficiently in AMPC. Unlike Theorem 2 in which having access to a connected contracting function is sufficient, here we also require a sibling contracting function. Consider a star tree with its center at the root.

Without a sibling contracting function, we are able to contract at most $O(n^{\epsilon})$ vertices in each round since the components we pass to the contracting functions must be disjoint. But having access to a sibling contracting function, we can rake up to O(n) leaf children of a high degree vertex in $O(1/\epsilon)$ rounds. For more details about the algorithm and proofs see Section 3.2.

▶ Theorem 4. Consider a degree-weighted tree T = (V, E, W) and a problem P. Given a connected contracting function and a sibling contracting function on T with respect to P, one can compute P(T) in $O(1/\epsilon^3)$ AMPC rounds with $O(n^{\epsilon})$ memory per machine and $\widetilde{O}(n)$ total memory.

Theorem 2 and Theorem 4 give us general tools that have the power to create efficient AMPC algorithms for any problem that admits a connected contracting function and a sibling contracting function. Intuitively, they reduce constant-round parallel algorithms for a specific problem on trees to designing black-box contracting functions that are sequential. We should be careful in designing contracting functions to make sure that the amount of data stored in the surviving vertex does not asymptotically exceed its degree in the contracted tree. Also note that a connected contracting function works with unknown values that depend on the result of other components.

Satisfying these conditions is a factor that limits the extent of problems that can be solved using our framework. For example, the framework of Bateni et. al [11] works on a wider range of problems on trees since their algorithm, roughly speaking, tolerates exponential growth of weight vectors using a careful decomposition of tree. Indeed, they achieve these benefits at the cost of an inherent requirement for at least $O(\log n)$ rounds due to the divide-and-conquer nature of their algorithm. However, their framework comes short on addressing problems such as MedianParent (defined in Remark 3) that are not reducible to binary trees. Nonetheless, we show several techniques for designing contracting functions that satisfy these conditions, in particular:

- 1. In Section 3.3, we prove a general approach for designing a connected contracting function and a sibling contracting function given a PRAM algorithm based on the original Miller and Reif [48] tree contraction. We do this by observing that in almost every conventional application of Miller and Reif's framework, the length of data stored at each vertex remains constant throughout the algorithm.
- 2. Storing a minimal tree representation of a connected component contracted into v in the weight vector W(v) enables us to simplify a recursive function defined on the subtree rooted at v in terms of yet-unknown values of its children, while keeping the length of W(v) asymptotically proportional to deg(v). For instance, our maximum weighted matching algorithm (See Section 4.1 of the full version for more details) uses this approach.

Ultimately, this is a highly efficient generalization of the powerful tree contraction algorithm. To illustrate the versatility of our framework, we show that it gives us efficient AMPC algorithms for many important applications of frameworks such as Miller and Reif [49]'s and Bateni et al. [11]'s by constructing sequential black-box contracting functions. In doing so, we utilize a diverse set of techniques, including the ones mentioned above, that are of independent interest and can be applied to a broad range of problems on trees. The proof Theorem 5 and more details about each application can be found in the full version.

▶ **Theorem 5.** Algorithms 1 and 2 can solve, among other applications, dynamic expression evaluation, tree isomorphism testing, maximal matching, and maximal independent set in $O(1/\epsilon^2)$ AMPC rounds, and maximum weighted matching and maximum weighted independent set in $O(1/\epsilon^3)$ AMPC rounds. All algorithms use $O(n^{\epsilon})$ memory per machine and $\widetilde{O}(n)$ total memory.

1.3 Paper Outline

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

2 Preliminaries

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

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

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

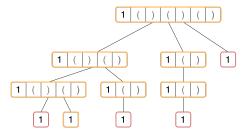
▶ **Definition 6.** A degree-weighted tree is a tree T = (V, E, W) with vertex set V, edge set E, and vertex weight vector function W such that for all $v \in V$, $W(v) \in \{0,1\}^{\widetilde{O}(\deg(v))}$.

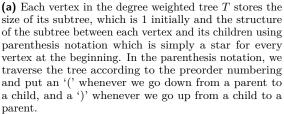
Notationally, we let $w(v) = \dim(W(v)) = \widetilde{O}(\deg(v))$ be the length of the weight vectors. Additionally, note that a tree T = (V, E) is a degree-weighted tree where $W(v) = \emptyset$ for all $v \in V$.

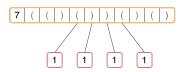
In order to implement our algorithm, we also require specific contracting functions whose properties allow us to achieve the desired result (§2.1). In addition, we will introduce a specific tree decomposition method, called a preorder decomposition, that we will efficiently implement and leverage in our final algorithms (§2.2).

 $^{^{1}}$ lca is the least common ancestor function.

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







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

Figure 1 A degree weighted tree T = (V, E, W) with |V| = 11. In Subfigure 1a, we have a degree weighted tree with $|W_v| \leq 4deg(v)$. We contract the subgraph with 7 vertices depicted by yellow in Subfigure 1a using a connected contracting function (Defined in Definition 8). The resulting degree weighted tree T' is depicted in Subfigure 1b. Note that the length of weight vectors in proportional to the degree of each vertex even after the contraction.

2.1 Tree Contractions and Contracting Functions

Our algorithms provide highly efficient generalizations to Miller and Reif's [49] tree contraction algorithms. At a high level, their framework provides the means to compute a global property with respect to a given tree in $O(\log n)$ phases. In each phase, there are two stages:

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

Repeated application of Compress and Rake alternatively results in a tree which has only one vertex. Intuitively, the COMPRESS stage aims to shorten the long chains, maximal connected sequences of vertices whose degree is equal to 1, and the RAKE stage cleans up the leaves. Both stages are necessary in order to guarantee that $O(\log n)$ phases are enough to end up with a single remaining vertex [49].

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

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

- **Definition 7.** In an α -tree-contraction of a tree T = (V, E), we repeat two stages in a number of phases until the whole tree is contracted into a single vertex:
- Compress stage: Contract every maximal connected component S containing only vertices with degree less than α , i.e., $deg(v) < \alpha \ \forall v \in S$, into a single vertex S'.
- Rake stage: Contract all the leaves into their parent.

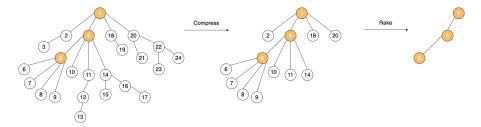


Figure 2 An example phase in α -tree-contraction for $\alpha = 4$. In the leftmost tree the initial tree is depicted, and the vertices are numbered from 1 to n in the preorder ordering. In the middle tree, we performed a Compress stage to get a tree with 12 vertices. Next, we Rake the leaves to end up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

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

In order to implement Compress and Rake, we need fundamental tools for contracting a single set of vertices into each other. We call these *contracting functions*. In the Compress stage, we must contract connected components. In the Rake stage, we must contract leaves with the same parent into a single vertex. These functions run locally on small sets of vertices.

- ▶ **Definition 8.** Let P be some problem on degree-weighted trees such that for some degree-weighted tree T, P(T) is the solution to the problem on T. A **contracting function** on T with respect to P is a function f that replaces a set of vertices in T with a single vertex and incident edges to form a degree-weighted tree T' such that $P(T) = P(T')^3$. There are two types:
- 1. f is a connected contracting function if f contracts⁴ connected components into a single vertex of T.
- 2. f is a sibling contracting function if f is defined on sets of leaf siblings (i.e., leaves that share a parent p) of T, and the new vertex is a leaf child of p.

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

2.2 Preorder Decomposition

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

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

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

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

▶ **Definition 9.** Given a tree T = (V, E), a "preorder decomposition" V_1, V_2, \ldots, V_k of T is defined by a vector $l \in \mathbb{Z}^{k+1}$, such that $0 = l_0 < l_1 < \ldots < l_k = n$, as $V_i = \{l_{i-1} + 1, l_{i-1} + 2, \ldots, l_i\}$ $\forall i \in [k]$. See Subfigure 3a for an example.

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

▶ **Observation 10.** Consider a given tree T = (V, E). For any parameter λ such that $deg(v) \leq \lambda$ for all $v \in V$, there is a preorder decomposition V_1, V_2, \ldots, V_k such that $\forall i \in [k]$, $\sum_{v \in V_i} deg(v) \leq \lambda$, and $k = O(n/\lambda)$.

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

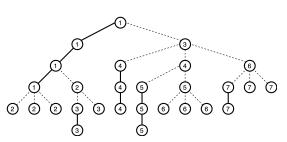
▶ Definition 11. Given a tree T = (V, E) and a decomposition of vertices V_1, V_2, \ldots, V_k , the dependency tree T' = (V', E') of T under this decomposition is constructed by contracting each connected component $C_{i,j}$ for all $j \in [c_i]$ in each group V_i . We call a component contracted to a leaf in T' an independent component, and a component contracted to a non-leaf vertex in T' a dependent component.

3 Constant-round Tree Contractions in AMPC

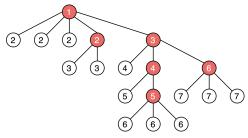
The main results of this paper are two new algorithms. The first algorithm applies α -tree-contraction-like methods in order to solve problems on trees where the degrees are bounded by n^{ϵ} . Though this algorithm is similar in inspiration to the notion of tree contractions, it is not a true α -tree-contraction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

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

This algorithm provides us with two benefits: (1) it is a standalone result that is quite powerful in its own right and (2) it is leveraged in our main algorithm for Theorem 4. The only differences between this result and our main result for generalized tree contractions is that we require $\deg(v) \leq n^{\epsilon}$, but it runs in $O(1/\epsilon^2)$ rounds, as opposed to $O(1/\epsilon^3)$ rounds.



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



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

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

Thus, if the input tree has degree bounded by n^{ϵ} , then clearly the precondition is satisfied. Additionally, if the tree can be decomposed into a tree with bounded degree such that we can still solve the problem on the decomposed tree, this result applies as well.

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

▶ Theorem 4. Consider a degree-weighted tree T = (V, E, W) and a problem P. Given a connected contracting function and a sibling contracting function on T with respect to P, one can compute P(T) in $O(1/\epsilon^3)$ AMPC rounds with $O(n^{\epsilon})$ memory per machine and $\widetilde{O}(n)$ total memory.

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

3.1 Contractions on Degree-Bounded Trees

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

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

Let $T_0 = T$ be the input tree. For every iteration $i \in [O(1/\epsilon)]$: (1) find a preorder decomposition $V_1, \ldots V_k$ of T_{i-1} , (2) contract each connected component in the preorder decomposition, and (3) put each maximal set of leaf-siblings (i.e., leaves that share a parent)

in one machine and contract them into their parent. We sometimes refer to these maximal sets of leaf-siblings by *leaf-stars*. After sufficiently many iterations, this should reduce the problem to a single vertex, and we can simply solve the problem on the vertex.

■ Algorithm 1 BoundedTreeContract

(Computing the solution P(T) of a problem P on degree-weighted tree T with max degree n^{ϵ} using connected contracting function \mathcal{C}).

```
Data: Degree-weighted tree T = (V, E, W) with degree bounded by n^{\epsilon} and a connected contracting function \mathcal{C}.

Result: The problem output P(T).

1 T_0 \leftarrow T;

2 for i \leftarrow 1 to l = O(1/\epsilon) do

3 | Let \mathcal{O} be a preordering of V_{i-1};

4 | Find a preorder decomposition V_1, V_2, \ldots, V_k of T_{i-1} with \lambda = n^{\epsilon} using \mathcal{O};

5 | Let S_{i-1} be the set of all connected components in V_i for all i \in [k];

6 | Let T'_{i-1} be the result of contracting \mathcal{C}(S_{i-1,j}) for all S_{i-1,j} \in S_{i-1};

7 | Let \mathcal{L}_{i-1} be the set of all maximal leaf-stars (containing their parent) in T'_{i-1};

8 | Let T_i be the result of contracting \mathcal{C}(L_{i-1,j}) for all L_{i-1,j} \in \mathcal{L}_{i-1};

9 end

10 return \mathcal{C}(T_l);
```

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

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

▶ Lemma 12. The dependency tree T' = (V', E') of a preorder decomposition V_1, V_2, \ldots, V_k of tree T = (V, E) contains at most 1 non-leaf vertex per group for a total of at most k non-leaf vertices. In other words, there are at most k dependent connected components in $\cup_{i \in [k]} F_i$.

Proof. Each group V_i induces a forest F_i on tree T, and recall that each F_i is consisted of multiple connected components $C_{i,1}, C_{i,2}, \ldots, C_{i,c_i}$, where c_i is the number of connected components of F_i . Assume w.l.o.g. component C_{i,c_i} is the component which contains vertex l_i , the vertex with the largest index in V_i . We show that every connected component in F_i except C_{i,c_i} is independent, and thus Lemma 12 statement is implied. See in Subfigure 3b that there is at most 1 dependent component, red vertices in T', for each group V_i . Also note that C_{i,c_i} , the only possibly dependent component in F_i , is always the last component if we sort the components based on their starting index since $l_i \in C_{i,c_i}$ and each $C_{i,j}$ contains a consecutive set of vertices.

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

Lemma 12 nicely fits with our result from Observation 10 to bound the total number of phases BoundedContract requires. In addition, we can show how to implement each phase to bound the complexity of our algorithm. Note that we are assuming that our component contracting function is defined to always yield a degree-weighted tree. We only need to show that the degrees stay bounded throughout the algorithm.

Proof of Theorem 2. In each phase of this algorithm, the only modifications to the graph that occur are applications of the connected contracting functions to connected components of the tree. Since these are assumed to preserve P(T) and we simply solve $P(T_l)$ for the final tree T_l , correctness of the output is obvious.

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

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

We now focus on round and space complexities. A preordering can be computed using the preorder traversal algorithm from Behnezhad et al. [16], which executed in $O(1/\epsilon)$ rounds with $O(n^{\epsilon})$ local space and $\widetilde{O}(n)$ total space w.h.p.⁵ This completes step 1. In steps 2 and 3, the contracting functions are applied in parallel for a total of O(1) rounds (based off our assumption about any given contracting functions) within the same space constraints. Thus, all phases require O(1) rounds except the first, which is $O(1/\epsilon)$ rounds, and satisfy the space constraints of our theorem.

Now we must count the phases. Lemma 12 tells us that for every group, we only have one non-leaf component in the dependency graph after each step 2. In step 3, we then "RAKE" all leaves into their parents. This means that the remaining number of vertices after step

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

3 is equal to the number of non-leaf vertices in the dependency graph after step 2, which is $k = n^{\epsilon}$. Observation 10 tells us that the resulting graph size is then $O(n/n^{\epsilon}) = O(n^{1-\epsilon})$. Therefore, in order to get a graph where $|T_l| = 1$, we require $O(1/\epsilon)$ phases. Combining this with the complexity of each phase yields the desired result.

3.2 Generalized α -Tree-Contractions

In the rest of this section we prove our main result: a generalized tree contraction algorithm, Algorithm 2. Building on top of Theorem 2, we can create a natural extension of tree contractions. Recall from §2 that in the COMPRESS stage, we must contract maximal connected components containing only vertices v with degree $d(v) < \alpha$. Conveniently, by Theorem 2, Algorithm 1 achieves precisely this. Therefore, to implement tree contractions, we simply need to:

- 1. Identify maximal connected components of low degree (Algorithm 2, line 3), which can be done in $O(1/\epsilon)$ rounds by Behnezhad et al. [18].
- 2. Use our previous algorithm to execute the COMPRESS stage on each component (Algorithm 2, line 5), which can be done by Algorithm 1 in $O(1/\epsilon^2)$ rounds.
- 3. Apply a function that can execute the RAKE stage (Algorithm 2, lines 7 through 14).

To satisfy the third step, we use a *sibling contracting function* (Definition 8), which can contract leaf-siblings of the same parent into a single leaf. Since a vertex might have up to n children, to do this in parallel, we may have to group siblings into n^{ϵ} -sized groups and repeatedly contract until we reach one leaf. Assuming sibling contractions are locally performed inside machines, this will then take $O(1/\epsilon)$ AMPC rounds.

Algorithm 2 TreeContract

(Computing the solution P(T) of a problem P on degree-weighted tree T using a connected contracting function C and a sibling contracting function R).

```
Data: Degree-weighted tree T = (V, E, W), a connected contracting function \mathcal{C}, and
               a sibling contracting function \mathcal{R}.
    Result: The problem output P(T).
 1 T_0 \leftarrow T;
 2 for i \leftarrow 1 to l = O(1/\epsilon) do
         Let S_{i-1} \leftarrow \mathsf{Connectivity}(T_{i-1} \setminus \{v \in V : deg(v) > n^{\epsilon});
 3
         Let \mathcal{K}_{i-1} \leftarrow \text{Components in } \mathcal{S}_{i-1,j} \text{ which represent a leaf in } T'_{i-1};
 4
         Contract each S_{i-1,j} \in \mathcal{K}_{i-1} into S'_{i-1,j} by applying BoundedTreeContract(S_j, \mathcal{C});
 5
         Let \mathcal{L}_{i-1} be the set of all maximal leaf-stars (excluding their parent) in T'_{i-1};
 6
         for L_{i-1,0} = \{v_1, v_2, \dots, v_k\} \in \mathcal{L}_{i-1} do
 7
               for j \leftarrow 1 to 1/\epsilon do
 8
                   Split L_{i-1,j-1} into k/n^{j\epsilon} parts L_{i-1,j,1},\ldots,L_{i-1,j,k/n^{j\epsilon}} each of size n^{\epsilon};
  9
                    Contract each L_{i-1,j,z} into L'_{i-1,j,z} by applying \mathcal{R}(L_{i-1,j,z});
10
                   Let L_{i-1,j} \leftarrow \{L'_{i-1,j,1}, \dots, L'_{i-1,j,k/n^{j\epsilon}}\};
11
12
               Contract L_{i-1,1/\epsilon} by applying \mathcal{R}(L_{i-1,1/\epsilon});
13
         end
14
         Let T_i \leftarrow T'_{i-1};
15
16 end
17 return C(T_l);
```

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

▶ Theorem 4. Consider a degree-weighted tree T = (V, E, W) and a problem P. Given a connected contracting function and a sibling contracting function on T with respect to P, one can compute P(T) in $O(1/\epsilon^3)$ AMPC rounds with $O(n^{\epsilon})$ memory per machine and $\widetilde{O}(n)$ total memory.

Recall the definition of α -tree-contraction (Definition 7) from §2.1. First, we prove Lemma 13 to bound the number of phases in α -tree-contraction.

▶ **Lemma 13.** For any $\alpha \geq 2$, the number of α -tree-contraction phases until we have a constant number of vertices is bounded by $O(\log_{\alpha}(n))$.

To show Lemma 13 we will introduce a few definitions. The first definition we use is a useful way to represent the resulting tree after each COMPRESS stage. Before stating the definition, recall the *Dependency Tree* T' of a tree T from Definition 11.

▶ Definition 14. An α -Big-Small Tree T' is the dependency tree of a tree T with weighted vertices if it is a minor of T constructed by contracting all components of T made up of low vertices v with $deg(v) < \alpha$ (i.e., the connected components of T if we were to simply remove all vertices v with $deg(v) \geq \alpha$) into a single node.

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

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

▶ **Observation 15.** No small component in an α -Big-Small tree can be the parent of another small component.

Consider our dependency tree T' based off a tree T that has been compressed. Obviously, T' is a minor of T constructed as described for α -Big-Small Tree because the weight of a vertex equals its number of children (by the assumption of Lemma 13). Note a small component refers to the compressed components, and a big node refers to nodes that were left uncompressed.

To show Lemma 13, we start by proving that the ratio of leaves to nodes in T' is large. Since Rake removes all of these leaves, this shows that T gets significantly smaller at each step. Showing that the graph shrinks sufficiently at each phase will ultimately give us that the algorithm terminates in a small number of phases.

▶ Lemma 16. Let T'_i be the tree at the end of phase i. Then the fraction of nodes that are leaves in T'_i is at least $\alpha/(\alpha+4)$ as long as w(v) is equal to the number of children of v for all $v \in T'_i$ and $\alpha \geq 2$.

Proof. For our tree T_i' , we will call the number of nodes n, the number of leaves ℓ , and the number of big nodes b. We want to show that $\ell > n\alpha/(\alpha+4)$. We induct on b. When b=0, we can have one small component in our tree, but no others can be added by Observation 15. Then $n=\ell=1$, so $\ell > n\alpha/(\alpha+4)$.

Now consider T_i' has some arbitrary b number of big nodes. Since T_i' is a tree, there must be some big node v that has no big node descendants. Since all of its children must be small components and they cannot have big node descendants transitively, then Observation 15 tells us each child of v is a leaf. Note that since v is a big node, it must have weight $w(v) > \alpha$, which also means it must have at least α children (who are all leaves) by the assumption that w(v) is equal to the number of children.

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

$$\ell = \ell^* - 1 + w(v)$$

$$= \frac{\alpha}{\alpha + 4} (n - w(v) - 1) - 1 + w(v)$$

$$= \frac{\alpha}{\alpha + 4} n - \frac{\alpha}{\alpha + 4} w(v) - \frac{\alpha}{\alpha + 4} - 1 + w(v)$$

$$= \frac{\alpha}{\alpha + 4} n + \frac{4}{\alpha + 4} w(v) - \frac{2\alpha + 4}{\alpha + 4}$$

$$> \frac{\alpha}{\alpha + 4} n + \frac{4}{\alpha + 4} \alpha - \frac{2\alpha + 4}{\alpha + 4}$$

$$\geq \frac{\alpha}{\alpha + 4} n$$

$$\geq \frac{\alpha}{\alpha + 4} n$$
(1)
$$\geq \frac{\alpha}{\alpha + 4} n$$

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

Now we can prove our lemma.

Proof of Lemma 13. To show this, we will prove that the number of nodes from the start of one Compress to the next is reduced significantly. Consider T_i as the tree before the ith Compress and T_i' as the tree just after. Let T_{i+1} be the tree just before the i+1st Compress, and let n_i be the number of nodes in T_i , n_i' be the number of nodes in T_i' , and n_{i+1} be the number of nodes in T_{i+1} . Since T_i' is a minor of T_i , it must have at most the same number of vertices as T_i , so $n_i' \leq n_i$. Since T_{i+1} is formed by applying RAKE to T_i' , then it must have the number of nodes in T_i' minus the number of leaves in T_i' (ℓ_i'). Therefore:

$$n_{i+1} = n'_i - \ell'_i \le n'_i - \frac{\alpha}{\alpha + 4} n'_i = \frac{4}{\alpha + 4} n'_i \le \frac{4}{\alpha + 4} n_i$$

Where we apply both Lemma 16 that says $\ell_i' \geq \frac{\alpha}{\alpha+4} n_i'$ and the fact that we just showed that $n_i' \leq n_i$. This shows that from the start of one compress phase to another, the number of vertices reduces by a factor of $\frac{4}{\alpha+4}$. Therefore, to get to a constant number of vertices, we require $\log_{\frac{\alpha+4}{4}}(n) = O(\log_{\alpha}(n))$ phases.

Now we are ready to prove our main theorem.

▶ Theorem 4. Consider a degree-weighted tree T = (V, E, W) and a problem P. Given a connected contracting function and a sibling contracting function on T with respect to P, one can compute P(T) in $O(1/\epsilon^3)$ AMPC rounds with $O(n^{\epsilon})$ memory per machine and $\widetilde{O}(n)$ total memory.

Proof. We will show that our Algorithm 2 achieves this result. Lemma 13 shows that there will be only at most $O(1/\epsilon)$ phases. In each phase i, we start by running a connectivity algorithm to find maximally connected components of bounded degree, which takes $O(1/\epsilon)$ time. Let \mathcal{K}_{i-1} be the set of connected components which are leaves in T'_{i-1} . Then for each component $S_{i-1,j} \in \mathcal{K}_{i-1}$, we run BoundedTreeContract (Algorithm 1) in parallel using only our connected contracting function \mathcal{C} . Since the total degree of vertices over all members of \mathcal{K}_{i-1} is not larger than $|T_{i-1}|$ and the amount of memory required for storing a degree-weighted trees is not larger than the total degree, the total number of machines is bounded above by $O(n^{1-\epsilon})$. By definition, the maximum degree of any $S_{i-1,j}$ is n^{ϵ} . By Theorem 2, each instance of BoundedTreeContract requires $O(1/\epsilon^2)$ rounds, $O(|S_{i-1,j}|^{\epsilon})$ local memory and $\widetilde{O}(|S_{i-1,j}|)$ total memory. As $|S_{i-1,j}| \leq |T_{i-1}|$ (we know $|T_0| = n$ and it only decreases over time), we only require at most $O(n^{\epsilon})$ memory per machine. Since Since the total degree of vertices over all members of \mathcal{K}_{i-1} is not larger than $|T_{i-1}|$, the total memory required is only $\widetilde{O}(|T_{i-1}|) = \widetilde{O}(n)$. This is within the desired total memory constraints.

Finally, \mathcal{R} is given to us as a sibling contractor. Consider the RAKE stage in our algorithm. We distribute machines across maximal leaf-stars. For any leaf-star with $n^{\epsilon} \leq deg(v) \leq kn^{\epsilon}$ for some (possibly not constant) k, we will allocate k machines to that vertex. Since again the number of vertices is bounded above by n, this requires only $O(n^{1-\epsilon})$ machines. On each machine, we allocate up to n^{ϵ} leaf-children to contract into each other. We can then contract siblings into single vertices using \mathcal{R} . Since there are at most n children for a single vertex, it takes at most $O(1/\epsilon)$ rounds to contract all siblings into each other. Then, finally, we can use \mathcal{C} to compress the single child into its parent, which takes constant time.

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

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

Due to Theorem 4, we can compute any P(T) on trees as long as we are provided with a connected contracting function and a sibling contracting function with respect to P. A natural question that arises is the following: for which class of problem P there exists black-box contracting functions? We argue that many problems P for which we have a 2-tree-contraction algorithm can also be computed in $O(1/\epsilon^3)$ AMPC rounds using n^{ϵ} -tree-contraction.

In many problems which are efficiently implementable in the Miller and Reif [49] Tree Contraction framework, we are given \mathcal{C} and \mathcal{R} contracting functions, for COMPRESS and RAKE stages respectively, which contract only one node: either a leaf in case of RAKE or a vertex with only one child in case of COMPRESS. Let us call this kind of contracting functions unary contracting functions and denote them by \mathcal{C}^1 and \mathcal{R}^1 . This is a key point of original variants of Tree Contraction which contract odd-indexed vertices, or contract a maximal independent set of randomly selected vertices. Working efficiently regardless of using only unary contracting functions is the reason Tree Contraction was considered a fundamental framework for designing parallel algorithms on trees in more restricted models such as PRAM. For example, in the EREW variant of PRAM, an $O(\log(n))$ rounds tree contraction requires to use only unary contracting functions \mathcal{R}^1 and \mathcal{C}^1 . More generally, we define i-ary contracting functions as follows.

▶ Definition 17. An "i-ary contracting function", denoted by C^i or \mathcal{R}^i , is a contracting function which admits a subset $S = \{v_1, v_2, \ldots, v_k\}$ of at most i+1 vertices at a time such that $\sum_{j=1}^k deg(v_j) = O(i)$. A special case of i-ary contracting functions, are "unary contracting functions", denoted by C^1 or \mathcal{R}^1 , which contract only one vertex at a time.

However, in the the AMPC model, we can contract the chains more efficiently, and thus we are allowed to utilize more relaxed variants of COMPRESS stage. Furthermore, as we show in Theorem 18, designing unary contracting functions \mathcal{C}^1 and \mathcal{R}^1 is not easier than designing *i-ary* contracting functions \mathcal{C}^i and \mathcal{R}^i in the AMPC model. We show this by reducing \mathcal{C}^i and \mathcal{R}^i to \mathcal{C}^1 and \mathcal{R}^1 in O(1) rounds for any $i = O(n^{\epsilon})$. In other words, the restrictions of PRAM model, which requires \mathcal{C}^1 and \mathcal{R}^1 exclusively, enables us to directly translate a vast literature of problems solved using tree contraction to efficient AMPC algorithms for the same problem given \mathcal{C}^1 and \mathcal{R}^1 .

As we have shown in Theorem 4, it is possible to solve any problem P(T) in $O(1/\epsilon^3)$ AMPC rounds given a connected contracting function \mathcal{C} and a sibling contracting function \mathcal{R} , where both are n^{ϵ} -ary contracting with respect to P. In what follows, we demonstrate the construction of n^{ϵ} -ary contracting functions given a unary connected contracting function \mathcal{C}^1 and a sibling contracting function \mathcal{R}^1 .

▶ Theorem 18. Given a unary connected contracting function C^1 and a unary sibling contracting function \mathcal{R}^1 with respect to a problem P defined on trees, one can build an i-ary connected contracting function C^i and an i-ary sibling contracting function \mathcal{R}^i with respect to P and both C^i and \mathcal{R}^i run in one AMPC rounds as long as $i = O(n^{\epsilon})$.

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

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

We run a 2-tree-contraction-like algorithm locally on $T^{\mathcal{C}}(v_1)$ using \mathcal{C}^1 and \mathcal{R}^1 . Note that we can only rake the known leaves since the data of unknown leaves depend on their children. We repeating COMPRESS and RAKE stages until there is no known leaf or a vertex with one child remain in $T^{\mathcal{C}}(v_1)$. Then, according to Lemma 13 for $\alpha = 2$, the number of remaining vertices in $T^{\mathcal{C}}(v_1)$ is bounded by O(l). We store the final $T^{\mathcal{C}}(v_1)$ in T' which requires a memory of $\widetilde{O}(l) = \widetilde{O}(w_{T'}(v_1))$. Hence, \mathcal{C}^i satisfies the size-constraint on the weight vectors of the resulting weighted-degree tree.

Finally, we present an algorithm for \mathcal{R}^i which is more straight-forward compared to that of \mathcal{C}^i . We are given a leaf-star $S = \{v_1, v_2, \ldots, v_k\}$ of T so that $\sum_{j=1}^k deg_T(v_j) = O(i)$. This implies that there are at most $O(n^\epsilon)$ vertices in S as long as $i = O(n^\epsilon)$, and we can fit the whole S into a memory of a single machine. To simulate \mathcal{R}^i , we only need to k-1 times apply \mathcal{R}^1 on $S_j = \{v_1, v_{j+1}\}$ at the j-th iteration. Note that every $v_j \in S$ is a leaf in T, so the data stored in $W(v_j)$ is just $\widetilde{O}(1)$ bits and not a tree structure. Theorem 18 statement is implied.

3.4 Reconstructing the Tree for Linear-sized Output Problems

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

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

Roughly speaking, our reconstruction algorithm is based on storing the information about components we contracted throughout the algorithm in an auxiliary memory of size O(n). It is easy to observe that if we store the degree-weighted subtree of every connected component or leaf-star that we contract during the algorithm we need at most O(n) addition total memory. Note that during each application of black-box contracting functions, we remove at least one vertex from the tree and each vertex except root is removed exactly once when the algorithm terminates. Namely, for every phase i we need to store S_i and L_i in Algorithm 1, and K_i and every $L_{i,j,z}$ in Algorithm 2 (In addition to the data stored by each black-box application of Algorithm 1). Since we have adaptive access to these subsets in AMPC, it is sufficient to index them by the id of the surviving vertex of each subset.

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

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

4 Conclusion

This paper introduces some of the first generalized techniques for solving various problems in the AMPC model. Specifically, we show that Miller and Reif's [48] $O(\log n)$ -time PRAM tree contraction algorithm can be efficiently extended to a constant-round low-memory AMPC algorithm. This implies $O(1/\epsilon^2)$ -round algorithms for expression evaluation, tree isomorphism

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

testing, maximal matching on trees, and maximal independent set on trees. It additionally implies $O(1/\epsilon^3)$ -round algorithms for maximum weighted matching and maximum weighted independent set on trees. However, we expect these algorithms to have much broader applications to tree-based problems, as did the original work by Miller and Reif.

It remains to be seen precisely which of extensions of the PRAM algorithm apply to the AMPC model. Many of them require computational overhead beyond the black-box application of the tree contraction process (which our algorithm can directly and efficiently simulate), therefore the extension of those applications to this work is highly non-trivial. Of notable interest is the application of tree contractions to graphs of bounded treewidth, where Bodlaender and Hagerup [21] showed how to construct low-width tree decompositions using PRAM tree contractions. If AMPC tree contractions can also solve this and additionally solve tree contractions on graphs of bounded tree-width, then our work can be notably generalized. Another potential course of research is the exploration of problems such as MedianParent, which cannot be simplified to problems on trees with bounded tree width. It is an open question if these, too can be solved in $O(1/\epsilon^2)$ rounds.

References

- 1 Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Shan Leung Maverick Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete, pages 531–540, 2004
- 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 on Symposium on Parallelism in Algorithms*, pages 202–211, 2015.
- 3 Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Symposium on Theory of Computing*, pages 574–583. ACM, 2014.
- 4 Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In 59th IEEE Annual Symposium on Foundations of Computer Science, pages 674–685. IEEE Computer Society, 2018.
- 5 Alexandr Andoni, Clifford Stein, and Peilin Zhong. Log diameter rounds algorithms for 2-vertex and 2-edge connectivity. In 46th International Colloquium on Automata, Languages, and Programming, pages 14:1–14:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- 6 Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab S. Mirrokni, and Cliff Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete, pages 1616–1635, 2019.
- 7 Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear algorithms for δ + 1 vertex coloring. In Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete, pages 767–786, 2019
- 8 Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively parallel algorithms for finding well-connected components in sparse graphs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed*, pages 461–470, 2019.
- 9 Mikhail J. Atallah, S. Rao Kosaraju, Lawrence L. Larmore, Gary L. Miller, and Shang-Hua Teng. Constructing trees in parallel. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 1989.
- MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab S. Mirrokni. Affinity clustering: Hierarchical clustering at scale. In Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems, pages 6864–6874, 2017.

- 11 MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Vahab S. Mirrokni. Brief announcement: Mapreduce algorithms for massive trees. In 45th International Colloquium on Automata, Languages, and Programming, pages 162:1–162:4, 2018.
- 12 Mohammad Hossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, Mohammad Taghi Hajiaghayi, and Vahab S. Mirrokni. Massively parallel dynamic programming on trees. CoRR, 2018.
- Soheil Behnezhad. Time-optimal sublinear algorithms for matching and vertex cover. arXiv preprint arXiv:2106.02942, 2021.
- Soheil Behnezhad, Sebastian Brandt, Mahsa Derakhshan, Manuela Fischer, Mohammad Taghi Hajiaghayi, Richard M Karp, and Jara Uitto. Massively parallel computation of matching and mis in sparse graphs. In ACM SIGACT, pages 481–490, 2019.
- Soheil Behnezhad, Mahsa Derakhshan, Mohammad Taghi Hajiaghayi, Marina Knittel, and Hamed Saleh. Streaming and massively parallel algorithms for edge coloring. In 27th Annual European Symposium on Algorithms, pages 15:1–15:14, 2019.
- Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Łacki, Vahab Mirrokni, and Warren Schudy. Massively parallel computation via remote memory access. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 59–68, 2019.
- 17 Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab S. Mirrokni. Near-optimal massively parallel graph connectivity. In 60th IEEE Annual Symposium on Foundations of Computer Science, pages 1615–1636, 2019.
- 18 Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, Vahab S. Mirrokni, and Warren Schudy. Parallel graph algorithms in constant adaptive rounds: Theory meets practice. *Proc. VLDB Endow.*, pages 3588–3602, 2020.
- Soheil Behnezhad, Mohammad Taghi Hajiaghayi, and David G. Harris. Exponentially faster massively parallel maximal matching. In 60th IEEE Annual Symposium on Foundations of Computer Science, pages 1637–1649, 2019.
- 20 Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. SIAM J. Comput., pages 317–378, 2016.
- 21 Hans L. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. In Zoltán Fülöp and Ferenc Gécseg, editors, Automata, Languages and Programming, 22nd International Colloquium, ICALP95, Szeged, Hungary, July 10-14, 1995, Proceedings, volume 944 of Lecture Notes in Computer Science, pages 268–279. Springer, 1995.
- 22 Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, Mohammad Taghi Hajiaghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and mapreduce. In Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete, pages 1170–1189, 2018.
- 23 Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In Benjamin G. Zorn and Alexander Aiken, editors, Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 363–375, 2010.
- Moses Charikar, Weiyun Ma, and Li-Yang Tan. Unconditional lower bounds for adaptive massively parallel computation. In SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, pages 141–151, 2020.
- Richard Cole and Uzi Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, pages 329–346, 1988.
- 26 Artur Czumaj, Jakub Lacki, Aleksander Madry, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. In *Proceedings of the* 50th Annual ACM SIGACT Symposium on Theory, pages 471–484, 2018.

- 27 Eliezer Dekel, Simeon Ntafos, and Shie-Tung Peng. Parallel tree techniques and code optimization. In Filia Makedon, Kurt Mehlhorn, T. Papatheodorou, and P. Spirakis, editors, VLSI Algorithms and Architectures, pages 205–216, 1986.
- 28 Apache Software Foundation. *Hadoop*. https://hadoop.apache.org/.
- 29 H. Gazit, Gary L. Miller, and ShangHua Teng. Optimal tree contraction in an EREW model. In Concurrent Computations: Algorithms, Architecture and Technology, pages 139–156, 1988.
- 30 Hillel Gazit and Gary L. Miller. A parallel algorithm for finding a separator in planar graphs. In 28th Annual Symposium on Foundations of Computer Science, pages 238–248, 1987.
- 31 Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrovic, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for mis, matching, and vertex cover. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed*, pages 129–138, 2018.
- 32 Mohsen Ghaffari, Christoph Grunau, and Ce Jin. Improved mpc algorithms for mis, matching, and coloring on trees and beyond. arXiv preprint arXiv:2002.09610, 2020.
- 33 Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. In 60th IEEE Annual Symposium on Foundations of Computer Science, pages 1650–1663, 2019.
- 34 Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In SODA, pages 1636–1653. SIAM, 2019.
- 35 Alan Gibbons and Wojciech Rytter. Optimal parallel algorithms for dynamic expression evaluation and context-free recognition. *Inf. Comput.*, pages 32–45, 1989.
- 36 Michael T. Goodrich and S. Rao Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. *J. ACM*, pages 331–361, 1996.
- 37 Martin Grohe and Oleg Verbitsky. Testing graph isomorphism in parallel by playing a game. In Automata, Languages and Programming, 33rd International Colloquium, pages 3–14, 2006.
- 38 Mohammad Taghi Hajiaghayi and Marina Knittel. Matching affinity clustering: Improved hierarchical clustering at scale with guarantees. In *Proceedings of the 19th International Conference on Autonomous Agents*, pages 1864–1866, 2020.
- 39 Mohammad Taghi Hajiaghayi, Hamed Saleh, Saeed Seddighin, and Xiaorui Sun. String matching with wildcards in the massively parallel computation model. In *SPAA*, pages 275–284, 2021.
- 40 Mohammad Taghi Hajiaghayi, Saeed Seddighin, and Xiaorui Sun. Massively parallel approximation algorithms for edit distance and longest common subsequence. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1654–1672. SIAM, 2019.
- Nicholas J. A. Harvey, Christopher Liaw, and Paul Liu. Greedy and local ratio algorithms in the mapreduce model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms*, pages 43–52, 2018.
- 42 Artur Jez and Markus Lohrey. Approximation of smallest linear tree grammar. *Inf. Comput.*, pages 215–251, 2016.
- 43 Howard J. 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, 2010.
- 44 Raimondas Kiveris, Silvio Lattanzi, Vahab S. Mirrokni, Vibhor Rastogi, and Sergei Vassilvitskii. Connected components in mapreduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 18:1–18:13, 2014.
- 45 Jakub Lacki, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Walking randomly, massively, and efficiently. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory*, pages 364–377, 2020.
- 46 Gary L. Miller and Vijaya Ramachandran. A new graph triconnectivity algorithm and its parallelization. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 335–344, 1987.

- 47 Gary L. Miller, Vijaya Ramachandran, and Erich Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. SIAM J. Comput., pages 687–695, 1988.
- 48 Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In 26th Annual Symposium on Foundations of Computer Science, pages 478–489, 1985.
- 49 Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. Adv. Comput. Res., pages 47–72, 1989.
- 50 Gary L. Miller and John H. Reif. Parallel tree contraction, part 2: Further applications. SIAM J. Comput., pages 1128–1147, 1991.
- Danupon Nanongkai and Michele Scquizzato. Equivalence classes and conditional hardness in massively parallel fcomputations. In 23rd International Conference on Principles of Distributed Systems, pages 33:1–33:16, 2019.
- 52 Dimitrios Papadopoulos, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Practical authenticated pattern matching with optimal proof size. *Proc. VLDB Endow.*, pages 750–761, 2015.
- 53 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*, pages 1–12, 2016.
- Grigory Yaroslavtsev and Adithya Vadapalli. Massively parallel algorithms and hardness for single-linkage clustering under ℓ_p distances. In *Proceedings of the 35th International Conference on Machine Learning*, pages 5596–5605, 2018.
- Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. Commun. ACM, pages 56–65, 2016.