Improved MPC Algorithms for MIS, Matching, and Coloring on Trees and Beyond

Mohsen Ghaffari
ETH Zürich, Switzerland
ghaffari@inf.ethz.ch

Christoph Grunau
ETH Zürich, Switzerland
cgrunau@student.ethz.ch

Ce Jin
Tsinghua University, Beijing, China
cejin@mit.edu

── Abstract ──

We present $O(\log \log n)$ round scalable Massively Parallel Computation algorithms for maximal independent set and maximal matching, in trees and more generally graphs of bounded arboricity, as well as for coloring trees with a constant number of colors. Following the standards, by a *scalable* MPC algorithm, we mean that these algorithms can work on machines that have capacity/memory as small as n^δ for any positive constant $\delta < 1$. Our results improve over the $O(\log^2 \log n)$ round algorithms of Behnezhad et al. [PODC'19]. Moreover, our matching algorithm is presumably optimal as its bound matches an $\Omega(\log \log n)$ conditional lower bound of Ghaffari, Kuhn, and Uitto [FOCS'19].

1 Introduction and Related Work

We present improved algorithms for some of the central graph problems in the study of large-scale algorithms – namely maximal matching and matching approximation, maximal independent set, and graph coloring – in the Massively Parallel Computation (MPC) setting. We first review the related model and known results, and then we present our contributions.

1.1 Massively Parallel Computation

The MPC model was introduced by Karloff et al. [25], and after some refinements [23, 8], it has by now become the standard theoretical abstraction for studying algorithmic aspects of large-scale data processing. This model captures the commonalities of popular practical frameworks such as MapReduce [18], Hadoop [33], Spark [34] and Dryad [24].

In the particular case of MPC for graph problems, we assume that our n-node m-edge input graph is partitioned arbitrarily among a number of machines, each with memory S. This *local memory* S is assumed to be considerably smaller than the entire graph size. Over all the machines, the summation of memories (called the *global memory*) should be at least $\Omega(m + n)$, and often assumed to be at most $\widetilde{O}(m + n)$. The machines communicate in synchronous message-passing rounds, subject to only one constraint: per round, each

machine can only send or receive a data of size at most S, which is all that it could fit in its memory. The objective is to solve various graph problems in the fewest possible number of rounds.

As the graph sizes are getting larger and larger (at a pace that exceeds that of single computers), we would like to have algorithms where each machine's local memory is much smaller than the entire graph size. However, dealing with smaller local memory is known to increase the difficulty of the problem, as we soon review in state-of-the-art. Based on this, the state-of-the-art in MPC algorithms can be divided into three regimes, depending on how the local memory S compares with the number of vertices n: (A) The strongly super-linear memory regime where $S \geq n^{1+\delta}$ for a constant $\delta > 0$, (B) The near-linear memory regime where $S = n \operatorname{poly}(\log n)$, and (C) the strongly sublinear memory regime where $S \leq n^\delta$ for a positive constant $\delta < 1$. We note that algorithms in the last category are the most desirable ones and they are usually referred to as *scalable MPC algorithms.*

1.2 State-of-the-Art

For the problems under consideration in this paper, simple $O(\log n)$ time algorithms follow from classic literature in the LOCAL model and PRAM [30, 1]. The primary objective in MPC is to obtain significantly faster algorithms than the PRAM counterpart, ideally just constant rounds or $O(\log \log n)$ rounds. Over the past few years, there has been progress toward this, gradually moving toward the lower memory regimes.

For the strongly superlinear memory regime, Lattanzi et al. [27] presented $O(1)$-round algorithms for maximal matching and maximal independent set. Progress to lower memory regimes was slow afterward. But that changed with a breakthrough of Czumaj et al. [17], who gave an $O(\log^2 \log n)$ round algorithm for $(2 + \varepsilon)$-approximation of maximum matching in the near-linear memory regime. Shortly after, for the same near-linear memory regime, Assadi et al. [3] gave an $O(\log \log n)$ round algorithm for $(1 + \varepsilon)$-approximation of maximum matching and constant approximation of minimum vertex cover and independently Ghaffari et al. [20] gave $O(\log \log n)$ round algorithms for $(1+\varepsilon)$-approximation of maximum matching, $(2 + \varepsilon)$-approximation of minimum vertex cover, and maximal independent set. Finally, Behnezhad et al. [12] gave an $O(\log \log n)$ round algorithm for maximal matching.

For the much more stringent strongly sublinear memory regime, there has also been some progress but much slower: For the case of general graphs, Ghaffari and Uitto [22] provide an $\widetilde{O}(\sqrt{\log \Delta} + \log \log n)$ algorithm for graphs with maximum degree Δ, which remains the best known. When $\Delta > n^{\Omega(1)}$, the only known algorithms that outperform this bound and reach the $\operatorname{poly}(\log \log n)$ regime are results for special graph families, namely trees [14, 6, 7] and more generally graphs of small, e.g., polylogarithmic, arboricity [13, 10, 9]. Recall that the *arboricity* α of a graph is the minimum number of forests into which its edges can be partitioned. Many natural graph classes, such as planar graphs and graphs with bounded treewidth, have small arboricity. Concretely, Behnezhad et al. [9] provide Maximal Matching and MIS algorithms that run in $O((\log \log n)^2)$ rounds and leave remaining graphs with maximum degree $\operatorname{poly}(\max(\log n, \alpha))$. By invoking the aforementioned algorithm of Ghaffari and Uitto [22], they obtain maximal matching and MIS algorithms that have round complexity $O(\sqrt{\log \alpha} \log \log \alpha + (\log \log n)^2)$. Finally, a result of Ghaffari, Kuhn, and Uitto [21] shows a conditional lower bound (for *component-stable* MPC algorithms[1]) of $\Omega(\log \log n)$ for maximal

---

matching and MIS in this strongly sublinear memory regime. In essence, they show that component-stable MPC algorithms in the strongly sublinear memory regime cannot be more than exponentially faster compared to their LOCAL-model counterpart, and then they use lower bounds in the LOCAL model. Their result is conditioned on the widely believed $\Omega(\log n)$ complexity of the connectivity problem in the strongly sublinear memory regime. We note that although an $\Omega(\log n)$ round complexity lower bound is widely believed to be true for the connectivity problem in the strongly sublinear memory regime, even for distinguishing an n-node cycle from two $n/2$-node cycles, proving any super-constant lower bound would imply $\mathsf{P} \not\subset \mathsf{NC}^1$, a major breakthrough in circuit complexity [32]. Concretely, in the case of maximal matching and MIS, the results of Ghaffari et al. [21] invoke the $\Omega(\sqrt{\log n/\log \log n})$ round LOCAL-model lower bound of Kuhn, Moscibroda, and Wattenhofer [26] to show that an $o(\log \log n)$-round MPC algorithm for maximal matching or MIS would imply an $o(\log n)$ round algorithm for connectivity, which would break the conjectured $\Omega(\log n)$ complexity. We emphasize that in the case of maximal matching, this $\Omega(\log \log n)$ round complexity lower bound for MPC algorithms holds even if the input graph is a tree.

1.3 Our Contribution

Our main contribution is an improvement over the algorithm of Behnezhad et al. [9]. We obtain MPC algorithms in the strongly sublinear memory regime that solve maximal matching and MIS in $O(\log \log n)$ rounds, in trees and constant-arboricity graphs. And more generally as a function of the arboricity α, our maximal matching and MIS algorithms run in $O(\sqrt{\log \alpha} \log \log \alpha + \log \log n)$ rounds, where the first term again comes from invoking the algorithm of Ghaffari and Uitto [22] for remaining graphs with maximum degree poly(α).

▶ **Theorem 1.** *There is an $O(\sqrt{\log \alpha} \log \log \alpha + \log \log n)$ round MPC algorithm using n^δ local memory and $\widetilde{O}(n + m)$ global memory that with high probability computes a maximal matching and a maximal independent set of a graph with arboricity α.*

We note that on graphs with polylogarithmic arboricity, our bound provides a quadratic improvement over the previous algorithms [10, 13, 9]. Moreover, our algorithm for maximal matching is presumably optimal, as the $\Omega(\log \log n)$ conditional lower bound shown by Ghaffari, Kuhn, and Uitto [21] holds even on trees (where $\alpha = 1$).

Our second contribution is providing a similarly fast algorithm for 4-coloring trees.

▶ **Theorem 2.** *There is an $O(\log \log n)$ round MPC algorithm using n^δ local memory and $O(n + m)$ global memory that with high probability computes a 4-coloring of any tree.*

This algorithm also matches a conditional lower bound of $\Omega(\log \log n)$, which follows from the conditional impossibility results of [21], showing that strongly sublinear memory MPC algorithms cannot be more than exponentially fast compared to their LOCAL model counterpart, and the $\Omega(\log n)$ LOCAL-model lower bound for $O(1)$-coloring trees of Linial [29].

1.4 Preliminaries and Notations

Let $[n]$ denote the set $\{1, 2, \ldots, n\}$. Let $G = (V, E)$ be an undirected graph. Let $\deg_G(u)$ denote the degree of a vertex u in graph G, and let $\mathrm{dis}_G(u, v)$ denote the minimum number of edges in a path connecting u and v (when G is clear from the context, we may also

---

matching using strongly sublinear local memory.

write $\deg(u)$ and $\mathrm{dis}(u, v)$ for simplicity). For a vertex subset $V' \subseteq V$, define $\mathrm{dis}_G(u, V') = \min_{v \in V'} \mathrm{dis}_G(u, v)$. The *k-hop neighborhood* of a vertex v in G is the subgraph containing all vertices u satisfying $\mathrm{dis}_G(u, v) \le k$, together with every edge that lies on some path starting from v with length at most k. The k-hop neighborhood of a vertex set $V' \subseteq V$ is the union of k-hop neighborhoods of $v \in V'$. We use $G[V']$ to denote the subgraph of G induced by the vertex subset $V' \subseteq V$. For simplicity we use $\mathrm{dis}_{V'}(u, v)$ as a shorthand for $\mathrm{dis}_{G[V']}(u, v)$, when the underlying graph G is clear from the context; similarly, we can define $\deg_{V'}(u)$.

The *arboricity* α of a graph is the minimum number of forests into which its edges can be partitioned. Nash-Williams [31] showed that the arboricity is equal to the maximum value of $\lceil m_S / (n_S - 1) \rceil$, where m_S, n_S are the number of edges and vertices in any subgraph S.

2   MIS and Matching

In this section we prove Theorem 1. In Section 2.1, we review the previous work by Behnezhad et al. and informally describe our new techniques for improving their algorithm. In Section 2.2, we give formal definitions and describe the structure of our pipelined MPC algorithm. Then in Section 2.3 and Section 2.4, we present the implementation and loop invariants of our algorithm in detail. In Section 2.5, we analyze the space complexity of the MPC algorithm.

2.1   Overview

Review of Behnezhad et al.'s algorithm.   We first briefly review the main ideas of the previous MPC algorithm for Maximal Independent Set (MIS) and Maximal Matching (MM) by Behnezhad et al. [9, 10]. Their result is based on the LOCAL algorithm due to Barenboim et al. [5]. This LOCAL algorithm is formally stated in Lemma 3. In a graph with small arboricity α and large maximum degree $\Delta \ge \mathrm{poly}(\alpha, \log n)$, the LOCAL algorithm finds a matching (or an independent set) in $O(1)$ rounds and removes the involved vertices from the graph, so that the number of high-degree vertices (i.e., with degree $> \sqrt{\Delta}$) decreases by a factor of Δ in the remaining graph.

By repeating this LOCAL algorithm $O(\log_\Delta n)$ times, all high-degree vertices are eliminated, that is, they either get removed from the graph or become low-degree. Thus, the remaining graph has maximum degree at most $\sqrt{\Delta}$. We call this procedure a *phase*. After $O(\log \log \Delta)$ phases, the maximum degree drops below $\mathrm{poly}(\alpha, \log n)$, and we switch to the algorithm by Ghaffari and Uitto [22] to find a maximal matching (or MIS) of the remaining low-degree graph. Combining with the partial solution obtained in previous phases, we then obtain a maximal matching (or MIS) of the input graph.

To implement the algorithm efficiently in the MPC model, Behnezhad et al. use the, by-now standard, graph exponentiation technique [28, 19]: For each vertex v, we store its d-hop neighborhood in one machine, with d initially being 1. In $O(1)$ MPC rounds we can double the radius to $2d$, by requesting and collecting the d-hop neighborhoods of all vertices u in the d-hop neighborhood of v. Hence in $O(\log d)$ MPC rounds we can collect the d-hop neighborhood of each vertex, which has at most Δ^d edges and can fit into the n^δ local memory when $d = \delta \log_\Delta n$. Then we can simulate $O(\log_\Delta n)$ LOCAL rounds in this phase using only $O(1)$ MPC rounds. As each phase requires $O(\log d) \le O(\log \log n)$ MPC rounds for graph exponentiation, the total round-complexity of this MPC algorithm is $O(\log \log n \log \log \Delta) \le O(\log^2 \log n)$.

The algorithm described above would need $\widetilde{O}(n^{1+\delta} + m)$ global memory. In order to reduce it to $\widetilde{O}(n + m)$, Behnezhad et al. used an additional idea: As our objective in a phase is to make high-degree vertices disappear, we can without loss of generality assume that

the LOCAL algorithm only removes vertices that are in the $O(1)$-hop neighborhood of the high-degree vertices, and other vertices are considered as irrelevant and do not participate in the LOCAL algorithm. As the number of high-degree vertices decreases after every execution of the LOCAL algorithm, the number of relevant vertices also decreases quickly. Then, we have enough space for expanding the stored neighborhood of each relevant vertex.

New techniques. Our new idea is a *pipelining* technique: as more vertices become irrelevant in the current phase, we can start running the next phase on these vertices, *concurrently* with the current phase which has not necessarily finished completely. In our algorithm, after starting phase $(\ell - 1)$, we wait for $O(1)$ MPC rounds and then start phase ℓ. There are $O(\log \log \Delta)$ phases in total, each taking $O(\log \log n)$ MPC rounds, so the round complexity is $O(\log \log \Delta) \cdot O(1) + O(\log \log n) = O(\log \log n)$ in total. Our pipelined algorithm produces exactly the same result as the unpipelined version does (using the same random bits).

When running phase ℓ, we need to deal with *pending* vertices that have not finished their computation in previous phases $1, 2, \ldots, \ell - 1$. As it is still unknown whether they will be removed by the start of phase ℓ, the messages sent from these vertices during phase ℓ are temporarily marked as *pending*. If a vertex receives a pending message, then its state also becomes pending, and so on. When simulating the LOCAL computation in phase ℓ, we are only able to simulate the behaviour of non-pending vertices; only upon finishing the simulation of previous phases $1, 2, \ldots, \ell - 1$ for some vertex, the initial state of this vertex in phase ℓ becomes known, which allows us to resume the phase ℓ computation around its neighborhood and remove the pending marks.

Now we take a closer look at what happens when we perform graph exponentiation. In the ideal scenario, after phase $\ell - 1$ completely finishes, the maximum degree drops below $\Delta_{\ell-1} := \Delta^{1/2^{\ell-1}}$, allowing us to store the $(\delta \log_{\Delta_{\ell-1}} n)$-hop neighborhood of a vertex into the local memory. However, in the actual situation, when expanding the neighborhood we may encounter pending vertices, which may not have degree upper bounded by $\Delta_{\ell-1}$. It would be problematic if we simply excluded these vertices when we expand the neighborhood, as they may later have degree reduced to the interval $(\Delta_\ell, \Delta_{\ell-1}]$ and become relevant in phase ℓ. So we have to store them as well, and this poses challenges to bounding the local memory and the global memory of our algorithm – this is the most technical part of our analysis.

In this overview, we provide some intuition on how we analyze the global memory used for storing the neighborhoods. We will analyze the neighborhoods of non-pending vertices and of pending vertices separately. In phase ℓ, after repeating the LOCAL algorithm k times, the number of non-pending vertices v with degrees in $(\Delta_\ell, \Delta_{\ell-1}]$ can be bounded by $n/\Delta_{\ell-1}^k$. The d-hop neighborhood (we will maintain $d = ck$ for some constant $c \in (0, 1)$) of such a non-pending vertex v only contains vertices u with degree at most $\Delta_{\ell-1}$, since otherwise u would still be waiting for phase $\ell - 1$ to finish, causing v to become a pending vertex. Hence the total size of d-hop neighborhoods of these non-pending vertices v is roughly at most $(n/\Delta_{\ell-1}^k) \cdot \Delta_{\ell-1}^d < n$. Now we consider the pending vertices. For any pending vertex v, it must be within k distance from a vertex u which is currently stuck at phase $\ell - q$, for some $q \geq 1$. We pick such u with the biggest q, so v can be reached from u by a k-step path on which all vertices have degree $\leq \Delta_{\ell-q}$ (since if any of them had degree greater than $\Delta_{\ell-q}$, it would be stuck at a phase earlier than $\ell - q$, contradicting the maximality of q). Hence we can bound the number of pending vertices v by $\sum_{q \geq 1} H_{\ell-q} \cdot \Delta_{\ell-q}^k$, where $H_{\ell-q}$ denotes the number of vertices currently stuck at phase $\ell - q$, and the total size of neighborhoods of these v can be bounded by a similar summation. In order to show a near-linear upper bound, we want to make this summation dominated by a geometric series. To do this, we use a similar argument to show an upper bound on $H_{\ell-q}$, and hence after setting appropriate parameters, we can use *induction on ℓ* to establish the desired memory bound.

2.2 Definitions and algorithm structure

Suppose each machine in the MPC model has n^δ local memory, for some constant $0 < \delta < 1$. We will use the following LOCAL algorithm [5] as a black box.

▶ **Lemma 3** ([5, Theorem 7.2], restated[2]; see also [10, Section 5.2]). *Let α and Δ be parameters satisfying $\Delta \geq \mathrm{poly}(\alpha, \log n)$. Suppose the input graph G has arboricity at most α and maximum degree at most Δ. Let $H_G := \{v : \deg_G(v) > \sqrt{\Delta}\}$ denote the set of high-degree vertices in graph G.*

There is a LOCAL algorithm of round complexity $r = O(1)$ that computes a matching M (or an independent set I) of G, such that $|H_{G'}| \leq |H_G|/\Delta$ w.h.p., where G' is the induced subgraph of G where matched vertices in M (or vertices in I and their neighbors) are removed. Moreover, the algorithm has the following properties.

(1) *The state description and random bits for each vertex, as well as communication along every edge in each round, have length at most $\mathrm{polylog}(n)$ bits. The space usage for computing the new state of vertex v is $\deg_G(v) \cdot \mathrm{polylog}(n)$, and the space usage for computing the message sent along each edge is $\mathrm{polylog}(n)$.*

(2) *Let H^+ denote the r-hop neighborhood of H_G in G. Then vertices outside H^+ are not affected by the algorithm (i.e., they will stay in graph G', and their degrees in G' are unchanged).*

Let $\Delta_\ell := \Delta^{1/2^\ell}$, where Δ is the maximum degree of the input graph. The unpipelined algorithm sequentially runs $L = O(\log \log \Delta)$ *phases*, where phase ℓ ($\ell = 1, 2, \ldots, L$) repeats the LOCAL algorithm $O(\log_{\Delta_\ell} n)$ times with degree parameter $\Delta_{\ell-1}$, and reduces the maximum degree from $\Delta_{\ell-1}$ to Δ_ℓ.

Our pipelined MPC algorithm runs in $O(\log \log n)$ *iterations* (starting from iteration 1), each taking $O(1)$ MPC rounds. In each iteration, there are multiple *phases* concurrently being simulated by our MPC algorithm. There is a small *lag* $t = O(1)$ between the start of phase ℓ and $\ell+1$, i.e., the simulation of phase ℓ starts at iteration $j = (\ell-1)t+1$, by running INITIALIZE(ℓ). In each iteration j, we perform SIMULATE(ℓ, j), UPDATE, and EXPAND(ℓ, j), concurrently for all active phases ℓ (i.e, phases that have already started). See Algorithm 1 for the algorithm structure. The value of parameter t will be determined in Section 2.5.

■ **Algorithm 1** Structure of our pipelined MPC algorithm.

Number of phases $L = O(\log \log \Delta)$.
Lag parameter $t = O(1)$.
for iteration $j \leftarrow 1, 2, \ldots, O(\log \log n)$ **do**
    $L_{\text{active}} \leftarrow \min\{L, \lceil (j-1)/t \rceil\}$ *(Number of phases that have already started)*
    if $j = (\ell-1)t + 1$ for some $\ell \in [L]$ **then**
        INITIALIZE(ℓ) *(Phase ℓ starts in this iteration)*
    SIMULATE(ℓ, j) for all $\ell \in [L_{\text{active}}]$ concurrently
    UPDATE
    EXPAND(ℓ, j) for all $\ell \in [L_{\text{active}}]$ concurrently
Switch to Ghaffari and Uitto's algorithm [22]

Our MPC algorithm maintains the subgraph induced by the remaining vertices. After iteration j finishes, the remaining induced subgraph is denoted by $G(j)$. We have $G = G(0) \supseteq G(1) \supseteq \cdots$. In iteration j, concurrently for each active phase ℓ, SIMULATE(ℓ, j) simulates

---

[2] The proof of [5, Theorem 7.2] presented an $O(1)$-round algorithm that reduces the number of vertices with degree greater than $t\alpha$ by a $t^{1/7}$ factor, for any parameter $t \geq \mathrm{poly}(\alpha, \log n)$. By choosing $t = \sqrt{\Delta}/\alpha$ and repeating the algorithm $O(1)$ times we obtain the claimed statement.

(part of) the LOCAL computation in phase ℓ, and removes the matched vertices (or vertices in the independent set and their neighbors) from the graph. After removing the vertices, in UPDATE we compute the degrees of vertices in the remaining graph $G(j)$, and update some other information. Then, in EXPAND(ℓ, j), we perform one graph exponentiation step and double the radius of the neighborhoods stored in memory.

2.3 LOCAL simulation with incomplete information

As discussed in Section 2.1, our pipelining algorithm is complicated by the presence of pending vertices. In this section we explain how to deal with them properly when simulating LOCAL algorithms.

When we simulate a LOCAL algorithm A on an input graph G, we may not have the complete information of the initial states of vertices. We say vertex v is *initially pending* if v's initial state is unknown.[3] We have the following simple observation:

▶ **Observation 4.** *Define a LOCAL algorithm A', which is the same as A except for the following additional rule: the messages sent from a pending vertex are marked as pending, and a vertex receiving a pending message becomes pending.*

If v's R-hop neighborhood in the input graph G does not contain an initially pending vertex, then v's state after running algorithm A' for R rounds is non-pending, and is the same as its state in algorithm A after R rounds.

Now we introduce another simple lemma which will be helpful in reducing the space usage of our MPC simulation. Consider a LOCAL algorithm A in which vertices may get eliminated during execution: after a vertex is eliminated, it no longer participates in later rounds of the algorithm. It is well-known that we can simulate the behaviour of a vertex v up to R LOCAL rounds provided that we know v's R-hop neighborhood in the input graph $G = (V, E)$. However, if we somehow know a subset U of vertices that will be eliminated by A, then we only need to collect v's R-hop neighborhood in the induced subgraph $G[V \setminus U]$ (plus the messages sent from U when they are alive), which could be much smaller than its R-hop neighborhood in G.

▶ **Lemma 5.** *Suppose a LOCAL algorithm A runs on an input graph $G = (V, E)$, and $U_0 \subseteq V$ is the set of vertices that are eliminated during the execution of A.*

Let $U \subseteq U_0$, and $v \in V \setminus U$. Then, we can simulate the behaviour of v in the first R rounds of the algorithm A, provided that we have the information of:
(a) *The R-hop neighborhood of v in the induced subgraph $G[V \setminus U]$.*
(b) *The complete communication history along every edge $(u, w) \in E$ where $u \in U$ and w belongs to the neighborhood defined in (a), until u gets eliminated by A.*
The proof of this lemma is trivial.

2.4 Algorithm in detail

Recall that each machine in the MPC model has n^δ memory. Define $s := \lceil 10/\delta \rceil$. Recall that $r = O(1)$ is the round complexity of the LOCAL algorithm from Lemma 3, and $G(j)$ is the graph induced by the remaining vertices at the end of iteration j. Let $H_\ell(j)$ denote the set

---

[3] Suppose LOCAL algorithm A refers to the computation in phase ℓ. Here, the *state* of a vertex v includes its degree, whether it has been removed, and some other information required by Lemma 3 Property (1). If we have not finished the phase $(\ell - 1)$ simulation for vertex v, then we do not know v's state at the beginning of phase ℓ, and we will consider v to be in a special *pending state*.

of vertices v with $\deg_{G(j)}(v) > \Delta_\ell$, and let $H_\ell^+(j)$ be the r-hop neighborhood of $H_\ell(j)$ in graph $G(j)$. From Lemma 3 (Property (2)), we observe that vertices outside $H_\ell^+(j)$ are not affected by phase $1, 2, \ldots, \ell$ in the following iterations $j + 1, j + 2, \ldots$.

At the end of iteration j, we maintain the following invariants for all active phases ℓ.

▶ **Property 6.** *Let $j = (\ell - 1)t + i$ ($i \geq 1$), i.e., j is the i-th iteration since the start of phase ℓ. Let $d_i := 2^i r$, and recall that $s := \lceil 10/\delta \rceil$. At the end of iteration j, the following hold:*

1. **(Number of finished LOCAL executions)** *For every $v \in H_\ell^+(j - 1)$, if there is no $u \in H_{\ell-1}^+(j - 1)$ satisfying $\operatorname{dis}_{H_\ell^+(j-1)}(u, v) \leq sd_i \cdot r$, then we have computed v's state after sd_i executions of the LOCAL algorithm (from Lemma 3) in phase ℓ.*
2. **(Radius of collected neighborhoods)** *For every $v \in H_\ell^+(j)$, we have collected the following information into one machine:*
   (a) *The d_i-hop neighborhood of v in graph $H_\ell^+(j)$.*
   (b) *The communication history during phase ℓ along every edge (u, w), where vertex $u \notin H_\ell^+(j)$ was eliminated during phase ℓ, and w belongs to the neighborhood defined in (a).*

Informally, Item 1 says that we have simulated sd_i executions of the LOCAL algorithm (each using r rounds) in phases ℓ, as long as the $sd_i \cdot r$-hop neighborhood contained no initially pending vertices when iteration j began.

Now we describe how to implement INITIALIZE(ℓ), SIMULATE(ℓ, j), UPDATE, EXPAND(ℓ, j) in iteration $j = (\ell - 1)t + i$ using $O(1)$ MPC rounds and maintain Property 6 (the purpose of these procedures has been informally described at the end of Section 2.2).

▶ **Lemma 7.** *Assume Property 6 holds at the end of iteration $j - 1$. We can implement SIMULATE(ℓ, j), UPDATE, EXPAND(ℓ, j) (concurrently for all phases $\ell \in [L_{\mathsf{active}}]$, see 4th line in Algorithm 1) using $O(1)$ MPC rounds so that at the end of iteration j, Property 6 holds for all phases $\ell \in [L_{\mathsf{active}}]$.*

Proof. Let $j = (\ell - 1)t + 1$. Recall that $d_i = 2^i r$. In the following algorithmic description we omit the low-level implementation details of sublinear-memory MPC model. We will briefly address them in Remark 9.

- SIMULATE(ℓ, j). We apply Observation 4, where A is the phase ℓ algorithm, and vertices in $H_{\ell-1}^+(j - 1)$ are initially pending. To maintain Property 6 Item 1, it suffices to simulate algorithm A' (defined in Observation 4) up to $sd_i \cdot r$ LOCAL rounds.

  Apply Lemma 5, with U being the set of vertices $u \notin H_\ell^+(j - 1)$ which were eliminated during phase ℓ. By Property 6 Item 2, we already have the d_{i-1}-hop neighborhood of every vertex in $H_\ell^+(j - 1)$ (as well as the communication history required by Lemma 5) stored into one machine. So we can simulate d_{i-1} LOCAL rounds for all $v \in H_\ell^+(j - 1)$ in one MPC round.

  After simulating d_{i-1} LOCAL rounds, we update the current states of the vertices in all stored d_{i-1}-hop neighborhoods, using $O(1)$ MPC rounds. Then we again simulate d_{i-1} LOCAL rounds for all vertices in one MPC round, so that the total number of simulated LOCAL rounds becomes $2d_{i-1}$. And we update the current states of vertices, and so on. In this way, we can simulate the first $sd_i \cdot r$ LOCAL rounds in phase ℓ using $O(sd_i \cdot r/d_{i-1}) = O(1)$ MPC rounds. (This is the "blind coordination" technique used by Behnezhad et al. [9, Section 5.3]) When we simulate the behaviour of v in phase ℓ, we also record the communication history of v. Since the degree of v in phase ℓ is at most $\Delta_{\ell-1}$, the size of the messages is $O(\Delta_{\ell-1} \cdot \operatorname{polylog} n)$.

- UPDATE. After vertices are removed from $G(j - 1)$ by SIMULATE(ℓ, j), let $G(j)$ be the induced subgraph of the remaining vertices, and compute the vertex degrees in graph $G(j)$. Then, for every ℓ, delete from $H_\ell(j - 1)$ the vertices whose degree dropped to $\leq \Delta_\ell$

and the ones that got removed, and obtain $H_\ell(j)$. Then similarly obtain $H_\ell^+(j)$. We also delete these vertices from the stored neighborhoods.

■ EXPAND(ℓ, j). To maintain Property 6 Item 2, we perform one graph exponentiation step in $O(1)$ MPC rounds. Every $u \in H_\ell^+(j)$ requests the neighborhood of every other v in the stored d_{i-1}-hop neighborhood of u. The new radius becomes $2d_{i-1} = d_i$. Then it is easy to collect the required communication history of vertices in this neighborhood.

Note that SIMULATE(ℓ, j) and EXPAND(ℓ, j) are independent across all phases ℓ and can be executed concurrently. ◀

▶ **Lemma 8.** *At iteration $j = (\ell - 1)t + 1$, we can implement* INITIALIZE(ℓ) *using $O(1)$ MPC rounds so that at the end of this iteration Property 6 holds for phase ℓ.*

Proof. We run the LOCAL algorithm in phase ℓ for sd_1 times (in the sense of Observation 4). This can be done in $O(sd_1 \cdot r) = O(1)$ MPC rounds with $\widetilde{O}(n + m)$ global memory. Then, we collect the $2r$-hop neighborhood of every vertex in $H_\ell^+(j)$ (and the required communication history) in $O(1)$ MPC rounds. ◀

▶ **Remark 9.** We did not spell out the low-level details, but it is not hard to verify that our algorithm only involves basic operations and can be easily implemented using the standard MPC primitives developed in previous works, e.g. [2, Section E] and the references therein. The only concern is that, when $\Delta \geq n^\delta$, we cannot gather the neighborhood of a vertex into one machine. This issue was already addressed in Behnezhad et al.'s unpipelined algorithm [10, Section 6], using a load-balancing technique. Hence we can first sequentially run $O(1)$ phases of their algorithm (in $O(\log \log n)$ MPC rounds) at the very beginning, which reduce the maximum degree of the remaining graph to below n^α for any desired constant $\alpha > 0$.

Now we use Property 6 Item 1 to show the round complexity of our algorithm.

▶ **Theorem 10.** *Our pipelined MPC algorithm runs in $O(\sqrt{\log \alpha} \log \log \alpha + \log \log n)$ rounds.*

Proof. First consider the unpipelined scenario: in phase ℓ, by Lemma 3, after $\log n$ executions of the LOCAL algorithm, w.h.p. there are no remaining vertices with degree higher than Δ_ℓ.

Now we look at our pipelined MPC simulation. Note that each iteration takes $O(1)$ MPC rounds. After $j_1 = O(t \cdot L) = O(\log \log \Delta)$ iterations, all phases have started their simulation. Then, after another $j_2 = O(\log \log n)$ iterations, every phase ℓ has at least been running for $i > \log \log n$ iterations. Let $j = j_1 + j_2$ denote the number of iterations so far. By Property 6 Item 1, if $H_{\ell-1}^+(j-1)$ is empty, then at the end of iteration j we will have finished $sd_i > \log n$ many LOCAL executions of phase ℓ, which (together with the previous paragraph) implies $H_\ell(j)$ is empty. Hence, at this point $H_1(j)$ is already empty (since $H_0^+(j-1)$ is always empty). This in turn implies that in the next iteration $j+1$, $H_2(j+1)$ will be empty. By a simple induction, after $L-1$ iterations, $H_L(j+L-1)$ becomes empty, which means that the maximum degree of the remaining graph $G(j+L-1)$ is at most $\Delta_L = \text{poly}(\alpha, \log n)$. Finally, we use Ghaffari and Uitto's algorithm [22], which takes $O(\sqrt{\log \Delta_L} \log \log \Delta_L + \sqrt{\log \log n}) \leq O(\sqrt{\log \alpha} \log \log \alpha) + \widetilde{O}(\sqrt{\log \log n})$ MPC rounds. Hence, the total round complexity is $O(\sqrt{\log \alpha} \log \log \alpha + \log \log n)$. ◀

2.5 Memory requirement

In this section we will show that our algorithm satisfies the local memory and global memory constraints. In the following, we will always assume $j = (\ell - 1)t + i$, that is, j is the i-th iteration since phase $\ell \in [L]$ starts. Recall that $\Delta_\ell := \Delta^{1/2^\ell}$. For a subgraph H, we use $|H|$ to denote the number of vertices in H. We need the following lemma.

▶ **Lemma 11.** *With high probability, $|H_\ell^+(j)| \leq 2n/\Delta_{\ell-1}^{(s-1)d_i}$.*

In order to prove Lemma 11, we first define the following quantity.

▶ **Definition 12.** *For every $v \in H_\ell^+(j)$, let $D_{\ell,j}(v) := \max\limits_{\substack{u \in G(j-1): \\ \mathrm{dis}_{G(j-1)}(u,v) \leq (sd_i+2)r}} \big\{ \deg_{G(j-1)}(u) \big\}.$*

Note that $D_{\ell,j}(v) \geq \deg_{G(j-1)}(v^-) > \Delta_\ell$, where $v^- \in H_\ell(j)$ and $\mathrm{dis}_{G(j-1)}(v^-, v) \leq r$. Then, we classify the vertices in $H_\ell^+(j)$ by the values of $D_{\ell,j}(v)$.

▶ **Definition 13.** *The vertices in $H_\ell^+(j)$ are partitioned as $\dot\bigcup_{0 \leq q \leq \ell-1} P_{\ell,j}^{(q)}$, where*

$$P_{\ell,j}^{(q)} := \big\{ v \in H_\ell^+(j) : D_{\ell,j}(v) \in (\Delta_{\ell-q}, \Delta_{\ell-q-1}] \big\}.$$

Informally, for $v \in P_{\ell,j}^{(q)}$, there exists a nearby vertex u which is currently stuck in phase $\ell - q$. In order to bound $|H_\ell^+(j)|$, we will bound the size of $P_{\ell,j}^{(q)}$ separately as follows.

▶ **Lemma 14.** *With high probability, the following upper bounds hold.*
1. $|P_{\ell,j}^{(0)}| \leq n/\Delta_{\ell-1}^{(s-1)d_i}$.
2. *For $1 \leq q \leq \ell - 1$, $|P_{\ell,j}^{(q)}| \leq |H_{\ell-q}(j-1)| \cdot \Delta_{\ell-q-1}^{(sd_i+3)r}$.*

Proof of Item 1. For $v^+ \in P_{\ell,j}^{(0)} \subseteq H_\ell^+(j)$, there exists $v \in H_\ell(j)$ such that $\mathrm{dis}_{G(j)}(v, v^+) \leq r$. Suppose for contradiction that there exists $u^+ \in H_{\ell-1}^+(j-1)$ satisfying $\mathrm{dis}_{G(j-1)}(u^+, v) \leq sd_i r$. Then there exists $u \in H_{\ell-1}(j-1)$ with $\mathrm{dis}_{G(j-1)}(u, u^+) \leq r$, implying that $\mathrm{dis}_{G(j-1)}(u, v^+) \leq \mathrm{dis}_{G(j-1)}(u, u^+) + \mathrm{dis}_{G(j-1)}(u^+, v) + \mathrm{dis}_{G(j-1)}(v, v^+) \leq r + sd_i r + r = (sd_i + 2)r$. However, since $\deg_{G(j-1)}(u) > \Delta_{\ell-1}$, this contradicts $D(v^+) \leq \Delta_{\ell-1}$. Hence such u^+ does not exist.

Then, by Property 6 Item 1, this implies that the state of v after sd_i executions of the LOCAL algorithm in phase ℓ is already determined. By the property of the LOCAL algorithm, the number of such vertices $v \in H_\ell(j)$ is at most $n/\Delta_{\ell-1}^{sd_i}$.

Note that v^+ is connected to v in graph $G(j)$ by a path of length at most r, and previous discussion implies that every vertex w on this path satisfies $w \notin H_{\ell-1}^+(j-1)$ and thus $\deg_{G(j)}(w) \leq \Delta_{\ell-1}$. So the number of such v^+ is at most $(n/\Delta_{\ell-1}^{sd_i}) \cdot \Delta_{\ell-1}^{r+1} \leq n/\Delta_{\ell-1}^{(s-1)d_i}$. ◀

Proof of Item 2. For $v \in P_{\ell,j}^{(q)}$, let $u \in G(j-1)$ be the maximizer in the definition of $D_{\ell,j}(v)$. Then v is connected to u in graph $G(j-1)$ by a path of length at most $(sd_i+2)r$, on which every vertex w (including u,v) has degree $\deg_{G(j-1)}(w) \leq \deg_{G(j-1)}(u) \leq \Delta_{\ell-m-1}$. Fixing a vertex u, the number of vertices v that can be reached from u in this way is at most $\Delta_{\ell-q-1}^{(sd_i+2)r+1} \leq \Delta_{\ell-q-1}^{(sd_i+3)r}$. The number of vertices $u \in G(j-1)$ with $\deg_{G(j-1)}(u) \in (\Delta_{\ell-q}, \Delta_{\ell-q-1}]$ is at most $|H_{\ell-q}(j-1)|$. Hence, we conclude that $|P_{\ell,j}^{(q)}| \leq |H_{\ell-q}(j-1)| \cdot \Delta_{\ell-q-1}^{(sd_i+3)r}$. ◀

Proof of Lemma 11. To show that $|H_\ell^+(j)| \leq 2n/\Delta_{\ell-1}^{(s-1)d_i}$, we prove by induction on ℓ.

For $\ell = 1$, by Lemma 14, we have $|H_1^+(j)| = |P_{1,j}^{(0)}| \leq n/\Delta_0^{(s-1)d_i} \leq 2n/\Delta_{\ell-1}^{(s-1)d_i}$.

For $\ell \geq 2$, by Lemma 14 and induction hypothesis,

$$
\begin{aligned}
|H_\ell^+(j)| = |P_{\ell,j}^{(0)}| + \sum_{1 \leq q < \ell} |P_{\ell,j}^{(q)}| &\leq \frac{n}{\Delta_{\ell-1}^{(s-1)d_i}} + \sum_{1 \leq q < \ell} |H_{\ell-q}^+(j-1)| \cdot \Delta_{\ell-q-1}^{(sd_i+3)r} \\
&\leq \frac{n}{\Delta_{\ell-1}^{(s-1)d_i}} + \sum_{1 \leq q < \ell} \frac{2n}{\Delta_{\ell-q-1}^{(s-1)d_{i+qt-1}}} \cdot \Delta_{\ell-q-1}^{(sd_i+3)r} \\
&\leq \frac{n}{\Delta_{\ell-1}^{(s-1)d_i}} + \sum_{1 \leq q < \ell} \frac{2n}{\Delta_{\ell-1}^{d_i((s-1)2^{qt-1}-(s+2)r)/2^q}} \\
&\leq \frac{n}{\Delta_{\ell-1}^{(s-1)d_i}} + \sum_{1 \leq q < \ell} \frac{2n}{\Delta_{\ell-1}^{(s-1)d_i q+1}},
\end{aligned}
$$

where in the second inequality we used the induction hypothesis and $j - 1 = (\ell - q - 1)t + i + qt - 1$, and in the last inequality we chose t to be a big enough constant. Then, assuming $\Delta_{\ell-1} \geq 4$, we conclude $|H_\ell^+(j)| \leq 2n/\Delta_{\ell-1}^{(s-1)d_i}$. ◄

Now we turn to analyzing the memory used for storing the d_i-hop neighborhood of $v \in H_\ell^+(j)$ (together with their communication history required by Property 6 Item 2).

▶ **Lemma 15.** *For $v \in P_{\ell,j}^{(q)}$ ($0 \leq q < \ell$), the memory for storing the d_i-hop neighborhood of v in graph $H_\ell^+(j)$ is at most $\Delta_{\ell-q-1}^{d_i+2} \cdot \text{polylog } n$.*

Proof. Since $v \in P_{\ell,j}^{(q)}$, we have $D_{\ell,j}(v) \leq \Delta_{\ell-q-1}$. By the definition of $D_{\ell,j}(v)$, every vertex u in the d_i-hop neighborhood of v in $H_\ell^+(j)$ has $\deg_{H_\ell^+(j)}(u) \leq \deg_{G(j-1)}(u) \leq D_{\ell,j}(v) \leq \Delta_{\ell-q-1}$, and thus the number of vertices u in the d_i-hop neighborhood of v is at most $\Delta_{\ell-q-1}^{d_i+1}$. Hence the total memory for storing this neighborhood (together with their communication history in phase ℓ) is at most $\Delta_{\ell-q-1}^{d_i+1} \cdot (\Delta_{\ell-1} \text{ polylog } n) \leq \Delta_{\ell-q-1}^{d_i+2} \cdot \text{polylog } n$. ◄

We prove that this memory does not violate the memory constraints.

▶ **Theorem 16** (Local memory constraint). *With high probability, for any $v \in H_\ell^+(j)$, the memory for storing the d_i-hop neighborhood of v in graph $H_\ell^+(j)$ is at most n^δ.*

Proof. Let q such that $v \in P_{\ell,j}^{(q)}$. Such v exists only if $|P_{\ell,j}^{(q)}| \geq 1$. We consider two cases.
1. If $q = 0$, by Lemma 14 and $|P_{\ell,j}^{(0)}| \geq 1$ we have $n/\Delta_{\ell-1}^{(s-1)d_i} \geq 1$. Then by Lemma 15, the memory is $\Delta_{\ell-1}^{d_i+2} \cdot \text{polylog } n \leq n^{(d_i+2)/((s-1)d_i)} \cdot \text{polylog } n \ll n^\delta$, where the last inequality follows from $s \geq 10/\delta$ and $d_i \geq 2$.
2. If $1 \leq q < \ell$, by Lemma 14 and $|P_{\ell,j}^{(q)}| \geq 1$ we have $|H_{\ell-q}(j-1)| \geq 1$, which then implies $2n/\Delta_{\ell-q-1}^{(s-1)d_{i+qt-1}} \geq 1$ by Lemma 11. Then similarly by Lemma 15, the memory requirement is $\Delta_{\ell-q-1}^{d_i+2} \cdot \text{polylog } n \leq (2n)^{(d_i+2)/((s-1)d_{i+qt-1})} \cdot \text{polylog } n \ll n^\delta$. ◄

▶ **Theorem 17** (Global memory constraint). *With high probability, the global memory for storing the neighborhoods is $\widetilde{O}(n)$.*

Proof. Since there are only $L = O(\log \log \Delta)$ phases running concurrently, it suffices to show that, for every $\ell \in [L]$, the total memory for storing the d_i-hop neighborhoods of all vertices v in graph $H_\ell^+(j)$ is at most $\widetilde{O}(n)$, where $j = (\ell - 1)t + i$.

By Lemma 15, the total memory can be bounded by

$$
|P_{\ell,j}^{(0)}| \cdot \Delta_{\ell-1}^{d_i+2} \cdot \text{polylog } n + \sum_{1 \leq q < \ell} |P_{\ell,j}^{(q)}| \cdot \Delta_{\ell-q-1}^{d_i+2} \cdot \text{polylog } n
$$

$$\leq \frac{n}{\Delta_{\ell-1}^{(s-1)d_i}} \cdot \Delta_{\ell-1}^{d_i+2} \cdot \text{polylog } n + \sum_{1 \leq q < \ell} |H_{\ell-q}^+(j-1)| \cdot \Delta_{\ell-q-1}^{(sd_i+3)r} \cdot \Delta_{\ell-q-1}^{d_i+2} \cdot \text{polylog } n$$

$$\leq \widetilde{O}(n),$$

where the last inequality follows from a similar calculation as in the proof of Lemma 11. ◄

3 4-coloring of trees in $O(\log \log n)$ rounds

In this section, we present an algorithm that colors a tree with 4 colors in $O(\log \log n)$ MPC rounds and $O(m)$ global memory, thus providing a proof for Theorem 2. We start with a high-level overview, and then fill in the details of the algorithm and its analysis.

High-Level Overview. Our 4-coloring algorithm starts by randomly partitioning the vertices of the input tree into two sets. Each set induces a forest such that each connected component has a diameter of $O(\log n)$, with high probability. Each connected component corresponds to a tree and will be rooted (i.e., orienting each edge towards the root) by using $\Theta(\log^2 n)$ parallel black-box invocations to the connected components algorithm of Behnezhad et al. [11] (which improved on the earlier work by Andoni et al. [2]). We note that this connected components algorithm runs $O(\log D + \log \log n)$ rounds, where D denotes the maximum component diameter, using strongly sublinear local memory and $O(m)$ global memory. Once the tree is rooted, computing a 2-coloring in $O(\log \log n)$ rounds is easy: we can learn the distance from the root, by pointer jumping along the outgoing edges and leveraging that the diameter is bounded by $O(\log n)$, and then 2-color nodes based on odd or even distances. Thus, each of the two forests can be colored with 2 colors. This results in a 4-coloring of the complete tree. The presented algorithm would need a slightly superlinear global memory of $\Omega(n \log^2 n)$. To alleviate this issue and work with only $O(n)$ global memory, our actual algorithm first reduces the number of vertices to $O(n/\text{poly}(\log n))$ by using the well-known peeling algorithm [4], which in $O(\log \log n)$ iterations repeatedly removes vertices of degree at most 2. These removed vertices are then colored at the very end of the algorithm, after coloring those $O(n/\text{poly}(\log n))$ remaining vertices, in $\Theta(\log \log n)$ additional MPC rounds.

3.1 Details + Analysis

In this section, we explain and analyze the algorithm in detail. Let T denote the input tree.

3.1.1 Reducing the number of vertices

First, we explain and analyze the peeling algorithm which reduces the number of vertices to $O(n/\log^2 n)$. The peeling algorithm consists of $N = \Theta(\log \log n)$ iterations. In each iteration, we remove all vertices that have at most 2 neighbors in the current graph. In each iteration of the peeling algorithm, a constant fraction of the remaining vertices are removed.

▶ **Lemma 18.** *Let $G = (V, E)$ be a forest. Then, $|\{v \in V \colon \deg(v) \geq 3)\}| \leq \frac{2}{3}|V|$.*

Proof. We have $3 \cdot |\{v \in V \colon \deg(v) \geq 3)\}| \leq \sum_{v \in V} \deg(v) \leq 2|E| \leq 2|V|$, where the last inequality follows, as each forest has at most $|V| - 1$ edges and the second-last inequality is the well-known handshaking lemma. ◄

Thus, after iteration i of the peeling algorithm, the number of remaining vertices is at most $n \cdot (2/3)^i$. Hence, after running $N = \lceil 2 \log_{3/2} \log n \rceil$ iterations of the peeling algorithm, the number of remaining vertices is at most $O(n/\log^2(n))$. Note that each iteration of the peeling algorithm can easily be implemented in $O(1)$ MPC rounds and linear global memory.

Next, we explain how to color the vertices removed by the peeling algorithm, once all the remaining vertices are colored, using no additional colors. The idea is well-known in the context of LOCAL algorithms. For $i \in [N]$, let W_i denote the set of vertices removed in the i-th iteration. The basic idea is to first color all vertices in W_N, then the ones in W_{N-1} and so on. By coloring the vertices in that order, the number of neighbors that previously got assigned a color is upper bounded by 2. Hence, we can assign each node one of the first 3 colors without creating any conflict. In order for this procedure to be efficient, we need to color multiple nodes simultaneously. However, coloring all nodes in W_i in parallel is problematic, as W_i may contain neighboring nodes. To circumvent this problem, we temporarily color $T[W_i]$ with a constant number of colors – these are not a part of the output coloring but merely used as a schedule in computing the output coloring. In fact, we can color all of $T[W_1], T[W_2], \ldots, T[W_N]$ simultaneously, each separately using constant many colors, in $O(\log^* n)$ MPC rounds. This is done by simulating the LOCAL algorithm of Cole and Vishkin [15]. Then, to compute the output coloring, for each $T[W_i]$, we iterate through the constant many schedule colors in $T[W_i]$, and we compute the output color of all the nodes that got assigned the same schedule color in parallel. Hence, one never assigns two neighboring nodes an output color simultaneously. Coloring the nodes of one schedule color class in W_i can be simulated in $O(1)$ MPC rounds. Thus, we can color all the deleted vertices in $O(1) \cdot O(1) \cdot O(\log \log n)$ MPC rounds and using linear global memory. Hence, what remains is to show how to color a forest consisting of $O(n/\log^2(n))$ nodes in $O(\log \log n)$ MPC rounds, using $O(n)$ global memory.

3.1.2   Random Partitioning

In order for our procedure to be efficient, we need each connected component under consideration to have a diameter of $O(\log n)$. We achieve this by randomly partitioning the remaining vertices into two sets V_1 and V_2, with each remaining vertex being in either one of them with a probability of $1/2$, independent of the other vertices. The partitioning leads to each connected component of $T[V_1]$ and $T[V_2]$ having a diameter of $O(\log n)$, with high probability. The argument is simple: each path of length $\omega(\log n)$ is present in $T[V_1]$ or $T[V_2]$ with a probability less than $1/\operatorname{poly}(n)$. Then, we can union bound over the at most $O(n^2)$ distinct paths in the tree to conclude that no path of length $\omega(\log n)$ will be present in $T[V_1]$ or $T[V_2]$, with probability $1 - 1/\operatorname{poly}(n)$.

3.1.3   Handling connected components in parallel

In the next paragraph, we discuss an algorithm that computes a 2-coloring of a given connected component in $O(\log \log n)$ rounds. The algorithm works under the assumption that each node of the component has a unique identifier in $[\eta]$, with η denoting the size of the connected component. Furthermore, it assumes that the edges of the connected component are the only input given to the algorithm. It requires each machine to have a local memory of $O(\min(\eta, n^\delta))$ and uses $O(\eta \log^2 n)$ global memory. Before going into the details of how the algorithm works, we first argue why the existence of such an algorithm readily implies that we can 2-color each connected component in $T[V_1]$ and $T[V_2]$ with 2 colors in $O(\log \log n)$ MPC rounds and $O(m)$ global memory.

Before 2-coloring each connected component, we start by computing the connected components of both $T[V_1]$ and $T[V_2]$ by using the connected component algorithm of [11]. The algorithm runs in $O(\log \log n + \log D)$ rounds, where D denotes the maximal component diameter. We can assume $D = O(\log n)$. Thus, we can find the connected components of both

$T[V_1]$ and $T[V_2]$ in $O(\log \log n)$ rounds. After having computed the connected components, we create one tuple per node with the first entry being equal to its component identifier and the second entry being equal to the identifier of the node itself. By sorting these tuples lexicographically in $O(1)$ rounds using the algorithm of [23], we can identify the size of each connected component. Furthermore, one can assign each connected component a number of machines proportional to its size, such that the total memory capacity of the assigned machines is $O(n)$. That is, each large connected component with $\eta \geq n^\delta$ nodes gets assigned $\Theta(\log^2(n)\eta/n^\delta)$ many machines. After relabeling the vertices of the connected component with unique identifiers between 1 and η, the machines assigned to such a large component can 2-color the component in $O(\log \log n)$ rounds. Each small component with less than n^δ vertices is stored on one machine, which may be responsible for multiple small components. A 2-coloring of such a small component can be computed locally on the corresponding machine.

3.1.4 Rooting and 2-coloring a tree with diameter $O(\log n)$

Next, we focus on a single connected component in either $T[V_1]$ or $T[V_2]$ and show how to root the connected component by using $O(\log^2 n)$ invocations of the connected component algorithm in parallel. As stated in the previous paragraph, we assume that each node of the connected component has a unique identifier in $[\eta]$, with η denoting the size of the component. We pick an arbitrary node as the root, i.e., the root with identifier 1. We want each node, except for the root, to learn which neighboring node is its parent. Our algorithm relies on the following simple, but crucial observation: the parent of a node is the only neighboring node that, when deleting a subset of the edges in the tree, can still remain in the same connected component as the root, while the node itself got disconnected from the root.

To make use of this observation, we remove each edge independently with a probability of $1/\log(n)$. Let v be an arbitrary node. Notice that if v is disconnected from the root but a neighbor u of v is connected to the root, then one can deduce that u is the parent of v. Let $p(v)$ be the parent of v. Node $p(v)$ remains in the same connected component as the root if all of the at most $O(\log n)$ edges on the path from $p(v)$ to the root remain in the graph. This happens with a probability of at least $(1 - 1/\log n)^{O(\log n)} = \Omega(1)$. Conditioning on this event, node v gets disconnected from the root if the edge between v and $p(v)$ gets removed, which happens with a probability of $1/\log(n)$. If both of these happen, we have a good event and v can identify its parent $p(v)$. Thus, we can conclude that a fixed vertex v can determine its parent with a probability of $\Omega(1/\log(n))$.

By running this procedure $\Theta(\log^2 n)$ times in parallel (and independently), we can conclude – by a Chernoff bound and a union bound over all vertices – that each vertex determines its parent with high probability. Hence, we can 2-color the connected component using $O(\min(\eta, n^\delta))$ memory per machine and $O(\log^2(n)\eta)$ global memory.

Once we have this orientation towards the root, and since the diameter of the tree is $O(\log n)$, each node can compute the distance of it from the root using $\Theta(\log \log n)$ steps of pointer jumping, as we explain next.

▶ **Lemma 19.** *Given a tree of depth $O(\log n)$ with η nodes, which is oriented towards the root, we can compute the distance of each node from the root in $O(\log \log n)$ rounds, using $O(\min(\eta, n^\delta))$ local memory and $O(\eta \log^2 n)$ global memory.*

Proof. We use a *pointer jumping* idea, along with some sorting subroutines of MPC. For each node v, we keep track of one pointer $p(v)$ which we initially set equal to its parent node (for the root r the pointer points to the node itself). Also, we define the ancestor list $AL(v)$, which is a set that initially only contains the node v.

Then, each iteration of pointer jumping consists of two steps: first, each node v updates its ancestor list as $AL(v) \leftarrow AL(v) \cup AL(p(v))$. Then, in the second step, we redefine $p(v) \leftarrow p(p(v))$. It is clear that after $O(\log \log n)$ iterations, $p(v)$ is equal to the root of its component and $AL(v)$ is equal to the set of all of its ancestors, out of which v learns its distance to the root.

Implementing the pointer jumping needs some care. In each iteration of pointer jumping, each node v needs to learn the value of $p(u)$, where $u = p(v)$. While node v wants to learn only one value, node u might have to inform many nodes about the value of $p(u)$. Essentially the same procedure needs to be done for learning $AL(u)$, so we focus on $p(u)$.

We use a basic sorting subroutine [23], which runs in constant time. For the purpose of sorting, for each node v, define an item $\langle p(v), v \rangle$. Here, v and $p(v)$ are the respective identifiers, which are numbers in $\{1, \ldots, \eta\}$. Moreover, add for each node v two extra items $\langle v, -\infty \rangle$ and $\langle v, +\infty \rangle$. Sort all these items lexicographically. At the end of the sorting, each machine that holds an item knows the rank of this item in the sorted list.

Once we have the items sorted, for each node u, in the sorted order, the items start with $\langle u, -\infty \rangle$, end with $\langle u, +\infty \rangle$, and in between these two are the entries of all nodes v such that $p(v) = u$. As a result, the machine that holds node u and generated the two items $\langle u, -\infty \rangle$ and $\langle u, +\infty \rangle$ knows the ranks of these two items in the sorted order.

Now, split the items among the machines, in an ordered way, so that the ith machine receives the items with rank $[S(i-1) + 1, Si]$. Finally, use a broadcast tree of constant depth [23] so that the machine that holds node u informs the machines that, in the sorted order of items, hold items with u as the first entry, about the value of $p(u)$. This way, any machine that holds an item $\langle p(v), v \rangle$ where $p(v) = u$ learns $p(u)$. Hence, that machine can add this value $p(u)$ as a third field to create $\langle p(v), v, p(u) \rangle$. At the very end, we send these three-entry messages back to the machine that initially held v. Hence, node v now knows the value of $p(u)$ where $u = p(v)$.

The same process can be used so that node v learns $AL(p(u))$, where $u = p(v)$. In that case, the message has $O(\log n)$ words – the maximum number of ancestors in an $O(\log n)$ depth tree. The algorithm uses $O(\min(\eta, n^\delta))$ local memory and $O(\eta \log^2 n)$ global memory, thus proving the lemma. ◀

Once the nodes know their distance from their respective root, a 2-coloring is immediate: nodes at odd distances get one color and nodes at even distances get the other. This gives a separate 2-coloring of each of $T[V_1]$ and $T[V_2]$. Overall, this leads to a 4-coloring of $T[V_1 \cup V_2]$.

Remark about a work of Brandt et al. [14]. The main result of Brandt et al. [14] is a $\Theta(\log^3 \log n)$ algorithm for finding a Maximal Independent Set on trees. As a subroutine, they used a simple, deterministic (graph exponentiation style) algorithm to root a tree with a diameter of D in $O(\log D \log \log n)$ rounds, under the assumption of initially having $\Omega(D^3)$ global memory per node. Using their rooting algorithm as a black-box in our approach, we can directly improve this complexity. Namely, after the step of reducing the number of vertices to $O(n/\operatorname{polylog} n)$, as explained in Section 3.1.1, and randomly splitting the vertices into two sets V_1 and V_2, as explained in Section 3.1.2, instead of our proposed rooting algorithm, one could invoke the rooting procedure of Brandt et al. [14], which runs in $\Theta(\log^2 \log n)$ rounds. Overall, this gives a 4-coloring in $O(\log^2 \log n)$ rounds. Having this coloring, we can easily solve MIS in $O(1)$ additional rounds, simply by iterating through color classes of nodes and greedily adding nodes to the MIS.

References

1 Noga Alon, László Babai, and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *J. Algorithms*, 7(4):567–583, 1986. `doi:10.1016/0196-6774(86)90019-2`.

2 Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *Proceedings of the 59th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 674–685, 2018. `doi:10.1109/FOCS.2018.00070`.

3 Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab Mirrokni, and Cliff Stein. Coresets meet EDCS: Algorithms for matching and vertex cover on massive graphs. In *Proceedings of the 30th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1616–1635, 2019. `doi:10.1137/1.9781611975482.98`.

4 Leonid Barenboim and Michael Elkin. Sublogarithmic Distributed MIS Algorithm for Sparse Graphs Using Nash-Williams Decomposition. *Distributed Computing*, 22(5-6):363–379, 2010. `doi:10.1007/s00446-009-0088-2`.

5 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM*, 63(3):20:1–20:45, 2016. `doi:10.1145/2903137`.

6 MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Vahab S. Mirrokni. Brief announcement: Mapreduce algorithms for massive trees. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 162:1–162:4, 2018. `doi:10.4230/LIPIcs.ICALP.2018.162`.

7 MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Vahab S. Mirrokni. Massively parallel dynamic programming on trees. *arXiv preprint*, 2018. `arXiv:1809.03685`.

8 Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in Parallel Query Processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 212–223, 2014. `doi:10.1145/2594538.2594558`.

9 Soheil Behnezhad, Sebastian Brandt, Mahsa Derakhshan, Manuela Fischer, MohammadTaghi Hajiaghayi, Richard M. Karp, and Jara Uitto. Massively parallel computation of matching and MIS in sparse graphs. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 481–490, 2019. `doi:10.1145/3293611.3331609`.

10 Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Richard M. Karp. Massively Parallel Symmetry Breaking on Sparse Graphs: MIS and Maximal Matching. *arXiv preprint*, 2018. `arXiv:1807.06701`.

11 Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Łącki, and Vahab Mirrokni. Near-optimal massively parallel graph connectivity. In *Proceedings of the 60th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1615–1636, 2019. `doi:10.1109/FOCS.2019.00095`.

12 Soheil Behnezhad, MohammadTaghi Hajiaghayi, and David G. Harris. Exponentially faster massively parallel maximal matching. In *Proceedings of the 60th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1637–1649, 2019. `doi:10.1109/FOCS.2019.00096`.

13 Sebastian Brandt, Manuela Fischer, and Jara Uitto. Matching and MIS for uniformly sparse graphs in the low-memory MPC model. *arXiv preprint*, 2018. `arXiv:1807.05374`.

14 Sebastian Brandt, Manuela Fischer, and Jara Uitto. Breaking the linear-memory barrier in MPC: Fast MIS on trees with strongly sublinear memory. In *Proceedings of the 26th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 124–138, 2019. `doi:10.1007/978-3-030-24922-9_9`.

15 Richard Cole and Uzi Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 206–219, 1986.

16   Artur Czumaj, Peter Davies, and Merav Parter. Graph sparsification for derandomizing massively parallel computation with low space. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 175–185, 2020. `doi:10.1145/3350755.3400282`.

17   Artur Czumaj, Jakub Łącki, Aleksander Mądry, Slobodan Mitrović, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. *SIAM Journal on Computing*, pages STOC18–1–STOC18–44, 2019. `doi:10.1137/18M1197655`.

18   Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008. `doi:10.1145/1327452.1327492`.

19   Mohsen Ghaffari. Distributed MIS via all-to-all communication. In *Proceedings of the 36th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 141–149, 2017. `doi:10.1145/3087801.3087830`.

20   Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for MIS, matching, and vertex cover. In *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, 2018. `doi:10.1145/3212734.3212743`.

21   Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. In *Proceedings of the 60th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1650–1663, 2019. `doi:10.1109/FOCS.2019.00097`.

22   Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Proceedings of the 30th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1636–1653, 2019. `doi:10.1137/1.9781611975482.99`.

23   Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, Searching, and Simulation in the MapReduce Framework. In *Proceedings of the 22nd International Symposium on Algorithms and Computation*, pages 374–383, 2011. `doi:10.1007/978-3-642-25591-5_39`.

24   Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, 2007. `doi:10.1145/1272996.1273005`.

25   Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A Model of Computation for MapReduce. In *Proceedings of the 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010. `doi:10.1137/1.9781611973075.76`.

26   Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *Journal of the ACM*, 63(2):17:1–17:44, 2016. `doi:10.1145/2742012`.

27   Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: A method for solving graph problems in MapReduce. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 85–94, 2011. `doi:10.1145/1989493.1989505`.

28   Christoph Lenzen and Roger Wattenhofer. Brief Announcement: Exponential Speed-Up of Local Algorithms Using Non-Local Communication. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 295–296, 2010. `doi:10.1145/1835698.1835772`.

29   Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. `doi:10.1137/0221015`.

30   Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986. `doi:10.1137/0215074`.

31   Crispin St. John Alvah Nash-Williams. Decomposition of finite graphs into forests. *J. London Math. Soc.*, 39(1):12, 1964. `doi:10.1112/jlms/s1-39.1.12`.

32   Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and circuits (on lower bounds for modern parallel computation). *Journal of the ACM*, 65(6):41:1–41:24, 2018. `doi:10.1145/3232536`.

33   Tom White. *Hadoop: The Definitive Guide*. O'Reilly, 2012.

34   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*, 2010.