Coloring Sparse Graphs with 3 Colors in the Massively Parallel Computation (MPC) Model Using Strongly Sublinear Memory

Rustam Latypov

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Helsinki 21.12.2020

Thesis supervisor and advisor:

Prof. Jara Uitto

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Rustam Latypov

Title: Coloring Sparse Graphs with 3 Colors in the Massively Parallel Computation (MPC) Model Using Strongly Sublinear Memory

Date: 21.12.2020 Language: English Number of pages: 5+22

Department of Mathematics and Systems Analysis

Professorship: –

Supervisor and advisor: Prof. Jara Uitto

The question of what problems can be solved, and how efficiently, has always been at the core of theoretical computer science. One such fundamental problem is graph coloring; it is well researched and has numerous applications in areas of computer science such as scheduling and pattern matching.

The challenges faced when designing graph coloring algorithms are dictated by the underlying graph family, the number of colors allowed, and the model of computation. In this work we consider the graph family of trees and the distributed Massively Parallel Computation (MPC) model, introduced by Karloff et al. [KSV10]. Our contribution to the field of distributed computing is a deterministic strongly sublinear MPC algorithm for 3-coloring unbounded degree trees with n nodes in $O(\log \log n)$ time. To the best of our knowledge, this is the current state-of-the-art algorithm, improving on the work of Ghaffari et al. [GGJ20]. It is loosely based on two previous works by Brandt et al. [BFU18, BFU19].

Before computing a 3-coloring, our algorithm partitions the input tree into disjoint node sets $H_1, H_2, \ldots, H_l$, in $O(\log \log n)$ time. For each node $v \in H_i$, it holds that v has at most two neighbors in the set $\bigcup_{j=i}^{l} H_j$. We consider this partitioning in and of itself an important contribution, since it has the potential of being a useful subroutine in future algorithms. For example, a similar technique was used by Chang et al. [CP17] in their seminal paper to establish an important time hierarchy theorem for the distributed LOCAL model on trees.

Keywords: Graph problems, Trees, 3-coloring, Distributed computing, Massively Parallel Computation, MPC, Strongly sublinear memory

AALTO-YLIOPISTO
PERUSTIETEIDEN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Rustam Latypov

Työn nimi: Harvojen verkkojen värittäminen kolmella värillä Massiivisen rinnakkaislaskennan (MPC) mallissa käyttäen vahvasti alilineaarista muistia

Päivämäärä: 21.12.2020 Kieli: Englanti Sivumäärä: 5+22

Department of Mathematics and Systems Analysis

Professuuri: –

Työn valvoja ja ohjaaja: Prof. Jara Uitto

Teoreettisen tietojenkäsittelytieteen tärkeimpiä tavoitteita on tutkia, mitkä ongelmat ovat ratkaistavissa, ja kuinka tehokkaasti nämä ratkeavat. Eräs sellainen ongelma on verkkojen väritysongelma, jolla on lukuisia sovelluksia tietojenkäsittelytieteen ongelmissa, kuten aikataulutusongelmissa ja hahmontunnistuksessa.

Verkkojen värittämiseen liittyvät haasteet riippuvat verkkoperheestä, sallitusta värien määrästä ja valitusta mallista. Tässä työssä keskitytään puiden verkkoperheeseen ja hajautettuun Massiivisen rinnakkaislaskennan (MPC) malliin, jonka esittivät Karloff ym. [KSV10]. Työn tulos on MPC-mallissa vahvasti alilineaarista muistia käyttävä deterministinen 3-väritysalgoritmi rajoittamattoman asteen n-solmuisissa puissa, jonka ajoaika on $O(\log \log n)$. Kyseessä on läpimurtoalgoritmi, joka parantaa Ghaffari ym. [GGJ20] esittämiä tuloksia ja pohjautuu löyhästi Brandt ym. [BFU18, BFU19] esittämiin algoritmeihin.

Ratkaistaessa 3-väritysongelmaa algoritmi ensin osittaa puun solmut erillisiin joukkoihin $H_1, H_2, \ldots, H_l$, ajassa $O(\log \log n)$. Osituksessa pätee, että jokaisella solmulla $v \in H_i$ on enintään kaksi naapuria joukossa $\bigcup_{j=i}^{l} H_j$. Kyseinen ositus on esille nostamisen arvoinen ja on myös sellaisenaan mittava tulos, koska osituksella on paljon potentiaalista käyttöä myös muissa algoritmeissa. Esimerkiksi Chang ym. [CP17] hyödynsivät vastaavanlaista ositusta mullistavassa artikkelissaan, missä he johtivat tärkeän hajautetun LOCAL-mallin aikahierarkian puissa.

Avainsanat: Verkko-ongelmat, Puut, 3-väritys, Hajautettu laskenta, Massiivinen rinnakkaislaskenta, MPC, Vahvasti alilineaarinen muisti

Preface

I want to thank my supervisor Professor Jara Uitto for his guidance and patience.

Helsinki, 21.12.2020

Rustam Latypov

Contents

1 Introduction

Many fascinating systems in the world are large complex networks, such as the human society, the internet, or the brain. Such systems have in common that they are composed of multiple individual entities, so-called *nodes*, and pairs of entities have some kind of bilateral connections, so-called *edges*. Nodes that have an edge in common are called *neighbors*. Nodes can represent human beings or neurons in the brain, and edges can represent friendships or synapses. A set of nodes together with a set of edges constitutes a *graph*. Hence, networks are often modeled by graphs.

Complex networks are often of local nature; nodes can directly communicate only with neighboring nodes, which is often a small fraction of the total number of nodes. For instance, most human communications happen between acquaintances or family members, and neurons are linked via synapses to a relatively small number of other neurons. However, despite each node being restricted to local communication, the system as a whole is supposed to work towards some kind of global goal or solution. The field of distributed computing studies how networks can achieve global solutions despite locality.

In a distributed setting, problems are solved under certain computational models that somehow limit the abilities of nodes and their pairwise communication, capturing the real-life conditions of the underlying networks. Distributed algorithms solve global problems involving multiple nodes and edges from the point of view of single nodes such that every node runs the same set of instructions; upon termination, each node knows its own part of the global solution. Note that as opposed to centralized algorithms, there is no all-seeing entity that controls the progression of the algorithm.

Since large complex networks can be modeled by graphs, real-life network problems can be posed and solved as distributed graph problems. The prevalence of such networks has resulted in heavy interest in distributed algorithms for fundamental graph problems.

1.1 The LOCAL Model

The LOCAL model was introduced in the 90's by Linial [Lin87, Lin92] and it has been instrumental for studying distributed algorithms ever since. As the name suggests, the model captures the locality of an underlying communication network, modeled by an simple connected graph. The nodes correspond to computational entities such as machines, and the edges correspond to bidirectional communication links. Each node has a unique identifier (ID) of size at most $O(\log n)$ bits. Initially, each node is only aware of its neighbors in the network. Each node is only responsible for outputting its own part of the global solution. Each node can only communicate with its neighbors by sending messages of unlimited size through edges. Nodes are infinitely powerful, meaning that all computation of a node, called *local computation*, happens instantly, no matter how difficult the computation may be. In addition, the memory of each node, called *local memory*, and the memory of the whole network, called *global memory*, is unbounded.

All computation and communication happens synchronously. There is a global clock that ticks such that with each clock tick every node performs three steps:

1. Send messages to neighbors
2. Receive messages from neighbors
3. Perform local computation

A clock tick is called a *communication round*, or simply a *round*. The system as a whole is fault-free, meaning that nodes do not crash and messages are not lost, delayed or corrupted during the execution of any algorithm. While asynchronous communication and fault-tolerance are practical and important concerns, they are ignored in favor of a cleaner model. Since local computation happens instantaneously, the time complexity measure of a LOCAL algorithm is the number of rounds it takes for an algorithm to terminate.

1.2 The MPC Model

The Massively Parallel Computation (MPC) model was introduced in 2010 by Karloff et al. [KSV10] and it has seen some refinements over the years [GSZ11, BKS14]. Despite of its young nature, it already serves as the basis for the systematic study of massively parallel algorithms. It constitutes a common abstraction of several popular large-scale computation frameworks such as Dryad [IBY+07], MapReduce [DG08], Spark [ZCF+10], and Hadoop [Whi12]. The MPC model is similar to the LOCAL model, with the exception of global communication, message size restrictions, and memory restrictions. It is important to note that the MPC model is neither strictly more restrictive nor strictly less restrictive than the LOCAL model. Global communication is one of the more interesting aspects of the MPC model; instead of only being able to communicate with neighbors in steps 1 and 2 as in the LOCAL model, each node can communicate with any node.

The MPC model is often defined using three parameters – the input data size N, the number of machines M available for computation, and the memory per machine S. The memory per machine is called *local memory* and the summation of the local memories across all machines is called *global memory*. Naturally, problems are easier and faster to solve the larger S is. In the extreme example when $S \gg N$, one could load the whole input on a single machine and solve it instantaneously using infinitely powerful local computation, yielding a trivial solution. However in practice, S is *not* chosen by the algorithm designer but rather dictated by the problem setting, the current state-of-the-art hardware, and the available computational resources. Typically, S is a polynomial of N such that $S = N^\delta$ for some $0 < \delta < 1$. The total size of sent and received messages of a machine in every communication round is not allowed to exceed its local memory capacity, which is bounded by S.

When solving graph problems in the MPC model, a graph with n nodes and m edges is given as input and each edge in the graph is assigned to a machine. Since the unique ID of a node is of size $O(\log n)$ and storing an edge is equivalent to storing

two nodes, storing an edge requires $O(\log n)$ memory. If follows that the input is of size $N = \widetilde{\Theta}(m)$, which is shorthand for $O(m \log^k m)$ for some k. Since initially, the edges are distributed across the machines, the number of machines M is required to be at least in the order of N/S so that there are enough machines to store the whole input. What is left is to define S; graph algorithms in the MPC model can typically be classified into three memory regimes, each facing different technical challenges:

Strongly superlinear memory: Memory $S = n^{1+\delta}$, for some $\delta > 0$.

Near-linear memory: Memory $S = \widetilde{\Theta}(n)$.

Strongly sublinear memory: Memory $S = n^{\delta}$, for some $0 < \delta < 1$.

Low-memory MPC algorithms reside in the strongly sublinear memory regime and are more desirable ones [GGJ20]. The focus of low-memory MPC algorithm design is minimizing the number of rounds, while also minimizing global memory use.

1.3 Practical Concerns

When solving graph problems in the MPC model, edges are initially distributed across the machines, preferably in a balanced way such that all machines have around the same number of edges in memory. Using universal hashing [LW79, WL81], one can accomplish just that, in $O(1)$ rounds.

When N is sufficiently large and S sufficiently small, it may not be feasible to have in the order of N/S physical machines. However, since each machine can simulate a number of virtual machines, algorithms can be designed with the assumption that each (virtual) machine has memory S and corresponds to exactly one node and all of its edges. If a node has so many edges that it cannot be stored on a single machine, one can apply the workaround by Brandt et al. [BFU19].

1.4 Related Works

Fundamental graph problems [Die17] such as maximal independent set (MIS), maximal matching, minimum matching and minimum vertex cover have enjoyed a lot of attention from the community in all memory regimes of the MPC model. While simple $O(\log n)$ algorithms follow from classic literature for MIS and maximal matching in the LOCAL model [NLA86, Lub86], the main objective in the MPC model is to obtain faster algorithms than in the LOCAL model, ideally just constant or $O(\log \log n)$ time. Over the years, there has been steady progress towards this, across all memory regimes.

Lattanzi et al. [LMSV11] presented $O(1)$ time algorithms for MIS and maximal matching in the strongly superlinear regime. The progress towards lower memory regimes slowed down afterwards, until a breakthrough by Czumaj et al. [CLM+18], who gave an $O(\log^2 \log n)$ time algorithm for $(2+\epsilon)$-approximation of maximum

matching in the near-linear memory regime. Soon after, for the same memory regime, Assadi et al. [ABB+19] gave an $O(\log\log n)$ time algorithm for $(1+\epsilon)$-approximation of maximum matching and an $O(1)$-approximation of minimum vertex cover. Concurrently, but independently, Ghaffari et al. [GGK+18] gave $O(\log\log n)$ time algorithms for $(1+\epsilon)$-approximation of maximum matching, $(2+\epsilon)$-approximation of minimum vertex cover, and MIS. Finally, Behnezhad et al. [BHH19] gave an $O(\log\log n)$ time algorithm for maximal matching.

For the more rigorous strongly sublinear memory regime, there has also been some progress but much slower. For general graphs with maximum degree Δ, Ghaffari and Uitto [GU19] gave $\widetilde{O}(\sqrt{\log\Delta})$ time algorithms for MIS, maximal matching, $(1+\epsilon)$-approximation of maximum matching, and 2-approximation of minimum vertex cover, which remain the best known. Only when considering special graph families, namely trees and other sparse graphs bounded by arboricity λ, this bound can be outperformed. Recall that the arboricity of a graph is the minimum number of forests into which its edges can be partitioned. Behnezhad et al. [BBD+19] gave MIS and maximal matching algorithms such that they first reduce problems in graphs with arboricity λ to corresponding problems in graphs with maximum degree $\text{poly}(\lambda, \log n)$, and then by invoking the aforementioned algorithm of Ghaffari and Uitto [GU19], they arrive at MIS and maximal matching algorithms with time complexity $O(\sqrt{\log\lambda}\cdot\log\log\lambda + \log^2\log n)$. Finally, a recent paper from Ghaffari et al. [GGJ20] presents, to the best of our knowledge, the current state-of-the-art time complexities of $O(\sqrt{\log\lambda}\cdot\log\log\lambda + \log\log n)$ for MIS and maximal matching. As a second contribution, they provide an $O(\log\log n)$ time algorithm for 4-coloring trees.

1.5 Our Contributions

Graph coloring in distributed computing has been a well-researched topic ever since Lineal [Lin87] introduced the LOCAL model and gave an $O(\log^* n)$ time $O(\Delta^2)$-coloring algorithm for graphs with maximum degree Δ. More recently, in the near-linear memory regime of the MPC model, Assadi et al. [ACK19] gave a constant time $(\Delta+1)$-coloring algorithm. In the sublinear memory regime, Ghaffari et al. [GGJ20] gave an $O(\log\log n)$ time algorithm for 4-coloring trees.

Our contribution is formalized in Theorem 1.1, which is an improvement over the randomized 4-coloring algorithm of Ghaffari et al. [GGJ20]. We present a deterministic $O(\log\log n)$ time algorithm for 3-coloring unbounded degree trees with n nodes. We achieve this by using a hybrid approach of two previous papers by Brandt et al. [BFU18, BFU19]. As an intermediate result, we construct an H-partition of trees in $O(\log\log n)$ time, which is a node partition $\{H_1, H_2, \ldots, H_l\}$ such that each node $v \in H_i$, has at most 2 neighbors in the set $\bigcup_{j=i}^{l} H_j$. In literature, the method of constructing an H-partition of trees is referred to as *rake-and-compress* [CP17] and *procedure partition* [BE13]. We consider our partitioning in and of itself an important contribution, since it has the potential of being a useful subroutine in future algorithms. For example, a similar technique was used by Chang et al. [CP17] in their seminal paper to establish an important time hierarchy theorem for

the LOCAL model on bounded degree trees.

Theorem 1.1. *There is a deterministic low-memory MPC algorithm that computes a 3-coloring of unbounded degree trees in $O(\log \log n)$ time where $S = O(n^{\delta})$ and $M \cdot S = O(n^{1+\delta})$. The result follows almost immediately after constructing a node partition $\{H_1, H_2, \ldots, H_l\}$ of unbounded degree trees in $O(\log \log n)$ time such that each node $v \in H_i$, has at most 2 neighbors in the set $\bigcup_{j=i}^{l} H_j$.*

Proof. Follows from Theorems 4.9 and 4.13. □

2 Graph Theory

One cannot approach graph problems without first understanding the relevant graph theoretical concepts. Some notations and definitions related to graph theory are well established, and some are not. We give a brief introduction of the topic to ease the understanding of our results and mitigate possible notational confusion.

In this work, we exclusively consider finite simple sparse graphs with unbounded degree and arboricity $\lambda = 1$, i.e., unbounded degree trees and forests. In this rather restrictive graph family, we are interested in a proper node 3-coloring, referred to simply as a 3-coloring.

2.1 Basics

A simple graph is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$. Hence, the elements of E are 2-element subsets of V. To avoid notational ambiguities, we shall always assume that $V \cap E = \emptyset$. The elements of V are called nodes (vertices, points), and the elements of E are called edges (links, lines, arcs, bonds). The number of nodes of a graph G is denoted by $|V|$ and the number of edges is denoted by $|E|$. A graph is finite if $|V|$ is a finite set; it follows that $|E|$ is also finite. The usual way to picture a graph is by drawing a dot for each node and joining two of these dots by a line if the corresponding two nodes form an edge.

The node set of a graph $G = (V, E)$ is referred to as $V(G)$ and its edge set as $E(G)$; often V and E for short, when it is clear from context that the underlying graph is G. We shall not always distinguish strictly between a graph and its node or edge set. For example, we may speak of a node $v \in G$ (rather than $v \in V(G)$), an edge $e \in G$, and so on.

If $e = \{v, u\} \in E$, then v and u are adjacent, or neighbors. Also, v and u are incident to e and vice versa. Two edges $e, f \in E$ such that $e \neq f$ are adjacent if $e \cap f \neq \emptyset$, i.e., they have one node in common. An edge is directed, or oriented, if there is an order between the two nodes, e.g., parents and children. A graph is oriented, or directed, if all of its edges are directed. A graph is partially oriented if at least one edge is oriented and at least one is not. An orientation of a graph G is the set of instructions that orients every edge in E. The usual way to picture an orientation of an edge is to draw an arrow as opposed to a line.

2.2 Terminology

We introduce a few essential graph theoretical concepts.

2.2.1 Degree

The degree of a node $v \in G$ is

$$\deg(v) = |\{u \in V : \{v, u\} \in E\}|.$$

The degree of node v is equivalent to the number of neighbors of v. The maximum degree of a graph G is denoted by Δ.

2.2.2 Paths

A path is a non-empty graph $P = (V, E)$ of the form

$$V = \{v_1, v_2, \ldots, v_k\}$$
$$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}\},$$

where all nodes v_i are distinct. The length of a path is $|E|$.

2.2.3 Subgraphs and Partitions

Let $G = (V, E)$ and $H = (V_2, E_2)$ be two graphs. If $V_2 \subseteq V$ and $E_2 \subseteq E$, we say that H is a subgraph of G. If $V_2 = V$, we say that H is a spanning subgraph of G. If $V_2 \subseteq V$ and $E_2 = \{\{v, u\} \in E : u, v \in V_2\}$, we say that H is an node-induced, or simply induced, subgraph. More specifically, H is the subgraph of G induced by the node set V_2. If $E_2 \subseteq E$ and $\bigcup E_2 = V_2$, we say that H is an edge-induced subgraph. More specifically, H is the subgraph of G induced by the edge set E_2.

A set $\mathcal{V} = \{V_1, V_2, \ldots, V_k\}$ of disjoint subsets of nodes V such that $\bigcup \mathcal{V} = V$ and $V_i \neq \emptyset,\ \forall i$ is a node-partition of graph G. Analogously, a set $\mathcal{E} = \{E_1, E_2, \ldots, E_k\}$ of disjoint subsets of edges E such that $\bigcup \mathcal{E} = E$ and $E_i \neq \emptyset,\ \forall i$ is an edge-partition of graph G.

2.2.4 Connectivity and Distances

A connected component of a graph G is a subgraph H where there exists a path between all nodes. The distance $d(v, u)$ in G between nodes $v, u \in G$ is the length of a shortest path from v to u in G. If no such path exists, we set $d(v, u) \coloneqq \infty$. Note that $d(v, v) = 0$ for any node v. The greatest distance between any two nodes in G is the diameter of G, denoted by $\operatorname{diam}(G)$.

For each node $v \in G$ and for every $r \in \mathbb{N}$, we define the r-hop neighborhood of v as

$$N(v, r) = \{u \in V : d(v, u) \leq r\}.$$

The topology of a neighborhood $N(v, r)$ of v is the subgraph of G induced by the node set $N(v, r)$. The radius of a neighborhood $N(v, r)$ is r.

2.3 Trees

A tree T is a simple connected graph such that there exists exactly one path from each node $v \in T$ to any node $u \in T$. A forest F is a disjoint union of trees. Trees and forests are related to the notion of the arboricity λ of a graph. The arboricity of a graph is the minimum number of forests into which its edges can be partitioned. The arboricity of a graph is a common measure of how dense, or sparse, the graph is; graphs with many edges tend to have high arboricity and graphs with few edges tend to have low arboricity.

A rooted tree T is a tree where one special node is singled out, the root node $v \in T$. Rooted trees give rise to a unique orientation, as we can direct each edge such that it points either outwards or towards the root. The depth of a node $u \in T$ is $d(u, v)$. The root node has a depth of 0.

2.4 Graph Coloring

A coloring of a graph G is a special case of graph labeling such that each node or edge is assigned a color from the palette of colors $\{1, 2, \ldots, c\}$. Finding a graph coloring that is subject to certain constraints constitutes a graph coloring problem. One such constraint is that a coloring is proper. A proper node coloring is such that no two adjacent nodes have the same color. A proper edge coloring is such that no two adjacent edges have the same color. Another constraint is such that only k colors may be used. When only k colors are used, a coloring is called a k-coloring; the smaller k is, the harder the problem. Some coloring problems are trivial, some difficult and some unsolvable; depending on the underlying graph and on the number of colors allowed.

3 Algorithm Prerequisites

3.1 Outline

Our algorithm consists of running two algorithms consecutively: partitioning algorithm P and coloring algorithm C. Algorithm P will output an H-partition (defined next) of the input graph. As a result, each node will know to which partition (layer) it belongs to. Algorithm C will first partially orient its input graph, and then output a 3-coloring.

Even though our main result applies to trees, as we will soon notice, an input tree may shatter into a forest during the execution of P. This is not a problem, since we will show in Theorem 4.9 that algorithm P applies to both trees and forests. In addition, in Theorem 4.13, we will show that also algorithm C applies to both trees and forests. Note that this implies that the input graph is allowed to be a forest. Nevertheless, we formulate our results in terms of trees to be in line with literature.

3.2 H-partition

Definition 3.1 (H-Partition)**.** *An H-partition of graph $G = (V, E)$ with out-degree d, is a node-partition $\{H_1, H_2, \ldots, H_l\}$, such that each node $v \in H_i$, has at most d neighbors in the set $\bigcup_{j=i}^{l} H_j$. Parameter l is called the size of the H-partition and i is called the layer index (or simply layer) of v. See Figure 1 for an illustration of an H-partition.*

Definition 3.2 (Greedy H-Partition)**.** *A greedy H-partition is a subclass of H-partitions that is constructed with the following greedy algorithm. Assign all nodes with degree at most d into the current layer and then remove said layer. Repeat until the graph is empty. Another name for this technique is peeling. Note that Figure 1 illustrates a greedy H-partition.*

An H-partition by Definition 3.1 is not unique for its underlying graph, since nodes can easily be located in a higher layer than nesessary. This can result in $l = O(n)$, which is not something useful. Definition 3.2 characterizes a subclass of H-partitions called greedy, which are unique for their underlying graphs, since they can be constructed with a deterministic greedy algorithm. In addition to uniqueness, a greedy H-partition has other desirable properties, in particular its small size, presented in Lemma 3.3.

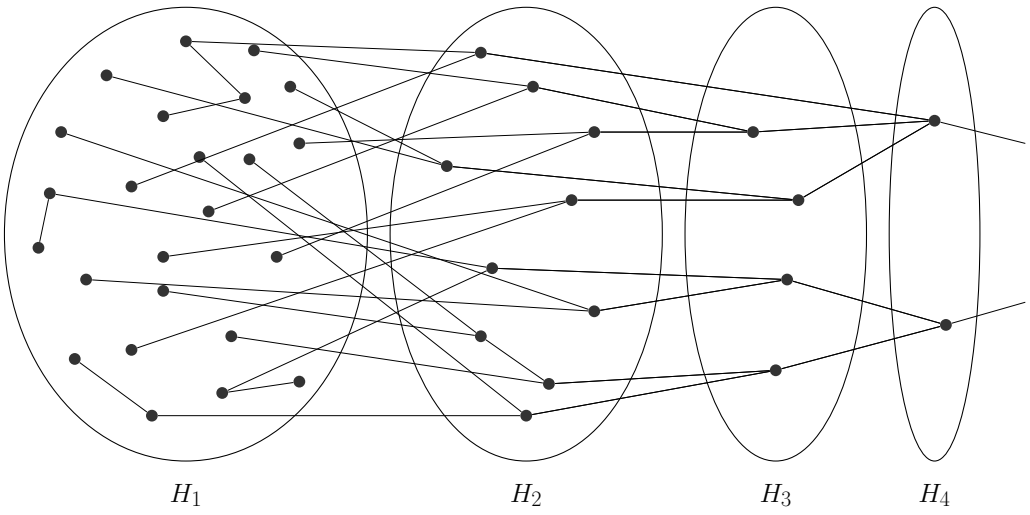

Figure 1: An illustration of a (greedy) H-partition with out-degree 2.

Lemma 3.3. *A greedy H-partition of a forest with out-degree 2 satisfies the following properties.*

(i) For all $1 \leq i \leq l$, if we remove all nodes in layer i from the set of nodes in layers $\geq i$, then the number of nodes drops by a factor of at least 3/2, i.e.,

$$\left| \bigcup_{j=i+1}^{l} H_j \right| \leq 2/3 \cdot \left| \bigcup_{j=i}^{l} H_j \right|.$$

(ii) There are at most $l = O(\log n)$ layers.

Proof. We prove *(i)* directly by observing that our greedy algorithm places a constant fraction of nodes in each layer. When $G = (V, E)$ is a forest,

$$3 \cdot |\{v \in V : \ \deg(v) \geq 3\}| \leq \sum_{v \in V} \deg(v) = 2|E| \leq 2|V|$$
$$|\{v \in V : \ \deg(v) \geq 3\}| \leq 2/3 \cdot |V|,$$

where the second-last equality is due the well known degree sum formula and the last inequality is due to the $|E| + 1 \leq |V|$ property of a forest [Die17]. We can prove *(ii)* using *(i)*. Let $n = |V|$. Since the total number of nodes decreases with each layer by a constant fraction,

$$(2/3)^l \cdot n \geq 1$$
$$\log_{3/2}(2/3)^l + \log_{3/2} n \geq 0$$
$$\log_{3/2} n \geq l,$$

proving the claim. □

3.3 Graph Exponentiation

The idea of graph exponentiation was first mentioned under the CONGESTED-CLIQUE model by Lenzen and Wattenhofer [LW10].

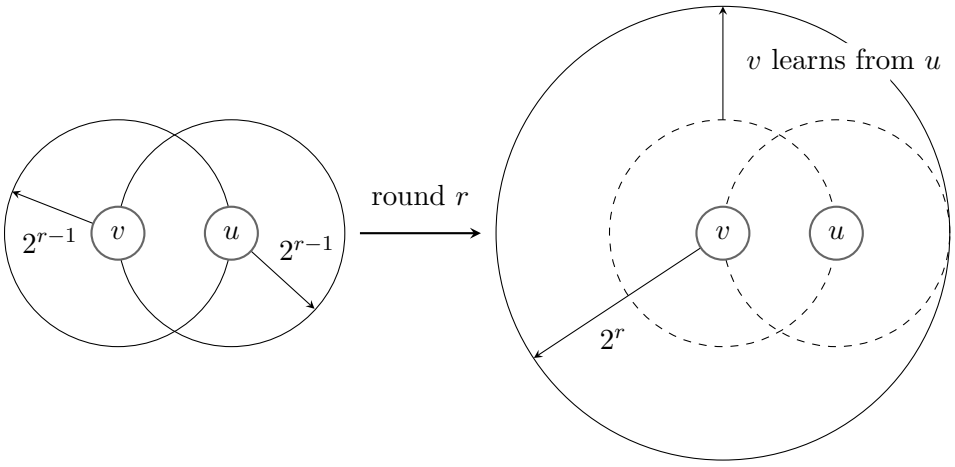

Figure 2: An illustration of v learning from u through graph exponentiation.

Lemma 3.4. *With $O(n)$ machines, disregarding local and global memory constraints, all nodes v in graph $G = (V, E)$ can learn $N(v, 2^r)$ in at most r MPC rounds through graph exponentiation.*

Proof. Assign a single machine M_v to each node $v \in G$. Let $N(v, r)$ denote the r-hop neighborhood of v at the beginning of the rth MPC round, when $N(v, r)$ is stored in M_v. Since initially the machine assigned to v knows the edges incident to v, it knows $N(v, 1)$. For rounds $r = \{1, \ldots, r'\}$: M_v sends its current neighborhood $N(v, r)$ to all nodes $u \in N(v, r)$. By induction over r, each machine M_v knows $N(v, 2^{r'})$ at the end of the r'th MPC round. Figure 2 illustrates the progression of the knowledge of v with the help of nodes u in round r. □

3.4 Simulation

Theorem 3.5 states the well known equivalence of time complexity and neighborhood topology proven by, e.g., Kuhn et al. [KMW16].

Theorem 3.5. *There is a one-to-one correspondence between the time complexity of distributed algorithms in the LOCAL model and the graph theoretic notion of neighborhood topology. Knowing the t-hop neighborhood enables the instant simulation of any t time LOCAL algorithm.*

3.5 Other

In the description of our algorithm, we rely to the following definitions.

Definition 3.6 (T)**.** *We define $T = (V, E)$ as the input tree.*

Definition 3.7 (T_i, T_{i-1})**.** *We define T_i as a single connected component of what is left of T in phase i. For technical reasons, we define T_{i-1} as the single connected component in phase $i - 1$ containing T_i.*

Definition 3.8 ($T_i(v)$)**.** *If T_i was oriented, $T_i(v)$ would be a oriented subtree rooted at v containing v and all of its descendant. Since T_i is not oriented, we define $T_i(v)$ as a subtree rooted at v such that $(T_i \setminus T_i(v)) \cup v$ is connected.*

In the low-memory MPC model, constant δ is useful when explicitly deriving runtimes; on the other hand it is a nuisance to write down repeatedly. Therefore, for the remainder of this work, constant δ is occasionally treated with generous disregard.

4 Algorithm Implementation

As mentioned previously, our algorithm consists of running algorithms P and C consecutively. In the following, we present the formal descriptions of P and C in the low-memory MPC model.

4.1 Partitioning Algorithm P

Algorithm P computes an H-partition with out-degree 2 of the input tree T. The algorithm is given an integer B as an input parameter that describes the memory budget for each node v, i.e., an upper bound on the number of edges that fit into

the memory of v. The exact value of B is given later in the analysis together with the reasoning behind it. The execution of P is subdivided into phases (i), iterations (j), repetitions (r) and rounds (k). We assume that all nodes know the total number of nodes n before the algorithm starts. During the algorithm, node v removing itself from the graph simply means that v has figured out its own part of the solution, i.e., its *final layer*, and turns inactive (disabling all incident edges). When all nodes have removed themselves, the graph is empty and P terminates.

Phase $i \in \{1, \ldots, i'\}$:

Iteration $j \in \{1, \ldots, j'\}$ **of phase** i:

Repetition $r \in \{1, \ldots, r'\}$ **of iteration** j: Node v performs one step of graph exponentiation if, for the resulting neighborhood, it holds that $|N(v, 2^r)| \leq B$. If not, v does nothing.

Round $k \in \{1, \ldots, k'\}$ **of iteration** j: Node v computes the *current layer* of all nodes in its learned neighborhood.

- If successful in determining its own current layer, node v informs the nodes in its learned neighborhood of their layers and removes itself from the graph.
- If unsuccessful in determining its own current layer, and some node(s) inform v of its layer, v accepts the assigned layer and removes itself from the graph.
- If unsuccessful in determining its own current layer, and no node has informed v of its layer, v does nothing except wipe all information added in round k.

Each phase concludes with one *final elimination round*, where node v tries to compute its current layer. If successful, it removes itself from the graph.

Definition 4.1 (Current layer)**.** *The index of the layer in the greedy H-partition of the current graph with all inactive nodes and disabled edges ommitted.*

Definition 4.2 (Final layer)**.** *The index of the layer in the H-partition of the input graph.*

Graph exponentiation. During graph exponentiation, the radii of neighborhoods increases by a factor of at most 2 per repetition (if local memory does not run out). In particular, disregarding any budget constraints, after r repetitions of graph exponentiation, node v knows $N(v, 2^r)$ by Lemma 3.4.

Computing the current layer. The greedy algorithm from Definition 3.2 is of local nature. In can be implemented in the LOCAL model such that if node v has at most two neighbors, v informs them that it belongs to the current lowest layer and v removes itself from the graph. Hence, r rounds are needed for node v to determine if it belongs to layer at most r. The well known result of Theorem 3.5 gives rise to local simulation: if node v knows the topology of $N(v, 2^r)$, it can locally compute if

it belongs to layer at most 2^r. In addition to computing its own layer, v can also compute if $u \in N(v, 2^r)$ belongs to layer at most $x < 2^r$, if $N(u, x) \subset N(v, 2^r)$.

Note that the previous simulation arguments hold both for nodes which *did not* run out of local memory as well as for nodes which *did* run out of local memory in some repetition, i.e., got stuck.

Each node v can gather the topology of its learned neighborhood $N(v, 2^r)$ in two communication rounds by first requesting and then receiving information. The topology of $N(v, 2^r)$ always fits into memory, since $|N^d(v, 2^r)| \leq B$ and in trees, the number of edges is bounded by the number of nodes. Due to the equivalence of time complexity and neighborhood topology, node v can simulate the greedy algorithm for all nodes in $N(v, 2^r)$.

> **Layer at most 2^r:** If the layer of v is at most 2^r, v knows its exact layer after performing local simulation and can remove itself. Before removing itself however, node v informs nodes in $N(v, 2^r)$ of their layers.
>
> **Layer higher than 2^r:** If the layer of v is higher than 2^r, node v does not know its layer after performing local simulation. If some other node(s) u knows the layer of v and informs v of its layer; v accepts the assigned layer and removes itself. Note that multiple nodes can inform v of its layer; we will further show that all nodes inform the same layer.

Correctness. Informing neighborhood nodes of their layers is done in order to maintain the correctness of the H-partition. Consider node v that has simulated the greedy algorithm for all nodes in $N(v, 2^r)$ and knows its own layer. If all nodes in $N(v, 2^r)$ place themselves in the layers simulated by v, the H-partition with regards to these nodes and v will be correct.

Consider node u such that u knows the topology of $N(u, r_u)$ with radius r_u and fails to determine its layer after performing local simulation. Now consider node v that knows the topology of $N(v, r_v)$ with radius r_v and succeeds in determining its layer after performing local simulation. If v informs u of its layer, it must be that $u \in N(v, r_v)$. Because v knows the layer of u and u does not, it must be that $N(u, r_u) \subset N(v, r_v)$, since v and u simulate the same greedy algorithm. Therefore, if u accepts the assigned layer from v, the H-partition with regards to u and all other nodes in $N(u, r_u)$ will be correct.

Now consider two nodes v and w informing u of its layer. Since v and w both simulate the same greedy algorithm with regards to u, if they are able to compute the layer of u, they compute the same exact layer. It follows that if multiple nodes inform u of its layer, they inform the same layer.

Computing the final layer. In the algorithm description, we make the distinction between the current and the final layer. The final layer can be computed using the current layer as follows. In each round, we remove $2^{r'}$ layers, even if they are empty from the point of view of some nodes. This is done to keep nodes which got stuck in sync with the ones which did not. Now we observe that each time a node is removed

from the graph, the whole layer containing this node is removed. Also, each time a layer is removed, all lower layers are removed at the same time or earlier. Hence, by knowing the number of deleted layers so far and the current layer, a node can compute the final layer. Since the number of deleted layers is uniquely defined by the i, j, and r, we implicitly assume that nodes keeps track of i, j, and r and can compute their final layer.

Type of H-partition. If no nodes get stuck during the algorithm, a greedy H-partition of the whole input graph is obtained. If some nodes get stuck, we only obtain a regular H-partition. However, we can show that locally, in certain subtrees, a greedy H-partition is always obtained. Regardless of whether or not nodes get stuck, the runtime analysis of the following section holds.

4.2 Runtime of P

In algorithm P, parameter B denotes an upper bound on the number of edges that fit into the memory of v. We set $B = n^\delta$ and justify the choice in Theorem 4.9. We further present a number of lemmas in order to determine the runtime of algorithm P, finalized in Lemma 4.7. Recall Definitions 3.6, 3.7 and 3.8.

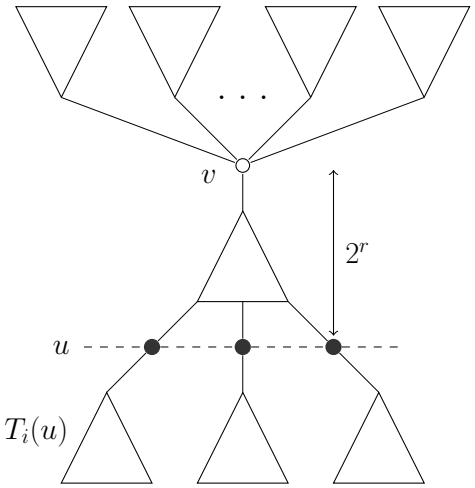

Figure 3: Subtree $T_i(v)$ rooted at v containing subtrees $T_i(u)$ rooted at nodes u.

Lemma 4.3. *Consider $T_i(v)$ during an arbitrary phase i such that $|T_i(v)| \leq n^\delta$. Nodes u in $T_i(v)$ with distance at least 2^r from v do not get stuck up to repetition r.*

Proof. Let r be some arbitrary non-negative integer and consider any node u in $T_i(v)$ with distance at least 2^r from v as illustrated in Figure 3. According to P, the radii of neighborhoods of nodes in T_i increase by a factor of at most 2 per repetition. Hence, after repetition r, all nodes contained in $N(u, 2^r)$ are actually also contained in $T_i(v)$. Since $|T_i(v)| \leq n^\delta$, the claim follows. □

Lemma 4.4. *Consider $T_i(v)$ during an arbitrary phase i. If $|T_i(v)| \leq n^\delta$, $T_i(v)$ gets removed in $O(1)$ iterations. Regarding the removal of node v, refer to Remark 4.5.*

Proof. Let us first focus on subtrees $T_i(u)$ rooted at nodes u with distance $2^{r'}$ from v in $T_i(v)$ as illustrated in Figure 3. Since $|T_i(v)| \leq n^\delta$, it holds that $|T_i(u)| \leq n^\delta$. According to Lemma 4.3, nodes in $T_i(u)$ do not run out of local memory during graph exponentiation. This ensures a greedy H-partitioning within each $T_i(u)$ and we can apply Lemma 3.3. In particular, the fact that there are $O(\delta \log n)$ layers within $T_i(u)$. In fact, there are $O(\delta \log n)$ layers in total within all subtrees $T_i(u)$, since they are disjoint.

Let's set $r' = \log\log n$ and $k' = \log\log n$. According to the algorithm description, in iteration j after r' repetitions and k' rounds, $k' \cdot 2^{r'} = \Omega(\log n)$ lowest layers are removed from the current graph. Combining this lower bound with the number of layers in $T_i(u)$, we conclude that all subtrees $T_i(u)$ get removed in $O(1)$ iterations.

After $O(1)$ iterations, we are possibly left with some nodes in $T_i(v)$ that were not in any $T_i(u)$. Since we have no guarantee on the topology of T_i outside $T_i(v)$, this may happen if, during graph exponentiation, these nodes get stuck. Since $r' = \log\log n$, what is left of $T_i(v)$ is at most of depth $\log n$. What is left of $T_i(v)$ gets removed in one additional iteration: after performing $r' = \log\log n$ repetitions and $k' = \log\log n$ rounds $T_i(v)$ should be empty, since by Lemma 4.6 the depth of $T_i(v)$ decreases from $\log n$ by a factor of $4/3$ with each round k. □

Remark 4.5. *Removal of root v during the removal of subtrees $T_i(v)$ in Lemma 4.4.*

After the final iteration of phase i, $T_i(v) \setminus v$ is removed and if v is still present, let it have s neighbors left. Due to the final elimination round at the end of phase i, if $s \leq 2$, v is able to compute its current layer index and remove itself from the graph.

Lemma 4.6. *Let $d \leq \log n$ denote the depth of $T_i(v)$. After $r' = \log\log n$ repetitions, nodes in $T_i(v)$ with distance at least $3/4 \cdot d$ from v remove themselves in one round.*

Proof. Recall that, disregarding any budget constraints, after r repetitions of graph exponentiation, node v knows $N(v, 2^r)$. Let $d' := \max_r \{2^r : 2^r \leq d\}$. Since $d \leq \log n$ and $|T_i(v)| \leq n^\delta$, after $r' = \log\log n$ repetitions, nodes w with depth at least $d - d'/2$ have learned a neighborhood containing all nodes connected to v through w. Hence, each node w is successful in computing its own layer index and removing itself in one round. Since $d' > d/2$, it holds that $d - d'/2 < 3/4 \cdot d$ and the claim follows. □

Lemma 4.7. *The runtime of P is $O(\log\log n)$.*

Proof. We proceed with proof by contradiction. We assume Lemma 4.8 and that T_i is non-empty and will arrive at a contradiction; this implies that if Lemma 4.8 holds, T_i is empty. Recall the definitions of T_i and T_{i-1} from Definition 3.7. Consider

phase i, where $|T_i| > 0$. Invoking Lemma 4.8 one time implies that $|T_{i-1}| \geq (n^\delta/3)$. Invoking Lemma 4.8 i times implies that

$$n = |T| = |T_0| \geq (n^\delta/3)^i = n^2 \cdot 3^{-2/\delta}, \quad \text{when } i = 2/\delta.$$

Hence, when $i' = O(1/\delta) = O(1)$, we reach a contradiction on the fundamental assumption that $|T| = n$. It follows that since Lemma 4.8 holds and $i' = O(1)$, T_i is empty, implying that after i' phases the whole graph is empty. By combining this observation with values $j' = O(1)$, $r' = \log\log n$, and $k' = \log\log n$ from Lemma 4.4, the claim follows. □

Lemma 4.8. *$|T_i| > 0$ implies that $|T_{i-1}| \geq |T_i| \cdot n^\delta/3$.*

Proof. Recall the definitions of T_i and T_{i-1} from Definition 3.7. Consider the start of phase i and some node v with at most two neighbors. Since node v has at most two neighbors in the start of phase i and it did not remove itself in the final elimination round of phase $i-1$, there had to exist a subtree $T_{i-1}(v)$ of some size, in the start of phase $i-1$. Our argument is simple: since v has at most two neighbors in the start of phase i, v was the root of subtree $T_{i-1}(v)$ and since v does not remove itself during phase $i-1$, by Lemma 4.4 and Remark 4.5, it must hold that $T_{i-1}(v) > n^\delta$. Note that we do not assume that all nodes in $T_{i-1}(v)$ are removed in phase $i-1$.

There are at least $|T_i|/3$ nodes v in T_i with at most two neighbors, by applying the same argument as in proof (i) of Lemma 3.3. It follows that $|T_{i-1}| \geq |T_i| \cdot n^\delta/3$, since all subtrees $T_{i-1}(v)$ are disjoint due to T_{i-1} being a tree. □

Theorem 4.9. *There is a deterministic low-memory MPC algorithm that constructs an H-partition of size $O(\log\log n \cdot \log n)$ with out-degree 2 in $O(\log\log n)$ time of unbounded degree trees where $S = O(n^\delta)$ and $M \cdot S = O(n^{1+\delta})$.*

Proof. Consider running algorithm P in parallel on all nodes in tree T. Since P does not contain any actions of global nature, no special care is needed. Already after the first iteration it is possible that tree T has shattered into a forest. If this happens, we can still run algorithm P in parallel on all nodes in separate trees in the forest.

We have to verify that Lemmas 4.3, 4.4, 4.6, 4.7 and 4.8 also hold for forests. In the case of Lemmas 4.3, 4.4 and 4.6 it is obvious, as the argumentation is local and thus also applies to forests. Arguments in Lemmas 4.7 and 4.8 consider trees but also account for the possibility of the graph being a forest.

Hence, algorithm P outputs an H-partition with out-degree 2 in $O(\log\log n)$ time of unbounded degree trees by Lemma 4.7. The size of the H-partition is $O(\log\log n \cdot \log n)$, since $i' = O(1)$, $j' = O(1)$ and $k' \cdot 2^{r'}$ lowest layers are removed in each iteration, where $k' = \log\log n$, $r' = \log\log n$.

Now we justify the choice of $B = n^\delta$. Recall that in the low-memory MPC model, $S = n^\epsilon$ for some $0 < \epsilon < 1$. Consider node v and its neighborhood $N(v, k)$ of size

at most n^δ. The maximum number of edges in this neighborhood is $n^\delta - 1$, since we are limited to trees. Storing one edge requires storing two unique identifier of size $O(\log n)$. Since $O(\log n)$ can be bounded by n^δ, storing the whole topology of $N(v, k)$ requires at most $2n^\delta \cdot (n^\delta - 1) \leq n^{3\delta}$ memory. By choosing $\delta = \epsilon/3$, we satisfy the constraint $S = n^\epsilon$. Therefore, we can assume that the local memory constraint of the low-memory MPC model is given directly as the upper bound on the number of edges that fit into the memory of v.

It remains to show that the claimed memory constraints are satisfied. By the description of P, each node (machine) does not add more than n^δ edges into local memory. Since at most n nodes perform graph exponentiation, the total number of edges added in any phase does not exceed $n^{1+\delta}$. Since all nodes left in the graph wipe their memory at the end of each phase, the claim follows. □

4.3 Coloring Algorithm C

Now that algorithm P has outputted an H-partition of size $O(\log \log n \cdot \log n)$ with out-degree 2 of the input tree T, we can proceed with the formal description of the coloring algorithm C. We perform two preprosessing steps.

Firstly, we color all layers $H_1, \ldots, H_l$ simultaneously, each separately, using $O(\Delta^2)$ colors in $O(\log^* n)$ time by directly simulating the LOCAL algorithm of Linial [Lin87]. Recall that Δ denotes the maximum degree of a graph. By Definition 3.1, each node $v \in H_i$ has at most two neighbors in layers $H_j,\ j \geq i$. Therefore, $\Delta \leq 2$ within each layer and simulating Linial's algorithm results in a $O(1)$-coloring within each layer. Since this $O(1)$-coloring is proper, we can perform the following $O(1)$ rounds. In each round, all nodes with the highest color among neighbors in the same layer, recolor themselves with the smallest color such that a proper coloring is preserved in each layer. Clearly, one color is eliminated in each round and since $\Delta \leq 2$ within each layer, we achieve a 3-coloring within all layers.

Secondly, we partially orient T. Each node $v \in H_i$ with a neighbor in $H_j,\ j > i$ directs outwards all incident edges with neighbors in $H_j,\ j \geq i$. In the conflict scenario where adjacent nodes $v, u \in H_i$ both have a neighbor in $H_j,\ j > i$, the higher ID node has the advantage. Note that node v cannot be adjacent to two nodes like u, since this would contradict the fact that v has at least one neighbor in $H_j,\ j > i$. We refer to w as a parent of v if an edge is directed from v towards w, regardless of their layers.

After preprocessing, we compute a 3-coloring of T by iteratively correcting the colorings of certain layers (correcting layers), resulting in proper colorings of subgraphs

$$S_i \coloneqq \bigcup_{j=l-i}^{l} H_j, \quad i = 0, \ldots, l-1.$$

The aim is to arrive at a proper coloring of $S_{l-1} = T$ after having corrected H_1. The base case is clear, since $S_0 = H_l$ already has a proper coloring. Let's assume

that S_{i-1} is properly colored and consider S_i. Subgraph S_i consists of the union of S_{i-1} and H_{l-i} which are both properly colored. The only conflicts between the two colorings can exist along the boundary between S_{i-1} and H_{l-i}. To unify their colorings, it is enough to correct H_{l-i}. Nodes in H_{l-i} will have to recolor themselves if and only if they have at least one parent in S_{i-1}. By Definition 3.1, node $v \in H_{l-i}$ has at most two neighbors in S_i and thus can always recolor itself by choosing the one color that is left over. If node v has to recolor itself, all of its edges in S_i are directed due to the second preprosessing step; v chooses the smallest color that its parents do not have. After one communication round, only nodes v and u from the conflict scenario in the second preprocessing step can be in conflict. After a second communication round, H_{l-i} is corrected and S_i has a proper coloring. Therefore, by the induction principle, when given a proper coloring of S_i, at most $2c$ rounds are needed to compute a proper coloring of S_{i+c}.

The well known result of Theorem 3.5 gives rise to local simulation: when given a proper coloring of S_i, nodes in $S_{i+c} \cap S_i$ need to know the coloring of at most their $2c$-hop neighborhoods to recolor themselves, resulting in a proper coloring of S_{i+c}. As a matter of fact, since only ancestors affect the recoloring of descendants, it is enough for nodes to know the coloring of their *directed* $2c$*-hop neighborhoods*, defined next.

Definition 4.10 (Directed r-hop neighborhood)**.** *For each node v in graph G and for every $r \in \mathbb{N}$, we define the directed r-hop neighborhood $N^d(v, r)$ of v as the regular r-hop neighborhood $N(v, r)$ of v of Section 2 with one additional requirement: the topology of $N^d(v, r)$ must be oriented as if it were a subtree rooted at v, with all edges oriented outwards from v. The topology of $N^d(v, r)$ is the subgraph of G induced by the node set $N^d(v, r)$. The radius of a directed neighborhood $N^d(v, r)$ is r.*

As an example of a directed neighborhood, in Figure 4 we present the directed neighborhood of node v in subgraph S_i after the preprosessing steps. Directed neighborhoods give rise to *directed graph exponentiation*, introduced next; it is similar to regular graph exponentiation of Section 3.3, except that it only considers directed neighborhoods and has a slightly longer runtime.

Lemma 4.11. *With $O(n)$ machines, disregarding local and global memory constraints, all nodes v in graph $G = (V, E)$ can learn $N^d(v, 2^r)$ in at most $2r$ MPC rounds through directed graph exponentiation.*

Proof. Assign a single machine M_v to each node $v \in G$. Let $N^d(v, r)$ denote the directed r-hop neighborhood of v at the beginning of the rth MPC round, when $N^d(v, r)$ is stored in M_v. Since initially the machine assigned to v knows the edges incident to v, it knows $N^d(v, 1)$. For rounds $r = \{1, \ldots, 2r'\}$: in odd rounds, M_v inquires the directed neighborhoods from all nodes $u \in N^d(v, r)$ and in even rounds, all nodes comply. By induction over r, each machine M_v knows $N^d(v, 2^{r'})$ at the end of the $2r'$th MPC round. □

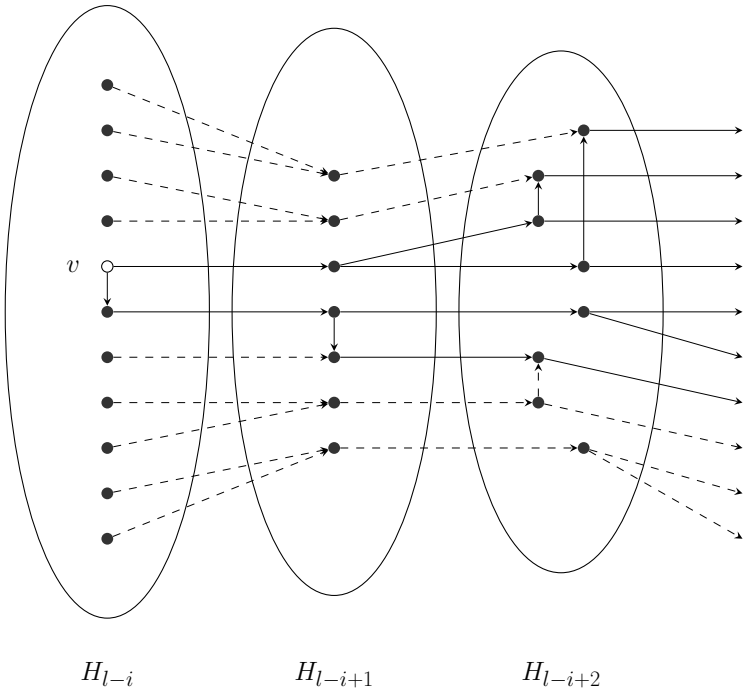

Figure 4: The first three layers of S_i after the preprocessing steps with the directed neighborhood of v emphasized with solid edges.

Algorithm C is given an integer B as an input parameter that describes the memory budget for each node v, i.e., an upper bound on the number of edges that fit into the memory of v. The exact value of B is given later in the analysis together with the reasoning behind it. When all nodes have recolored themselves, the algorithm terminates.

Repetition $r \in \{1, \ldots, r'\}$: Node v performs one step of directed graph exponentiation, resulting in the knowledge of $N^d(v, 2^r)$.

Round $k \in \{1, \ldots, k'\}$: All nodes in $c = 2^{r-1}$ highest layers, of those not yet corrected, recolor themselves.

Directed graph exponentiation. During directed graph exponentiation, the radii of directed neighborhoods increase by a factor of at most 2 per repetition (if local memory does not run out). In particular, disregarding any budget constraints, after r repetitions of directed graph exponentiation, node v knows $N^d(v, 2^r)$ by Lemma 4.11. Note that one repetition takes two communication rounds, since one step of directed graph exponentiation consists of first requesting and then receiving information.

Recoloring. In each round, node v gathers the topology of its learned directed neighborhood $N^d(v, 2^{r'})$ and its coloring in two communication rounds by first requesting and then receiving information. The topology of $N^d(v, 2^{r'})$ always fits

into memory, since $|N^d(v, 2^{r'})| \leq B$ and in trees, the number of edges is bounded by the number of nodes. Recall that given a proper coloring of S_i, nodes v in $S_{i+c} \cap S_i$ need to know the coloring of at most $N^d(v, 2c)$ to recolor themselves, resulting in a proper coloring of S_{i+c}. Since all nodes v in $c = 2^{r'-1}$ highest layers, of those not yet corrected, gather the current coloring of $N^d(v, 2^{r'})$, they can recolor themselves. Since the number of corrected layers is uniquely determined by k, we implicitly assume that nodes keep track of k and know when to recolor themselves.

4.4 Runtime of C

We set $B = n^\delta$ for the same reasons as in Theorem 4.9.

Lemma 4.12. *The runtime of C is $O(\log \log n)$.*

Proof. After $r' = \log \log n^\delta$ repetitions, if node v does not run out of local memory, it knows $N^d(v, \delta \log n)$. Since nodes have at most two parents, $N^d(v, \delta \log n)$ contains at most n^δ nodes. Consequently, nodes do not run out of local memory during any repetition. In each round,

$$c = 2^{r'-1} = \delta/2 \cdot \log n$$

layers get corrected. Since the total number of layers is $O(\log \log n \cdot \log n)$, after $k' = O(\log \log n)$ rounds every node has recolored itself and the claim follows. □

Theorem 4.13. *There is a deterministic low-memory MPC algorithm that, given an H-partition of size $O(\log \log n \cdot \log n)$ with out-degree 2, computes a 3-coloring of unbounded degree trees in $O(\log \log n)$ time where $S = O(n^\delta)$ and $M \cdot S = O(n^{1+\delta})$.*

Proof. Running algorithm C in parallel on all nodes in T yields a 3-coloring in $O(\log \log n)$ time by Lemma 4.12. Since C does not contain any actions of global nature, no special care is needed. Also, since nothing requires the input graph to be a tree, and colorings of trees in a forest cannot be in conflict, C is also compatible with forests. By the description of C, each node (machine) does not add more than n^δ edges into local memory. Since n nodes perform directed graph exponentiation, the total number of edges added during algorithm C does not exceed $n^{1+\delta}$. □

5 Open Questions

In this work, we have presented a state-of-the-art $O(\log \log n)$ time algorithm for 3-coloring unbounded degree trees in the low-memory MPC model. An important intermediate result was the partitioning algorithm P, which constructed an H-partition of unbounded degree trees in $O(\log \log n)$ time. These results prompt the following open questions.

P1 *Can this coloring approach be extended to graphs with arboricity $\lambda > 1$?*

P2 *What other problems can be solved using algorithm P as a subroutine?*

References

[ABB+19] Sepehr Assadi, Mohammad Hossein Bateni, Aaron Bernstein, Vahab Mirrokni, and Cliff Stein. Coresets meet EDCS: Algorithms for matching and vertex cover on massive graphs. In *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'19)*, pages 1616–1635, 2019. https://doi.org/10.1137/1.9781611975482.98.

[ACK19] Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear algorithms for $(\Delta + 1)$ vertex coloring. In *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'19)*, pages 767–786, 2019. https://doi.org/10.1137/1.9781611975482.48.

[BBD+19] 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 *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC'19)*, pages 481–490, 2019. https://doi.org/10.1145/3293611.3331609.

[BE13] Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Morgan & Claypool, 2013. https://doi.org/10.2200/S00520ED1V01Y201307DCT011.

[BFU18] Sebastian Brandt, Manuela Fischer, and Jara Uitto. Matching and MIS for uniformly sparse graphs in the low-memory MPC model. *CoRR*, 2018. arXiv:1807.05374.

[BFU19] Sebastian Brandt, Manuela Fischer, and Jara Uitto. Breaking the linear-memory barrier in MPC: Fast MIS on trees with strongly sublinear memory. In *Structural Information and Communication Complexity (SIROCCO'19)*, pages 124–138, 2019. https://doi.org/10.1007/978-3-030-24922-9_9.

[BHH19] Soheil Behnezhad, Mohammad Taghi Hajiaghayi, and David G. Harris. Exponentially faster massively parallel maximal matching. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS'19)*, pages 1637–1649, 2019. https://doi.org/10.1109/FOCS.2019.00096.

[BKS14] 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'14)*, pages 212–223, 2014. https://doi.org/10.1145/2594538.2594558.

[CLM+18] Artur Czumaj, Jakub Łącki, Aleksander Mądry, Slobodan Mitrović, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC'18)*, pages 471–484, 2018. https://doi.org/10.1145/3188745.3188764.

[CP17] Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS'17)*, pages 156–167, 2017. https://doi.org/10.1109/FOCS.2017.23.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, pages 107–113, 2008. https://doi.org/10.1145/1327452.1327492.

[Die17] Reinhard Diestel. *Graph Theory*. Springer, 2017. https://doi.org/10.1007/978-3-662-53622-3.

[GGJ20] Mohsen Ghaffari, Christoph Grunau, and Ce Jin. Improved MPC algorithms for MIS, matching, and coloring on trees and beyond. In *34th International Symposium on Distributed Computing (DISC'20)*, pages 34:1–34:18, 2020. https://arxiv.org/abs/2002.09610v2.

[GGK+18] 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 2018 ACM Symposium on Principles of Distributed Computing (PODC'18)*, pages 129–138, 2018. https://doi.org/10.1145/3212734.3212743.

[GSZ11] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the MapReduce framework. In *International Symposium on Algorithms and Computation (ISAAC'11)*, pages 374–383, 2011. https://doi.org/10.1007/978-3-642-25591-5_39.

[GU19] Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'19)*, pages 1636–1653, 2019. https://doi.org/10.1137/1.9781611975482.99.

[IBY+07] 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 (EuroSys'07)*, pages 59–72, 2007. https://doi.org/10.1145/1272996.1273005.

[KMW16] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *Journal of the ACM*, pages 17:1–17:44, 2016. https://doi.org/10.1145/2742012.

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the 2010 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'10)*, pages 938–948, 2010. https://doi.org/10.1137/1.9781611973075.76.

[Lin87] Nathan Linial. Distributive graph algorithms – Global solutions from local data. In *28th Annual Symposium on Foundations of Computer Science (FOCS'87)*, pages 331–335, 1987. https://doi.org/10.1109/SFCS.1987.20.

[Lin92] Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, pages 193–201, 1992. https://doi.org/10.1137/0221015.

[LMSV11] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: A method for solving graph problems in MapReduce. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*, pages 85–94, 2011. https://doi.org/10.1145/1989493.1989505.

[Lub86] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, pages 1036–1053, 1986. https://doi.org/10.1137/0215074.

[LW79] Carter J. Lawrence and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, pages 143–154, 1979. https://doi.org/10.1016/0022-0000(79)90044-8.

[LW10] Christoph Lenzen and Roger Wattenhofer. Brief announcement: Exponential speed-up of local algorithms using non-local communication. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'10)*, pages 295–296, 2010. https://doi.org/10.1145/1835698.1835772.

[NLA86] Alon Noga, Babai László, and Itai Alon. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, pages 567–583, 1986. https://doi.org/10.1016/0196-6774(86)90019-2.

[Whi12] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, 2012. https://dl.acm.org/doi/book/10.5555/1717298.

[WL81] Mark N. Wegman and Carter J. Lawrence. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, pages 265–279, 1981. https://doi.org/10.1016/0022-0000(81)90033-7.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael 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 (HotCloud'10)*, page 10, 2010. https://dl.acm.org/doi/10.5555/1863103.1863113.