Distributed MST and Routing in Almost Mixing Time

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

Fabian Kuhn
University of Freiburg, Germany
kuhn@cs.uni-freiburg.de

Hsin-Hao Su
MIT, USA
hsinhao@mit.edu

ABSTRACT

We present a randomized distributed algorithm that computes a minimum spanning tree in $\tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n \log\log n})}$ rounds, in any n-node graph G with mixing time $\tau_{\text{mix}}(G)$. This result provides a sub-polynomial complexity for a wide range of graphs of practical interest, and goes below the celebrated $\tilde{\Omega}(D + \sqrt{n})$ lower bound of Das Sarma et al. [STOC'11] which holds for some worst-case general graphs.

The core novelty in this result is a distributed method for *permutation routing*. In this problem, one is given a number of source-destination pairs, and we should deliver one packet from each source to its destination, all in parallel, in the shortest span of time possible. Our algorithm allows us to route and deliver all these packets in $\tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n \log\log n})}$ rounds, assuming that each node v is the source or destination for at most $d_G(v)$ packets. The main technical ingredient in this routing result is a certain *hierarchical embedding of good-expansion random graphs* on the base graph, which we believe can be of interest well beyond this work.

CCS CONCEPTS

• **Theory of computation → Distributed algorithms**;

KEYWORDS

Distributed Graph Algorithms; CONGEST Model; Routing; Minimum Spanning Tree

1 INTRODUCTION & RELATED WORK

We present a distributed algorithm for computing a *minimum spanning tree* (MST) with a time complexity close to the network's *mixing time*. The core technical novelty in this result is the introduction of a certain hierarchical routing structure, which we believe will pave the road towards achieving a similar time complexity for a wider range of graph problems.

Throughout, we work with the standard CONGEST model of distributed computation: the network is abstracted as an n-node graph $G = (V, E)$. Initially each node knows only its own edges. Per synchronous round, each node can send an $O(\log n)$-bit message to each of its neighbors.

PODC '17, July 25-27, 2017, Washington, DC, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4992-5/17/07…$15.00
https://doi.org/10.1145/3087801.3087827

Background: The MST problem has played a central role in *distributed graph algorithms*, by providing a setup for developing new algorithmic or impossibility techniques, which were then extended to other problems. The work on distributed MST computations started with the seminal algorithm of Gallager, Humblet, and Spira [28], which has time complexity $O(n \log n)$ and which itself is a variant of the 1926 algorithm of Boruvka [58]. The time complexity was gradually improved [15, 27], eventually leading to the "*optimal*" algorithm of Awerbuch, which has time complexity $O(n)$ [7]. A round complexity of $O(n)$ was usually treated as optimal, because there are graphs of diameter $D = \Omega(n)$ on which one cannot do better (e.g., the n-node cycle).

Starting with the pioneering work of Garay, Kutten and Peleg [29, 44], the area took a turn, and moved towards seeking sublinear time algorithms, when this "excuse" of graphs with $D = \Omega(n)$ is ruled out. In particular, an $O(D + n^{0.61})$-round MST algorithm was presented in [29], which was subsequently improved in [44] to $O(D + \sqrt{n} \log^* n)$. This $\tilde{O}(D + \sqrt{n})$ time complexity turned out to be "optimal" for MST, and in fact for an extraordinarily wider range of problems, in the following sense: Peleg and Rubinovich [64] showed that there are graphs of diameter $D = O(\log n)$ on which $\Omega(\sqrt{n}/\log n)$ rounds are necessary for computing an MST. Subsequently, building on a construction of Elkin [23], Das Sarma et al. [17] showed that even approximating MST within any non-trivial factor requires $\Omega(D + \sqrt{n/\log n})$ time on some graphs with $D = O(\log n)$. They also showed that the same lower bound holds for several other fundamental problems, e.g., computing single-source shortest paths or an approximate minimum cut.

Faced with this seemingly omnipresent $\tilde{\Omega}(D + \sqrt{n})$ lower bound, there are two ways ahead. One is to take this lower bound as a last word and be contempt with the current $\tilde{O}(D + \sqrt{n})$ algorithm, because it is the best possible in some graphs. The other is to look for mathematically clean and practically relevant graph properties that allow one to go below the lower bound. This second approach has been pursued in three directions in the prior work:

- **Graphs with Small Constant Diameter:**
  Lotker et al. [51] showed an $O(\log n)$ round algorithm for diameter 2 graphs, and proved that for graphs of diameter 3 and 4, there are lower bounds of $\Omega(\sqrt[3]{n}/\sqrt{\log n})$ and $\Omega(\sqrt[4]{n}/\sqrt{\log n})$, respectively. Later, Lotker et al. [53] showed that in graphs with diameter $D = 1$—i.e., if the graph is a weighted complete graph, which is also known as the *congested clique* model—an MST can be computed in time $O(\log\log n)$. This complexity was improved recently by Hegeman et al. to $O(\log\log\log n)$ [36], and subsequently, by Ghaffari and Parter to $O(\log^* n)$ [33].

- **Graphs with Small MST Radius or Local Shortest-Path Diameter (SPD):**

Elkin [22] introduced the notion of MST radius, which roughly speaking is maximum radius each node needs to check to check whether any of its edges is in the MST. This is denoted by $\lambda(G, w)$ and it depends on both the topology and the weights. Elkin provided an algorithm with complexity $\tilde{O}(\lambda(G, w) + \sqrt{n})$. Khan and Pandurangan [41] introduced the notion of *local shortest path diameter* $L(G, w)$, which roughly speaking is the maximum number of hops of the minimum-weight path between each node and the other endpoint of its heaviest edge. Note that $L(G, w)$ dependends both on the topology and on the weights. The paper presented an $O(\log n)$ approximation algorithm for MST with time complexity $\tilde{O}(D + L(G, w))$.

- **Planar and Near-Planar Graphs:**

  Ghaffari and Haeupler [31] developed the general concept of *low-congestion shortcuts*, and applied it to obtain an MST algorithm with time complexity $\tilde{O}(D)$ in planar graphs. This was then extended by Haeupler, Izumi, and Zuzic [34, 35] to graphs with bounded genus or bounded tree-width.

Our Contribution: In this paper, we focus on graphs with good *expansion* properties. In particular, we analyze the round complexity of computing an MST and related problems as a function of the random walk mixing time $\tau_{\text{mix}}(G)$ of the network graph G. For a formal definition of $\tau_{\text{mix}}(G)$, we refer to Section 2. Our main result proves that an MST can be computed in almost the mixing time.

Theorem 1.1. *There is a distributed, randomized Las Vegas algorithm that computes an MST in time $\tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n \log\log n})}$ in the* CONGEST *model, with high probability*[1].

We remark that while the $2^{O(\sqrt{\log n \log\log n})}$ term is more than any polylogarithmic function of n, it is a slowly growing function of n and in particular substantially smaller than any fixed power of n. Theorem 1.1 in particular implies that in graphs with mixing time at most $2^{O(\sqrt{\log n \log\log n})}$ or conductance at lest $2^{-O(\sqrt{\log n \log\log n})}$, an MST can be computed in time $2^{O(\sqrt{\log n \log\log n})}$. This includes all graphs with maximum degree $2^{O(\sqrt{\log n \log\log n})}$ and edge or vertex expansion at least $2^{-O(\sqrt{\log n \log\log n})}$. Note that a wide range of the common (overlay) networks used in practical distributed applications fall in this category, see e.g. [4, 8, 45, 55, 60–62, 69].

By combining the above algorithm with techniques developed in [32, 57] and especially a method of [31], the same upper bound as in Theorem 1.1 can also be achieved for computing a $(1 + \varepsilon)$-approximate minimum cut.

At its heart, our MST algorithm is based on a new random-walk based distributed routing scheme to serve a large number of concurrent pair-wise routing requests. This routing scheme is then incorporated into an MST algorithm in the style of Boruvka's classic approach [58], with the help of a few additional algorithmic ingredients, to obtain the MST algorithm of Theorem 1.1. The routing scheme is the main technical contribution of the present paper.

Theorem 1.2. *Consider a graph $G = (V, E)$ and a set of point-to-point routing requests, each given by the IDs of the corresponding source-destination pair. If each node of G is the source and the destination of at most $d_G(v) \cdot 2^{O(\sqrt{\log n \log\log n})}$ messages, there is a randomized distributed algorithm that delivers all messages in time $\tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n \log\log n})}$, w.h.p.*

[1]As standard, *with high probability* (w.h.p.) indicates that an event has probability $1 - 1/n^c$ for a constant $c \geq 2$.

Our Routing Method in a Nutshell: A random walk starting from the packet source would be unlikely to get to the correct destination, unless it is very long, or we use a very large number of such walks. In our routing algorithm, we use random walks to construct a *hierarchical structure*, which roughly speaking recursively embeds good-expansion random graphs on the base graph. For the construction, we replace every node $v \in V$ by $d_G(v)$ virtual nodes so that overall, we have $2|E|$ virtual nodes. On the bottom level, we use parallel random walks of length $\tau_{\text{mix}}(G)$ to compute a low-congestion embedding of an approximate Erdős-Rényi random graph with average degree $O(\log n)$ among the virtual nodes. Because in the stationary distribution of a random walk on G, the probability of a node v is proportional to its degree, we can run $O(\log n)$ random walks per virtual node with only a logarithmic slowdown compared to running a single random walk in the graph. On the second level, we partition the set of virtual nodes into $\alpha = 2^{O(\sqrt{\log n \log\log n})}$ parts and for each part, we compute a low-congestion embedding of another approximate Erdős-Rényi random graph on top of the bottom random graph. The time for computing such an embedding is $\alpha \operatorname{poly}\log n$. For each of the α parts, we proceed recursively, we divide the part into α smaller parts and embed a new random graph in each part. The construction has $O\Big(\sqrt{\frac{\log n}{\log\log n}}\Big)$ recursion levels and can be constructed in time $\alpha \operatorname{poly}\log n = 2^{O(\sqrt{\log n \log\log n})}$. As the random graph embedding on each recursion level adds a polylogarithmic stretch factor, the cost for routing a message is also $2^{O(\sqrt{\log n \log\log n})}$. The random graph embeddings guarantee that in parallel we can cope with up to $2^{O(\sqrt{\log n \log\log n})}$ messages per virtual node. By using a small number of shared random bits, which can be disseminated in $O(D \operatorname{poly}\log n)$ time, the position of a given node ID in the hierarchical partition can be determined by a globally known random hash function.

A More Efficient Solution For Denser Routing Requests: We also present more efficient routing algorithms for instances where the set of routing requests is larger. Due to the space limitations, the explanations of this part are deferred to the full version of the paper. Here, we simply overview the end-result:

We focus on routing instances where each node has to send one $O(\log n)$-bit message to each other node. Such a routing algorithm can be seen as emulating the *congested clique model* [53] on top of a general underlying graph. The congested clique model has recently received considerable attention [10, 11, 14, 20, 21, 30, 33, 36–39, 42, 50, 56, 63] and for general graphs, admitting a fast algorithm for the above *clique emulation problem* is considered as one of the desirable network 'axioms' in [5]. A simple cut-based argument shows that the clique emulation problem requires time at least $\Omega(n/h(G))$, where $h(G)$ is the edge expansion of G. We prove the following upper bound.

Theorem 1.3. *There is a distributed algorithm that emulates the clique in a graph G, w.h.p., in $O\Big(\frac{n}{h(G)} \cdot \Big(1 + \frac{\Delta}{n} \cdot \frac{\Delta}{h(G)} \cdot \log n\Big) \cdot \log n \log^* n\Big)$ rounds, where Δ denotes the maximum degree of G. Moreover, if $h(G) = \Omega(\Delta)$ and $\Delta \geq n^{0.5+\varepsilon}$ for constant $\varepsilon > 0$, the bound can be improved to $O\Big(\frac{n}{h(G)} \cdot \log n \cdot \log^* n\Big)$.*

Remark: We note that as a simple corollary, this theorem yields an improvement of a recent result of Balliu et al. [9] about the time needed for emulating the clique in Erdős-Rényi [24] random graphs. From Theorem 1.3, we get that an Erdős-Rényi random graph $G_{n,p}$ above the connectivity threshold — i.e., where $p = \Omega(\frac{\log n}{n})$ — can emulate the clique in $O(1/p + \log n)$ rounds. This is because, by a simple calculation, we can see that in such graphs, with high probability, $h(G) = \Theta(np)$ and also $\Delta = \Theta(np)$. This $O(1/p + \log n)$-round emulation improves over the result of Balliu et al.[9], which emulates the clique in $O(\min\{1/p^2, np\})$ rounds assuming $p \geq \sqrt{\log n/n}$. Notice that $\Omega(1/p)$ is a clear lower bound as each node has degree $\Theta(np)$ and it should receive $n-1$ messages. Hence, this $O(1/p + \log n)$-round bound is nearly optimal.

Other Related Work—the Centralized or Existential Analogues

We conclude the introduction by mentioning some other related work which concern the existential aspects of the routing problem that we consider, or the centralized algorithms for it. In the setting that we focus on in this paper, initially, the communication network G is known only locally, i.e., each node knows only its neighbors. This is important for most of the plausible applications of our work. For instance, in overlay networks, the topology is usually dynamic and changes with time, and it is never fully known to one node, see e.g. [4, 43, 45, 60–62]. The variant of the routing problem where G is globally known relates to some well-studied problems in the area of centralized computation, which we briefly review next.

Multi-Commodity Flow with Short Paths: If G is known to a centralized computer, the seminal work of Leighton and Rao [48] can be used to route the packets: we formulate the problem as a *multi-commodity flow* instance with a unit flow demand per node pair, then do a randomized rounding on the flows to generate short paths, and finally schedules the packet routing of these paths using Lovasz Local Lemma. However, this method seems unlikely to lead to a fast distributed algorithm, the multi-commodity flow part itself needs a large polynomial centralized time complexity.

Edge-Disjoint Paths in Expanders: A problem that relates to the routing question is that of finding edge-disjoint paths in expanders, between given source-destination pairs. This problem began with the work of Peleg and Upfal[65], who showed a polynomial time algorithm for finding n^{ε} paths, for a small constant ε. This problem has since receive extensive attention[12, 13, 25, 46, 47, 49]. This line of work was improved to optimality by [26], which finds $\Theta(\frac{n}{\log n})$ paths in a (large) polynomial time in regular-expanders with sufficiently large expansion. It does not seem likely to us that these computationally intensive centralized methods would yield a fast distributed routing algorithm.

2 PRELIMINARIES

Mathematical notations: Throughout the paper, we use n to denote the number of nodes and m to denote the number of edges of the network graph $G = (V, E)$. Further, we use $d_G(v)$ to denote the degree of some node $v \in V$, and Δ for the maximum degree of G. Moreover, given a nontrivial partition of V to S and $V \setminus S$, we use $e(S, V \setminus S)$ to denote the number of edges connecting S and $V \setminus S$. The *edge expansion* $h(G)$ of a graph $G = (V, E)$ is defined as $h(G) := \min_{S \subseteq V, 1 \leq |S| \leq n/2} \frac{e(S, V \setminus S)}{|S|}$.

Random Walks: Our algorithms use random walks as a main tool. Note that random walks are by now quite a standard tool, used extensively throughout all branches of theory of computation, see, e.g.[1, 2, 6, 16, 18, 19, 54, 59, 67, 68, 70]. In order to guarantee that the resulting Markov chain is aperiodic, throughout the paper when referring to the random walk on a graph G, we implicitly assume the usual so-called lazy random walk: In every step, the walk remains at the current node with probability $1/2$ and it transitions to a uniformly random neighbor otherwise. In case of multi-graphs, the transision is on a uniformly chosen random edge. The *stationary distribution* of such a random walk is proportional to the degree distribution, i.e., when performing enough steps of the random walk, the probability for ending in a node v converges to $d_G(v)/2m$. We formally define the *mixing time* of the random walk as follows.

Definition 2.1. [Mixing Time] For $V = \{v_1, \dots, v_n\}$ and a node $v \in V$, let $P_v^t = \big(P_v^t(v_1), \dots, P_v^t(v_n)\big)$ be the probability distribution on the nodes after t steps of a lazy random walk starting at v. We define the mixing time $\tau_{\text{mix}}(G)$ as the minimum t such that for all $v, u \in V$, we have $\left|P_v^t(u) - \frac{d_G(v)}{2m}\right| \leq \frac{d_G(v)}{2mn}$. When the graph G is clear from the context, we sometimes use τ_{mix} instead of $\tau_{\text{mix}}(G)$.

Remark. By running the walk for $O(\tau_{\text{mix}})$ steps, one can improve the deviation from the mixing time to $\left|P_v^{\tau_{\text{mix}}}(u) - \frac{d_G(v)}{2m}\right| \leq 1/n^c$ for an arbitrary constant $c > 0$.

We sometimes need to perform random walks that have a uniform stationary distribution. By using standard ideas—see e.g., [6, 40]—we can "regularize" the graph to achieve this.

Definition 2.2 (2Δ-Regular Random Walk). Let $G = (V, E)$ be an n-node graph with maximum degree Δ and let $G' = (V, E')$ be the multi-graph that is obtained from G by adding $\Delta - d_G(v)$ self-loops to each node v. A 2Δ-*regular random walk* on G is defined as the usual lazy random walk on G'. We use $\overline{\tau}_{\text{mix}}(G) = \tau_{\text{mix}}(G')$ for the mixing time of the 2Δ-regular random walk on a graph G.

Hence, when running a 2Δ-regular random walk on G, in each step, the walk remains at the current node with probability $1 - d(v)/2\Delta$ and otherwise, it transitions to each of the $d_G(v)$ neighbors with probability $1/\Delta$. In previous work, similar regularized random walks have also been called Δ-lazy random walks [6] or max-degree random walks [40]. The following lemma uses Cheeger's theorem (see e.g. [54]) to upper bound the mixing time $\overline{\tau}_{\text{mix}}$ of a 2Δ-regular random on G walk as a function of the edge expansion $h(G)$ of G.

Lemma 2.3. *The mixing time $\overline{\tau}_{\text{mix}}$ of a 2Δ-regular random walk on a graph G is $\overline{\tau}_{\text{mix}} \leq 8 \cdot \frac{\Delta^2}{h^2(G)} \cdot \ln n$.*

Proof of Lemma 2.3. The convergence of a random walk can expressed as a function the spectral properties of the stochastic random walk matrix and the most important connection to graph expansion is captured by Cheeger's inequality. For more details, we refer to [54], the following analysis is also based on [67]. The congergence of random walks to the stationary distribution can be upper bounded in the following way. For a node $v \in V$ and $V = \{v_1, \ldots, v_n\}$, let $P_v^t = \left(P_v^t(v_1), \ldots, P_v^t(v_n)\right)$ be the probability distribution on the nodes after t steps of a *lazy random walk* on a graph G starting at a node v of G. In a lazy random walk, in each step, with probability $1/2$, the walk stays at the same node and with probability $1/2$ it moves on a uniformly random edge incident to the current node. Let $\pi = (\pi(w_1), \ldots, \pi(w_n))$ denote the stationary distribution. For lazy random walks in a regular graph G, the stationary distribution is the uniform distribution and we have: $\left|P_v^t(w) - \pi(w)\right| \leq \left(1 - \frac{\phi^2(G)}{4}\right)^t$. Here, $\phi(G)$ denotes the *conductance* of a graph $G = (V, E)$, which is defined as follows:

$$\phi(G) := \min_{S \subseteq V : \mathrm{vol}(S) \leq m} \frac{e(S, V \setminus S)}{\mathrm{vol}(S)},$$

where for each $X \subseteq V$, we have $\mathrm{vol}(X) := \sum_{v \in X} d_G(v)$.

Note that the 2Δ-regular random walk can be seen as a lazy random walk on the Δ-regular multi-graph $G' = (V, E')$ which is obtained from G in the following way. To each node v of degree $d_G(v)$, we add $\Delta - d_G(v)$ self loops. Taking a uniformly random edge with probability $1/2$ now implies that each edge of G incident to the current node is picked independently with probability $1/2\Delta$ and thus the lazy random walk on G' is equivalent to the 2Δ-regular random walk on G. For the auxiliary multi-graph G', we have $\phi(G') = h(G)/\Delta$. Therefore, after $t = \overline{\tau}_{\mathrm{mix}}(G) = 8\Delta^2 \ln n / h^2(G) = 8 \ln n / \phi^2(G)$ steps, we have

$$\left|P_v^t(w) - \frac{1}{n}\right| \leq \left(1 - \frac{\phi^2(G')}{4}\right)^{8 \ln n / \phi^2(G')} \leq e^{-2 \ln n} = \frac{1}{n^2}.$$

□

Parallel Random Walks: In all of our results, we run many random walks in parallel. We next discuss how we do this. We first prove a simple helper lemma. We comment that variants of these simple helper lemmas appear in prior work, see e.g. [18, 19].

Lemma 2.4. *Let $G = (V, E)$ be a n-graph and assume that we (synchronously) run at most $n^{O(1)}$ steps of a collection of independent, parallel random walks on G. If each node is starting node v of at most $kd_G(v)$ random walks, w.h.p., after each parallel step, there are at most $O(kd_G(v) + \log n)$ random walks at each node $v \in V$.*

Proof of Lemma 2.4. The stationary distribution of a random walk on a graph G is proportional to the degree distribution. Hence, if we start exactly $kd_G(v)$ random walks at every node v of G, after every step, the expected number of random walks at every node v is $kd_G(v)$. If at most $kd_G(v)$ random walks start at every node, at each time, the expected number of random walks at every node v is upper bounded by $kd_G(v)$. Because the walks are run independently, the lemma follows from a standard Chernoff bound. □

When performing several random walks in parallel in a distributed way, we assume that in each round each node can send up to one random walk over each of its edges. If running several random walks in parallel such that each node is the starting node v of $kd_G(v)$ random walks, in each parallel random walk step, each node has to forward around k random walks over each of its edges. Simulating T steps of all the random walks will therefore require at least around kT rounds. The following lemma shows that even if all the random walks are run in an independent way, we can run them distributedly in a way that nearly matches this lower bound.

Lemma 2.5. *Let $G = (V, E)$ be an n-node graph and let $k \geq 1$ be a positive integer. Assume that we want to perform $T = n^{O(1)}$ steps of a collection of independent random walks in parallel. If each node $v \in V$ is the starting node v of at most $kd_G(v)$ random walks, w.h.p., the T steps of all the random walks can be performed in $O\Big((k + \log n) \cdot T\Big)$ rounds in a distributed way.*

Proof of Lemma 2.5. We perform all the walks synchronously, and we show that we can perform one step of all walks in parallel, in one phase, where each phase has $c(k+\log n)$ rounds for a sufficiently large constant $c > 0$. From Lemma 2.4 and a union bound over all the T steps, we know that in each step, the number of walks in each node v is at most $O(kd_G(v) + \log n)$, w.h.p. Each of these walks chooses each edge e incident on v with probability $1/2d_G(v)$. Hence, the number of walks that need to go through e in one phase is upper bounded by $O(k + \log n)$ in expectation and by a standard Chernoff bound also w.h.p. Therefore, if choosing the constant c sufficiently large, the number of walks that need to go through e is less than the length of one phase, which means that the phase suffices for performing one step of all walks in parallel. □

The bound $O\Big((k + \log n) \cdot T\Big)$ in Lemma 2.5 is asymptotically optimal, matching the kT lower bound, when $k = \Omega(\log n)$. However, when $k = o(\log n)$, the additive $\log n$ term becomes dominant and makes the bound sub-optimal. To overcome this issue, for some range of parameters, we will run the random walks in a carefully correlated fashion. The related details are deferred to the full version of the paper.

3 DISTRIBUTED PERMUTATION ROUTING IN ALMOST MIXING TIME

In this section, we explain our distributed algorithm for *permutation routing* and slightly more general routing instances. This algorithm constructs a certain *hierarchical routing structure*, which allows us to solve the following packet routing problem: given several packet routing requests, each specified by a *packet source* and the corresponding *packet destination*, and such that each node v is the source or destination for at most $d_G(v)$ packets, we can route all these packets from their sources to their destinations in $\tau_{\mathrm{mix}} \cdot 2^{O(\sqrt{\log n \log \log n})}$ rounds.

The construction of this *hierarchical routing structure* and the general method for using it are explained in Section 3.1 and Section 3.2, respectively. In Section 4, we show the applications of this routing, by explaining a distributed algorithm for computing a minimum spanning tree in $\tau_{\mathrm{mix}} \cdot 2^{O(\sqrt{\log n \log \log n})}$ rounds.

3.1 The Routing Structure via Hierarchical Embedding of Random Graphs

Here, we explain the construction of our hierarchical routing structure. The two key aspects of this structure are: (1) it allows us to route many messages on this structure simultaneously and fast, (2) we can construct this structure fast.

3.1.1 Embedding a Sparse Random Graph $\mathbf{G_0}$ *on* G. As a preparation step, which can be viewed the level zero of our hierarchical construction, we embed an Erdős-Rényi random graph G_0 on top of our base graph. After that, our construction will continue atop G_0. Working with the random graph G_0 simplifies some of our construction and analysis, in comparison to directly dealing with G, while as we will see, this transition has only a poly $\log n$ factor overhead in the round complexities.

Let $G = (V, E)$ be the base graph, and let $n = |V|$ and $m = |E|$. As we assume G to be connected, we have $\log m = \Theta(\log n)$, and thus, we express all logarithmic terms as a function of $\log n$.

The Level-Zero Random Graph $\mathbf{G}_0$: Graph G_0 has $2m$ nodes, where each node $v \in G$ simulates $d_G(v)$ many G_0-nodes. The construction of the random graph G_0 atop G will be such that each communication round of G_0 can be implemented in poly($\log n$) rounds of G. The graph G_0 will be an Erdős-Rényi $G_0 = G(m, p)$ random graph, where $p = \frac{200 \log n}{m}$. To construct the overlay G_0, we select $mp = 100 \log n$ independent random outgoing neighbors for each node $v_0 \in G_0$, with a uniform distribution for each neighbor[2]. Then, we forget the direction of the edges.

The process of identifying the edges in G_0 is as follows. We start $200 \log n$ many independent random walks from each G_0-node. This is a total of $200 d_G(v) \log n$ random walks from each node $v \in G$. We run all these walks in parallel in G, each for $\tau_{\text{mix}}(G)$ steps. This can be done in $O(\tau_{\text{mix}} \text{ poly} \log n)$ rounds, by Lemma 2.5. At the end, the tokens of each G_0-node v_0 are distributed (essentially) uniformly among the nodes of G_0. Thus, the tokens of node $v_0 \in G_0$ are in at least $100 \log n$ different G_0-nodes. These are the outgoing neighbors of v_0 in overlay G_0.

We then run the random walks in the reverse direction of their forward traversal, thus getting them back to their sources. Running the walks in reverse is possible because during the forward traversal, each node can remember in which direction it forwarded each random walk token. At the end, each G_0-node v_0 knows at least $100 \log n$ randomly selected other G_0-nodes. Node v_0 selects exactly $100 \log n$ of these, say at random, and considers them as its own out-neighbors in the graph G_0. Thus, each node v_0 knows its outgoing edges. By running the walks one more time in the forward direction, we can make all nodes $u_0 \in G_0$ also know their incoming edges. At this point, we forget the directions of the edges. These are the edges of G_0. Hence, we now have the random graph G_0 computed.

We can emulate a single communication round of G_0 in $\tau_{\text{mix}} \cdot$ poly($\log n$) rounds of G, by re-running the exact random walks as above. That is, we can deliver one message from each G_0-node to each of its G_0-neighbors, using $\tau_{\text{mix}} \cdot$ poly($\log n$) rounds of G.

[2]Note that the Erdős-Rényi model is usually described as connecting each node pair with probability p. For the range of p in our discussion, this graph distribution is equivalent, modulo a negligible inverse-polynomially small probability, to the graph distribution resulting from randomly and uniformly picking $mp/2$ outgoing neighbors for each node.

3.1.2 The Hierarchical Construction. We now work on top of G_0. From here on, the construction of our hierarchical embedding is recursive. We start by explaining the first level; the other levels are repetitions of same method. Throughout, we have one main parameter β, and as we will see, setting $\beta = 2^{O(\sqrt{\log n \log \log n})}$ gives the best trade-off.

First Level of the Recursion: Partition the nodes of G_0 into β disjoint sets $A_1, A_2, \ldots, A_\beta$ such that for each i, we have $|A_i| = \Theta(\frac{|G_0|}{\beta}) = \Theta(\frac{m}{\beta})$. We later discuss how this partitioning is done. For now, it is convenient to think that each G_0-node is put in one of these β parts at random.

We generate a virtual random graph G_1 on the same vertex set as G_0. The graph G_1 is a disjoint union of β random graphs, one for each A_i. That is, the graph G_1 has β disconnected components, one for each set A_i. The connected component induced by the set A_i is defined by having each node of A_i pick $O(\log n)$ uniformly random neighbors from the same set A_i. The construction of G_1 will be such that one communication round of the graph G_1 can be emulated in $O(\log^2 n)$ communication rounds of graph G_0. In short terms, each round of G_1 embeds into $O(\log^2 n)$ rounds of G_0. After the construction, we will be able to think of G_1 as a collection of β independent random graphs, one for each A_i, such that these β random graphs can work effectively in parallel, modulo a multiplicative $O(\log^2 n)$ round complexity overhead.

To construct G_1, we start $O(\beta \log n)$ many 2Δ-regular random walks from each node $v \in G_0$, defined as in Definition 2.2. Each of these walk tokens carries the ID of node v and also the number $i \in \{1, 2, \ldots, \beta\}$ for which $v \in A_i$. We run all the $O(m\beta \log n)$ random walks for $\tau_{\text{mix}}(G_0) = O(\log n)$ steps. As Lemma 2.5 shows, this can be performed in $O(\beta \log^2 n)$ rounds of G_0. We consider a random walk token *successful* if its starting point and end point are in the same set A_i. At the end, for each node v, there are $O(\beta \log n)$ tokens of $v \in A_i$ spread uniformly over the graph G_0. Since $|A_i| = \tilde{\Theta}(\frac{m}{\beta})$, w.h.p., $\Theta(\log n)$ of these random walk tokens of v are successful, meaning that they ended in a node of A_i. Hence, v has $\Theta(\log n)$ successful tokens, each of which gives v one randomly sampled neighbor in A_i. We then run these random walks backwards, thus letting node v know its G_1-neighbors. These successful walks, which are $O(\log n)$ many per node, can be re-run in $O(\log^2 n)$ rounds of G_0.

Lemma 3.1. *Each round of* G_1 *can be emulated in* $O(\log^2 n)$ *rounds of* G_0.

Proof of Lemma 3.1. The argument is essentially by invoking Lemma 2.5. Some care is needed because conditioning on the each random walk having a fixed source and destination can bias the distribution of the walks and conceivably increase the number of walks that need to go through one edge. However, this is not a real problem in our case. We can assume that the walks are run for a length more than a 2 factor of the mixing time. Then, given the source and destination, the walk can be viewed as going from the source to a random middle point, and then going from there to the destination, and each of these two parts clearly induces a distribution of tokens on edges similar to basic random walks, thus allowing us to repeat the arguments of Lemma 2.5. □

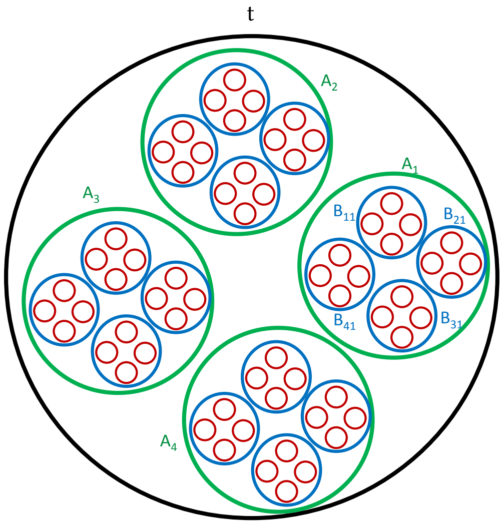

Figure 1: **An illustration of three levels of the hierarchical subsets. We have one random graph on each ball and random graphs of balls of each level can be implemented in $O(\log^2 n)$ rounds of one of the balls of the lower layer. Thus, for instance, we can run one round of graphs of B_{11}, B_{21}, B_{31}, and B_{41} all in $O(\log^2 n)$ rounds of the graph of A_1.**

To conclude, using $O(\beta \log^2 n)$ rounds of G_0, we constructed the graph G_1, made of one random graph per set A_i. Moreover, each round of G_1 can be emulated in $O(\log^2 n)$ rounds of G_0. We are now ready to proceed to the next levels of our hierarchical embedding.

Next Levels of the Recursion: The next levels of the recursive hierarchical construction are similar. See Figure 1 for a pictorial illustration. In the second level, we partition each set A_i into sets $B_{1i}, B_{2i}, B_{3i}, \dots, B_{\beta i}$, of (almost) the same size. The exact partition will be discussed later. We then construct a random graph G_2 where each node of B_{ji} is matched to $O(\log n)$ randomly chosen neighbors of B_{ji}, by running random walks as before, this time on G_1. This construction of G_2 takes $O(\beta \log^2 n)$ rounds of G_1, and thus $O(\beta \log^4 n)$ rounds of G_0. Moreover, the each round of the graph G_2 can be emulated in $O(\log^2 n)$ rounds of G_1.

Continuing k levels of this recursive construction will produce graphs $G_0, G_1, \dots, G_k$, where each graph is a structured random graph on some subsets, where each of these subsets is a refinement of those of the previous level. Since in each iteration, the size of each subset goes down by about a β factor, we repeat this procedure for at most $k = O(\log_\beta m)$ iterations, until each subset has size at most $O(\log n)$. At that point, we just take the complete graph on each of those subsets as their "random" graph. The following lemma summarizes the complexity of this construction.

Lemma 3.2. *We can perform $k = \log n / \log \beta$ levels of this construction in $O(k\beta)(\log n)^{O(k)} = 2^{O(\sqrt{\log n \log\log n})}$ rounds of G_0. Moreover, each round of graph G_i can be emulated in at most $(\log n)^{O(i)}$ rounds of graph G_0. This is at most $2^{O(\sqrt{\log n \log\log n})}$ rounds, even for the last level $k = \log n / \log \beta$.*

There are two aspects of the construction of our hierarchical embedding which remain to be discussed: (A) One aspect is how we partition the nodes, in different levels of recursion. In particular, this partition should be into parts of almost equal size, but also such that the source of each packet can know the part to which the corresponding destination belongs. (B) The other aspects is some *portals* that we add to the construction, which we use for moving the packets between different parts of the partition. We next explain these two aspects.

Pseudo-Random Partitions, and the Hierarchical Labeling: As we described above, we have k iterative levels of partitioning, and as a result, each node $v \in G_0$ has a sequence of length $k = \log n / \log \beta$ of partition labels $(\ell_1, \ell_2, \dots, \ell_k)$. Here, each partition label ℓ_p is a number in $\{1, 2, \dots, \beta\}$, that indicates the subset that node v chose in level p. Particularly, $\ell_1 = i$ means that $v \in A_i$; then $\ell_2 = j$ means that $v \in B_{ji}$, etc. It is intuitive to think of this partitioning as a β-ary tree with depth $k = \log_\beta \frac{m}{\log m}$. Then, the p^{th} label ℓ_p indicates that in the p^{th}, node v chose the branch of the parititon tree corresponding to the child number ℓ_p.

We would like to have two properties about this partitioning: (P1) We want the partitioning to be nearly-uniform in all levels. More concretely, we want that for each label prefix $(\ell_1, \ell_2, \dots, \ell_p)$ for $p \in [1, k]$, the number of nodes whose label starts with this prefix should be $\Theta(\frac{m}{\beta^p})$. In other words, at each level of the partitioning tree, the size of the subtrees rooted in the nodes of that level should be almost equal. (P2) We would like that each node v can know the full label of each other node u.

Random partitioning gives (P1) but not (P2). More concretely, if we make each node v put itself on a random one of the leaves of the β-ary tree of depth $k = \log_\beta \frac{m}{\log m}$, then with high probability, all the leaves would have $\Theta(\log m)$ nodes. Therefore, we would have the near-uniformity property of (P1). However, with such a fully random assignment, we will not have property (P2), meaning that a node v would not know the partition label of another node u.

To overcome this problem and satisfy property (P2), instead of using fully random assignments, we use a pseudo-random partitioning. In particular, we use $\Theta(\log n)$-wise independent hash functions that map the set of IDs to the leaves of the β-ary tree of depth $k = \log_\beta \frac{m}{\log m}$. Here, W-wise independence for $W = \Theta(\log n)$ means that for any set of W IDs, the probability that they are all mapped to one leaf is the same as in the fully random mapping, i.e., equal to $\frac{1}{(\beta^k)^W}$. It is known that to construct such a hash function, $\Theta(W \log n) = \Theta(\log n)$ bits of randomness suffice[3]. That is, one can use these $\Theta(\log^2 n)$ bits to pick a random hash function among $2^{\Theta(\log^2 n)}$ deterministic hash functions, in a way that ensures the desired $\Theta(\log n)$-wise independence. We make the leader of the network node pick $\Theta(\log^2 n)$ random bits, and then we deliver these bits to all nodes. Note that this can be done easily in $O(D \log n) = O(\tau_{\text{mix}} \log n)$ rounds of G. That allows nodes to compute these hash functions.

On the other hand, one can use extensions of the Chernoff bound to $\Theta(\log n)$-wise independent random variables[66] to conclude that when mapping m nodes of G_0 using these $\Theta(\log n)$-wise independent hash functions, each leaf gets $\Theta(\log m)$ nodes, w.h.p. Hence, we have the near-uniformity described in property (P1).

Adding Portals to the Hierarchical Construction: During our routing, some of the packets will be kept and routed within the same branch of the partition, whereas others need to be hopped to the other branch. For instance, a packet residing in a node $s \in A_i$ may be destined for a node $t \in A_j \neq A_i$. In this case, we need to first deliver the packet to a node $t' \in A_i$ such that t' has a G_0-neighbor s' in the set A_j, and then to hop the packet from t' to s', and finally route it from s' to s, recursively. In this case, we refer to t' as the *portal of node* $s \in A_i$ *towards the set* A_j. We now describe how we augment our hierarchical routing structure with these portals.

We describe the process for the first level of the hierarchical construction. A similar procedure is used for the other levels. For the first level, we would like that for each $i, j \in \{1, 2, \ldots, \beta\}$, each node $s \in A_i$ knows the label of a node $t' \in A_i$, such that t' has a G_0-neighbor s' in the set A_j. In this case, node t' will be known as the *portal of node* $s \in A_i$ *to the set* A_j. Furthermore, we want an additional *uniformity property* for these portals, as follows. We would like that the portal of each node $s \in A_i$ towards set A_j should be chosen uniformly at random among nodes of A_i that have G_0-neighbors in A_j. The randomness in the choice of the portals t' for different nodes $s \in A_i$ will be independent. The next lemma states that we can add these portals to our hierarchical embedding, within the promised round complexity.

Lemma 3.3. *We can add the random portals to the hierarchical structure in* $2^{O(\sqrt{\log n \log\log n})}$ *rounds.*

Proof of Lemma 3.3. This part of the construction will be in $\tilde{O}(\beta^2)$ rounds of G_0, where we spend $\tilde{\Theta}(\beta)$ rounds to identify the portals towards set A_j, for each $j \in \{1, 2, \ldots, \beta\}$. Fix one A_j, we desribe how various sets A_i can find their portals towards A_j in $\Theta(\beta)$ rounds.

We start β random walks from each node $s \in A_i$ and we run these random walks on $G_1[A_i]$ for $O(\log n)$ steps. We do this simultaneously for all A_i. This will take $O(\beta \log n)$ rounds of G_1, which means $O(\beta \log^3 n)$ rounds of G_0. At the end, random walks that are sitting on nodes that have a connection to A_j are considered *successful.* Notice that since G_0 is a $G(m, p)$ random graph, there are $\Theta(m/\beta)^2 p = \Theta(m \log n/\beta^2)$ nodes of A_i that have neighbors in A_j, that is, a $O(\log n/\beta)$ fraction of nodes of A_i. Hence, among the β random walks of s, we expect $O(\log n)$ to be successful. That means, with high probability, $\Theta(\log n)$ of them are successful. We then run these successful random walks backwards towards their sources, thus letting each node s know the label of a random portal towards A_j. This process takes $O(\beta \log n)$ rounds of G_1, and determines the portals towards one fixed A_j. Hence, performing this for all j takes $O(\beta^2 \log n)$ rounds of G_1.

We will perform a similar construction for each level of the hierarchical structure. In level k, this portal construction takes $O(\beta^2 \log n)$ rounds of the graph G_k. Since $\beta = 2^{O(\sqrt{\log n \log\log n})}$ and as each round of G_k is embedded in at most $2^{O(\sqrt{\log n \log\log n})}$ rounds of G_0, per level this is at most $2^{O(\sqrt{\log n \log\log n})}$ rounds of graph G_0. Moreover, even summed up over all the $k = O(\log n/\log \beta)$ levels, this is still $O(\log n/\log \beta) \cdot 2^{O(\sqrt{\log n \log\log n})} = 2^{O(\sqrt{\log n \log\log n})}$ rounds, for learning all the portals. □

3.2 Routing on the Hierarchical Structure

In this subsection, we explain how we use the described hierarchical structure to solve packet routing. Concretely, the problem we address is defined as follows: the input is a number of source-destination pairs (s_i, t_i) where for each i we should deliver one message from the source node $s_i \in G$ to the destination node $t_i \in G$. For each packet, the source s_i knows the ID of the destination t_i. We are also given the promise that each node $v \in G$ is the source or destination in at most $d_G(v) \cdot O(\log n)$ packets. We explain a method that solves this problem in $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n \log\log n})}$ rounds[3]. The method is recursive, with one level of recursion for each level of the hierarchical structure.

A Preparation Step: Before starting the recursion, we redistribute the packets uniformly and randomly on G_0. Make each packet take a regular walk of length τ_{mix} on G, starting from its source. At the end, if the packet ended up in a node $u \in G$, assign it to a random G_0-node u_0 simulated by u. By Lemma 2.5, all these walks can be performed in $O(\tau_{\text{mix}} \log^2 n)$ rounds. At the end, the packets are distributed uniformly among nodes of G_0, and there are $O(\log n)$ source tokens in each G_0-node, with high probability. From now on, we work with these randomly redistributed packets.

The Recursion: We now explain the recursion for the packet routing problem, corresponding to our hierarchical structure. We solve the problem of routing on an m-node random graph G_0 by two consecutive iterations of routing on random graphs with $\tilde{O}(m/\beta)$ nodes, and some smaller transitional steps between these two recursions.

Consider the first random graph G_0 as well as the sets $A_1, A_2, \ldots, A_\beta$. For each packet with source and destination pair (s_ℓ, t_ℓ), there are two cases: either (1) the source s_ℓ and destination t_ℓ are in the same set A_i, or (2) the source is in set A_i and the destination is in a different set $A_j \neq A_i$. The former case can be directly solved as a problem of routing on the smaller graph $G_1[A_i]$, which has $\Theta(m/\beta)$ nodes, thus allowing recursion. The more interesting case is the latter. For the latter case, our approach is to first route the source token from s_ℓ to the portal node $t' \in A_i$ of s_ℓ towards A_j, which is adjacent to A_j in graph G_0. This portal node t' will be considered as the temporary destination while routing in A_i. Notice that node s_ℓ can know j because node s_ℓ knows the hierarchical partition label of the destination t_ℓ of the packet, by applying the common random hash function to the ID of t_ℓ, as described above. Since s_ℓ knows j, it can choose the portal t' as the temporary destination of the packet, while routing in A_i. Once the packet arrives at the portal t', we hop it over to a node $s' \in A_j$ that is G_0-neighbor of t'. Then, we route the packet from s' to its destination t_ℓ in A_j. The first and last part in this method are essentially smaller instances of the same routing problem, respectively, on $G_1[A_i]$ and $G_1[A_j]$. Hence, to solve the routing on the random graph G_0 with m nodes, we use two times of routing on each of disconnected random graphs of G_1 with $\Theta(m/\beta)$ nodes, as well as one time of hopping between the two sets. We next examine the round complexity of this recursion.

[3]Notice that the method can be easily extended to the case where each node is source or destination for up to $K = 2^{O(\sqrt{\log n \log\log n})}$ messages: randomly put each message in one of K phases. This ensures that per phase, each node is source or destination for $O(\log n)$ pairs. Solve each phase using the algorithm claimed above. Overall this becomes $K \cdot 2^{O(\sqrt{\log n \log\log n})} = 2^{O(\sqrt{\log n \log\log n})}$ rounds.

Lemma 3.4. *The round complexity of the described hierarchical routing problem is $2^{O(\sqrt{\log n \log\log n})}$ rounds of G_0, which means $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n \log\log n})}$ rounds of the base network G.*

Proof. Let us use $T(m)$ to denote the complexity of routing on m-node random graph G_0 with $O(\log n)$ tokens per node. We argue that $T(m) = 2T(\Theta(m/\beta)) \cdot O(\log^2 n) + O(\log n)$. Once that is established, given that $\beta = 2^{O(\sqrt{\log n \log\log n})}$ and $\log m = \Theta(\log n)$, we get $T(m) = 2^{O(\sqrt{\log n \log\log n})}$.

In the recursion $T(m) = 2T(\Theta(m/\beta)) \cdot O(\log^2 n) + O(\log n)$, the first term appears because of the following reason: to solve the m-node problem, we need to run two instances of $\Theta(m/\beta)$ node routing, one after the other. Moreover, these two subproblems are routing on $G_1[A_i]$ and $G_1[A_j]$, i.e., they are by communicating over graph G_1, which itself is embedded on G_0 with a emulation round complexity overhead of an $O(\log^2 n)$ factor. That is, each communication round of the graph G_1 can be emulated in $O(\log^2 n)$ communication rounds of the graph G_0.

We next explain the second term of the recursion. We particularly explain that we have enough capacity in G_0 to route the messages between different sets A_i in just $O(\log n)$ rounds. Fix an arbitrary pair of subsets A_i and A_j. Note that since G_0 is an Erdős-Rényi graph $G(m, p)$, the number of G_0-edges between A_i and A_j is $\Theta(m/\beta)^2 p = \Theta(m \log n/\beta^2)$. We note that this is under the assumption that $\beta \leq \sqrt{m}$, which is clearly satisfied for our choice of $\beta = 2^{O(\sqrt{\log n \log\log n})}$. On the other hand, since the source tokens are distributed uniformly and each node is destination for $O(\log n)$ tokens, the number of source tokens residing in A_i destined for A_j is $\Theta(m \log n/\beta) \cdot 1/\beta = \Theta(m \log n/\beta^2)$. Hence, the number of the edges between A_i and A_j in G_0 is, up to constant factor, equal to the number of the messages that should be passed from A_i to A_j. This means hopping the messages from A_i to A_j can be done in $O(\log n)$ rounds, as we spread the messages in A_i essentially uniformly over the endpoints of edges connecting to A_j, i.e., the portals. □

4 APPLICATIONS: MINIMUM SPANNING TREE AND MIN CUT

In this section, we explain our distributed MST algorithm. In any weighted graph $G = (V, E, W)$, this algorithm computes a minimum spanning tree in $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n \log\log n})}$ rounds. The core novelty in this result the hierarchical routing method developed in the previous section. Here, we explain how to incorporate this hierarchical routing method in the standard approach of Boruvka[58] for computing a minimum spanning tree. This will require a few smaller new ideas, as we next explain.

The overall outline of our algorithm follows the classic approach of Boruvka[58] for computing an MST, which is also the base of most distributed MST algorithms [28, 29, 44, 52]. This approach has $O(\log n)$ iterations. The key ingredient for us is to implement each iteration in $2^{O(\sqrt{\log n \log\log n})}$ rounds, using our hierarchical routing scheme. Let us first briefly recall Boruvka's approach.

Recalling Boruvka's Approach: We start with an empty spanning forest $F = (V, \emptyset)$. In $O(\log n)$ iterations, we gradually add more and more edges to F, until it becomes a tree. Consider one iteration and the connected components of the current forest F. Assuming that all edge weights are distinct, it is well-known that the MST is unique, and moreover, for each component C of F, the minimum weight edge that connects C to $V \setminus C$ belongs to the MST. We can add all such edges to F. This is one iteration of Boruvka's approach. One can see that this reduces the number of components by (at least) a factor 2. Thus, after $\log n$ iterations, we have a tree.

A Small Change in Boruvka's Approach: To have efficient distributed implementation of each iteration, we make a minor change, as follows: per iteration, each component tosses a fair coin and decides randomly to be either a *head* component or a *tail* component, each with probability 1/2. Different components decide independently. We then add to the MST only minimum-weight outgoing edges that go from a tail component to a head component. It is easy to see that per iteration, the number of components goes down by a constant factor, in expectation. Hence, using Markov's inequality, we see that after $O(\log n)$ iterations, we reach a tree, w.h.p. Thus, even with this random merging, $O(\log n)$ iterations suffice. The additional nice property provided by these random coins is as follows: now, each new component is the result of a *star shaped* merge of old components, where tail components merge with a central head component. Thus, in terms of the number of components, each merge has diameter 2. This simplifies the distributed computation of new components and the related structures.

Implementing Each Iteration of Boruvka using our Hierarchical Routing: We now explain how in each iteration, we compute the minimum-weight outgoing edges of different components, in $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n \log\log n})}$ rounds, and how we compute the identify the resulting merged components. Let F be the current forest, at the beginning of the iteration. For each component C of the forest F, we maintain a *virtual tree* $\mathcal{T}(C)$ spanning C. These *virtual trees* will be used for organizing the process of routing messages while computing the minimum weight edges. Particularly, we will preserve three key properties for these trees: (2) the depth of each virtual tree $\mathcal{T}(C)$ is at most $O(\log^2 n)$, (2) in each virtual tree $\mathcal{T}(C)$, each node $v \in G[C]$ has at most $d_G(v) \cdot O(\log n)$ edges, and (3) each virtual node knows its parent in the virtual tree $\mathcal{T}(C)$.

We explain in the proof of Lemma 4.1 how to preserve these properties throughout the iterations. Let us discuss how we use these trees. To compute the minimum weight outgoing edge of each component C, we use a simple minimum computation upcast on the virtual tree $\mathcal{T}(C)$ of the component, as follows: we start with the bottom level of $\mathcal{T}(C)$ and each node sends to its $\mathcal{T}(C)$-parent its minimum-weight edge going out of C. This is done in $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n \log\log n})}$ rounds, by invoking the hierarchical routing scheme described in the previous section. We can invoke the routing scheme because of two reasons: (1) each node v in $\mathcal{T}(C)$ has degree at most $d_G(v) \cdot O(\log n)$, which means each node is the source or destination of at most $d_G(v) \cdot O(\log n)$ messages, and (2) each node knows its parent, which is the destination of its packet.

We repeat this procedure once for each level of $\mathcal{T}(C)$, and thus $O(\log^2 n)$ times in total. Each time, when a node receives some outgoing edges from its virtual tree children, and possibly having some of its own, it only remembers the outgoing edge with the smallest weight, and forwards this minimum to its own $\mathcal{T}(C)$-parent in the next upcast step. Since the virtual tree has depth $O(\log^2 n)$, after $O(\log^2 n)$ upcast steps, the root of the virtual tree gets the minimum weight outgoing edge of component C. Then, by reversing the direction of these communications, we can deliver this minimum weight outgoing edge e^* to all nodes of the component, within the same round complexity. This particularly informs the node at the endpoint of this chosen minimum weight outgoing edge e^* that edge e^* should be added to the MST. The next lemma shows that we can maintain these virtual trees throughout the merges.

Lemma 4.1. *Throughout the iterations, using $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n \log\log n})}$ rounds of computation per iteration, we can preserve the three key properties of virtual trees: (1) the depth of each virtual tree $\mathcal{T}(C)$ is at most $O(\log^2 n)$, (2) in each virtual tree $\mathcal{T}(C)$, each node $v \in G[C]$ has at most $d_G(v) \cdot O(\log n)$ edges, and (3) each node knows its parent in the virtual tree.*

Proof of Lemma 4.1. Initially, when each F-component is just a singleton node, the related virtual tree is also simply that singleton node. What we need to explain is how maintain the virtual tree, when some components of F merge. That is, how to construct the virtual tree of a new component resulting from the merge of a number of F-components $C_0, C_1, \ldots, C_k$ merge, where C_0 is the component of the merge, who had a head coin toss, and C_1 to C_k are the tail components of the merge. This construction/maintenance is done in a manner that provides the above three properties.

We merge the their virtual trees of $\mathcal{T}(C_0), \mathcal{T}(C_1), \ldots, \mathcal{T}(C_k)$, as follows: For each $i \in [1, k]$, let e_i be the minimum weight outgoing edge connecting C_i to C_0, which is to be added to F, and let node $v_i \in C_0$ be the physical endpoint of e_i in C_0. Let $v'_i \in \mathcal{T}(C_i)$ be the virtual node corresponding to v_i. We then define v'_i as the virtual parent of the root of $\mathcal{T}(C_i)$. For each node $v_i \in C_0$, this adds at most $d_G(v)$ new incoming virtual edges for v_i. Hence, over the $O(\log n)$ iterations of Burovka's approach, the number of virtual edges incoming to v_i is at most $d_G(v) \cdot O(\log n)$, as desired.

This step may increase the depth of the virtual tree. We now start a sort of a balancing process, where we redefine the parents of these virtual nodes v'_i, and consequently their new parents, in a way that allows us to bound the depth of the virutal tree.

This process of finding new parents will be done by walking up the virtual tree of $\mathcal{T}(C_0)$, starting from nodes v'_i, which are the virtual nodes corresponding to the C_0 endpoints of recently added F-edges. We start from the bottom level of $\mathcal{T}(C_0)$ and move up in a synchronous manner, each time going one level higher in $\mathcal{T}(C_0)$. During this process, we try to redefine the parents of v'_i in a way that they induce a shallow tree, similar to a balanced binary tree.

Concretely, the process is as follows: we create one token at each virtual node v'_i. Then, we upcast these tokens from the deepest level of $\mathcal{T}(C_0)$ towards its root. Whenever two or more tokens arrive (simultaneously) at a node $v \in \mathcal{T}(C_0)$, we essentially merge them. Then, we create a new token at this merge point, and we upcast only this new token towards the root of $\mathcal{T}(C_0)$.

The merge process has some steps, as follows. Suppose two or more tokens arrived at $v \in \mathcal{T}(C_0)$. For each of these tokens, if the token's creation point was a child u of v, we do not do anything and just keep v as the parent of u in the virtual tree. Otherwise, we create a new virtual edge, which connects the creation point w of the token to a child u of v, where u is the child through which the token arrived at v. Then, node u is the new parent of node w. Node v still remains as the parent of u. Notice that in this case, node u was not one of the places were two or more tokens merged, and hence, we have increased the in-degree of u by at most 1. On the other hand, the in-degree of node v does not change. Now, node v creates a new token, and sends it upwards in $\mathcal{T}(C_0)$, effectively searching for a new parent for itself.

Throughout this balancing process, each node's in-degree grows by an additive one. This is in addition to the at most $d_G(v)$ in-edges added at the beginning to each node v, which correspond to the just merged F-edges. Hence, over all the $O(\log n)$ iterations of Boruvka, the in-degree of each node v is at most $d_G(v) \cdot O(\log n)$.

We now discuss the depth. In each iteration of Boruvka's approach, when we do the virtual tree maintenance described above, each virtual node's distance from its root increases by at most an additive $O(\log n)$. Hence, overall the distance from the root remains bounded to $O(\log^2 n)$. The $O(\log n)$ increase is because the addition to the distance from the root is exactly the depth in the newly defined part of the $\mathcal{T}(C_0)$, from v'_i to the root. This part has depth $O(\log n)$, because it is a tree where in each root-leaf path, each other node has at least 2 children. □

By incorporating this method into the framework of [31, Section 5], essentially in a black-box manner, we can compute a $(1+\varepsilon)$ approximation of the min-cut, for any constant $\varepsilon > 0$, in $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n \log\log n})}$ rounds. We defer the details to the full version.

REFERENCES

[1] David Aldous and Jim Fill. 2002. Reversible Markov chains and random walks on graphs. (2002).

[2] Noga Alon, Chen Avin, Michal Koucký, Gady Kozma, Zvi Lotker, and Mark R Tuttle. 2011. Many random walks are faster than one. *Combinatorics, Probability and Computing* 20, 04 (2011), 481–502.

[3] Noga Alon and Joel H Spencer. 2004. *The probabilistic method.* John Wiley & Sons.

[4] John Augustine, Gopal Pandurangan, Peter Robinson, Scott Roche, and Eli Upfal. 2015. Enabling Robust and Efficient Distributed Computation in Dynamic Peer-to-Peer Networks. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on.* IEEE, 350–369.

[5] Chen Avin, Michael Borokhovich, Zvi Lotker, and David Peleg. 2014. Distributed computing on core-periphery networks: Axiom-based design. In *International Colloquium on Automata, Languages, and Programming.* Springer, 399–410.

[6] Chen Avin, Michal Koucký, and Zvi Lotker. 2008. How to explore a fast-changing world (cover time of a simple random walk on evolving graphs). In *International Colloquium on Automata, Languages, and Programming.* Springer, 121–132.

[7] B. Awerbuch. 1987. Optimal distributed algorithms for minimum-weight spanning tree, counting, leader election and related problems. In *Proc. of the 19th Symposium on Theory of Computing (STOC).* 230–240.

[8] Baruch Awerbuch and Christian Scheideler. 2004. The hyperring: a low-congestion deterministic data structure for distributed environments. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics, 318–327.

[9] Alkida Balliu, Pierre Fraigniaud, Zvi Lotker, and Dennis Olivetti. 2016. Sparsifying congested cliques and core-periphery networks. In *International Colloquium on Structural Information and Communication Complexity.* Springer, 307–322.

[10] Florent Becker, Antonio Fernandez Anta, Ivan Rapaport, and Eric Reémila. 2015. Brief Announcement: A Hierarchy of Congested Clique Models, from Broadcast to Unicast. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC) (PODC '15).* ACM, 167–169.

[11] Andrew Berns, James Hegeman, and Sriram V Pemmaraju. 2012. Super-fast distributed algorithms for metric facility location. In *Automata, Languages, and Programming.* Springer, 428–439.

[12] AZ Broder, AM Frieze, and E Upfal. 1997. Existence and construction of edge low congestion paths on expander graphs. In *Proc. of the Symp. on Theory of Comp. (STOC).* 531–539.

[13] Andrei Z Broder, Alan M Frieze, and Eli Upfal. 1994. Existence and construction of edge-disjoint paths on expander graphs. *SIAM J. Comput.* 23, 5 (1994), 976–989.

[14] Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. 2015. Algebraic Methods in the Congested Clique. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*. ACM, 143–152.

[15] F Chin and HF Ting. 1985. An almost linear time and O(nlogn+ e) messages distributed algorithm for minimum-weight spanning trees. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*. 257–266.

[16] Colin Cooper, Alan Frieze, and Tomasz Radzik. 2009. Multiple random walks in random regular graphs. *SIAM J. on Discrete Math.* 23, 4 (2009), 1738–1761.

[17] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. 2011. Distributed Verification and Hardness of Distributed Approximation. In *Proc. of the Symp. on Theory of Comp. (STOC)*. 363–372.

[18] Atish Das Sarma, Danupon Nanongkai, and Gopal Pandurangan. 2009. Fast distributed random walks. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 161–170.

[19] Atish Das Sarma, Danupon Nanongkai, Gopal Pandurangan, and Prasad Tetali. 2010. Efficient Distributed Random Walks with Applications. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*. 201–210.

[20] Danny Dolev, Christoph Lenzen, and Shir Peled. 2012. âĂIJTri, Tri AgainâĂİ: Finding Triangles and Small Subgraphs in a Distributed Setting. In *Distributed Computing*. Springer, 195–209.

[21] Andrew Drucker, Fabian Kuhn, and Rotem Oshman. 2014. On the Power of the Congested Clique Model. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*. ACM, 367–376.

[22] Michael Elkin. 2004. A faster distributed protocol for constructing a minimum spanning tree. In *Pro. of ACM-SIAM Symp. on Disc. Alg. (SODA)*. 359–368.

[23] Michael Elkin. 2004. Unconditional Lower Bounds on the Time-approximation Tradeoffs for the Distributed Minimum Spanning Tree Problem. In *Proc. of the Symp. on Theory of Comp. (STOC)*. 331–340.

[24] Pául Erdos. 1942. On an elementary proof of some asymptotic formulas in the theory of partitions. *Annals of Mathematics* (1942), 437–450.

[25] Alan M Frieze. 1998. Disjoint paths in expander graphs via random walks: A short survey. In *Rand. and Approx. Techniques in Comp. Sci.* Springer, 1–14.

[26] Alan M Frieze. 2001. Edge-disjoint paths in expander graphs. *SIAM J. on Computing* 30, 6 (2001), 1790–1801.

[27] Eli Gafni. 1985. Improvements in the time complexity of two message-optimal election algorithms. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*. ACM, 175–185.

[28] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. 1983. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Languages and systems (TOPLAS)* 5, 1 (1983), 66–77.

[29] J. Garay, S. Kutten, and D. Peleg. 1998. A sub-linear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comput.* 27 (1998), 302–316.

[30] Mohsen Ghaffari. 2016. An Improved Distributed Algorithm for Maximal Independent Set. In *Pro. of ACM-SIAM Symp. on Disc. Alg. (SODA)*.

[31] Mohsen Ghaffari and Bernhard Haeupler. 2016. Distributed Algorithms for Planar Networks II: Low-Congestion Shortcuts, MST, and Min-Cut. In *Pro. of ACM-SIAM Symp. on Disc. Alg. (SODA)*.

[32] M. Ghaffari and F. Kuhn. 2013. Distributed Minimum Cut Approximation. In *Proc. of the Int'l Symp. on Dist. Comp. (DISC)*. 1–15.

[33] Mohsen Ghaffari and Merav Parter. 2016. MST in Log-Star Rounds of Congested Clique. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*. 19–28.

[34] Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. 2016. Low-congestion shortcuts without embedding. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. ACM, 451–460.

[35] Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. 2016. Near-Optimal Low-Congestion Shortcuts on Bounded Parameter Graphs. In *International Symposium on Distributed Computing*. Springer, 158–172.

[36] James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Scquizzato. 2015. Toward Optimal Bounds in the Congested Clique: Graph Connectivity and MST. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*. ACM, 91–100.

[37] James W Hegeman and Sriram V Pemmaraju. 2014. Lessons from the congested clique applied to MapReduce. In *the Proceedings of the International Colloquium on Structural Information and Communication Complexity*. Springer, 149–164.

[38] James W Hegeman, Sriram V Pemmaraju, and Vivek B Sardeshmukh. 2014. Near-constant-time distributed algorithms on a congested clique. In *Distributed Computing*. Springer, 514–530.

[39] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2016. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 489–498.

[40] Mark Jerrum and Alistair Sinclair. 1989. Approximating the permanent. *SIAM journal on computing* 18, 6 (1989), 1149–1178.

[41] Maleq Khan and Gopal Pandurangan. 2008. A fast distributed approximation algorithm for minimum spanning trees. *Distributed Comp.* 20, 6 (2008), 391–402.

[42] Janne H Korhonen. 2016. Deterministic MST Sparsification in the Congested Clique. *arXiv preprint arXiv:1605.02022* (2016).

[43] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. 2010. Towards worst-case churn resistant peer-to-peer systems. *Distributed Comp.* 22, 4 (2010), 249–267.

[44] Shay Kutten and David Peleg. 1995. Fast Distributed Construction of K-dominating Sets and Applications. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*. 238–251.

[45] Ching Law and Kai-Yeung Siu. 2003. Distributed construction of random expander networks. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, Vol. 3. IEEE, 2133–2143.

[46] FT Leighton and Satish Rao. 1996. Circuit switching: a multicommodity ow based approach,". In *Proc. of a Workshop on Randomized Parallel Comp.*

[47] Tom Leighton, Chi-Jen Lu, Satish Rao, and Aravind Srinivasan. 2001. New algorithmic aspects of the Local Lemma with applications to routing and partitioning. *SIAM J. Comput.* 31, 2 (2001), 626–641.

[48] Tom Leighton and Satish Rao. 1999. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM (JACM)* 46, 6 (1999), 787–832.

[49] Tom Leighton, Satish Rao, and Aravind Srinivasan. 1998. Multicommodity flow and circuit switching. In *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, Vol. 7. IEEE, 459–465.

[50] Christoph Lenzen. 2013. Optimal Deterministic Routing and Sorting on the Congested Clique. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*. 42–50.

[51] Zvi Lotker, Boaz Patt-Shamir, and David Peleg. 2001. Distributed MST for constant diameter graphs. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. ACM, 63–71.

[52] Zvi Lotker, Boaz Patt-Shamir, and David Peleg. 2006. Distributed MST for constant diameter graphs. *Distributed Computing* 18, 6 (2006), 453–460.

[53] Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. 2003. MST construction in O(log $logn$) communication rounds. In *the Proceedings of the Symposium on Parallel Algorithms and Architectures*. ACM, 94–100.

[54] László Lovász. 1993. Random walks on graphs: A survey. *Combinatorics, Paul erdos is eighty* 2, 1 (1993), 1–46.

[55] Peter Mahlmann and Christian Schindelhauer. 2005. Peer-to-peer networks based on random transformations of connected regular undirected graphs. In *the Proceedings of the Symposium on Parallel Algorithms and Architectures*. 155–164.

[56] Danupon Nanongkai. 2014. Distributed Approximation Algorithms for Weighted Shortest Paths. In *Proc. of the Symp. on Theory of Comp. (STOC)*.

[57] D. Nanongkai and H.-H. Su. 2014. Almost-Tight Distributed Minimum Cut Algorithms. In *Proc. of the Int'l Symp. on Dist. Comp. (DISC)*. 439–453.

[58] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. 2001. Otakar Boruvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics* 233, 1 (2001), 3–36.

[59] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the web. (1999).

[60] Gopal Pandurangan, Peter Robinson, and Amitabh Trehan. 2014. DEX: self-healing expanders. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 702–711.

[61] Gopal Pandurangan and Amitabh Trehan. 2011. Xheal: localized self-healing using expanders. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*. ACM, 301–310.

[62] Gopal Peer-to-Peer Networks Pandurangan, Prabhakar Raghavan, Eli Upfal, et al. 2003. Building low-diameter peer-to-peer networks. *Selected Areas in Communications, IEEE Journal on* 21, 6 (2003), 995–1002.

[63] Boaz Patt-Shamir and Marat Teplitsky. 2011. The round complexity of distributed sorting. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*. 249–256.

[64] D. Peleg and V. Rubinovich. 2001. A near-tight lower bound on the time complexity of distributed MST construction. *SIAM J. Comput.* 30, 5 (2001), 1427–1442.

[65] David Peleg and Eli Upfal. 1989. Constructing disjoint paths on expander graphs. *Combinatorica* 9, 3 (1989), 289–313.

[66] Jeanette P Schmidt, Alan Siegel, and Aravind Srinivasan. 1995. Chernoff-Hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics* 8, 2 (1995), 223–250.

[67] Daniel A. Spielman. 2015. Spectral Graph Theory : Random Walks on Graphs. lecture notes. (2015). http://www.cs.yale.edu/homes/spielman/561/2012/lect10-12.pdf.

[68] Frank Spitzer. 2013. *Principles of random walk*. Vol. 34. Springer Science & Business Media.

[69] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review* 31, 4 (2001), 149–160.

[70] G Weiss. 2005. Aspects and Applications of the Random Walk (Random Materials & Processes S.). (2005).