A Time- and Message-Optimal Distributed Algorithm for Minimum Spanning Trees

Gopal Pandurangan*
University of Houston
Houston, TX, USA
gopalpandurangan@gmail.com

Peter Robinson
Royal Holloway, University of London
London, UK
peter.robinson@rhul.ac.uk

Michele Scquizzato
University of Houston
Houston, TX, USA
michele@cs.uh.edu

ABSTRACT

This paper presents a randomized (Las Vegas) distributed algorithm that constructs a minimum spanning tree (MST) in weighted networks with optimal (up to polylogarithmic factors) time and message complexity. This algorithm runs in $\tilde{O}(D+\sqrt{n})$ time and exchanges $\tilde{O}(m)$ messages (both with high probability), where n is the number of nodes of the network, D is the diameter, and m is the number of edges. This is the first distributed MST algorithm that matches *simultaneously* the time lower bound of $\tilde{\Omega}(D+\sqrt{n})$ [Elkin, SIAM J. Comput. 2006] and the message lower bound of $\Omega(m)$ [Kutten et al., J. ACM 2015], which both apply to randomized Monte Carlo algorithms.

The prior time and message lower bounds are derived using two completely different graph constructions; the existing lower bound construction that shows one lower bound does not work for the other. To complement our algorithm, we present a new lower bound graph construction for which any distributed MST algorithm requires both $\tilde{\Omega}(D+\sqrt{n})$ rounds and $\Omega(m)$ messages.

CCS CONCEPTS

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

KEYWORDS

Distributed algorithms; Minimum spanning trees

ACM Reference format:
Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. 2017. A Time- and Message-Optimal Distributed Algorithm for Minimum Spanning Trees. In *Proceedings of 49th Annual ACM SIGACT Symposium on the Theory of Computing, Montreal, Canada, June 2017 (STOC'17),* 14 pages.
DOI: 10.1145/3055399.3055449

1 INTRODUCTION

The minimum-weight spanning tree (MST) construction problem is one of the central and most studied problems in distributed computing. A long line of research aimed at developing efficient distributed algorithms for the MST problem started more than thirty years ago with the seminal paper of Gallager, Humblet, and Spira [13], which presented a distributed algorithm that constructs an MST in $O(n \log n)$ rounds and exchanging $O(m+n\log n)$ messages[1] (throughout, n and m will denote the number of nodes and the number of edges of the network, respectively). The message complexity of this algorithm is (essentially) optimal,[2] but its time complexity is not. Hence further research concentrated on improving the time complexity. The time complexity was first improved to $O(n \log\log n)$ by Chin and Ting [5], further improved to $O(n\log^* n)$ by Gafni [12], and then to $O(n)$ by Awerbuch [2] (see also Faloutsos and Molle [11]). The $O(n)$ bound is existentially optimal in the sense that there exist graphs for which this is the best possible.

This was the state of the art till the mid-nineties when Garay, Kutten, and Peleg [14] raised the question of whether it is possible to identify graph parameters that can better capture the complexity of distributed network computations. In fact, for many existing networks, their diameter[3] D is significantly smaller than the number of vertices n, and therefore it is desirable to design protocols whose running time is bounded in terms of D rather than in terms of n. Garay, Kutten, and Peleg [14] gave the first such distributed algorithm for the MST problem with running time $O(D+n^{0.614}\log^* n)$, which was later improved by Kutten and Peleg [23] to $O(D+\sqrt{n}\log^* n)$. However, both these algorithms are not message-optimal,[4] as they exchange $O(m+n^{1.614})$ and $O(m+n^{1.5})$ messages, respectively. All the above results, as well as the one in this paper, hold in the synchronous CONGEST model of distributed computing, a well-studied standard model of distributed computing [30] (see Section 1.1).

The lack of progress in improving the result of [23], and in particular breaking the $\tilde{O}(\sqrt{n})$ barrier,[5] led to work on lower bounds for the distributed MST problem. Peleg and Rubinovich [31] showed that $\Omega(D+\sqrt{n}/\log n)$ time is required by any distributed algorithm for constructing an MST, even on networks of small diameter ($D=\Omega(\log n)$); thus, this result establishes the asymptotic near-tight optimality of the algorithm of [23]. The lower bound of Peleg and Rubinovich applies to exact, deterministic algorithms. Later,

*Supported, in part, by NSF grants CCF-1527867, CCF-1540512, and IIS-1633720.

STOC'17, Montreal, Canada

978-1-4503-4528-6/17/06…$15.00
DOI: 10.1145/3055399.3055449

[1] The original algorithm has a message complexity of $O(m\log n)$, but it can be improved to $O(m+n\log n)$.

[2] It has been shown in [22] that the message complexity lower bound of leader election (and hence any spanning tree as well) is $\Omega(m)$, and this applies even to randomized Monte Carlo algorithms. On the other hand, it can be shown that an MST can be constructed using $O(m)$ messages (but time can be arbitrarily large) in any synchronous network [22, 28].

[3] In this paper, by diameter we always mean unweighted diameter.

[4] In this paper, henceforth, when we say "optimal" we mean "optimal up to a polylog(n) factor".

[5] $\tilde{O}(f(n))$ and $\tilde{\Omega}(f(n))$ denote $O(f(n)\cdot \text{polylog}(f(n)))$ and $\Omega(f(n)/\text{polylog}(f(n)))$, respectively.

the same lower bound of $\tilde{\Omega}(D + \sqrt{n})$ was shown for randomized (Monte Carlo) and approximation algorithms as well [6, 9].

To summarize, the state of the art for distributed MST algorithms is that there exist algorithms which are either time-optimal (i.e., they run in $\tilde{O}(D + \sqrt{n})$ time) or message-optimal (i.e., they exchange $\tilde{O}(m)$ messages), but not simultaneously both. Indeed, the time-optimal algorithms of [8, 23] (as well as the sublinear time algorithm of [14]) are not message-optimal, i.e., they require asymptotically much more than $\Theta(m)$ messages. In contrast, the known message-optimal algorithms for MST (in particular, [2, 13]) are not time-optimal, i.e., they take significantly more time than $\tilde{O}(D+\sqrt{n})$. Peleg and Rubinovich [31] in their 2000 SICOMP paper raise the question of whether one can design a distributed MST algorithm that is *simultaneously* optimal with respect to time and message complexity. In 2011, Kor, Korman, and Peleg [20] also raise this question and showed that distributed *verification* of MST, i.e., verifying whether a given spanning tree is MST or not, can be done in optimal messages and time, i.e., there exists a distributed verification algorithm that uses $\tilde{O}(m)$ messages and runs in $\tilde{O}(D + \sqrt{n})$ time, and that these are optimal bounds for MST verification. However, the original question for MST construction remained open.

The above question addresses a fundamental aspect in distributed algorithms, namely the relationship between the two basic complexity measures of time and messages. The simultaneous optimization of both time and message complexity has been elusive for several fundamental distributed problems (including MST, shortest paths, and random walks), and consequently research in the last three decades in distributed algorithms has focused mainly on optimizing either one of the two measures separately. However, in various modern and emerging applications such as resource-constrained communication networks and distributed computation of large-scale data, it is crucial to design distributed algorithms that optimize both measures *simultaneously* [15, 19].

1.1 Model and Definitions

We first briefly describe the distributed computing model in which our algorithm (as well as all the previously discussed MST algorithms [2, 5, 8, 12–14, 23]) is specified and analyzed. This is the CONGEST model (see, e.g., the book by Peleg [30]), which is now standard in the distributed computing literature.

A point-to-point communication network is modeled as an undirected weighted graph $G = (V, E, w)$, where the vertices of V represent the processors, the edges of E represent the communication links between them, and $w(e)$ is the weight of edge $e \in E$. Without loss of generality, we assume that G is connected. We also assume that the weights of the edges of the graph are all distinct. This implies that the MST of the graph is unique. The definitions and the results generalize readily to the case where the weights are not necessarily distinct. Each node hosts a processor with limited initial knowledge. Specifically, we make the common assumption that each node has unique identity numbers (this is not essential, but simplifies presentation), and at the beginning of computation each vertex v accepts as input its own identity number and the weights of the edges incident to it. Thus, a node has only *local* knowledge. Specifically we assume that each node has ports (each port having a unique port number); each incident edge is connected to one distinct port. A node does not have any initial knowledge of the other endpoint of its incident edge (which node it is connected to or the port number that it is connected to). This model is referred to as the *clean network model* in [30] and is also sometimes referred to as the KT_0 model, i.e., the initial (K)nowledge of all nodes is restricted (T)ill radius 0 (i.e., just the local knowledge) [30]. The KT_0 model is a standard model in distributed computing and typically used in the literature (see e.g., [1, 25, 30, 33]), including all the prior results on distributed MST (e.g., [2, 5, 8, 12–14, 23]) with a notable exception ([18], discussed in some detail in Section 1.3).

The vertices are allowed to communicate through the edges of the graph G. It is assumed that communication is synchronous and occurs in discrete rounds (time steps). In each time step, each node v can send an arbitrary message of $O(\log n)$ bits through each edge $e = (v, u)$ incident to v, and each message arrives at u by the end of this time step. (If unbounded-size messages are allowed—this is the so-called LOCAL model—the MST problem can be trivially solved in $O(D)$ time [30].) The weights of the edges are at most polynomial in the number of vertices n, and therefore the weight of a single edge can be communicated in one time step. This model of distributed computation is called the CONGEST($\log n$) model or simply the CONGEST model [30].

Singular Optimality vs. Time-Message Tradeoff. The efficiency of distributed algorithms is traditionally measured by their time and message (or, communication) complexities. Time complexity measures the number of synchronous rounds taken by the algorithm, whereas message complexity measures the total amount of messages sent and received by all the processors during the execution of the algorithm. Both complexity measures crucially influence the performance of a distributed algorithm. We say that a problem enjoys *singular optimality* when it admits a distributed algorithm whose time and message complexity are both optimal. When the problem fails to admit such a solution, namely, algorithms with better time complexity for it necessarily incur higher message complexity and vice versa, we say that the problem exhibits a *time-message tradeoff*.

1.2 Our Results

Distributed MST Algorithm. In this paper we present a distributed MST algorithm in the CONGEST model which is simultaneously time- and message-optimal. The algorithm is randomized Las Vegas, and always returns the MST. The running time of the algorithm is $\tilde{O}(D + \sqrt{n})$ and the message complexity is $\tilde{O}(m)$, and both bounds hold with high probability.[6] This is the first distributed MST algorithm that matches *simultaneously* the time lower bound of $\tilde{\Omega}(D + \sqrt{n})$ [6, 9] and the message lower bound of $\Omega(m)$ [22], which both apply even to randomized Monte Carlo algorithms, thus closing a more than thirty-year-old line of research in distributed computing. In terms of the terminology introduced earlier, we can therefore say that the distributed MST problem exhibits singular optimality up to polylogarithmic factors. Table 1 summarizes the known upper bounds on the complexity of distributed MST. We also observe that in our algorithm the local computation performed by the vertices is not very heavy.

[6]Throughout, with high probability (w.h.p.) means with probability $\geqslant 1 - 1/n^{\Omega(1)}$, where n is the network size.

Table 1: Summary of upper bounds on the complexity of distributed MST.

Reference	Time Complexity	Message Complexity
Gallager et al. [13]	$O(n \log n)$	$O(m + n \log n)$
Awerbuch [2]	$O(n)$	$O(m + n \log n)$
Garay et al. [14]	$O(D + n^{0.614} \log^* n)$	$O(m + n^{1.614})$
Kutten and Peleg [23]	$O(D + \sqrt{n} \log^* n)$	$O(m + n^{1.5})$
Elkin [8]	$\tilde{O}(\mu(G, w) + \sqrt{n})$	$O(m + n^{1.5})$
This paper	$\tilde{O}(D + \sqrt{n})$	$\tilde{O}(m)$

Lower Bound. Both the aforementioned time and message lower bounds are existential, and are derived using two completely different graph constructions. However, the graph used to show one lower bound *does not* work for the other. To complement our main result, in Section 4 we present a new graph construction for which any distributed MST algorithm requires *both* $\tilde{\Omega}(D + \sqrt{n})$ rounds and $\Omega(m)$ messages.

1.3 Other Related Work

Given the importance of the distributed MST problem, there has been significant work over the last 30 years on this problem and related aspects. Besides the prior work already mentioned in Section 1, we now discuss other relevant work on distributed MST.

Other Distributed MST Algorithms. Elkin [8] showed that a parameter called "MST-radius" captures the complexity of distributed MST algorithms better. He devised a distributed protocol that constructs the MST in $\tilde{O}(\mu(G, w) + \sqrt{n})$ time, where $\mu(G, w)$ is the "MST-radius" of the graph [8] (is a function of the graph topology as well as the edge weights). The ratio between diameter and MST-radius can be as large as $\Theta(n)$, and consequently, on some inputs, this protocol is faster than the protocol of [23] by a factor of $\Omega(\sqrt{n})$. However, a drawback of this protocol (unlike the previous MST protocols [5, 12–14, 23]) is that it cannot detect the termination of the algorithm in that time (unless $\mu(G, w)$ is given as part of the input). On the other hand, it can be shown that for distributed MST algorithms that correctly terminate $\Omega(D)$ is a lower bound on the running time [21, 31]. (In fact, [21] shows that for every sufficiently large n and every function $D(n)$ with $2 \leqslant D(n) < n/4$, there exists a graph G of $n' \in \Theta(n)$ nodes and diameter $D' \in \Theta(D(n))$ which requires $\Omega(D')$ rounds to compute a spanning tree with constant probability.) We also note that the message complexity of Elkin's algorithm is $O(m + n^{3/2})$.

Time Bounds. From a practical perspective, given that MST construction can take as much as $\Omega(\sqrt{n}/\log n)$ time even in low-diameter networks, it is worth investigating whether one can design distributed algorithms that run faster and output an approximate minimum spanning tree. The question of devising faster approximation algorithms for MST was raised in [31]. Elkin [9] later established a hardness result on distributed MST approximation, showing that *approximating* the MST problem on a certain family of graphs of small diameter (e.g., $O(\log n)$) within a ratio H requires essentially $\Omega(\sqrt{n/H \log n})$ time. Khan and Pandurangan [17] showed that there can be an exponential time gap between exact and approximate MST construction by showing that there exist graphs where any distributed (exact) MST algorithm takes $\Omega(\sqrt{n}/\log n)$ rounds, whereas an $O(\log n)$-approximate MST can be computed in $O(\log n)$ rounds. The distributed algorithm of Khan and Pandurangan [17] outputs a $O(\log n)$-approximate MST, and is message-optimal but not time-optimal.

Das Sarma et al. [6] settled the time complexity of distributed approximate MST by showing that this problem, as well as approximating shortest paths and about twenty other problems, satisfies a time lower bound of $\tilde{\Omega}(D + \sqrt{n})$. This applies to deterministic as well as randomized algorithms, and to both exact and approximate versions. In other words, any distributed algorithm for computing a H-approximation to MST, for any $H > 0$, takes $\tilde{\Omega}(D + \sqrt{n})$ time in the worst case. Lower bounds are known even for quantum algorithms [10].

Message Bounds. Kutten et al. [22] fully settled the message complexity of leader election in general graphs, even for randomized algorithms and under very general settings. Specifically, they showed that any randomized algorithm (including Monte Carlo algorithms with suitably large constant success probability) requires $\Omega(m)$ messages; this lower bound holds for any n and m, i.e., given any n and m, there exists a graph with $\Theta(n)$ nodes and $\Theta(m)$ edges for which the lower bound applies. Since a distributed MST algorithm can also be used to elect a leader (where the root of the tree is the leader, which can be chosen using $O(n)$ messages once a tree is constructed) the above lower bound applies to distributed MST construction as well, for all $m \geqslant cn$, where c is a sufficiently large constant. The above bound holds even for *non-comparison* algorithms, that is algorithms that may also manipulate the actual value of node's identities, not just compare identities with each other, and even if nodes have initial knowledge of n, m, and D. They also hold for synchronous networks, and even if all the nodes wake up simultaneously. Finally, they hold not only for the CONGEST model [30], where sending a message of $O(\log n)$ bits takes one unit of time, but also for the LOCAL model [30], where the number of bits in a message is allowed to be arbitrary.

Optimality in the KT_1 Model: Comparison-Based and Randomized Algorithms. It is important to point out that this paper and all the prior results discussed above (including the prior MST results [2, 5, 8, 12–14, 23]) assume the so-called *clean network model*, a.k.a. KT_0 [30] (cf. Section 1.1), where nodes do not have initial knowledge of the identity of their neighbors. However, one can assume a model where nodes have initial knowledge of the identity of their neighbors. This model is called the KT_1 *model*. We note that the time lower bound of $\tilde{\Omega}(D + \sqrt{n})$ holds in the KT_1 model as well. Awerbuch et al. [3] show that $\Omega(m)$ is a message lower bound for MST for the KT_1 model, if one allows only comparison-based algorithms (i.e., algorithms that can operate on IDs only by comparing them); this lower bound for comparison-based algorithms applies to *randomized* algorithms as well. (We note that all prior MST algorithms mentioned earlier are comparison-based, including ours.) Hence, the result of [3] implies that our MST algorithm (which is comparison-based and randomized) is *message- and time-optimal* in the KT_1 model if one considers comparison-based algorithms.

Awerbuch et al. [3] also show that the $\Omega(m)$ message lower bound applies even to non-comparison based (in particular, algorithms that can perform arbitrary local computations) *deterministic* algorithms in the CONGEST model that terminate in a time bound that depends only on the graph topology (e.g., a function of n). On the other hand, for *randomized non-comparison-based* algorithms, it turns out that the message lower bound of $\Omega(m)$ does not apply in the KT_1 model. Recently, King et al. [18] showed a surprising and elegant result: in the KT_1 model one can give a randomized Monte Carlo algorithm to construct a MST in $\tilde{O}(n)$ messages ($\Omega(n)$ is a message lower bound) and in $\tilde{O}(n)$ time (this algorithm uses randomness and is not comparison-based). While this algorithm shows that one can get $o(m)$ message complexity (when $m = \omega(n \operatorname{polylog} n)$), it is *not* time-optimal (it can take significantly more than $\tilde{\Theta}(D + \sqrt{n})$ rounds). It is an open question whether one can design a randomized (non-comparison based) algorithm that takes $\tilde{O}(D + \sqrt{n})$ time and $\tilde{O}(n)$ messages in the KT_1 model.

2 HIGH-LEVEL OVERVIEW OF THE ALGORITHM

The time- and message-optimal distributed MST algorithm of this paper builds on prior distributed MST algorithms that were either message-optimal or time-optimal but *not both*. We provide a high-level overview of our algorithm and some intuition behind it; we also compare and contrast it with previous MST algorithms. The full description of the algorithm and its analysis are given in Section 3. The algorithm can be divided into two parts as explained below.

2.1 First Part: Controlled-GHS

We first run the so-called Controlled-GHS algorithm, which was first used in the sublinear-time distributed MST algorithm of Garay, Kutten, and Peleg [14], as well as in the time-optimal algorithm of Kutten and Peleg [23]. Controlled-GHS is the (synchronous version of the) classical Gallager-Humblet-Spira (GHS) algorithm [13, 30], with some modifications. We recall that the synchronous GHS algorithm, which is essentially a distributed implementation of Boruvka's algorithm—see, e.g., [30], consists of $O(\log n)$ phases. In the initial phase each node is an *MST fragment*, by which we mean a connected subgraph of the MST. In each subsequent phase, every MST fragment finds a lightest (i.e., minimum-weight) outgoing edge (LOE)—these edges are guaranteed to be in the MST by the cut property [32]. The MST fragments are merged via the LOEs to form larger MST fragments. The number of phases is $O(\log n)$, since the number of MST fragments gets at least halved in each phase. The message complexity is $O(m + n \log n)$ (which essentially matches the optimal message bound of $\tilde{\Omega}(m)$) and the time complexity is $O(n \log n)$. The time complexity is not optimal because much of the communication during a phase uses *only the MST fragment edges*. Since the diameter of an MST fragment can be as large as $\Omega(n)$ (and this can be significantly larger than the graph diameter D), the time complexity of the GHS algorithm is not optimal.

The Controlled-GHS algorithm alleviates this situation by controlling the growth of the diameter of the MST fragments during merging. At the end of Controlled-GHS, $\sqrt{n}$ fragments remain, each of which has diameter $O(\sqrt{n})$. These are called as *base fragments*. Controlled-GHS can be implemented using $\tilde{O}(m)$ messages in $\tilde{O}(\sqrt{n})$ rounds. (Note that Controlled-GHS as implemented in the time-optimal algorithm of [23] is not message-optimal—the messages exchanged can be $\tilde{O}(m + n^{3/2})$; however, a modified version can be implemented using $\tilde{O}(m)$ messages as explained in Section 3.1.)

2.2 Second Part: Merging the $\sqrt{n}$ Remaining Fragments

The second part of our algorithm, after the Controlled-GHS part, is different from the existing time-optimal MST algorithms. The existing time-optimal MST algorithms [8, 23], as well as the algorithm of [14], are not message-optimal since they use the Pipeline procedure of [14, 29]. The Pipeline procedure builds a breadth-first search (BFS) tree of the network, collects all the *inter-fragment* edges (these are edges between the $\sqrt{n}$ MST fragments) at the root of the BFS tree and then finds the MST locally. The Pipeline algorithm uses the cycle property of the MST [32] to eliminate those inter-fragment edges that cannot belong to the MST en route of their journey to the root. While the Pipeline procedure (due to the pipelining of the edges to the root) takes $O(\sqrt{n})$ time (since there are at most so many MST edges left to be discovered after the end of the first part), it is not message-optimal. The Pipeline procedure exchanges $O(m + n^{1.5})$ messages, since each node in the BFS tree can send up to $O(\sqrt{n})$ edges leading to $O(n^{1.5})$ messages overall (the BFS tree construction takes $O(m)$ messages).

Our algorithm uses a different strategy to achieve optimality in both time and messages. The main novelty of our algorithm (Algorithm 1) is how we merge the $\sqrt{n}$ base fragments which remain at the end of the Controlled-GHS procedure into one resulting fragment (the MST) in a time- and message-efficient way. Unlike previous time-optimal algorithms [8, 14, 23], we do not use the Pipeline procedure of [14, 29] which is not message-optimal (as explained above). Instead, we continue to merge fragments, a la Boruvka-style. Our algorithm uses two main ideas to implement the Boruvka-style merging efficiently. (Merging is achieved by renaming the IDs of the merged fragments to a common ID, i.e., all nodes in the combined fragment will have this common ID.) The first idea is a procedure to efficiently merge when D is small (i.e., $D = O(\sqrt{n})$) or when the number of fragments remaining is small (i.e., $O(n/D)$). The second idea is to use *sparse neighborhood covers* and efficient communication between fragments to merge fragments when D is large *and* the number of fragments is large. Accordingly, the second part of our algorithm can be divided into three phases, which are described next.

2.2.1 *Phase 1: When D is $O(\sqrt{n})$.*

Phase 1 can be treated as a special case of Phase 3 (as in Algorithm 1). However, we describe Phase 1 separately as it helps in the understanding of the other phases as well.

We construct a BFS tree on the entire network and do the merging process as explained below. Each base fragment finds its LOE by convergecasting *within* each of its fragments. This takes $O(\sqrt{n})$ time and $O(\sqrt{n})$ messages per base fragment, leading to $O(n)$ messages overall. The $O(\sqrt{n})$ LOE edges are sent by the leaders of the respective base fragments to the root by *upcasting* (see, e.g., [30]). This takes $O(D + \sqrt{n})$ time and $O(D\sqrt{n})$ messages, as each of the $\sqrt{n}$ edges has to traverse up to D edges on the way to the root. The root merges the fragments and sends the renamed fragment IDs to the respective leaders of the base fragments by *downcast* (which has the same time and message complexity as upcast [30]). The leaders of the base fragments broadcast the new ID to all other nodes in their respective fragments. This takes $O(\sqrt{n})$ messages per fragment and hence $O(n)$ messages overall. Thus one iteration of the merging can be done in $O(D + \sqrt{n})$ time and using $O(D\sqrt{n})$ messages. Since each iteration reduces the number of fragments by at least half, the number of iterations is $O(\log n)$. At the end of this iteration, several base fragments may share the same label. In subsequent iterations, each base fragment finds its LOE (i.e., the LOE between itself and the other base fragments which do not have the same label) by convergecasting within its own fragment and (the leader of the base fragment) sends the LOE to the root; thus $O(\sqrt{n})$ edges are sent to the root (one per base fragment), though there are a lesser number of combined fragments (with distinct labels). The root finds the overall LOE of the combined fragments and does the merging. This is still fine, since the time and message complexity per merging iteration is $O(D + \sqrt{n})$ time and $O(D\sqrt{n}) = O(n)$ messages respectively, which are as required.

2.2.2 Phase 2: When D and the Number of Fragments are Large. When D is large (say $n^{1/2+\varepsilon}$, for some $0 < \varepsilon \leqslant 1/2$) and the number of fragments is large (say, $\Theta(\sqrt{n})$) the previous approach of merging via the root of the global BFS tree does not work directly, since the message complexity would be $O(D\sqrt{n})$. The second idea addresses this issue: we merge in a manner that respects *locality*. That is, we merge fragments that are close by using a *local* leader (thus the LOE edges do not have to travel too far). The high-level idea is to use a *hierarchy of sparse neighborhood covers* to accomplish the merging.[7] A sparse neighborhood cover is a decomposition of a graph into a set of overlapping clusters that satisfy suitable properties (see Definition 3.4 in Section 3.4). The main intuitions behind using a cover are the following: (1) the clusters of the cover have relatively smaller diameter (compared to the strong diameter of the fragment and is always bounded by D) and this allows efficient communication for fragments contained within a cluster (i.e., the weak diameter of the fragment is bounded by the cluster diameter); (2) the clusters of a cover overlap only a little, i.e., each vertex belongs only to a few clusters; this allows essentially congestion-free (overhead is at most polylog(n) per vertex) communication and hence operations can be done efficiently in parallel across all the clusters of a cover. This phase continues till the number of fragments reduces to $O(n/D)$, when we switch to Phase 3. We next give more details on the merging process in Phase 2.

Communication-Efficient Paths. An important technical aspect in the merging process is constructing efficient communication paths between nearby fragments; the algorithm maintains and updates these efficient paths during the algorithm. Our algorithm requires fragments to be "communication-efficient", in the sense that there is an additional set of *short paths* between the fragment leader f and fragment members. Such a path might use "shortcuts" through vertices in $V(G)\setminus V(F)$ to reduce the distance. The following definition formalizes this idea.

Definition 2.1 (Communication-Efficient Fragment and Path). Let F be a fragment of G, and let $f \in F$ be a vertex designated as the *fragment leader* of F. We say that fragment F is *communication-efficient* if, for each vertex $v \in F$, there exists a path between v and f (possibly including vertices in $V(G) \setminus V(F)$) of length $O(\mathrm{diam}_G(F) + \sqrt{n})$, where $diam_G(F)$ is the weak diameter of F. Such a path is called *communication-efficient path for F*.

Section 3.2 defines the routing data structures that are used to maintain communication-efficient paths. Later, in Section 3.4, we describe the construction of the paths (and routing data structures) inductively. We show that, in each iteration, all fragments find their respective LOEs in time $\tilde{O}(\sqrt{n} + D)$ and using a total of $\tilde{O}(m)$ messages. While we cannot merge all fragments (along their LOEs), as this will create long chains, we use a procedure called ComputeMaximalMatching (Section 3.5) to merge fragments in a controlled manner. ComputeMaximalMatching finds a maximal matching in the fragment graph $\mathcal{F}_i$ induced by the LOE edges. The crucial part is using communication-efficient paths to communicate efficiently (both time and message-wise) between the fragment leader and the nodes in the fragment (while finding LOEs) as well as between fragment leaders of adjacent fragments (while merging as well as implementing ComputeMaximalMatching). The procedure FindLightest (see Section 3.3) describes the LOE finding process assuming communication-efficient fragments. The maintenance of such efficient fragments is shown recursively: the base fragments are efficient and after merging the resulting fragments are also efficient.

We use a hierarchy of sparse neighborhood covers to construct communication-efficient fragments (see Section 3.4). Each cover in the hierarchy consists of a collection of clusters of certain radius—the lowest cover in the hierarchy has clusters of radius $O(\sqrt{n})$ (large enough to contain at least one base fragment which have radius $O(\sqrt{n})$; subsequent covers in the hierarchy have clusters of geometrically increasing radii (the last cover in the hierarchy is simply the BFS tree of the entire graph). Initially, it is easy to construct communication-efficient paths in base fragments, since they have strong diameter $O(\sqrt{n})$ (cf. Section 3.2, Lemma 3.2). In subsequent iterations, when merging two adjacent fragments, the algorithm finds a cluster that is (just) large enough to contain both the fragments. Figure 1 gives an example of this process. The neighborhood property of the cluster allows the algorithm to construct communication-efficient paths between merged fragments (that might take shortcuts outside the fragments, and hence have small *weak diameter*) assuming that the fragments before merging are efficient. Note that it is important to make sure that the number of fragments in a cluster is not too large in relation to the radius of the cluster—otherwise the message complexity will be high (as in the Pipeline scenario). Hence, a *key invariant* that is maintained through all the iterations is that the *cluster depth times the number of fragments that are contained in the cluster of such depth is always bounded by* $\tilde{O}(n)$, and this helps in keeping the message complexity low. This invariant is maintained by making sure that the number of fragments per cluster *goes down* enough to compensate for the increase in cluster radius (Lemma 3.8 in Section 3.4). At the end of

[7] We use an efficient randomized cover construction algorithm due to Elkin [8]; this is the only randomization used in our algorithm. We note that neighborhood covers was used by Elkin [8] to improve the running time of the Pipeline procedure of his distributed MST algorithm; on the other hand, here we use it to *replace* the Pipeline part entirely in order to achieve message optimality as well.

Phase 3, the invariant guarantees that when the cluster radius is D, the number of fragments is $O(n/D)$.

2.2.3 Phase 3: When the Cluster Radius is D. When the cluster radius becomes D (i.e., the cover is just the BFS tree), we switch to Phase 3. The number of remaining fragments will be $O(n/D)$ (which is guaranteed at the end of Phase 2). Phase 3 uses a merging procedure very similar to that of Phase 1. In Phase 1, in every merging iteration, each base fragment finds their respective LOEs (i.e., LOEs between itself and the rest of the fragments) by convergecasting to their respective leaders; the leaders send at most $O(\sqrt{n})$ edges to the root by upcast. The root merges the fragments and sends out the merged information to the base fragment leaders by downcast. In Phase 3, we treat the $O(n/D)$ remaining fragments as the "base fragments" and repeat the above process. An important difference to Phase 1 is that the merging leaves the leaders of these base fragments intact: in the future iterations of Phase 3, each of these base fragments again tries to find an LOE using the procedure FindLightest, whereby only edges that have endpoints in fragments with distinct labels are considered as candidate for the LOE.

Note that the fragment leaders communicate with their respective nodes as well as the BFS root via the hierarchy of communication-efficient routing paths constructed in Phase 2; these incur only a polylogarithmic overhead. This takes $\tilde{O}(D+n/D)$ time (per merging iteration) since $O(n/D)$ LOE edges are sent to the root of the BFS tree via communication-efficient paths (in every merging iteration) and a message complexity of $\tilde{O}(D \cdot n/D) = \tilde{O}(n)$ (per merging iteration) since, in each iteration, each of the $O(n/D)$ edges takes $\tilde{O}(D)$ messages to reach the root. Since there are $O(\log n)$ iterations overall, we obtain the desired bounds.

3 DESCRIPTION AND ANALYSIS OF THE ALGORITHM

The algorithm operates on the *MST forest*, which is a partition of the vertices of a graph into a collection of trees $\{T_1, \ldots, T_\ell\}$ where every tree is a subgraph of the (final) MST. A *fragment* F_i is the subgraph induced by $V(T_i)$ in G. We say that an MST forest is an *(α, β)-MST forest* if it contains at most α fragments, each with a strong diameter[8] of at most β. Similarly, an MST forest is a *weak (α, β)-MST forest* if it contains at most α fragments each of (weak) diameter at most β.

We define the *fragment graph*, a structure that is used throughout the algorithm. The fragment graph $\mathcal{F}_i$ consists of vertices $\{F_1, \ldots, F_k\}$, where each F_j ($1 \leqslant j \leqslant k$) is a fragment at the start of iteration $i \geqslant 1$ of the algorithm. The edges of $\mathcal{F}_i$ are obtained by contracting the vertices of each $F_j \in V(\mathcal{F})$ to a single vertex in G and removing all resulting self-loops of G. We sometimes call the remaining edges *inter-fragment* edges. As our algorithm proceeds by finding lightest outgoing edges (LOEs) from each fragment, we operate partly on the *LOE graph $\mathcal{M}_i$ of iteration i*, which shares the same vertex set as $\mathcal{F}_i$, i.e., $\mathcal{M}_i \subseteq \mathcal{F}_i$, but where we remove all inter-fragment edges except for one (unique) LOE per fragment.

[8] Recall that the *strong diameter* $\mathrm{diam}_F(F)$ *of fragment F* refers to the longest shortest path (ignoring weights) between any two vertices in F that only passes through vertices in $V(F)$, whereas the *weak diameter* $\mathrm{diam}_G(F)$ allows the use of vertices that are in $V(G) \setminus V(F)$.

3.1 The Controlled-GHS Procedure

Our algorithm starts out by running the Controlled-GHS procedure introduced in [14] and subsequently refined in [23] and in [24].

Algorithm 2 Procedure Controlled-GHS: builds a $(\sqrt{n}, O(\sqrt{n}))$-MST forest in the network.

1: **procedure** Controlled-GHS:
2: $\mathcal{F} = \emptyset$ // initial MST forest
3: **for** $i = 0, \ldots, \lceil \log \sqrt{n} \rceil$ **do**
4: $\quad C$ = set of connectivity components of $\mathcal{F}$ (i.e., maximal trees).
5: $\quad$ Each $C \in C$ of diameter at most 2^i determines the LOE of C and adds it to a candidate set S.
6: $\quad$ Add a maximal matching $S_M \subseteq S$ in the graph (C, S) to $\mathcal{F}$.
7: $\quad$ If $C \in C$ of diameter at most 2^i has no incident edge in S_M, it adds the edge it selected into S to $\mathcal{F}$.

Controlled-GHS (Algorithm 2) is a modified variant of the original GHS algorithm, whose purpose is to produce a balanced outcome in terms of number and diameter of the resulting fragments (whereas the original GHS algorithm allows an uncontrolled growth of fragments). This is achieved by computing, in each phase, a maximal matching on the fragment forest, and merging fragments accordingly. Here we shall resort to the newest variant presented in [24], since it incurs a lower message complexity than the two preceding versions. Each phase essentially reduces the number of fragments by a factor of two, while not increasing the diameter of any fragment by more than a factor of two. Since the number of phases of Controlled-GHS is capped at $\lceil \log \sqrt{n} \rceil$,[9] it produces a $(\sqrt{n}, O(\sqrt{n}))$-MST forest. The fragments returned by the Controlled-GHS procedure are called the *base fragments*, and we denote their set by $\mathcal{F}_1$.

The following result about Controlled-GHS procedure follows from [24].

Lemma 3.1. *Algorithm 2 outputs a $(\sqrt{n}, O(\sqrt{n}))$-MST forest in $O(\sqrt{n} \log^* n)$ rounds and sends $O(m \log n + n \log^2 n)$ messages.*

Proof. The correctness of the algorithm is established trough Lemma 6.15 and Lemma 6.17 of [24]. By Corollary 6.16 of [24], the i-th iteration of the algorithm can be implemented in time $O(2^i \log^* n)$. Hence the time complexity of Controlled-GHS is

$$O\left(\sum_{i=0}^{\lceil \log \sqrt{n} \rceil} 2^i \log^* n\right) = O\left(\sqrt{n} \log^* n\right)$$

rounds.

We now analyze the message complexity of the algorithm. Consider any of the $\lceil \log \sqrt{n} \rceil$ iterations of the algorithm. The message complexity for finding the lightest outgoing edge for each fragment (Line 5) is $O(m)$. Then (Line 6) a maximal matching is built using the Cole-Vishkin symmetry-breaking algorithm. As argued in the proof of Corollary 6.16 of [24], in every iteration of this algorithm, only one message per fragment needs to be exchanged. Since the Cole-Vishkin algorithm terminates in $O(\log^* n)$

[9] Throughout, log denotes logarithm to the base 2.

Algorithm 1 A Time- and Message-Optimal Distributed MST Algorithm.

** **Part 1:**
1: Run Controlled-GHS procedure (Algorithm 2).
2: Let $\mathcal{F}_1$ be the base fragments obtained from Controlled-GHS.
** **Part 2:**
* **Start of Phase 1:**
3: **for** every fragment $F \in \mathcal{F}_1$ **do**
4: Construct a BFS tree T of F rooted at the fragment leader.
5: Every $u \in F$ sets $\mathrm{up}_u(F,1)$ to its BFS parent and $\mathrm{down}_u(F,1)$ to its BFS children.
6: Run the leader election algorithm of [22] to find a constant approximation of diameter D.
7: **if** $D = O(\sqrt{n})$ **then** set $\mathcal{F}' = \mathcal{F}_1$ and skip to Phase 3 (Line 32).
* **Start of Phase 2:**
8: **for** $i = 1, \dots, \lceil \log(D/\sqrt{n}) \rceil$ **do** // All nodes start iteration i at the same time
9: Construct cover $C_i = \text{ComputeCover}(2^i c_1 \sqrt{n})$ (c_1 is a suitably chosen constant).
10: Every node locally remembers its incident edges of the directed trees in C_i.
11: **for** each fragment $F_1 \in V(\mathcal{F}_i)$ **do**
12: Let $(u,v) = \mathsf{FindLightest}(F_1)$ where $u \in F_1$ and $v \in F_2$. // (u,v) is the LOE of F_1. See Section 3.3.
13: **if** $v \in F_2$ has an incoming lightest edge e_1 from F_1 **then**
14: v forwards e_1 to leader $f_2 \in F_2$ along its $((F_2,1),\dots,(F_2,i))$-upward-path.
15: $\mathsf{FindPath}(F_1,F_2)$. // Find a communication-efficient path for the merged fragment that connects leaders $f_1 \in F_1$ and $f_2 \in F_2$; this is needed for merging of fragments and also for iteration $i+1$. See Section 3.4.
// **Merging of fragments:**
16: **for** each fragment $F_1 \in V(\mathcal{F}_i)$ **do**
17: **if** F_1 has a weak diameter of $\leqslant 2^i c_1 \sqrt{n}$ **then** F_1 is marked active.
18: Let $\mathcal{M}_i \subseteq \mathcal{F}_i$ be the graph induced by the LOE edges whose vertices are the active fragments.
19: Let D be the edges output by running $\mathsf{ComputeMaximalMatching}$ on $\mathcal{M}_i$. // We simulate inter-fragment communication using the communication-efficient paths.
20: **for** each edge $(F,F') \in D$: Mark fragment pair for merging.
21: **for** each active fragment F not incident to an edge in D: Mark LOE of F for merging.
22: Orient all edges marked for merging from lower to higher fragment ID. A fragment leader whose fragment does not have an outgoing marked edge becomes *dominator*.
23: Every non-dominator fragment leader sends merge-request to its adjacent dominator.
24: **for** each dominating leader f **do**
25: **if** leader f received merge-requests from $F_1,\dots,F_\ell$ **then**
26: Node f is the leader of the merged fragment $F \cup F_1 \cup \dots \cup F_\ell$, where F is f's current fragment.
27: **for** $j = 1,\dots,\ell$ **do**
28: f sends $\mu = \langle \text{MergeWith}, F \rangle$ along its (F_j,i)-path to the leader f_j of F_j.
29: When f_j receives μ, it instructs all nodes $v \in F_j$ to update their fragment ID to F and update all entries in up and down previously indexed with F_j, to be indexed with F.
30: Let $\mathcal{F}_{i+1}$ be the fragment graph consisting of the merged fragments of $\mathcal{M}_i$ and the inter-fragment edges.
end of iteration i**.**
31: Let $\mathcal{F}' = \mathcal{F}_{\lceil \log(D/\sqrt{n}) \rceil + 1}$.
* **Start of Phase 3:** // Compute final MST given a fragment graph $\mathcal{F}'$.
32: **for** $\Theta(\log n)$ iterations **do**
33: Invoke $\mathsf{FindLightest}(F')$ for each fragment $F' \in \mathcal{F}'$ in parallel and then upcast the resulting LOE in a BFS tree of G to a root u.
34: Node u receives the LOEs from all fragments in $\mathcal{F}'$ and computes the merging locally. It then sends the merged labels to all the fragment leaders by downcast via the BFS tree.
35: Each fragment leader relays the new label (if it was changed) to all nodes in its own fragment via broadcast along the communication-efficient paths.
36: At the end of this iteration, several fragments in $\mathcal{F}'$ may share the same label. At the start of the next iteration, each fragment in $\mathcal{F}'$ individually invokes $\mathsf{FindLightest}$, whereby only edges that have endpoints in fragments with distinct labels are considered as candidates for the LOE.

iterations, the message complexity for building the maximal matching is $O(n \log^* n)$. Afterwards, adding selected edges into S to $\mathcal{F}$ (Line 7) can be done with an additional $O(n \log n)$ message complexity. The message complexity of algorithm Controlled-GHS is therefore $O(m \log n + n \log^2 n)$. □

3.2 Routing Data Structures for Communication-Efficient Paths

For achieving our complexity bounds, our algorithm maintains efficient fragments in each iteration. To this end, nodes locally maintain routing tables. In more detail, every node $u \in G$ has 2 two-dimensional arrays up_u and down_u (called *routing* arrays), which are indexed by a (fragment ID,level)-pair, where level stands for the iteration number, i.e., the for loop variable i in Algorithm 1. Array up_u maps to one of the port numbers in $\{1, \ldots, d_u\}$, where d_u is the degree of u. In contrast, array down_u maps to a set of port numbers. Intuitively speaking, $\mathrm{up}_u(F, i)$ refers to u's parent on a path p towards the leader of F where i refers to the iteration in which this path was constructed. Similarly, we can think of $\mathrm{down}_u(F, i)$ as the set of u's children in all communication efficient paths originating at the leader of F and going through u and we use down_u to disseminate information from the leader to the fragment members. Oversimplifying, we can envision up_u and down_u as a way to keep track of the parent-child relations in a tree that is rooted at the fragment leader. (Note that level is an integer in the range $[1, \Theta(\log \sqrt{n})]$ that corresponds to the iteration number of the main loop in which this entry was added; see Lines 8-30 of Algorithm 1.) For a fixed fragment F and some value level $= i$, we will show that the up and down arrays induce directed chains of incident edges.

Depending on whether we use array up or array down to route along a chain of edges, we call the chain an (F, i)-*upward-path* or an (F, i)-*downward-path*. When we just want to emphasize the existence of a path between a fragment node v and its leader f, we simply say that there is a *communication-efficient* (F, i)-*path between* v *and* f and we omit "(F, i)" when it is not relevant. We define the nodes specified by $\mathrm{down}_u(F, i)$ to be the (F, i)-*children of* u and the node connected to port $\mathrm{up}_u(F, i)$ to be the (F, i)-*parent of* u. So far, we have only presented the definitions of our routing structures. We will explain their construction in more detail in Section 3.4.

We now describe the routing of messages in more detail: Suppose that $u \in F$ generates a message μ that it wants to send to the leader of F. Then, u encapsulates μ together with F's ID, the value level $= 1$, and an indicator "up" in a message and sends it to its neighbor on port $\mathrm{up}_u(F, 1)$; for simplicity, we use F to denote both, the fragment and its ID. When node v receives μ with values F and level $= 1$, it looks up $\mathrm{up}_v(F, 1)$ and, if $\mathrm{up}_v(F, 1) = a$ for some integer a, then v forwards the (encapsulated) message along the specified port.[10] This means that μ is relayed to the root w of the $(F, 1)$-upward-path. For node w, the value of $\mathrm{up}_w(F, 1)$ is undefined and so w attempts to lookup $\mathrm{up}_w(F, 2)$ and then forwards μ along the $(F, 2)$-upward-path and so forth. In a similar manner, μ is forwarded along the path segments $p_1 \ldots p_i$ where p_j is the (F, j)-upward-path $(1 \leqslant j \leqslant i)$ in the i-th iteration of the algorithm's main-loop. We will show that the root of the (F, i)-upward-path coincides with the fragment leader at the start of the i-th iteration.

On the other hand, when the iteration leader u in the i-th iteration wants to disseminate a message μ to the fragment members, it sends μ to every port in the set $\mathrm{down}_u(F, i)$. Similarly to above, this message is relayed to the root v of each (F, i)-downward-path, for which the entry $\mathrm{down}_v(F, i)$ is undefined. When $i > 1$, node v then forwards μ to the ports in $\mathrm{down}_v(F, i-1)$ and μ traverses the path segments $q_i \ldots q_1$ where q_j is the (F, j)-downward-path. For convenience we call the concatenation of $q_i \ldots q_1$ a $((F, i), \ldots, (F, 1))$-*downward path* (or simply $((F, i), \ldots, (F, 1))$-*path*), and define a $((F, 1), \ldots, (F, i))$-*upward path* similarly.

We are now ready to describe the individual components of our algorithm in more detail. To simplify the presentation, we will discuss the details of Algorithm 1 inductively.

We assume that every node $u \in F \in \mathcal{F}_1$ knows its parent and children in a BFS tree rooted at the fragment leader $f \in F$. (BFS trees for spanning each respective fragment can easily be constructed in $O(\sqrt{n})$ time and using a total of $O(m)$ messages—this is because the fragments in $\mathcal{F}_1$ are disjoint and have strong diameter $O(\sqrt{n})$.) Thus, node u initializes its routing arrays by pointing $\mathrm{up}_u(F, 1)$ to its BFS parent and by setting $\mathrm{down}_u(F, 1)$ to the port values connecting its BFS children.

Lemma 3.2. *At the start of the first iteration, for any fragment F and every $u \in F$, there is an $(F, 1)$-path between F's fragment leader and u with a path length of $O(\sqrt{n})$.*

Proof. From the initialization of the routing tables up and down it is immediate that we reach the leader when starting at a node $u \in F$ and moving along the $(F, 1)$-*upward-path*. Similarly, starting at the leader and moving along the $(F, 1)$-*downward-path*, allows us to reach any fragment member. The bound on the path length follows from the strong diameter bound of the base fragments, i.e., $O(\sqrt{n})$ (see Lemma 3.1). □

3.3 Finding the Lightest Outgoing Edges (LOEs): Procedure FindLightest

We now describe Procedure FindLightest(F), which enables the fragment leader f to obtain the lightest outgoing edge, i.e., the lightest edge that has exactly 1 endpoint in F. Consider iteration $i \geqslant 1$. Initially, FindLightest(F) requires all fragment nodes to exchange their fragment IDs with their neighbors to ensure that every node v knows its set of incident outgoing edges E_v. If a node v is a leaf in the BFS trees of its base fragment, i.e., it does not have any $(F, 1)$-children, it starts by sending the lightest edge in E_v along the $((F, 1), \ldots, (F, i))$-upward-path. In general, a node u on an (F, j)-upward-path $(j \geqslant 1)$ waits to receive the lightest-edge messages from all its (F, j)-children (or its $(F, j-1)$-children if any), and then forwards the lightest outgoing edge that it has seen to its parent in the $((F, j), \ldots, (F, i))$-upward-path.

The following lemma proves some useful properties of FindLightest. Note that we do not yet claim any bound on the message complexity at this point, as this requires us to inductively argue on the structure of the fragments, which requires properties that we introduce in the subsequent sections. Hence we postpone the message complexity analysis to Lemma 3.12.

Lemma 3.3 (Efficient LOE Computation). *Suppose that every fragment in $F \in \mathcal{F}_i$ is communication-efficient at the start of iteration $i \geqslant 1$. Then, the fragment leader of F obtains the lightest outgoing*

[10] Node v is free to perform additional computations on the received messages as described by our algorithms, e.g., v might aggregate simultaneously received messages in some form. Here we only focus on the forwarding mechanism.

edge by executing Procedure FindLightest(F) in $O(\sqrt{n} + diam_G(F))$ rounds.

Proof. To accurately bound the congestion, we must consider the simultaneous invocations of FindLightest for each fragment in $\mathcal{F}_i$. Since, by assumption, every fragment is communication-efficient, every fragment node u can relay its lightest outgoing edge information to the fragment leader along a path p of length $O(\mathrm{diam}_G(F) + \sqrt{n})$. Note that p is precisely the $((F,1),\ldots,(F,i))$-upward path to the leader starting at u. To bound the congestion, we observe that the $(F,1)$-upward subpath of p is confined to nodes in F_u where F_u is the base fragment that u was part of after executing Controlled-GHS. As all base fragments are disjoint and lightest edge messages are aggregated within the same base fragment, the base fragment leader (who might *not* be the leader of the current fragment F) accumulates this information from nodes in F_u within $O(\sqrt{n})$ rounds (cf. Lemma 3.2). After having traversed the $(F,1)$-upward path (i.e., the first segment of p) of each base fragment, the number of distinct messages carrying lightest edge information is reduced to $O(\sqrt{n})$ in total. Hence, when forwarding any such message along a subsequent segment of p, i.e., an (F_j)-upward path for $j > 1$, the maximum congestion at any node can be $O(\sqrt{n})$. Using a standard upcast (see, e.g., [30]) and the fact that the length of path p is $O(\mathrm{diam}_G(F) + \sqrt{n})$, it follows that the fragment leader receives all messages in $O(\mathrm{diam}_G(F) + \sqrt{n})$ rounds, as required. □

3.4 Finding Communication-Efficient Paths: Procedure FindPath

After executing FindLightest(F_0), the leader f_0 of F_0 has obtained the identity of the lightest outgoing edge $e = (u,v)$ where v is in some distinct fragment F_1. Before invoking our next building block, Procedure FindPath(F_0, F_1), we need to ensure that both leaders are aware of e and hence we instruct the node v to forward e along its $((F_1,1),\ldots,(F_1,i))$-upward-path to its leader f_1 (see Lines 13-14 of Algorithm 1).

We now describe FindPath(F_0, F_1) in detail. The main goal is to compute a communication-efficient path between leaders f_0 and f_1 that can be used to route messages between nodes in this fragment. In Section 3.5, we will see how to leverage these communication-efficient paths to efficiently merge fragments.

A crucial building block for finding an efficient path are the *sparse neighborhood covers*, which we precompute initially (see Line 9 of Algorithm 1), and which we recall here. (Note that the cover definition assumes the underlying unweighted graph, i.e., all distances are just the hop distances.)

Definition 3.4. A *sparse* (κ, W)*-neighborhood cover* of a graph is a collection C of trees, each called a *cluster*, with the following properties.

(1) *(Depth property)* For each tree $\tau \in C$, depth$(\tau) = O(W \cdot \kappa)$.
(2) *(Sparsity property)* Each vertex v of the graph appears in $\tilde{O}(\kappa \cdot n^{1/\kappa})$ different trees $\tau \in C$.
(3) *(Neighborhood property)* For each vertex v of the graph there exists a tree $\tau \in C$ that contains the entire W-neighborhood of vertex v.

Sparse neighborhood covers were introduced in [4], and were found very useful for various applications. We will use an efficient distributed (randomized) cover construction due to Elkin [8], which we recall here.[11]

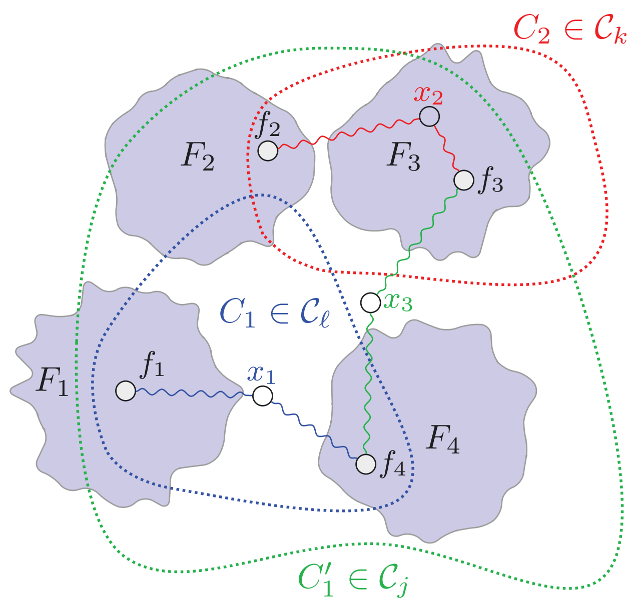

Figure 1: Fragments $F_1,\ldots,F_4$. In the first iteration, F_1, F_4 and F_2, F_3 form adjacent fragment pairs that communicate along communication-efficient paths. F_1 and F_4 execute FindPath and send probe messages along clusters of covers $C_1,\ldots,C_\ell$ and finally succeed to find a communication-efficient path in a cluster $C_1 \in \mathcal{C}_\ell$, which goes through the cluster leader $x_1 \in C_1$. Similarly F_2 and F_3 obtain a communication-efficient path in cluster $C_2 \in C_k$, after sending probe messages in clusters of covers $C_1,\ldots,C_k$. In the next iteration, the merged fragments $F_1 \cup F_4$ and $F_2 \cup F_3$ are (respectively) adjacent and proceed to construct a communication-efficient path in cluster $C_1' \in C_j$, after probing covers $C_1,\ldots,C_j$.

Theorem 3.5 ([8, Theorem A.8]). *There exists a distributed randomized (Las Vegas) algorithm (which we call ComputeCover) that constructs a (κ, W)-neighborhood cover in time $O(\kappa^2 \cdot n^{1/\kappa} \cdot \log n \cdot W)$ and using $O(m \cdot \kappa \cdot n^{1/\kappa} \cdot \log n)$ messages (both bounds hold with high probability) in the CONGEST model.*

In our MST algorithm, we shall invoke Elkin's ComputeCover procedure with $\kappa = \log n$, and write ComputeCover(W), where W is the neighborhood parameter.

We are now ready to describe the communication-efficient paths construction. As we want to keep the overall message complexity low, we start at the smallest cover construction C_1 and carefully probe for a cluster (tree) in C_1 that induces a communication-efficient path between f_0 and f_1. Recall that every node locally keeps track of its incident cluster edges for each of the precomputed covers but we need to keep in mind that these structures are

[11] Although the algorithm as described in [8] is Monte Carlo, it can be easily converted to Las Vegas.

independent of the up and down arrays. We instruct both leaders f_0 and f_1 to send a copy of their probe message to each of their C_1-parents. The parent nodes forward u's probe message along their cluster tree to the root of their respective cluster tree. Depending on whether a root receives the probe message in a timely fashion, we consider two cases:

Case 1: If there exists a $C_w \in C_1$ such that $f_0, f_1 \in C_w$, then the probe message of both leaders reaches the root $w \in C_w$ within $2^1 c_1 \sqrt{n} + O(\sqrt{n} \log^2 n)$ rounds, where the first term is $\mathrm{depth}(C_1)$ and the second term is to account for congestion caused by simultaneous probe messages from the other fragment leaders (cf. Lemma 3.7). Suppose that w receives the probe message from f_0 on path p_0 and f_1's probe message on path p_1 within $2^1 c_1 \sqrt{n} + O(\sqrt{n} \log^2 n)$ rounds. Then, w replies by sending a "success" message back to f_0 and f_1 by reversing p_0 and p_1 to inform the leaders that they have found a communication-efficient path.

Note that it is possible for f_0 to receive multiple "success" reply messages. However, since a cluster root only sends a success message if it receives probe messages from both leaders, f_0 and f_1 receive exactly the same set M of success messages. Thus they both pick the same success message sent by the cluster root node with the largest ID in M (without loss of generality, assume that it is w) to identify the communication-efficient path and discard the other messages in M.

Suppose that f_0 received the message from w along a path p_0 in cluster tree C_w. Then, f_0 sends a message along p_0 and instructs every node v in p_0 to set $\mathrm{up}_v(F_1, i)$ to the port of its successor (towards the root w) in p_0 and points $\mathrm{up}_v(F_0, i)$ to its predecessor in p_0. When a node v updates its $\mathrm{up}_v(F_1, i)$ array to some port a, it contacts the adjacent node v' connected at this port who in turn updates $\mathrm{down}_{v'}(F_1, i)$ to point to v. Similarly, leader f_1 and all nodes on the path p_1 proceeds updating their respective up and down entries with the information provided by p_1 towards w. Then, f_0 contacts its successor in p_0 to update its routing information whereas f_1 sends a similar request to its successor in p_1. After these requests reach the cluster root w, the concatenated path $p_0\, p_1$ is a communication-efficient path between leaders f_0 and f_1.

Case 2: On the other hand, if there is no appropriate cluster in C_1 that covers both leader nodes, then at least one of the two probe messages will arrive untimely at every cluster root and the leaders do not receive any success messages. Then, f_0 and f_1 rerun the probing process by sending a probe message along their incident C_2 cluster edges and so forth. Note that all fragment leaders synchronize before executing the probing process. Eventually, f_0 and f_1 obtain a value k, where C_k is the cover having the smallest depth such that f_0 and f_1 are covered by some cluster in C_k (but not by any cluster in C_{k-1}) and we can apply Case 1.

Figure 1 gives an example for the construction of communication-efficient paths.

Lemma 3.6. *The number of probe messages that are generated by distinct fragment leaders and that are in transit simultaneously during an iteration of FindPath is $O(\sqrt{n} \log^2 n)$ w.h.p.*

Proof. Since, by Lemma 3.1, there are $O(\sqrt{n})$ base fragments, the total number of leaders at any point that are sending probe messages simultaneously is $O(\sqrt{n})$. Note that, when exploring the communication efficient paths of a cover C_j, a leader needs to send a copy of its probe message to its parent in each of its $O(\log^2 n)$ clusters of C_j that it is contained in. ◻

Lemma 3.7. *After the execution of FindPath(F_0, F_1), there exists a communication-efficient path between leader f_0 and leader f_1 of length at most $2^k c_1 \sqrt{n}$, where k is the smallest integer such that there exists a cluster tree $C \in C_k$ such that $f_0, f_1 \in C$. FindPath(F_0, F_1) requires $O(2^k \sqrt{n} \log^2 n)$ messages and terminates in*

$$O\Big(\sqrt{n} \log^2 n + \min\{2^k \sqrt{n}, diam(G)\}\Big)$$

rounds with high probability.

Proof. By description of FindPath, leaders f_0 and f_1 both start sending a probe message along their incident C_j-edges towards the respective cluster roots, for $j = 1, \ldots, \lceil \log \sqrt{n} \rceil$. First, note that f_0 and f_1 will not establish an efficient communication path for a cluster C' in some C_j $(j < k)$, since, by definition, f_0 and f_1 are not both in C' and hence one of the probe messages will not reach the root of C'. Let w be the root of C.

We now argue the message complexity bound. Apart from the probe messages sent to discover the communication-efficient path in a cluster of cover C_k, we also need to account for the probe messages sent along cluster edges of covers $C_1, \ldots, C_{k-1}$, thus generating at most

$$\begin{aligned}\sum_{j=1}^{k} O(\mathrm{depth}(C_j) \log^2 n) &= \sum_{j=1}^{k} O(2^j \sqrt{n} \log^2 n)\\ &\leqslant 2^{k+1} O(\sqrt{n} \log^2 n)\\ &= O(\mathrm{depth}(C_k) \log^2 n)\end{aligned}$$

messages, as required.

Since f_0 and f_1 can communicate efficiently via a path p leading through a cluster of cover C_k, it follows that the length of p is $\leqslant 2\mathrm{depth}(C_k)$. Applying Lemma 3.6 to take into account the additional congestion caused by simultaneous probe messages, yields a time complexity of $O(\mathrm{depth}(C_k) + \sqrt{n} \log^2 n)$. ◻

Lemma 3.8. *At the start of each iteration i, the fragment graph $\mathcal{F}_i$ induces a weak $(O(\sqrt{n}/2^i), O(2^i \sqrt{n}))$-MST forest in G.*

Proof. We adapt the proof of Lemmas 6.15 and 6.17 of [24]. For the case $i = 0$, the claim follows directly from Lemma 3.1. We now focus on the inductive step $i > 0$.

Suppose that $\mathcal{F}_i$ is a weak $(\sqrt{n}/2^i, 6 \cdot 2^i c_1 \sqrt{n})$-MST forest. We first argue that every new fragment in $\mathcal{F}_{i+1}$ must have a weak diameter of at most $6 \cdot 2^{i+1} c_1 \sqrt{n}$.

Consider the subgraph M of $\mathcal{F}_i$ induced by the edges marked for merging. By Lines 20-21 of Algorithm 1, each component of M can contain at most one marked edge that was in the output of ComputeMaximalMatching. Thus, analogously to Lemma 6.15 in [24], it follows that each component in M contains at most one fragment of weak diameter $> 2^i c_1 \sqrt{n}$, since only fragments of weak diameter at most $2^i c_1 \sqrt{n}$ become active and participate in the matching. Note that the maximality of the matching implies that each component of M has diameter (in the fragment subgraph M) at most 3. Moreover, all except at most 1 fragment of such a component must have a weak diameter of at most $2^i c_1 \sqrt{n}$ since a fragment of a larger weak diameter does not select any edges for merging in this iteration. It follows by the inductive hypothesis that the merged component has a weak diameter of at most $6 \cdot 2^i c_1 \sqrt{n} + 3 \cdot 2^i c_1 \sqrt{n} \leqslant 6 \cdot 2^{i+1} c_1 \sqrt{n}$.

We now argue that each fragment contains at least $2^i c_2 \sqrt{n}$ nodes at the start of iteration $i > 0$, assuming that it is true for all $j = 0, \ldots, i-1$. To this end, consider the merging of fragments in iteration $i-1$. If a fragment $F \in \mathcal{F}_i$ contains less than $2^i c_2 \sqrt{n}$ nodes it must have a weak diameter of at most $2^i c_2 \sqrt{n}$ and hence marks itself as active in Line 17. By the description of the merging process, F is guaranteed to merge with at least one other fragment F'. By the inductive hypothesis, both F and F' consist of at least $2^{i-1} c_2 \sqrt{n}$ nodes and hence the merged fragment must have at least $2^i c_2 \sqrt{n}$ nodes, as required. □

Lemma 3.9. *Consider an iteration i and suppose that FindPath is invoked simultaneously for each lightest outgoing edge. Then, the total message complexity of all invocations is $O(n \log^3 n)$ and the time complexity is $O(\sqrt{n} + diam(G))$ with high probability.*

Proof. From Lemma 3.8, we know that every fragment in $\mathcal{F}_i$ has weak diameter of $O(2^i \sqrt{n})$. Thus, every pair of adjacent fragments $F_0, F_1 \in \mathcal{F}_i$ is covered by some cluster in cover C_{i+1}. In this case, Lemma 3.7 tells us that a single invocation of FindPath requires $O(2^{i+1} \sqrt{n} \log^2 n)$ messages. Lemma 3.8 tells us that there are $O(\sqrt{n}/2^i)$ fragments in $\mathcal{F}_i$ (and thus also $O(\sqrt{n}/2^i)$ LOEs). Hence the total number of messages incurred by all pairs of fragments connected by an LOE is

$$O(2^{i+1} \sqrt{n} \log^2 n) \cdot O(\sqrt{n}/2^i) = O(n \log^2 n).$$

Summing up over all i, we obtain the claimed bound on the message complexity.

Finally we observe that Lemma 3.7 already takes into account the congestion caused by simultaneous invocations, which yields the bound on the time complexity. □

To summarize, Procedure FindPath enables leaders of adjacent fragments to communicate with each other by sending messages along the communication-efficient paths given by the routing tables up and down.

3.5 Merging Fragments

We will avoid long chains of merged fragments by using procedure ComputeMaximalMatching. Procedure ComputeMaximalMatching in [24] outputs a maximal matching on a fragment forest, where fragments in $\mathcal{F}_i$ are treated as super-vertices of a graph connected by inter-fragment edges. Procedure ComputeMaximalMatching simulates the Cole-Vishkin symmetry-breaking distributed algorithm, which terminates in $O(\log^* n)$ iterations [24, Theorem 1.7]. We next show how to do the simulation efficiently in the fragment graph.

Procedure FindPath enables communication via communication-efficient paths between any two adjacent fragment leaders in $\mathcal{M}_i$. This allows us to simulate ComputeMaximalMatching on the network induced by $\mathcal{M}_i$, where the leaders in $\mathcal{M}_i$ perform the computation required by ComputeMaximalMatching. The following lemma follows directly from Lemma 3.9.

Lemma 3.10. *Suppose that every fragment in $\mathcal{F}_i$ is efficient and let $\mathcal{M}_i \subset \mathcal{F}_i$ be the lightest outgoing edge graph obtained by running FindPath. Then, we can simulate* ComputeMaximalMatching *on the network defined by $\mathcal{M}_i$, requiring $\tilde{O}(diam(G) + \sqrt{n})$ rounds and $\tilde{O}(n)$ messages.*

Every non-dominator fragment F'_1 sends a ⟨MergeReq⟩ message to the leader f'_1 of an arbitrarily chosen adjacent dominator fragment F. The dominator fragment processes all merge-requests in parallel and replies by sending a ⟨MergeWith, F⟩ message to the leader f' of each fragment F' from which it received ⟨MergeReq⟩; in turn, f' forwards this request along the $((F', i), \ldots, (F', 1))$-downward path to every node in F'. Upon receiving a ⟨MergeWith, F⟩ message, node $u' \in F'$ updates its fragment ID to F, and also updates its routing table by setting $\mathrm{up}_{u'}(F, \ell) = \mathrm{up}_{u'}(F', \ell)$ and $\mathrm{down}_{u'}(F, \ell) = \mathrm{down}_{u'}(F', \ell)$, for every value of ℓ. Note that the leader of the dominator fragment becomes the new leader of the merged fragment.

Lemma 3.11. *Consider iteration i. If, for every $j \leqslant i$, every fragment in $\mathcal{F}_j$ is communication-efficient, then the following hold.*

1. *With high probability, the message complexity for merging fragments in iteration i is $\tilde{O}(m)$ and the process completes within $\tilde{O}(diam(G) + \sqrt{n})$ rounds.*
2. *Every fragment in $\mathcal{F}_{i+1}$ is communication-efficient.*

Proof Sketch. To show (1), we argue recursively starting at iteration i, as follows: note that forwarding the ⟨MergeWith⟩ and ⟨MergeReq⟩ messages requires communicating between neighboring fragments and thus by Lemma 3.10 we require $O(\mathrm{diam}(G) + \sqrt{n})$ rounds and $O(n \log^2 n)$ messages. Consider an adjacent pair of fragments F_0 and F_1 and suppose that F_0 merges with the dominator fragment F_1. Since we eventually need to broadcast the new fragment ID to every node $u \in F_0$ we need to ensure that the routing tables $\mathrm{up}_u(F_1, \cdot)$ and $\mathrm{down}_u(F_1, \cdot)$ are updated correctly to route messages towards the new leader $f_1 \in F_1$ (and vice versa from f_1 to all nodes in F_1), when we compute the lightest outgoing edge of the merged fragment $F_0 \cup F_1$ in subsequent iterations. If $i > 1$, then F_0 might be composed of merged fragments $F'_0 \cup \cdots \cup F'_\ell$ that merged in previous iterations; without loss of generality, suppose that this iteration is $i-1$. By assumption, $\mathcal{F}_{i-1}$ consisted of efficient fragments. As nodes do not remove routing information from up and down, the leader f_0 can use the communication-efficient paths obtained by invoking FindPath in iteration $i-1$ to forward the new fragment ID to the leaders of the $F'_0, \ldots, F'_\ell$, which we call the $(i-1)$-*iteration fragments.* Applying Lemma 3.10 to $\mathcal{M}_{i-1}$ reveals that we can use the paths obtained by invoking FindPath in iteration $i-1$ to relay the new fragment ID to $(i-1)$-iteration fragments while incurring only $O(\mathrm{diam}(G) + \sqrt{n})$ rounds and $O(n \log^2 n)$ messages in total. Recursively applying this argument until iteration 1, allows us to reason that $O((\mathrm{diam}(G) + \sqrt{n}) \log n)$ rounds and $O(n \log^3 n)$ messages are sufficient to relay all new fragment IDs to the base fragment leaders. At this point, every base fragment leader uses the BFS tree of the base fragments to broadcast this information to the base fragment nodes, requiring $O(\sqrt{n})$ rounds and $O(m)$ messages.

To show (2), we observe that $\mathcal{F}_i$ consists of communication-efficient fragments, and hence every fragment node $u \in F_j$ of a newly merged fragment $F = F_1 \cup \cdots \cup F_\ell$ ($\ell \geqslant j$) can already communicate efficiently with the leader f_j in its subfragment F_j, which has now become part of F. Moreover, the paths obtained by

FindPath ensure that f_j can communicate efficiently with leader $f \in F$ and hence it follows transitively that u has a communication-efficient path to f, as required. □

The analysis of the message complexity of merging fragments allows us to obtain a bound on the number of messages required for computing a lightest outgoing edge in each fragment.

Lemma 3.12. *The message complexity of all parallel invocations of FindLightest is $\tilde{O}(m)$ in total w.h.p.*

Proof Sketch. In the first step of FindLightest, each node exchanges messages with its neighbors requiring $\Theta(m)$ messages. Let $F = F_1 \cup \cdots \cup F_\ell$ where $F_1, \ldots, F_\ell$ are base fragments and consider some $u \in F_1$. As argued above, u relays its LOE information along the $((F,1), \ldots, (F_i))$-upward path to the fragment leader and the segment formed by the $(F,1)$-upward path ends at the base fragment leader of F_1, which are exactly the BFS trees yielded by Controlled-GHS. A crucial observation is that u only sends its LOE information to its parent in the path, *after* receiving the LOE messages from all its children (see Section 3.3). This ensures that each node sends exactly one message and hence we obtain a bound of $\sum_{j=1}^{\ell} O(|V(F_j)|) = O(|V(F)|)$ on the number of messages sent in the $(F,1)$-upward path of the nodes in F. This is subsumed in the message complexity of exchanging messages with neighbors in the first step, which is $O(m)$.

At this point, each base fragment leader f_j of F_j $(j = 1, \ldots, \ell)$ holds exactly one (aggregated) lightest outgoing edge information message μ_j, which needs to be relayed to the fragment leader f of F along the respective $((F,2), \ldots, (F,i))$-upward path of $O(\mathrm{diam}_G(F))$ hops (see Definition 2.1).

By reversing the argument used for proving part (2) of Lemma 3.11, we can inductively apply Lemma 3.10 to finally obtain a bound of $O(n \log^3 n)$ messages per iteration and thus the total message complexity is $O(m + n \log^3 n) = \tilde{O}(m)$. □

Lemma 3.13. *Phase 3 of the algorithm requires $\tilde{O}(m)$ messages and $\tilde{O}(D + \sqrt{n})$ time and ensures that all fragments have the same label (i.e., are merged).*

Proof. Note that our algorithm either executes Phase 3 directly after Phase 1 (thus skipping Phase 2) or after executing Phase 2. First we argue (for both cases) that all fragments have the same fragment ID after the $\Theta(\log n)$ iterations in Phase 3. To see that the number of fragment labels is at least halved in each iteration, note that, when executing FindLightest, all nodes exchange their fragment IDs with their neighbors (requiring $O(m)$ messages) and then only choose candidate LOE edges that have their endpoint in fragments with distinct IDs. This ensures that every fragment pairs up with another fragment and hence one of the two distinct IDs will be removed; note that long "chains" of fragments connected by LOE edges are possible and result in an even faster reduction of distinct labels—all fragments in the chain adapt the root fragment ID (cf. Phase 3 in the pseudo code). Thus, after the last iteration of Phase 3, all fragments carry the same fragment ID and no more LOE edges are required as all fragments are considered to be merged.

Now we consider the message and time complexity of Phase 3. According to Lemma 3.3, the time complexity of finding the LOEs is $O(D + \sqrt{n})$, and according to Lemma 3.12 $\tilde{O}(m)$ messages are required to find the LOEs. This is true independently of whether we called Phase 3 directly after Phase 1 or after Phase 2.

Now, consider the case where we execute Phase 3 directly after Phase 1 (thus skipping Phase 2), i.e., $D = O(\sqrt{n})$. Here, FindLightest results in each node locally determining the incident LOE and then aggregating the LOE to the base fragment leader. In addition to the base fragment BFS trees, we also construct a global BFS tree T, which, has $O(\sqrt{n})$ diameter by assumption. The base fragment leaders then forward their respective LOE along towards the root u of T. Since we have $O(\sqrt{n})$ distinct base fragments, there are at most $O(\sqrt{n})$ LOE edges sent upward in T, thus resulting in an additional message complexity of $O(D\sqrt{n}) = O(n)$. Taking into account that it takes $O(\sqrt{n})$ rounds for the base fragment leaders to determine the LOE of their fragment, the time complexity amounts to $O(D + \sqrt{n})$.

We now argue the message and time complexity for the case where we execute Phase 3 after Phase 2. Here, we start out with $O(n/D)$ distinct fragments each having their own fragment ID and a global BFS tree T of depth $O(D)$. Since each fragment finds 1 LOE which is first aggregated at the fragment leader and then forwarded along T to the global BFS root, this requires $O(\frac{n}{D}D) = O(n)$ messages in total and $O(D + n/D) = O(D)$ rounds, since $D = \Omega(\sqrt{n})$ by assumption, completing the proof. □

Combining the complexity bounds from the previous lemmas we obtain the following theorem.

Theorem 3.14. *Consider a synchronous network (in the KT0 model) of n nodes, m edges, and diameter D, and suppose that at most $O(\log n)$ bits can be transmitted over each link in every round. Algorithm 1 computes an MST and, with high probability, runs in $\tilde{O}(D + \sqrt{n})$ rounds and exchanges $\tilde{O}(m)$ messages.*

4 A SIMULTANEOUSLY TIGHT LOWER BOUND

As mentioned in Section 1.2, the existing graph construction of [6, 9] that shows the time lower bound of $\tilde{\Omega}(D + \sqrt{n})$ rounds does not simultaneously yield the message lower bound of $\tilde{\Omega}(m)$; similarly the existing lower bound graph construction of [22] that shows the message lower bound of $\tilde{\Omega}(m)$ does not simultaneously yield the time lower bound of $\tilde{\Omega}(D + \sqrt{n})$ (note that these lower bound constructions apply to randomized algorithms). Previously, [6] presented a sparse graph of $O(n)$ edges to obtain the $\tilde{\Omega}(D + \sqrt{n})$ time bound for almost all choices of D, while [22] showed that $\Omega(m)$ messages are required to solve broadcast and hence also for constructing a (minimum) spanning tree.[12]

The following result presents a "universal lower bound" for MST in the sense that it shows that for essentially any n, m, and D, there exists a class of graphs of n nodes, m edges, and a diameter of D, for which every randomized MST algorithm takes $\tilde{\Omega}(D + \sqrt{n})$ rounds *and* $\Omega(m)$ messages to succeed with constant probability. Our proof combines two lower bound techniques: hardness of distributed symmetry breaking, used to show the lower bound on message complexity [22], and communication complexity, used to show the

[12] Any algorithm that constructs an spanning tree using $O(f(n))$ messages can be used to elect a leader using $O(f(n) + n)$ messages in total, by first constructing a spanning tree and then executing any broadcast algorithm restricting its communication to the $O(n)$ spanning tree edges.

lower bound on time complexity [6]. The full proof is deferred to the full version of the paper.

Theorem 4.1. *There is a class of graphs of n nodes, m edges (for $n \leqslant m \leqslant \binom{n}{2}$), and diameter $D = \Omega(\log n)$ for which every ε-error distributed MST algorithm requires $\Omega(m)$ messages and $\tilde{\Omega}(D + \sqrt{n})$ time in expectation in the KT0 model, for any sufficiently small constant $\varepsilon > 0$. This holds even if nodes have unique IDs and have knowledge of the network size n.*

5 CONCLUSION

We presented a distributed algorithm for the fundamental minimum spanning tree problem which is simultaneously time- and message-optimal (up to polylog(n) factors). This algorithm is randomized: an intriguing open question is whether randomization is necessary to simultaneously achieve time and message optimality.

Currently, it is not known whether other important problems, such as shortest paths and random walks, enjoy singular optimality. These problems admit distributed algorithms which are (essentially) time-optimal but not message-optimal [7, 16, 26, 27]. Further work is needed to address these questions.

REFERENCES

[1] Hagit Attiya and Jennifer Welch. 1998. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill, Inc.

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

[3] Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. 1990. A Trade-Off between Information and Communication in Broadcast Protocols. *J. ACM* 37, 2 (1990), 238–256.

[4] Baruch Awerbuch and David Peleg. 1990. Sparse partitions. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS)*. 503–513.

[5] Francis Y. L. Chin and H. F. Ting. 1985. An almost linear time and $O(n \log n + e)$ messages distributed algorithm for minimum-weight spanning trees. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science (FOCS)*. 257–266.

[6] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. 2012. Distributed Verification and Hardness of Distributed Approximation. *SIAM J. Comput.* 41, 5 (2012), 1235–1265.

[7] Atish Das Sarma, Danupon Nanongkai, Gopal Pandurangan, and Prasad Tetali. 2013. Distributed Random Walks. *J. ACM* 60, 1, Article 2 (2013), 2:1–2:31 pages.

[8] Michael Elkin. 2006. A faster distributed protocol for constructing a minimum spanning tree. *J. Comput. Syst. Sci.* 72, 8 (2006), 1282–1308.

[9] Michael Elkin. 2006. An Unconditional Lower Bound on the Time-Approximation Trade-off for the Distributed Minimum Spanning Tree Problem. *SIAM J. Comput.* 36, 2 (2006), 433–456.

[10] Michael Elkin, Hartmut Klauck, Danupon Nanongkai, and Gopal Pandurangan. 2014. Can Quantum Communication Speed Up Distributed Computation?. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC)*. 166–175.

[11] Michalis Faloutsos and Mart Molle. 2004. A linear-time optimal-message distributed algorithm for minimum spanning trees. *Distributed Computing* 17, 2 (2004), 151–170.

[12] Eli Gafni. 1985. Improvements in the time complexity of two message-optimal election algorithms. In *Proceedings of the 4th Symposium on Principles of Distributed Computing (PODC)*. 175–185.

[13] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira. 1983. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Trans. Program. Lang. Syst.* 5, 1 (1983), 66–77.

[14] Juan A. Garay, Shay Kutten, and David Peleg. 1998. A Sublinear Time Distributed Algorithm for Minimum-Weight Spanning Trees. *SIAM J. Comput.* 27, 1 (1998), 302–316.

[15] 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 *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing (PODC)*. 91–100.

[16] 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 ACM Symposium on Theory of Computing (STOC)*.

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

[18] Valerie King, Shay Kutten, and Mikkel Thorup. 2015. Construction and Impromptu Repair of an MST in a Distributed Network with $o(m)$ Communication. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*. 71–80.

[19] Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. 2015. Distributed Computation of Large-Scale Graph Problems. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 391–410.

[20] Liah Kor, Amos Korman, and David Peleg. 2013. Tight Bounds for Distributed Minimum-Weight Spanning Tree Verification. *Theory Comput. Syst.* 53, 2 (2013), 318–340.

[21] Shay Kutten, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. 2014. Distributed Symmetry Breaking in Hypergraphs. In *Proceedings of the 28th International Symposium on Distributed Computing (DISC)*. 469–483.

[22] Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. 2015. On the Complexity of Universal Leader Election. *J. ACM* 62, 1, Article 7 (2015), 7:1–7:27 pages.

[23] Shay Kutten and David Peleg. 1998. Fast Distributed Construction of Small k-Dominating Sets and Applications. *J. Algorithms* 28, 1 (1998), 40–66.

[24] Christoph Lenzen. 2016. Lecture notes on Theory of Distributed Systems. https://www.mpi-inf.mpg.de/fileadmin/inf/d1/teaching/winter15/tods/ToDS.pdf.

[25] Nancy Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers.

[26] Danupon Nanongkai. 2014. Distributed Approximation Algorithms for Weighted Shortest Paths. In *Proceedings of the 46th ACM Symposium on Theory of Computing (STOC)*. 565–573.

[27] Danupon Nanongkai, Atish Das Sarma, and Gopal Pandurangan. 2011. A tight unconditional lower bound on distributed randomwalk computation. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. 257–266.

[28] Gopal Pandurangan, David Peleg, and Michele Scquizzato. 2016. Message lower bounds via efficient network synchronization. In *Proceedings of the 23rd International Colloquium on Structural Information and Communication Complexity (SIROCCO)*. 75–91.

[29] David Peleg. 1998. Distributed Matroid Basis Completion via Elimination Upcast and Distributed Correction of Minimum-Weight Spanning Trees. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP)*. 164–175.

[30] David Peleg. 2000. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics.

[31] David Peleg and Vitaly Rubinovich. 2000. A Near-Tight Lower Bound on the Time Complexity of Distributed Minimum-Weight Spanning Tree Construction. *SIAM J. Comput.* 30, 5 (2000), 1427–1442.

[32] Robert E. Tarjan. 1983. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics.

[33] Gerard Tel. 1994. *Introduction to Distributed Algorithms.* Cambridge University Press.