An Almost Singularly Optimal Asynchronous Distributed MST Algorithm

Fabien Dufoulon
Department of Computer Science, University of Houston, Houston, TX, USA

Shay Kutten
Faculty of Industrial Engineering and Management,
Technion – Israel Institute of Technology, Haifa, Israel

William K. Moses Jr.
Department of Computer Science, University of Houston, Houston, TX, USA

Gopal Pandurangan
Department of Computer Science, University of Houston, Houston, TX, USA

David Peleg
Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot, Israel

Abstract

A singularly (near) optimal distributed algorithm is one that is (near) optimal in *two* criteria, namely, its time and message complexities. For *synchronous* $\mathcal{CONGEST}$ networks, such algorithms are known for fundamental distributed computing problems such as leader election [Kutten et al., JACM 2015] and Minimum Spanning Tree (MST) construction [Pandurangan et al., STOC 2017, Elkin, PODC 2017]. However, it is open whether a singularly (near) optimal bound can be obtained for the MST construction problem in general *asynchronous* $\mathcal{CONGEST}$ networks.

In this paper, we present a randomized distributed MST algorithm that, with high probability, computes an MST in *asynchronous* $\mathcal{CONGEST}$ networks and takes $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time and $\tilde{O}(m)$ messages[1], where n is the number of nodes, m the number of edges, D is the diameter of the network, and $\varepsilon > 0$ is an arbitrarily small constant (both time and message bounds hold with high probability). Since $\tilde{\Omega}(D + \sqrt{n})$ and $\Omega(m)$ are respective time and message lower bounds for distributed MST construction in the standard KT_0 model, our algorithm is message optimal (up to a polylog(n) factor) and almost time optimal (except for a D^{ε} factor). Our result answers an open question raised in Mashregi and King [DISC 2019] by giving the first known asynchronous MST algorithm that has sublinear time (for all $D = O(n^{1-\varepsilon})$) and uses $\tilde{O}(m)$ messages. Using a result of Mashregi and King [DISC 2019], this also yields the first asynchronous MST algorithm that is sublinear in both time and messages in the KT_1 $\mathcal{CONGEST}$ model.

A key tool in our algorithm is the construction of a low diameter rooted spanning tree in asynchronous $\mathcal{CONGEST}$ that has depth $\tilde{O}(D^{1+\varepsilon})$ (for an arbitrarily small constant $\varepsilon > 0$) in $\tilde{O}(D^{1+\varepsilon})$ time and $\tilde{O}(m)$ messages. To the best of our knowledge, this is the first such construction that is almost singularly optimal in the asynchronous setting. This tree construction may be of independent interest as it can also be used for efficiently performing basic tasks such as *verified* broadcast and convergecast in asynchronous networks.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Probabilistic algorithms; Mathematics of computing → Discrete mathematics

Keywords and phrases Asynchronous networks, Minimum Spanning Tree, Distributed Algorithm, Singularly Optimal

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.19

Related Version *Full Version*: https://arxiv.org/abs/2210.01173

[1] The $\tilde{O}$ notation hides a polylog(n) factor and the $\tilde{\Omega}$ notation hides a $1/$ polylog(n) factor.

36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 19; pp. 19:1–19:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding *Fabien Dufoulon*: This work was supported in part by NSF grants CCF-1717075, CCF-1540512, IIS-1633720, and BSF grant 2016419.

Shay Kutten: This work was supported in part by the Bi-national Science Foundation (BSF) grant 2016419 and supported in part by ISF grant 1346/22.

William K. Moses Jr.: This work was supported in part by NSF grants CCF1540512, IIS-1633720, CCF-1717075, and BSF grant 2016419.

Gopal Pandurangan: This work was supported in part by NSF grants CCF-1717075, CCF-1540512, IIS-1633720, and BSF grant 2016419.

David Peleg: This work was supported in part by the US-Israel Binational Science Foundation grant 2018043.

1 Introduction

1.1 Background and Motivation

Singularly (near) optimal distributed algorithms are those that are (near) optimal both in their message complexity and in their time complexity.[2] The current paper is intended as a step in expanding the study of "which problems admit singularly optimal algorithms" from the realm of synchronous $\mathcal{CONGEST}$ networks to that of *asynchronous* ones.

An important example of a problem that has been studied in the context of singularly (near) optimal algorithms is minimum-weight spanning tree (MST) construction. This has become a rather canonical problem in the sub area of distributed graph algorithms and was used to demonstrate and study various concepts such as the congested clique model (Lotker et al. [40]), proof labeling schemes (Korman et al. [36]), networks with latency and capacity (Augustine et al. [3]), cognitive radio networks (Rohilla et al. [52]), distributed applications of graph sketches (King et al. [33]), distributed computing with advice (Fraigniaud et al. [21]), distributed verification and hardness of approximation (Kor et al. [34], Korman and Kutten [35] and Das Sarma et al. [14]), self-stabilizing algorithms (Gupta and Srimani [29] and many other papers), distributed quantum computing (Elkin et al. [19]) and more. The study of the MST problem in what we now call the $\mathcal{CONGEST}$ model started more than forty years ago, see Dalal, and also Spira [12, 13, 56].

The seminal paper of Gallager, Humblet, and Spira (GHS) [22] presented a distributed algorithm for an *asynchronous* network that constructs an MST in $O(n \log n)$ time using $O(m + n \log n)$ messages, where n and m denote the number of nodes and the number of edges of the network, respectively. The time complexity was later improved by Awerbuch and by Faloutsos and Moelle to $O(n)$ [5, 20], while keeping the same order of message complexity.

The message complexity of GHS algorithm is (essentially) optimal, since it can be shown that for any $1 \leq m \leq n^2$, there exists a graph with $\Theta(m)$ edges such that $\Omega(m)$ is a lower bound on the message complexity of constructing even a spanning tree (even for randomized algorithms) [38].[3] Moreover, the time complexity bound of $O(n)$ bound is *existentially* optimal (in the sense that there exist graphs (of high diameter) for which this is the best possible). However, the time bound is not optimal if one parameterizes the running time in terms of the network diameter D, which can be much smaller than n. In a *synchronous* network, Garay, Kutten, and Peleg [23] gave the first such distributed algorithm for the MST

[2] In this paper, henceforth, when we say "near optimal" we mean "optimal up to a polylog(n) factor", where n is the network size.

[3] This message lower bound holds in the so-called KT_0 model, which is assumed in this paper. See Section 1.4 for more details.

problem with running time $\tilde{O}(D+n^{0.614})$, which was later improved by Kutten and Peleg [39] to $\tilde{O}(D+\sqrt{n})$ (again for a synchronous network).However, both these algorithms are not message-optimal as they exchange $O(m+n^{1.614})$ and $O(m+n^{1.5})$ messages, respectively.

Conversely, it was established by Peleg and Rubinovich [51] that $\tilde{\Omega}(D+\sqrt{n})$ is a lower bound on the time complexity of distributed MST construction that applies even to low-diameter networks ($D=\Omega(\log n)$), and to the synchronous setting. The lower bound of Peleg and Rubinovich applies to exact, deterministic algorithms. This lower bound was further extended to randomized (Monte Carlo) algorithms, approximate constructions, MST verification, and more (see [41, 40, 17, 14]).

Pandurangan, Robinson and Scquizzato [47, 49] showed that MST admits a randomized singularly near optimal algorithm in *synchronous* $\mathcal{CONGEST}$ networks; their algorithm uses $\tilde{O}(m)$ messages and $\tilde{O}(D+\sqrt{n})$ rounds. Subsequently, Elkin [18] presented a simpler, singularly optimal deterministic MST algorithm, again for synchronous networks.

For *asynchronous* networks, one can obtain algorithms that are separately time optimal (by combining [39] with a synchronizer, see Awerbuch [4]) or message optimal [22] for the MST problem, but it is open whether one can obtain an asynchronous distributed MST algorithm that is singularly (near) optimal. This is one of the main motivations for this work. An additional motivation is to design tools that can be useful for constructing singularly optimal algorithms for other fundamental problems in asynchronous networks.

In general, designing singularly optimal algorithms for *asynchronous* networks seems harder compared to synchronous networks. In *synchronous* networks, besides MST construction, singularly (near) optimal algorithms have been shown in recent years for leader election, (approximate) shortest paths, and several other problems [38, 30]. However, all these results *do not* apply to asynchronous networks. Converting synchronous algorithms to work on asynchronous networks generally incur heavy cost overhead, increasing either time or message complexity or both substantially. In particular, using *synchronizers* [4] to convert a singularly optimal algorithm to work in an asynchronous network generally renders the asynchronous algorithm not singularly optimal. Using a synchronizer can significantly increase either the time or the message complexity or both far beyond the complexities of the algorithm presented here. Furthermore, there can be a non-trivial cost associated with *constructing* such a synchronizer in the first place.

For example, applying the simple α synchronizer [4] (which does not require the a priori existence of a leader or a spanning tree) to the singularly optimal synchronous MST algorithm of [47, 49] or [18] yields an asynchronous algorithm with message complexity of $\tilde{O}(m(D+\sqrt{n}))$ and time complexity of $\tilde{O}(D+\sqrt{n})$; this algorithm is time optimal, but *not* message optimal. Some other synchronizers (see, e.g., Awerbuch and Peleg [9]), do construct efficient synchronizers that can achieve near optimal conversion from synchronous to asynchronous algorithms with respect to both time and messages, but constructing the synchronizer itself requires a substantial preprocessing or initialization cost. For example, the message cost of the synchronizer setup protocol of [9] can be as high as $O(mn)$.

Another rather tempting idea to derive an MST algorithm that would be efficient both in time and in messages would be to convert a result of Mashreghi and King [44] (see also [43] and discussion in Section 1.4), originally designed in the asynchronous KT_1 $\mathcal{CONGEST}$ model[4] to the more common KT_0 model assumed here. In particular, they give an asynchronous MST algorithm that takes $O(n)$ time and $\tilde{O}(n^{1.5})$ messages. Note that one can convert an

[4] In KT_1 model it is assumed that nodes know the identities of their neighbors (cf. Section 1.4), unlike the KT_0 model, where nodes don't have that knowledge.

algorithm in the KT_1 model to work in the KT_0 model by allowing each node to communicate with all its neighbors in one round; this takes an additional $\tilde{O}(m)$ messages. Hence, with such a conversion the message complexity of the above algorithm would be essentially optimal (i.e., $\tilde{O}(m)$), but the time complexity would be $O(n)$ which is only existentially optimal, and can be significantly higher than the lower bound of $\tilde{O}(D + \sqrt{n})$. In fact, as we will discuss later, our result answers an open question posed in [44] and gives MST algorithms with improved bounds in asynchronous KT_1 model (cf. Section 1.3).

Instead of using a synchronizer, a better approach might be to design an algorithm directly for an asynchronous network. As an example, consider the fundamental leader election problem, which is simpler than the MST construction problem. Till recently, a singularly optimal asynchronous leader election algorithm was not known. Applying a synchronizer to known *synchronous* singularly optimal leader election algorithms *does not* yield singularly optimal asynchronous algorithms. For example, applying the simple α synchronizer to the singularly optimal synchronous leader election algorithm of [38] yields an asynchronous algorithm with message complexity of $O(mD \log n)$ and time complexity of $O(D)$; this algorithm is not message optimal, especially for large diameter networks. Other synchronizers such as β and γ of [4] and that of [9], require the a priori existence of a *leader* or *a spanning tree* and hence cannot be used for leader election. The work of Kutten et al. [37] presented a singularly (near) optimal leader election for asynchronous networks that takes $\tilde{O}(m)$ messages and $\tilde{O}(D)$ time.[5] That algorithm did not use a synchronizer and was directly designed for an asynchronous network. The leader election algorithm of [37] is a useful subroutine in our MST algorithm.

1.2 The Distributed Computing Model

The distributed network is modeled as an arbitrary undirected connected weighted graph $G = (V, E, w)$, where the node set V represent the processors, the edge set E represents the communication links between them, and $w(e)$ is the weight of edge $e \in E$. D denotes the hop-diameter (that is, the unweighted diameter) of G, in this paper, diameter always means hop-diameter. 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.) We make the common assumption that each node has a unique identity (this is not essential, but simplifies presentation), and at the beginning of computation, each node v accepts as input its own identity number (ID) and the weights of the edges incident to it. Thus, a node has only *local* knowledge. 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 (the identity of the 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 [50] 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) [50]. The KT_0 model is extensively used in distributed computing literature including MST algorithms (see e.g., [50, 48] and the references therein). While we design an algorithm for the KT_0 model, our algorithm also yields an improvement in the KT_1 model [7, 50] where each node has an initial knowledge of the identities of its neighbors.

[5] This algorithm is singularly near optimal, since $\Omega(m)$ and $\Omega(D)$ are message and lower bounds for leader election even for randomized Monte Carlo algorithms [38].

We assume that nodes have knowledge of n (in fact a constant factor approximation of n is sufficient), the network size. We note that quite a few prior distributed algorithms require knowledge of n, see e.g. [6, 53, 2, 37]. We assume that processors can access *private unbiased random bits*.

We assume the standard *asynchronous* $\mathcal{CONGEST}$ communication model [50], where messages (each message is of $O(\log n)$ bits) sent over an edge incur unpredictable but finite delays, in an error-free and FIFO manner (i.e., messages will arrive in sequence). As is standard, it is assumed that a message takes *at most one time unit* to be delivered across an edge. Note that this is just for the sake of the analysis of time complexity, and does not imply that nodes know an upper bound on the delay of any message. As usual, local computation within a node is assumed to be instantaneous and free; however, our algorithm will involve only lightweight local computations.

We assume an *adversarial wake-up* model, where node wake-up times are scheduled by an adversary (who may decide to keep some nodes dormant) which is standard in prior asynchronous protocols (see [1, 22, 55]). Nodes are initially asleep, and a node enters the execution when it is woken up by the environment or upon receiving messages from other nodes.[6]

The time complexity is measured from the moment the first node wakes up. The adversary wakes up nodes and delays each message in an *adaptive* fashion, i.e., when the adversary makes a decision to wake up a node or delay a message, it has access to the results of all previous coin flips. In the asynchronous setting, once a node enters execution, it performs all the computations required of it by the algorithm, and sends out messages to neighbors as specified by the algorithm. At the end of the computation, we require each node to know which of its incident edges belong to the MST. When we say that an algorithm has termination detection, we mean that all nodes detect termination, i.e., each node detects that its own participation in the algorithm is over.

1.3 Our Contributions

Almost Singularly Optimal Asynchronous MST Algorithm. Our main contribution is a randomized distributed MST algorithm that, with high probability, computes an MST in *asynchronous* $\mathcal{CONGEST}$ networks and takes $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time and $\tilde{O}(m)$ messages, where n is the number of nodes, m the number of edges, D is the diameter of the network, and $\varepsilon > 0$ is an arbitrarily small constant (both time and message bounds hold with high probability) (cf. Theorem 9). Since $\tilde{\Omega}(D + \sqrt{n})$ and $\Omega(m)$ are respective time and message lower bounds for distributed MST construction in the KT_0 model, our algorithm is message optimal (up to a polylog(n) factor) and almost time optimal (except for a $\tilde{O}(D^{\varepsilon})$ factor).

Asynchronous MST in KT_1 in Sublinear Messages and Time. Our result answers an open problem raised in Mashregi and King [44] (see also [45, 43]). They ask if there exists an asynchronous MST algorithm that takes sublinear time if the diameter of the network is low, and has $\tilde{O}(m)$ message complexity. They remark that if such an algorithm exists, then it would improve their result giving better bounds for asynchronous MST in KT_1

[6] Although standard, the adversarial wake up model, in our setting, is not more difficult compared to the alternative *simultaneous wake up* model where all nodes are assumed to be awake at the beginning of the computation. Indeed, in the adversarial wake up model, awake nodes can broadcast (by simply flooding) a "wake up" message which can wake up all nodes; this takes only $O(m)$ messages and $O(D)$ time and hence within the singularly optimal bounds.

$\mathcal{CONGEST}$. Our result answers their question in the affirmative by giving the first known asynchronous MST algorithm that has sublinear time (for all $D = O(n^{1-\delta})$, where $\delta > 0$ is an arbitrarily small constant) and uses $\tilde{O}(m)$ messages. Furthermore, as indicated in Mashregi and King [44], this also yields the first asynchronous MST algorithm that is *sublinear* in *both* time and messages in the KT_1 $\mathcal{CONGEST}$ model. More precisely, plugging our asynchronous MST algorithm in the result of [44]([Theorem 1.2]) gives an asynchronous MST algorithm that takes $\tilde{O}(D^{1+\varepsilon} + n^{1-2\delta})$ time and $\tilde{O}(n^{3/2+\delta})$ messages for any small constant $\varepsilon > 0$ and for any $\delta \in [0, 0.25]$ (cf. Theorem 10). This gives a tradeoff result between time and messages. In particular, setting $\delta = 0.25$ yields an asynchronous MST algorithm that has (almost optimal) time complexity $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ and message complexity $\tilde{O}(n^{7/4})$.

Low Diameter Spanning Tree Construction. A key tool in our algorithm is the construction of a low diameter rooted spanning tree in asynchronous $\mathcal{CONGEST}$ that has depth $\tilde{O}(D^{1+\varepsilon})$ (for an arbitrarily small constant $\varepsilon > 0$) in time $\tilde{O}(D^{1+\varepsilon})$ time and $\tilde{O}(m)$ messages. To the best of our knowledge, this is the first such construction that is almost singularly optimal in the asynchronous setting. This tree construction is of independent interest as it can also be used for *efficiently* (under both time and messages) performing tasks such as upcast and downcast which are very common tools in distributed algorithms (these are described, for completeness, in Appendix A). Informally, an upcast (using the tree) provides a feedback (i.e., verification) to the broadcast (downcast) initiator such that (1) the broadcast initiator knows when the broadcast terminates (based on acknowledgements from all nodes) and (2) the initiator can get compute a value based on the inputs of all the nodes (e.g., their sum). This verified broadcast is crucial in the asynchronous setting that allows the initiator to know when the broadcast has reached all nodes and thereafter proceed to the next step of the computation.

We note that one could have used a BFS tree instead of a low-diameter tree. However, the best known BFS tree construction in the asynchronous setting is due to Awerbuch [6] which takes $O(D^{1+\varepsilon})$ time and $O(m^{1+\varepsilon})$ messages (for arbitrarily small constant $\varepsilon > 0$). This algorithm (which is deterministic) is not message optimal, unlike ours, and hence will only yield an MST algorithm with $O(m^{1+\varepsilon})$ message complexity. Furthermore, though our algorithm does not compute a BFS (but it is sufficient for MST purposes) and is randomized, it is significantly simpler to understand and prove correctness for when compared to Awerbuch's algorithm. We also note that apart from the leader election and spanning tree primitives, the rest of the MST algorithm is deterministic.

1.4 Additional Related Work

The distributed MST problem has been studied intensively for the last four decades and there are several results known in the literature, including several recent results, both for synchronous and asynchronous networks (including the ones mentioned in Section 1), see e.g., [18, 16, 48, 24, 30, 32, 45, 42, 47, 49] and the references therein.

We note that the results of this paper and that of leader election of [37] (for asynchronous networks) as well as those of [47, 49] and [18] (for synchronous networks) assume the so-called *clean network model*, a.k.a. KT_0 [50] (see Section 1.2), where nodes do not have initial knowledge of the identity of their neighbors. But the optimality of above results does not in general apply to the KT_1 model, where nodes have initial knowledge of the identities of their neighbors. It is clear that for time complexity by itself, the distinction between KT_0 and KT_1 does not matter (as one can simulate KT_1 in KT_0 in one round/time unit by each node sending its ID to all its neighbors) but it is significant when considering message complexity (as the just mentioned simulation costs $\Theta(m)$ messages). Awerbuch et al. [7] show that $\Omega(m)$ is a message lower bound for broadcast (and hence for construction of a spanning tree as well) in the KT_1 model, if one allows only (possibly randomized Monte Carlo) comparison-based algorithms, i.e., algorithms that can operate on IDs only by comparing them. (We note that all algorithms mentioned earlier in this subsection are comparison-based, including ours.)

On the other hand, for *randomized non-comparison-based* algorithms, the message lower bound of $\Omega(m)$ does not apply in the KT_1 model. King et al. [33] presented a randomized, non-comparison-based Monte Carlo algorithm in the KT_1 model for MST construction in $\tilde{O}(n)$ messages ($\Omega(n)$ is a message lower bound) (see also [42]). While this algorithm achieves $o(m)$ message complexity (when $m = \omega(n \operatorname{polylog} n)$), it is *not* time-optimal, as it takes time $\tilde{O}(n)$ rather than $\tilde{O}(D + \sqrt{n})$. Algorithms with improved round complexity but worse message complexity, and more generally, trade-offs between time and messages, are shown in [26, 27]. We note that all these results are for *synchronous* networks. As discussed in Section 1, the works of [44, 43, 45] address asynchronous MST construction in KT_1 model and present algorithms that take $o(m)$ messages.

2 Low Diameter Spanning Tree Algorithm

Let us now describe a novel algorithm for constructing a low diameter spanning tree in a time-efficient and (near) message-optimal manner in an *asynchronous* network. This serves as a crucial ingredient for our MST algorithm of Section 3.

2.1 Randomized Low Diameter Decomposition (MPX)

Let $\overline{G} = (\overline{V}, \overline{E})$ be any (undirected, unweighted) graph with $\overline{n} \leq n$ nodes and $\overline{m} \leq m$ edges; in particular, $\overline{G}$ can be different from the communication graph. A probabilistic (β, r) *low diameter decomposition* of $\overline{G}$ is a partition of $\overline{V}$ into disjoint node sets $\overline{V}_1, \ldots, \overline{V}_t$ called *clusters*. The partition satisfies (1) each cluster $\overline{V}_i$ has strong diameter r, i.e., $dist_{\overline{G}[V_i]}(u, v) \leq r$ for any two nodes $u, v \in \overline{V}_i$, and (2) the probability that an edge $e \in \overline{E}$ is an inter-cluster edge (that is, the endpoints of e are in different clusters) is at most β.

MPX Decomposition in Synchronous $\mathcal{CONGEST}$. Let us describe a simple distributed variant of the MPX decomposition algorithm of Miller et al. [46] – Procedure **MPX** – executed in a synchronous setting with simultaneous wakeup on graph $\overline{G}$. In Subsect. 2.2, we execute the algorithm on virtual cluster graphs (where each node is in fact a set of nodes in the communication graph G) and also describe the distributed simulation required to do so.

Let $\delta_{max} = \lfloor 2 \cdot \frac{\ln n}{\beta} \rfloor$. Initially, each node $v \in \overline{V}$ draws a random variable δ_v from the exponential random distribution with parameter β and sets its *start-time* variable S_v to $\max\{1, \delta_{max} - \lfloor \delta_v \rfloor\}$. Procedure **MPX** guarantees the following through simple flooding: (1) each node $v \in \overline{V}$ is assigned to the cluster of the node $u = argmin_{w \in \overline{V}}\{(dist_{\overline{G}}(v, w) + S_w, id_w)\}$ and (2) each cluster has a spanning tree of depth at most δ_{max}. (Each node locally keeps information about the edge to its parent in the spanning tree. In other words, the spanning tree is oriented towards the root.)

More precisely, the "simple flooding" is done in $\delta_{max} + 1$ rounds. Initially, all nodes are *unassigned*. In round i, each newly-assigned node v (i.e., assigned in round $i - 1$) sends to its neighbors a message containing the ID of the cluster leader. Other assigned nodes do nothing. Finally, for each unassigned node v, let M_{id} be the set containing all received IDs, as well as id_v if $S_v = i$. If M_{id} is the empty set, v does nothing. Otherwise, v assigns itself to the cluster of the node u with the lexicographically smallest ID in M_{id}. If $u \neq v$, v keeps the edge (an arbitrary one if there are multiple such edges) along which it receives id_u as the edge to its parent. (Note that this spanning tree guarantees that the cluster is connected and has strong diameter at most $\frac{4\ln n}{\beta}$.)

Analysis. The following lemmas are known results from [46, 31, 10, 11]. For completeness, proofs are given in Appendix B.

▶ **Lemma 1.** *Procedure* ***MPX*** *computes a* $(2\beta, \frac{4\ln n}{\beta})$ *low-diameter decomposition of* $\overline{G}$ *w.h.p. in* $O(\frac{\ln n}{\beta})$ *time and* $O(m\frac{\ln n}{\beta})$ *messages in the synchronous setting.*

From the low diameter decomposition computed by Procedure **MPX** (or in fact, from any partition $\mathcal{P}$ of $\overline{V}$ into disjoint node sets $\overline{V}_1, \ldots, \overline{V}_t$), one can define a cluster graph $\overline{G}^* = (\overline{V}^*, \overline{E}^*)$, as follows. Its node set $\overline{V}^* = \{\overline{V}_1, \ldots, \overline{V}_t\}$ consists of cluster nodes, one for each cluster $\overline{V}_i$ of the decomposition, and two cluster nodes $\overline{V}_i$ and $\overline{V}_j$ are adjacent in $\overline{G}^*$ if there exist two nodes w, w' in $\overline{V}$ such that $w \in \overline{V}_i$, $w' \in \overline{V}_j$ and $(w, w') \in \overline{E}$. We call $\overline{G}^*$ the *cluster graph induced by* $\mathcal{P}$.

▶ **Lemma 2.** *For any positive integer* $k \geq 1$*, if the diameter of* $\overline{G}$ *satisfies* $\overline{D} \geq k\frac{\ln^2 n}{\beta^4}$*, then the diameter of the cluster graph* $\overline{G}^*$ *is at most* $2\beta\overline{D}$*, with probability at least* $1 - \frac{1}{n^{k-2}}$.

2.2 Rooted Spanning Tree

Let us now describe an asynchronous distributed algorithm to construct a low diameter rooted spanning tree, given a pre-specified root, in a time-efficient and (near) message-optimal manner – see Theorem 3. We assume that each node knows whether it is the pre-specified root prior to the start of the algorithm. We also assume initially that the diameter of the original graph, D, is known to the nodes. We explain how to remove this assumption at the end of the section.

▶ **Theorem 3.** *Given a graph* G *with* n *nodes,* m *edges and diameter* D*, as well as a distinguished node* R*, and a constant parameter* $1 \geq \varepsilon > 0$*, the asynchronous distributed Procedure* ***ST-Cons***(ε) *computes an* $\tilde{O}(D^{1+\varepsilon})$*-diameter spanning tree rooted in* R *with termination detection, using* $\tilde{O}(D^{1+\varepsilon})$ *time with high probability and* $\tilde{O}(m)$ *messages with high probability.*

Brief Description. We construct the low diameter spanning tree in a two stage process. The first stage consists of building a sequence of increasingly coarser partitions of $G = (V, E)$. Each partition decomposes V into disjoint node sets, called clusters, with strong diameter $\tilde{O}(D^{1+\varepsilon})$; in fact, each cluster C is spanned by a tree $\hat{T}(C)$ of depth $\tilde{O}(D^{1+\varepsilon})$. (Unlike in Subsect. 2.1, this spanning tree is oriented away from the root.) The unique cluster containing the root node R will be denoted C_R. The cluster graph induced by the final partition (defined in Subsect. 2.1) has diameter $\tilde{O}(1)$. These partitions are obtained by simulating the synchronous MPX decomposition algorithm (see Subsect. 2.1) on G, then on the obtained cluster graph, and so on, for $i_m = \lceil \log_{1/(3\beta)} D \rceil$ times (where $\beta = \ln^{-\frac{1}{\varepsilon'}} n$ and $\varepsilon' \leq 1$ is to be derived in the analysis). In the second stage, we construct a breadth first search (BFS) tree T^{BFS} over the final cluster graph of phase 1, where the cluster C_R containing the pre-specified root R serves as the root of the BFS tree. We then use T^{BFS} to decide which edges of the original graph should be kept to obtain the desired rooted spanning tree $\tilde{T}$ of G with depth $\tilde{O}(D^{1+\varepsilon})$.

Detailed Description. Consider the initial graph $G(V, E) = G_0(V_0, E_0)$ and the initial trivial partition $\mathcal{P}_0$ in which each node $v \in V$ is its own cluster.

- **Stage 1:** The first stage consists of $i_m = \lceil \log_{1/(3\beta)} D \rceil$ phases, where $\beta = \ln^{-1/\varepsilon'} n$ and we assume $\ln \ln n \geq 2\varepsilon' \ln 3$. (If $\ln \ln n \leq 2\varepsilon' \ln 3 \leq 2 \ln 3$, then constructing a low diameter spanning tree efficiently is trivial.) Phase i starts with a partition $\mathcal{P}_{i-1}$ of V and the cluster graph induced by $\mathcal{P}_{i-1}$ is denoted by $G_{i-1}(V_{i-1}, E_{i-1})$. We simulate one instance of Procedure **MPX** (with parameter β) on G_{i-1} in an asynchronous setting by running an α-synchronizer between clusters, and within each cluster C, using the spanning tree $\hat{T}(C)$ to simulate the behavior of each cluster node of V_{i-1}. (Note that this well-known synchronizer is described in more detail in Appendix A.) More precisely, the root of the spanning tree $\hat{T}(C)$ simulate the behavior of cluster C (in the simulated Procedure **MPX**). To send a (same) message to its adjacent clusters, C broadcasts along $\hat{T}(C)$. To receive the message with the minimum ID (which is sufficient information for Procedure **MPX**), C convergecasts along $\hat{T}(C)$.

  The output is a partition $\mathcal{P}_i^*$ of V_{i-1} into disjoint (cluster node) sets $U_1, \ldots, U_t$ such that each U_j has a spanning tree T_j^{super} of depth $O(\frac{\ln n}{\beta})$. We transform $\mathcal{P}_i^*$ into a partition $\mathcal{P}_i$ of V, the node set of the *original* graph, into disjoint node sets $W_1, \ldots, W_t$, such that each W_j has a spanning tree $\hat{T}(W_j)$ of depth $O((\frac{\ln n}{\beta})^i)$. (In fact, we only show how to compute the spanning trees $\hat{T}(W_j)$, which induces the node sets W_j.)

  To transform $\mathcal{P}_i^*$ to $\mathcal{P}_i$, we use a simple Procedure **Transform**, sketched next. Recall that each cluster node in U_j keeps information about its parent in the spanning tree T_j^{super}. Procedure **Transform** consists of $2\frac{\ln n}{\beta}$ iterations. Each cluster node keeps an iteration counter and these counters are kept locally synchronized by running an α-synchronizer between cluster nodes. In the first iteration, the root cluster node C_R sends its ID to each adjacent cluster node C (which is its child in T_j^{super}) over the edges of the set $E_{inter} = \{(u, w) \in E(G) \mid u \in C_R, w \in C\}$, namely, all (original) inter-cluster edges between C_R and C. (Note that in fact, C_R sends its ID to all adjacent cluster nodes, but cluster nodes which are not children of C_R simply ignore that message.) Among these inter-cluster edges, every child cluster node C keeps $(u^*, w^*) = argmin_{(u,w) \in E_{inter}}\{id_w\}$, i.e., the edge whose endpoint w in C has the minimum ID. Cluster node C then reorients its tree $\hat{T}(C)$ to be rooted in w (and the inter-cluster edge is oriented towards w, i.e., from parent to child). In the next iteration, each C sends the ID of R to its children cluster nodes, if they exist, which in turn reorient their tree in the same fashion. After all iterations are done, the "combined" spanning tree $\hat{T}(W_j)$ is completed, and a simple broadcast allows all nodes in the newly computed cluster W_j to move on to the next phase. (Note that $\hat{T}(W_j)$ is oriented from the root outwards.)

- **Stage 2:** At the end of stage 1, the final partition decomposes V into clusters with strong diameter $\tilde{O}(D^{1+\varepsilon})$ and induces a cluster graph $G_f(V_f, E_f)$ of diameter $O(\log^{2+4/\varepsilon'} n)$; in fact, each cluster C is spanned by a tree $\hat{T}(C)$ of depth $\tilde{O}(D^{1+\varepsilon})$. During stage 2, the naive synchronous BFS tree construction algorithm (based on flooding, see [50]) is simulated on G_f for $O(\log^{2+4/\varepsilon'} n)$ rounds, where the designated root in V_f is the cluster C_R that contains the pre-specified root in V. Once again, this is done by running an α-synchronizer between clusters, and within each cluster, using the spanning tree $\hat{T}(C)$ to simulate the behavior of each cluster node C. After computing the BFS tree T^{BFS} on G_f, we use Procedure **Transform** – but this time for $O(\log^{2+4/\varepsilon'} n)$ rounds – to compute a spanning tree $\tilde{T}$ of G, similarly to stage 1. This final output $\tilde{T}$ is a $\tilde{O}(D^{1+\varepsilon})$ diameter spanning tree of G.

Analysis. Lemma 4 upper bounds, for each phase, the diameter of the cluster graph as well as that of the partition's clusters. Corollary 5 is obtained from Lemma 4 by considering the last phase. After which, we prove Theorem 3 using Lemma 4 and Corollary 5. The proofs of Lemma 4 and Corollary 5 are deferred to Appendix B.

▶ **Lemma 4.** *For each phase $1 \leq i \leq i_m$, (1) diam$(G_{i-1}) = \max\{(3\beta)^{i-1}D, O(\log^{2+4/\varepsilon'} n)\}$ w.h.p., and (2) each cluster C of the partition $\mathcal{P}_{i-1}$ is spanned (in the original graph G) by a tree $\hat{T}(C)$ with diam$(\hat{T}(C)) = (\frac{5 \ln n}{\beta})^{i-1}$.*

▶ **Corollary 5.** *At the end of phase i_m, (1) diam$(G_{i_m}) = O(\log^{2+4/\varepsilon'} n)$ w.h.p., and (2) each cluster C of the partition $\mathcal{P}_{i_m}$ is spanned (in the original graph G) by a tree $\hat{T}(C)$ with diam$(\hat{T}(C)) = \tilde{O}(D^{1+\varepsilon})$.*

Proof of Theorem 3. The correctness of the first stage follows from that of the simulation (using an α-synchronizer between clusters), Procedure **MPX** and Procedure **Transform**. Next, let us show the time and message complexity of the first stage. During each phase $1 \leq i \leq i_m$, Procedure **MPX** is simulated on G_{i-1} for $O(\frac{\log n}{\beta}) = \tilde{O}(1)$ rounds. Hence, each cluster C simulates $\tilde{O}(1)$ rounds. In each round, the cluster broadcasts once over the cluster's spanning tree $\hat{T}(C)$, sends one message per inter-cluster edge over to adjacent clusters, and convergecasts once over $\hat{T}(C)$. By Lemma 4, $\hat{T}(C)$ has depth $\tilde{O}(D^{1+\varepsilon})$. Hence, each round of Procedure **MPX** is simulated in at most $\tilde{O}(D^{1+\varepsilon})$ time and using $O(m)$. Adding up over all phases results in $\tilde{O}(D^{1+\varepsilon})$ time and $\tilde{O}(m)$ messages. Note that running an α-synchronizer (between the clusters) induces only an $\tilde{O}(1)$ message overhead per (inter-cluster) edge over all rounds, but no time overhead. Thus Procedure **MPX** is simulated in $\tilde{O}(D^{1+\varepsilon})$ time and using $\tilde{O}(m)$ messages. Similarly, in Procedure **Transform**, each cluster C simulates $\tilde{O}(1)$ rounds. In each round, the cluster broadcasts twice over the cluster's spanning tree $\hat{T}(C)$, sends one message per inter-cluster edge over to adjacent clusters, and convergecasts twice over $\hat{T}(C)$ (where the additional broadcast and convergecast allows to reorient $\hat{T}(C)$). Therefore, it can be seen that Procedure **Transform** also takes $\tilde{O}(D^{1+\varepsilon})$ time and uses $\tilde{O}(m)$ messages. Finally, the first stage has at most $i_m = \tilde{O}(1)$ phases, and thus takes $\tilde{O}(D^{1+\varepsilon})$ time and uses $\tilde{O}(m)$ messages.

By Corollary 5, the final cluster graph has a diameter of $O(\log^{2+4/\varepsilon'} n)$. Given that, the correctness of the second stage follows from that of the simulation (using an α-synchronizer between clusters), the naive synchronous BFS tree construction algorithm and Procedure **Transform**. As for the time and message complexity, the same approach (used for stage 1 above) shows that the second stage takes $\tilde{O}(D^{1+\varepsilon})$ time and uses $\tilde{O}(m)$ messages. ◀

Removing the Requirement of the Knowledge of D. In the previously described algorithm, we assumed that each node knew the value of D, the diameter of the original graph. This assumption can be removed by having each node guess the value of $D = 2^1, 2^2, \ldots$ until we arrive at the correct guess (an at most 2-approximation of D).

An issue that must be addressed, however, is that nodes need some way to determine whether they have correctly guessed the value of D or not. This can be done at the end of the second stage. Recall that the naive synchronous BFS tree construction is simulated for $O(\log^{2+4/\varepsilon'} n) = \tilde{O}(1)$ rounds. If the estimate of D is too small, the cluster graph obtained at the end of the first stage, G_f, may have diameter strictly greater than $O(\log^{2+4/\varepsilon'} n)$, in which case T^{BFS} may not cover the whole graph G_f. As a result, once $\tilde{T}$ is constructed from T^{BFS} using Procedure **Transform**, some nodes may exist outside the spanning tree $\tilde{T}$. This condition can be detected by the leaves of $\tilde{T}$ and a simple convergecast can be used to check if this condition holds true. In case it does, the root of $\tilde{T}$ can initiate a broadcast over the entire original graph to update the guess of D and run the algorithm with this updated guess. (Note that if the estimate of D is too small, it may still happen that T^{BFS} covers the whole graph G_f, in which case we correctly compute a low diameter spanning tree $\tilde{T}$ of G and the algorithm terminates.)

This modification increases the time complexity of the algorithm by at most a constant factor, and its message complexity by a factor of at most $O(\log D)$.

3 The Asynchronous MST Algorithm

In this section, we develop a randomized algorithm to construct an MST with high probability for a given graph in $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time with high probability and $\tilde{O}(m)$ messages with high probability (for any constant $\varepsilon > 0$).

3.1 High-level Overview of the Algorithm

We implement on an asynchronous network a variant of the singularly near optimal *synchronous* MST algorithms of [18, 47]. The algorithm can be divided into three stages. In stage I, we pre-process the network so that subsequent processes are fast and message efficient. Stages II and III correspond to the actual MST algorithm.

In order to ensure that nodes participate in this multi-stage algorithm in the proper sequence, we append a constant number of bits to each message to indicate the stage number that message corresponds to. A node u knows which stage number it is currently in and can queue received messages that belong to a later stage. These messages will be processed later, once u reaches to the corresponding stage.

Stage I: Pre-Processing the Graph. In this stage, we run a few preparatory procedures on the graph. Specifically, we first elect a leader, then construct a low diameter spanning tree $\mathcal{T}$, and finally estimate the diameter of $\mathcal{T}$. In more detail, for the first stage we utilize the singularly (near) optimal algorithm of [37] to elect a unique leader $\mathcal{L}$ in $O(D + \log^2 n)$ time and $O(m \log^2 n)$ messages. Subsequently, we run the **ST-Cons**(ε) algorithm of Section 2 (for a constant parameter $1 \geq \varepsilon > 0$) to construct a low diameter spanning tree $\mathcal{T}$ on G rooted at $\mathcal{L}$. Then, we use a known application of the Wave&Echo technique (see, e.g., [54, 58]) to have the root calculate the diameter of the constructed spanning tree D', which we know is an $\tilde{O}(D^{\varepsilon})$ approximation of the diameter D of the original graph G, in $O(D')$ time and $O(n)$ messages. Finally, all nodes in the tree participate in a simple broadcast on the spanning tree $\mathcal{T}$ to send this knowledge of D' to all nodes in the graph in $O(D')$ time and $O(n)$ messages.

Stage II: Controlled-GHS. The **Controlled-GHS** algorithm, introduced in [23, 39], is a *synchronous* version of the classical Gallager-Humblet-Spira (GHS) algorithm [22, 50] with some modifications, aiming to balance the size and diameter of the resulting fragments. Here, we convert to the asynchronous setting a variant of the (synchronous) **Controlled-GHS** as described in [47, 49].

Recall that the synchronous GHS algorithm (see, e.g., [50]) 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 minimum-weight outgoing edge (MOE) – these edges are guaranteed to be in the MST [57]. The MST fragments are merged via the MOEs to form larger 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 is essentially optimal, and the time complexity is $O(n \log n)$. Unfortunately, the time complexity of the GHS algorithm is not optimal, because much of the communication during a phase uses *only the MST fragment edges*, and the diameter of an MST fragment can be significantly larger than the graph diameter D (possibly as large as $\Omega(n)$).

In order to obtain a time-optimal algorithm, the **Controlled-GHS** algorithm controls the growth of the diameter of the MST fragments during merging. This is achieved by computing, in each phase, a maximal matching on the fragment forest with additional edges being carefully chosen to ensure enough fragments merge together, and merging fragments accordingly. 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 $\max\{\lceil \log_2 \sqrt{n}\rceil, \lceil \log_2 D'\rceil\}$, it produces at most $\min\{\sqrt{n}, n/D'\}$ fragments, each of which has diameter $O(D' + \sqrt{n})$. These are called *base fragments.* **Controlled-GHS** up to phase $\max\{\lceil \log_2 \sqrt{n}\rceil, \lceil \log_2 D'\rceil\}$ can be implemented using $\tilde{O}(m)$ messages in $\tilde{O}(D' + \sqrt{n})$ rounds in a synchronous network.

Stage II executes the **Controlled-GHS** algorithm in an asynchronous network. We postpone the discussion of the technical details involved in efficiently implementing the asynchronous algorithm to Section 3.2. The main challenge, however, is that the synchronous version heavily relies on the phases being synchronized. Here, we cannot naively use a synchronizer (such as α) for synchronization, as it would have increased the message complexity substantially. Instead we use a light-weight synchronization that incurs only $\tilde{O}(m)$ overhead in messages.

Finally, we ensure that all nodes know the exact number of fragments that were constructed at the end of this phase. The root of each fragment T calculates the number of nodes present in T and forms a tuple consisting of this value and the ID of T. Subsequently, each fragment root participates in the upcast of its tuple in the low diameter spanning tree $\mathcal{T}$ on G'. All tuples are accumulated at $\mathcal{L}$ in $O(\min\{\sqrt{n}, n/D'\} + D')$ time and $O(n)$ messages. $\mathcal{L}$ continues to listen for messages until the total number of nodes in all fragments it has heard from is equal to n, i.e., all fragments have been heard from. Now $\mathcal{L}$ broadcasts the number of fragments over $\mathcal{T}$ to all nodes in the graph in $O(D')$ time and $O(n)$ messages.

Stage III: Merging the Remaining Fragments. This stage completes the fragment merging process. However, the merging is done in a "soft" manner. The at most $\min\{\sqrt{n}, n/D'\}$ base fragments (constructed at the end of Stage II) are still retained, but each base fragments takes on an additional ID–a cluster ID, initially set to the base fragment ID. (A cluster is a collection of base fragments; at the beginning of this stage, each base fragment forms its own cluster.) Each base fragment finds an MOE to a different cluster, if such an MOE exists, and merging consists of base fragments modifying their associated cluster IDs and marking the corresponding MOE connecting clusters. All nodes participate in a simple upcast over $\mathcal{T}$, where the root of each base fragment is responsible to send up a tuple consisting of its fragment & cluster IDs, a possible MOE and the associated fragment & cluster IDs the MOE leads to.[7] It is similar to the approach of [18, 47], which uses a BFS tree to upcast these values to the root of tree; here, instead of BFS, we use the low-diameter spanning tree of

[7] It is required that each base fragment's root sends up this tuple even if it does not have an MOE (in which case the tuple only has info on the fragment ID and cluster ID of the base fragment). This is to ensure that the nodes detect termination as the root of $\mathcal{T}$, $\mathcal{L}$, already knows the fragment and cluster IDs of the base fragments so it knows how many such messages to wait for.

Section 2. Subsequently, the root calculates the appropriate MOEs (and the fragments they connect and the clusters they lead to) for each cluster and downcast these values. Each fragment then performs a broadcast of its (possibly new) cluster ID over the fragment tree (to all nodes within the fragment). This process is repeated for $O(\log n)$ phases until only one cluster remains, which represents the MST of the original graph.

Let us examine each phase i in more detail. Each base fragment finds its respective MOE, if any, and sends it to $\mathcal{L}$ via an upcast.[8] All fragment leaders can find their MOEs in $O(D' + \sqrt{n})$ time and $O(m)$ messages. Upcasting these values to $\mathcal{L}$ using tree $\mathcal{T}$ takes $O(\min\{\sqrt{n}, n/D'\} + D')$ time and $O(n)$ messages. $\mathcal{L}$ locally computes the overall MOEs of the (soft-merged) base fragments and then merges them (locally). Subsequently, all nodes of $\mathcal{T}$ participate in a downcast of these MOEs and modified cluster IDs (that $\mathcal{L}$ previously calculated) in $O(D' + \sqrt{n})$ time and $O(n)$ messages. Each base fragment performs a broadcast of its (possibly new) cluster ID to all nodes in its base fragment utilizing the base fragment tree. For all base fragments to do this, it takes a total of $O(D' + \sqrt{n})$ time and $O(n)$ messages.

3.2 Detailed Algorithm Description

We now look at each stage in more detail.

Stage I. In this stage, the nodes first run Procedure **LE** on G to elect a unique leader $\mathcal{L}$ with high probability. As a side benefit, the procedure also wakes up all nodes. Next, the nodes participate in Procedure **ST-Cons**(ε) to construct an $\tilde{O}(D^{1+\varepsilon})$ diameter spanning tree $\mathcal{T}$ of G with $\mathcal{L}$ as its root. Subsequently, all nodes participate in Procedure **Diam-Calc** so that $\mathcal{L}$ is now aware of the diameter D' of $\mathcal{T}$. Finally, all nodes participate in **Frag-Bcast** over $\mathcal{T}$ to transmit this information of D' to all nodes in the graph. (Procedures **LE**, **Diam-Calc** and **Frag-Bcast** are described in Appendix A.)

Stage II. In this stage, the nodes execute an asynchronous version of the **Controlled-GHS** algorithm [23, 47, 49]. Let us first recall the original (synchronous) **Controlled-GHS** algorithm. This algorithm merges fragments (subtrees of the MST) in phases, similarly to GHS. However, it guarantees two additional properties to hold at the end of each phase i: (a) there are at most $n/2^i$ fragments, and (b) each fragment has diameter $O(2^i)$. These guarantees are ensured through two measures. First, at the beginning of phase i, only fragments with diameter $\leq 2^i$ will participate in this phase and find MOEs. Second, in a phase i, consider the *fragment graph* whose "nodes" are the fragments (including those that do not participate) and whose edges are all the MOEs found. The algorithm first performs a maximal matching on this fragment graph and removes from the fragment graph edges that do not participate in this matching. Then, those fragments who participate in this phase and remain unmatched add their MOEs back to the fragment graph. Connected components of fragments in this final fragment graph then merge together. The algorithm is run from phase $i = 0$ to phase $i = \max\{\lceil \log_2 \sqrt{n} \rceil, \lceil \log_2 D' \rceil\}$. Due to a lack of space, the details of the adaptation of the **Controlled-GHS** algorithm to the asynchronous setting can be found in the full version of the paper.

[8] Note that as the algorithm progresses, two adjacent base fragments may belong to the same overall cluster, possibly resulting in one of those base fragments having no MOE to a different cluster.

After completing the last phase of the above process, we are almost ready to move to stage III of the algorithm.[9] Some final cleanup is first needed. We need two things in order to ensure our subsequent upcasts and downcasts over $\mathcal{T}$ have termination detection: (i) $\mathcal{L}$ needs to be made aware of how many base fragments are present and their IDs and (ii) each node in $\mathcal{T}$ needs routing information related to any fragment roots located in the subtree rooted at that node in $\mathcal{T}$.[10]

We need each fragment F to inform $\mathcal{L}$ of its existence and fragment ID. Now, the root of each fragment F, with ID ID_F, initiates **Tree-Count** to determine the number of nodes in the fragment, $size_F$. (Procedure **Tree-Count** is described in Appendix A.) Subsequently, all nodes in the graph participate in Procedure **Upcast** over $\mathcal{T}$ where each base fragment's root sends up the tuple $\langle ID_F, size_F\rangle$.[11] $\mathcal{L}$ accumulates these messages until $\sum_F size_F = n$, at which point $\mathcal{L}$ knows the exact number of base fragments, say NUM-OF-BASE-FRAGMENTS, and their IDs. Once $\mathcal{L}$ recognizes that it has received all the messages, it initiates a broadcast of NUM-OF-BASE-FRAGMENTS over $\mathcal{T}$. Now all nodes are aware of the number of base fragments.

Stage III. In this stage, each node u maintains two sets of variables. One set of variables relates to the base fragment B node u it belongs to at the end of phase two. These variables store information about the base fragment such as the base fragment ID ID_B, u's parents in B, and u's children in B. The second set of variables relates to what we term a *cluster*, a connected subgraph in $\mathcal{H}$ consisting of base fragments and MOEs between them, and they store information that includes a cluster ID and cluster edges. Each node belonging to base fragment B initially sets its cluster ID $\mathsf{CLUSTER\text{-}ID}_B$ to be the same as its base fragment ID. Each node u also stores a set of cluster edges adjacent to it in the set $\mathsf{CLUSTER\text{-}EDGES}_u$, which is initially empty. Edges are added to $\mathsf{CLUSTER\text{-}EDGES}_u$ in the course of stage III. At the end of stage III, for a given node u, the set of edges in the MST is the union of the set of edges in $\mathsf{CLUSTER\text{-}EDGES}_u$ and its children and parent in B. Node $\mathcal{L}$ maintains, in addition, information on the supergraph $\mathcal{H}$ formed by the base fragments (including the updated cluster IDs of those base fragments) and any MOE edges that $\mathcal{L}$ computes in the phases of stage III, to be described below.

In stage III, each node participates in the following process for $\lceil \log_2 n \rceil$ phases until it terminates. Once again, nodes use a β-synchronizer over $\mathcal{T}$ to keep track of the phase number in stage III. In each phase, each base fragment B with root R_B, fragment ID ID_B, and cluster ID $\mathsf{CLUSTER\text{-}ID}_B$ runs Procedure **Find-MOE** to find its minimum outgoing edge, say $\mathsf{MOE\text{-}VALUE}_B$, to a node with a different cluster ID, if there is any. All nodes in the graph then participate in Procedure **Upcast** over $\mathcal{T}$ to send informatino on the fragments up to $\mathcal{L}$. Specifically, each base fragment B's root sends up the tuple consisting of information on B as well as the computed MOE, if any.

Once $\mathcal{L}$ receives this tuple from all base fragments, it locally computes the MOE edges for each cluster in the supergraph $\mathcal{H}$. Recall that a cluster is a connected subgraph of base fragments in $\mathcal{H}$. Thus, the MOE from a cluster is really an MOE from one of the base

[9] As we use a β-synchronizer to keep track of which phase a node is in, it is possible to know when $\max\{\lceil \log_2 \sqrt{n} \rceil, \lceil \log_2 D' \rceil\}$ phases are over.

[10] Consider a node u and let node v be the root of a fragment located in the subtree rooted at u in $\mathcal{T}$. We say node u has routing information on v when u knows which of its children in $\mathcal{T}$ to send a message destined for v

[11] It is important to note that during Procedure **Upcast**, each node u in $\mathcal{T}$ learns about which of its children in $\mathcal{T}$ lead to which fragment roots. In other words, u learns routing information related to any fragments roots located in the subtree in $\mathcal{T}$ rooted at u, satisfying our second requirement from the previous paragraph.

fragments that constitutes it. Define $\mathsf{FINAL\text{-}MOE\text{-}VALUE}_B$ as the MOE, if any, for base fragment B. For each base fragment B, $\mathcal{L}$ computes its new cluster ID $\mathsf{CLUSTER\text{-}ID}_B$ (if multiple clusters merge, the smallest cluster ID becomes the ID of the new merged cluster), and its $\mathsf{FINAL\text{-}MOE\text{-}VALUE}_B$ (if the original value of $\mathsf{FINAL\text{-}MOE\text{-}VALUE}_B$ broadcast by B was selected as a new edge in $\mathcal{H}$, $\mathsf{FINAL\text{-}MOE\text{-}VALUE}_B$ is set to $\mathsf{MOE\text{-}VALUE}_B$, else it is set to a null value).

All nodes participate in Procedure **Downcast** so that $\mathcal{L}$ may inform each base fragment's root about its possibly new cluster ID and MOE edge. (Procedure **Downcast** is described in Appendix A.) Subsequently each base fragment participates in Procedure **Frag-Bcast** to send these values to all nodes in the fragment. Each node updates its cluster ID if needed. If there is information on a new MOE edge out of one of the nodes u, then u adds this edge to $\mathsf{CLUSTER\text{-}EDGES}_u$. Once the final phase of stage III is complete, all nodes terminate the algorithm.

4 Analysis of the MST Algorithm

We argue that Algorithm **Sing-MST** correctly outputs the MST with high probability and subsequently analyze its running time and message complexity.

It is easy to see that the algorithm faithfully simulates **Controlled-GHS** in the asynchronous setting. Recall that **Controlled-GHS** requires us to maintain two properties in each phase of the algorithm: (i) at the end of phase i, there are at most $n/2^i$ fragments and (ii) at the end of phase i, each fragment has diameter $O(2^i)$. Since the algorithm faithfully simulates **Controlled-GHS**, it follows from the analysis of **Controlled-GHS** (see e.g., [18, 47]) that these properties are maintained in stage II. In stage III, they are also maintained via the "soft merge" process in a way that is time and message efficient. Note that in stage III, we ensure that those properties hold now on clusters instead of on fragments. These two properties guarantee that after the algorithm is over, there exists one cluster such that all nodes belong to the cluster and the only edges in the cluster are MST edges of the original graph. The high probability guarantee comes from the usage of (randomized) Procedures **LE** and **ST-Cons**.

We now bound the running time and message complexity in each stage of the algorithm. Due to a lack of space, the proofs of the following lemmas are deferred to the full version.

▶ **Lemma 6.** *Stage I of Algorithm* ***Sing-MST*** *takes $\tilde{O}(D^{1+\varepsilon})$ time with high probability and $\tilde{O}(m)$ messages with high probability, for any constant $\varepsilon > 0$.*

▶ **Lemma 7.** *Stage II takes $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time and $\tilde{O}(m)$ messages.*

▶ **Lemma 8.** *Stage III takes $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time and $\tilde{O}(m)$ messages to complete.*

By Lemmas 6, 7, and 8 and our initial discussion about correctness, we get the following theorem.

▶ **Theorem 9.** *Algorithm* ***Sing-MST*** *computes the minimum spanning tree of an arbitrary graph with high probability in the asynchronous KT_0 $\mathcal{CONGEST}$ model in $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time with high probability and $\tilde{O}(m)$ messages with high probability. Furthermore, nodes know their edges in the MST and terminate when the algorithm is over.*

As a consequence of the above theorem and a theorem due to Mashregi and King [44, Theorem 1.2] we also get the following result in the KT_1 model.

▶ **Theorem 10.** *There is an asynchronous algorithm that computes the minimum spanning tree of an arbitrary graph with high probability in the asynchronous KT_1 $\mathcal{CONGEST}$ model in $\tilde{O}(D^{1+\varepsilon} + n^{1-2\delta})$ time and $\tilde{O}(n^{3/2+\delta})$ messages for any small constant $\varepsilon > 0$ and for any $\delta \in [0, 0.25]$.*

The above theorem gives the first asynchronous MST algorithm in the KT_1 $\mathcal{CONGEST}$ model that has *sublinear time* (for all $D = O(n^{1-\varepsilon'})$ for any arbitrarily small constant $\varepsilon' > 0$) and *sublinear* messages complexity.

5 Conclusion and Open Problems

Recall that while most of the paper deals with the common KT_0 model, Theorem 10 includes a contribution also under the KT_1 model. This model has grown in popularity in recent years first because one can claim it is a more natural model [8] and second because it allows reducing the communication to $o(m)$. Initially, it looked as if this reduction carries a significant cost in time complexity, trading off the attempt to go below $\Omega(n)$ when the diameter is smaller [33]. This went against the direction for the KT_0 model, where algorithms managed to be efficient both in time complexity and message complexity [18, 47, 28, 26]. Those results, however, were in the synchronous model. Theorem 10 (together with [44][Theorem 1.2]) is the first result that approaches optimal time while keeping message complexity $o(m)$. It would be interesting to see whether this is the best that can be obtained in this direction. Results showing that other tasks can be obtained with $o(m)$ messages but time efficiently in KT_1 would also be interesting.

The asynchronous distributed MST algorithm for KT_0 presented here continues a long line of work in distributed MST algorithms. Our algorithm essentially (up to a polylog(n) factor) matches the respective time and message lower bounds, but for an arbitrarily small constant factor ε in the exponent of D (with respect to time). Yet, several open problems remain. Is it possible to achieve near singular optimality? That is, can we achieve optimality within a polylog(n) factor in both time and messages? This seems related to constructing a $\tilde{O}(D)$ diameter spanning tree in a singularly optimal fashion which is also open. Our low-diameter spanning tree construction comes close to achieving this, but for a $\tilde{O}(D^{\varepsilon})$ factor in the diameter and run time. This is also closely related to constructing a BFS (or nearly BFS) tree in a singularly optimal fashion.

The tools and techniques used in this paper for accomplishing various tasks in a (almost) singularly optimal fashion in an asynchronous setting can also be useful in solving other fundamental problems such as shortest paths, minimum cut etc. In particular, the techniques of this paper can be useful in showing that the partwise aggregation operation of Ghaffari and Haeupler [25] can be implemented in the asynchronous setting in $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ and $\tilde{O}(m)$ messages. This would imply that problems such as exact minimum cut and $(1+\varepsilon)$-single source shortest path can be solved almost singularly optimally. We will elaborate on these in more detail in the full version of the paper.

For our singularly optimal algorithms we focused on being (existentially) optimal in time with respect to parameters n and D (i.e., with respect to the $\tilde{\Omega}(D + \sqrt{n})$ bound). An interesting direction of future work is obtaining asynchronous algorithms that are "universally optimal" (Haeupler, Wajc, and Zuzic [32]) (with respect to time) and also optimal with respect to messages.

References

1 Yehuda Afek and Eli Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. *SICOMP*, 20(2):376–394, 1991.

2 Yehuda Afek and Yossi Matias. Elections in anonymous networks. *Information and Computation*, 113(2):312–330, 1994.

3 John Augustine, Seth Gilbert, Fabian Kuhn, Peter Robinson, and Suman Sourav. Latency, capacity, and distributed minimum spanning tree. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 157–167. IEEE, 2020.

4 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.

5 Baruch Awerbuch. 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)*, pages 230–240, 1987.

6 Baruch Awerbuch. Distributed shortest paths algorithms (extended abstract). In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 490–500, 1989.

7 Baruch Awerbuch, Oded Goldreich, Ronen Vainish, and David Peleg. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37:238–256, 1990.

8 Baruch Awerbuch, Oded Goldreich, Ronen Vainish, and David Peleg. A trade-off between information and communication in broadcast protocols. *Journal of the ACM (JACM)*, 37(2):238–256, 1990.

9 Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *31st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 514–522, 1990.

10 Yi-Jun Chang, Varsha Dani, Thomas P. Hayes, Qizheng He, Wenzheng Li, and Seth Pettie. The energy complexity of broadcast. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 95–104, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3212734.3212774.

11 Yi-Jun Chang, Varsha Dani, Thomas P. Hayes, and Seth Pettie. The energy complexity of bfs in radio networks. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 273–282, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405713.

12 Yogen K Dalal. *A Distributed Algorithm for Constructing Minimal Spanning Trees in Computer-Communication Networks.* Stanford University, 1976.

13 Yogen K. Dalal. A distributed algorithm for constructing minimal spanning trees. *IEEE Trans. Software Eng.*, 13(3):398–405, 1987.

14 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012.

15 Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms.* Cambridge University Press, 2009. URL: http://www.cambridge.org/gb/knowledge/isbn/item2327542/.

16 Michael Elkin. A faster distributed protocol for constructing minimum spanning tree. *Journal of Computer and System Sciences*, 72(8):1282–1308, 2006.

17 Michael Elkin. An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. *SIAM J. Comput.*, 36(2):433–456, 2006.

18 Michael Elkin. A simple deterministic distributed MST algorithm, with near-optimal time and message complexities. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 157–163, 2017.

19 Michael Elkin, Hartmut Klauck, Danupon Nanongkai, and Gopal Pandurangan. Can quantum communication speed up distributed computation? In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 166–175. ACM, 2014.

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

21 Pierre Fraigniaud, Amos Korman, and Emmanuelle Lebhar. Local mst computation with short advice. *Theory of Computing Systems*, 47(4):920–933, 2010.

22 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

23 Juan A. Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comput.*, 27(1):302–316, 1998.

24 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, mst, and min-cut. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 202–219. SIAM, 2016.

25 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, MST, and min-cut. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 202–219, 2016.

26 Mohsen Ghaffari and Fabian Kuhn. Distributed MST and broadcast with fewer messages, and faster gossiping. In *Proceedings of the 32nd International Symposium on Distributed Computing (DISC)*, pages 30:1–30:12, 2018.

27 Robert Gmyr and Gopal Pandurangan. Time-message trade-offs in distributed algorithms. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 32:1–32:18, 2018.

28 Robert Gmyr and Gopal Pandurangan. Time-message trade-offs in distributed algorithms. In *Proceedings of the 32nd International Symposium on Distributed Computing (DISC)*, pages 32:1–32:18, 2018.

29 Sandeep KS Gupta and Pradip K Srimani. Self-stabilizing multicast protocols for ad hoc networks. *Journal of Parallel and Distributed Computing*, 63(1):87–96, 2003.

30 Bernhard Haeupler, D. Ellis Hershkowitz, and David Wajc. Round-and message-optimal distributed graph algorithms. In *PODC*, pages 119–128, 2018.

31 Bernhard Haeupler and David Wajc. A faster distributed radio broadcast primitive: Extended abstract. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 361–370, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2933057.2933121.

32 Bernhard Haeupler, David Wajc, and Goran Zuzic. Universally-optimal distributed algorithms for known topologies. In *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1166–1179. ACM, 2021.

33 Valerie King, Shay Kutten, and Mikkel Thorup. 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)*, pages 71–80, 2015.

34 Liah Kor, Amos Korman, and David Peleg. Tight bounds for distributed MST verification. In *Proc. 28th Symp. on Theoretical Aspects of Computer Science (STACS)*, volume 9 of *LIPIcs*, pages 69–80. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011.

35 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Computing*, 20(4):253–266, 2007.

36 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. In *Proc.24th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 9–18, 2005.

37 Shay Kutten, William K. Moses Jr., Gopal Pandurangan, and David Peleg. Singularly near optimal leader election in asynchronous networks. In *35th International Symposium on Distributed Computing (DISC)*, pages 27:1–27:18, 2021.

38 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the complexity of universal leader election. *J. ACM*, 62(1), 2015.

39 Shay Kutten and David Peleg. Fast distributed construction of small k-dominating sets and applications. *J. Algorithms*, 28(1):40–66, 1998.

40 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $O(\log \log n)$ communication rounds. *SIAM J. Comput.*, 35:120–131, 2005.

41 Zvi Lotker, Boaz Patt-Shamir, and David Peleg. Distributed MST for constant diameter graphs. In *Proc. 20th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 63–71, 2001.

42 Ali Mashreghi and Valerie King. Time-communication trade-offs for minimum spanning tree construction. In *Proceedings of the 18th International Conference on Distributed Computing and Networking (ICDCN)*, 2017.

43 Ali Mashreghi and Valerie King. Broadcast and minimum spanning tree with o(m) messages in the asynchronous CONGEST model. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 37:1–37:17, 2018.

44 Ali Mashreghi and Valerie King. Brief announcement: Faster asynchronous MST and low diameter tree construction with sublinear communication. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 49:1–49:3, 2019.

45 Ali Mashreghi and Valerie King. Broadcast and minimum spanning tree with o(m) messages in the asynchronous CONGEST model. *Distributed Computing*, pages 1–17, 2021.

46 Gary L. Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 196–203, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2486159.2486180.

47 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A time- and message-optimal distributed algorithm for minimum spanning trees. In *Proceedings of the 49th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 743–756, 2017.

48 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. The distributed minimum spanning tree problem. *Bulletin of the EATCS*, 125, 2018.

49 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A time- and message-optimal distributed algorithm for minimum spanning trees. *ACM Transactions on Algorithms (TALG)*, 16(1):1–27, 2019.

50 David Peleg. *Distributed Computing: A Locality Sensitive Approach.* SIAM, 2000.

51 David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM J. Comput.*, 30(5):1427–1442, 2000.

52 Deepak Rohilla, Mahendra Kumar Murmu, and Shashidhar Kulkarni. An efficient distributed approach to construct a minimum spanning tree in cognitive radio network. In *First International Conference on Sustainable Technologies for Computational Intelligence*, pages 397–407. Springer, 2020.

53 Baruch Schieber and Marc Snir. Calling names on nameless networks. *Information and Computation*, 113(1):80–101, 1994.

54 Adrian Segall. Distributed network protocols. *IEEE transactions on Information Theory*, 29(1):23–35, 1983.

55 Gurdip Singh. Efficient leader election using sense of direction. *Distributed Computing*, 10(3):159–165, 1997. doi:10.1007/s004460050033.

56 Philip Spira. Communication complexity of distributed minimum spanning tree algorithms. In *Proceedings of the second Berkeley conference on distributed data management and computer networks*, 1977.

57 Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

58 Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

A Toolbox

In this section, we present several procedures that are used as blackboxes in the current paper. As these procedures are either from other papers or minor variations of those in other papers, we merely mention what they do and their guarantees here.

Synchronization

Synchronizers are mechanisms that allow nodes to run synchronous algorithms in an asynchronous network with some overhead, either in time or messages.

α-synchronizer. An *alpha*-synchronizer, presented by Awerbuch [4], is a well known mechanism for nodes to run synchronous algorithms in an asynchronous network in the same running time (with a diameter overhead to time) while suffering a message overhead equivalent to the product of the run time of the synchronous algorithm and $O(m)$. Informally, when simulating some synchronous algorithm Alg, each node v sends a "pulse" message to all its neighbors after all of v's messages in the current round of Alg were acknowledged. Thus, v's neighbors can keep track of which pulse, or "clock tick", v has simulated. Additionally, note that it takes $O(D)$ time to initialize the α-synchronizer. A good description appears also in [50]. We know the following about an α-synchronizer.

▶ **Lemma 11** (Adapted from [50]). *Consider a graph G with n nodes, m edges, and diameter D in an asynchronous setting. The nodes of the graph may simulate a synchronous algorithm that takes $O(T)$ rounds and $O(M)$ messages in the synchronous setting by utilizing an α-synchronizer. The resulting simulated algorithm takes $O(T + D)$ time and $O(M + Tm)$ messages and has termination detection.*

β-synchronizer. A β-synchronizer is another type of synchronizer that reduces the message overhead at the expense of time. An assumption is made that there exists a spanning tree $\mathcal{T}$, rooted at some node $\mathcal{L}$, of depth d overlaid on top of the original graph and that each node knows its parent and children in the tree, if any. Now, as with the α-synchronizer, a synchronous algorithm that takes $O(T)$ rounds and $O(M)$ messages may be simulated in an asychronous network with the help of pulses. However, here each node sends a pulse to its parent once the current round is done and it has received pulses from each of its children in the tree. Once the root receives the pulse and finishes the current round, it broadcasts a message to move to the next round along the tree. The resulting simulated algorithm takes $O(T \cdot d)$ time and $O(M + Tn)$ messages.

▶ **Lemma 12** (Adapted from [50]). *Consider a graph G with n nodes in an asynchronous setting. Assume that there exists a rooted spanning tree $\mathcal{T}$ of depth d overlaid on G such that each node knows its parent and children, if any, in the tree. The nodes of the graph may simulate a synchronous algorithm that takes $O(T)$ rounds and $O(M)$ messages in the synchronous setting by utilizing a β-synchronizer over $\mathcal{T}$. The resulting simulated algorithm takes $O(T \cdot d)$ time and $O(M + Tn)$ messages and has termination detection.*

Notice that both α- and β-synchronizers can be used by nodes to enact a type of global round counter up to any number that can be encoded using $O(\log n)$ bits.

Leader Election

We make use of the leader election procedure, call it Procedure **LE**, of Kutten et al. [37] to elect a leader with high probability. Adapting Theorem 11 to this setting, we have the following lemma. Note that in the course of the procedure, all nodes are woken up but such information was not mentioned in the theorem statement in [37], so we add it here.

▶ **Lemma 13** (Theorem 11 in [37]). *Procedure **LE** solves leader election with termination detection with high probability in any arbitrary graph with n nodes, m edges, and diameter D in $O(D + \log^2 n)$ time with high probability using $O(m \log^2 n)$ messages with high probability in an asynchronous system with adversarial node wake-up. At the end of the procedure, all nodes are awake.*

Operations on a Fragment

In the course of our algorithm, we reach a situation where the graph G is partitioned into a set of disjoint trees (called fragments), each with a distinct root, an associated fragment ID, and an associated cluster ID (which may be different from its fragment ID). Each node knows its parent and children in the fragment, if any. We now describe some common operations that are to be performed on such trees.

Consider a tree T spanning a subset of the nodes of G, oriented towards a distinct root R. Let the tree have fragment ID F, known to all nodes in T. Furthermore, all nodes of T have the same cluster ID, say C, which may or may not be equal to F. Let $\mathit{size}(T)$ and $\mathit{depth}(T)$ denote the number of vertices and the depth of T, respectively.

Broadcast on a Fragment. Suppose a message M, originating at the root R, must be distributed to all nodes of the tree. Procedure **Frag-Bcast** performs this operation in a straightforward manner. The root R sends M to all its neighbors. Intermediate nodes receiving M on some round forward it to all their children in T in the next round.

To ensure termination detection, the procedure then performs a *convergecast* of acknowledgements on T as follows. Each leaf, upon receiving M, sends back an “ack” message. Each intermediate node waits until it receives an “ack” from all its children, and then sends an “ack” to its parent. The operation terminates once the root receives an “ack” from all its children.

▶ **Lemma 14.** *Procedure **Frag-Bcast**, run by nodes in the tree T, performs broadcast of a message originating at the root of T with termination detection in $O(\mathit{depth}(T))$ time and $O(\mathit{size}(T))$ messages.*

Upcast on a Fragment. Suppose m distinct and uncombinable messages, originating at arbitrary locations in the tree, must be gathered to the root R. Procedure **Upcast** performs this operation in a straightforward manner. Each node in the tree pipelines the messages it has seen upwards in the tree (towards R), in some arbitrary order.

We assume that R knows the number m of such messages it expects to receive and ensure this is true everywhere the procedure is called. Thus, R knows when it has received all m messages. To ensure termination detection, the procedure then performs **Frag-Bcast**.

▶ **Lemma 15.** *Procedure **Upcast**, run by nodes in the tree T, performs upcasting of m distinct messages with termination detection in $O(m + \mathit{depth}(T))$ time and $O(m \cdot \mathit{depth}(T))$ messages.*

Downcast on a Fragment. Suppose m distinct and uncombinable messages $M_1, \ldots, M_m$, originating at the root R, must be distributed to arbitrary destinations $w_1, \ldots, w_m$ in the tree, respectively. Procedure **Downcast** performs this operation in a straightforward manner. In each round i, R sends the pair (M_i, w_i) to its neighbor on the unique R-w_i path in T. Intermediate nodes receiving a pair (M_i, w_i) on some round forward it towards w_i in the next round. (Note that tie-breaking is not required.)

To ensure termination detection, the procedure then performs a convergecast of acknowledgements, backtracking on the subtree T' marked by the downcast messages; namely, each intermediate node that received ℓ messages from its parent and forwarded ℓ_j messages to its child x_j expects "ack - ℓ_j" from x_j. After receiving all such "ack" messages from its children, it sends "ack - ℓ" to its parent. The root detects termination upon receiving "ack" messages from all relevant children.

▶ **Lemma 16.** *Procedure **Downcast**, run by nodes in the tree T, performs downcasting of m distinct messages with termination detection in $O(m + depth(T))$ time and $O(m \cdot depth(T) + size(T))$ messages.*

Finding MOE of a Fragment. Informally, minimum outgoing edge (MOE) out of T is the least weight edge out of T to a node with a different cluster ID (i.e, $\neq C$). Formally, it is a tuple $\langle u, v, C, C' \rangle$ such that edge (u, v) is the MOE from T where $u \in T$ with cluster ID C and $v \notin T$ with cluster ID $C' (\neq C)$. Note that nodes not belonging to T but adjacent to T may have the same cluster ID C as the nodes of T, and as such it is possible for T to not have any MOE. Yet another application of Wave&Echo, taken from the algorithm of [22], results in R being made aware of the MOE of T if such exists. Let us call this module procedure **Find-MOE**.

▶ **Lemma 17.** *Procedure **Find-MOE**, when run by the nodes of a tree T with distinct root R, and cluster ID C, results in R knowing the minimum outgoing edge from T, if one exists, where only edges to nodes with a cluster ID $\neq C$ are considered outgoing edges, in $O(depth(T))$ time and $O(\sum_{u \in T} deg(u))$ messages, where $depth(T)$ is the depth of T and $deg(u)$ is the degree of node u. Furthermore, every node participating in procedure **Find-MOE** can detect termination.*

Size Calculation of a Fragment. We make use of a known tool (essentially a known application of Wave&Echo, see PIF in [54]), to be run by the nodes of the tree and result in R being made aware of how many nodes (including itself) belong to T. Let us call this Procedure **Tree-Count**.

▶ **Observation 18.** *Procedure **Tree-Count**, when run by the nodes of a tree T with distinct root R, results in R knowing the total number of nodes in T in $O(depth(T))$ time and $O(size(T))$ messages, where $depth(T)$ is the depth of T and $size(T)$ is the number of nodes in T. Furthermore, nodes participating in procedure **Tree-Count** can detect termination.*

Diameter Calculation of a Fragment. Another known application of Wave&Echo allows R to calculate the diameter of the tree T, let us call that Procedure **Diam-Calc**.

▶ **Observation 19.** *Procedure **Diam-Calc**, when run by the nodes of a tree T with distinct root R, results in R knowing the diameter of T in $O(depth(T))$ time and $O(size(T))$ messages, where $depth(T)$ is the depth of T and $size(T)$ is the number of nodes in T. Furthermore, nodes participating in procedure **Diam-Calc** can detect termination.*

B Low Diameter Spanning Tree - Relegated Proofs

We first provide definitions and an auxiliary lemma (see Lemma 20) followed by proofs of Lemmas 1 and 2, stated in Section 2.1. After which, we provide the proofs of Lemma 4 and Corollary 5, stated in Section 2.2.

Consider some fixed execution of the algorithm and node $v \in \overline{V}$. Then $D_u = S_u + dist(u,v) - 1 = \delta_{max} - \lfloor \delta_u \rfloor + dist(u,v) - 1$ denotes the *(arrival) round* of u, that is, the first round in which v can receive a message from u's cluster. For every integer $1 \leq j \leq n$, let z_j be the node with the jth smallest arrival round in the execution. For every integer $1 \leq k \leq n$, let $S_k = \{z_1, \ldots, z_k\}$. Building upon these definitions, for a node $v \in \overline{V}$, positive integers $1 \leq k, r \leq n$, let $\mathcal{E}_{v,k,r}$ denote the event that after the execution of the algorithm, $D_{z_{k+1}} - D_{z_1} \leq r$.

▶ **Lemma 20.** *For any node $v \in \overline{V}$ and positive integers $1 \leq k, r \leq n$,*

$$\Pr(\mathcal{E}_{v,k,r}) \;\leq\; (1 - \exp(-(r+1)\beta))^k$$

Proof. We condition on S_k and $D^* = D_{z_{k+1}}$. The proof is based on first showing the stated upper bound on the probability of $\mathcal{E}_{v,k,r}$ conditioned on S_k and D^*, and then applying the law of total probability to derive the lemma statement. We next describe the first half of the proof in more detail.

For any integer $i \geq 1$, let $c_{z_i} = \delta_{max} + dist(z_i, v) - 1$. We have $\Pr(\mathcal{E}_{v,k,r} \mid S_k, D^*) \leq p$ for

$$p \;=\; \Pr\left(\bigwedge_{i=1}^{k} [D^* - D_{z_i} \leq r]\right) \;=\; \Pr\left(\bigwedge_{i=1}^{k} [\delta_{z_i} \leq r + 1 + c_{z_i} - D^*]\right) \;=\; \prod_{i=1}^{k} \Pr(\delta_{z_i} \leq r+1+c_{z_i} - D^*),$$

where the last equality holds since the random variables δ_{z_i} are independent. Next, note that $D^* \geq D_{z_i}$ for any integer $1 \leq i \leq k$, and thus $\Pr(\lfloor \delta_{z_i} \rfloor \geq c_{z_i} - D^*) = 1$. Hence, $\Pr(\delta_{z_i} \geq c_{z_i} - D^*) = 1$ and

$$p \;=\; \prod_{i=1}^{k} \Pr(\delta_{z_i} \leq r + 1 + c_{z_i} - D^* \mid \delta_{z_i} \geq c_{z_i} - D^*).$$

Finally,

$$p \;\leq\; \prod_{i=1}^{k} \Pr(\delta_{z_i} \leq r+1) \;=\; \prod_{i=1}^{k} (1 - \exp(-(r+1)\beta)) \;=\; (1 - \exp(-(r+1)\beta))^k$$

where the inequality holds by the memorylessness of the exponential distribution. ◀

Proof of Lemma 1. We first note for any node $v \in \overline{V}$, $\Pr[\lfloor \delta_v \rfloor > \delta_{max}] = \Pr[\delta_v > \frac{2 \ln n}{\beta}] = \exp(-2 \ln n) = \frac{1}{n^2}$. Hence, by union bound, $\lfloor \delta_v \rfloor \leq \delta_{max}$ for every node $v \in \overline{V}$ with high probability. We hereafter exclude this unlikely event and assume $\delta_{max} \geq \max_{v \in \overline{V}} \{\lfloor \delta_v \rfloor\}$. This implies that all nodes belong to a cluster.

Next, note that by the algorithm description, each cluster is spanned by a tree of depth at most $\frac{2 \ln n}{\beta}$. Hence, all clusters have strong diameter at most $\frac{4 \ln n}{\beta}$. Finally, an edge is cut if its two endpoints u and v are in different clusters. This implies that for node v (without loss of generality), the two smallest arrival rounds differ by at most 1, which corresponds to event $\mathcal{E}_{v,1,1}$. By Lemma 20, $\Pr(\mathcal{E}_{v,1,1}) \leq (1 - \exp(-2\beta)) \leq 2\beta$. The lemma follows. ◀

Proof of Lemma 2. Again, we assume $\delta_{max} \geq \max_{u \in \overline{V}} \{\delta_u\}$, which holds with high probability. For any node $v \in \overline{V}$, let C_v denote the cluster containing v after the execution of the algorithm.

Consider any two nodes $u, v \in \overline{V}$ such that $l = dist_{\overline{G}}(u, v) > 3\beta\overline{D}$. (Note that if $l \le 3\beta\overline{D}$, then $dist_{\overline{G}^*}(C_u, C_v) \le 3\beta\overline{D}$.) Let $(w_1, \ldots, w_{l+1})$ be the shortest path between u and v in $\overline{G}$ (where $w_1 = u$ and $w_{l+1} = v$). Moreover, for any integer $i \in [1, l]$, let X_i be the indicator random variable of w_i and w_{i+1} being in the same cluster. Then, the random variable $X = \sum_{i=1}^{l} X_i$ is an upper bound on $dist_{\overline{G}^*}(C_u, C_v)$. By Lemma 1, each edge is an inter-cluster edge with probability at most 2β. Hence, by the linearity of expectation, $E[X] \le 2\beta l$.

Next, let us provide a concentration bound for X by showing that the random variables X_i are only locally dependent. First, for any two integers $i, j \in [1, l]$ such that $|i - j| > \lfloor 4\frac{\ln n}{\beta} \rfloor$, X_i and X_j are independent (since the same node cannot affect w_i and w_j with our choice of δ_{max}). Then, we can color the random variables $\{X_i\}_{i=1,\ldots,l}$ using $\chi = \lfloor 4\frac{\ln n}{\beta} \rfloor$ – by coloring X_i with $i \mod (\chi + 1)$ – such that variables with the same color are independent. In other words, the random variables X_i are only locally dependent and thus we can apply a specific Chernoff-Hoeffding bound (Theorem 3.2 from [15]): $\Pr(X \ge E[X] + t) \le \exp(-2t^2/(\chi \cdot l))$. Hence, $\Pr(X \ge 3\beta l) \le \exp(-2(\beta l)^2/(\chi \cdot l)) \le \exp(-2\beta^2 l/\chi)$. Since $l > 3\beta\overline{D} > 3k\frac{\ln^2 n}{\beta^3}$, $\Pr(X \ge 3\beta l) \le \exp(-\frac{3}{2}k \ln n) \le \frac{1}{n^k}$. By taking a union bound over all n^2 possible pairs of nodes $u, v \in \overline{V}$, the lemma statement follows. ◀

Proof of Lemma 4. By induction on i. The base case, $i = 1$, holds trivially.

Next, consider some $i \ge 1$ for which the inductive hypothesis holds, i.e., $diam(G_{i-1}) = \max\{(3\beta)^{i-1}D, O(\log^{2+4/\varepsilon'} n)\}$ w.h.p. and each cluster node C of the partition $\mathcal{P}_{i-1}$ is spanned (in the original graph G) by a tree $\hat{T}(C)$ with $diam(\hat{T}(C)) = (\frac{5\ln n}{\beta})^{i-1}$. Running Procedure **MPX** on G_{i-1} yields a $(2\beta, \frac{4\ln n}{\beta})$ low-diameter decomposition of G_{i-1}. In fact, each super cluster C' of this decomposition on G_{i-1} is spanned (in the cluster graph G_{i-1}) by a tree $\hat{T}(C')$ of diameter $\frac{4\ln n}{\beta}$. Hence, the "combined" spanning tree computed by Procedure **Transform** for the "analog" C'' of cluster C' on G, which is a cluster of the newly constructed G_i, has diameter $diam(\hat{T}(C'')) = (\frac{4\ln n}{\beta} + 1) \cdot (\frac{5\ln n}{\beta})^{i-1} \le (\frac{5\ln n}{\beta})^{i}$. Next, the diameter of G_i is the same as that of the cluster graph H induced by partition $\mathcal{P}_i^*$. By Lemma 2, the diameter of H is $\max\{(3\beta)^{i}D, O(\log^{2+4/\varepsilon'} n)\}$ w.h.p., and thus the lemma statement holds. ◀

Proof of Corollary 5. By Lemma 4 (and applying one extra induction step), the diameter of G_{i_m} is $D_f = \max\{(3\beta)^{i_m}D, O(\log^{2+4/\varepsilon'} n)\}$ and each cluster C of the partition $\mathcal{P}_{i_m}$ is spanned in G by a tree $\hat{T}(C)$ of depth $d_f = (\frac{5\ln n}{\beta})^{i_m}$. Since $i_m = \lceil \log_{1/(3\beta)} D \rceil$, we have that $(3\beta)^{i_m} \le 1/D$, so $D_f = O(\log^{2+4/\varepsilon'} n)$. Moreover, by going through the computations, we get:

$$\begin{aligned}
d_f = \;& \exp(i_m \ln(5\ln^{1+1/\varepsilon'} n)) \;\le\; (5\ln^{1+1/\varepsilon'} n) \exp\left(\frac{\ln D \ln(5\ln^{1+1/\varepsilon'} n)}{\ln(\frac{1}{3}\ln^{1/\varepsilon'} n)}\right) \\
= \;& (5\ln^{1+1/\varepsilon'} n) \exp\left(\ln D \cdot \frac{\ln 5 + (1 + 1/\varepsilon')\ln\ln n}{\frac{1}{\varepsilon'}\ln\ln n - \ln 3}\right) \\
= \;& (5\ln^{1+1/\varepsilon'} n) \exp\left(\ln D \cdot \left(1 + \frac{\ln 5 + \ln 3 + \ln\ln n}{\frac{1}{\varepsilon'}\ln\ln n - \ln 3}\right)\right) \\
\le \;& (5\ln^{1+1/\varepsilon'} n) \exp(\ln D \cdot (1 + 2\varepsilon' \ln 15)) \;\le\; (5\ln^{1+1/\varepsilon'} n)\; D^{1+\varepsilon} \;,
\end{aligned}$$

where, in order to make the last inequality hold, Procedure **ST-Cons**(ε) selects $\varepsilon' \le \varepsilon/(2\ln 15)$. ◀