

A Near-Optimal Deterministic Distributed Synchronizer

Mohsen Ghaffari
MIT
ghaffari@mit.edu

Anton Trygub
MIT
trygub@mit.edu

Abstract

We provide the first deterministic distributed synchronizer with near-optimal time complexity and message complexity overheads. Concretely, given any distributed algorithm \mathcal{A} that has time complexity T and message complexity M in the synchronous message-passing model (subject to some care in defining the model), the synchronizer provides a distributed algorithm \mathcal{A}' that runs in the asynchronous message-passing model with time complexity $T \cdot \text{poly}(\log n)$ and message complexity $(M + m) \cdot \text{poly}(\log n)$. Here, n and m denote the number of nodes and edges in the network, respectively. The synchronizer is deterministic in the sense that if algorithm \mathcal{A} is deterministic, then so is algorithm \mathcal{A}' . Previously, only a randomized synchronizer with near-optimal overheads was known by seminal results of Awerbuch, Patt-Shamir, Peleg, and Saks [STOC 1992] and Awerbuch and Peleg [FOCS 1990]. We also point out and fix some inaccuracies in these prior works.

As concrete applications of our synchronizer, we resolve some longstanding and fundamental open problems in distributed algorithms: we get the first asynchronous deterministic distributed algorithms with near-optimal time and message complexities for leader election, breadth-first search tree, and minimum spanning tree computations: these all have message complexity $\tilde{O}(m)$ message complexity. The former two have $\tilde{O}(D)$ time complexity, where D denotes the network diameter, and the latter has $\tilde{O}(D + \sqrt{n})$ time complexity. All these bounds are optimal up to logarithmic factors. Previously all such near-optimal algorithms were either restricted to the synchronous setting or required randomization.

Contents

1	Introduction and Related Work	3
1.1	Models and the synchronizer definition	3
1.2	State of the art	4
1.3	Our contribution	6
1.3.1	A near-optimal deterministic distributed synchronizer.	6
1.3.2	Inaccuracy or correctness issues.	7
2	Preliminaries	7
2.1	Sparse covers	8
2.2	Running several algorithms in the asynchronous environment	8
2.2.1	Running in parallel	8
2.2.2	Running sequentially	9
3	Collecting information in covers	10
3.1	Collecting information in $(d \cdot \ell)$ -ball with sparse d -cover	10
3.2	Registration in clusters	11
3.2.1	The Abstraction	11
3.2.2	Registration	13
3.2.3	Deregistration	13
3.2.4	Sending <i>Go_Aheads</i>	14
3.2.5	Analysis	14
4	Asynchronous multi-source BFS	16
4.1	Single-source 2^t -thresholded BFS, given a layered sparse 2^{t+6} -cover	17
4.1.1	Basic definitions	17
4.1.2	The algorithm	18
4.1.3	Analysis	19
4.2	Multi-source 2^t -thresholded BFS, given a layered sparse 2^{t+6} -cover	22
4.3	Multi-source $2^t \cdot \ell$ -thresholded BFS, given a layered sparse 2^{t+6} -cover	23
4.4	Synchronous deterministic construction of the d -cover	23
4.4.1	Synchronous construction of k -separated Network Decomposition	23
4.4.2	Constructing d -cover in the synchronous setting	24
4.5	Asynchronous construction of sparse 2^{t+7} -cover, given a layered sparse 2^{t+6} -cover	24
4.6	The complete BFS algorithm in $\tilde{O}(D)$ time and $\tilde{O}(m)$ messages	25
5	Polylogarithmic synchronizer for event-driven algorithms	27
5.1	The general structure of the event-driven algorithms	27
5.2	The setup	28
5.3	A synchronizer, given layered sparse $O(T(\mathcal{A}))$ -cover	29
5.3.1	Algorithm	29
5.3.2	Analysis of correctness	29
5.3.3	Analysis of the time and message complexity	31
5.4	A synchronizer, without being given layered sparse $O(T(\mathcal{A}))$ -cover	31
6	Applications	32
A	A review of Awerbuch's α, β, and γ synchronizers	35
B	Model subtleties	36
C	Algorithm for Theorem 4.20	39

1 Introduction and Related Work

Distributed graph algorithms have been studied extensively over the past five decades. The prevalent models for designing such algorithms are *synchronous* message-passing models, which assume that the processors compute and communicate in lockstep *rounds*. In particular, messages sent over different network links are assumed to traverse at the same speed: per round, all messages are sent at the beginning of the round, and they all arrive by the end of the round. Of course, the reality is far from such a nice and clean picture. In most practical settings, different messages will experience different delays, and even delays that are not predictable to the algorithm.¹ Arguably, the synchronous abstraction is still extremely valuable as it provides a clean context for algorithm design, allowing us to focus on (other) core issues such as graph-theoretic challenges and communication limitations. It also vastly simplifies the analysis because one does not have to deal with all possibilities of the delays experienced by different messages. But can one take these “lab-grown” algorithms, developed in the sterile synchronous setup, and use them in the asynchronous world?

Synchronizers are the fundamental theoretical concept introduced to provide a principled and general answer to this question. Roughly speaking, a synchronizer is a mechanism that can be added to *any* synchronous message-passing algorithm, so that the resulting combination would work correctly in the asynchronous setting. Furthermore, ideally, the synchronizer only incurs a slight loss in the algorithm’s efficiency, most notably, its time and message complexities. In this paper, we present the first deterministic synchronizer with near-optimal time and message complexity overheads. Informally, our synchronizer increases time and message complexities by factors only logarithmic in the network size. This leads to the first near-optimal deterministic asynchronous distributed algorithms for some of the most basic graph problems.

1.1 Models and the synchronizer definition

The synchronous message-passing model. The network is abstracted as an undirected graph $G = (V, E)$, and we use the notations $n := |V|$ and $m := |E|$. Each node represents one computer/processor, equipped with a unique identifier, typically assumed to have $b = O(\log n)$ bits. Computation and communications occur in synchronous rounds $1, 2, 3, \dots$. Per round, each computer/node performs some computations on its data. Then, it can send one B -bit message to each of its neighbors. Typically, we assume $B = O(\log n)$, and this model variant is usually called CONGEST [Pel00]. The variant which allows unbounded message sizes, i.e., $B = \infty$, is often called LOCAL [Lin92, Pel00]. All the messages sent in a round are assumed to arrive by the end of the round. Then, the algorithm proceeds to the next round. When talking about randomized algorithms, we assume that each source has a source of random bits, and these are independent across different nodes.

When discussing algorithms for graph problems, the standard assumption is that at the start of the algorithm, nodes do not know the global network topology G . They only know their own identifier, and perhaps some estimates on basic global parameters, e.g., a constant-factor upper bound on $O(\log n)$. At the end of the algorithm, each node should know its own output. For instance, if we are computing a breadth first search (BFS) tree from a given source node s , each node should know its own distance to s and perhaps also its parent in the BFS tree.

The two main measures of interest for an algorithm are its time complexity and message complexity. For an algorithm A in the synchronous setting, its *time complexity* $T(A)$ is defined as the number of rounds until all nodes generate their output. See the time complexity paragraph

¹In many computer networks—e.g., the Internet—each distributed algorithm is run alongside a myriad of other protocols that are using the network, and the delay in each link is influenced by the amount of congestion on it.

of [Appendix B](#) for more on this. Its *message complexity* $M(A)$ is defined as the total number of messages sent in the entire network during the algorithm. We comment that sometimes the phrase *communication complexity* is used instead of message complexity.

The asynchronous message-passing model. In the asynchronous model, there is no notion of rounds (known to the algorithms). Any node can perform some computation on the data it holds and then send messages to its neighbors. These messages are guaranteed to arrive eventually, but there is no time bound known to the algorithm. As such, algorithms are generally described in an *event-driven* language, e.g., “upon receiving message ..., do this: ...”

Analyzing algorithms in the asynchronous model is non-trivial because the model involves non-determinism in the timing of the arrival of different messages. A standard approach [Pe100], which makes algorithmic results stronger, is to take a worst-case view: we allow the message delays to be determined by an adversary, who knows the algorithm. To provide a performance bound, we assume an upper bound τ on the delay of each message (the time from the send event to the corresponding arrive event). The time complexity $T(A)$ of the algorithm A is defined as follows: Let T be the longest possible execution time until all nodes generate their output. This longest possible time is with regard to the worst-case execution of the algorithm, where the adversary controls the delays of different messages, subject to this bound τ . Then, we define the time complexity $T(A) = T/\tau$, i.e., we normalize the time bound by considering τ as one time unit. We emphasize that, crucially, the value of τ is not known to the algorithm. The message complexity $M(A)$ of the algorithm A is defined as the maximum number of messages the algorithm sends in the network, again in the worst-case possible execution where the adversary controls the message delays.

Synchronizers. Defined initially in the pioneering work of Awerbuch [Awe85], a synchronizer \mathcal{S} is an algorithmic module that can be added to *any* synchronous distributed algorithm \mathcal{A} for any problem \mathcal{P} so that the combined algorithm \mathcal{A}' would be guaranteed to run correctly in the asynchronous environment and solve \mathcal{P} . The key measures of interest for a synchronizer are its *time complexity overhead* and *message complexity overhead*, as well as its *initialization time complexity* and *initialization message complexity*. We define these next. Let us first ignore the initialization part and assume that the synchronizer initialization has already been performed. Then, the *time complexity overhead* of \mathcal{S} is defined as $\text{time-overhead}(\mathcal{S}) = T(\mathcal{A}')/T(\mathcal{A})$. Similarly, the *message complexity overhead* of \mathcal{S} is defined as $\text{message-overhead}(\mathcal{S}) = M(\mathcal{A}')/M(\mathcal{A})$. Besides these, the synchronizer may have some initialization phase, which is an algorithm that should run in the asynchronous setting and set up some structures before the start of the synchronizer-augmented variant of \mathcal{A} . This initialization itself has time complexity $T^{init}(\mathcal{S})$ and message complexity $M^{init}(\mathcal{S})$. Then, the synchronizer should satisfy the following for every synchronous algorithm \mathcal{A} :

$$T(\mathcal{A}') \leq T^{init}(\mathcal{S}) + \text{time-overhead}(\mathcal{S}) \cdot T(\mathcal{A}), \text{ and}$$

$$M(\mathcal{A}') \leq M^{init}(\mathcal{S}) + \text{message-overhead}(\mathcal{S}) \cdot M(\mathcal{A}).$$

1.2 State of the art

Synchronizers with global pulse generation per round. A natural approach to building synchronizers, set forth by Awerbuch [Awe85], is to have each node v generate a pulse for each round. Concretely, node v will generate pulses 1, 2, 3, ..., and these pulses demarcate for node v the transition between consecutive rounds. These pulses are known only to v . A message is called a pulse p message if it is sent by node v between its pulses p and $p + 1$. To simulate the synchronous algorithm, each node v should generate its pulse $p + 1$ only after its pulse p and, more crucially, after v has received every message of pulse p sent to it by each neighbor u . Of course, the

challenge is that node v does not know which neighbors send messages to it, and simply waiting for a predetermined amount of time cannot help; some messages might take unpredictably long.

Awerbuch introduced three synchronizers, known as α , β , and γ , which follow this outline of each node generating all the pulses 1, 2, 3, \dots . We review these in [Appendix A](#). Here, we only note that even ignoring their initialization costs, these synchronizers do not achieve a $\text{poly}(\log n)$ time and message complexity overhead. Indeed, in each case, at least one of these two overheads is $\Omega(n)$. Moreover, this problem is inherent to the approach that tries to generate all pulses at all nodes. This overly strong requirement necessitates some communication for each round/pulse for each node, while the synchronous algorithm might have each node send only in very few rounds.

Synchronizers with succinct pulse generation per round. Intuitively, to have small complexity overheads, each node should demarcate only rounds in which the node has some message to send/receive in the synchronous algorithm. In a pair of marvelous papers [[AP90b](#), [AP90a](#)], in 1990, Awerbuch and Peleg stated the existence of synchronizers with only $\text{poly}(\log n)$ time and message complexity overheads. The first paper [[AP90b](#)] presented the graph-theoretic notions of sparse partitions/covers and gives sequential algorithms for their construction. The other [[AP90a](#)], published in the same venue, sketched that once such sparse covers are constructed, one can transform any synchronous algorithm into an asynchronous algorithm at the cost of only $\text{poly}(\log n)$ time and message complexity overheads. However, there are two drawbacks.

The first drawback is that the sparse cover construction provided in [[AP90b](#)] was sequential and would require a high time and message complexity. A follow-up work of Awerbuch, Patt-Shamir, Peleg, and Saks [[APSPS92](#)] provided an efficient *randomized* asynchronous distributed construction, based on the synchronous low-diameter decomposition of Linial and Saks [[LS93](#)]. This led to the first randomized distributed synchronizer with only $\text{poly}(\log n)$ time and message complexity overheads. However, it remained open whether there is an efficient deterministic synchronizer.

The second drawback is, unfortunately, inaccuracy or incorrectness issues. The papers [[AP90a](#), [APSPS92](#)] involve several ingenious ideas and a generally viable approach to building synchronizers with small overheads. However, regrettably, the only available versions of [[AP90a](#), [APSPS92](#)] are their conference versions.² In many places, these versions do not provide proofs, or even concrete lemma statements stating the desired and guaranteed behavior of parts of the algorithm. This leaves much to be filled out by the reader, and to the best of our understanding, there are at least two critical inaccuracies:

(I) The synchronizer scheme presented in [[AP90a](#)], as described, is technically incorrect. It overlooks a small but critical issue of congestion, which, once corrected, increases the time complexity overhead to $\Omega(n)$. The follow-up work of Awerbuch et al. [[APSPS92](#)], mentioned above, described the synchronization approach of [[AP90a](#)] in a different way, which incidentally goes around the congestion issue. This was without explicitly pointing out any incorrectness issues in the latter. But this adaptation opens up a subtle correctness problem. One needs a highly non-trivial statement and analysis to show that the algorithm still operates as desired despite the asynchrony and all possibilities of message delays. In fact, we do not see how to fix the algorithm in [[APSPS92](#)] in the exact way that it is written.

(II) The synchronizers in [[AP90a](#), [APSPS92](#)] are described primarily for synchronizing a BFS algorithm from a given source. The above incorrectness/incompleteness issues apply even to this restricted case. But the work of [[AP90a](#), [APSPS92](#)] claims synchronization for an arbitrary synchronized algorithm. Both papers contain a brief follow-up section, after discussing BFS, that claims how to generalize the BFS approach (without much proof). We argue that this extension's

²We have had personal communication with David Peleg, who confirmed that no full version of these papers exists.

correctness depends on the interpretation of algorithms in the synchronous model. If in the synchronous algorithm, a node is allowed to do something like “wait for r many rounds, or wait for round number r , and then send message m ”, we argue that the message complexity overhead of the synchronizers of [AP90a, APSPS92] can be $\Omega(n)$. See Appendix B for more on this.

1.3 Our contribution

To summarize, there are two drawbacks to the state-of-the-art: (a) the synchronizers of [APSPS92, AP90a] are randomized, (b) they contain incompleteness/incorrectness issues. We address both.

1.3.1 A near-optimal deterministic distributed synchronizer.

Developing *deterministic* distributed algorithms for graph problems has been one of the leading research themes over the past four decades, with significant recent breakthroughs; see, e.g., [RG20, GHK18, GKS17]. However, without a deterministic synchronizer, these deterministic synchronous algorithms cannot be transported to the more realistic asynchronous world (without significant overheads). Obtaining a deterministic synchronizer with $\text{poly}(\log n)$ time and message complexities has been a long-standing open problem since [APSPS92]. See e.g. [Mic]. Our main contribution is to remedy this. We provide the first deterministic distributed synchronizer with $\text{poly}(\log n)$ time and message complexities:

Theorem 1.1. (Informal) *There is a deterministic synchronizer, with no initialization, with time and message complexity overheads $\text{poly}(\log n)$. Concretely, any synchronous algorithm can be transformed into an asynchronous algorithm with a $\text{poly}(\log n)$ loss in time and message complexities.*³

Applications. Our near-optimal deterministic synchronizer leads to the first deterministic distributed asynchronous algorithms with near-optimal time and message complexities for many of the basic graph problems, including single-source or multi-source BFS, leader election, and minimum spanning tree, as we state in corollaries next. See Section 6 for proofs of these corollaries.

Corollary 1.2. *There is a deterministic asynchronous distributed algorithm that, given a source node s , computes a breadth first search tree rooted in this source node in $\tilde{O}(D)$ time and using $\tilde{O}(m)$ messages. Each node v learns its distance from source s and knows its parent in the BFS tree. The algorithm can be extended to the multi-source setting, where there is a set S of multiple sources, and each node should join the BFS tree of the closest source.*

Corollary 1.3. *There is a deterministic asynchronous distributed algorithm that elects a leader in the network in $\tilde{O}(D)$ time and using $\tilde{O}(m)$ messages. Every node learns the identifier of the leader.*

We remark that for leader election, even the randomized asynchronous case was open until a very recent work of Kutten et al. [KMJPP21], and their randomized algorithm assumes that nodes know n up to a constant factor. Our algorithm is deterministic, and the more standard assumption of nodes known n up to a polynomial suffices for it, i.e., knowing a constant-factor upper bound on $O(\log n)$, which is the identifier/message sizes and needed as part of the CONGEST model.

Corollary 1.4. *There is a deterministic asynchronous distributed algorithm that computes a minimum spanning tree in $\tilde{O}(D + \sqrt{n})$ time and using $\tilde{O}(m)$ messages.*

³Technically, we assume that the synchronous algorithm sends at least one message along each edge, and thus has message complexity $\Omega(m)$. Without this, we should state the message complexity of the asynchronous version with an additive $m \text{poly}(\log n)$ term, as done in the abstract. The same condition applies to the synchronizers of [APSPS92, AP90a], and this $\Omega(m)$ message complexity is satisfied by a wide range of graph algorithms of interest.

As a side remark, we note that for asynchronous MST, even the randomized case was open. Mashreghi and King [MK19] asked whether there is a randomized asynchronous MST algorithm with $\tilde{O}(m)$ messages that takes even just $o(n)$ time in graphs that have $D = o(n)$. A very recent work of Dufoulon et al. [DKMJ⁺22] gives a randomized algorithm with $\tilde{O}(m)$ messages that take even just $\tilde{O}(D^{1+\varepsilon} + \sqrt{n})$ time, for any small constant $\varepsilon > 0$. Corollary 1.4 is deterministic and achieves the optimal time and message complexity bounds up to logarithmic factors.

1.3.2 Inaccuracy or correctness issues.

As mentioned before, there are two types of issues in [APSPS92, AP90a]: (I) problems that appear even for synchronizing BFS algorithms (concretely in a subroutine for registration and deregistration), and (II) problems in the extension to general synchronous algorithms (concretely, in the definition of the synchronous model and the message complexity blow-up of the extension).

For issue (I), the fix needs some algorithmic adjustments and careful lemma statements. We describe a variant of the approach of [APSPS92, AP90a] and provide a correctness analysis that considers all possibilities in the asynchronous environment. See Section 3.2 for more on this.

Issue (II) turns out to depend critically on the interpretation of synchronous algorithms. We are not aware of any explicit discussion about this subtlety in prior work. We show a natural and useful interpretation with which the $\text{poly}(\log n)$ time and message complexity overheads claims remain correct. Informally, the synchronous algorithm should have no explicit reference to the round numbers. It should be only *event-driven*, meaning it is described as: *Upon receiving messages ... (potentially several in the same round), send messages ... (immediately after, as soon as possible for the given number of messages to be sent, and without waiting for many rounds)*. See Appendix B for more on this. Fortunately, a wide range of synchronous algorithms of interest are event-driven in this sense or can be paraphrased to be event-driven without any asymptotic loss in their complexities. This includes usual algorithms for BFS, leader election, and MST.⁴

Finally, we note another subtle point that is not explicitly discussed in [APSPS92, AP90a]: The time complexity definition for which the $\text{poly} \log n$ time complexity overhead holds is the time from the start until all nodes generate their output (in both the synchronous and asynchronous algorithms). In the asynchronous version of a synchronous algorithm generated by the synchronizers of [APSPS92, AP90a], it is possible that some nodes might continue to perform some auxiliary communications longer, after outputting their output from the synchronous algorithm (or when in the synchronous algorithm they have no output). This can continue for up to $\tilde{O}(D)$ time. Our deterministic synchronizer is also with respect to the same definition of time complexity. See the time complexity paragraph in Appendix B for more on this.

2 Preliminaries

Notations. We assume that the network is an n -node m -edge undirected graph $G = (V, E)$, and we use D to denote the diameter of the graph. We use $\text{dist}(v, u)$ to denote the distance of nodes v and u in the graph. For a set S of vertices, we define $\text{dist}(v, S) := \min_{s \in S} \text{dist}(v, s)$.

⁴Furthermore, even algorithms that depend on time can be turned into this event-driven language, by each node generating a clock for itself by communicating back and forth with one of its neighbors, but that would be at the expense of nT additional message complexity, where T denotes the time complexity of the synchronous algorithm.

2.1 Sparse covers

Definition 2.1. For any positive integer d , we define a **sparse d -cover** with stretch s of an n -node graph to be a set of clusters, such that the following conditions hold:

- The diameter of each cluster is $O(d \cdot s)$.
- Each node is in $O(\log n)$ clusters.
- For any $u, v \in V$ s.t. $\text{dist}(u, v) \leq d$, there is at least one cluster that includes both u and v . In fact, a stronger statement holds. For each node v , there exists a cluster such that all node u for which $\text{dist}(u, v) \leq d$ are included in this cluster.

We assume that each node knows in which clusters it is. Furthermore, for each cluster, we have a tree of depth $O(d \cdot s)$ that spans all cluster nodes, where each node knows its parent and its children.

Often we need sparse covers for all powers of 2 up to roughly d . To capture that, we define a **layered sparse d -cover** for a parameter d to be a collection of sparse 2^j -covers for all $j \in \{0, 1, 2, \dots, \lceil \log_2 d \rceil\}$.

The optimal value of stretch in sparse covers is $O(\log n)$ [AP90b, LS93]. We work with deterministic distributed constructions of sparse cover that, while being time efficient, have slightly higher stretch [RG20]. For the rest of the paper, we will build and use only sparse d -covers with the properties stated below. See [Theorem 4.21](#) and [Theorem 4.22](#) for the precise theorem statements in the synchronous and asynchronous settings.

- The cluster stretch is $O(\log^3 n)$. More concretely, each cluster has a $O(d \log^3 n)$ -depth cluster tree, which spans all cluster nodes.
- Each edge is used in only $O(\log^4 n)$ cluster trees.

2.2 Running several algorithms in the asynchronous environment

In our algorithms, we often have several algorithmic subroutines. A scheduling question arises when these subroutines want to use the same edge for communications. In general, we might need to run all these subroutines in parallel in the sense that each of them relies on the progress of the others, or we might want to run them sequentially, in the sense that the i^{th} one depends only on subroutines 1 to $i - 1$. In the synchronous setting, this is usually simple. In this subsection, we discuss how to perform these in the asynchronous environment, and the related time guarantees.

2.2.1 Running in parallel

Suppose we want to run several subroutines in parallel. These subroutines may desire to communicate over the same edge e . Notice that, in both the synchronous setting (with the event-driven interpretation) and the asynchronous setting, we have restricted the algorithms to waiting for each injected message's acknowledgment, before sending the next message. Hence, different subroutines might appear to slow down each other. How do we schedule the communications of these subroutines over edge e , so they all work in a small time complexity?

Let us call the subroutines $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_t$. It would be very convenient if we had a separate copy of the edge e for each subroutine. Then the messages sent for different subroutines would not get in the way of each other. To deal with these issues, we will use the following abstraction principle to simulate “making copies” of the edges:

Lemma 2.2. For a given graph G and an integer k , let us define $G(k)$ as a graph G , in which each edge was duplicated k times. In particular, $G(1) = G$. Suppose that for a given k , some

asynchronous algorithm \mathcal{A} runs on the graph $G(k)$ in time $O(f)$, where f is some function from the parameters of the graph. Then, we can simulate algorithm \mathcal{A} on the graph G in time $O(kf)$.

Proof. For each node u , and any node v connected to it, let u number edges from u to v in $G(k)$ by $1, 2, \dots, k$. u will simulate sending messages to v in $G(k)$ by sending messages in turns. At the i -th turn, v will check if there is a message that v wants to send to u through the $(i \bmod k)$ -th edge. If yes, v will send this message and proceed to the $i + 1$ -st turn; otherwise, it will proceed to the $i + 1$ -st turn right away.

All the nodes are sending the same messages as they would according to \mathcal{A} in $G(k)$. The only difference is that while previously, they were guaranteed to be delivered in 1 time unit, now they are guaranteed to be delivered in k time units, as a message has to wait for at most k turns. So, the time complexity of the algorithm increases by at most a factor of k . \square

Corollary 2.3. *Suppose that we want to run several procedures $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \dots$ on a graph G , and the following conditions hold: (a) Each edge is involved in sending messages for $O(k)$ procedures. An edge might not know in advance in which procedures it is involved; see [Remark 2.4](#) for more. (b) If for each edge e we had a separate copy of e for every procedure that involves e , then all the procedures would be completed in time $O(t)$. Then, we can run all these procedures on G in $O(kt)$ time.*

Remark 2.4. *Note that the edge (u, v) does not have to know in advance how many procedures it will be involved in. The statement holds if each edge is going to be involved in sending messages of at most $O(k)$ procedures, even if it does not know whether it will receive messages of new procedures in the future and does not know the value of k . Each endpoint of e will simply perform communications of all of the procedures that want to send a message along e in a round-robin fashion, along this edge e . In this sense, this abstraction is adaptive.*

2.2.2 Running sequentially

Suppose now that our algorithm consists of several stages $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_k$, where \mathcal{S}_i relies only on the results of previous stages. In the synchronous world, if we knew the time bounds on each stage, we could run them sequentially: before starting the stage \mathcal{S}_{i+1} , we would wait until the entire \mathcal{S}_i is finished. Unfortunately, in the asynchronous world, we cannot have any such notion of "waiting" for a certain amount of time, for the previous stages to be over. Note that we do not require the entire \mathcal{S}_i to be finished before we start sending any of the messages of \mathcal{S}_{i+1} : messages may go through faster in some parts of the graph than in others.

How do we bound the time spent on running all of these stages? With the approach from [Section 2.2.1](#), we could do it as follows: Suppose that if for each edge we had a separate copy of it for each stage, all the procedures would be over in time $O(t)$. Then the best we could say according to the [Corollary 2.3](#) is that all stages would be over in $O(kt)$ time. We, however, can prove a stronger statement.

Lemma 2.5. *Suppose that there is an algorithm \mathcal{A} that involves several sequential stages $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_k$, and the algorithm works on a graph G , with the following properties:*

- *Each \mathcal{S}_i relies only on the previous stages $\mathcal{S}_1, \dots, \mathcal{S}_{i-1}$. In other words, it is possible that \mathcal{S}_i starts after $\mathcal{S}_1, \dots, \mathcal{S}_{i-1}$ are completed, and no stage \mathcal{S}_j , for $j > i$, has even started yet.*
- *If all stages $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_{i-1}$ are over, and \mathcal{S}_i has not started yet, then \mathcal{S}_i is over in at most T_i additional time. We will refer to T_i as its **isolated time complexity**.*

Then, we can run this algorithm in time $O(T_1 + T_2 + \dots + T_k)$.

Proof. We run the algorithms together, with the following scheduling rule for messages: whenever a node has several messages to send over an edge $e = \{v, u\}$, node v prioritizes messages of lower stages and sends them first. With this rule, the only way for a message of \mathcal{S}_i to be delayed is that it is waiting for some messages of stages $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_{i-1}$ to go through the edge. Notice that since the stages are devised for the asynchronous setting, each of them should work correctly even if its messages are delayed adversarially by an arbitrary amount of time. Now, by an induction on i , we argue that by time $T_1 + T_2 + \dots + T_i$, all stages $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_i$ are over. For the base case, the algorithm of \mathcal{S}_1 is not delayed by any other stage and thus finishes in T_1 time. For the inductive step, notice that by time $T_1 + T_2 + \dots + T_j$, all stages $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_j$ are over. By that point of time, the algorithm of \mathcal{S}_{j+1} might have made some partial progress in its execution, but it always remains in a valid state of the algorithm. Hence, by the second condition in the lemma statement, we know that from any valid state, the \mathcal{S}_{j+1} algorithm takes at most T_{j+1} more time to finish. The reason is this: consider the adversarial execution where we first execute and complete $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_j$, then we start \mathcal{S}_{j+1} but we adjust the delays to reach the same valid state, and then we run the rest of the algorithm; this should still terminate within T_{j+1} more time. Hence, all stages $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_{j+1}$ are over by time $T_1 + T_2 + \dots + T_{j+1}$. This completes the inductive proof. \square

Our entire algorithm will consist of several stages, satisfying the constraints of the [Lemma 2.5](#), so we will bound its total runtime by the sum of the isolated time complexities of all its stages.

3 Collecting information in covers

3.1 Collecting information in $(d \cdot \ell)$ -ball with sparse d -cover

In the course of our algorithm, sometimes we need a node to gather some information about other nearby nodes. We use the notion of sparse covers described in [Section 2.1](#) to do this efficiently.

Consider some algorithmic subroutine \mathcal{P} . This can be simply a part of the algorithm, e.g., a phase in the synchronous algorithm. Suppose that each node will run this process \mathcal{P} at most once. We will say that node v **is done with \mathcal{P}** when process \mathcal{P} terminates in node v , or when node v learns that v will not run \mathcal{P} at all. In this subsection, we discuss a way to collect information about the completion of \mathcal{P} using sparse d -covers, assuming these covers have already been constructed.

Information gathering up to the sparse cover radius. Let us say that we want each node v to learn when all the nodes in d -neighborhood of v are done with \mathcal{P} . The following theorem statement provides the desired procedure.

Theorem 3.1. *Under the conditions above, there exists a procedure \mathcal{A} that lets each node learn when all the nodes in its d -neighborhood are done with \mathcal{P} . The procedure has the following properties:*

- *Let t be the time by which all nodes are done with \mathcal{P} (the value of t is not known only for the analysis and is unknown to the algorithms). Then \mathcal{A} terminates by time $t + O(d \log^7 n)$.
In other words, the isolated time complexity of collecting this information is $O(d \log^7 n)$.*
- *Procedure \mathcal{A} uses only $O(m \log^4 n)$ extra messages.*

Proof. We do a convergecast in each cluster of the cover separately. Consider one particular cluster. We use a convergecast and broadcast in the cluster. In particular, node v waits until all the nodes in its subtree are done with \mathcal{P} . If v has any children, it gets this update from its children. Once all descendants of v are done with \mathcal{P} , node v notifies its parent that v and its subtree are done with \mathcal{P} . Once the cluster root has learned that the entire cluster is done with \mathcal{P} , it starts a broadcast, propagating the confirmation that everyone in the cluster is done with \mathcal{P} . Each node that receives

this confirmation from its cluster parent sends it to its children in the cluster. If the node is a leaf, then it does nothing (effectively, the broadcast terminates in its branch).

Now let us consider a node v . Notice that the node is included in many clusters. Node v waits until it receives such a confirmation broadcast from all the clusters in which v is present. Since for every two nodes u, v at a distance at most d there is a cluster in which both of them are present, node v will finish collecting the information only after u is done with \mathcal{P} .

We can imagine this as an algorithm consisting of two parts: the first part is running \mathcal{P} , and at the end of it each node should know that itself is done with \mathcal{P} . The second part is ensuring that we wait until all nodes up to distance d are done with \mathcal{P} . This second part is done by a convergecast and broadcast in the cluster trees. Remember that each convergecast/broadcast is up to distance $O(d \log^3 n)$, and each edge is in $O(\log^4 n)$ cluster trees, so, by [Corollary 2.3](#), the isolated time complexity of the second part is $O(d \log^7 n)$. Since the first part is not dependent on the second one, by [Lemma 2.5](#), we get that, if the first part finishes by time t , then the entire algorithm finishes by time $t + O(d \log^3 n \cdot \log^4 n) = t + O(d \log^7 n)$.

For message complexity, note that each edge is in $O(\log^4 n)$ clusters, each cluster sends only $O(1)$ messages through each of its edges. Thus, we have only $O(m \log^4 n)$ extra messages. \square

Information gathering slightly above the sparse cover radius. What if we want to collect information up to distance of $d\ell$, which is somewhat greater than the radius d that is guaranteed to be covered by the provided sparse cover? We show that this is also possible, though we end up paying an ℓ factor in time and message complexities. The time complexity increase is natural as we care about the greater radius and it takes $\Omega(d\ell)$ time for any information to traverse this distance. The message complexity increase is in some sense an inefficiency of this extension. The following theorem statement provides the desired procedure.

Theorem 3.2. *Under the conditions above there exists a procedure \mathcal{A} , letting each node learn when all the nodes in its $d\ell$ -neighborhood are done with \mathcal{P} , with the following properties:*

- *If all nodes are guaranteed to be done with \mathcal{P} by some time t , \mathcal{A} is guaranteed to be over in time $t + O(d\ell \log^7 n)$.*
- *It takes only $O(m\ell \log^4 n)$ extra messages.*

Proof. Let process \mathcal{P}_ℓ be: “knowing that my $d \cdot \ell$ -neighborhood is done with \mathcal{P} ”. Then v can finish process $\mathcal{P}_{\ell+1}$ by collecting the information about completion of \mathcal{P}_ℓ in its d -cover. So, in addition to waiting until nodes complete the initial process \mathcal{P} , we sequentially run ℓ stages. In the i -th stage, we make all nodes be done with \mathcal{P}_i .

We then apply the same argument as in [Theorem 3.1](#), and get that the entire algorithm will be over by time $t + O(\ell \cdot d \log^3 n \cdot \log^4 n) = t + O(d\ell \log^7 n)$. As we do ℓ stages, we have only $O(m\ell \log^4 n)$ extra messages. \square

3.2 Registration in clusters

3.2.1 The Abstraction

In [Section 3.1](#), we considered a process \mathcal{P} such that each node v eventually learns that v is done with \mathcal{P} , i.e., that either v participated in \mathcal{P} and it has terminated, or v learned that v will not run process \mathcal{P} . Here, we discuss a somewhat similar setup, but with a critical difference: a node v does not know whether it will participate in \mathcal{P} or not, and it may never participate in process \mathcal{P} . In an informal sense, now node v is done with \mathcal{P} only if it participates and has terminated process \mathcal{P} . We would again like to gather information about nodes that are done with \mathcal{P} (this will be primarily

relevant only for other nodes that are done with \mathcal{P} themselves). The precise properties that we desire are more subtle and are described below.

Definition 3.3 (Registration Abstraction). *Consider some cluster C , whose nodes are performing some process \mathcal{P} . We will use the following abstraction to collect the information about the completion of the processes \mathcal{P} . When node v wants to start the process \mathcal{P} , it will **register** in the cluster C . When node v has finished the process \mathcal{P} , it will **deregister** in the cluster C and wait until C sends it a message **Go_Ahead**, allowing it to proceed to the next steps of the algorithm. We emphasize that a node does not necessarily know in advance whether it will take part in \mathcal{P} or not; it might end up participating in it because of receiving some message.*

Ideally, we would want node v to receive *Go_Ahead* after all the nodes who are going to register in the cluster have already registered and deregistered. That is, we do not want v , who has participated in \mathcal{P} , to proceed to the next steps of the algorithm if some node from some cluster C that includes v is still performing \mathcal{P} and has not terminated \mathcal{P} . We will state the precise desired guarantees soon.

Let us also briefly discuss the desired complexities: Intuitively, we also desire that each registration/deregistration operation takes time only proportional to the height of the cluster tree, and that the message complexity of the solution is proportional to the number of the nodes that register/deregister (up to a factor of the height of the cluster), but critically not proportional to all the nodes/edges in the entire graph. We discuss the precise complexity bounds of our solution in [Section 3.2.5](#).

Is a simple convergecast/broadcast enough? We could try to collect the above information with a convergecast in the tree: node gathers the information about registration/deregistration inside its subtree and passes it to its parent. When everyone who is going to register has already deregistered, the root would issue a *Go_Ahead*, and propagate it to all the nodes who have participated in the registration process. Unfortunately, this is not possible in this simple way: a node does not know in advance whether it will perform \mathcal{P} or not. Hence, it cannot know in advance whether it will register or not, and its parents do not know in advance whether their subtree is done running \mathcal{P} or not, and simply waiting will not solve the problem. This is because it is possible that one node never participates in \mathcal{P} and will never know that it will not participate.

Provided guarantees by the registration process. We will ask for two guarantees from this registration process:

Lemma 3.4 (Register Guarantee 1). *When node v receives *Go_Ahead*, all the nodes that have registered before v has deregistered are already deregistered. If the distance from the node v to the root r is h , then both its registration and deregistration take $O(h)$ time and messages.*

Lemma 3.5 (Register Guarantee 2). *Suppose eventually no new registration happens, and all nodes who have registered have deregistered. Then, if the height of the tree is h , then every node that has ever registered will receive *Go_Ahead* in time $O(h)$. The total number of messages spent on sending *Go_Aheads* is proportional to the total number of messages spent on registration and deregistration.*

A natural attempt. A natural way to implement the operations of registration, deregistration, and sending *Go_Ahead* would be to send all the messages directly to the cluster root, as follows:

- To register node v in the cluster C , send a registration message to the root of C with the id of v and let the root remember it. Once v receives a response back from the root, it is considered registered.

- To deregister node v , send a deregistration message to the root, again with the id of v , and let the root remember it. Once v receives a response back from the root, it is considered deregistered.
- When the root detects that all registration messages have been matched with all deregistration messages, it will issue a *Go Ahead* and propagate down the tree to all the nodes that are waiting for it.

Indeed, this is essentially the version of registration and deregistration described in the pioneering work of Awerbuch and Peleg [AP90a]. Unfortunately, this approach suffers from a congestion problem, and can require $\Omega(n)$ time even if the cluster tree has a very small height. The reason is this: consider an edge e connecting v to its parent u , and suppose we have many—up to $\Theta(n)$ —nodes in the subtree below v who want to register. All of their registration messages have to pass through e , and this requires $\Omega(n)$ time.

The right approach. We will use a different approach, which is close to the variant described in [APSPS92]. Although, we will need to adjust a part of their algorithm and provide a highly non-trivial correctness proof. Unfortunately, no such proof or even guarantee statement is provided in [APSPS92]. Intuitively, we will collect the global OR of the registration and deregistration status in the cluster, using processes similar to convergecast and broadcast. We will do this by marking some edges as **dirty** during the registration process. Since this OR computation does not occur momentarily, and asynchrony might change the timing of different parts of the OR computation, we need careful statements for the algorithm and the proof of its guarantees. Below we delve into the details of this process.

3.2.2 Registration

When a node v registers in the cluster C , its goal is to ensure that all the edges on the path from v to the cluster root r are marked as dirty. Intuitively, this will be by starting a wave of messages that go toward the root and come back to the node. The precise description needs more care. Let us call a node u **finished** only once it knows that all the edges from u to r are marked as dirty. Therefore, the registration of the node v once it becomes finished.

The registration of the node v is a recursive procedure, and we call it $\mathcal{R}(v)$. When node v wants to register, it will invoke $\mathcal{R}(v)$. After it's over, it will mark itself as **registered**. Now, let us get to the details of the procedure \mathcal{R} : suppose that we invoked it in a node u .

If u already is finished (either because it's a root r , which is always finished, or because it became finished during the invocation of this procedure for some other node), then $\mathcal{R}(u)$ terminates immediately. Otherwise, let par denote the parent of u in the cluster tree. If the edge (par, u) is not dirty, u will mark it as dirty (and notify par of this, sending the corresponding message). Then, it will invoke $\mathcal{R}(par)$ by sending a message to par , and wait until par notifies it that $\mathcal{R}(par)$ is over. After par notifies u that $\mathcal{R}(par)$ is over, u knows that u is also finished. It then notifies any children that had invoked u , and then $\mathcal{R}(u)$ is over.

3.2.3 Deregistration

Intuitively, the goal of the deregistration of node v is to “cancel” the effect of the registration of v . Again, the intuitive thing would be to send a wave of messages from v toward the cluster root r , now erasing dirty marks along the path (until reaching a node that has another child edge that remains dirty). The precise description needs more care.

When node v wants to deregister, it immediately marks itself as **deregistered**. Then, node v starts the recursive deregistration procedure $\mathcal{D}(v)$. When $\mathcal{D}(u)$ is invoked at a node u , this node starts by looking at the edges from u to its direct children. If any of them are dirty, $\mathcal{D}(u)$ is immediately terminated. If $u = r$, or if u is still registered, then $\mathcal{D}(u)$ is also immediately terminated. Otherwise, let par denote the parent of u in the cluster tree. As we are deregistering some descendants of u , the entire path from u to the root is dirty, and, in particular, the edge (par, u) is dirty. Then, u does the following three actions:

- Remove the dirty mark from the edge (par, u) , and mark it as **waiting** instead. Intuitively, a waiting edge indicates that a *Go_Ahead* has to be sent through it.
- Remove the finished mark from node u (because edge (par, u) is not dirty anymore).
- Inform par to start $\mathcal{D}(par)$.

3.2.4 Sending *Go_Aheads*

As soon as the cluster root r receives an update that the edge from r to one of its children has had its dirty mark removed, it does this: if now all the edges from r to its children are not dirty, root r issues a *Go_Ahead*. It will start propagating this *Go_Ahead* through the waiting edges with a recursive procedure \mathcal{G} , by invoking $\mathcal{G}(r)$.

Let the current node in which \mathcal{G} is invoked be u . If u has deregistered, but it has not received a \mathcal{G} yet, once it receives the \mathcal{G} , then node u is immediately marked as **free**: Here, **free** means that node u has received the *Go_Ahead* that it was waiting for and can proceed to the next stages of the algorithm.

Any node u in which \mathcal{G} is invoked also may need to propagate the *Go_Ahead* to some of its children. Let ch be some particular child of u . Node u propagates *Go_Ahead* to child ch only if the edge (u, ch) is waiting. If the edge (u, ch) is not waiting (e.g., if it stopped being waiting before u has propagated the *Go_Ahead* to ch), then u will not propagate this *Go_Ahead* to its child ch . When child ch receives this propagating message, it will invoke $\mathcal{G}(ch)$.

3.2.5 Analysis

In this subsection, we prove that the register guarantees mentioned earlier hold, and bound the time and message complexities of registration, deregistration, and sending *Go_Aheads*.

Lemma 3.6. *If ch is a child of v in the cluster tree, and ch is finished, then v is also finished.*

Proof. Let us look at a maximal contiguous interval of time such that ch remains finished during this entire time interval. Node ch can become finished only after getting to know that v is finished. Now, we have to show that v remains finished until the finished mark of ch is removed. Notice that the only way node v might stop being finished is if $\mathcal{D}(v)$ is invoked. However, since the edge (v, ch) remains dirty, $\mathcal{D}(v)$ will be immediately terminated. \square

Lemma 3.7. *From the moment when v is registered, till the moment when v is deregistered, all the nodes on the path from v to r are finished, and all the edges on the path from v to r are dirty.*

Proof. Note that v remains finished during this entire time interval: it is finished when it just registered, and it can stop being finished only during $\mathcal{D}(v)$, which will be immediately terminated if v is still registered. Then, by the [Lemma 3.6](#), the parent of v will also remain finished during this time interval, as well as the parent of the parent of v , and so on: all the nodes on the path from v to r will remain finished. As all nodes on this path are finished during the entire interval, all the edges on this path remain dirty during this interval. \square

Lemma 3.8. *Suppose that node v received `Go_Ahead`. From the moment when it was issued by the root r , till the moment when v received it, at least one edge on the path from v to r is not dirty.*

Proof. Consider the way `Go_Ahead` is propagated from r to v . There are only two things that it's doing during this time interval: it may sit in some node v , waiting until v decides whether to propagate it to its child ch , or it may be propagated from v to one its children ch through the edge (v, ch) . In both cases, the edge (v, ch) has to remain not dirty: otherwise, v will simply not propagate the `Go_Ahead` to ch . \square

Lemma 3.9. *If node v receives `Go_Ahead`, this `Go_Ahead` was issued by r after v has deregistered.*

Proof. Suppose that it was issued by r before node v has deregistered. By definition, it can reach v only after v has deregistered. Consider the point in time right before v has deregistered, but after it has registered, and after this `Go_Ahead` has already been issued by r . By [Lemma 3.8](#), at this point in time, at least one edge on the path from v to r had to be not dirty, while by [Lemma 3.7](#), all these edges had to be dirty. Contradiction. \square

Lemma 3.4 (Register Guarantee 1). *When node v receives `Go_Ahead`, all the nodes that have registered before v has deregistered are already deregistered. If the distance from the node v to the root r is h , then both its registration and deregistration take $O(h)$ time and messages.*

Proof of [Lemma 3.4](#). Suppose that node v has received `Go_Ahead`, but node u hasn't deregistered yet, and that u has registered before v has deregistered. Let us look at the moment when this `Go_Ahead` was issued by the r . This could not happen after u has registered, as the entire path from u to the root is marked dirty until u deregisters, by [Lemma 3.7](#). So, this `Go_Ahead` was issued before u registered and, therefore, before v has deregistered, contradicting [Lemma 3.9](#). \square

Lemma 3.5 (Register Guarantee 2). *Suppose eventually no new registration happens, and all nodes who have registered have deregistered. Then, if the height of the tree is h , then every node that has ever registered will receive `Go_Ahead` in time $O(h)$. The total number of messages spent on sending `Go_Aheads` is proportional to the total number of messages spent on registration and deregistration.*

Proof of [Lemma 3.5](#). Let us show that every node will eventually receive `Go_Ahead`. Consider a moment when all the nodes that have registered have deregistered, after which no new registrations will happen, and all the procedures are over: basically, the moment after which there will not be any more updates to the status of the nodes and edges of the cluster. Suppose that node v has registered and deregistered but has not received the `Go_Ahead`.

Let us say that v is in a **good** state, if one of the following holds:

- All the edges on the path from v to r are marked as waiting.
- There is some node $v_1 \neq r$ on the path from v to r , such that all the edges on the path from v to v_1 are marked as waiting, and the edge from v_1 to its parent is marked as dirty.

Right before v has deregistered, it is in a good state. How can v stop being in a good state? At each point in time, consider the higher ancestor h of v such that all the edges of the entire path from v to h are currently waiting. If v is in a good state, then h is the root of the cluster, or the edge from h to its parent is dirty.

This state can be broken in one of the following ways. We analyze all of them and show that v will inevitably end up in a good state again after some time.

1. *Go_Ahead* is propagated down on the path from h to v . By our assumption, it does not reach v , so it must get stuck somewhere on the path, which happens when it detects a dirty edge, which once again moves v to the good state of the second type.
2. Some edge on the path from v to h becomes dirty during the registration process of some node. Then v immediately ends up in a good state of the second type.
3. The edge from h to its parent stops being dirty during the deregistration of some descendants of h and becomes waiting instead. Let us follow this deregistration process. If it reaches the root, then v ends up in a good state of the first type. Otherwise, it reaches a dirty edge, and v ends up in a good state of the second type.

Now, consider the last time any dirty edge has been marked as waiting. It has happened in the last \mathcal{D} call, which could be only from some child of r . By the invariant, it follows that all the edges on the path from v to r have to be waiting from that moment on. But then the *Go_Ahead* issued by r after marking this edge as waiting will reach v , a contradiction.

From the above proof, it also follows this *Go_Ahead* will reach v in time at most $O(h)$. To show that the total number of messages spent on sending *Go_Aheads* is proportional to the total number of messages spent on registration and deregistration, note that every *Go_Ahead* message is removing a waiting mark from an edge. \square

4 Asynchronous multi-source BFS

The (multi-source) BFS problem in the synchronized setting takes $O(D_1)$ rounds and $O(m)$ messages, where D_1 is the largest value of $\text{dist}(v, S)$ over all nodes v . Here, S denotes the set of all sources, and we want each node to learn its distance only to the closest source. In this section, we develop an asynchronous BFS algorithm with complexities similar to the synchronous variant. In [Section 5](#), we explain how we extend this scheme to a general synchronizer for an arbitrary event-driven synchronous algorithm, thus proving (the formal variant of) [Theorem 1.1](#).

Theorem 4.1. *There exists an asynchronous deterministic algorithm that computes single-source BFSs in $\tilde{O}(D)$ time and using $\tilde{O}(m)$ messages.*

In the case of multiple-source BFS, there exists an asynchronous deterministic algorithm that computes multiple-source BFS in $\tilde{O}(D_1)$ time and using $\tilde{O}(m)$ messages, where D_1 is the largest value of $\text{dist}(v, S)$ over all nodes v .

As a concrete step toward this theorem, we define a thresholded version of the BFS where we want to learn the distance only for nodes whose distance to the sources is upper bounded by this threshold. The formal definition is as follows.

Definition 4.2. *For any nonnegative integer τ , we define the τ -thresholded BFS problem as follows: Let S be the set of source nodes. For each node v , if $\text{dist}(v, S) \leq \tau$, then v should output $\text{dist}(v, S)$. If $\text{dist}(v, S) > \tau$, then v should output a special symbol ∞ , which indicates that $\text{dist}(v, S) > \tau$ and thus a BFS of depth τ from S does not reach v .*

Roadmap. We build our BFS algorithm step by step, and over different subsections.

In [Section 4.1](#), we show how to run a single-source 2^t -thresholded BFS, assuming that a layered sparse 2^{t+6} -cover has been provided. In [Section 4.2](#), we show how to extend the approach from [Section 4.1](#) to the multi-source BFS problem. In [Section 4.3](#), we show how to run $2^t \cdot \ell$ -thresholded

BFS, even for multiple sources, for some integer ℓ . This still assumes that we are given a layered sparse 2^{t+6} -cover, and we incur only an extra ℓ -multiplicative increase in complexities.

The three subsections mentioned above assume that we are already given a layered sparse 2^{t+6} -cover. We build those recursively. Concretely, in [Section 4.4](#) and [Section 4.5](#), we show that if we are given a layered sparse 2^{t+6} -cover, we can construct sparse 2^{t+7} -cover. [Section 4.4](#) is mostly a recap on the synchronous algorithm, and [Section 4.5](#) describes the adaptation to the asynchronous environment. The key ingredient in the latter reduces to the task of running several multi-source BFSs in the asynchronous environment, for which we can use the approach developed in [Section 4.1](#) to [Section 4.3](#). Hence, the entire construction works recursively, and the construction for each sparse cover depends on covers of lower radii.

Finally, in [Section 4.6](#), we show how to construct a layered sparse $2^{O(1)}$ -covers and complete the algorithm: having the layered sparse cover, we build the next sparse covers one by one, based on the already built ones. There, we also discuss the complete BFS algorithm and its termination.

We remark that our algorithm—especially much of the terminology and design choices in [Section 4.1](#)—builds on the approach of Awerbuch and Peleg [[AP90a](#)] and [[APSPS92](#)]. In their case, they could rely on some nice properties of the randomized sparse cover constructions of Linial and Saks [[LS93](#)]. Unfortunately, we do not see a black-box replacement of this randomized construction with the deterministic synchronous construction of Rozhon, Ghaffari [[RG20](#)]. However, we arrange the overall algorithm such that the key asynchronous task in building sparse covers boils down to an asynchronous multi-source BFS computation, and we solve that recursively by leveraging lower radii sparse covers. Furthermore, as mentioned before, we need to adapt their approach in some parts and provide some highly nontrivial correctness and complexity analysis.

4.1 Single-source 2^t -thresholded BFS, given a layered sparse 2^{t+6} -cover

Here, we describe a polylogarithmic synchronizer for single-source 2^t -thresholded BFS, assuming that we are given 2^i -covers for i from 1 to $t+6$. In the rest of this subsection, we denote the source of this BFS by s .

4.1.1 Basic definitions

In the synchronous environment, a single-source BFS is implemented as follows: at pulse p , nodes at a distance p from the source send the “join” proposal to all their neighbors. Node v accepts only the first “join” proposal it receives. If it receives several of them at the same pulse, it accepts an arbitrary one. If v received this proposal at the end of the round p , then v determines that it is at distance p from the source.

Now, let us move to the asynchronous world. Let $pulse(v)$ denote the **pulse** of node v , as we define next (we will prove, as an ingredient of the algorithm’s correctness, that these pulse values are indeed equal to the distance of the node from the source). In particular, $pulse(s) = 0$. As in the synchronous case, nodes will send the “join” proposals. A node will accept only the first “join” proposal that it receives. Each node will respond to each “join” proposal it receives, indicating whether it accepts or declines. We define $parent(v)$ as a node from which v received the first “join” proposal, i.e., the proposal which it accepted. The graph formed on edges $(v, parent(v))$ will form a tree, to which we will refer as the **execution tree**.

A node will set its own $pulse(v)$ to $pulse(parent(v)) + 1$. Our goal is devise the algorithm so that $pulse(v)$ will indeed be equal to $dist(v, s)$. That is, we have to ensure that the first “join” proposal that the node at a distance $p+1$ from s will receive will be from a node with pulse p . We will prove that our algorithm satisfies this constraint in the [Lemma 4.10](#).

Definition 4.3. The *level* $\ell(p)$ of the pulse p is ∞ for $p = 0$, and otherwise equal to i such that there exists an integer j satisfying $p = (2j + 1)2^i$. That is, the logarithm of the highest power of 2 that divides p .

Definition 4.4. For any value $p > 0$, let \tilde{p} be the largest number satisfying the following conditions:

- $\ell(\tilde{p}) = \ell(p) + 1$
- $\tilde{p} \leq p - 2^{\ell(p)}$.

The pulse $\text{prev}(p)$ for a pulse p is defined as $\max(\tilde{p}, 0)$. And we define $\text{prev}(0) = 0$.

Definition 4.5. The *host* $h(v)$ of a node v is defined as the maximum-depth ancestor h of v in the execution tree that satisfies $\text{pulse}(h) = \text{prev}(\text{pulse}(v))$. In particular, $h(s) = s$.

4.1.2 The algorithm

Intuitively, a node of pulse p should send its join proposals only after it has received all the join proposals from nodes with pulses up to $p - 1$. Toward this, we make $\text{host}(v)$ and $\text{host}(\text{host}(v))$ gather some safety information, as we describe in the next definition.

Definition 4.6. For a node v , let us call it ***p*-empty** if there are no nodes of pulse p in its subtree in the execution tree. Let us call it ***p*-safe** if it is *p*-empty or if all nodes of pulse up to $p - 1$ in its subtree have already sent their “join” proposals and received answers to all of them (declines/accepts).

For each pulse $p > 0$, and for every node v with $\text{prev}(\text{prev}(p)) \leq \text{pulse}(v) \leq p$, we want to know if it is *p*-empty, and if not, we want to know when it becomes *p*-safe. One can gather this information using a convergecast along the already built partial execution tree: each node simply sends the information to its parent when it receives it. For each pulse p , we propagate the information about nodes being *p*-safe and *p*-empty only up to the first node of level $\text{prev}(\text{prev}(p))$.

How do we use this information? For every pulse p up to 2^t , for every node v with $\text{pulse}(v) = \text{prev}(\text{prev}(p))$, we do the following:

- If v is not $\text{prev}(p)$ -empty, and it got to know that it is $\text{prev}(p)$ -safe, then v will ***p*-register** in all clusters of the $2^{\ell(p)+5}$ -cover that contain v , using the registration abstraction from the [Section 3.2](#). Then, node v is called ***p*-registered**.

Once node v receives confirmation that it is *p*-registered, node v passes the information about $\text{prev}(p)$ -safety to its parent (this information will be passed up to the pulse $\text{prev}(\text{prev}(p))$).

- If $\text{pulse}(v) = \text{prev}(\text{prev}(p))$, then, if v is not $\text{prev}(p)$ -empty but *p*-safe, then it ***p*-deregisters** in all clusters of the $2^{\ell(p)+5}$ -cover that contain v . Then, node v is called ***p*-deregistered**.

After that, node v waits until it receives a *Go_Ahead*(p) from all these clusters.

For a single cluster, the registration for different pulses are disjoint. In the terminology of the [Lemma 2.5](#), we imagine *p*-registrations/*p*-deregistrations/sending *Go_Ahead*(p) as different stages of the algorithm, for different p . It is possible for a node v to have to register/deregister for more than one pulse: v does it for all p , for which $\text{pulse}(v) = \text{prev}(\text{prev}(p))$ (and v is not $\text{prev}(p)$ -empty).

When node v receives *Go_Ahead*(p), it propagates this *Go_Ahead* down the execution tree to all its children with pulse p . When node v of pulse p receives such a *Go_Ahead*(p), it starts sending the messages to its neighbors. This is done for all pulses p up to $2^t - 1$ (so that nodes at distance 2^t from the source learn their pulse, but do not send “join” proposals to their neighbors).

Checking stage. At the end of this process, all nodes at a distance at most 2^t from s will learn their pulses. However, nodes at larger distances will not learn that they are at a distance larger than 2^t from s : they will keep waiting. Notice that in the thresholded BFS, we would like these nodes to output ∞ . To solve this issue, we make one last addition to the algorithm.

Let us use an abstraction from [Section 3.1](#). Let process \mathcal{P} denote “being the source s and becoming 2^t -safe”, so that node v is done with \mathcal{P} when it becomes 2^t -safe if $v = s$, or at the very beginning, if $v \neq s$. Let each node learn when its 2^t -neighborhood is done with \mathcal{P} . When node v learns that this, then, if it was reached by the BFS, it knows its pulse. Otherwise, it learns that $\text{dist}(v, s) > 2^t$. Indeed, if $\text{dist}(v, s) \leq 2^t$, then v and s are in some cluster of 2^t -cover together, but then v must have learned its pulse before s becomes 2^t -safe and v learns that its 2^t -neighborhood is done with \mathcal{P} .

4.1.3 Analysis

First, in [Lemma 4.8](#), we show that if two nodes of the same pulse are “nearby”, then their hosts are also pretty close to each other. We first need some more basic facts.

Lemma 4.7. *For any pulse $p > 0$, the following holds: (a) $p - \text{prev}(p) \leq 3 \cdot 2^{\ell(p)}$. (b) $p - \text{prev}(\text{prev}(p)) \leq 9 \cdot 2^{\ell(p)}$. Furthermore, this implies that we have the following for any node v with pulse $p > 0$: (c) $\text{dist}(p, h(v)) \leq 3 \cdot 2^{\ell(p)}$. (d) $\text{dist}(p, h(h(v))) \leq 9 \cdot 2^{\ell(p)}$.*

Proof. For the first claim, numbers $p - 2^{\ell(p)}$ and $p - 3 \cdot 2^{\ell(p)}$ are divisible by $2^{\ell(p)+1}$, but only one of them can be divisible by $2^{\ell(p)+2}$, so $\text{prev}(p) \geq p - 3 \cdot 2^{\ell(p)}$.

For the second claim, if $\text{prev}(p) = 0$, then $\text{prev}(\text{prev}(p)) = 0 = \text{prev}(p) \geq p - 3 \cdot 2^{\ell(p)} > p - 9 \cdot 2^{\ell(p)}$. Otherwise, $\ell(\text{pulse}(p)) = \ell(p) + 1$, and $\text{prev}(p) - \text{prev}(\text{prev}(p)) \leq 3 \cdot 2^{\ell(p)+1}$. As $p - \text{prev}(p) \leq 3 \cdot 2^{\ell(p)}$, we can add these inequalities to obtain $p - \text{prev}(\text{prev}(p)) \leq 9 \cdot 2^{\ell(p)}$. \square

Lemma 4.8. *For any two nodes u and v of pulse $p > 0$ in the execution tree such that $\text{dist}(u, v) \leq 2^{\ell(p)+3}$ in the graph, we have that $\text{dist}(h(u), h(v)) \leq 2^{\ell(p)+4}$.*

Proof. We have $\text{dist}(h(u), h(v)) \leq \text{dist}(h(u), u) + \text{dist}(h(v), v) + \text{dist}(u, v) \leq 3 \cdot 2^{\ell(p)} + 3 \cdot 2^{\ell(p)} + 2^{\ell(p)+3} = 14 \cdot 2^{\ell(p)} < 2^{\ell(p)+4}$. \square

Now, we get to the core lemma of the correctness proof.

Lemma 4.9. *For a pulse $p > 0$ consider any two nodes v_1, v_2 with pulse $\text{prev}(\text{prev}(p))$, such that $\text{dist}(v_1, v_2) \leq 2^{\ell(p)+5}$, v_1 is not p -empty, and v_2 is not $\text{prev}(p)$ -empty. Then, for any cluster C of the $2^{\ell(p)+5}$ -cover that includes both of v_1 and v_2 , the following holds:*

- v_1 will p -deregister in C only after v_2 p -registers in C .
- v_1 will receive $\text{Go_Ahead}(p)$ in C only after v_2 p -deregisters in C .

Proof. The proof is by induction based on the pulse number. If $\text{prev}(\text{prev}(p)) = 0$, then there is only one node with pulse $\text{prev}(\text{prev}(p))$: the source s . Then $v_1 = v_2 = s$, and the statement holds.

Consider the case $\text{prev}(\text{prev}(p)) > 0$, and suppose that the statement holds for all smaller values of p . Consider any such cluster C . Suppose that v_1 is about to p -deregister in it. Consider the hosts of v_1, v_2 : $h(v_1), h(v_2)$. By [Lemma 4.8](#) we have $\text{dist}(h(v_1), h(v_2)) \leq 2^{\ell(p)+6}$. It follows that there is some cluster C_1 of $2^{\ell(p)+6}$ -cover, in which both $h(v_1)$ and $h(v_2)$ are present.

Let us start by proving the first claim. As v_1 has already p -deregistered in C , some nodes of pulse $\geq \text{prev}(p)$ in the subtree of v_1 have already sent their messages. This is possible only after $h(v_1)$ has received $\text{Go_Ahead}(\text{prev}(p))$ in C_1 , which, by the induction hypothesis, is possible only after $h(v_2)$ $\text{prev}(p)$ -deregisters in C_1 . This is possible only after v_2 p -registers in C .

Now the second claim. Consider the moment when v_1 is about to receive $Go_Ahead(p)$. This means that v_1 has already p -deregistered. By the first claim, v_2 has p -registered before that. By [Lemma 3.4](#), as v_2 has p -registered before v_1 has p -deregistered, v_1 will receive $Go_Ahead(p)$ only after v_2 p -deregisters. \square

Note that by now, we have only used that $pulse(v) = pulse(parent(v)) + 1$. Now, we will show that our execution tree is actually a BFS tree.

Lemma 4.10. *The above algorithm computes a 2^t -thresholded BFS: For every node v such that $dist(v, s) \leq 2^t$, we have $pulse(v) = dist(v, s)$. In other words, this execution tree is indeed a BFS tree. And for any node v such that $dist(v, s) > 2^t$, the node learns that its distance is greater than 2^t .*

Proof. The second part of the lemma statements is guaranteed by the checking stage. Here, we focus on the first part. We will prove by induction by pulse p the following statement: for every node v with $dist(v, s) = p$ holds $pulse(v) = p$. This statement is trivially true for $p = 0$.

Suppose that this is proved for all pulses up to p ; we prove it for pulse $p + 1$. Consider any node v with distance $p + 1$ to the root. We show that the first “join” proposal it receives will be from a node with pulse p . Suppose the contrary. Let v_1 be a neighbor with $dist(v_1, s) = p$, and suppose that v received the first “join” proposal from some node v_2 with $pulse(v_2) = p_1 > p$. Consider the moment when v has already received a “join” proposal from v_2 but has not yet received it from v_1 . Let x be the integer from the segment $[p + 1, p_1]$ divisible by the largest power of 2 (with largest $\ell(x)$), and let w be the ancestor of v_2 with pulse x . As v was reached in a subtree of s , w has already sent at least some messages of pulse x . This is possible only after w receives $Go_Ahead(x)$, which is possible only after $h(h(w))$ receives $Go_Ahead(x)$.

Now note that $pulse(h(w)) \leq p$. Indeed, $pulse(h(w)) \leq w - 2^{\ell(w)}$, and as number $x - 2^{\ell(x)}$ is divisible by larger power of 2 than x is, it cannot be in $[p + 1, p_1]$ by our choice of x . So, $pulse(h(w)) \leq x - 2^{\ell(x)} \leq p$. Let y_1 be the ancestor of v_1 with pulse $prev(prev(x))$, y_2 be the ancestor of v_2 with pulse $prev(prev(x))$ (so that $y_2 = h(w)$). As $pulse(h(w)) \leq p$, y_1 is not $prev(x)$ -empty. y_2 is not x -empty by the choice of x . Consider the distance between y_1, y_2 . $dist(y_1, y_2) \leq dist(y_1, v) + dist(v, y_2) \leq (1 + (p - prev(prev(x)))) + (1 + (p_1 - prev(prev(x)))) = 2 + 2(x - prev(prev(x))) + (p - x) + (p_1 - x)$. By [Lemma 4.7](#), $x - prev(prev(x)) \leq 9 \cdot 2^{\ell(p)}$. $p - x \leq -1$ as $x \in [p + 1, p_1]$, and $p_1 - x \leq 2^{\ell(p)} - 1$ as otherwise we could have chosen $x + 2^{\ell(p)}$, which is divisible by higher power of 2. So, $dist(y_1, y_2) \leq 2 + 18 \cdot 2^{\ell(p)} + 2^{\ell(p)} - 2 = 19 \cdot 2^{\ell(p)} \leq 2^{\ell(p)+5}$. Then, by [Lemma 4.9](#), $y_2 = h(h(w))$ can receive $Go_Ahead(x)$ only after y_1 has x -deregistered, which is not possible until v_1 has sent its messages of pulse p , contradiction. \square

Theorem 4.11. *The 2^t -thresholded single-source BFS given a layered sparse 2^{t+6} -cover finishes in time $O(2^t \cdot \log^8 n)$ and uses message complexity $O(m \log^5 n)$.*

We discuss time and message complexity parts separately.

Analysis of the time complexity. Next, we analyze the time complexity of our single-source BFS algorithm and prove the time complexity part of [Theorem 4.11](#). We first discuss some helper lemmas.

We divide all our operations into stages, one stage \mathcal{S}_p for each pulse p . To \mathcal{S}_p we include all the p -registrations, p -deregistrations, sending $Go_Ahead(p)$ s, collecting information about p -safety, and sending “join” proposals from nodes of pulse $p - 1$ to nodes of pulses p : basically, everything related to pulse p . It’s clear that each stage depends only on the previous pulses, so the runtime of the algorithm is bounded by the isolated time complexities of each stage.

Lemma 4.12. *The isolated time complexity of \mathcal{S}_p is $O(2^{\ell(p)} \log^7 n)$.*

Proof. Each edge is in $O(\log^4 n)$ clusters of the sparse $2^{\ell(p)+5}$ -cover. In the execution tree, each edge is involved in collecting the information about p -safety only $O(1)$ times. So, we may say that each edge is involved in $O(\log^4 n)$ separate tasks for \mathcal{S}_p , and multiply the runtime by this factor in the end, by [Corollary 2.3](#).

After all nodes of pulses at most $p - 1$ have sent their “join” proposals, all of them will be acknowledged in $O(1)$ time units. All nodes of pulse $prev(p)$ will receive information about their p -safety/ p -emptiness in at most $3 \cdot 2^{\ell(p)}$ time units after that. All these nodes (which are not p -empty) will p -register in the corresponding clusters in at most $O(2^{\ell(p)} \cdot \log^3 n)$ times units. All nodes of pulse $prev(prev(p))$ will receive information about their p -safety in at most $6 \cdot 2^{\ell(p)}$ time units after that. All these nodes will p -deregister in the corresponding clusters in at most $O(2^{\ell(p)} \cdot \log^3 n)$ times units.

By [Lemma 3.5](#), all clusters of the $2^{\ell(p)+5}$ -cover will then send $Go_Ahead(p)$ s, which will be received in $O(2^{\ell(p)} \cdot \log^3 n)$ time units. They will then be propagated to all the nodes of pulse p in at most $9 \cdot 2^{\ell(p)}$ after that. The total time complexity of this is $O(2^{\ell(p)} \cdot \log^3 n)$

Taking into the account that there are $O(\log^4 n)$ tasks for each edge, the isolated time complexity of \mathcal{S}_p becomes $O(2^{\ell(p)} \cdot \log^7 n)$. \square

Lemma 4.13. *For any integer t , $\sum_{p=1}^{2^t} 2^{\ell(p)} = O(2^t \cdot t)$.*

Proof. The number of values p such that $\ell(p) = k$ is $\leq 2^{t-k}$. Hence, $\sum_{p=1}^{2^t} 2^{\ell(p)} \leq \sum_{k=0}^t 2^k \cdot 2^{t-k} = (t+1)2^t = O(2^t \cdot t)$. \square

Proof of the time complexity part of [Theorem 4.11](#). The total isolated time complexity of all stages \mathcal{S}_i becomes $\sum_{p=1}^{2^t} O(2^{\ell(p)} \cdot \log^7 n) = O(2^t \cdot \log^8 n)$. The isolated time complexity of the checking stage is, by [Theorem 3.1](#), $O(2^t \log^7 n)$. So, the entire runtime is $O(2^t \cdot \log^8 n)$. \square

Analysis of the message complexity. Next, we analyze the message complexity of our single-source BFS algorithm and provide the proof for the message complexity part of [Theorem 4.11](#). First, we need to prove a small lemma.

Lemma 4.14. *For any p_1 , there exist $O(t)$ pulses $p \leq 2^t$ such that $prev(prev(p)) \leq p_1 \leq p$.*

Proof. It is enough to show that there are $O(1)$ pulses of each particular level. For a pulse p of given level ℓ , we know $prev(prev(p)) \geq p - 9 \cdot 2^\ell$. So, for any such pulse p , we must have $p - 9 \cdot 2^\ell \leq p_1 \leq p$. There can be at most 10 such pulses. \square

Proof of the message complexity part of [Theorem 4.11](#). The number of messages sent for building the execution tree is $O(m)$. The number of messages propagated in the execution tree is $O(nt) = O(n \log n)$, as, by [Lemma 4.14](#), there are messages of only $O(t)$ pulses propagated through each edge. Now, consider messages in the clusters of the sparse covers.

Consider some level ℓ . For a pulse p with $\ell(p) = \ell$, node v with pulse $prev(prev(p))$ will p -register in the clusters of the sparse $2^{\ell+5}$ -cover only if it has some children of pulse $prev(p)$, and therefore has at least 2^ℓ nodes on the path to such a child. As these 2^ℓ nodes are unique for each such node v , it follows that for a given level ℓ , the total number of nodes who will p -register for some pulse p with $\ell(p) = \ell$ is $O(\frac{n}{2^\ell})$. Such nodes will register in clusters of $2^{\ell+5}$ covers and therefore take $O(2^\ell \log^3 n)$ messages to the root and back. Every node is in $O(\log n)$ clusters, so, there will be $O(2^\ell \log^3 n \cdot \log n \cdot \frac{n}{2^\ell}) = O(n \log^4 n)$ messages corresponding to registration at a given level, and

the total number for all levels is $O(n \log^5 n)$. We assume that the graph is connected, so $n = O(m)$, and thus message complexity is $O(m \log^5 n)$.

Finally, the number of messages used in the checking stage is $O(m \log^4 n)$. So, the total number of messages is $O(m \log^5 n)$. \square

4.2 Multi-source 2^t -thresholded BFS, given a layered sparse 2^{t+6} -cover

Here, we describe a multi-source variant of [Section 4.1](#). The entire algorithm works the same way as before, and all the proofs remain valid, except for one critical issue: the base case of the [Lemma 4.9](#). Notice that [Lemma 4.9](#) has to hold even if $\text{prev}(\text{prev}(p)) = 0$ (that is, for the case when both v_1, v_2 are sources). In the case of a single source BFS there was only one source, so the statement was true for the base case automatically. Now we have to take care of this algorithmically.

We will make the statement hold for sources by changing the registration procedure for them. Consider any pulse $0 < p \leq 2^t$ with $\text{prev}(\text{prev}(p)) = 0$ and any cluster C of $2^{\ell(p)+5}$ cover.

Registration. We will make sure that all sources in C have p -registered, with a convergecast (note that we do not require them to be $\text{prev}(p)$ -safe). After the root of the cluster learns that all its sources have been p -registered, it propagates this information down to all the sources. A source may send its messages only after it has registered for all pulses p with $\text{prev}(\text{prev}(p)) = 0$ in all clusters of $2^{\ell(p)+5}$ covers and received a confirmation from all of them.

Deregistration. After source learns that it is p -safe, it will p -deregister. This p -deregistration will be collected in the cluster in the same manner, with a simple convergecast.

Sending *Go Ahead*. After the root of the cluster learns that all the sources in it have p -deregistered, it propagates down the *Go Ahead*(p)s. This way, the condition of the lemma is easily satisfied: for every cluster, all the registrations happen before any deregistration is even possible.

Theorem 4.15. *The 2^t -thresholded multi-source BFS given a layered sparse 2^{t+6} -cover finishes in time $O(2^t \cdot \log^8 n)$ and takes $O(m \log^5 n)$ messages.*

Before providing the proof of this statement, we discuss a helper lemma.

Lemma 4.16. *The number of pulses p with $0 < p \leq 2^t$ and $\text{prev}(\text{prev}(p)) = 0$ is $O(t)$.*

Proof. We show that for each level $\ell \leq t$, there are only $O(1)$ pulses p with $\ell(p) = \ell$ with $\text{prev}(\text{prev}(p)) = 0$. Indeed, if $\ell(p) = \ell$, then we get $0 = \text{prev}(\text{prev}(p)) \geq p - 9 \cdot 2^\ell$, so $p \leq 9 \cdot 2^\ell$, there are only $O(1)$ such p . \square

Proof of [Theorem 4.15](#). Time complexity. We can view this convergecast simply as an extra stage of the algorithm, the very first one. If we had a copy of each edge for every cluster in which it's present, this stage would run in $O(2^t \cdot \log^3 n)$. As each edge is present in $O(\log^4 n)$ clusters per cover, and we are doing this for $O(\log n)$ pulses, the total complexity of this stage, by [Corollary 2.3](#), is $O(2^t \cdot \log^8 n)$. As the remaining part also works in $O(2^t \cdot \log^8 n)$, the resulting complexity is $O(2^t \cdot \log^8 n)$.

Message complexity. For a fixed pulse p , the total number of messages spent on collecting this info for all the clusters is $O(m \log^4 n)$, as each edge is in $O(\log^4 n)$ clusters.

As the total number of pulses p with $\text{prev}(\text{prev}(p)) = 0$ is $O(t) = O(\log n)$, the total message complexity of this process is $O(m \log^5 n)$. The message complexity of the remaining part of the algorithm remains $O(m \log^5 n)$, as in [Theorem 4.11](#), so we get a total message complexity of $O(m \log^5 n)$. \square

4.3 Multi-source $2^t \cdot \ell$ -thresholded BFS, given a layered sparse 2^{t+6} -cover

Let us assume that we are given 2^i -covers for all i from 1 to $t + 6$. In [Section 4.2](#), we discussed how to perform a multiple-source 2^t -thresholded BFS. Now, we discuss how we do a multi-source $2^t \cdot \ell$ -thresholded BFS, where ℓ is some positive integer, using a layered sparse 2^{t+6} -cover.

Let us divide our BFS into ℓ stages. The T -th of these stages will have to be a 2^t -thresholded BFS with nodes of pulse $T \cdot 2^t$ as sources. Now, Let us use an abstraction from [Section 3.1](#). Let process \mathcal{P}_T denote “doing the BFS up to 2^t as a source of the T -th stage”, so that node v completes \mathcal{P}_T when it completes this BFS or learns that it is not a source in the T -th stage. For each $T < \ell$, let each node learn when its 2^t -neighborhood is done with \mathcal{P}_T . When node v learns this, he has two options. If v was reached by a BFS, then, if its pulse is $T \cdot 2^t$, then it is a source at the $T + 1$ -st stage; otherwise, it is not. If it has not received a pulse, then this means that there are no sources of the T -th stage in its 2^t -neighborhood, as if there were some, v would have received a pulse from them in the T -th stage. Therefore, it cannot be at a distance precisely $T \cdot 2^t$ from the sources.

Theorem 4.17. *The $2^t \cdot \ell$ -thresholded multi-source BFS given a layered sparse 2^{t+6} -cover finishes in time $O(2^{t\ell} \cdot \log^8 n)$ and takes $O(m\ell \log^5 n)$ messages.*

Proof. Time complexity. We have ℓ stages, where each stage is dependent only on the previous ones, so we can apply [Lemma 2.5](#). The isolated time complexity of each stage is $O(n \log^8 n)$, and the isolated time complexity of the convergecast is $O(n \log^7 n)$. We get a total runtime of $O(n\ell \log^8 n)$.

Message complexity. Each of ℓ stages requires $O(m \log^5 n)$ messages. As there are ℓ stages, the total message complexity is $O(m\ell \log^5 n)$. \square

Remark 4.18. *Note that we also get an algorithm d -thresholded BFS for any $d \leq 2^t \cdot \ell$ with the same complexity. That is, this algorithm works not only for the multiples of 2^t .*

4.4 Synchronous deterministic construction of the d -cover

So far, we have assumed that we are given sparse covers. In this subsection, we review the synchronous construction of Rozhon and Ghaffari [[RG20](#)]. In the next subsection, we explain how to adapt this construction to the asynchronous environment, using our asynchronous multi-source BFS.

We will start by introducing a deterministic algorithm for constructing the d -cover in $O(d \cdot \text{polylog}(n))$ rounds in a synchronous environment. For that, we will need a network decomposition.

Definition 4.19. (*k -separated Weak Diameter Network Decomposition*) *Given a graph $G = (V, E)$, we define a $(\mathcal{C}, \mathcal{D})$ k -separated weak diameter network decomposition to be a partition of G into vertex-disjoint graphs $G_1, G_2, \dots, G_{\mathcal{C}}$ such that for each $i \in \{1, 2, \dots, \mathcal{C}\}$, we have the following property: the graph G_i is made of a number of vertex-disjoint clusters $X_1, X_2, \dots, X_{\ell}$, so that:*

- For any X_a and any two vertices $v, u \in X_a$, $\text{dist}(u, v) \leq D$ in graph G .
- For any two X_a, X_b with $j \neq l$ and any $u \in X_a, v \in X_b$ holds $\text{dist}(u, v) > k$.

4.4.1 Synchronous construction of k -separated Network Decomposition

Before stating the result we will use, we recall the notion of Steiner trees. A Steiner tree of a cluster is a tree with nodes labeled as *terminal* and *nonterminal*; the aim is to connect all terminals, possibly using some nonterminals.

Theorem 4.20. [Rozhon and Ghaffari [RG20]] *There is an algorithm that, given a value k known to all nodes, in $O(k \log^{10} n)$ communication rounds outputs a k -separated weak-diameter network decomposition of G with $O(\log n)$ color classes, each one with $O(k \cdot \log^3 n)$ weak-diameter in G .*

Moreover, for each color and each cluster \mathcal{C} of vertices with this color, we have a Steiner tree $T_{\mathcal{C}}$ with radius $O(k \cdot \log^3 n)$ in G , for which the set of terminal nodes is equal to \mathcal{C} . Furthermore, each edge in G is in $O(\log^4 n)$ of these Steiner trees.

To remain self-contained, we provide a (simplified) description of the algorithm of [RG20] in the [Appendix C](#). Our asynchronous construction will refer to the language used in this description. For proof of the correctness of their algorithm, we refer to [RG20].

4.4.2 Constructing d -cover in the synchronous setting

Theorem 4.21. *There is a synchronous algorithm that, given a value d that is known to all nodes, in $O(d \log^{10} n)$ communication rounds outputs a sparse d -cover of this graph, together with the Steiner trees of its clusters, such that each edge appears in only $O(\log^4 n)$ cluster trees.*

Proof. We start by constructing a $2d + 1$ separated weak network decomposition in $O(d \log^{10} n)$ rounds, according to [Theorem 4.20](#). Then, for each color separately, we expand each of its clusters to its d -neighborhood. As different clusters of the same color are at least $2d + 1$ apart, the clusters remain disjoint, so each node will be in $O(\log n)$ clusters (at most one per color). Additionally, each edge will join at most one cluster tree, so each edge still appears in only $O(\log^4 n)$ cluster trees. \square

4.5 Asynchronous construction of sparse 2^{t+7} -cover, given a layered sparse 2^{t+6} -cover

The algorithm described in [Section 4.4](#) for constructing sparse d -covers consists of simple multi-sources BFSs up to distance $O(d)$. And we already know how to do that, via the approach we described in [Section 4.3](#). Using this connection, below we show that we can run the sparse cover construction in the asynchronous environment, resulting in the following statement:

Theorem 4.22. *There is an asynchronous algorithm that, given a layered sparse 2^{t+6} -cover,*

- *Constructs sparse 2^{t+7} -cover together with the Steiner trees of the clusters*
- *Lets each node learn when all the Steiner trees in which it is have been built*
- *Works in time $O(2^t \cdot \log^{11} n)$ and takes $O(m \log^8 n)$ messages*

Proof. We start by providing an asynchronous algorithm to build k -separated weak decomposition from the [Theorem 4.20](#). We build one color at a time. We let each node start to build the next color only after it is done with the previous colors. We will formalize “being done” with color and the way to learn this later.

Consider the algorithm for building the clusters of a particular color from [Lemma C.1](#). It contains $O(\log n)$ phases; we will synchronize each phase separately and let each node enter the next phase only after it learns that it is done with the previous phases. We will formalize “being done” with a phase and the way to learn this later.

One phase, as in the synchronous algorithm, consists of $O(\log^2 n)$ steps. We will synchronize each of these steps separately and let each node enter the next step only after it is done with the previous steps. We will formalize “being done” with a phase and the way to learn this later.

Denote $2 \cdot 2^{t+7} + 1 = d$. One step consists of two substeps.

1. d -thresholded BFS from all blue nodes. According to the [Theorem 4.17](#), we can do it in an asynchronous environment in just $O(2^t \log^8 n)$ time and $O(m \log^5 n)$ messages. At the end of this BFS, by the definition of the d -thresholded BFS, every node will be reached or will learn that its distance from any blue node is larger than d .
2. In the same way as in the original algorithm from [Lemma C.1](#), the roots of all Steiner trees will collect the number of proposing red nodes and make corresponding decisions, with a convergecast by the Steiner tree. A node will propagate the information up the tree only after it is done with the previous substep.

We say that node learns that it is done with a step as soon as it receives (and propagates down the Steiner tree) all the decisions. When a node is done with a step, it knows whether it is a source for the next stage (it knows whether it is a blue node). We say that node learns that it is done with a phase once it learns that it is done with all its steps. We say that a node is done with a color once it learns that is done with all the phases of the algorithm for building that color. After a node knows that it is done with building all $O(\log n)$ colors, it starts performing a 2^{t+7} -thresholded BFS for every cluster that it is a part of, as in [Theorem 4.21](#).

We now discuss time complexity. Let us bound the isolated time complexity of each step of the algorithm. Isolated time complexity of the BFS substep is $O(2^t \cdot \log^8 n)$, of the convergecast substep is $O(2^t \log^6 n)$, so the entire step takes $O(2^t \log^8 n)$. As there are $O(\log^2 n)$ steps per phase and $\log n$ phases per color, we get the resulting runtime of $O(2^t \log^{11} n)$.

We now discuss message complexity. The message complexity of a single step is $O(m \log^5 n)$ per BFS plus $O(m \log^3 n)$ per convergecast, as each edge is in at most $\log^3 n$ clusters for a particular color, which is $O(m \log^5 n)$ in total. As there are $O(\log^2 n)$ steps per phase and $\log n$ phases per color, we get the resulting message complexity of $O(m \log^8 n)$. \square

4.6 The complete BFS algorithm in $\tilde{O}(D)$ time and $\tilde{O}(m)$ messages

Theorem 4.23. *There is an algorithm for a single-source BFS that terminates in time $O(D \log^{11} n)$, where D is the diameter of the graph and uses $O(m \log^{10} n)$ messages.*

Proof. The algorithm works in several steps.

Step 1: Initialization – constructing layered sparse 2^6 -cover. We learned how to construct sparse 2^{t+7} cover given sparse 2^i -covers for $5 \leq i \leq t + 6$ (note that we do not ever use smaller sparse 2^i -covers for $i \leq 4$). Now the question is, how to construct 2^5 -cover and 2^6 -cover.

For a constant c , the synchronous algorithm for constructing c -cover works in $O(\log^{10} n)$ rounds. So, we can afford to synchronize it with a simple α -synchronizer: just sending a message through every edge at each pulse. This takes $O(\log^{10} n)$ time and $O(m \log^{10} n)$ messages. Every node will wait until it learns that it is done with this step before proceeding to the next one. See also [Appendix A](#) for a more general description of the α synchronizer.

Step 2: One iteration. Every iteration will consist of two parts.

1. Running 2^t -thresholded BFS from all the sources. Each node will wait until it is done with constructing layered sparse 2^{t+6} -cover before it can participate in this 2^t -thresholded BFS.
2. Constructing sparse 2^{t+7} -cover. We will simply apply the algorithm from [Theorem 4.22](#), with the same restriction: a node will be allowed to participate in it only after it is done with constructing layered sparse 2^{t+6} -cover.

Step 3: Termination. If all the nodes knew D , we could simply run this algorithm for $\lceil \log_2 D \rceil$ iterations. However, they do not know it. Even though every node will learn the distance to the root in the first $\lceil \log_2 D \rceil$ iterations, they will keep doing these iterations, blowing up the time and message complexity: since running t -th iteration takes $O(2^t \cdot \log^{11} n)$ time and $O(m \log^8 n)$ messages for constructing the next sparse cover, we cannot afford to simply run $\Theta(\log n)$ iterations. There are two strategies to solve this problem. In both of them, we are looking for a way to inform every node when all the nodes have already been reached by the BFS.

Approach 1 for step 3. In [Section 2.1](#), we mentioned that the sparse d -covers that we are constructing satisfy the following condition: for each node v , there exists a cluster such that all nodes u for which $\text{dist}(u, v) \leq d$ are included in this cluster. Thus, by the $\lceil \log_2 D \rceil$ -th iteration, some cluster of some sparse cover that we have constructed will contain all the nodes of the graph.

For each cluster of the sparse 2^t -cover, add an extra convergecast: checking whether it contains all the nodes of the graph. For each cluster, every node in it checks whether all its neighbors are in the same cluster and propagates this information up. After the root of the cluster of this cover learns whether this cluster contains all the nodes of the graph, it will broadcast this information to all the nodes in the cluster. Every node waits until it receives such verdicts from all its clusters. Note that if no cluster contains all the nodes, every node will receive only “NO”s; otherwise, every node will receive a “YES” from the cluster that contains all the nodes.

If some cluster of the 2^t -cover contains all the nodes, there is no point in constructing sparse 2^{t_1} -covers for larger t_1 : we can simply use the 2^t -cover instead, so we will not have to waste extra time and messages on constructing those covers. Additionally, after each iteration starting from this one, for every cluster that contains all nodes, we will collect, with a convergecast, whether all the nodes have already been reached by the BFS by the end of that iteration. When the root of the cluster detects this, it will broadcast this to all the nodes in the cluster. Every node waits for all these responses before proceeding to the next iteration. If all nodes have already been reached by the BFS, every node will learn this, and they will simply not go to the next iteration.

Approach 2 for step 3. Let us add a slight modification to the d -thresholded BFS. Each node at a distance d from the source will check whether any of its neighbors is at a distance larger than d from the source. They will send this information up the execution tree with a convergecast. This way, a source will learn whether the farthest distance from it is larger than d . Then, the source will broadcast this information down to all the nodes at its execution tree.

This way, every node will learn whether the largest distance from the source to some other node is at least d : if it was reached by BFS, it will learn this from the source. Otherwise, it already knows that it is at a distance larger than d from the source.

So, at the end of the 2^t -thresholded BFS, every node will learn whether they have to proceed to the next iteration (2^{t+1} -thresholded BFS and construct the next sparse cover). We will terminate in $\lceil \log_2 D \rceil + O(1)$ iterations.

Time complexity. The isolated time complexity of the initialization is $O(\log^{10} n)$; the isolated time complexity of the t -th iteration is $O(2^t \cdot \log^8 n)$ for BFS plus $O(2^t \cdot \log^{11} n)$ per cover construction. For the first approach, the convergecasts in the cluster trees add only $O(2^t \log^7 n)$ time messages per iteration; for the second approach, the convergecast in the execution tree adds only $O(2^t)$ time per iteration. In both cases, the total time complexity becomes $O(D \cdot \log^{11} n)$, as we are doing $O(\lceil \log_2 D \rceil + O(1))$ iterations.

Message complexity. The total message complexity of the initialization is $O(m \log^{10} n)$; the message complexity of each iteration is $O(m \log^5 n)$ for BFS, $O(m \log^8 n)$ for sparse cover construction. plus $O(m)$ for convergecast. For the first approach, the convergecasts in the cluster trees add

only $O(m \log^4 n)$ messages per iteration; for the second approach, the convergecast in the execution tree adds only $O(m)$ messages per iteration. In both cases, the total message complexity becomes $O(m \log^{10} n)$. \square

Theorem 4.24. *There is an algorithm for a multiple-source BFS that terminates in time $O(D_1 \log^{11} n)$, where D_1 is the largest of the values $\text{dist}(v, S)$ over all nodes v , and uses $O(m \log^{10} n)$ messages.*

Proof. By applying **Approach 1** of **Theorem 4.23**, we would get an algorithm that terminates in $O(D \log^{11} n)$ time, where D is the diameter of the graph, and uses $O(m \log^{10} n)$ messages. But we can do better.

Let us start with some intuition: In the case of the multi-source BFS, the first two stages of the algorithm from **Theorem 4.23** work the same. The differences start with the termination. Let D_1 be the largest of the values $\text{dist}(v, S)$ over all nodes v , where S is the set of sources. In the synchronous environment, our algorithm would run in time $O(D_1)$. However, the termination **Approach 1** from **Theorem 4.23** only guarantees that we finish in time $\tilde{O}(D)$, not $\tilde{O}(D_1)$, while for the **Approach 2** it is crucial that there is only one source so that all the nodes will be in the same execution tree when they are reached, and we can run a convergecast. Fortunately, there is a nice modification of **Approach 2**, that allows to run the multi-source in $\tilde{O}(D_1)$ in this case too.

We now present the proof. We will once again do the BFS and build the sparse covers in iterations, but here we would need an extra trick.

Let us call all nodes **alive** initially. If a source runs a d -thresholded BFS, then it will collect the same information as in the **Approach 2** from the **Theorem 4.23**: from the nodes in its execution subtree at depth d it will collect whether they have neighbors that have not been reached by the BFS. A source becomes **dead**, if the depth of its execution subtree is less than d , or if there is no node of depth d in this subtree that has a neighbor not reached by the BFS. If a source becomes dead, it tells this to all the nodes in its execution subtree, and all of them will become dead too. A node that has not been reached by the BFS knows that it remains alive, and a node that is in a subtree of a source that stays alive also stays alive.

Let us say that we did d -thresholded BFS. Consider all the alive nodes and sources. The distance from every alive node to the closest alive source has not changed. So, we can run the next iteration only on these alive nodes.

So, the algorithm would work as follows: each node keeps track of whether it is dead or alive for a particular iteration, and only the alive nodes are involved in constructing the corresponding sparse cover (which will be a sparse cover of the subgraph on only the alive nodes). After $\lceil \log_2 D_1 \rceil$ iterations, all nodes will become dead, and the algorithm will terminate.

The time and message complexity bounds are proved in the same way as in **Theorem 4.23**. \square

5 Polylogarithmic synchronizer for event-driven algorithms

In this section, we describe the generalization that extends the scheme described in **Section 4** to a synchronizer for a general event-driven synchronous algorithm, thus proving (formal variants of) **Theorem 1.1**. The formal theorem statements appear later as **Theorem 5.4** and **Theorem 5.5**.

5.1 The general structure of the event-driven algorithms

Let us briefly recall the interpretation of event-driven synchronous algorithms, as discussed in **Appendix B**. In event-driven algorithms, nodes cannot refer to the round numbers, i.e., the nodes cannot explicitly access the clock value. A node may send a message only because of receiving some

messages (or sending some prior messages and getting their acknowledgement). In other words, a node cannot simply wait for several rounds before sending a particular message.

In the synchronous environment, every message is sent at a certain pulse. The messages of pulse 0 are sent by some **initiators**; these messages can be perceived as triggered by the outside environment. Every message of pulse $p > 0$ has to be triggered by some message of pulse $p - 1$. It can be triggered in two possible ways:

- Node v has received some messages of pulse $p - 1$, which triggers sending a message of pulse p
- Node v has sent some messages of pulse $p - 1$, which triggers sending a message of pulse p

5.2 The setup

Let \mathcal{A} denote the synchronous algorithm that we are trying to run in the asynchronous environment. As before, let $T(\mathcal{A})$ denote its runtime, and $M(\mathcal{A})$ denote its message complexity.

We will create a virtual node (v, p) for any node v sending one or more messages at pulse $p > 0$. Consider any such node v , sending a message m with pulse $p > 0$. This message is triggered by some message of pulse $p - 1$. Choose any of them, let's say, message m_1 of pulse $p - 1$, from node u . Then, let's declare node $(u, p - 1)$ as a **parent** of the node (v, p) . Note that it's possible that $u = v$, if v sending a message at pulse p is triggered by v sending a message at pulse $p - 1$.

For any p for which node u is sending a message at pulse p , node u will be responsible for performing all the actions of the virtual node (u, p) . We describe the exact actions later.

From now on, the setting is very similar to the setting in [Section 4.1](#); we define all the same notions, but for the virtual nodes now. Let's define the **pulse** of a virtual node (u, p) . In some sense, our goal is to ensure that $pulse((u, p)) = p$.

Node v is keeping track of the messages that it has received/sent. As soon as it is triggered by some of them (and some other actions of our synchronizer) to send some message m at pulse p , it creates a virtual node (v, p) . (v, p) picks any of the messages of pulse $p - 1$ that it was triggered by, say, m_1 , sent from u , and declares $(u, p - 1)$ as its **parent**. Virtual node (v, p) will then set $pulse((v, p)) = pulse((u, p - 1)) + 1$.

So far, everything looks very similar to the [Section 4.1](#), but there are several important distinctions. Before even discussing the time and message complexity and the way to actually simulate the work of these virtual messages, we have to understand what the desired correctness guarantees are. In the case of BFS, we had to show that the execution tree will actually be the BFS tree: that the pulse that each node receives will be equal to its distance from the closest source. What do we have to show in this case?

Note that each node acts as a state machine. It does not know anything except what other nodes tell it. So, its messages depend only on the messages it receives from them. If it receives these messages in some different order, it might decide to send very different messages. For example, for the BFS, for the correct functioning of the algorithm, it is important that the first "join" request that a node v at a distance $p > 0$ from the sources receives is from a node at a distance $p - 1$ from the sources. If this condition is not satisfied, the algorithm will be completely broken.

Consider a concrete example. Let us say that the node v in the synchronous environment receives messages m_1, m_2, m_3 at pulses $p, p + 1, p + 2$ correspondingly. After receiving m_2 , it generates a new message r . Then, it may be important for v to receive messages m_1, m_2, m_3 precisely in this order. Indeed, it might be the case that v would want to generate some other message r_1 if it has received only message m_2 (without m_1), and we have to avoid this. Similarly, it might be the case that v would want to generate some other message r_2 if it has received only messages m_1, m_3

(without m_2), and we have to avoid this. The main takeaway is this: when node v decides to send a message based on having received some messages m_1, m_2, \dots, m_k , it has to be sure that these m_1, m_2, \dots, m_k correspond to some valid prefix of messages that it would have received in a synchronous environment.

5.3 A synchronizer, given layered sparse $O(T(\mathcal{A}))$ -cover

5.3.1 Algorithm

Let's first consider the case when we are given a layered sparse $(2^6 \cdot T(\mathcal{A}))$ -cover. We will try to run almost the same algorithm as in [Section 4.2](#) on the virtual nodes, with a few small changes. As a reminder, the physical node responsible for the actions of the virtual node (u, v, p) is simply a node u . We discuss the adaptations needed to make below.

Notion of distance. For virtual nodes $(u_1, p_1), (u_2, p_2)$, we define $dist((u_1, p_1), (u_2, p_2))$ as equal to $dist(u_1, u_2)$. Note that it follows that $dist(parent(v), v) \leq 1$ for any virtual node v .

Sending messages. In the BFS case, node v was sending “join” requests to all its neighbors. In the current setting, the virtual node (v, p) , sending some messages at pulse p , will send only the corresponding messages.

Simulation of virtual nodes by the physical nodes. Physical node v simulates all the actions of the virtual node (v, p) directly: as if it was actually sending those through the same edges. We will point out just one subtlety. If v has a choice, the message from which virtual node to send through a given edge, it chooses the order that would make [Corollary 2.3](#) and [Lemma 2.5](#) still hold. In particular, between messages corresponding to different pulses, it prioritizes smaller ones; for the same pulse, it still does it in a round-robin fashion.

Creation of the virtual nodes. When a physical node v receives/sends a message of pulse $p - 1$, it does the following. First, it waits until it receives $Go_Ahead(p)$. Then, it looks at all the messages with a pulse at most $p - 1$ that it has received/sent. It determines whether, in the synchronous world, it would send any messages at pulse p , given the history of the received and sent messages up to this point. Then, if it would send messages at pulse p , it creates a virtual node (v, p) . Finally, it chooses any virtual node of pulse $p - 1$ from which it has received a message as a parent. It then notifies all these virtual nodes of pulse $p - 1$ whether they have been chosen.

Convergecast and broadcast in the execution tree. Consider the process of collecting the information about the p -safety/emptiness of all (virtual) nodes with pulses in $[prev(prev(p)), p]$ for some pulse $p > 0$. In the BFS problem, it was easy, as every physical edge appeared only once in the execution tree. Now, a particular physical edge may appear in the execution tree many times, as there might be many messages sent through some particular edge. However, there is still at most one such message per pulse, so this does not lead to any problems, as we see later.

The final convergecast for node termination. In the d -thresholded BFS we had to make sure that every node at a distance larger than d from the sources learned it. Here, we do not require any termination of this form.

Everything else remains the same.

5.3.2 Analysis of correctness

[Lemma 4.7](#), [Lemma 4.8](#), [Lemma 4.9](#) translate into the current setting as they are.

Lemma 5.1 (Counterpart of [Lemma 4.10](#)). *For any physical node v , the following claims hold:*

- For any messages m_1, m_2 with pulses p_1, p_2 correspondingly with $p_1 < p_2$, v will receive m_1 before it receives m_2
- For any pulse $p > 0$, v will receive $Go_Ahead(p)$ only after it has received all the messages of pulse $p - 1$.

Proof. First claim. Let message m_2 be sent from (virtual) node v_2 of pulse p_2 , and message m_1 be sent from (virtual) node v_1 of pulse p_1 . Suppose that v receives message m_2 before m_1 . Consider the moment when v has already received message m_2 but hasn't received m_1 yet.

Let x be the integer from the segment $[p_1 + 1, p_2]$ divisible by the largest power of 2 (with largest $\ell(x)$), and let w be the ancestor of v_2 with pulse x . As the message from w in this execution tree has already been sent, w has already received $Go_Ahead(x)$, which is possible only after $h(h(w))$ receives $Go_Ahead(x)$.

Now note that $pulse(h(w)) \leq p_1$. Indeed, $pulse(h(w)) \leq x - 2^{\ell(x)}$, and as number $x - 2^{\ell(x)}$ is divisible by larger power of 2 than x is, it can't be in $[p + 1, p_1]$ by our choice of x . So, $pulse(h(w)) \leq x - 2^{\ell(x)} \leq p$.

Let y_1 be the ancestor of v_1 with pulse $prev(prev(x))$, y_2 be the ancestor of v_2 with pulse $prev(prev(x))$ (so that $y_2 = h(w)$). As $pulse(h(w)) \leq p$, y_1 is not $prev(x)$ -empty. y_2 is not x -empty by the choice of x .

Next, we show that $dist(y_1, y_2) \leq 2^{\ell(p)+5}$ with the argument completely mirroring the argument from [Lemma 4.10](#). Then, by [Lemma 4.9](#), $y_2 = h(h(w))$ can receive $Go_Ahead(x)$ only after y_1 has x -deregistered, which is not possible until m_1 has been sent, contradiction.

Second claim. Suppose the opposite. Suppose that v has already received $Go_Ahead(p)$ from some virtual node v_1 of pulse $p - 1$ before it receives a message m_2 of pulse $p - 1$ from some virtual node v_2 of pulse $p - 1$. Consider this moment. $h(h(v_1))$ must have already received $Go_Ahead(p)$.

As $dist(v_1, v_2) \leq 2$, by part (d) of [Lemma 4.7](#) we get $dist(h(h(v_1)), h(h(v_2))) \leq dist(h(h(v_1)), v_1) + dist(v_1, v_2) + dist(v_2, h(h(v_2))) \leq 2 + 2 \cdot (9 \cdot 2^{\ell(p)}) \leq 2^{\ell(p)+5}$, so we can apply [Lemma 4.9](#). As $h(h(v_1))$ has already received $Go_Ahead(p)$, $h(h(v_2))$ has already deregistered from all their common clusters, which happens only after m_2 has already been sent. □

Theorem 5.2. *This algorithm correctly synchronizes algorithm \mathcal{A} .*

Proof. Our goal is to show that the set of messages sent by all nodes is exactly the same as it would have been in the synchronous environment. Consider the smallest p such that one of the following happens: some node v that had to send a message m of pulse $p + 1$ has not sent it, or some node v sent a message m of pulse $p + 1$ that it would not send in the synchronous setting.

Suppose that the first case happened. In the synchronous setting, m is triggered by some message m_1 of pulse p . From [Lemma 5.1](#), by the time v receives $Go_Ahead(p + 1)$, it has received all the messages of pulse p , and, therefore, all the valid messages of pulses up to p . Then v would be able to detect that in the synchronous setting, it would have sent m , so it would send m now.

Suppose that the second case happened. Suppose that m has sent some message m of pulse $p + 1$ that it would not send in the synchronous setting. This action must have been triggered by receiving $Go_Ahead(p + 1)$. But then, once again, by that time, the set of messages with pulses up to p that v received would coincide with the set it would receive in the synchronous setting, so v would correctly detect that it does not send m . □

5.3.3 Analysis of the time and message complexity

The main result of this section is the following theorem:

Theorem 5.3. *For an algorithm \mathcal{A} with worst-case time complexity $T(\mathcal{A})$ and message complexity $M(\mathcal{A})$, this synchronized version, given a layered sparse $O(T(\mathcal{A}))$ -cover, finishes in $O(T(\mathcal{A}) \cdot \log^7 n \cdot (\log T(\mathcal{A}) + \log n))$ time and takes $O(M(\mathcal{A}) \cdot \log n^4 \cdot \log T(\mathcal{A}))$ messages.*

The proof follows the proof of [Theorem 4.15](#) closely. We divide all our pulse-related operations into stages, one stage \mathcal{S}_p for each pulse p . Then, we note that the isolated time complexity of \mathcal{S}_p is still $O(2^{\ell(p)} \log^7 n)$. Due to the [Lemma 4.13](#), the total isolated time complexity of all stages \mathcal{S}_i becomes $\sum_{p=1}^{T(\mathcal{A})} O(2^{\ell(p)} \cdot \log^7 n) = O(T(\mathcal{A}) \cdot \log^7 n \cdot \log T(\mathcal{A}))$. The initial convergecast for registering all the originators takes $O(T(\mathcal{A}) \cdot \log^8 n)$ time. So, the entire runtime is $O(T(\mathcal{A}) \cdot \log^7 n \cdot (\log T(\mathcal{A}) + \log n))$.

Now the message complexity. The number of messages sent for building the execution tree is $O(M(\mathcal{A}))$. The number of messages propagated in the execution tree is $O(M(\mathcal{A}) \log T(\mathcal{A}))$, as, by [Lemma 4.14](#), there are messages of only $O(\log T(\mathcal{A}))$ pulses propagated through each edge. Note the difference: in the BFS there were $O(n)$ edges in the execution tree, now $O(M(\mathcal{A}))$.

Now let's consider the messages sent through the clusters for registration/deregistration. Once again, for a given level ℓ , the total number of nodes who will p -register for some pulse p with $\ell(p) = \ell$ is $O(\frac{M(\mathcal{A})}{2^\ell})$. Such nodes will register in clusters of $2^{\ell+5}$ covers and therefore take $O(2^\ell \log^3 n)$ messages to the root and back. Every node is in $O(\log n)$ clusters, so there will be $O(2^\ell \log^3 n \cdot \log n \cdot \frac{M(\mathcal{A})}{2^\ell}) = O(M(\mathcal{A}) \log^4 n)$ messages corresponding to registration at a given level, and the total number for all levels is $O(M(\mathcal{A}) \cdot \log n^4 \cdot \log T(\mathcal{A}))$.

5.4 A synchronizer, without being given layered sparse $O(T(\mathcal{A}))$ -cover

In the [Section 5.3](#) we have constructed a synchronizer that works assuming that the layered sparse $O(T(\mathcal{A}))$ -cover is already provided. In this section, we discuss the ways to synchronize from scratch.

With the algorithm from [Section 4.6](#), we know how to build layered sparse $T(\mathcal{A})$ -covers from scratch. However, it is possible that the exact value of $T(\mathcal{A})$, or even its multiplicative approximation, is not known to us. We provide slightly different synchronization approaches in cases when this value is known to us versus when it is not known.

Theorem 5.4. *Consider an algorithm \mathcal{A} with worst-case time complexity $T(\mathcal{A})$ and message complexity $M(\mathcal{A})$. Suppose that the nodes **do not know** the bound on $T(\mathcal{A}(M))$. There is a synchronizer for \mathcal{A} such that*

- *The entire algorithm terminates in $O(D \log^{11} n + T(\mathcal{A}) \cdot \log^7 n \cdot (\log T(\mathcal{A}) + \log n))$ time*
- *All the outputs of \mathcal{A} are produced by the time $O(T(\mathcal{A}) \cdot \log^{11} n)$. Thus, with the notion of time complexity as the time until all nodes generate their output, the synchronizer has polylogarithmic overhead.*⁵
- *It takes $O(m \log^{10} n + M(\mathcal{A}) \cdot \log^5 n)$ messages.*

Proof. The algorithm will be very similar to the algorithm from the [Theorem 4.23](#). All nodes will start by constructing the initial layered sparse $2^{O(1)}$ -cover, then proceed to the iterations. One iteration will consist of simulating the next 2^t pulses of \mathcal{A} and constructing the next sparse $2^{t+O(1)}$ -cover.

⁵See [Appendix B](#) for discussion on the time complexity definition.

The question, once again, is how the nodes learn that it's time to stop constructing the sparse covers. We will do this with the **Approach 1** from the proof of [Theorem 4.23](#). By the $\lceil \log_2 D \rceil$ -th iteration, some constructed cluster would contain all the nodes. It would then notify all the nodes that there is a cluster that contains all the nodes, and they will not construct the next sparse cover: they will use the sparse cover containing this cluster that includes all the nodes instead.

The isolated time complexity of the initialization remains $O(\log^{10} n)$ and the isolated time complexity of constructing 2^t -sparse cover remains $O(2^t \cdot \log^{11} n)$. Furthermore, after the first $\lceil \log_2 D \rceil$ iterations, no sparse covers will be constructed. So, the total isolated runtime spent on the sparse cover construction is $O(D \log^{11} n)$. According to the [Theorem 5.3](#), the actual synchronization of \mathcal{A} in isolation would take only $O(T(\mathcal{A}) \cdot \log^7 n \cdot (\log T(\mathcal{A}) + \log n))$ time, so the total runtime is $O(D \log^{11} n + T(\mathcal{A}) \cdot \log^7 n \cdot (\log T(\mathcal{A}) + \log n))$.

All the messages of \mathcal{A} will be delivered after the first $\lceil \log_2 T(\mathcal{A}) \rceil$ iterations are over; this will take $O(T(\mathcal{A}) \cdot \log^{11} n)$.

Moving on to the message complexity, sparse cover construction takes only $O(m \log^{10} n)$ messages. The registration/deregistration process for the level ℓ take $O(\frac{M(\mathcal{A})}{2^\ell} \cdot \min(D, 2^\ell) \log^3 n \cdot \log n)$ time. If $T(\mathcal{A}) < D$, this sums up to $O(M(\mathcal{A}) \cdot \log^4 n \cdot \log T(\mathcal{A}))$, otherwise, this sum telescopes to $O(M(\mathcal{A}) \log^4 n \log D)$. In both cases, we will bound it by $O(M(\mathcal{A}) \log^5 n)$. So, the message complexity is $O(m \log^{10} n + M(\mathcal{A}) \log^5 n)$. □

Theorem 5.5. *Consider an algorithm \mathcal{A} with worst-case time complexity $T(\mathcal{A})$ and message complexity $M(\mathcal{A})$. Suppose that all the nodes **know** the bound on $T(\mathcal{A}(M))$. There is a synchronizer for \mathcal{A} such that*

- *All the outputs are produced and the entire algorithm terminates in $O(\min(D, T(\mathcal{A})) \log^{11} n + T(\mathcal{A}) \cdot \log^7 n \cdot (\log T(\mathcal{A}) + \log n))$ time*
- *It takes $O(m \log^{10} n + M(\mathcal{A}) \log^5 n)$ messages.*

Proof. The algorithm remains the same as in [Theorem 5.4](#) with a single modification: all nodes will simply run only the first $\lceil \log_2 T(\mathcal{A}) \rceil$ iterations of constructing the sparse cover (but they may still skip going to the next iterations if they detect some cluster that contains all the nodes). □

6 Applications

We already described in [Section 4](#) how to transform the synchronous BFS algorithm to the asynchronous environment so that it runs in time complexity $\tilde{O}(D)$ and message complexity $\tilde{O}(m)$. This already proves [Corollary 1.2](#). We remark that $\tilde{O}(D)$ is also the time until all nodes terminate.

We next provide proofs for our other two applications, deterministic asynchronous leader election and minimum spanning tree with near-optimal time and message complexities.

Proof of [Corollary 1.3](#). We first describe a relatively simple deterministic leader election algorithm that runs in $\tilde{O}(D)$ rounds and uses $\tilde{O}(m)$ messages, with the following guarantees: in the end of the algorithm, there is a unique leader elected, each node knows the identifier of the elected leader, and all nodes terminate.

Initially, all nodes are candidates. The algorithm consists of epochs. Epoch i runs for $\tilde{O}(2^i)$ rounds and using $\tilde{O}(m)$ messages. During epoch i , we first build a sparse 2^i cover on all nodes, using the synchronous algorithm of [\[RG20\]](#). This algorithm runs in $\tilde{O}(2^i)$ rounds and generates clusters of depth $\tilde{O}(2^i)$, with the following properties: (a) each edge is included in $\text{poly}(\log n)$ clusters, with a tree of depth $\tilde{O}(2^i)$, and (b) for each node v , its entire 2^i -hop neighborhood —the set of all nodes

within distance 2^i of v —is included in one cluster. Then, in each cluster, using the cluster tree, we perform a convergecast of the minimum candidate identifier, and we then broadcast the value to all nodes of the cluster. This convergecast and broadcast takes $\tilde{O}(2^i)$ rounds. Any candidate node v whose identifier is not the minimum in one of the clusters that includes v ceases to be a candidate. We then would like to move to the next epoch. But there is a termination detection before that, as we describe next.

We would like to detect termination. Intuitively, we want to terminate as soon as the cluster of a candidate contains all nodes of the graph. To detect this, we do as follows. Each node v sends to its neighbors u all the cluster ids that include v . This is doable in $\text{poly}(\log n)$ time and with $\tilde{O}(m)$ messages. Then, we perform a convergecast and broadcast along the cluster tree whether any node v in the cluster C has a neighbor u that is not in cluster C . If there is no such node, then the cluster contains all nodes, and the sole remaining candidate — which is actually the node that had the globally minimum id — is the leader and each node of the cluster knows the id of the leader.

Notice that in the course of the algorithm, the node with minimum id always remains a candidate. As soon as we reach epoch $i = \lceil \log_2 D \rceil$, there will be one cluster that contains all the nodes in it, and that cluster will detect termination, hence the entire algorithm will terminate.

The time complexity of the algorithm is $\sum_{i=1}^{\lceil \log_2 D \rceil} \tilde{O}(2^i) = \tilde{O}(D)$. The message complexity in each epoch is $\tilde{O}(m)$ and since there are $\lceil \log_2 D \rceil$ epochs until global termination, the overall message complexity is also $\tilde{O}(m)$.

Finally, this deterministic synchronous leader election algorithm with time complexity $\tilde{O}(D)$ and message complexity $\tilde{O}(m)$ can be directly fed into our synchronizer, hence generating a deterministic asynchronous leader election algorithm with time complexity (and indeed time to termination) $\tilde{O}(D)$ and message complexity $\tilde{O}(m)$. \square

Proof of Corollary 1.4. The work of Elkin [Elk20] describes a deterministic synchronous MST algorithm that runs in $\tilde{O}(D + \sqrt{n})$ rounds and uses $\tilde{O}(m)$ messages. By invoking our synchronizer on this, we get an asynchronous deterministic algorithm that runs and terminates in $\tilde{O}(D + \sqrt{n})$ time and uses $\tilde{O}(m)$ messages. \square

References

- [AP90a] Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 514–522, 1990.
- [AP90b] Baruch Awerbuch and David Peleg. Sparse Partitions. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 503–513, 1990.
- [APSPS92] Baruch Awerbuch, Boaz Patt-Shamir, David Peleg, and Michael Saks. Adapting to asynchronous dynamic networks. In *ACM symposium on Theory of Computing (STOC)*, pages 557–570, 1992.
- [Awe85] Baruch Awerbuch. Complexity of Network Synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.
- [DKMJ⁺22] Fabien Dufoulon, Shay Kutten, William K Moses Jr, Gopal Pandurangan, and David Peleg. An Almost Singularly Optimal Asynchronous Distributed MST Algorithm. In *International Symposium on Distributed Computing (DISC)*, 2022.
- [Elk20] Michael Elkin. A simple deterministic distributed mst algorithm with near-optimal time and message complexities. *Journal of the ACM (JACM)*, 67(2):1–15, 2020.
- [GGR21] Mohsen Ghaffari, Christoph Grunau, and Vavlav Rozhon. Improved Deterministic Network Decomposition. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2904–2923, 2021.
- [GHK18] Mohsen Ghaffari, David G Harris, and Fabian Kuhn. On derandomizing local distributed algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 662–673, 2018.
- [GKS17] Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed MST and routing in almost mixing time. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, 2017.
- [HHW18] Bernhard Haeupler, D Ellis Hershkowitz, and David Wajc. Round-and message-optimal distributed graph algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 119–128, 2018.
- [KMJPP21] Shay Kutten, William K Moses Jr, Gopal Pandurangan, and David Peleg. Singularly Near Optimal Leader Election in Asynchronous Networks. In *International Symposium on Distributed Computing (DISC)*, 2021.
- [Lin92] N Linial. Locality in Distributed Graph Algorithms. *SIAM Journal on Computing (SICOMP)*, 21(1):193–201, 1992.
- [LS93] Nati Linial and Michael Saks. Low Diameter Graph Decompositions. *Combinatorica*, 13(4):441–454, 1993.
- [Mic] Michael Elkin. Synchronizers, spanners. <https://www.cs.bgu.ac.il/~elkinm/awer.pdf>. Accessed: January 8, 2023.

- [MK19] Ali Mashreghi and Valerie King. Brief announcement: Faster asynchronous MST and low diameter tree construction with sublinear communication. In *International Symposium on Distributed Computing (DISC)*, 2019.
- [Pel00] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.
- [PRS17] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A time- and message-optimal distributed algorithm for minimum spanning trees. In *ACM-SIGACT Symposium on Theory of Computing (STOC)*, pages 743–756, 2017.
- [RG20] Vaclav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *ACM Symposium on Theory of Computing (STOC)*, pages 350–363, 2020.

A A review of Awerbuch’s α , β , and γ synchronizers

As mentioned before, the main challenge in synchronization is the following: consider the time that node v has generated pulse p and should next generate pulse p after it has received every message of pulse p sent to it by each neighbor u . The difficulty is that node v does not know which neighbors sent messages to it and which did not, and simply waiting for a certain amount of time cannot resolve the issue; some messages might take an unpredictably long time. Generally, we can add an *acknowledgment* to the messages: whenever a node w sends a message to a neighbor w' , node w' should send an acknowledgment of that message back to w . This increases message complexity by only a 2 factor, and waiting for the acknowledgment can increase the time by only a 2 factor. Let us say node v is *safe* for pulse p when it has received an acknowledgment for each of the messages it sent after its pulse p (and before the next pulses). For a node v to generate its pulse $p + 1$, node v needs to ensure that v has received any message of pulse p sent by any neighbor u . One way of achieving the latter is to gather the information that ensures that all neighbors of v are safe for pulse p .

Awerbuch [Awe85] presented three synchronizer algorithms that follow this outline; he called them α , β , and γ . These exhibit different tradeoffs between time and message complexity overhead.

Synchronizer α is the trivial scheme where each node u , once u is safe for pulse p , sends a message to each neighbor v informing them. Then, node v can generate its pulse $p + 1$ once it has received this pulse- p safety information from each of its neighbors. Then, node v can send the messages of round $p + 1$ of the synchronous algorithm. This α synchronizer has an ideal $O(1)$ time complexity overhead. However, for each pulse, the scheme communicates safety information over all edges of the graph. Hence it increases the message complexity of algorithm \mathcal{A} to $O(M(\mathcal{A}) + T(\mathcal{A}) \cdot m)$. This is a terrible message complexity, as it is asymptotically equal to the highest message complexity possible for the given time complexity.

Synchronizer β first uses an initialization phase to build a low-depth tree, e.g., to elect a root and compute a BFS tree rooted at the leader. Then, for each pulse, β performs a convergecast and broadcast along the tree so that each node v learns that all other nodes of the graph are safe for pulse p . Only after that does the node v generate pulse $p + 1$. Hence, this scheme has time complexity overhead $O(D)$, where D denotes the network diameter, and it increases the message complexity to $O(M(\mathcal{A}) + T(\mathcal{A}) \cdot n)$. There is also a high time and message complexity for the initialization—computing a leader and building a BFS rooted in it—but we will ignore that here.

Synchronizer γ comes close to achieving the best of α and β . Picking one particularly useful parameterization of the γ scheme, it achieves a small time complexity overhead of $O(\log n)$ while

increasing the message complexity to only $O(M(\mathcal{A}) + T(\mathcal{A}) \cdot n)$. The rough idea is to compute a low-diameter clustering of the vertices, where each cluster has $O(\log n)$ -diameter, and keeping only $O(n)$ edges between neighboring clusters such that every two clusters that had adjacent nodes in the graph have at least one edge connecting them. Then, roughly speaking, one applies the β scheme inside the clusters and uses the α scheme afterward in between the clusters. This way, each node v will learn when all its neighbors are safe for a pulse p within $O(\log n)$ extra time, and the safety messages of each pulse travel through only $O(n)$ edges, hence implying that the message complexity is increased to $O(M(\mathcal{A}) + T(\mathcal{A}) \cdot n)$.

Awerbuch provides only a sequential algorithm for computing such a clustering. However, a follow-up work of Linial and Saks [LS93] presented a $O(\log^2 n)$ -time and $O(m \log^2 n)$ message synchronous algorithm for this clustering. Given the low time complexity, this algorithm itself can be made asynchronous using the α scheme. The combination provides a randomized variant of the γ synchronizer that has initialization time complexity $O(\log^2 n)$ and initialization message complexity $O(m \log^2 n)$.

We comment that this can be made deterministic using recent progress on deterministic synchronous network decomposition, with only $\text{poly}(\log n)$ increase in the parameters. In particular, by plugging in a deterministic network decomposition of Rozhon and Ghaffari [RG20] instead of the algorithm of Linial and Saks [LS93], one can obtain a deterministic variant of the γ synchronizer, which would turn any synchronous algorithm A with time complexity $T(A)$ and message complexities $M(A)$ into an asynchronous algorithm A' with time complexity $T(A) \cdot \text{poly}(\log n)$ and message complexity $O(M(A) + (m + T(A) \cdot n) \cdot \text{polylog} n)$. This is including the initialization complexities. Improved variants of the decomposition improve the logarithmic factors [GGR21].

Optimality of Awerbuch’s bounds for generating all pulses at all nodes. Awerbuch also showed that his bounds are nearly the best possible for the approach of having each node generate each pulse $1, 2, 3, \dots$, one per round. In particular, he showed that even for a 2 round synchronous algorithm, any asynchronous variant that runs in at most $O(k)$ time must have message complexity at least $\Omega(n^{1+1/k})$.

These synchronizers do not have a small time and message complexity overhead. Even ignoring the initialization complexities, the synchronizers mentioned above do not necessarily provide good time complexity and message complexity overheads. In particular, consider an algorithm A such that $M(A) \ll T(A) \cdot n$. For such an algorithm, the message complexity overhead $(T(A) \cdot n)/M(A)$ can be quite high.⁶ The problem is rooted in trying to generate pulses of all rounds at each node. The synchronous algorithm might have each node send only in few rounds, and it would suffice to generate only those pulses at this node.

B Model subtleties

We presented the synchronous model of distributed message passing in [Section 1.1](#). However, there are some subtleties not clarified in this definition; we discuss those here.

Can algorithms reference the time? One subtlety is whether the algorithm can explicitly refer to the round numbers or not. This can lead to a critical issue for synchronizers. We discuss two natural interpretations of the model. We are not aware of any prior work that draws this

⁶We note that many algorithms for important graph problems have this property. Consider, for example, a simple BFS problem, where the synchronous algorithm has time complexity $O(D)$ and message complexity $O(m)$; note that m can be much smaller than Dn . See also the message and time-efficient synchronous MST algorithms of [PRS17, Elk20, HHW18].

distinction. The first interpretation provides a stronger model for algorithms. However, the (randomized) synchronizers of [APSPS92, AP90a] do not work for the first interpretation; they would not provide $\text{poly}(\log n)$ time and message complexity overheads. But we argue that they work for the second interpretation, and our deterministic synchronizer is also presented for the second interpretation. Fortunately, many graph algorithms of interest adhere to the second interpretation or can be paraphrased to adhere to it without more than a $\text{poly}(\log n)$ increase in time and message complexities.

Clock-based synchronous algorithms. In the first interpretation of the model, the algorithm can have explicit reference to the round numbers. That is, each node can have explicit access to the clock that counts the rounds, and it can base its action on that clock. For instance, it can do something like: “in round number \dots , send message \dots ”? Per se, this is not forbidden in the model. Indeed many of the standard synchronous algorithms are written in such a way. For instance, the algorithm is made of phases, each phase consisting of a prespecified number of rounds. Each node starts the second phase only after the rounds allotted for the first phase have passed. We call such synchronous algorithms *clock-based*.

Event-driven synchronous algorithms. The second interpretation, which is also natural but more restrictive, is that the algorithm cannot explicitly refer to round numbers, i.e., the nodes cannot explicitly access the clock value. The only guarantee is that all messages sent in the network experience the same delay. For instance, this implies that, if there are two paths connecting v to u , and node v forwards a message along these two paths, the message will arrive earlier along the shorter path. Algorithms without explicit reference to the clock should base their actions only on new inputs that they receive: “upon receiving message(s) \dots , perform some computation and send message(s) \dots ” Note that the response may involve sending several messages, even several to the same neighbor. Hence, the response might take more than a single time unit. However, an algorithm in this interpretation cannot do something like “send one message, wait for x rounds, and then send another message.” Fortunately, many graph algorithms of interest can be paraphrased to adhere to this event-driven interpretation, without more than a $\text{poly}(\log n)$ increase in time and message complexities.⁷

As mentioned before, the synchronizers of [APSPS92, AP90a] do not work for the clock-based interpretation. They are primarily written for BFS, and then an extension is claimed for general synchronous algorithms. The issue is in the extensions. Consider synchronous algorithms where each node might send several messages (e.g., in response to receiving one message). There are brief sections of explanations for the extension in [AP90a, Section 6] and [AP90a, Section 6]. These suggest that we transform the case where one node sends multiple messages to a chain of “virtual messages” inside the same node, one leading to the next. This is necessary as a part of the “execution tree” through which all communications get synchronized. In the clock-driven interpretation, if the node sends only two messages but $\Theta(n)$ rounds apart, this chain will include $\Theta(n)$ virtual messages for those two real messages. This can increase the algorithm’s message complexity by up to an $\Theta(n)$ factor. Unfortunately, the synchronizer’s message complexity depends linearly on the total number of messages, virtual or real, and thus it can have an overhead of $\Omega(n)$ for the message complexity of clock-based synchronous algorithms. Fortunately, the issue seems fixable for the event-driven interpretation of synchronous algorithms, and this is also the interpretation for

⁷We note that there is also a general transformation. One can transform any clock-based synchronous algorithm to an event-driven one with an $O(1)$ factor time overhead: simply have each node generate a clock for itself by sending a message to one of its neighbors and getting a response. However, this transformation has an additive increase of nT in message complexity, where T denotes the algorithm’s round complexity, and this message complexity increase is often too high.

which we describe our synchronizer. We provide the correctness proof only for our deterministic synchronizer, which subsumes their randomized synchronizer (up to $\log n$ factors).

A subtlety in message delays, in both synchronous and asynchronous models. Another subtlety is with regard to the message delays. In writing event-driven synchronous algorithms, we need to ensure that no node injects more than one message in each edge per time unit. We can ensure that this way: each node v that sends a message to a neighbor u waits for an acknowledgment from u , before v can send a new message to u . These acknowledgments increase time and message complexities by at most a 2 factor.

In a sense, this subtlety also appears in the definition of the asynchronous model. In defining the asynchronous model, as described in [Section 1.1](#), we said that each message injected into the network link takes one (normalized) time unit to arrive at the destination. The value of the time unit is not known to the nodes. But what if the node simultaneously injects k many messages into the link? This has not been explicitly forbidden in the model description. The right interpretation, which takes congestion into account and is closest to the synchronous model, would be as follows: in this batch of simultaneously injected messages, the i^{th} message arrives after i time units. A cleaner resolution is to use the same acknowledgment scheme mentioned above: each node has to wait for an acknowledgment of each message it injected into the link, before injecting the next message. Notice that this way, at each moment in time, each node v can have at most two active messages in its link toward neighbor u : one message of the algorithm sent by v to u and at most one acknowledgment for a message initially sent from u to v . Node v will not have two acknowledgments active in the link because it does not receive the next message from u before u has received the acknowledgment of the former message it sent to v . Thus, again, this acknowledgment mechanism has only a 2 factor cost in time and message complexities. And it conveniently removes the issue of having to model delays experienced by messages injected into one network link simultaneously or in close succession.

Time complexity: time to output or time to termination. Finally, how do we measure time complexity precisely? As mentioned in [Section 1.1](#), throughout the paper, we define time complexity as the time from the start of the algorithm to the time that the last node generates its output. We use this for both synchronous algorithms and asynchronous algorithms, and this is indeed the definition for which the $\text{poly}(\log n)$ time complexity overhead claim of [\[APSPS92\]](#) holds (after other fixes). Unfortunately, this point was not made clear in [\[APSPS92\]](#). A stronger time complexity measure would be to bound the time until all nodes terminate and know that they have terminated. For this stronger definition, both the synchronizers of [\[APSPS92\]](#) and ours will have a larger time complexity overhead. It remains an interesting open problem to develop synchronizers with $\text{poly}(\log n)$ time and message complexity overheads for this stronger definition of time complexity.

Fortunately, for most applications for concrete graph problems, there is a workaround: (1) The asynchronous algorithm generated by the synchronizers will terminate within $\tilde{O}(D)$ time. Hence, if the synchronous algorithm that is being made asynchronous has time complexity $\Omega(D)$, then the time complexity overhead even for the stronger definition is $\text{poly}(\log n)$. Notice that this is the case for a wide range of applications, especially those for global graph problems, including the breadth first search, leader election, and minimum spanning tree computation. And for synchronous algorithms that run faster—e.g., $\text{poly}(\log n)$ round synchronous algorithms—the simple α synchronizer described in [Appendix A](#) suffices. (2) If the synchronizer algorithm is given a parameter τ (i.e., if the value is provided to all nodes), such that the synchronous algorithm is promised to terminate within $O(\tau)$ rounds, then the asynchronous algorithm will not only generate all the outputs within $\tilde{O}(\tau)$ time but also terminate in all nodes by time $\tilde{O}(\tau)$.

C Algorithm for **Theorem 4.20**

Theorem 4.20. [Rozhon and Ghaffari [RG20]] *There is an algorithm that, given a value k known to all nodes, in $O(k \log^{10} n)$ communication rounds outputs a k -separated weak-diameter network decomposition of G with $O(\log n)$ color classes, each one with $O(k \cdot \log^3 n)$ weak-diameter in G .*

Moreover, for each color and each cluster \mathcal{C} of vertices with this color, we have a Steiner tree $T_{\mathcal{C}}$ with radius $O(k \cdot \log^3 n)$ in G , for which the set of terminal nodes is equal to \mathcal{C} . Furthermore, each edge in G is in $O(\log^4 n)$ of these Steiner trees.

For the practical needs of this paper, we will briefly recall their algorithm. In the next subsection, we will show that we can run this algorithm in an asynchronous environment with the tools we developed in the previous sections. For proof of the correctness of this synchronous algorithm, we refer to the original paper [RG20].

In the following lemma, we describe the process for constructing the clusters of one color of the network decomposition (e.g., the first color), in a way that it clusters at least half of the vertices. Since after each application of this lemma only half of the vertices remain, by $\log n$ repetitions, we get a decomposition of all vertices, with $\log n$ colors.

Lemma C.1. *Let $S \subseteq V$ denote the set of living vertices. There is a deterministic distributed algorithm that, in $O(k \log^9 n)$ communication rounds, finds a subset $S' \subseteq S$ of living vertices, where $|S'| \geq |S|/2$, such that the subgraph $G[S']$ induced by set S' is partitioned into disjoint clusters, so that every two of these clusters are at a distance larger than k , and each cluster has weak-diameter $O(k \cdot \log^3 n)$ in graph G .*

Moreover, for each such cluster \mathcal{C} , we have a Steiner tree $T_{\mathcal{C}}$ with radius $O(k \cdot \log^3 n)$ in G where all nodes of \mathcal{C} are exactly the terminal nodes of $T_{\mathcal{C}}$. Furthermore, each edge in G is in $O(\log^3 n)$ of these Steiner trees.

We obtain **Theorem 4.20** by $c = \log n$ iterations of applying **Lemma C.1**, starting from $S = V$. For each iteration $j \in [1, \log n]$, the set S' are exactly nodes of color j in the network decomposition, and we then continue to the next iteration by setting $S \leftarrow S \setminus S'$.

In the proof of the lemma, the following observation is useful. Once again, its proof is omitted.

Algorithm for Lemma C.1 The construction has $b = O(\log n)$ phases, corresponding to the number of bits in the identifiers. Initially, we think of all nodes of S as **living**. During this construction, some living nodes **die**. We use S'_i to denote the set of living vertices at the beginning of phase $i \in [0, b - 1]$. Slightly abusing the notation, we let S'_b denote the set of living vertices at the end of phase $b - 1$ and define S' to be the final set of living nodes, i.e., $S' := S'_b$.

Moreover, we label each living node v with a b -bit string $\ell(v)$, and we use these labels to define the clusters. At the beginning of the first phase, $\ell(v)$ is simply the unique identifier of node v . This label can change over time. For each b -bit label $L \in \{0, 1\}^b$, we define the corresponding cluster $S'_i(L) \subseteq S'_i$ in phase i to be the set of all living vertices $v \in S'_i$ such that $\ell(v) = L$. We will maintain one Steiner tree T_L for each cluster $S'_i(L)$ where all nodes $S'_i(L)$ are the terminal nodes of T_L . Initially, each cluster consists of only one vertex and this is also the only (terminal) node of its respective Steiner tree.

Construction invariants. The construction is such that, for each phase $i \in [0, b - 1]$, we maintain the following invariants:

- (I') For each i -bit string Y , the set $S'_i(Y) \subseteq S'_i$ of all living nodes whose label ends in suffix Y has no other living nodes $S'_i \setminus S'_i(Y)$ in its k -hop neighbourhood.

In other words, the set $S'_i(Y)$ is a union of some connected components of the subgraph $G[S'_i]$ induced by living nodes S'_i and in the k -hop neighbourhood in G around $S'_i(Y)$ all nodes are either dead or they do not belong to the set S (they were colored by previous application of the algorithm).

- (II') For each label L and the corresponding cluster $S'_i(L)$, the related Steiner tree T_L has radius at most $i \cdot k \cdot R$, where $R = O(\log^2 n)$.
- (III) We have $|S'_{i+1}| \geq |S'_i|(1 - 1/2b)$.

These invariants, together with the observation that each edge is used in $O(\log n)$ Steiner trees, prove [Lemma C.1](#).

In particular, from the first invariant, we conclude that at the end of b phases, different clusters are at a distance at least $k + 1$ from each other. From the second invariant, we conclude that each cluster has a Steiner tree with radius $bR = O(k \log^3 n)$. Finally, from the third invariant, we conclude that for the final set of living nodes $S' = S'_b$, we have $|S'| \geq (1 - 1/2b)^b |S| \geq |S|/2$.

Outline of one phase of construction. We now outline the construction of one phase and describe its goal. Let us think about some fixed phase i . We focus on one specific i -bit suffix Y and the respective set $S'_i(Y)$. Let us categorize the nodes in $S'_i(Y)$ into two groups of **blue** and **red**, based on whether the $(i + 1)^{th}$ least significant bit of their label is 0 or 1. Hence, all blue nodes have labels of the form $(* \dots * 0Y)$, and all red nodes have labels of the form $(* \dots * 1Y)$, where $*$ can be an arbitrary bit. During this phase, we make some small number of the red vertices die, and we change the labels of some of the other red vertices to blue labels (and then the node is also colored blue). All blue nodes remain living and keep their label. The eventual goal is that, at the end of the phase, among the living nodes, there is no blue node b and red node r with $dist(b, r) \leq k$. This leads to invariant (I) for the next phase. The construction ensures that we kill at most $|S'_i(Y)|/2b$ red vertices of set $S'_i(Y)$, during this phase. We next describe this construction.

Steps of one phase. Each phase consists of $R = 10b \log n = O(\log^2 n)$ steps, each of which will be implemented in $O(k \cdot \log^6 n)$ rounds. Hence, the overall round complexity of one phase is $O(k \log^8 n)$ and over all the $O(\log n)$ phases, the round complexity of the whole construction of [Lemma C.1](#) is $O(k \cdot \log^9 n)$ as advertised in its statement. Each step of the phase works as follows: all blue nodes start a BFS from them up to distance k ; a node that is reached in the BFS is added to the cluster of the node from which it received the first “join” proposal. Next, all red nodes that were added to the cluster will try to join it, to adopt its label.

For each blue cluster A , we have two possibilities:

- (1) If the number of adjacent red nodes that requested to join A is less than or equal to $|A|/2b$, then A does not accept any of them and all these requesting red nodes die (because of their request being denied by A). In that case, cluster A **stops** for this whole phase and does not participate in any of the remaining steps of this phase.
- (2) Otherwise — i.e., if the number of adjacent red nodes that requested to join A is strictly greater than $|A|/2b$ — then A accepts all these requests and each of these red nodes change their label to the blue label that is common among all nodes of A . In this case, we also grow the Steiner tree of cluster A by one hop to include all these newly joined nodes.

After the breadth first search algorithm finishes, roots of all Steiner trees collect the number of proposing red nodes and each root decides to either accept all proposing red vertices and recolor

them to blue, or it makes them die and stops growing. The Steiner trees, however, stay the same even if some of its vertices die, the red nodes that died are just labeled as nonterminals. This finishes the description of one step of current phase.

As each edge is in $O(\log^3 n)$ cluster trees, and diameters of the clusters are $O(k \log^3 n)$, one step can be implemented in $O(k \log^6 n)$ iterations. As there are $O(\log^2 n)$ steps per phase and $O(\log n)$ phases per color, we get the resulting runtime of $O(k \log^9 n)$.