Near-Optimal Scheduling of Distributed Algorithms

Mohsen Ghaffari
MIT
ghaffari@mit.edu

Abstract

This paper studies the question of how to run many distributed algorithms, solving independent problems, together as fast as possible.

Suppose that we want to run distributed algorithms $\mathcal{A}_1, \mathcal{A}_2 \ldots, \mathcal{A}_k$ in the $\mathsf{CONGEST}$ model, each taking at most $\mathsf{dilation}$ rounds, and where for each network edge, at most $\mathsf{congestion}$ messages need to go through it, in total over all these algorithms. A celebrated work of Leighton, Maggs, and Rao [22] shows that in the special case where each of these algorithms is simply a *packet routing*—that is, sending a message from a source to a destination along a given path—there is an $O(\mathsf{congestion} + \mathsf{dilation})$ round schedule. Note that this bound is trivially optimal.

Generalizing the framework of LMR [22], we study scheduling general distributed algorithms and present two results: (a) an existential schedule-length lower bound of $\Omega(\mathsf{congestion} + \mathsf{dilation} \cdot \frac{\log n}{\log \log n})$ rounds, (b) a distributed algorithm that produces a near-optimal $O(\mathsf{congestion} + \mathsf{dilation} \cdot \log n)$ round schedule, after $O(\mathsf{dilation} \cdot \log^2 n)$ rounds of pre-computation.

A key challenge in the latter result is to solve the problem with only private randomness, as globally-shared randomness simplifies it significantly. The technique we use for this problem is in fact more general, and it can be used to remove the assumption of having shared randomness from a broad range of distributed algorithms, at the cost of a slow down factor of $O(\log^2 n)$.

1 Introduction and Related Work

Computer networks are constantly running many applications at the same time and because of the bandwidth limitations, each application gets slowed down due to the activities of the others. Despite that, for the vast majority of the distributed algorithms introduced in the area of theoretical distributed computing, the initial design and analysis have been carried out with the assumption that each algorithm uses the network alone. In this paper, we investigate the issue of what happens when many distributed algorithms are to be run together.

Specifically, we study the questions of *how to run these distributed algorithms simultaneously as fast as possible* and *what are the limitations on how fast this can be done*. While being arguably a fundamental issue, to the best of our knowledge, these questions have not been investigated in their full generality. Although, we describe a number of special cases that were studied in the past.

Throughout the paper, we work with the CONGEST model [30], which is a standard distributed model that takes bandwidth limitations into account. The communication network is represented by an undirected graph $G = (V, E)$ where $|V| = n$. Communications occur in synchronous rounds and in each round each node can send one $O(\log n)$-bit message to each of its neighbors.

The general scenario we consider is as follows: We want to run independent distributed algorithms $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ together, but we do not know what problem is being solved by each algorithm. Hence, we must run each algorithm essentially in a black-box manner without altering the content of its messages, except for potentially adding a small amount of information to its header. We now mention some special cases of this problem that have been studied in the past:

(I) Broadcasting k messages from different sources, each to the h-hop neighborhood of its source. Note that this can be viewed as running k single-message broadcast algorithms together. Classical analysis [36] shows that the natural algorithm in which at each round each node sends one message that it has not sent before (and has traveled less than h hops so far) solves the problem in $O(k + h)$ rounds. The significant aspect is the additive appearance of k which in a sense implies a perfect pipelining between the k broadcast algorithms.

(II) Running breadth-first search algorithms from different sources. Holzer and Wattenhofer [18] show that one can run n BFSs starting in different nodes all together in $O(n)$ rounds. This is done by delaying BFSs carefully in a way that they do not interfere (once started). More generally, Lenzen and Peleg [24] show that k many h-hop BFSs from different sources can be performed in $O(k + h)$ rounds. This is in a sense a strengthening of the broadcast result of [36], as it shows that in fact each BFS-token (or equivalently, broadcast message) will be delivered to each related destination along a shortest path.

(III) Routing many packets, each from a source to a destination, along a given path. This problem has received the most attention [7, 10, 22, 23, 28, 29, 31, 33, 34, 37] among these special cases. Viewing our distributed algorithm scheduling problem as a generalization of this packet routing problem, we adopt a terminology close to that introduced for packet routing in [22]. We discuss some known results for packet routing after introducing this (generalized) terminology.

Generally, if one of the algorithms $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ that are to be run together takes d rounds, running all of them together will clearly require at least d rounds. We refer to the maximum running time of the algorithms as $\mathsf{dilation}$. Furthermore, there is another simple lower bound due to the bandwidth limitations. Consider a particular edge e of the graph. Let $c_i(e)$ be the number of rounds in which algorithm $\mathcal{A}_i$ sends a message over e. Then, running all the algorithms together will also require at least $\mathsf{congestion} = \max_{e \in E} \mathsf{congestion}(e)$ rounds where $\mathsf{congestion}(e) = \sum_{i=1}^{k} c_i(e)$. Hence, we can conclude that running all algorithms together will need at least $\max\{\mathsf{congestion}, \mathsf{dilation}\} \geq (\mathsf{congestion} + \mathsf{dilation})/2$ rounds. The key question of interest is:

Question: Can we always find a schedule close to the $\Omega(\mathsf{congestion} + \mathsf{dilation})$ lower bound?

A simple yet powerful technique that proves helpful in this regard is *random delays* [22], which was first introduced in the context of packet routing — the problem we described in item (III) above. In packet routing, $\mathsf{dilation}$ is the length of longest path and $\mathsf{congestion}$ is the maximum number of paths that go through an edge. The random delays method achieves an upper bound within an $O(\log n)$ factor of the lower bound; more precisely, a schedule of size $O(\mathsf{congestion} + \mathsf{dilation} \cdot \log n)$. Improving this simple $O(\log n)$-approximation to an $O(1)$-approximation received quite an extensive amount of attention and by now there are many existence proofs and also algorithmic constructions that give schedules of length $O(\mathsf{congestion} + \mathsf{dilation})$ for packet routing. That is, schedules within an $O(1)$ factor from the trivial lower bound. See e.g. [10, 22, 23, 28, 31, 33, 34, 37]. The classical method [22] is based on $\log^* n$ levels of recursively applying the Lovasz's local lemma[1], each time reducing the parameter $\mathsf{congestion} + \mathsf{dilation}$ of the new problem to a polylogarithmic function of the same parameter in the problem of the previous level.

Going back to the question of scheduling general distributed algorithms, the random delays technique proves useful here as well. If nodes have access to shared randomness, the same simple random delays trick as in [22] provides a schedule for general distributed algorithms that is within an $O(\log n)$ factor of the trivial lower bound. Here the shared randomness is helpful because for each distributed algorithm, there can be many (potentially faraway) nodes that start it and delaying the algorithm by a random delay (with a controlled probability distribution) requires all nodes to do so in a consistent manner.

Theorem 1.1 (**Simple extension of [22]**). *Given shared randomness, one can distributedly find a schedule that runs all the distributed algorithms in $O(\mathsf{congestion} + \mathsf{dilation} \cdot \log n)$ round, with high probability.*

Proof Sketch. Break time into phases, each having $\Theta(\log n)$ rounds. We treat each phase as one round in the sense that the communications of each round of each algorithm will be executed during one phase. We delay the start of each algorithm by a uniform random delay in $[O(\mathsf{congestion}/\log n)]$ phases. Chernoff bound shows that w.h.p., for each edge and each phase, $O(\log n)$ messages are scheduled to traverse this edge in this phase. Hence, all algorithms will run along each other with no interference, and each will be done after at most $O(\mathsf{congestion}/\log n) + \mathsf{dilation}$ phases. □

The above theorem leaves us with two main questions:

Questions:

1. Can one remove the $\log n$ factor in the bound of Theorem 1.1 and get an $O(\mathsf{congestion} + \mathsf{dilation})$ round schedule for general distributed algorithms, perhaps using ideas similar to [10, 22, 23, 28, 31, 33, 34, 37]?
2. What can we do with only private randomness, i.e., without any shared randomness?

Technical Contributions: Regarding the first question, we show that interestingly, unlike in packet routing, when scheduling general distributed algorithms, the lower bound is the one that can be improved to essentially match the simple upper bound. Concretely, using the probabilistic method [6], we construct a hard instance of the scheduling problem which shows that:

[1] In fact, the packet routing problem and this LLL-baed method of it is now one of the material typically covered in courses on (or around) randomized algorithms for introducing the Lovasz's Local Lemma, see e.g. [1, 2, 4, 5].

Theorem 1.2. *There are instances of distributed algorithm scheduling for which any schedule, even when constructed centralized, requires* $\Omega(\mathsf{congestion} + \mathsf{dilation} \cdot \log n / \log\log n)$ *rounds.*

As for the second question, that is shared randomness, we face two rather orthogonal issues: One is about the amount of randomness to be shared and the other is how (and where) to share even a single bit of randomness. While the first issue is simple, the second requires more work. For the first issue, the saving grace is that for the proof of Theorem 1.1, $\Theta(\log n)$-wise independence between the values of random delays is enough[2] and thus, thanks to the standard bounded-independence randomness constructions (e.g., via Reed-Solomon codes), sharing simply $O(\log^2 n)$ bits of randomness is sufficient. As for the second issue, clearly one can elect a leader to pick the required initial "shared" randomness and broadcast it to all nodes. However, this, and moreover any such global sharing procedure, will need at least $\Omega(D)$ rounds, for D being the network diameter, which is not desirable. We explain how to solve the problem with private randomness in a time close to the case with shared randomness.

Theorem 1.3. *For any instance of distributed algorithm scheduling, there is a randomized distributed algorithm using only private randomness that, with high probability, produces a schedule that runs all the algorithms in* $O(\mathsf{congestion} + \mathsf{dilation} \cdot \log n)$ *rounds, after* $O(\mathsf{dilation} \log^2 n)$ *rounds of pre-computation.*

Roughly speaking, the approach is to break each algorithm into sparsely overlapping sub-algorithms that each span only small areas of the network. Particularly, each of these sub-algorithms will span an area of (weak) diameter $O(\mathsf{dilation} \log n)$ hops. Then, we only share randomness inside these smaller areas and show how to run these sub-algorithms in a way that they simulate the main algorithms.

This approach is in fact more general and it can be used to remove the assumption of having shared randomness in a broad family of randomized distributed algorithms, at the cost of an $O(\log^2 n)$ factor increase in their running time. Roughly speaking, the family that this result applies to is those algorithms in which each node outputs one (canonical) output in the majority of the executions of the algorithm (for each given input). That is, algorithms where the randomness does not effect the output (significantly) and is used only to speed up the computation. We note that recently this class was termed *Bellagio algorithms* [15, 17] as a subclass of randomized algorithms with some desirable pseudo-deterministic behavior. Due the space limitations, the explanation of this generalization is deferred to Appendix A.

2 Preliminaries

To talk about running many distributed algorithms, we need to be precise about how we view each algorithm. Particularly, the aspect that we will focus on is their *communication pattern* [25, Page 143], that is, in which rounds and over which edges does each algorithm send messages.

Communication Pattern: We define the T-round *time-expanded* graph $G \times [T]$ of a network $G = (V, E)$ as follows: We have $T + 1$ copies of V, denoted by $V_0, V_1, \ldots, V_T$, and T copies of E, denoted $E_1, E_2, \ldots, E_T$, where E_i defines the set of edges between V_{i-1} and V_i. For ease of reference, we will refer to the copy of node $v \in V$ that is in V_i as v_i. For each $i \in [1, T]$, $v_i \in V_i$ is connected with a direct edge to $u_{i+1} \in V_{i+1}$, that is $(v_i, u_{i+1}) \in E_{i+1}$, if and only if $(v, u) \in E$.

[2] See [35, Theorem 5] for a Chernoff bound for k-wise independent random variables.

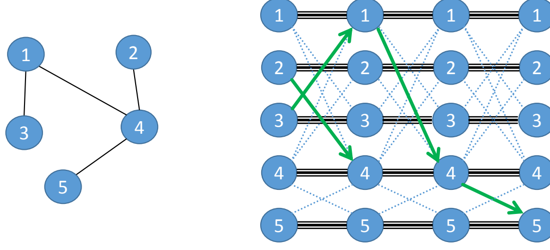

Figure 1: A simple graph, its 3 round time-expanded version, and a communication pattern on it. In round 1, nodes 2 and 3 send messages respectively to nodes 4 and 1; in round 2, node 1 sends a message to node 4; in round 3, node 4 sends a message to node 5.

The communications of a T-round algorithm A on network G correspond naturally to a subgraph of $G \times [T]$: For $v \in V_i$ and $u \in V_{i+1}$, we have $(v_i, u_{i+1}) \in A$ if and only if algorithm A sends a message from v to u in round i. We call this subgraph the *communication pattern of* A. Figure 1 shows a simple example. Note that the communication pattern of an algorithm simply captures the footprint of its communications and does not consider the content of the messages sent by the algorithm.

Generally, the communication pattern of an algorithm is non-deterministic and can depend on the inputs to the nodes and also on their randomness. Hence, it is vital that we do not assume it to be known a priori. Often the communication pattern itself already conveys significant information about the things that the algorithm is supposed to compute, which means that nodes cannot know it in advance, i.e., before running the algorithm. Even throughout the execution, a node v might not know which of its neighbors will send a message to v in the next round. A simple example to stress the vitality of this issue is the case of breadth first search: before running the BFS, node v does not know at which round and from which neighbors it will receive a message[3].

Simulation: Formally, we describe a simulation of a T-round algorithm A in a larger time-span $T' \geq T$ to be a mapping f from A to a sub-graph $B \subseteq G \times [T']$. More precisely, function f maps the edges (v_i, u_{i+1}) of algorithm $A \subseteq G \times [T]$, i.e., its communications, to edges in $G \times [T']$ in a way that preserves the *causal precedence*: that is, if in algorithm A, edge (v_i, u_{i+1}) causally precedes edge (v'_j, u'_{j+1}), then in B, edge $f((v_i, u_{i+1}))$ causally precedes edge $f((v'_{k_\ell}, u'_{j+1}))$. Here, *causal precedence* is formalized as follows: In an algorithm A, we say edge (v_i, u_{i+1}) *causally precedes* (v'_j, u'_{j+1}) if there is chain of edges $e_1 = (v_i, u_{i+1})$, $(u_{k_1}, w_{k_1+1}), \ldots,$ $(w_{k_\ell}, v'_{k_\ell+1})$, (v'_j, u'_{j+1}) that are all in A and we have $i + 1 \leq k_1 \leq \cdots \leq k_\ell \leq k_\ell + 1 \leq j$.

The Distributed Algorithm Scheduling (DAS) Problem: We are given k algorithms $\mathcal{A}_1$ to $\mathcal{A}_k$, and we should produce an execution so that for each algorithm, each node outputs the same value as if that algorithm was run alone.

Our lower bounds are strong in the sense that they apply to centralized scheduling, which means that everything—particularly the whole network topology and also the communication patterns of all the algorithms—is known to the scheduler, and the lower bound simply says that *there is no "short" schedule*, rather than saying that *a "short" schedule cannot be computed (distributedly)*.

For the purpose of our distributed scheduling algorithm (i.e., upper bound), we assume that the distributed algorithms $\mathcal{A}_1$ to $\mathcal{A}_k$ are given in the following format: For each algorithm $\mathcal{A}_i$, when this algorithm is run alone, in each round each node knows what to send in the next round. This clearly

[3]One can add dummy messages to BFS to fix its communication pattern. However, this will significantly increase the load on edges, i.e., congestion, which is undesirable.

depends on the node's input and also what messages the node has received up to, and including, that round. We stress that, we do not assume that the nodes know the correct communication pattern a priori, and in fact, if for some reason the schedule is mixed up and the node does not receive one of the messages that it is supposed to receive in the alone execution of $\mathcal{A}_i$, the node might not notice this and it can proceed with executing the algorithm, although generating a wrong execution. Our algorithms are designed such that they ensure that this does not happen. As for the randomness used by the algorithms $\mathcal{A}_1$ to $\mathcal{A}_k$, we consider it as a part of the input to the node, that is, at the start of the execution, each node samples its bits of randomness, thus fixing them, and it uses these bits throughout the execution, similar to the way it uses its input.

In our algorithms, we will have (a number of) executions of the algorithm $\mathcal{A}_i$ that globally might look different than a simulation of the algorithm $\mathcal{A}_i$, but from the viewpoint of some nodes, they are exactly as if the whole simulation is run. To capture this formally, we define a *partial simulation* of algorithm A for node v to be an execution in which, all the communications that causally influence v are identical to those in a (full) simulation of A.

For our upper bound result, we assume that nodes know constant-factor approximations of $\mathsf{congestion}$ and $\mathsf{dilation}$; although both of these assumptions can be removed using standard *doubling* techniques. We defer the explanation of these steps to the full version of this paper.

3 Lower Bound

Here we show that the $\log n$ factor in Theorem 1.1 is essentially unavoidable. That is, unlike packet routing [22], general distributed algorithms do not always admit a schedule of length $O(\mathsf{congestion} + \mathsf{dilation})$.

Theorem 3.1. *There is a distributed algorithm scheduling problem instance for which any schedule, even those computed centralized, requires at least* $\Omega(\mathsf{congestion} + \mathsf{dilation} \cdot \log n / \log\log n)$ *rounds.*

Proof. We first explain the general outline of the proof. We use the probabilistic method [6] to show the existence of such a "*hard*" problem instance. Particularly, we present a probability distribution space for distributed algorithm scheduling problems and show that, for a sample instance taken from this distribution, with a nonzero probability (in fact with probability almost 1), no schedule of length $o(\mathsf{congestion} + \mathsf{dilation} \cdot \log n / \log\log n)$ exists.

For that, roughly speaking, we study each fixed schedule of length $o(\mathsf{congestion} + \mathsf{dilation} \cdot \log n / \log\log n)$ against a random sample from the scheduling problem distribution, and we show that this one schedule has an extremely small probability to be good for the sample scheduling problem. This probability will be so small that, even after we take a union bound over all possible schedules, the probability that there exists one of these schedules that is good for the sample scheduling problem is strictly less than 1, and in fact close to 0. Hence, we conclude that there exists one problem instance of distributed algorithm scheduling which does not admit any $o(\mathsf{congestion} + \mathsf{dilation} \cdot \log n / \log\log n)$ round schedule. We next explain how we realize this outline.

We start with describing the probability distribution of the algorithms $\mathcal{A}_1$ to $\mathcal{A}_k$ that are supposed to be scheduled. The network $G = (V, E)$ is as follows: $V = \{v_0, v_1, \dots, v_L\} \cup U_1 \cup U_2 \cup \dots \cup U_L$, where $L = n^{0.1}$. Each set U_i contains $\eta = n^{0.9}$ nodes, and each node $u \in U_i$ is connected to v_{i-1} and to v_i. Figure 2 illustrates the structure of this network (and also the edges in the communication pattern of one sample algorithm, which will be discussed soon). The general format of each algorithm $\mathcal{A}_i$ is as follows: in round 1, node v_0 sends a message to a subset $S_1 \subset U_1$, where $|S_1| = \Theta(n^{0.9})$, the choice of S_1 will be described. Then in round 2, these nodes S_1 each send a message to v_1. In round 3, node v_1 sends a message to a subset $S_2 \subset U_2$, and in round 4 these nodes send messages to v_2.

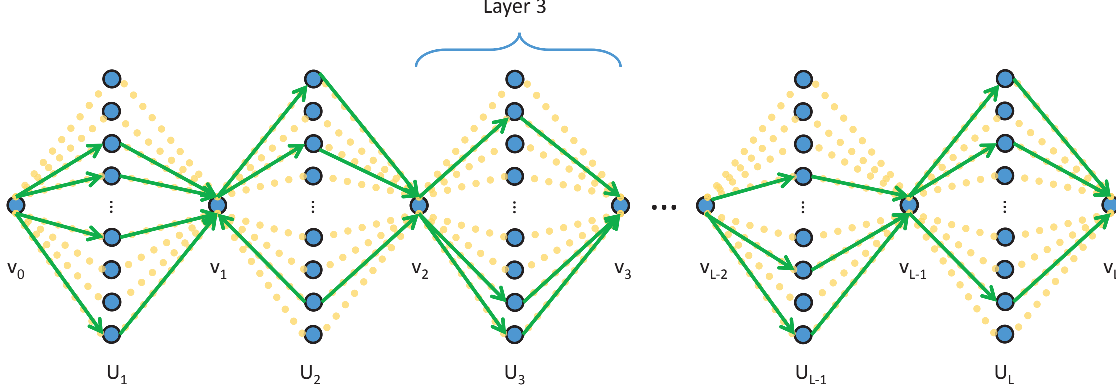

Figure 2: A sample algorithm in the hard distribution. The green links indicate all the edges used by one sampled algorithm (i.e., roughly speaking those in its communication pattern), while the orange dotted lines present all the possible edges that the algorithms in the distribution can use.

The algorithm proceeds similarly over the next blocks, continuing with a speed of one hop progress per round, and containing an arbitrary subset $S_i \subset U_i$ where $|S_i| = \Theta(n^{0.9})$. In a random sample DAS problem from our distribution, for each algorithm $\mathcal{A}_i$, each S_j is determined randomly where each node $u \in U_j$ is added to S_j with probability $n^{-0.1}$, and the choices are independent between different nodes of the same layer, different layers, and different algorithms. Note that if we have $k = n^{0.2}$ algorithms, then $\mathbb{E}[\mathsf{congestion}] = kn^{-0.1} = n^{0.1}$, and thanks to the independence, we know that

$$\Pr[\mathsf{congestion} \geq 2n^{-0.1}] \leq e^{-\Theta(n^{0.1})}.$$

We claim that there is a sample DAS problem in this distribution where $\mathsf{congestion} = O(n^{0.1})$, $\mathsf{dilation} = 2n^{0.1}$, and for which there is no schedule of length at most $\frac{n^{0.1}\log n}{1000\log\log n}$.

To prove this claim, we break the time into $0.1n^{0.1}$ phases of $\frac{\log n}{100\log\log n}$ rounds each. Furthermore, in the schedule, we say algorithm $\mathcal{A}_i$ crosses layer j in a given phase if the message of v_{j-1} is sent in this phase and v_j receives the responses of all the related nodes in U_j during this phase. Even though it is possible that crossing a layer does not fully fall into one phase, it must be true that for at least $\frac{9}{10}$ of the layers, they are each crossed in a phase. This is because otherwise the length of the schedule would be more than $\frac{n^{0.1}}{10} \cdot \frac{\log n}{100\log\log n} = \frac{n^{0.1}\log n}{1000\log\log n}$. For each of the algorithms, we define its *crossing pattering* to be the (partial) assignment of layers to phases which indicates that layer $j \in [n^{0.1}]$ is crossed during phase $t \in [0.1n^{0.1}]$, and at most 0.1 fraction of the layers do not have a phase number assigned to them. For each layer $j \in [n^{0.1}]$ and each phase $t \in [0.1n^{0.1}]$, the load $L(j,t)$ in layer j at phase t is defined to be total number of algorithms that are scheduled to cross layer j during phase t. Since there are $k = n^{0.2}$ algorithms and each of them should cross at least $0.9n^{0.1}$ layers during phases, we have

$$\sum_{j,t} L(j,t) \geq n^{0.2} \cdot 0.9n^{0.1} = 0.9n^{0.3}.$$

Now there are $0.1n^{0.2}$ choices for pair (j,t). Hence, the average load over these pairs is at least $0.9n^{0.1}$, which means that there is at least one pair with such a "semi-large" load of at least $0.9n^{0.1}$. We next focus on this layer-phase pair.

When we are studying this heavily loaded layer-phase in a fixed schedule against the randomly chosen set of algorithms, we get that on average, in this phase, the expected number of messages

that have to go through an edge is at least $0.9n^{0.1} \cdot n^{-0.1} = 0.9$. This is because, each algorithm uses each edge with probability $n^{-0.1}$ and by the choice of this layer-phase, there are $0.9n^{0.1}$ algorithms that cross this layer in this phase. We next use a simple anti-concentration type of argument to say that there is a non-negligible chance that a given edge gets considerably more than this expected load (in this phase). Note that if more than $\frac{\log n}{100 \log\log n}$ algorithms are supposed to cross a single edge during one phase, then the schedule fails simply because the phase has only $\frac{\log n}{100 \log\log n}$ rounds. The probability that the number of algorithms that have to use a given fixed edge is more than $\frac{\log n}{100 \log\log n}$ is at least

$$\sum_{\ell=\frac{\log n}{100 \log\log n}}^{0.9n^{0.1}} \binom{0.9n^{0.1}}{\ell} (n^{-0.1})^{\ell} (1 - n^{-0.1})^{0.9n^{0.1}-\ell} \geq n^{-0.2}.$$

Let us now zoom in on one side of the layer, e.g., the side from U_j to node v_j. Note that for each algorithm $\mathcal{A}_i$, the choices of different nodes being in S_j—whether different edges of this side of a layer j are used in $\mathcal{A}_i$—are independent. Thus, the load on different edges of this side of layer j are independent. This means that the probability that there is no such heavily loaded edge is at most $(1 - n^{-0.2})^{n^{0.9}} \leq 1 - e^{-n^{0.7}}$. That is, the probability that the one fixed crossing pattern we were studying is good for the given random instance of the distributed algorithm scheduling is extremely small, at most $e^{-n^{0.7}}$.

Given that one crossing pattern is very unlikely to succeed, we now explain why none of them is likely to succeed, using a union bound. The number of all possible crossing patterns for all algorithms is at most $e^{\Theta(n^{0.3})}$. This is because, for a single algorithm, to define a possible crossing patterns, we need to specify at most $0.1n^{0.1}$ layers that are not crossed during a phase, and also specify the crossing phase for the rest of the layers. For the layers that are not crossed during a phase, there are $\binom{n^{0.1}}{0.1n^{0.1}} = e^{O(n^{0.1})}$ options. To assign increasing phase numbers in range $[n^{0.1}]$ to the remaining $0.9n^{0.1}$ layers, there are $\binom{1.9n^{0.1}-1}{0.9n^{0.1}} = e^{O(n^{0.1})}$ options[4]. Hence, the total number of possible crossing patterns for a single algorithm is at most $e^{\Theta(n^{0.1})}$, which means that over all the $k = n^{0.2}$ algorithms, the number of all possible crossing patterns is at most $e^{\Theta(n^{0.3})}$.

Now, a union bound over all crossing patterns tells us that the probability that there is one of the crossing patterns that does not fail for the sampled DAS problem is at most $e^{-n^{0.7}} \cdot e^{\Theta(n^{0.3})} \ll 1$. Therefore, there is one DAS problem in the described family for which there is no (short) crossing pattern, and hence, also no schedule of length at most $\frac{n^{0.1} \log n}{1000 \log\log n}$ rounds. □

Remark: We note that the schedule length bound that appears in the proof of the above theorem is indeed tight up to a constant factor for the described setting. This is because, using some small parameter tuning in Theorem 1.1, we can generate a schedule of length $\Omega((\mathsf{congestion} + \mathsf{dilation}) \cdot \log n / \log\log n)$, which matches the bound in the proof as there $\mathsf{congestion} = \Theta(\mathsf{dilation})$. To get this schedule, we use phases of $\Theta(\log n / \log\log n)$ rounds and delay each algorithm by a random number of phases uniformly distributed in $[\Theta(\mathsf{congestion})]$. Thus, the expected number of messages to be sent across an edge per phase is $O(1)$ which means w.h.p., this number will not exceed $O(\log n / \log\log n)$.

4 Upper Bound

We start by giving an intuitive explanation of the challenge in scheduling distributed algorithms and a high level description of the ideas we use. We then go on to describe the details.

[4]Recall that this is essentially a case of the *stars and bars* problem: phases are the stars and layers are the bars, and how many stars there are before a bar indicates the phase in which the related layer is crossed.

In Appendix A, we sketch out how to use this approach to remove the assumption of having shared randomness in a broad range of distributed algorithms, at the cost of a slow down factor of $O(\log^2 n)$.

4.1 The Challenge, and the Outline of Our Approach

The key insight in the random delays trick of [22], explained in Section 1, is that this random per-algorithm delay spreads different algorithms over time such that the number of packets scheduled to go through an edge per time goes down. When working with distributed algorithms, implementing such a random delay (with a controlled probability distribution) might become a non-trivial task. This is mainly because of the local nature of distributed algorithms: typically, while the algorithm is being run over the whole network, each node is influenced only by a small neighborhood around itself, particularly a ball of radius dilation-hops, in our terminology. If we are to keep the running time small, there cannot be any causal dependency between nodes that are far from each other. Hence, particularly, unless we have access to shared randomness, we cannot have a randomly chosen delay for a given algorithm in a way that the delay is "consistent" over the whole network.

The good news however is that we actually do not need such a consistency over the whole network; it is sufficient if we have a consistently delayed execution in each dilation-neighborhood. This is because each node actually sees only a dilation-neighborhood of itself. However, leveraging this positive point is non-trivial, mainly because each edge can be in the dilation-neighborhoods of a large number of nodes, and executing algorithms in each of these neighborhoods separately will need an undesirably large running time.

The solution we use is based on a generic method of *graph partitioning* [9,11,13,21], colloquially referred to as *ball carving*. Particularly, the closest to what we do is the approach Bartal [8] used for probabilistically approximating arbitrary metric spaces with tree-metrics. The end result of the graph partitioning part will be a number of clusters of weak radius $O(\mathsf{dilation} \cdot \log n)$ such that each dilation-neighborhood is covered completely in at least one, and in fact $\Theta(\log n)$, of the clusters and moreover, each edge is in at most $O(\log n)$ clusters. To efficiently implement this idea in the CONGEST, we also use a number of small tricks, which we believe might be useful in other distributed graph partitioning algorithms for the CONGEST model.

Once we have the aforementioned partitioning, we share a sufficient amount of randomness in each of these clusters (efficiently); these randomness are to be fed to a $\Theta(\log n)$-wise independent pseudo-random generator. We then show how to use these locally shared randomness bits to simulate all the distributed algorithms in all the clusters in $O(\mathsf{congestion} + \mathsf{dilation} \cdot \log n)$ rounds, i.e., producing a schedule of length $O(\mathsf{congestion} + \mathsf{dilation} \cdot \log n)$ rounds. We also explain how to let each node pick the right execution for each of the algorithms, from among all the simulations in different clustering layers the node is involved in.

4.2 Scheduling Distributed Algorithms via Locally Sharing Randomness

Here, we describe the algorithm that realizes the above outline and achieves the following result:

Theorem 4.1. *There is a* DAS *algorithm with only private randomness with* $O(\mathsf{dilation} \log^2 n)$ *rounds of pre-computation that simulates all the algorithms in* $O(\mathsf{congestion} + \mathsf{dilation} \cdot \log n)$ *rounds.*

For the graph partitioning mentioned above, which effectively breaks each algorithm into sub-algorithms in local areas, we present the following lemma:

Lemma 4.2. *There is a distributed algorithm that runs in* $O(\mathsf{dilation} \log^2 n)$ *rounds and creates* $\Theta(\log n)$ *layers of clustering of the graph such that: (1) in each layer, the clusters are node-disjoint,*

(2) each cluster has weak diameter $O(\mathsf{dilation} \cdot \log n)$, (3) w.h.p, for each node v, there are $\Theta(\log n)$ layers such that the dilation*-neighborhood of v is fully contained in one of the clusters of this layer, and (4) in each layer, each node knows what is the maximum radius around it that is fully contained in a cluster.*

Proof. What we do in the $\Theta(\log n)$ layers of clustering is independent repetitions of the following scheme: each node u picks a random radius $r(u)$ distributed according to a truncated exponential distribution, where $\Pr[r(u) = z] = (\frac{n}{n-1})\frac{1}{R}e^{-z/R}$ for $R = \Theta(\mathsf{dilation})$. Node u also picks a random label $\ell(u) \in \{0,1\}^{4\log n}$. Note that w.h.p., for each two nodes $u \neq u$, $\ell(u) \neq \ell(u')$. The radius $r(u)$ defines a ball of radius $r(u)$ centered at u, which we denote $B(u)$. Each node v joins the cluster centered at node w^* where w^* is defined as the node that has the smallest label $\ell(w^*)$ among the labels of nodes w such that $v \in B(w)$. Properties (1) and (2) follow immediately from the definition and the property (3) follows from the analysis of Bartal [8, Section 3] which shows that in each layer, each dilation-neighborhood is fully contained in one of the clusters with constant probability.

What is left to discuss is the distributed implementation and its time complexity, and achieving property (4). We run the layers of clustering separately, each in $O(\mathsf{dilation} \log n)$ rounds, thus all together in $O(\mathsf{dilation} \log^2 n)$ rounds. To save time in the distributed implementation, we will add a small trick. Interestingly, once we move to Lemma 4.3, this simple change will end up saving us two logarithmic factors in the running time. Each node u initiates a message m_u containing its id and label, and an initial hop-count $H - r(u)$, where $H = \Theta(\mathsf{dilation} \log n)$. Note that w.h.p., for each node u, we have $r_u < H$. This (fake) initial hop-count virtually implies that the message m_v is coming from a different node and it has already traveled $H - r(u)$ hops by now and it can only travel $r(u)$ more hops, as we allow each message to travel H hops in total.

Then in each round $i \in [1, H]$, each node v forwards the message with hop-count i that has the smallest label among the messages of hop-count i or smaller that have reached v. Each received message gets its hop-count incremented by 1. Due to the initial hop-counts, the message m_u can reach only nodes that are in $B(u)$. Furthermore, for any node v, if $\ell(u)$ is the smallest label among the labels of balls that contain v, then m_u will indeed reach v. The reason is as follows: if m_u does not reach v, it must be that in a round i, there is a node w on the shortest path from u to v that has received m_u but does not forward m_u because it instead forwards (or has forwarded) $m_{v'}$ for a node v' such that $\ell(v') < \ell(v)$ and at w, the hop count of $m_{v'}$ was less than or equal to that of m_v. But this implies that u is in fact in $B(v')$, which is in contradiction with v having the smallest label among the balls that contain u. Finally, $\ell(u)$ will be the smallest label that v hears, and thus v will join the cluster centered at u.

To achieve property (4), that is each node knowing what radius of it is fully contained in a cluster, we do as follows for each layer: First, each node sends its cluster label to each neighbor. Then, if a node sees a different label in that round, it marks itself as cluster boundary. In the next dilation rounds, each boundary node sends a message declaring it is boundary and for the next $O(\mathsf{dilation} \log n)$ rounds, if a node hears a boundary message, it forwards it to its neighbors (if it has not done so in the past). Hence, if h' neighborhood of node v is the maximum neighborhood fully contained in a cluster, v will receive the boundary message after h' rounds, thus letting node v learn the value of h'. □

Lemma 4.3. *We can share $\Theta(\log^2 n)$ bits of randomness in each cluster, all together in $O(\mathsf{dilation} \log^2 n)$ rounds. Furthermore, via local computations, this can be turned to $\mathrm{poly}(n)$ many $\Theta(\log n)$-bit random values that are $\Theta(\log n)$-wise independent.*

Proof of Lemma 4.3. If we had to share $\Theta(\log n)$ bits of randomness in each cluster, then each cluster center node would simply pick this randomness and append it to its initial message. Sharing the

$\Theta(\log^2 n)$ bits requires sending $O(\log n)$ messages from each cluster center. Thus, a naive extension of the approach in Lemma 4.2 would become $O(\mathsf{dilation} \log^2 n)$ rounds per layer, i.e., $O(\mathsf{dilation} \log^3 n)$ in total.

We can do the spreading in each layer in $O(\mathsf{dilation} \log n)$ rounds as follows: make $\Theta(\log n)$ messages from each source v, each containing $\Theta(\log n)$ bits of randomness, all with the same label $\ell(v)$, and an additional smaller label $\ell'(v)$ picked from $\{1, 2, \Theta(\log n)\}$ to distinguish these messages. The initial hop-count is set as in Lemma 4.2 for all these messages. Then, in each round, each node forwards the message with lexicographically smallest (hop-count, $\ell(v), \ell'(v)$) that it has not sent before. The pipelining analysis of Lenzen [24] shows that after $H + \Theta(\log n) = O(\mathsf{dilation} \log n)$ rounds of this protocol, each node v will receive the smallest $\Theta(\log n)$ messages starting in the H neighborhood of v, where smallest is with respect to the lexicographical ordering of $(\ell(v), \ell'(v))$. Because of the initial hop-count settings, this H-neighborhood translates to all centers that could potentially reach v, and particularly means node v will receive all the $\Theta(\log n)$ messages coming from the center u of the cluster that contains v. Repeating this scheme $\Theta(\log n)$ times, once for each clustering layer, solves the local randomness sharing and gets us to $O(\mathsf{dilation} \log^2 n)$ rounds[5].

We now talk about increasing the "size" of randomness as the $O(\log^2 n)$ bits of randomness shared in each cluster is not (directly) sufficient for determining the random delays of all the algorithms. Each node v feeds the $\Theta(\log^2 n)$ bits of shared randomness of each cluster that contains v into the classical k-wise independent pseudo-randomness construction via Reed-Solomon codes, for $k = \Theta(\log n)$, (see e.g., [3, Section 3] and also [6, Theorem 15.2.1][6]). This transforms the randomness to poly(n) many $\Theta(\log n)$-bit random values that are $\Theta(\log n)$-wise independent. These random values will be used to determine the random delays of different algorithms in a way that is consistent in each cluster and for each set of up to $\Theta(\log n)$ algorithms, their random delays are independent. Particularly, we assume each algorithm $\mathcal{A}_i$ has a unique algorithm identifier $\mathsf{AID}(i)$ in a range of $K = \text{poly}(n)$ size, say $[n^{10}]$, and we divide the generated random values into K buckets, each containing poly(n) random values, and the algorithm $\mathcal{A}_i$ will pick its random delays based on the random values in bucket $\mathsf{AID}(i)$. □

We next explain how, given this shared randomness, we derive a schedule of length $O(\mathsf{congestion} + \mathsf{dilation} \cdot \log n)$ rounds that simulates all the algorithms.

Lemma 4.4. *After the local randomness sharing as done in Lemma 4.3, we can run all the distributed algorithms in all the cluster, altogether in $O(\mathsf{congestion} + \mathsf{dilation} \cdot \log n)$ rounds.*

Proof. We first explain a simpler solution that leads to a schedule of length $O(\mathsf{congestion} \cdot \log n + \mathsf{dilation} \cdot \log n)$ and then explain how to improve this to the near-optimal schedule length $O(\mathsf{congestion} + \mathsf{dilation} \cdot \log n)$ using a nonuniform distribution of random delays.

We run a copy of each algorithm in each cluster. Hence, for each edge e, for each algorithm that uses e, there can be up to $\Theta(\log n)$ copies of each of the messages that are to be sent over e, at most one per layer. Thus, we potentially have increased the load on edges to $O(\mathsf{congestion} \cdot \log n)$.

Since we have partitioned algorithms into smaller clusters, we need to be careful that from the viewpoint of the nodes that their $\mathsf{dilation}$-neighborhood is fully contained in a cluster, the behavior of the locally-cut execution of the algorithm stays the same as in the full algorithm. Consider an

[5] We note that, it sounds plausible that one might get a bound of $O(\mathsf{dilation} \log n + \log^3 n)$ rounds for this part.

[6] We note that [6, Theorem 15.2.1] only describes the construction on the field GF(2) which yields binary random values that are k-wise independent. But, as also described in [3, Section 3], the construction readily extends to $GF(p)$, for any prime number $p \in \text{poly}(n)$. Furthermore, when desiring random delays in range $[\Theta(R)]$ for a given R, we can pick them from a range $\{1, \ldots, p\}$ for a prime $p \in \Theta(R)$. Note that by Bertrand's postulate, there are many such primes; there is at least one in $[a, 2a]$, for any $a \geq 1$.

algorithm $\mathcal{A}_i$ and a clustering layer. If for a node v, only h'-neighborhood of v is fully contained in a cluster of this layer, then v will execute only the first h' rounds of the algorithm and will discard the messages that it would be sending in the later rounds. This change does not effect the execution from the viewpoint of the nodes that their dilation-neighborhood is fully contained in a cluster. This is because, if dilation-neighborhood of node w is fully contained in a cluster, this partial execution will still contain all the messages that can effect w. More concretely, for each node v in the dilation-neighborhood of w, in the algorithm, w causally depends only on the actions of v in the first $h' = \mathsf{dilation} - dist(v, w)$ rounds, and since dilation-neighborhood of w is fully contained in a cluster, so is the h'-neighborhood of v. Moreover, it is easy to see that generally, for any message in the algorithm that it is not discarded, none of the previous messages in the same algorithm that can causally effect it is discarded.

For each cluster, for each algorithm, the start of that algorithm is delayed by a random delay of $\delta \in [\mathsf{congestion}]$ big-rounds, where each big-round is $\Theta(\log n)$ rounds. Then, each algorithm proceeds in a synchronous manner at a rate of one algorithm-round per each big-round. The execution finishes after $\mathsf{dilation} + \mathsf{congestion}$ big-rounds. W.h.p., the number of the messages scheduled per big-round per edge is $\Theta(\log n)$, thanks to the $\Theta(\log n)$-independence of the random delays. Then, each node v picks its output for each algorithm based on one of the clustering layers that fully contains the dilation-neighborhood of v in a cluster.

We now explain how to improve this schedule length, to the near-optimal bound of $O(\mathsf{congestion} + \mathsf{dilation} \cdot \log n)$ rounds. The key point we leverage is the fact that we only need one copy of each message, instead of all the potentially up to $O(\log n)$ copies (one in each layer). However, the challenge is to keep the executions synchronous (roughly speaking). The main ingredient we use is a nonuniform distribution of the random delays, suited to our setting. We next describe this distribution, and then explain how it affects the schedule.

The probability mass function of the random delay is as follows: We have a parameter $L = \Theta(\frac{\mathsf{congestion}}{\log n})$. The support of the distribution is divided to $\beta = \Theta(\log n)$ blocks. The total probability mass given to each block is $1/\beta$. The ith block contains $L\alpha^{i-1}$ points, and the distribution of the probability mass inside the block is uniform, i.e., each point of the ith block has probability $\frac{1/\beta}{L\alpha^{i-1}}$. Here, α is a positive constant slightly less than 1, which we fix later. Note that the total support of the distribution has $\sum_{i=1}^{\Theta(\log n)} L\alpha^{i-1} \leq \frac{L}{1-\alpha} = \Theta(\frac{\mathsf{congestion}}{\log n})$ points.

The above describes the distribution for the initial delay in each algorithm. Once started, the algorithms will again proceed synchronously one algorithm-round per each big-round. However, this time, if there is a message scheduled to be sent over e and a copy of it has been sent before, this message gets dropped. On the other hand, when a node is about to create a message when simulating a round j of algorithm, it takes into account all the messages that it has received in the past about rounds up to $j-1$ of the simulations of the same algorithm.

What is left is to show that with the new distribution of delays, w.h.p., for each big-round, there will be at most $\Theta(\log n)$ messages that actually need to go through an edge e. Note that a message must go through e in a given big-round if no copy of it has done so in the past, i.e., if among the $\Theta(\log n)$ copies of the simulations of the corresponding algorithm, this is the first execution scheduled. For a copy of an algorithm to be the first scheduled, all the $\Theta(\log n)$ copies must be scheduled afterward. We show that for each big-round and each edge, the probability that the first copy of a message is scheduled to go through this edge in that round is at most $\Theta(\frac{\log n}{\mathsf{congestion}})$. This is trivially true for delays in the first block of distribution of the delays as the probability for each of those delays is $\frac{1}{L\beta}$. For the second block, the probability that none of the $\Theta(\log n)$ copies of the algorithm have a delay in the first block is $(1-\frac{1}{\beta})^{\Theta(\log n)}$ which is a constant, say $\gamma < 1$. It is sufficient to pick the constant α equal to this constant γ. Then, the probability that a delay that is in the

second block is picked and is the first among the delays is at most $\frac{\gamma/\beta}{L\alpha} \leq \frac{1}{L\beta}$. Generally, for a delay in the ith block, the probability that it is the first of delays is at most $(1-\frac{1}{\beta})^{(i-1)\Theta(\log n)} \leq \gamma^{i-1}$, and hence, the probability that it is picked and is the first among the related delays is at most $\frac{\gamma^{i-1}/\beta}{L\alpha^{i-1}} \leq \frac{1}{L\beta} = \Theta(\frac{\log n}{\mathsf{congestion}})$. Hence, we conclude that the total number of messages to be actually sent over each edge per big-round is at most $\Theta(\log n)$, with high probability. Since each big-round is made of $\Theta(\log n)$ rounds, we can afford to send all these messages across the edge in that big-round. Note that the range of the initial delays is $\Theta(\frac{\mathsf{congestion}}{\log n})$ big-rounds and each algorithm runs for only dilation big-rounds. Hence, the whole schedule has a length $O(\mathsf{congestion} + \mathsf{dilation} \cdot \log n)$ rounds, thus completing the proof of the lemma. □

5 Concluding Remarks and Future Work

This paper is centered around the issue of running many independent distributed algorithms concurrently. We presented an existential $\Omega(\mathsf{congestion} + \mathsf{dilation} \log n / \log\log n)$ round schedule length lower bound and an algorithm that produces a schedule of length $O(\mathsf{congestion} + \mathsf{dilation} \log n)$ rounds, after $O(\mathsf{dilation} \log^2 n)$ rounds of pre-computation.

We believe that this paper is merely a starting point, as there are many seemingly fundamental aspects that are not addressed here. We next discuss some of these issue, which are conceptually deeper and perhaps somewhat less clear, as well as some other more detailed technical questions.

Broader Questions: Throughout the paper, we worked under the assumption that the algorithms $\mathcal{A}_1$ to $\mathcal{A}_k$ are fixed, and we just want to run them. However, one can imagine a different viewpoint, when considering a higher-level picture: The end goal that we have is to solve the problems that $\mathcal{A}_1$ to $\mathcal{A}_k$ were designed to solve, which means we are allowed to change these algorithms to make them fit with each other better, so long as they solve the same problems. In other words, given k problems, what is the best collection of algorithms $\mathcal{A}_1$ to $\mathcal{A}_k$ solving these problems that can be run together in the shortest possible span of time. This roughly coincides with, what is the collection of algorithms $\mathcal{A}_1$ to $\mathcal{A}_k$ solving these problems that leads to the smallest possible congestion and dilation. To unify these two measures and make the problem well-defined, one might consider $\mathsf{congestion} + \mathsf{dilation} \cdot \log n$ as the objective that is to be minimized. In fact, once we design a set of algorithms optimizing this measure, then we can use the algorithms presented in this paper to run $\mathcal{A}_1$ to $\mathcal{A}_k$ together essentially optimally, which would be a near-optimal method overall. Hence, one can say that, the algorithmic results in this paper essentially (and approximately) reduce the general question of how to solve k problems together fast to that of designing algorithms that minimize the measure $\mathsf{congestion} + \mathsf{dilation} \cdot \log n$.

Now this latter design question still seems rather broad and it is not clear how to tackle such a question for the general choice of k problems. A clean special case which might provide us with valuable insights is the setting where the k problems are independent instances of the same problem. That is, the setting where we have k independent shots of a single problem, and we want to solve all of them.

We note that this question loosely resembles the well-studied *parallel repetitions* problem (see e.g. [32]) in the complexity theory which, roughly speaking, asks can one solve k independent shots of a given problem using "resources" less than k times that of the single-shot version. However, the author believes that this is rather a superficial similarity and the connection is not strong, mainly because our focus is on the distributed round complexity and we clearly know that pipelining often allows us to solve k shots faster than k times the round complexity of solving a single shot. Although, other less-direct connections between the two areas are plausible. For instance, one can use *direct product* or *direct sum theorems* of communication complexity to derive distributed round complexity

lower bounds for k-shot versions of a given distributed problem.

Let us now take a closer look at this distributed k-shot question, by using a classical (single-shot) problem as an example. Consider the k-shot version of the MST problem: For the network graph $G = (V, E)$, we are given k different weight functions w_1 to w_k, each $w_i : E \to \mathbb{R}$, and we want to find the k Minimum Spanning Trees corresponding to these weight functions. It is easy to see that for MST (and in fact for most problems), the naive idea of running k copies of the best single-shot algorithm, even with the best possible pipelining/scheduling, is not optimal. The core of the matter is that, the single-shot algorithm is designed to minimize its running time, i.e., dilation in our terminology, but it typically does not lead to the best congestion.

Let us make the discussion more concrete by considering some examples. For MST, the almost a century-old algorithm of Boruvka [26] from 1926, which has the same outline as the one used by Galleger, Humblet, and Spira [14], has dilation $= \tilde{O}(n)$ rounds and running it once gives a very low congestion of $O(\log n)$. Other algorithms give different values for these two parameters. For instance, an alternative near-linear time algorithm can be achieved by filtering edges—discarding heaviest edge in each cycle—while they are being upcast on a tree towards the root, which leads to dilation and congestion both being $\tilde{O}(n)$. Furthermore, the newer algorithm of Kutten and Peleg [20] has an almost optimal running time of dilation $= \tilde{O}(D+\sqrt{n})$ rounds, but (as is) it has congestion $= \Theta(\sqrt{n})$. This simple example is a witness to the clear fact that the algorithms designed to have optimal dilation do not necessarily have good congestion.

For the case of the MST problem, we can actually understand the tradeoff between congestion and dilation quite well: Using constructs similar to those of Das Sarma et al. [12], one can see that there is indeed an inevitable tradeoff and an MST algorithm that has congestion at most L will require a dilation of at least $\tilde{\Omega}(n/L)$ rounds. Interestingly, using some simple parameter tuning in the algorithm of Kutten and Peleg [20], one can algorithmically match this tradeoff of

$$\mathsf{congestion} = L, \mathsf{dilation} = \tilde{\Theta}(D + n/L).$$

This single-short tradeoff also has direct implications for the k-shot version of the problem. For the k-shot version, the aforementioned lower bound can be combined with a *strong direct sum theorem* for communication complexity of k independent shots of two-party set disjointness [19], and some parameter tuning, to show that solving k-shots of MST requires at least $\tilde{\Omega}(D + \sqrt{kn})$ rounds. Again, this can be matched algorithmically: we use the parameter-optimized algorithm of Kutten and Peleg, with the parameter choice of $L = \sqrt{n/k}$, as our single-shot algorithm, and we run k copies of it in parallel using (even a simplified version of) our $O(\mathsf{congestion} + \mathsf{dilation} \log n)$ round schedule. As a result, we get an algorithm that solves k independent instances of MST in $\tilde{O}(D + \sqrt{nk})$ rounds, thus essentially matching the aforementioned lower bound.

Perhaps it is simply a lucky coincidence that in the case of MST, we already know how to solve k-shots of it optimally, simply by doing some parameter tuning in the known single-shot algorithms and then running k copies of it in parallel. Can we achieve such a result for a broader family of problems (hopefully, ones which have a different "nature" compared to that of MST[7])?

To summarize, we believe that when designing distributed algorithms, it is valuable to keep track of both dilation and congestion, as opposed to merely dilation (i.e., round complexity) which is the standard objective in the recent years of theoretical distributed computing. This is because, congestion gives us a sense of what will happen when we run this algorithm along with others. Of course the ideal case would be if one can have an algorithm with a controllable (and efficient)

[7] One can see that for instance, a k-shot version of minimum cut approximation can also be computed in $\tilde{O}(D+\sqrt{kn})$ rounds, by extending [16], and that also is optimal due to essentially the same lower bound constructions. However, this is not really a novel addition but merely an extension of the same tradeoff "nature".

tradeoff between the two parameters, as we saw above for the MST problem. Furthermore, this tradeoff between congestion and dilation deserves a study from a lower bound viewpoint, and even congestion deserves a study of its own. For instance, given a problem, can we solve it with a congestion below the given threshold?

As a side note, we remark that a measure that received more attention in the old days of theoretical distributed computing was *message complexity*. In fact message complexity has some correlation with congestion. However, this correlation is not strong and the message complexity alone does not characterize the related congestion. For instance, an algorithm with message complexity $O(m)$ can have congestion anywhere between $O(1)$ to $O(m)$.

Smaller Technical Questions: We next mention a few smaller questions about the technical results presented in the paper. A particular question is, the lower bound we presented in this paper is merely existential. Is it possible to provide a necessary and sufficient classification of special cases of distributed algorithm scheduling problems which admit $O(\mathsf{congestion} + \mathsf{dilation})$ round schedules? Alternatively, can we get at least a sufficient condition that covers a broad range of problems of interest? If yes, for those, can we also find these schedules distributedly and efficiently?

Another question is regarding Theorem 1.3, which gives a schedule of length $O(\mathsf{congestion} + \mathsf{dilation} \log n)$ but after $O(\mathsf{dilation} \log^2 n)$ rounds of pre-computation. Can we improve the latter, hopefully to $O(\mathsf{dilation} \log n)$ rounds?

Acknowledgment

The author thanks Nancy Lynch and Fabian Kuhn for valuable conversations about the conceptual point of the paper, i.e., studying the question of running many distributed algorithms together; Badih Ghazi, Bernhard Haeupler, Stephan Holzer, and Christoph Lenzen for discussions about different technical aspects of it; and Tsvetomira Radeva for feedback about the writeup.

This work was supported in part by a Simons award for graduate students in Theoretical Computer Science (number 318723), as well as AFOSR contract number FA9550-13-1-0042, NSF award 0939370-CCF, NSF award CCF-1217506, and NSF award CCF-AF-0937274.

References

[1] Lecture notes on Algorithmic Power Tools. http://www.ccs.neu.edu/home/rraj/Courses/7880/F09/Lectures/GeneralLLL_Apps.pdf. Accessed: 2015-02.

[2] Lecture notes on Algorithmic Power Tools. http://www.ccs.neu.edu/home/rraj/Courses/7880/F09/Lectures/PacketRouting.pdf. Accessed: 2015-02.

[3] Lecture notes on k-wise uniform (randomness) generators. http://pages.cs.wisc.edu/~dieter/Courses/2013s-CS880/Scribes/PDF/lecture04.pdf. Accessed: 2015-02.

[4] Lecture notes on Randomized Algorithms and Probabilistic Analysis. http://courses.cs.washington.edu/courses/cse525/13sp/scribe/lec10.pdf. Accessed: 2015-02.

[5] Lecture notes on Randomness and Computation. http://www.cs.berkeley.edu/~sinclair/cs271/n22.pdf. Accessed: 2015-02.

[6] N. Alon and J. H. Spencer. *The probabilistic method*. John Wiley & Sons, 2004.

[7] F. M. Auf der Heide and B. Vöcking. Shortest-path routing in arbitrary networks. *Journal of Algorithms*, 31(1):105–131, 1999.

[8] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*, pages 184–193. IEEE, 1996.

[9] Y. Bartal. Graph decomposition lemmas and their role in metric embedding methods. In *Algorithms–ESA 2004*, pages 89–97. Springer, 2004.

[10] D. Bertsimas and D. Gamarnik. Asymptotically optimal algorithms for job shop scheduling and packet routing. *Journal of Algorithms*, 33(2):296–318, 1999.

[11] G. Calinescu, H. Karloff, and Y. Rabani. Approximation algorithms for the 0-extension problem. *SIAM Journal on Computing*, 34(2):358–372, 2005.

[12] A. Das Sarma, S. Holzer, L. Kor, A. Korman, D. Nanongkai, G. Pandurangan, D. Peleg, and R. Wattenhofer. Distributed verification and hardness of distributed approximation. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 363–372, 2011.

[13] J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 448–455. ACM, 2003.

[14] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77, 1983.

[15] E. Gat and S. Goldwasser. Probabilistic search algorithms with unique answers and their cryptographic applications. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 18, page 136, 2011.

[16] M. Ghaffari and F. Kuhn. Distributed minimum cut approximation. In *Proc. of the Int'l Symp. on Dist. Comp. (DISC)*, pages 1–15, 2013.

[17] S. Goldwasser. Pseudo-deterministic algorithms. In *STACS'12 (29th Symposium on Theoretical Aspects of Computer Science)*, volume 14, pages 29–29. LIPIcs, 2012.

[18] S. Holzer and R. Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 355–364. ACM, 2012.

[19] H. Klauck. A strong direct product theorem for disjointness. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 77–86. ACM, 2010.

[20] S. Kutten and D. Peleg. Fast distributed construction of k-dominating sets and applications. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 238–251, 1995.

[21] J. R. Lee and A. Naor. Extending lipschitz functions via random metric partitions. *Inventiones mathematicae*, 160(1):59–95, 2005.

[22] F. T. Leighton, B. M. Maggs, and S. B. Rao. Packet routing and job-shop scheduling in O(congestion+ dilation) steps. *Combinatorica*, 14(2):167–186, 1994.

[23] T. Leighton, B. Maggs, and A. W. Richa. Fast algorithms for finding O(congestion+ dilation) packet routing schedules. *Combinatorica*, 19(3):375–401, 1999.

[24] C. Lenzen and D. Peleg. Efficient distributed source detection with limited bandwidth. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 375–382. ACM, 2013.

[25] N. A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[26] J. Nešetřil, E. Milková, and H. Nešetřilová. Otakar boruvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1):3–36, 2001.

[27] I. Newman. Private vs. common random bits in communication complexity. *Information processing letters*, 39(2):67–71, 1991.

[28] R. Ostrovsky and Y. Rabani. Universal O(congestion+ dilation+ $\log^{1+\epsilon} n$) local control packet switching algorithms. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 644–653. ACM, 1997.

[29] B. Peis and A. Wiese. Universal packet routing with arbitrary bandwidths and transit times. In *Integer Programming and Combinatoral Optimization*, pages 362–375. Springer, 2011.

[30] D. Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[31] Y. Rabani and É. Tardos. Distributed packet switching in arbitrary networks. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 366–375. ACM, 1996.

[32] R. Raz. A parallel repetition theorem. *SIAM Journal on Computing*, 27(3):763–803, 1998.

[33] T. Rothvoß. A simpler proof for O($\mathsf{congestion} + \mathsf{dilation}$) packet routing. In *Integer Programming and Combinatorial Optimization*, pages 336–348. Springer, 2013.

[34] C. Scheideler. *Universal routing strategies for interconnection networks*, volume 1390. Springer, 1998.

[35] J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.

[36] D. M. Topkis. Concurrent broadcast for information dissemination. *Software Engineering, IEEE Transactions on*, (10):1107–1112, 1985.

[37] A. Wiese. *Packet Routing and Scheduling*. Cuvillier, 2011.

A Removing Shared Randomness in Distributed Algorithms

Generally, having shared randomness between the nodes of a network is a questionable assumption and it is ideal to not depend on such an assumption. Here, we sketch how the approach used in Section 4 can be applied for removing the assumption of having shared randomness in a large class of randomized distributed algorithms.

The class that our approach will apply to is what Goldwasser recently named Bellagio algorithms [17], although a wide range of classical randomized algorithms fall into this category. The definition translated to distributed algorithms is that each node must output the same value—i.e., its *canonical* value—in at least $2/3$ of the executions; that is, with probability at least $2/3$. Note that in many problems, the correct solution is unique, or the problem can be slightly changed to make the solution unique, and in these problems, any randomized algorithm will indeed be a Bellagio algorithm. On the other hand, a classical distributed problem for which obtaining a fast (polylogarithmic rounds) Bellagio algorithm seems hard is the Maximal Independent Set problem.

The reason that our approach only applies to Bellagio algorithms is that we will try to simulate the algorithm with shared randomness by only locally sharing randomness, as we did in Section 4. Thus, we will have many partial executions of the algorithm, each cut to a small local area, and the Bellagio property allows us *paste* these together and make sure that the outputs of these partial executions are consistent.

Meta-Theorem A.1. *For any problem that has a T-round Bellagio randomized distributed algorithm which uses R bits of shared randomness and where each node outputs a canonical solution with probability at least $2/3$, there is a randomized algorithm with round complexity $O(T\log^2 n + R)$ that only uses private randomness and w.h.p. each node outputs its canonical solution. Furthermore, if the input given to each node can be described using $\mathrm{poly}(n)$ bits, a different technique can be used to reduce R to $O(\log n)$, thus giving a $O(T\log^2 n)$ round algorithm.*

We first use an example to explain the first part of the meta-theorem. Then, we describe the general approach for the second part which reduces the amount of the shared randomness in the original shared-randomness Bellagio algorithm to $O(\log^2 n)$, in most cases of interest, while keeping it Bellagio. Simulating this new algorithm via our techniques in Lemmas 4.2, 4.3, and 4.4, leads to the claimed round complexity of $O(T\log^2 n)$ round.

One of the key multi-party computation operations in which shared randomness gets used frequently is hashing. Many variants of hashing are using in different algorithms, but typically the hash function is constructed via a seed that comes from shared randomness. We now explain how this shared randomness assumption can be removed. To have a concrete explanation, we use a very simple case of *dimensionality reduction* via hashing as our example. We note that we will only present a rough sketch which illustrates the main idea without getting caught up in the details of the particular example problem.

Suppose each node $v \in V$ receives as input a string $s_v \in \{0,1\}^L$ for $L = \mathrm{poly}(n)$, and the objective is for each node v to know the number of distinct strings within d-hops of v, to within a $(1+\varepsilon)$ factor, for a small $\varepsilon > 0$. Note that even two neighbors exchanging their strings would require $L = \mathrm{poly}(n)$ rounds, so intuitively a first natural step is to reduce the dimension $L$of the problem, and this we do via a simple hashing.

Using shared randomness, nodes can pick a random hash function $h : \{0,1\}^L \to \{0,1\}^{\Theta(\log n)}$ at random, which would give the property that among the at most n strings in the graph, w.h.p., there is no hash-collision. This dimension reduction to $O(\log n)$ already captures the usage of shared randomness that we wanted to illustrate and it opens the road for computing the number of distinct elements via standard algorithm. Later, we will talk more about the details of how to actually solve this approximate distinct elements problem, but let us here focus more on the heart of the story, i.e., how to mimic the shared randomness via only local sharing

To keep the hash function collision free w.h.p., pairwise independence is enough which means sharing $O(\log n)$ bits of randomness is sufficient in this example. Now, this solution relies on that for each node v, all nodes within its h-neighborhood have picked the same hash function, i.e., the

same $\Theta(\log n)$ bits of hashing seed. Obviously we cannot have this for all nodes, unless one is willing to spend $O(D)$ rounds for achieving global shared randomness, but that might be undesirable.

Using the techniques in Lemma 4.2, we can carve $\Theta(\log n)$ layers of clustering, where each cluster has radius $\Theta(d \log n)$, and such that each node's h-neighborhood is fully contained in $\Theta(\log n)$ of such balls. This takes $\Theta(d \log^2 n)$ rounds. Then, we share $\Theta(\log n)$ bits of the randomness in each cluster picked by the center, similar to Lemma 4.3, in $\Theta(d \log n)$ rounds. In fact here, in this simple example where we want $\Theta(\log n)$ bits, this sharing can be done at the same time as we are carving the clusters. But in general, especially when $R = \omega(\log n)$, we would do it after the clustering, similar to Lemma 4.3, and it would take $O(d \log n + R)$ rounds. Finally, we can simulate the hash functions locally for each of the clusters. Hence, the hash function construction part takes $O(h \log^2 n + R)$ rounds.

Note that each node will be in $\Theta(\log n)$ clusters and thus will have $\Theta(\log n)$ hash functions, one for each cluster layer. Hence, given an algorithm $\mathcal{A}$ that uses the hash-functions to solve the problem, we still need to simulate $\mathcal{A}$ for each of these cluster layers, as we did in Lemma 4.4. But if $\mathcal{A}$ takes T rounds, we can similalry simulate $\mathcal{A}$ also in local clustering areas, each of diameter at most $O(T \log n)$, in total in $O(T \log^2 n)$ time.

Before going to the issue of reducing shared randomness, since we have started the discussion, let us actually finish the story of this particular problem of distinct elements: we explain how to approximate it to within $1+\varepsilon$, in $O(d \log n/\varepsilon^3)$ rounds, and just present a rough sketch: Consider a threshold $k = (1+\varepsilon)^j$. We can compare the number of distinct elements in each node's h-hop neighborhood with this threshold k as follows: Use a hash function $h'_1 = \{0,1\}^{\Theta(\log n)} \to \{0,1\}$ where for each $s \in \{0,1\}^{\Theta(\log n)}$, $\Pr[\forall s \in h'_1(s) = 1] = 1 - 2^{-1/k} \approx 1/k$, and these events are independent among different k. Then, in d rounds, each node v can know whether there is a node u in its d-hop neighborhood for which $h'_1(h(s_u)) = 1$. One can see that if the number of distinct elements in the d-hop neighborhood of v is at least $(1+\varepsilon/2)k$, then the probability that there is at least one node node u in its d-hop neighborhood for which $h'_1(h(s_u)) = 1$ is at least $0.5 + \Theta(\varepsilon)$. Furthermore, if the number is at most $k/(1+\varepsilon/2)$, then the probability is at most $0.5 - \Theta(\varepsilon)$. This $\Theta(\varepsilon)$ gap is the key distinguishing element. If we repeat this process for $\Theta(\log n/\varepsilon^2)$ iterations (different binary hash functions h'_i), Hoeffding's bound tells us that at the end, w.h.p, each node v knows whether the number of distinct elements in its d-neighborhood is above $(1+\varepsilon/2)k$ or below $k/(1+\varepsilon/2)$, just by seeing what the majority of the experiments say. Repeating this for all the $\Theta(\log n/\varepsilon)$ different thresholds (i.e., different values of $j \in \{1, \ldots, \Theta(\log n/\varepsilon)\}$), node v can know the number of distinct elements in its neighborhood to within $1+\varepsilon$ factor. In fact, each $\Theta(\log n)$ iterations can be bundled together as the CONGEST model admits $O(\log n)$ bit messages, now a bit-wise or of these message will be propagated. This gets us to the bound $O(d \log n/\varepsilon^3)$.

Reducing the Shared Randomness: Now we explain a distributed generalization of a classical observation of Newman [27] for two-party protocols, which shows that $O(\log n)$ bits of shared randomness is sufficient if in the problem, there are at most $2^{\text{poly}(n)}$ possibilities for the input given to each node, i.e., if each node's input can be described in $\text{poly}(n)$ bits. Note that almost all problems of interest in theoretical distributed computing fall within this category, e.g., the edges of a node, which are typically a key part of the input, can be described in at most $n \log n$ bits.

An algorithm with R bits of shared randomness is simply a collection $\mathcal{F}$ of 2^R deterministic algorithms. We know that for each fixed set of inputs, in $2/3$ or more of the algorithms, node v is outputting the same canonical output. We claim that we can find a smaller collection $\mathcal{F}'$ of these deterministic algorithms, with size $\text{poly}(n)$, such that for each set of fixed input, each node outputs the same canonical output with probability say at least $3/5$. The argument is by an application of the probabilistic method: simply pick $\text{poly}(n)$ many of the deterministic algorithms in $\mathcal{F}$ at

random and define the resulting collection to be a candidate for being collection $\mathcal{F}'$. Regarding one fixed set of inputs, using the Chernoff bound, we know that the probability that node v outputs the correct canonical value in at least $3/5$ of the algorithms in the candidate collection is at least $1-2^{-\Theta(\text{poly}(n))}$. Now, there are $2^{\text{poly}(n)}$ sets of possible inputs to all nodes, n nodes, and furthermore at most $2^{O(n^2)}$ possible graphs between the nodes. We can union bound over all of these possibilities and say that the probability that the candidate $\mathcal{F}'$ collection is good for all these choices is at least $1 - 2^{-\Theta(\text{poly}(n))} \times 2^{\text{poly}(n)} \geq 1 - 2^{-\Theta(\text{poly}(n))}$. That is, with very high probability, the candidate collection is good. Hence, in fact there exists such a good smaller collection $\mathcal{F}'$ with $|\mathcal{F}'| = \text{poly}(n)$. Now, note that to pick one of the algorithms in collection $\mathcal{F}'$ only takes $O(\log n)$ bits.

We note that this argument is simply existential in the sense that it proves that there exists a good collection $\mathcal{F}'$, and thus an algorithm which uses $O(\log n)$ bits of shared randomness. The argument does not provide a fast (centralized) method for finding this algorithm. However, if we ignore the local computations, which is a standard practice in distributed computing [25,30], nodes can deterministically search through the space of all collections, using a simple deterministic brute force, each running it on its own, and consistently find the first good collection $\mathcal{F}'$. Here, "first" is with respect to the deterministic search order.