SIAM J. Comput.
Vol. 47, No. 6, pp. 2078–2117
© 2018 Society for Industrial and Applied Mathematics

NEAR-OPTIMAL DISTRIBUTED MAXIMUM FLOW*

MOHSEN GHAFFARI[†], ANDREAS KARRENBAUER[‡], FABIAN KUHN[§], CHRISTOPH LENZEN[‡], AND BOAZ PATT-SHAMIR[¶]

Abstract. We present a near-optimal distributed algorithm for $(1 + o(1))$-approximation of single-commodity maximum flow in undirected weighted networks that runs in $(D + \sqrt{n}) \cdot n^{o(1)}$ communication rounds in the CONGEST model. Here, n and D denote the number of nodes and the network diameter, respectively. This is the first improvement over the trivial bound of $O(n^2)$, and it nearly matches the $\tilde{\Omega}(D + \sqrt{n})$-round complexity lower bound. The development of the algorithm entails two subresults of independent interest: (i) A $(D + \sqrt{n}) \cdot n^{o(1)}$-round distributed construction of a spanning tree of average stretch $n^{o(1)}$. (ii) A $(D + \sqrt{n}) \cdot n^{o(1)}$-round distributed construction of an $n^{o(1)}$-congestion approximator consisting of the cuts induced by $O(\log n)$ virtual trees. The distributed representation of the cut approximator allows for evaluation in $(D + \sqrt{n}) \cdot n^{o(1)}$ rounds. All our algorithms make use of randomization and succeed with high probability.

Key words. CONGEST model, congestion approximator, approximation algorithm, gradient descent

AMS subject classification. 86W

DOI. 10.1137/17M113277X

1. Introduction.

Computing a maximum flow is a fundamental task in network optimization. While the problem has a decades-old history rich with developments and improvements in the sequential setting, little is known in the distributed setting. In fact, prior to this work, the best known distributed time complexity in the standard CONGEST model remained at the trivial bound of $O(m)$, where m is the number of edges, i.e., the time to collect the entire topology and solve locally. This article improves this unsatisfying state to near-optimality.

THEOREM 1.1. *Given any $\varepsilon > 0$ and an undirected weighted graph G with distinguished vertices s and t, a $(1 + \varepsilon)$-approximate maximum s-t flow in G can be computed in the CONGEST model with high probability in $(D + \sqrt{n}) \cdot n^{o(1)}\varepsilon^{-3}$ rounds, where D and n are the diameter and the number of vertices in G, respectively.*

This round complexity almost matches the $\tilde{\Omega}(D + \sqrt{n})$ lower bound of Das Sarma et al. [13]. This bound extends to any nontrivial approximation, i.e., any $\varepsilon \in \operatorname{poly} n$. Note also that, w.l.o.g., $\varepsilon > 1/\sqrt{n}$, as otherwise the trivial algorithm collecting the entire input in $O(n^2)$ rounds and solving locally is faster than our approximation algorithm.

Before we proceed, let us formalize the model and the problem.

---

*Received by the editors July 5, 2017; accepted for publication (in revised form) September 13, 2018; published electronically November 27, 2018. A preliminary version of this article appeared in *Proceedings of the 34th Annual ACM Symposium on Principles of Distributed Computing* (PODC 2015).

http://www.siam.org/journals/sicomp/47-6/M113277.html

[†]Computer Science Department, ETH Zurich, Zurich 8092, Switzerland (ghaffari@inf.ethz.ch).

[‡]MPI for Informatics, Saarland Informatics Campus, Saarbrücken 66123, Germany (karrenba@mpi-inf.mpg.de, clenzen@mpi-inf.mpg.de).

[§]Department of Computer Science, University of Freiburg, 79085 Freiburg, Germany (kuhn@cs.uni-freiburg.de).

[¶]School of Electrical Engineering, Tel Aviv University, Tel Aviv 6997801, Israel (boaz@tau.ac.il).

1.1. Model and problem.

Computational model. We use the standard CONGEST model of synchronous computation [26]. We are given a simple, connected, weighted graph $G = (V, E, \text{cap})$, where $\text{cap} : E \to \mathbb{R}^+$ specifies the *capacity* of each edge. We follow the notational convention that $n = |V|$ and $m = |E|$. We shall assume henceforth that capacities are natural numbers of polynomial (in n) magnitude. This assumption is made without loss of generality, but possibly with some cost in complexity: See Appendix A for details. By D, we denote the (unweighted) diameter of G. Each node hosts a processor with a unique identifier of $O(\log n)$ bits, and $O(\log n)$ bits can be sent in each synchronous round of communication over each edge; we assume that nodes have access to infinite strings of independent unbiased random bits. We say that an event occurs *with high probability* (w.h.p.) if it happens with probability $1 - n^{-c}$ for any desired constant $c > 0$ specified upfront.[1] Initially, each node only knows its identifier, its incident edges, and their capacities. We fix an arbitrary orientation of the edges: for $\{u, v\} \in E$ we write $(u, v) \in E$ if the orientation is $u \to v$. This orientation is known to the nodes.

Problem statement. An instance of the *max-flow* problem is given by G and two distinguished nodes $s, t \in V$ called *source* and *sink*, respectively (in the distributed version, each node knows whether it is a source, sink, or neither). A (feasible) *flow* is a vector $\boldsymbol{f} \in \mathbb{R}^m$ satisfying

1. capacity constraints (edges): $\forall e \in E: \ |f_e| \leq \text{cap}(e)$;
2. conservation constraints (nodes): $\forall u \in V \setminus \{s, t\}: \ \sum_{(u,v)\in E} f_e - \sum_{(v,u)\in E} f_e = 0$; and
3. $\sum_{(s,u)\in E} f_e - \sum_{(u,s)\in E} f_e = -\sum_{(t,u)\in E} f_e + \sum_{(u,t)\in E} f_e$. We denote this common value by $F \in \mathbb{R}$ and call it the *value of the flow* $\boldsymbol{f}$.

A *max flow* is a flow of maximum value. For $\varepsilon > 0$, a $(1 + \varepsilon)$-*approximate max flow* is a flow whose value is at most a factor $1 + \varepsilon$ smaller than that of a max flow. In this work, we focus on solving the problem of finding a $(1 + \varepsilon)$-approximate max flow in the above model, where it suffices that each node u learns f_e for its incident edges $\{u, v\} \in E$.

1.2. Related work.

Network flow, being one of the canonical and most useful optimization problems, has been the target of innumerable research efforts since the 1930s (see, e.g., [30], the classic book [2], and the more recent survey [16]). For the general, directed case, the fastest known sequential algorithm is by Goldberg and Rao [15]; it solves the max-flow problem in time $\tilde{O}(\min\{m^{3/2}, mn^{2/3}\})$.[2] Particularly relevant from the point of view of the present paper are recent efforts to obtain fast algorithms to compute (approximate) max-flow solutions in the undirected case. Using the graph sparsification technique of Benczúr and Karger [10], any graph can be partitioned into $k = \tilde{O}(m\varepsilon^2/n)$ sparse graphs, each with $\tilde{O}(n/\varepsilon^2)$ edges, such that the max-flow problem can be approximately solved by combining max-flow solutions for each of these sparse graphs. Using the algorithm of Goldberg and Rao, this results in an algorithm with running time $\tilde{O}(mn^{1/2})$. In [12], Christiano et al. improved this running time to $\tilde{O}(mn^{1/3})$ by applying the nearly linear-time Laplacian solver of Spielman and Teng [33] to iteratively minimize a soft-max approximation of the edge congestions. Kelner et al. [17] and Sherman [32] independently published two

[1] Conversely, "low probability" means $n^{-\Omega(1)}$. By the union bound, the probability that any of polynomially many low-probability events occurs is still "low." We use this fact frequently and implicitly throughout the paper.

[2] We use the soft asymptotic notation like $\tilde{O}$ to hide factors of $(\log n)^{O(1)}$.

algorithms which allow computation of a $(1+\varepsilon)$-approximation to an undirected max flow problem in time almost linear in m. Finally, Peng [27] proved the first running time in $O(m\,\mathrm{polylog}(n))$.

However, to the dismay of many, and despite the fact that the word "network" even appears in the problem's name, only little progress was made over the years from the standpoint of distributed algorithms. For example, Goldberg and Tarjan's push-relabel algorithm, which is very local and simple to implement in the CONGEST model, requires $\Omega(n^2)$ rounds to converge, where n is the number of nodes. This is very disappointing, because in the CONGEST model, any problem whose input and output can be encoded with $O(\log n)$ bits per edge can be trivially solved in $O(m)$ rounds by collecting all input at a single node, solving it there, and distributing the results back.

Early attempts focused, as was customary in those days, on reducing the number of messages in asynchronous executions. For example, Segall [31] gives an $O(nm^2)$-messages, $O(n^2m)$-time algorithm for exact max flow, and Marberg and Gafni [23] give an algorithm whose message and time complexities are $O(n^2m^{1/2})$. Awerbuch has attacked the problem repeatedly with the following results. In an early work [5], he adapts Dinic's centralized algorithm using a synchronizer, giving rise to an algorithm whose time and message complexities are $O(n^3)$. With Leighton, in [8], he gives an algorithm for solving multicommodity flow approximately in $O(\ell m \log m)$ rounds, where $\ell < n$ is the length of the longest flow path. Later he considers the model where each flow path (variable) has an "agent" which can find the congestion of all links on its path in *constant time*. In this model, he shows with Khandekar [6] how to approximate any positive linear program (max flow with given routes included) to within $(1-\epsilon)$ in time polynomial in $\log(mnA_{\max}/\epsilon)$ (here n is the number of variables, which is at least the number of paths considered). The same model is used with Khandekar and Rao in [7], where they show how to approximate multicommodity flow to within $(1-\epsilon)$ in $O(\ell \log n)$ rounds. Using a straightforward implementation of this algorithm in the CONGEST model results in an $\tilde{O}(n^2)$-time algorithm.

Thus, up until the current paper, there has been no distributed implementation of a max-flow algorithm which always requires a subquadratic number of rounds. Even an $O(n)$-time algorithm would have been considered a significant improvement, even for the 0/1 capacity case.

1.3. Organization of this article. Our result builds heavily on a few major breakthroughs in the understanding of max flow in the centralized setting, most notably [32], as well as a few other contributions. Our presentation goes in a top-down fashion for the most part.

We start by carefully revisiting Sherman's approach [32] and the main building blocks he relies on in section 2. This sets the stage for shedding light on the challenges that must be overcome for its distributed implementation and presentation of our results in section 3. There, we also provide a top-level view of the components of the algorithm, alongside pointers to the detailed proofs showing that we can implement each of them by efficient distributed algorithms; these are given in sections 5–8. However, before delving into the technical details of our construction, we outline the distributed construction of an $n^{o(1)}$-*congestion approximator* in section 4. This is our key technical contribution; the role of a congestion approximator is to *estimate* the congestion induced by optimally routing an arbitrary demand vector very quickly, which lies at the heart of our algorithm.

2. Overview of the centralized framework. Sherman's approach [32] is based on *gradient descent* (see, e.g., [24]) for *congestion minimization* with a clever dualization of the flow conservation constraints. The flow problem is reformulated as a *demand vector* $\boldsymbol{b} \in \mathbb{R}^n$ such that $\sum_{i \in V} b_i = 0$. In the case of the *s*-*t* flow problem, we have a positive b_s and negative b_t with the same absolute value and the demand is zero everywhere else. The objective is to find a flow $\boldsymbol{f}^*$ that meets the given demand vector, i.e., the total excess flow in node i is equal to b_i, and minimizes the maximum *edge congestion*, which is the ratio of the flow over an edge to its capacity. Formally,

$$\text{minimize } \left\|C^{-1}\boldsymbol{f}\right\|_\infty \text{ subject to } B\boldsymbol{f} = \boldsymbol{b}\,, \tag{1}$$

where $C = (C_{ee'})_{e,e' \in E}$ is an $m \times m$ diagonal matrix with

$$C_{ee'} = \begin{cases} \text{cap}(e) & \text{if } e = e', \\ 0 & \text{else}\,, \end{cases}$$

and $B = (B_{ve})_{v \in V, e \in E}$ is an $n \times m$ matrix with

$$B_{ve} = \begin{cases} 1 & \text{if } e = (u,v) \text{ for some } u \in V, \\ -1 & \text{if } e = (v,u) \text{ for some } u \in V, \\ 0 & \text{else}\,. \end{cases}$$

In the following, we refer to the condition $B\boldsymbol{f} = \boldsymbol{b}$ as the *flow constraints*. Note that given a general vector $\boldsymbol{f} \in \mathbb{R}^m$, when viewed as a flow, $(B\boldsymbol{f})_v$ is exactly the excess flow at node v. If we can solve problem (1), we can find an approximate maximum flow using binary search as follows. Given a value F, set $b_s = F$, $b_t = -F$, $b_v = 0$ for all other nodes v, test whether $\left\|C^{-1}\boldsymbol{f}\right\|_\infty \leq 1$, and adjust the value of F accordingly.

Instead of directly solving this constrained system, Sherman allows for general flows and adds a penalty term for any violation of the flow constraints $B\boldsymbol{f} = \boldsymbol{b}$, i.e.,

$$\text{minimize } \left\|C^{-1}\boldsymbol{f}\right\|_\infty + 2\alpha \left\|R(\boldsymbol{b} - B\boldsymbol{f})\right\|_\infty\,,$$

where $\alpha \geq 1$ and the matrix $R \in \mathbb{R}^{x \times n}$ for some $x \in \mathbb{N}$ are chosen so that the optimum of this unconstrained optimization problem does not violate the flow constraints. As we are interested in an approximate maximum flow, we can compute an approximate solution and argue that the violation of the flow constraints will be small, too. We thus get a flow that almost satisfies the demand vector $\boldsymbol{b}$. To obtain a flow that satisfies the flow constraints, we have to superpose the existing flow with a flow that satisfies the demand vector $\boldsymbol{b} - B\boldsymbol{f}$. One can find this remaining flow in a trivial manner, e.g., on a spanning tree, to obtain a near-optimal solution. Finally, to ensure that the objective function is differentiable (i.e., a gradient descent is actually possible), $\|\cdot\|_\infty$ is replaced by the so-called soft-max function (a differentiable approximation of the maximum; cf. (2)).

The congestion approximator R. The *congestion* of an edge e (for a given flow $\boldsymbol{f}$) is defined as the ratio $|f_e|/\text{cap}(e)$. When referring to the congestion of a cut in a given flow, we mean the ratio between the net flow crossing the cut (in a given direction) to the total capacity of the cut. Suppose for a moment that $\alpha = 1$ and R contains one row for each of the $2^{n-1} - 1$ cuts of the graph, chosen such that each entry of the vector $RB\boldsymbol{f}$ equals the congestion of the corresponding cut. In particular, R would correctly reproduce the congestion of min cuts (which give rise to maximal congestion). Moreover, the vector $R\boldsymbol{b}$ describes the inevitable congestion of the cuts for any feasible flow. Thus, the components of $R(\boldsymbol{b} - B\boldsymbol{f})$ are the residual congestions

to be dealt with to make $\boldsymbol{f}$ feasible (neglecting possible cancellations within $\boldsymbol{f}$). The max-flow min-cut theorem and the factor of 2 in the second term of the objective function imply that routing the demands arising from a violation of flow constraints optimally always improves the value of the objective function. Moreover, gradient descent focuses on the most congested edges and those that are contained in cuts with the top residual congestion. In particular, flow is pushed over the edges into the cut with the highest residual congestion to satisfy its demand until other cuts become more important in the second part of the objective, i.e., determine the second infinity-norm. The first part of the objective impedes flow on edges the more they are congested (on an absolute scale and relative to others). Thus, approximately minimizing the objective function is equivalent to simultaneously approximating the minimum congestion and having a small violation of the flow constraints; solving up to polynomially small error and naively resolving the remaining violations then yields sufficiently accurate results.

Unfortunately, trying to make R capture congestion *exactly* is far too inefficient. Instead, one uses an *α-congestion approximator*, that is, a matrix R such that for any demand vector $\boldsymbol{b}$, it holds that

$$\|R\boldsymbol{b}\|_\infty \leq \mathrm{opt}(\boldsymbol{b}) \leq \alpha \|R\boldsymbol{b}\|_\infty ,$$

where $\mathrm{opt}(\boldsymbol{b})$ is the maximum congestion caused on any cut by optimally routing $\boldsymbol{b}$. Since the second term in the objective function is scaled up by factor α, we are still guaranteed that optimally routing any excess demands improves the objective function. However, this implies that the gradient of the objective function may be dominated by the second term, and thus emphasis is shifted to feasibility rather than optimality. Sherman proves that this slows down the gradient descent by at most a factor of α^2, and hence, if $\alpha \in n^{o(1)}$, then the number of gradient descent iterations needs to be increased only by an $n^{o(1)}$ factor.

Congestion approximators: Räcke's construction. For any spanning tree T of G, deleting an edge partitions the nodes into two connected components and thus induces an (edge) cut of G. Note that in T, this cut contains only the single deleted edge, and that in terms of congestion, any cut of T is dominated by such an edge-induced cut: For any cut, the maximum congestion of an edge is at least the average congestion of the cut, and in T, there is a cut containing only this edge.

These basic properties motivate the question of how well the cut structure of an arbitrary graph can be approximated by trees. Intuitively, the goal is to find a tree T (not necessarily a subgraph) spanning all nodes with edge weights such that routing *any* demand vector in G and in T results in roughly the same maximal congestion. Because routing flows on trees is trivial, such a tree T would give rise to an efficient congestion approximator R: R would consist of one row for each cut induced by an edge (u, v) of T with capacity C, where the matrix entry corresponding to node w is $1/C$ if w is on u's "side" of the cut and 0 otherwise; multiplying a demand vector with the row then yields the flow that needs to pass through (u, v) divided by the capacity of the cut.

In a surprising result [29], Räcke showed that, using multiplicative weight updates (see, e.g., [4, 28, 34]), one can construct a distribution of $\tilde{O}(m)$ trees so that (i) in each tree of the distribution, each cut has at least the same capacity as in G, and (ii) given any cut of G of total capacity C, sampling from the distribution results in a tree T where this cut has *expected* capacity $O(\alpha C)$, where α here is the approximation ratio of a *low average stretch spanning tree* algorithm, used as a subroutine in Räcke's construction. Note that this bound on the expectation implies that for any cut of

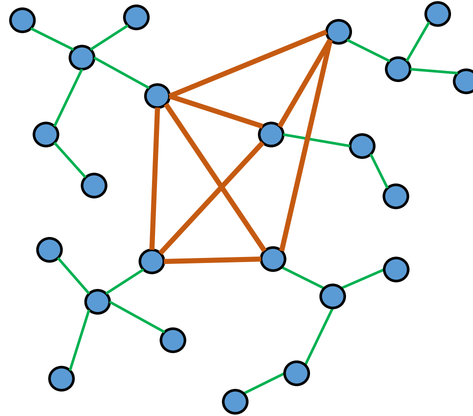


FIG. 1. *A j-tree for $j = 5$. The core links are drawn thicker.*

capacity C, there must be a tree in the distribution for which the cut has capacity $O(\alpha C)$. Hence, the cuts given by *all* trees in the distribution give rise to an $O(\alpha)$-congestion approximator R with $\tilde{O}(mn)$ rows.

Low average stretch spanning trees. To implement Räcke's construction, an efficient algorithm for computing low average stretch spanning trees is required. More precisely, given a graph $G = (V, E, \ell)$ with polynomially bounded lengths $\ell : E \to \mathbb{N}$, the goal is to construct a spanning tree T of G so that

$$\sum_{\{u,v\}\in E} d_T(u, v) \leq \alpha \sum_{\{u,v\}\in E} \ell(\{u, v\}) ,$$

where $d_T(u, v)$ is the sum of the lengths of the unique path from u to v in T and α is the *stretch* factor.

Sherman's algorithm builds on a sophisticated low average stretch spanning tree algorithm that achieves $\alpha \in O(\log n(\log\log n)^2)$ within $\tilde{O}(m)$ centralized steps [1]. We use a simpler approach providing $\alpha \in 2^{O(\sqrt{\log n \log\log n})}$ [3] that has been shown to parallelize well, i.e., has an efficient implementation in the PRAM model [11].

Congestion approximators: Madry's construction. Räcke's construction has the drawback that one needs to sequentially compute a linear number of trees, which is prohibitively expensive from the distributed computation point of view. Madry [21] generalized Räcke's approach to a construction that produces a distribution over $\tilde{O}(m/j)$ so-called *j-trees*, where j is a parameter. A j-tree consists of a *forest* of j connected components (trees) and a *core graph*, which is an arbitrary connected graph with j nodes: one from each tree (see Figure 1).

The properties of the distribution are the same as for Räcke's: sampling from the distribution preserves cut capacities up to an $O(\alpha)$-factor in expectation, where α is the average stretch of a utilized spanning tree algorithm. Given a j-tree, we say that a cut is *dominant* if it is induced either by a single edge of the forest or by a cut of the core. All dominant cuts of all j-trees in the distribution to construct R yield an $O(\alpha)$-congestion approximator. Note that any cut in a j-tree is *dominated* by some dominant cut in the following sense: Consider any demand vector and any "mixed" cut that either separates the endpoints of more than one of the tree edges or separates the endpoints of both tree and core edges. If there is an edge in the forest crossing the

cut that has at least the same congestion as the whole cut, then the cut induced by the forest edge dominates the mixed cut. Otherwise, we can remove all forest edges from the mixed cut without reducing its congestion. As routing demands in the forest part of the graph is trivial, Madry's construction can be seen as an efficient reduction of the problem size.

Congestion approximators: Combining cut sparsifiers with Madry's construction. Using j-trees, Sherman derives a suitable congestion approximator, i.e., one with $\alpha \in n^{o(1)}$ that can be constructed and evaluated in $\tilde{O}(m+n^{1+o(1)})$ rounds, as follows. First, a *cut sparsifier* is applied to G. A $(1+\varepsilon)$-sparsifier computes a subgraph of G with modified edge weights so that the capacities of all cuts are preserved up to factor $1+\varepsilon$. It is known how to compute a $(1+o(1))$-sparsifier with $\tilde{O}(n)$ edges in $\tilde{O}(m)$ steps using randomization [10]. As the goal is merely to compute a congestion approximator with $\alpha \in n^{o(1)}$, the multiplicative $1+o(1)$ approximation error is negligible. Hence, this essentially breaks the problem of computing a congestion approximator down to the same problem on sparse graphs.

Next, Sherman applies Madry's construction with $j = n/\beta$, where $\beta = 2^{\sqrt{\log n}}$. This yields a distribution of $\tilde{O}(\beta)$ many n/β-trees. The issue is now that the cores are arbitrary graphs, implying that it may be difficult to evaluate congestion for cuts in the cores. However, the number of nodes in the core is $n' = n/\beta$. Thus, recursion does the trick: apply the cut sparsifier to the core, use Madry's construction on the resulting graph (with $j' = n'/\beta = n/\beta^2$), rinse and repeat. In total, there are $\log_\beta n = \sqrt{\log n}$ levels of recursion until the core becomes trivial, i.e., we arrive at a tree. For each level of recursion, the approximation ratio deteriorates by a multiplicative $\alpha \in \operatorname{polylog} n$, where α is the stretch factor of the low-stretch spanning tree algorithm, and a multiplicative $1+o(1)$, for applying the cut sparsifier. This yields an α'-congestion approximator with

$$\alpha' \in ((1+o(1))\alpha)^{\sqrt{\log n}} \subset 2^{O(\sqrt{\log n}\log\log n)} \subset n^{o(1)}\,.$$

While the total number of constructed trees is still $\tilde{O}(\beta^{\log_\beta n}) = \tilde{O}(n)$, the number of nodes in a graph (i.e., a core from the previous level) on the ith level of recursion is only n/β^{i-1}. The cut sparsifier ensures that the number of edges in this graph is reduced to $\tilde{O}(n/\beta^{i-1})$ *before* recursing. Since the number of edges in the core is (trivially) bounded by the number of edges of the graph in Madry's construction, the total number of sequential computation steps for computing the distribution is thus bounded by

$$\tilde{O}(m) + \sum_{i=1}^{\log_\beta n} \tilde{O}(\beta^i \cdot n/\beta^{i-1}) \subset \tilde{O}(m+n^{1+o(1)})\,.$$

Step complexity of the flow algorithm. The above recursive structure can also be exploited to *evaluate* the α'-congestion approximator Sherman uses in $n^{1+o(1)}$ steps. As mentioned earlier, the cuts of a j-tree are dominated by those induced by edges of the forest and those which are crossed by core edges only (cf. Figure 1). In the forest component, routing demand is unique, takes linear time in the number of nodes (simply start at the leaves), and results in a modified demand vector at the core on which it is recursed. Sherman shows that the algorithm obtains a $(1+\varepsilon)$-approximate flow in $O(\varepsilon^{-3}\alpha^2\log^2 n)$ gradient descent steps, provided R is an α-congestion approximator. It is straightforward to see (cf. section 3.2) that each of these steps requires $O(m)$ computational steps besides doing two matrix-vector multiplications with R and $R^\top$, respectively. Using the above observation and plugging in the time to construct the (implicit) representation of R, one arrives at a total step complexity of $\tilde{O}(mn^{o(1)})$.

3. Distributed algorithm.

3.1. The distributed toolchain. For a distributed implementation of Sherman's approach, many subproblems need to be solved sufficiently quickly in the CONGEST model. We summarize them in the following list, in which stars indicate readily available components.

* Decomposing trees into $O(\sqrt{n})$ components of strong diameter $O(\sqrt{n})$, within $\tilde{O}(\sqrt{n}+D)$ rounds. This can, e.g., be done by techniques pioneered by Kutten and Peleg for the purpose of minimum-weight spanning tree construction [20].
* Constructing cut sparsifiers. Koutis [18] provides a solution that runs in $\operatorname{polylog} n$ rounds of the CONGEST model. In section 7, we specify an emulation strategy that allows us to use it in our recursive construction.

1. Constructing low average stretch spanning trees on multigraphs (section 8).
2. Applying Madry's construction in the CONGEST model, even when recursing in the context of Sherman's framework (section 6).
3. Sampling from the recursively constructed distribution of spanning trees (section 6).
4. Avoiding the use of the entire distribution for constructing the congestion approximator (see below).
5. Performing a gradient descent step. This involves matrix-vector multiplications with R, $R^\top$, and C^{-1}, evaluation of the soft-max, etc. (section 3.2).

Let us elaborate a little on the distributed implementation of tasks 1–4 from the list above.

3.1.1. Low average stretch spanning trees. In section 8, we prove the theorem given below. But first, let us formalize the intuitive notion of edge contraction.

DEFINITION 3.1. Edge contraction *is an operation that takes a graph $G = (V, E)$ and an edge $(u, v) \in E$ and produces a graph $G' = (V', E')$ defined by*

- $V' = V \setminus \{v, u\} \cup \{\breve{w}\}$*, where $\breve{w}$ is a fresh node (i.e., $w \notin V$), and*
- $E' = \dot{\bigcup}_{\{x,y\} \in E \setminus \{\{v,u\}\}} \{\{f(x), f(y)\}\}$*, where*

$$f(x) = \begin{cases} \breve{w} & \text{if } x \in \{u, v\}, \\ x & \text{otherwise.} \end{cases}$$

Note that this means that the resulting set of edges is a multiset and the resulting graph is a multigraph.

THEOREM 3.2. *Suppose H is a multigraph obtained from G by assigning arbitrary edge lengths in $[1, 2^{n^{o(1)}}]$ to the edges of G (known to incident nodes) and performing an arbitrary sequence of contractions. Then we can compute a spanning tree of H of expected stretch $2^{O(\sqrt{\log n \log \log n})}$ within $(\sqrt{n} + D)n^{o(1)}$ rounds.*

To obtain this result, we adapt a PRAM algorithm by Blelloch et al. [11] to the CONGEST model. The main issue in the adaptation is that in the PRAM model, any piece of information can be accessed instantaneously by any processor, while in the CONGEST model, access takes time due to distance and congestion in the underlying network. The standard solution to this issue is using pipelining over a depth-$O(D)$ spanning tree of G (dubbed "global breadth-first search (BFS) tree"); communication over $O(\sqrt{n})$ hops of already computed parts of the spanning tree is handled using the edges that have already been selected for inclusion in the spanning tree and the spanning trees of the contracted regions of G.

3.1.2. Implementing Madry's scheme. This is technically the most challenging part. Also here, we have to overcome the potential difficulty of how to communicate a large amount of information over many hops; doing this naively results in excessive contention, which slows the algorithm down. We therefore modify Madry's construction as follows.

- Instead of "aggregating" edges so that the core becomes a graph, we allow the core to be a multigraph.
- Instead of explicitly constructing the core, we simulate both the sparsifier and the low average stretch spanning tree algorithms using the abstraction of *cluster graphs* (see section 5).
- In the simulation, we maintain the invariant that every core edge is also a graph edge, allowing us to use it for communication.
- Clusters (of the cluster graph) are the forest components rooted at core nodes. We maintain the invariant that forest components have depth $\tilde{O}(\sqrt{n})$. While not strictly necessary, this simplifies the description of the corresponding distributed algorithms, as the communication within each cluster can then be performed via its (previously constructed) spanning tree.
- The cluster hierarchy, as established during the construction, allows for a straightforward recursive evaluation of the corresponding congestion approximator.

Section 6 gives the details of the construction.

3.1.3. Sampling from the distribution. This task is actually straightforward in our framework, because for each sample, on each level of the recursion we need to construct only $n^{o(1)}$ different j-trees for some j. More specifically, we show that if the number of j-trees in the distribution constructed when recursing on a core is $\tilde{O}(\beta)$, then one can sample a virtual tree from the distribution used in Sherman's framework within $\tilde{O}((\sqrt{n}+D)\beta)$ rounds. Moreover, the distributed representation allows us to evaluate the dominant cuts of the tree when using it in a congestion approximator within $\tilde{O}(\sqrt{n}+D)$ rounds. See Theorem 6.13 for details.

3.1.4. Avoiding the use of the entire distribution for constructing the congestion approximator. While sequentially, one can afford to use all trees in the (recursively constructed) distribution, the result mentioned above is not sufficient to allow for fast evaluation of all $\tilde{\Theta}(n)$ trees. As Madry points out [21], it suffices to sample and use $O(\log n)$ j-trees from the distribution he constructs to speed up any β-approximation algorithm for an "undirected cut-based minimization problem," at the cost of an increased approximation ratio of $2\alpha\beta$, where α is the approximation ratio of the congestion approximator corresponding to the distribution of j-trees. The reasoning is as follows:

- The number of cuts that need to be considered for such a problem is polynomially bounded.
- The expected approximation ratio for any fixed cut when sampling from the distribution is α. By Markov's inequality, with probability at least $1/2$ it is at most 2α.
- For $O(\log n)$ samples, the union bound shows that w.h.p. *all* relevant cuts are 2α-approximated.
- Applying a β-approximation algorithm relying on the samples only, which can be evaluated much faster, results in a $2\alpha\beta$-approximation w.h.p.

Recall that the problem of approximating the maximum flow was reduced to minimizing congestion for demands F at s and $-F$ at t, and performing binary search

over F. The max-flow min-cut theorem implies the respective congestion to be the function of a single cut, which can be used to verify that the problem falls under Madry's definition.

Applying Madry's sampling strategy directly will yield an approximation which is too loose for our $(1+\varepsilon)$ goal. Alternatively, we can apply it in the construction of a congestion approximator. However, a congestion approximator must return a good approximation for *any* demand vector. There are exponentially many such vectors even if we restrict the demand in each node to be in $\{-1,0,1\}$, and we are not aware of any result showing that the number of min cuts corresponding to the respective optimal flows is polynomially bounded.

We resolve this issue with the following simple, but essential, insight, at the cost of squaring the approximation ratio of the resulting congestion approximator.

LEMMA 3.3. *Suppose we are given a distribution $\mathcal{D}$ over $K \in n^{O(1)}$ trees so that given any cut of G of capacity C, sampling from $\mathcal{D}$ results in a tree whose corresponding cut has capacity at least C always, and at most αC in expectation. Then sampling $O(\log n)$ trees from $\mathcal{D}$ and constructing a congestion approximator from their single-edge induced cuts results, w.h.p., in a $2\alpha^2$-congestion approximator of G.*

Proof. Recall that an approximator estimates the maximum congestion when optimally routing an arbitrary demand. Consider any demand vector $\boldsymbol{b}$ and let C denote the capacity of the corresponding cut that is most congested when routing $\boldsymbol{b}$. As sampling from $\mathcal{D}$ yields approximation factor αC in expectation, there must be *some* tree T in the support of $\mathcal{D}$ whose corresponding cut has capacity at most αC. However, this means that when routing $\boldsymbol{b}$ over T, there is some edge $e \in T$ that experiences congestion at least $1/\alpha$ times the maximum when routing the demand optimally in G. As $\text{cap}(e) \geq C$, it follows that the corresponding cut of G has congestion at least $1/\alpha$ of the congestion of e when routing the demand $\boldsymbol{b}$.

As there are $K \in n^{O(1)}$ trees in the support of $\mathcal{D}$, each of which has $n-1$ edges, this shows that there is a set of $K(n-1) \in n^{O(1)}$ cuts of G, such that for *any* demand vector, one of these cuts has congestion at least $1/\alpha$ times the optimal maximum congestion for that demand vector. By Markov's inequality, when sampling from the distribution, for each such cut, with probability at least $1/2$, the sampled tree approximates the congestion up to factor 2α. Accordingly, when taking $O(\log n)$ samples, w.h.p. this approximation guarantee is achieved for each such cut. By the union bound, we thus obtain the desired $2\alpha^2$-congestion approximator of G. ◻

3.2. The high-level algorithm. As mentioned already, our algorithm, presented in Algorithm 1, is a distributed implementation of Sherman's algorithm [32]. It calls Algorithm AlmostRoute (described shortly) a logarithmic number of times, followed by computing a maximum-weight spanning tree and routing the leftover demand through this tree.

Before we explain the implementation AlmostRoute (which is the heart of the algorithm), let us first quickly outline how to implement the endgame of Steps 5–6.

LEMMA 3.4. *Steps* 5–6 *of Algorithm* 1 *can be implemented in the* CONGEST *model in $\tilde{O}(D+\sqrt{n})$ rounds w.h.p.*

Proof. A maximum weight spanning tree T can be computed in $\tilde{O}(D+\sqrt{n})$ rounds using the minimum weight spanning tree algorithm of Kutten and Peleg [20] (say, by assigning weight $w(e) := -\text{cap}(e)$ for each edge e). To compute the flow, we use the following observation: if T were rooted at one of its nodes, then to route the demand $\boldsymbol{b}_t$ over T, it would be sufficient for each node v to learn the total demand d_v in the

Algorithm 1. Max flow. Input: demand vector $\boldsymbol{b} \in \mathbb{R}^n$; output: flow vector $\boldsymbol{f} \in \mathbb{R}^m$.

1: $\boldsymbol{b}_0 \leftarrow \boldsymbol{b}$; $\boldsymbol{f}_0 \leftarrow \boldsymbol{0}$
2: **for** $i \leftarrow 1$ to $(\log m + 1)$ **do**
3: $\quad \boldsymbol{f}_i \leftarrow \mathsf{AlmostRoute}(\boldsymbol{b}_i, \frac{1}{2})$
4: $\quad \boldsymbol{b}_i \leftarrow \boldsymbol{b}_{i-1} - B\boldsymbol{f}_{i-1}$.
5: Compute a maximum weight spanning tree T on G, where weights are the capacities of edges.
6: Route the residual demand $\boldsymbol{b}_t$ through T; let $\boldsymbol{f}_T$ be the resulting flow.
7: Output $\boldsymbol{f}_T + \sum_{i=1}^{1+\log m} \boldsymbol{f}_i$.

subtree rooted at v. In this case each node v assigns d_v units of flow to the edge leading from v to its parent.

We now show how to root the tree and find the total demand in each subtree in $\tilde{O}(D + \sqrt{n})$ rounds. The algorithm is as follows. Remove each edge of the tree independently with probability $1/\sqrt{n}$. With high probability,

(i) each connected component induced by the remaining edges has strong diameter $\tilde{O}(\sqrt{n})$,
(ii) $O(\sqrt{n})$ edges are removed, and hence
(iii) the number of components is $O(\sqrt{n})$.

Within each component, all demands are summed up, and this sum is made known to all nodes. The summation takes $\tilde{O}(\sqrt{n})$ rounds due to (i), and we can pipeline the announcement of the sums over the global BFS tree in $\tilde{O}(\sqrt{n} + D)$ rounds due to (iii).

Moreover, within the same asymptotic time we assign unique identifiers to the components (e.g., the minimum identifier) and broadcast the "cluster tree" (which results from contracting components) to all. Using local computation only, nodes then can root this tree (e.g., at the cluster of minimum identifier) and determine the sum the demands of the clusters that are fully contained in their subtree. Using a simple broadcast, the orientation of edges within components is determined, and using a convergecast on the components, each node can determine the sum of demands in its subtree. These steps take another $\tilde{O}(\sqrt{n})$ rounds. ◻

Algorithm AlmostRoute*: The gradient descent.* We now explain how to implement Algorithm AlmostRoute in a distributed setting. The idea is to use gradient descent with the potential function

$$\phi(\boldsymbol{f}) = \operatorname{smax}(C^{-1}\boldsymbol{f}) + \operatorname{smax}(2\alpha R(\boldsymbol{b} - B\boldsymbol{f})) \ ,$$

where the "soft-max" function, defined by

$$\operatorname{smax}(\boldsymbol{y}) = \log\left(\sum_{i=1}^{k} e^{y_i} + e^{-y_i}\right) \quad \text{for all } \boldsymbol{y} \in \mathbb{R}^k \ , \tag{2}$$

is used as a differentiable approximation to the "max" function.[3]

Given this potential function, AlmostRoute performs $O(\alpha^2\varepsilon^{-3}\log n)$ updates on $\boldsymbol{f}$ and outputs a flow $\boldsymbol{f}$ that optimizes the potential function up to a $(1+\varepsilon)$ factor. Pseudocode for this centralized algorithm is given in Algorithm 2.

To implement this algorithm in a distributed setting, we need to compute R, and to perform multiplications by R or its transpose $R^\top$, distributively. These multiplications are needed for evaluating $\phi(\boldsymbol{f})$ and its partial derivatives (distributed

[3]Throughout this paper, we consider natural logarithms whenever the base is relevant.

Algorithm 2. AlmostRoute($\boldsymbol{b}, \varepsilon$).

1: $k_b \leftarrow 2\alpha \|R\boldsymbol{b}\|_\infty \varepsilon/(16 \log n)$; $\boldsymbol{b} \leftarrow k_b \boldsymbol{b}$.
2: **repeat**
3: $\quad k_f \leftarrow 1$
4: $\quad$ **while** $\phi(\boldsymbol{f}) < 16\varepsilon^{-1} \log n$ **do**
5: $\quad\quad \boldsymbol{f} \leftarrow \boldsymbol{f} \cdot (17/16)$; $\boldsymbol{b} \leftarrow \boldsymbol{b} \cdot (17/16)$; $k_f \leftarrow k_f \cdot (17/16)$
6: $\quad \delta \leftarrow \sum_{e \in E} |\mathrm{cap}(e) \frac{\partial \phi}{\partial f_e}|$
7: $\quad$ **if** $\delta \geq \varepsilon/4$ **then**
8: $\quad\quad f_e \leftarrow f_e - \mathrm{sgn}\left(\frac{\partial \phi}{\partial f_e}\right) \cdot \mathrm{cap}(e) \frac{\delta}{1+4\alpha^2}$ for all edges $e \in E$
9: $\quad$ **else**
10: $\quad\quad f_e \leftarrow f_e / k_f$ for all edges $e \in E$
11: $\quad\quad b_v \leftarrow b_v/(k_b k_f)$ for all nodes $v \in V$
12: $\quad\quad$ **return**
13: **until** done

multiplications by B are easy). We remark that R and $R^\top$ are not constructed explicitly, as we need to ensure a small time complexity for each iteration. Assuming that we can perform these operations, each step of AlmostRoute can be completed in $\tilde{O}(D)$ additional rounds, required to compute and broadcast global quantities such as δ.

We maintain the invariant that at the beginning of each iteration of the **repeat** loop, each node v knows the current flow over each of the links incident to v, and the current demand at v (i.e., $(\boldsymbol{b} - B\boldsymbol{f})_v$).

We proceed as follows. We break the potential function ϕ in two, i.e.,

$$\begin{aligned} \phi(\boldsymbol{f}) &= \phi_1(\boldsymbol{f}) + \phi_2(\boldsymbol{f}) \ , \text{ where} \\ \phi_1(\boldsymbol{f}) &= \mathrm{smax}(C^{-1}\boldsymbol{f}) \ , \text{ and} \\ \phi_2(\boldsymbol{f}) &= \mathrm{smax}(2\alpha R(\boldsymbol{b} - B\boldsymbol{f})) \ . \end{aligned}$$

First, we compute $\phi_1(\boldsymbol{f})$: to find $\mathrm{smax}(C^{-1}\boldsymbol{f})$, it is sufficient to sum $\exp(f_e/\mathrm{cap}(e))$ and $\exp(-f_e/\mathrm{cap}(e))$ over all edges e, which can be done in $O(D)$ rounds. As Sherman points out, $\phi(\boldsymbol{f}) = \Theta(\varepsilon^{-1} \log n)$ due to the scaling in line 5 of Algorithm 2, and thus, any desired relative accuracy that is polynomial in $\frac{\varepsilon}{n}$ can be achieved with messages of $O(\log n)$ bits. To avoid messages that contain numbers in the order of $\exp(\phi(\boldsymbol{f}))$, we use a well-known trick for numerical stable computation of $\mathrm{smax}(y)$. That is, we first aggregate $||y||_\infty$, broadcast the result, and then exploit that

$$\mathrm{smax}(y) = ||y||_\infty + \log\left(\sum_{i=1}^{k} \underbrace{\left[1 + e^{-2|y_i|}\right]}_{\leq 2} \cdot \underbrace{e^{|y_i| - ||y||_\infty}}_{\leq 1}\right) .$$

Hence, we only have to accumulate values in the range $[0, 2]$, which we can again do with a granularity that is polynomial in $\frac{\varepsilon}{n}$ using messages of $O(\log n)$ bits. The error introduced by rounding these values is small enough to not affect the asymptotics of the running time.

Define

$$\boldsymbol{y} = 2\alpha R(\boldsymbol{b} - B\boldsymbol{f}).$$

To determine $\phi_2(\boldsymbol{f})$, we compute the vector $\boldsymbol{y}$ and then do an aggregation on a BFS tree as for $\phi_1(\boldsymbol{f})$. Since $\boldsymbol{b} - B\boldsymbol{f}$ can be computed instantly (because $(B\boldsymbol{f})_v$ is exactly

the net flow into v), this boils down to multiplying a locally known vector with R. To explain how to implement this operation, we first discuss the structure of R and how we determine $\frac{\partial \phi}{\partial f_e}$, as required in lines 6 and 8 of the algorithm.

The linear operator R is induced by graph cuts. More precisely, in the matrix representation of R, there is one row for each cut our congestion approximator (explicitly) considers. We will clarify the structure of R shortly; for now, let I denote the set of row indices of R. Observe that

$$\frac{\partial \phi(\boldsymbol{f})}{\partial f_e} = \frac{\exp(f_e/\mathrm{cap}(e)) - \exp(-f_e/\mathrm{cap}(e))}{\mathrm{cap}(e)\exp(\phi_1(\boldsymbol{f}))} + \frac{\partial \phi_2(\boldsymbol{f})}{\partial f_e}, \tag{3}$$

and hence, given that ϕ_1 is known, the first term is locally computable. The second term expands to

$$\frac{\partial \phi_2(\boldsymbol{f})}{\partial f_e} = \sum_{i\in I} \frac{\partial \phi_2(\boldsymbol{f})}{\partial y_i} \cdot \frac{\partial y_i}{\partial f_e} = \sum_{i \in I} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(\boldsymbol{f}))} \cdot \frac{2\alpha B_{i,e}}{\mathrm{cap}(i)},$$

where $\mathrm{cap}(i)$ is the capacity of cut i in the congestion approximator and $B_{i,e} \in \{-1, 0, 1\}$ depends on whether e is outgoing (-1), incoming (1), or not crossing cut i.[4]

The cuts $i \in I$ are induced by the edges of a collection of (rooted, virtual, capacitated) spanning trees $\mathbb{T}$ that we construct. The cuts are induced as follows. For $\mathcal{T} \in \mathbb{T}$, we write $(v, \hat{v}) \in \mathcal{T}$ if $\hat{v}$ is the parent of v and denote by $\mathcal{T}_v$ the subtree rooted at v. Given $\mathcal{T} \in \mathbb{T}$, each edge $(v, \hat{v}) \in \mathcal{T}$ induces a (directed) cut $(T_v; \overline{T_v})$ with index $i(\mathcal{T}, (v, \hat{v}))$. We denote the set of edges crossing this cut by $\delta(\mathcal{T}_v)$. Let us also define

$$p(\mathcal{T}, (v, \hat{v})) = \frac{\exp(y_{i(\mathcal{T},(v,\hat{v}))}) - \exp(-y_{i(\mathcal{T},(v,\hat{v}))})}{\exp(\phi_2)} \cdot \frac{2\alpha}{\mathrm{cap}_{\mathcal{T}}((v, \hat{v}))}.$$

With this notation, we have that

$$\frac{\partial \phi_2}{\partial f_e} = \sum_{\mathcal{T}\in\mathbb{T}} \sum_{(v,\hat{v})\in\mathcal{T}} \sum_{e\in\delta(\mathcal{T}_v)} p(\mathcal{T}, (v, \hat{v})) \cdot B_{i(\mathcal{T},(v,\hat{v})),e}.$$

We call $p(\mathcal{T}, (v, \hat{v}))$ the *price* of the (virtual) edge $(v, \hat{v}) \in \mathcal{T}$. Let $\mathcal{P}_{v,\mathcal{T}}$ denote the edge set of the unique path in $\mathcal{T}$ from v to the root of $\mathcal{T}$. We define a *node potential* for each node v by

$$\pi_v := \sum_{\mathcal{T}\in\mathbb{T}} \sum_{(w,\hat{w})\in\mathcal{P}_{v,\mathcal{T}}} p(\mathcal{T}, (w, \hat{w})).$$

For any $e = (u, v)$, the cuts induced by edges in $\mathcal{T} \in \mathbb{T}$ that e crosses correspond to the edges on the unique path from u to v in $\mathcal{T}$. For all edges $(w, \hat{w}) \in \mathcal{T}$ on the path from u to the least common ancestor of u and v in $\mathcal{T}$, $B_{i(\mathcal{T},(w,\hat{w})),e} = -1$, while $B_{i(\mathcal{T},(w,\hat{w})),e} = +1$ for the edges on the path between v and this least common ancestor. Thus, for any $e = (u, v)$ we have

$$\frac{\partial \phi_2}{\partial f_e} = \pi_v - \pi_u, \tag{4}$$

and our task boils down to determining the value of the potential π_v at each node $v \in V$. To this end, we need two subroutines to distributively compute the following key quantities.

[4] Technically, $B_{i,e} = \sum_{v\in S_i} B_{ve}$ where S_i is the set of nodes defining cut i.

- For each cut i: y_i. As mentioned above, $\boldsymbol{b} - B\boldsymbol{f}$ is known distributively; i.e., each node knows its own coordinate of this vector. For each tree in $\mathcal{T} \in \mathbb{T}$, we need to aggregate this information from the leaves to the root. This means to simulate a convergecast on the virtual tree $\mathcal{T}$.
- For each node v: π_v. Provided that each (virtual) tree edge knows its corresponding y-value and ϕ_2, the prices can be computed locally. Then the contribution of each tree to the node potentials can be computed by a downcast from the corresponding root to its leaves.

With these routines, one iteration of the **repeat** loop is now executed as follows:

1. Compute ϕ_1, $\boldsymbol{y}$ (local knowledge), and ϕ_2 (aggregation on the BFS tree once $\boldsymbol{y}$ is known).
2. Check the condition in line 4. If it holds, locally update $\boldsymbol{b}$, $\boldsymbol{f}$, and k_f, and go to Step 1.
3. Compute the potential $\boldsymbol{\pi}$ (local knowledge).
4. For each $e \in E$, its incident edges determine $\frac{\partial \phi}{\partial f_e}$ (based on (3) and (4), it suffices to exchange π_u and π_v over e).
5. Compute δ (aggregation on the BFS tree).
6. Locally update f_e and b_v for all $e \in E$ and $v \in V$.

Note that all of the individual operations except for computation of $\boldsymbol{y}$ and $\boldsymbol{\pi}$ can be completed in $O(D)$ rounds. It is shown in [32] that AlmostRoute terminates after $\tilde{O}(\varepsilon^{-3}\alpha^2)$ iterations. As it is only called $O(\log n)$ times by Algorithm 1, Theorem 1.1 follows once we show how to compute $\boldsymbol{y}$ and $\boldsymbol{\pi}$ in $(\sqrt{n} + D)n^{o(1)}$ rounds, for an α-congestion approximator with $\alpha = n^{o(1)}$. Hence, the remainder of the article is concerned with constructing a suitable congestion approximator.

4. Congestion approximator: Outline of the distributed construction. In this section, we outline how to adapt Madry's construction to its recursive application in the distributed setting. In section 6, we formally prove that we achieve the same guarantees as Madry's distribution [21] in each recursive step and that our distributed implementation is fast. Here, we focus on presenting the main ideas of the required modifications to Madry's scheme and its distributed implementation; to this end, it suffices to consider the construction of a single step of the recursion.

Centralized algorithm. As a starting point, let us summarize the main steps of one iteration of the centralized construction. It is convenient to state a slightly simplified variant of Madry's construction, which offers the same worst-case performance. It is assumed that an edge length function ℓ_e is known (in the distributed setting, this knowledge will be local). Given $j \leq n-1$, the following construction yields a $\Theta(j)$-tree.

1. Compute a spanning tree $\mathcal{T}$ of G of stretch α.
2. For each edge $e = \{v, w\} \in E$ of the graph G, route $\text{cap}(e)$ units of a commodity com_e from v to w on (the unique path from v to w in) $\mathcal{T}$.[5] Denote by $\boldsymbol{f}$ the vector of the sum of absolute flows passing through the edges of $\mathcal{T}$. Recall that edge capacities are polynomial in n, and hence so are the flows.
3. For $e \in \mathcal{T}$, define the *relative load* of e as $\text{rload}(e) = |f_e|/\text{cap}(e) \in n^{O(1)}$. Partition the edge set of $\mathcal{T}$ into $O(\log n)$ subsets $\mathcal{F}_i$, $i \in \{1, \ldots, \lceil \log(\|\boldsymbol{f}\|_\infty + 1) \rceil\}$, of roughly equal relative load: Let W denote the maximal relative load over all edges. Then F_i contains all edges with relative load in $(W/2^i, W/2^{i-1}]$. As $\mathcal{T}$ has $n-1 \geq j$ edges, there must be some $\mathcal{F}_i$ with $\Omega(j/\log n)$ edges; let

[5]For our purpose, the difference between multi- and single-commodity flows is just that distinct flows in opposing directions do not cancel out, and hence *any* feasible (i.e., congestion-1) flow in G can be routed on T with at most the congestion of this multicommodity flow.

i_0 be minimal with this property. Define $\mathcal{F} := \{e \in \mathcal{T} \mid \text{rload}(e) > W/2^{i_0-1}\}$. Note that $|\mathcal{F}| \leq j$, because it is the union of the subsets $\mathcal{F}_1 \ldots, \mathcal{F}_{i_0-1}$, each containing fewer than $j/\lceil \log(\|\boldsymbol{f}\|_\infty + 1) \rceil \in \Omega(j/\log n)$ edges.

4. $\mathcal{T} \setminus \mathcal{F}$ is a forest of at most $j+1$ components. Define H as the graph on node set V whose edge set is the union of $\mathcal{T} \setminus \mathcal{F}$ and all edges of G that connect different components of $(V, \mathcal{T} \setminus \mathcal{F})$.
5. For any two components C and C' of $(V, \mathcal{T} \setminus \mathcal{F})$, let $p(C, C')$ be the node in C closest to C' in $\mathcal{T}$. Replace each edge between different components C, C' of $(V, \mathcal{T} \setminus \mathcal{F})$ by an edge $\{p(C, C'), p(C', C)\}$ of the same weight. This results in a multigraph with (possibly) parallel edges.
6. Let $P = \{p(C, C') \mid C, C' \text{ are components of } \mathcal{T} \setminus \mathcal{F}\}$ be the set of *portal* nodes. In the multigraph defined above, iteratively delete nodes from $V \setminus P$ of degree 1 until no such node remains. Note that the leaves of the induced subtree of $\mathcal{T}$ must be in P, showing that the number of remaining nodes in $V \setminus P$ of degree larger than 2 is bounded by $|P| - 1 < 2j$. Add all such nodes to P; i.e., $V \setminus P$ then only contains nodes of degree 2.
7. For each path with endpoints in P and no inner nodes in P, delete an edge of minimum capacity and replace it by an edge of the same capacity between its endpoints.
8. Re-add the nodes and edges of $\mathcal{T} \setminus \mathcal{F}$ that have been deleted in Step 6.
9. For any $p, q \in P$, merge all parallel edges $\{p, q\}$ into a single one whose capacity is the sum of the individual capacities. The result is a j'-tree for $j' = |P| < 4j$.

In his paper, Madry provides a scheme for updating the edge lengths between iterations so that this construction results in a distribution on $\tilde{O}(m/j)$ $\Theta(j)$-trees that approximate cuts up to an expected $O(\alpha)$-factor, where α is the stretch of the spanning tree construction. Updating the edge length function poses no challenge, so here we focus on the distributed implementation of the procedure above steps in this section.

Modifications of the centralized algorithm. Before we specify the distributed algorithm, we introduce a few changes to the algorithm in centralized terms. These do not affect the reasoning underlying the scheme but greatly simplify its distributed implementation.

- We omit Step 9, and instead operate on cores that are multigraphs. This changes the computed distribution, as we formally use a different graph as input to the recursion. However, Räcke's arguments (and Madry's generalization) work equally well on multigraphs. To see that, replace each edge of the multigraph by a path of length 2, where both edges have the same capacity as the original edge. This recovers a graph of $2m$ edges from a multigraph of m edges without changing the min-cut structure, and the resulting trees can be interpreted as trees on the multigraph by contraction of the previously expanded edges. Similarly, both the low average stretch spanning tree construction and the cut sparsifier work on multigraphs without modification.
- After computing the spanning tree in Step 1, we delete a subset of $\tilde{O}(\sqrt{n})$ edges to ensure (w.h.p.) that the new clusters have low-depth spanning trees. The deleted edges are replaced by all edges of G crossing the corresponding cuts and will end up in the core. The same procedure is, in fact, applied to all edges selected into $\mathcal{F}$ in Step 3 of the centralized routine; Madry shows that replacing *any* subset of edges of $\mathcal{T}$ this way can only improve the quality of cut approximation. In our case, the number of replaced edges is negligible

relative to the size of the core, and there is no effect on the asymptotic progress guarantee.

- In the counterpart to Step 6 in Madry's routine, nodes from P may also be removed if their degree drops to 1. Also here, there is no asymptotic difference in the worst-case performance of our routine from Madry's.

To simplify presentation, in this section we assume that all trees involved in the construction have depth $\tilde{O}(\sqrt{n})$. This means that we can omit the deletion of $\tilde{O}(\sqrt{n})$ additional edges and further related technicalities. The general case is handled by standard techniques for decomposing trees into $O(\sqrt{n})$ components of depth $\tilde{O}(\sqrt{n})$ and relying on a BFS tree to communicate "summaries" of the components to all nodes in the graph within $\tilde{O}(\sqrt{n}+D)$ rounds (full details are given in section 6). This approach was first used for MST construction [20]; we use a simpler randomized variant (cf. Lemma 6.5).

Cluster graphs. Recall that we shall recursively invoke (a variant of) the above centralized procedure on the core. We need to simulate the algorithm on the core by communicating on G. To this end we use *cluster graphs* (see section 5), in which G is decomposed into components that play the role of core nodes. We maintain the following invariants during the recursive construction:

1. There is a 1-1 correspondence between core nodes and *clusters*.
2. Each cluster c has a rooted spanning tree of depth $\tilde{O}(\sqrt{n})$.
3. No other edges exist inside clusters. Contracting clusters yields the multigraph resulting from the above construction without Step 9. From now on, we shall refer to this multigraph as the core.
4. All edges in the (noncontracted) graph are also edges of G, and the nodes at their endpoints know their assigned lengths.

Overview of the distributed routine. We follow the same strategy as the centralized algorithm, with the modifications discussed above. This implies that the core edges for the next recursive call will simply be the graph edges between the newly constructed clusters. The main steps of our distributed implementation are as follows.

1. Compute a spanning tree $\mathcal{T}$ of stretch α of the core. This is done by the spanning tree algorithm of Theorem 3.2, which can operate on the cluster graph.
2. For each edge $e \in \mathcal{T}$, determine its absolute flow $|f_e|$ (and thus also $\text{rload}(e) = |f_e|/\text{cap}(e)$): this is not completely trivial because the edge represents all edges connecting nodes in the two sides of the cut (in G) induced by e in the cluster tree (cf. Figure 2). We use the following idea.
   (a) Let each cluster c learn of all its ancestor clusters in the spanning tree.
   (b) Let $e = (c, \hat{c})$ where c is the child of $\hat{c}$ in the tree. Observe that an edge contributes to the cut corresponding to e if and only if *exactly* one of its endpoints has c as an ancestor. This condition can be tested by letting each node exchange ancestor information with its neighbors.
   (c) Using aggregation on the spanning tree, we compute $|f_e|$ as the sum of capacities contributing to it, which is known to the nodes in $\mathcal{T}_c$. These aggregations are performed concurrently for all clusters c, where we exploit pipelining to handle the resulting contention.
3. Determine the index i_0 (as in Step 3 of the centralized routine). Given that $\text{rload}(e)$ for each $e \in \mathcal{T}$ is locally known, this is performed in $\tilde{O}(D)$ rounds using binary search in combination with convergecasts and broadcasts on a BFS tree. We set $\mathcal{F} := \{e \in \mathcal{T} \mid \text{rload}(e) > W/2^{i_0-1}\}$.

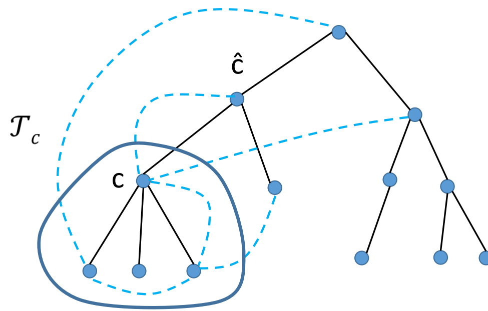


FIG. 2. *Illustration of the underlying idea of the aggregation scheme for the cut capacities. For a cluster c with parent $\hat{c}$, the cut corresponding to edge $(c,\hat{c})$ has a total capacity given by all graph edges leaving the subtree $\mathcal{T}_c$ (dashed lines). By having each node learn the list of its ancestor clusters and sending them to its neighbors, each node can determine for each ancestor cluster which of its incident edges contribute to the respective cut. Using pipelining, all communication can be done in time proportional. Pipelining the respective aggregations on the tree for each such edge $(c,\hat{c})$, all aggregations can be completed within time proportional to the depth of the tree.*

4. Define P as the set of clusters incident to edges in $\mathcal{F}$. A simple broadcast on the cluster spanning trees makes membership of each cluster in P known to all nodes.
5. Iteratively mark clusters $c \notin P$ with at most one unmarked neighboring cluster, until this process stops. Add all unmarked clusters that retain more than two unmarked neighboring clusters to P.
6. For each path with endpoints in P whose inner nodes are unmarked clusters not in P, find the edge $e \in \mathcal{T} \setminus \mathcal{F}$ of minimal capacity and add it to $\mathcal{F}$. This disconnects any two clusters $c, c' \in P$, $c \neq c'$, in $\mathcal{T} \setminus \mathcal{F}$.
7. Each component of $\mathcal{T} \setminus \mathcal{F}$ and the spanning trees of clusters induce a spanning tree of the corresponding component of G. Each such component is a new cluster. Make the identifier of the unique $c \in P$ of each cluster known to its nodes and delete all edges between nodes in the cluster that are not part of its spanning tree.

If all trees have depth $\tilde{O}(\sqrt{n})$, all the above steps can be completed in $\tilde{O}(\sqrt{n} + D)$ rounds. Clearly, the first three stated invariants are satisfied by the given construction. As mentioned earlier, it is also straightforward to update the edge lengths, i.e., establish the fourth invariant. Once the distribution on the current level of recursion is computed, one can hence sample and then move on to the next level.

For the detailed description of the algorithm and its analysis, the reader is referred to section 6.

5. Cluster graphs.

On several levels, our distributed congestion approximator construction is done in a hierarchical way. As a consequence many of the distributed computations used by our algorithm have to be run on a graph induced by clusters of the network graph. In order to be able to deal with such cluster graphs in a systematic way, we formally define cluster graphs and we describe how to simulate distributed computations on a cluster graph by running a distributed algorithm on the underlying network graph.

DEFINITION 5.1 (distributed cluster graph). *Given an n-network graph $G = (V, E)$, a distributed N-node cluster graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}, \mathfrak{T}, \psi)$ of size n is defined by a set of N clusters $\mathcal{V} = \{S_1, \ldots, S_N\}$ partitioning the vertex set V, a set (or multiset) of*

edges $\mathcal{E} \subseteq \binom{\mathcal{V}}{2}$*, a set of cluster leaders* $\mathcal{L}$*, a set of cluster trees* $\mathfrak{T}$*, and a function* ψ *that maps the edges* $\mathcal{E}$ *of the cluster graph to edges in* E*. Formally, the tuple* $(\mathcal{V}, \mathcal{E}, \mathcal{L}, \mathfrak{T}, \psi)$ *has to satisfy the following conditions.*

(I) *The clusters* $\mathcal{V} = (S_1, \dots, S_N)$ *form a partition of the set of vertices* V *of the network graph, i.e.,* $\forall i \in [N] : S_i \subseteq V$*,* $\forall 1 \leq i < j \leq N : S_i \cap S_j = \emptyset$*, and* $\bigcup_{i=1}^{N} S_i = V$*.*

(II) *For each cluster* S_i*,* $|S_i \cap \mathcal{L}| = 1$*. Hence, each cluster has exactly one cluster leader* $\ell_i \in \mathcal{L} \cap S_i$*. The ID of the node* ℓ_i *also serves as the ID of the cluster* S_i *and for the purpose of distributed computations, we assume that all nodes* $v \in S_i$ *know the cluster ID and the size* $n_i := |S_i|$ *of their cluster* S_i*.*

(III) *Each cluster tree* $T_i = (S_i, E_i)$ *is a rooted spanning tree of the subgraph* $G[S_i]$ *of* G *induced by* S_i*. The root of* T_i *is the cluster leader* $\ell_i \in S_i \cap \mathcal{L}$*. We assume that each node of* $u \in S_i \setminus \{\ell_i\}$ *knows its parent node* $v \in S_i$ *in the tree* T_i*.*

(IV) *The function* $\psi : \mathcal{E} \to E$ *maps each edge* $\{S_i, S_j\} \in \mathcal{E}$ *to an (actual) edge* $\{v_i, v_j\} \in E$ *connecting the clusters* S_i *and* S_j*; i.e., it holds that* $v_i \in S_i$ *and* $v_j \in S_j$*. The two nodes* v_i *and* v_j *know that the edge* $\{v_i, v_j\}$ *is used to connect clusters* S_i *and* S_j*. If the cluster graph is weighted, the two nodes* v_i *and* v_j *also know the weight of the edge* $\{S_i, S_j\}$*.*

Note that (III) in particular implies that the subgraph of G induced by each cluster S_i is connected. When dealing with a concrete distributed cluster graph $\mathcal{G}$, we use $\mathcal{G}_{\mathcal{V}}$, $\mathcal{G}_{\mathcal{E}}$, $\mathcal{G}_{\mathcal{L}}$, $\mathcal{G}_{\mathfrak{T}}$, and $\mathcal{G}_{\psi}$ to denote the corresponding sets of clusters, edges, etc. Further, when only arguing about the cluster graph and not its mapping to G, we only use the pair $(\mathcal{V}, \mathcal{E})$ to refer to it. In the following, we say that a cluster $S \in \mathcal{V}$ *knows* something if all nodes $v \in S$ know it. That is, e.g., the last part of condition (II) says that every cluster knows its ID and its size.

We next define a weak version of the (synchronous) CONGEST model, and we show that algorithms in this model can be efficiently simulated in distributed cluster graphs.

Definition 5.2 (B-Bounded Space CONGEST model). *For a given parameter* B*, the* B-Bounded Space CONGEST model *is the CONGEST computational model that uses messages of size* $O(B)$ *bits with the following additional requirement: For any* $d \geq 0$*, each step (i.e., a round of receive messages, compute, and send messages) of a node* v *in the* B*-Bounded Space model can be emulated as follows. Node* v *is replaced by a tree* $T(v)$ *of depth at most* d*. Each node of* $T(v)$ *knows the entire state of* v *and each edge incident to* v *is incident to some node of* $T(v)$*. Then, after each node of* $T(v)$ *has received the messages on its incident edges, the new state of* v *and the correct messages to be sent on all incident edges can be computed in* $O(d)$ *CONGEST model rounds on* $T(v)$*.*

The definition of the B-Bounded Space model is tailored for emulation. The definition immediately implies that if each node is emulated by a tree, then emulating a global step in time proportional to the maximal tree depth (which could be $\Omega(n)$) is trivial. However, the following lemma shows that this can actually be done in time $O(D + \sqrt{n})$.

Lemma 5.3. *Given an underlying* n*-node graph* $G = (V, E)$ *and a cluster graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}, \mathfrak{T}, \psi)$*, a* t*-round distributed algorithm* $\mathcal{A}$ *in* $\mathcal{G}$ *in the* B*-Bounded Space CONGEST model can be simulated in the (ordinary) CONGEST model in* G *with messages of size at most* B *in* $O\big((D + \sqrt{n}) \cdot t\big)$ *rounds, where* D *is the diameter of* G*.*

Proof. We assume that we are given a global BFS tree of G. If such a BFS

tree is not available, it can be computed in $O(D)$ rounds in the CONGEST model. We simulate the algorithm $\mathcal{A}$ in a round-by-round manner. Consider the end of the simulation of round $r-1$ and assume that in each cluster $S_i \in \mathcal{V}$, the leader node ℓ_i knows the message M_i to be sent in round r. (For $r = 1$, we assume that this is true at the beginning of the simulation.)

To start the simulation of round r, we first make sure that for every $S_i \in \mathcal{V}$, every node $v \in S_i$ knows the message M_i to be sent to the neighbors in round r. In clusters S_i of size at most $\sqrt{n}$, this can be done in at most $\sqrt{n}$ rounds by broadcasting M_i on the spanning tree T_i of $G[S_i]$. For larger clusters, we use the global BFS tree to disseminate the information. We first send all the messages M_i of clusters S_i of size larger than $\sqrt{n}$ to the root of the global BFS tree. Because the BFS tree has radius at most D and because there are at most $\sqrt{n}$ clusters of size larger than $\sqrt{n}$, this can be done in $D + \sqrt{n}$ rounds (using pipelining). Now, in another $D + \sqrt{n}$ rounds, all these messages can be broadcast to all nodes of G (and thus also to the nodes of the clusters that need to know them).

Now, for every two clusters S_i and S_j such that $\{S_i, S_j\} \in \mathcal{E}$, let $\{u_i, u_j\} = \psi(\{S_i, S_j\})$ be the physical edge connecting S_i and S_j. The node $u_i \in S_i$ sends the message M_i to u_j and the node $u_j \in S_j$ sends M_j to u_i. This step can be done in a single round. Now, in each cluster S_i, each incoming message of round r is known by one node in S_i and we need to aggregate these messages in order to compute the outgoing message of each cluster. In clusters of size at most $\sqrt{n}$, this can again be done locally inside the cluster (by Definition 5.2). Also, for the clusters of size larger than $\sqrt{n}$ (there are at most $\sqrt{n}$ of them), we again use the global BFS tree. Since in a tree of depth D, k independent convergecasts and broadcasts[6] can be done in time $D + k$, the messages of the large clusters can be computed and disseminated to the cluster leaders in time $O(D + \sqrt{n})$. ∎

6. Distributed construction of *j*-trees. From the distributed implementation point of view, the core technical challenge is to efficiently compute a congestion approximator in a distributed way. As already pointed out, the congestion approximator is constructed based on applying the j-tree construction of Madry [21] recursively. In the following, we review Madry's construction and we show how to implement an adapted version of it in a distributed network. The main objective of the construction is to approximate the flow structure of a given graph by a distribution of graphs from a simpler class of graphs (i.e., j-trees). Formally, the similarity of the flow structure of two graphs is captured by the following definition from [21].

DEFINITION 6.1 (graph embeddability [21]). *Given $\beta \geq 1$ and two multigraphs $G = (V, E, \mathrm{cap})$ and $G' = (V, E', \mathrm{cap}')$ on the same set of nodes, we say that graph G is β-embeddable into G' if there exists a* multicommodity flow $\boldsymbol{f}' = (\boldsymbol{f}'_e)_{e \in E}$ *such that for every edge $e \in E$ of G connecting nodes u and v, $\boldsymbol{f}'_e$ is a flow on $(V, E', \beta\,\mathrm{cap}')$ that routes* $\mathrm{cap}(e)$ *units of flow between u and v, and for every edge $e' \in E'$ of G', it holds that $|\boldsymbol{f}'(e')| := \sum_{e \in E} |(\boldsymbol{f}'_e)(e')| \leq \beta\,\mathrm{cap}'(e')$.*

Intuitively, a graph G is β-embeddable into a graph G' if for every (multicommodity) flow problem there is a solution in G' such that the maximum relative congestion of all edges is at most a factor β larger than for the optimal solution in G. As a generalization of the cut-based graph decompositions of Räcke [29], Madry defines

[6] In the broadcast operation, the root sends a message to all nodes, where each node sends the message to all its children; convergecast is the inverse operation, where nodes have messages whose aggregate value is to reach the root. See [26] for details.

the notion of an $(\alpha, \mathbb{G})$-decomposition.

DEFINITION 6.2 ($(\alpha, \mathbb{G})$-decomposition [21]). *Given a (multi)graph $G = (V, E, \mathrm{cap})$ and a family $\mathbb{G}$ of graphs on the nodes V, an $(\alpha, \mathbb{G})$-decomposition of G is a set of pairs $\{(\lambda_i, G_i)\}_{i\in I}$ satisfying that*

- $\forall i \in I: \ \lambda_i > 0$;
- $\sum_{i\in I} \lambda_i = 1$;
- $\forall i \in I: \ G_i = (V, E_i, \mathrm{cap}_i)$ *is a graph in* $\mathbb{G}$;
- $\forall i \in I: \ G$ *is* 1*-embeddable into* G'; *and*
- *the graph defined by the convex combination*[7] $\sum_{i\in I} \lambda_i \cdot G_i$ *is* α*-embeddable into* G.

In words, $\{(\lambda_i, G_i)\}_{i\in I}$ *is a distribution on* I *graphs from* $\mathbb{G}$*, each of which can be* 1*-embedded into* G*, such that the distribution* α*-embeds into* G*.*

Observe that such a decomposition can form the basis for a good congestion approximator: 1-embeddability of each G_i into G guarantees that congestion is never overestimated, and the embeddability of the convex combination ensures that when sampling from the distribution, the expected factor by which we underestimate congestion on a cut is at most α. Our goals are now to choose $\mathbb{G}$ and the distribution such that

- α is small,
- we can construct the distribution efficiently, and
- we can evaluate the induced congestion when routing demand optimally on a graph from the distribution efficiently.

The plan. Let $G = (V, E, \mathrm{cap})$ be a weighted (multi)graph, $0 \le j \le |V|$ be an integer, and $\mathbb{J}$ be the family of j-trees over the node set V. In [21], it is shown that based on a protocol for computing spanning trees with average stretch α, there exists an $(\alpha, \mathbb{J})$-decomposition of G. This is shown in several steps. It is first shown that a sparse $(\alpha, \mathbb{H})$-decomposition exists for a graph family $\mathbb{H}$ which contains graphs that are closer to the original graph G, and it is then shown that every graph $\mathcal{H} \in \mathbb{H}$ can be $O(1)$-embedded into a j-tree and vice versa.

As described, we have to apply the j-tree construction recursively to the core graph. Each node in the core graph is represented by a set of nodes (a cluster) in the network graph. On the network graph, the core graph therefore corresponds to a graph between clusters of nodes. We therefore have to be able to apply the j-tree construction on a cluster graph. As we will see, we can construct j-trees such that whenever two nodes u and v of the core are connected by a (virtual) edge, there also is a physical edge between the two trees (i.e., clusters of nodes) corresponding to u and v. Throughout our algorithm, we can therefore work with a cluster multigraph such that (a) the induced graph of each cluster is connected, and (b) for every edge between two clusters c and c', there are nodes $u \in c$ and $v \in c'$ such that u and v are connected by an edge in the underlying network graph. When doing distributed computations, we assume that each cluster has a leader and that every node knows the ID of the leader and also its parent in a rooted spanning tree which is rooted at the leader. In section 5, we gave a precise definition of a *distributed cluster graph* and showed that several basic algorithms that we use as building blocks can be run efficiently in distributed cluster graphs.

[7]The sum of two weighted graphs $G_1 = (V, E_1, \mathrm{cap}_1)$ and $G_2 = (V, E_2, \mathrm{cap}_2)$ is defined as $G_1 + G_2 = (V, E_1 \cup E_2, \mathrm{cap}_{12})$, where $\mathrm{cap}_{12}(e) = \begin{cases} \mathrm{cap}_1(e) + \mathrm{cap}_2(e) & \text{if } e \in E_1 \cap E_2, \\ \mathrm{cap}_i(e) & \text{if } e \in E_i \setminus E_{3-i}. \end{cases}$

In the following, we go through Madry's j-tree construction step by step and describe how to adapt it so that we can implement it efficiently on a distributed cluster graph (i.e., in the CONGEST model in the underlying network graph).

6.1. Low-stretch spanning trees. In the following, we consider the computation of the $(\alpha, \mathbb{J})$-decomposition of some core graph. Assume that the core graph is given as a distributed cluster graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \text{cap}, \mathfrak{T}, \psi)$, where for each edge $e \in \mathcal{E}$, $\text{cap}(e)$ is the capacity of $\psi(e)$.[8] As the time complexity of some of the steps for computing an $(\alpha, \mathbb{J})$-decomposition of $\mathcal{G}$ depends on the number of edges of $\mathcal{G}$, as a first step, we sparsify $\mathcal{G}$. In Lemma 7.1, it is shown that in $O\big((D+\sqrt{n})\cdot \text{polylog}\, n\big)$ rounds, it is possible to compute an $(1 + 1/\text{polylog}\, n)$-spectral sparsifier of $\mathcal{G}$ with at most $O(|\mathcal{V}| \cdot \text{polylog}\, n)$ edges. Further, for each edge $\{c, c'\} \in \mathcal{E}$ of the sparsifier, one of the nodes of the edge manages the edge. As in general $\mathcal{G}$ is a cluster graph, c and c' are clusters of physical nodes, and an edge connecting clusters c and c' is represented by a physical edge $\{u, v\}$ for two (network) nodes $u \in c$ and $v \in c'$. We will maintain that every pair of nodes $u \in c$ and $v \in c'$ needs to represent at most one edge between c and c' in $\mathcal{G}$. The two nodes u and v know about the edge between c and c' and its capacity.

In the following, we assume that $\mathcal{G}$ is the graph after sparsification. If the number of nodes $|\mathcal{V}|$ of $\mathcal{G}$ is less than $n^{1/2+o(1)}$, using a global BFS tree of the network graph, the whole structure of $\mathcal{G}$ can be collected in $O(D + |\mathcal{V}|\, \text{polylog}\, n) = O(D + n^{1/2+o(1)})$ rounds. In that case, we can therefore perform all remaining operations locally at the nodes (recall that local computation is free). Consequently, we will henceforth assume that $|\mathcal{V}| \geq n^{1/2+o(1)}$.

During the construction of the $(\alpha, \mathbb{J})$-decomposition of $\mathcal{G}$, each edge $e \in \mathcal{E}$ is assigned a length $\ell(e)$. At the beginning $\ell(e)$ is proportional to $1/\text{cap}(e)$, and before adding each j-tree, $\ell(e)$ is adapted for each edge. As the first step of constructing each j-tree in the decomposition, Madry computes a spanning tree $\mathcal{T}$ of $\mathcal{G}$ for which it holds that

$$\sum_{e=\{c,c'\}\in\mathcal{E}} d_{\mathcal{T}}(c,c')\cdot \text{cap}(e) \;=\; \sum_{e=\{c,c'\}\in\mathcal{E}} \text{stretch}_{\mathcal{T}}(e)\cdot \ell(e)\cdot \text{cap}(e) \;\leq\; \delta\alpha\cdot \sum_{e=\{c,c'\}\in\mathcal{E}} \ell(e)\cdot \text{cap}(e) \tag{5}$$

for a sufficiently small positive constant δ. In the above expression, $d_{\mathcal{T}}(u, v)$ denotes the sum of edge lengths on the path between u and v on $\mathcal{T}$. Hence, $\mathcal{T}$ is a spanning tree with a bounded weighted average stretch. Such a spanning tree can be computed by computing an (unweighted) low average stretch spanning tree for a multigraph $\tilde{\mathcal{G}}$ which is obtained from $\mathcal{G}$ by (logically) replacing some of the edges of $\mathcal{G}$ with multiple copies of the same edge (overall, the number of edges is as most doubled) [3, 21].

In our distributed implementation of Madry's j-tree construction, we adapt the parallel low average stretch spanning tree algorithm from [11] to our setting. The algorithm of [11] already works in a mostly decentralized fashion, and we can therefore also apply it in a distributed setting. In section 8, we show how to run the algorithm of [11] on a distributed cluster graph. We note that the low average stretch spanning tree algorithm of [11] directly tolerates multiedges as described above, even if the same physical edge has to be used to represent multiple edges between the same clusters.

Theorem 6.3 (Theorem 3.2 restated and rephrased). *Suppose $\mathcal{G}$ is a multigraph obtained from G by assigning arbitrary edge lengths in $2^{n^{o(1)}}$ to the edges of G (known*

[8]Recall that ψ maps an edge of the cluster graph to an edge in the original graph.

to incident nodes) and performing an arbitrary sequence of contractions. Then we can compute a rooted spanning tree of $\mathcal{G}$ of expected stretch $2^{O(\sqrt{\log n \log\log n})}$ within $(\sqrt{n}+D)n^{o(1)}$ rounds, where the edges of the tree in $\mathcal{G}$ and their orientation are locally known to the endpoints of the corresponding edges in G.

Given the spanning tree $\mathcal{T} = (\mathcal{V}_\mathcal{T}, \mathcal{E}_\mathcal{T})$ of $\mathcal{G}$, we need to compute capacities $\text{cap}_\mathcal{T}(e)$ for the spanning edges such that $\mathcal{G}$ is embeddable into $\mathcal{T}$, which essentially boils down to computing the absolute value of the multicommodity flow $\boldsymbol{f}'$ routing $\text{cap}(e)$ units of flow on $\mathcal{T}$ for each $e \in E$. As routing is trivial in trees, $\boldsymbol{f}'$ is unique. Once $|\boldsymbol{f}'|$ is computed, the edge capacities of $\mathcal{T}$ can be chosen accordingly, and it is straightforward to pick a suitable λ_i and update the length function $\ell(e)$. However, computing the absolute value of the multicommodity flow *quickly* in the distributed setting requires some work.

Computing the multicommodity flow. In the following, assume that the edges of $\mathcal{T}$ are oriented towards the root; i.e., we will write $(c, \hat{c}) \in \mathcal{T}$ if $\hat{c} \in \mathcal{V}$ is the parent of cluster $c \in \mathcal{V}$. Denote by $\mathcal{T}_c$ the subtree of $\mathcal{T}$ rooted at c. When embedding $\mathcal{G}$ into $\mathcal{T}$, we have to route a total of

$$|\boldsymbol{f}'(c,\hat{c})| = \sum_{\substack{c_1 \in \mathcal{T}_c \\ c_2 \notin \mathcal{T}_c}} \sum_{\{c_1,c_2\}_{uv} \in \mathcal{E}} \text{cap}(\{c_1, c_2\}_{uv})$$

units of commodity through the edge $\{c, \hat{c}\} \in \mathcal{T}$, where we indexed the different edges of the multigraph $(V, \mathcal{E})$ by using that for each $e \in \mathcal{E}$ between c_1 and c_2, the embedding maps e to a unique edge $\{u, v\} \in E$ with $u \in c_1$ and $w \in c_2$. Note that u and v know that $\{c_1, c_2\}_{uv} \in \mathcal{E}$, that they are in the clusters c_1 and c_2, respectively, and what $\text{cap}(\{c_1, c_2\}_{uv})$ is. We thus have to solve the task of determining this sum for each edge $\{c, \hat{c}\} \in \mathcal{T}$ via computations on the graph G underlying $(\mathcal{V}, \mathcal{E})$.

Observe that the spanning trees of the clusters together with $\mathcal{T}$ induce a (rooted) spanning tree T of G. Essentially, we would like to perform, for each edge $\{c, \hat{c}\} \in \mathcal{T}$, an aggregation on T and pipeline these aggregations to achieve good time complexity. However, as shown in the following lemma, the result is a running time linear in the depth of the tree, which may be $\Omega(n)$ irrespective of D.

LEMMA 6.4. *If T has depth d, for each edge $e = (c, \hat{c}) \in \mathcal{T}$, c can determine $|\boldsymbol{f}'(e)|$ within $O(d)$ rounds.*

Proof. Consider the following algorithm.

1. For each cluster, all of its nodes learn the ancestors of the cluster in $\mathcal{T}$.
2. For each edge $\{c_1, c_2\}_{uv} \in \mathcal{E}$, u and v exchange the ancestor lists of c_1 and c_2.
3. Each node $u \in c_1 \in \mathcal{T}$ locally computes for each ancestor c of c_1 the value

$$\text{cap}_c(u) := \sum_{\substack{\{c_1,c_2\}_{uv} \in \mathcal{E} \\ c \text{ is not ancestor of } c_2}} \text{cap}(\{c_1, c_2\}_{uv}).$$

4. For each edge $e = (c, \hat{c}) \in \mathcal{T}$, we aggregate $\sum_{u \in T_c} \text{cap}_c(u)$ on T_c, where T_c is the subtree of T corresponding to $\mathcal{T}_c$.

Observe that, by definition,

$$|\boldsymbol{f}'(c,\hat{c})| = \sum_{\substack{c_1 \in \mathcal{T}_c \\ c_2 \notin \mathcal{T}_c}} \sum_{\{c_1,c_2\}_{uv} \in \mathcal{E}} \text{cap}(\{c_1, c_2\}_{uv}) = \sum_{u \in T_c} \text{cap}_c(u),$$

as each edge $\{c_1, c_2\}_{uv} \in \mathcal{E}$ with $c_1 \in \mathcal{T}_c$ and $c_2 \in \mathcal{T} \setminus \mathcal{T}_c$ satisfies that either $u \in T_c$ or $v \in T_c$. Hence, it remains to show that the above routine can be implemented with a running time of $O(d)$.

Clearly, the first step takes d rounds: T has depth d, and we can perform concurrent floodings on all subtrees without causing contention. The second step requires at most $d-1$ rounds, as no node has more than $d-1$ ancestors. The third step requires local computations only. Finally, the fourth step can be performed in d rounds as well, since we can perform concurrent convergecasts on all subtrees without causing contention. ∎

To handle the general case, i.e., $d \gg \sqrt{n}$, we first decompose $\mathcal{T}$ into $O(\sqrt{n})$ parts of small diameter. There are different ways to achieve such a decomposition of $\mathcal{T}$ efficiently in a distributed way (e.g., by using techniques from [20]). The easiest way is to use randomization. Suppose c' is the parent cluster of nonroot cluster c. We sample edge $e = (c, \hat{c}) \in \mathcal{T}$ into the edge set $\mathcal{R}$ with independent probability $q_e := \min\{1, |c|/\sqrt{n}\}$. Then, w.h.p. the forest $T \setminus \psi(\mathcal{R})$ consists of $\tilde{O}(\sqrt{n})$ trees of depth $\tilde{O}(\sqrt{n})$.

LEMMA 6.5. *Let $\mathcal{T}$ be a rooted spanning tree of a cluster (multi)graph $\mathcal{G}$, let $\mathcal{R}$ be a subset of the edges chosen at random as described above, and assume that the spanning tree of each cluster has depth at most d. With high probability, the forest $T \setminus \psi(\mathcal{R})$ induced by the edges $\mathcal{T} \setminus \mathcal{R}$ and the cluster spanning trees consist of $O(\sqrt{n})$ rooted trees of depth $d + O(\sqrt{n}\log n)$.*

Proof. Clearly, the number of trees in the forest induced by $\mathcal{E}_{\mathcal{T}} \setminus \mathcal{R}$ is equal to $|\mathcal{R}| + 1$. The expected value for $|\mathcal{R}|$ is given by the sum of the probabilities q_e and thus $\mathbb{E}[|\mathcal{R}|] \leq \sqrt{n}$. A standard Chernoff bound implies that $|\mathcal{R}|$ does not exceed $\sqrt{n}$ by more than a constant factor w.h.p.

To bound the depth of each (rooted) tree in $T \setminus \psi(\mathcal{R})$, consider a cluster $c \in \mathcal{T}$ and a path p from the leader $\{r\} := \mathcal{L} \cap c$ to some node in the subtree T_r of T rooted at r. The depth of $T_r \cap c$ is bounded by d. Denote by E_p the set of edges of p that correspond to edges in $\mathcal{T}$, i.e., each $e \in E_p$ satisfies that $e = \psi(e')$ for some $e' \in \mathcal{T}$. Denote by $c(e)$ the child cluster of e, i.e., the endpoint further away from the root of T. By construction, $c(e) \in e'$ is also the cluster further away from the root of $\mathcal{T}$ (for the $e' \in \mathcal{T}$ with $\psi(e') = e$). Therefore, the length of p is bounded by

$$d + |E_p| + \sum_{e \in E_p} (|c(e)| - 1) = d + \sum_{e \in E_p} |c(e)|,$$

i.e., the sum of the number of its edges in c, the number of edges between clusters $|E_p|$, and the total number of nodes in all traversed clusters except c. The probability that no edge of p was removed by the sampling procedure is

$$P[\mathcal{R} \cap \psi^{-1}(E_p) = \emptyset] = \prod_{e \in E_p} (1 - q_e).$$

For each $e \in E_p$, we have that $q_e = \min\{1, |c(e)|/\sqrt{n}\}$. If $q_e = 1$ for any $e \in E_p$, the above probability is 0. Otherwise, we have that

$$P[\mathcal{R} \cap \psi^{-1}(E_p) = \emptyset] = \prod_{e \in E_p} \left(1 - \frac{|c(e)|}{\sqrt{n}}\right) \leq e^{-\sum_{e \in E_p} |c(e)|/\sqrt{n}}.$$

By the above upper bound on the length of p, we conclude that p has length $d +$

$O(\sqrt{n}\log n)$ or is w.h.p. not present in $T \setminus \psi(\mathcal{R})$. As the number of simple paths in a tree is bounded by $O(n^2)$, applying the union bound completes the proof. ∎

For simplicity, in the following we assume that the high probability statements of the above lemma hold with certainty; the final statements then follow by applying the union bound.

Throughout the construction, we will maintain that edges of $\mathcal{R}$ are never retained. As there will be $o(\log n)$ levels of recursion, by inductive use of the above lemma, it follows that clusters always have spanning trees of depth $\tilde{O}(\sqrt{n})$. Exploiting this property together with the small number of connected components of $\mathcal{T} \setminus \mathcal{R}$, we obtain a fast routine for the general case.

LEMMA 6.6. *With high probability, within $\tilde{O}(\sqrt{n}+D)$ rounds, for each edge $e = (c,\hat{c}) \in \mathcal{T}$, c can determine $|\boldsymbol{f}'(e)|$.*

Proof. Denote by $\mathcal{C}$ the connected components of $\mathcal{T}\setminus\mathcal{R}$. For $c \in \mathcal{T}$, denote by $C \in \mathcal{C}$ its connected component. We rewrite (the absolute value of) the multicommodity flow

$$|\boldsymbol{f}'(c,\hat{c})| = \sum_{\substack{c_1 \in \mathcal{T}_c\\ c_2 \notin \mathcal{T}_c}} \sum_{\{c_1,c_2\}_{uv} \in \mathcal{E}} \operatorname{cap}(\{c_1,c_2\}_{uv}) = |\boldsymbol{f}'_1(c,\hat{c})| + |\boldsymbol{f}'_2(c,\hat{c})| - |\boldsymbol{f}'_3(c,\hat{c})|,$$

where

$$|\boldsymbol{f}'_1(c,\hat{c})| := \sum_{\substack{c_1 \in \mathcal{T}_c\setminus C\\ c_2 \notin \mathcal{T}_c\setminus C}} \sum_{\{c_1,c_2\}_{uv} \in \mathcal{E}} \operatorname{cap}(\{c_1,c_2\}_{uv}),$$

$$|\boldsymbol{f}'_2(c,\hat{c})| := \sum_{\substack{c_1 \in \mathcal{T}_c\cap C\\ c_2 \notin \mathcal{T}_c}} \sum_{\{c_1,c_2\}_{uv} \in \mathcal{E}} \operatorname{cap}(\{c_1,c_2\}_{uv}), \text{ and}$$

$$|\boldsymbol{f}'_3(c,\hat{c})| := \sum_{\substack{c_1 \in \mathcal{T}_c\cap C\\ c_2 \in \mathcal{T}_c\setminus C}} \sum_{\{c_1,c_2\}_{uv} \in \mathcal{E}} \operatorname{cap}(\{c_1,c_2\}_{uv}).$$

Note that $|\boldsymbol{f}'_1(c,\hat{c})|$ does not depend on the component of c; i.e., we need to determine and make known only $|\mathcal{C}| \in O(\sqrt{n})$ values to cover this term. For the other terms, we will reduce the problem to an aggregation on the spanning tree of C in the vein of Lemma 6.4.

Concerning $|\boldsymbol{f}'_1(c,\hat{c})|$, we employ the following routine.

1. Using its spanning tree, each component $C \in \mathcal{C}$ determines a unique identifier (say, the smallest cluster identifier) and makes it known to all its nodes.
2. For each $\{c_1,c_2\}_{uv} \in \mathcal{E}$, u and v exchange their component identifiers.
3. The list of component identifiers and edges $(C,\hat{C})$ for each $(c,\hat{c}) \in \mathcal{R}$ is made known to all nodes. This enables each node to locally compute the tree resulting from contracting the components $C \in \mathcal{C}$ in $\mathcal{T}$.
4. For each $C \in \mathcal{C}$, fix an arbitrary $c \in C$. Each node $u \in \mathcal{T}_c\setminus C$ locally computes

$$\operatorname{cap}_C(u) := \sum_{\substack{\{c_1,c_2\}_{uv} \in \mathcal{E}\\ c_2 \notin \mathcal{T}_c\setminus C}} \operatorname{cap}(\{c_1,c_2\}_{uv});$$

   we set $\operatorname{cap}_c(u) := 0$ for all $u \notin \mathcal{T}_c \setminus C$ (nodes can determine whether they are in $\mathcal{T}_c \setminus C$ based on the information collected in the previous two steps).

5. For each $C \in \mathcal{C}$, make $\sum_{u \in V} \text{cap}_C(u)$ known to all nodes via a BFS tree of G. For all $c \in C$, we have that $|\boldsymbol{f}'_1(c, \hat{c})| = \sum_{u \in V} \text{cap}_C(u)$.

As discussed earlier, components' spanning trees have depth $\tilde{O}(\sqrt{n})$ and $|\mathcal{C}| \in O(\sqrt{n})$. Hence, Step 1 takes $\tilde{O}(\sqrt{n})$ rounds, and Steps 3 and 5 take $O(\sqrt{n} + D)$ rounds. Step 2 requires only one round of communication, and Step 4 is local. Overall, the routine requires $\tilde{O}(\sqrt{n} + D)$ rounds.

To determine $|\boldsymbol{f}'_2(c, \hat{c})|$ and $|\boldsymbol{f}'_3(c, \hat{c})|$ for each c, we proceed similarly to Lemma 6.5.

1. For each $C \in \mathcal{C}$ and each $c \in C$, all nodes in c learn the list of ancestors of c that are in C (using the spanning tree of C in G).
2. For each $\{c_1, c_2\}_{uv} \in \mathcal{E}$, u and v exchange their component identifiers, as well as the ancestor lists determined in the previous step.
3. The list of component identifiers and edges $(C, \hat{C})$ for each $(c, \hat{c}) \in \mathcal{R}$ is made known to all nodes.
4. For each $C \in \mathcal{C}$, $c \in C$, and $u \in \mathcal{T}_c \cap C$, u locally computes

$$\text{cap}_c(u) := \sum_{\substack{\{c_1,c_2\}_{uv} \in \mathcal{E} \\ v \in c_2 \notin \mathcal{T}_c}} \text{cap}(\{c_1, c_2\}_{uv}) - \sum_{\substack{\{c_1,c_2\}_{uv} \in \mathcal{E} \\ v \in c_2 \in \mathcal{T}_c \setminus C}} \text{cap}(\{c_1, c_2\}_{uv}).$$ [9]

5. For each edge $e = (c, \hat{c}) \in \mathcal{T}$, we aggregate $\sum_{u \in T_c \cap C} \text{cap}_c(u)$ on $T_c \cap C$, where T_c is the subtree of T (the spanning tree of G) corresponding to $\mathcal{T}_c$.

Note that, by definition of $\text{cap}_c(u)$, we have that

$$\sum_{u \in T_c \cap C} \text{cap}_c(u) = |\boldsymbol{f}'_2(c, \hat{c})| - |\boldsymbol{f}'_3(c, \hat{c})|.$$

Hence, it remains to analyze the running time of this second subroutine. Again, using that components' spanning trees have depth $\tilde{O}(\sqrt{n})$ and that $|\mathcal{C}| \in O(\sqrt{n})$, we can conclude that Steps 1, 2, and 5 take $\tilde{O}(\sqrt{n})$ rounds, while Step 3 takes $O(\sqrt{n} + D)$ rounds. As Step 4 requires local computation only, the resulting running time is $\tilde{O}(\sqrt{n})$ rounds. Overall, we conclude that $|\boldsymbol{f}'(c, \hat{c})|$ can be computed for each c within $\tilde{O}(\sqrt{n} + D)$ rounds, by running each of the two subroutines and summing up their outputs. ∎

6.2. Approximating $\mathcal{G}$ by a distribution over simpler graphs. Using the techniques of [29] and the above construction of low average stretch spanning trees, it is possible to design a distributed algorithm to compute a distribution of such spanning trees which approximates the cut structure of the underlying network graph within a factor $2^{O(\sqrt{\log n \log \log n})}$ (i.e., in the order of the average stretch of the computed spanning trees). However, when doing this, the number of spanning trees we need to compute can be linear in the size of $\mathcal{G}$. We follow the same general idea as Sherman, who applied the construction by Madry [21] recursively to decrease the *step* complexity, to avoid this sequential bottleneck and achieve a small *time* complexity in the distributed setting.

For each edge $e \in \mathcal{T}$, we define $\text{rload}_{\mathcal{T}}(e) := \text{cap}_{\mathcal{T}}(e)/\text{cap}(e) \geq 1$ to be the relative load of e (edges $e \in \mathcal{E} \setminus \mathcal{T}$ have $\text{rload}_{\mathcal{T}}(e) = 0$). The construction of [29] builds up a potential for each edge e of $\mathcal{G}$, where with each new tree added to the distribution, the potential of e grows by a term proportional to $\text{rload}_{\mathcal{T}}(e)/\max_{e' \in \mathcal{E}}\{\text{rload}_{\mathcal{T}}(e')\}$. The potential of each edge is bounded by $\alpha = n^{o(1)}$, and hence with every additional

[9] If $v \in C$, u can decide whether $v \in \mathcal{T}_c$ based on the ancestor lists. If $v \notin C$, u can decide whether $v \in \mathcal{T}_c$ based on v's component identifier and the information collected in Step 3.

spanning tree we are guaranteed to make progress for all edges $e \in \mathcal{E}$ with $\mathrm{rload}_{\mathcal{T}}(e)$ close to $\max_{e' \in \mathcal{E}}\{\mathrm{rload}_{\mathcal{T}}(e')\}$. In the worst case, this can just be a single edge for each spanning tree $\mathcal{T}$. The key idea of Madry [21] is to augment the tree $\mathcal{T}$ with additional edges in order to reduce the maximum relative load so that in the new graph, a large number of edges have a relative load close to the maximum one.

Basically, we can reach a large number of edges with relative load close to the maximum relative load by repeatedly deleting the edge with largest relative load until a large number of the remaining edges have a relative load that is within a constant factor of the remaining maximum relative load. When deleting some edges of $\mathcal{T}$, one has to add back some of the original edges of $\mathcal{G}$ in order to maintain the property that $\mathcal{G}$ is embeddable into the resulting graph. Formally, let $\mathcal{F} \subseteq \mathcal{T}$ be a subset of the spanning tree edges. The edge set $\mathcal{T} \setminus \mathcal{F}$ defines a spanning forest of $\mathcal{G}$ consisting of $|\mathcal{F}| + 1$ components. We define a subgraph $\mathcal{H}(\mathcal{T}, \mathcal{F})$ of $\mathcal{G}$ as follows. The node set of $\mathcal{H}(\mathcal{T}, \mathcal{F})$ is $\mathcal{V}$. Further, $\mathcal{H}(\mathcal{T}, \mathcal{F})$ contains all edges in $\mathcal{T} \setminus \mathcal{F}$ and it contains all edges $\{c, c'\}_{uv} \in \mathcal{E}$ of $\mathcal{G}$ for which c and c' are in different components in the forest induced by the edges in $\mathcal{T} \setminus \mathcal{F}$. Let $\mathcal{E}_{\mathcal{H}}$ be the set of edges of $\mathcal{H}$. We set the capacities $\mathrm{cap}_{\mathcal{H}}(e)$ of edges $e \in \mathcal{E}_{\mathcal{H}}$ to be $\mathrm{cap}_{\mathcal{H}}(e) := \mathrm{cap}_{\mathcal{T}}(e)$ if $e \in \mathcal{T} \setminus \mathcal{F}$ and $\mathrm{cap}_{\mathcal{H}}(e) := \mathrm{cap}(e)$ otherwise. Note that this guarantees that $\mathcal{G}$ is 1-embeddable into $\mathcal{H}$. For the following discussion, we define $\mathbb{H}[j]$ to be the set of graphs $\mathcal{H}(\mathcal{T}, \mathcal{F})$ for a spanning tree $\mathcal{T}$ of $\mathcal{G}$ and a set of edges $\mathcal{F}$ of $\mathcal{T}$ of size $|\mathcal{F}| \leq j$.

Assume that the weighted average stretch of the spanning tree $\mathcal{T}$ as given by (5) is bounded from above by α. Also recall that we assume that all capacities of G are integers that are polynomially bounded in the number of nodes n. As throughout our construction, each edge capacity always approximately corresponds to the capacity of some cut in G, it is not hard to guarantee that all capacities of $\mathcal{G}$ are integers between 1 and $\mathrm{poly}(n)$. Given a spanning tree $\mathcal{T}$ of $\mathcal{G}$, let $R := \max_{e \in \mathcal{T}}\{\mathrm{rload}_{\mathcal{T}}(e)\}$ be the largest relative load of all edges of $\mathcal{T}$. In order to determine the set of edges $\mathcal{F}$, we start by partitioning the edges in $\mathcal{T}$ into $i_{\max} = O(\log n)$ classes $\mathcal{F}_1, \ldots, \mathcal{F}_{i_{\max}}$, where class $\mathcal{F}_i$ contains all edges with relative load in $(R/2^i, R/2^{i-1}]$. Now, for any $j_0 \leq |\mathcal{T}|$, there exists an edge class $\mathcal{F}_i$ such that $\left|\bigcup_{i'<i} \mathcal{F}_{i'}\right| \leq j_0$ and $|\mathcal{F}_i| \geq j_0/i_{\max} = \Omega(j_0/\log n)$; otherwise, $|\mathcal{T}| = \left|\bigcup_i \mathcal{F}_i\right| < j_0 \leq |\mathcal{T}|$, a contradiction. We define $\mathcal{F}' := \bigcup_{i'<i} \mathcal{F}_{i'}$.

In [22], this set of edges is used to construct the graph $\mathcal{H}(\mathcal{T}, \mathcal{F}')$. For the distributed computation, it will be useful to have a graph $\mathcal{H}(\mathcal{T}, \mathcal{F})$ in which all the trees of the forest induced by $\mathcal{E}_{\mathcal{T}} \setminus \mathcal{F}$ have small diameter. We therefore rely on the same technique as for computing the capacities of $\mathcal{T}$ and remove a few random additional edges of $\mathcal{T}$. In fact, we can simply use the same subset of edges $\mathcal{R} \subseteq \mathcal{T}$ that has been determined and used before, prior to Lemma 6.5. We define $\mathcal{F} := \mathcal{F}' \cup \mathcal{R}$ and use the graph $\mathcal{H} = (\mathcal{V}, \mathcal{E}_{\mathcal{H}}, \mathrm{cap}_{\mathcal{H}}) := \mathcal{H}(\mathcal{T}, \mathcal{F})$. Since all the edges of $\mathcal{T}$ with $\mathrm{rload}_{\mathcal{T}}(E) > R/2^{i-1}$ are removed, all edges of $\mathcal{H}$ have relative load at most $R/2^{i-1}$. Further, all the $\Omega(j/\log n)$ edges of $\mathcal{F}_i$ have relative load larger than $R/2^i$. Based on Theorem 5.2 and Corollary 5.6 of [22] and on Theorem 3.2, we can show the following lemma.

LEMMA 6.7. *Given are a distributed cluster (multi)graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathrm{cap}_{\mathcal{G}})$ consisting of $|\mathcal{V}| = N$ clusters and $|\mathcal{E}| = N\,\mathrm{polylog}(n)$ edges and a parameter $j \geq 1$ such that $j = \omega(\sqrt{n}\log n)$. There is a distributed algorithm to compute a $\left(2^{O(\sqrt{\log N \log\log N})}, \mathbb{H}[j]\right)$-decomposition of $\mathcal{G}$ on $2^{O(\sqrt{\log N \log\log N})} \cdot N/j$ graphs, which runs in the CONGEST model on the underlying network graph G in $(D + \sqrt{n}) \cdot n^{o(1)} \cdot N/j$ rounds.*

Proof. Let $\alpha = 2^{O(\sqrt{\log n \log\log n})}$ be the average stretch guarantee of the spanning

tree algorithm. It follows directly from Theorem 5.2 and Lemma 5.5 in [22] and Lemma 6.6 that we can compute an $(O(\alpha), \mathbb{H}[|\mathcal{E}| \cdot \alpha \log(n)/s])$-decomposition of $\mathcal{G}$ on s graphs in time $s \cdot T_{\text{tree}}$ if the following conditions are satisfied:

(1) The time for computing one low average stretch spanning tree is upper bounded by T_{tree}.

(2) Given $\mathcal{T}$ and edge set $\mathcal{F}$ as computed above, let $\text{rload}_{\max} := \max_{e \in \mathcal{T} \setminus \mathcal{F}} \{\text{rload}_{\mathcal{T}}(e)\}$. The number of edges of $\mathcal{H}(\mathcal{T}, \mathcal{F})$ with relative load at least $\text{rload}_{\max}/2$ is $\Omega(|\mathcal{E}|\alpha/s)$.

As observed above, in $\mathcal{T}$, all edges in the set $\mathcal{F}_i$ have relative load between $\text{rload}_{\max}/2$ and $\text{rload}_{\max}$. When constructing $\mathcal{H}(\mathcal{T}, \mathcal{F})$, the relative load of edges in $\mathcal{T} \setminus \mathcal{F}$ does not change, and thus, all nodes in $\mathcal{F}_i \setminus \mathcal{F} = \mathcal{F}_i \setminus \mathcal{R}$ have relative load between $\text{rload}_{\max}/2$ and $\text{rload}_{\max}$. Recall that $|\mathcal{F}_i| = \Omega(j/\log n)$. By Lemma 6.5, w.h.p., we have $|\mathcal{R}| = O(\sqrt{n})$. Since we assumed that $j = \omega(\sqrt{n}\log n)$, we have $|\mathcal{F}_i| = \omega(|\mathcal{R}|)$, and thus $|\mathcal{F}| = \Omega(j/\log n)$. The second condition is now satisfied by choosing $s = \Theta(|\mathcal{E}|\alpha \log(n)/j) = 2^{O(\sqrt{\log N \log\log N})} \cdot N/j$.

Assuming that the time to compute a single low average stretch spanning tree can be bounded from above by $T_{\text{tree}} = (D + \sqrt{n}) \cdot n^{o(1)}$, the lemma now follows. By Theorem 3.2, this is guaranteed as long as all edge lengths are integers between 1 and $2^{n^{o(1)}}$. Inspecting the construction in [22] and [29], we can observe that the edge lengths cannot get larger than a value exponential in α. By rounding them to integers, we introduce an additional multiplicative error of factor 2, which does not affect the asymptotic behavior. As $\alpha = 2^{O(\sqrt{\log n \log\log n})}$, the claim of the lemma follows. ∎

6.3. Transforming $\mathcal{H}(\mathcal{T}, \mathcal{F})$ into a j-tree. Given a graph $\mathcal{H}(\mathcal{T}, \mathcal{F}) \in \mathbb{H}[j]$, it remains to transform $\mathcal{H}(\mathcal{T}, \mathcal{F})$ into an $O(j)$-tree $\mathcal{J}$ such that the two graphs are $O(1)$-embeddable into each other. In the following, we first describe the construction and we formally prove that the resulting $O(j)$-tree $\mathcal{J}$ and the given graph $\mathcal{H}(\mathcal{T}, \mathcal{F})$ are $O(1)$-embeddable into each other. We then show how to efficiently construct $\mathcal{J}$ in a distributed way.

Assume that we are given a spanning tree $\mathcal{T}$ of $\mathcal{G}$ and a graph $\mathcal{H}(\mathcal{T}, \mathcal{F})$ which is constructed as described above. Consider the forest induced by the edges in $\mathcal{T} \setminus \mathcal{F}$.

Let $P_1 \subseteq \mathcal{V}$ be the set of clusters of $\mathcal{T} \setminus \mathcal{F}$ which are incident to one of the deleted tree edges in $\mathcal{F}$. We call P_1 the *primary portals* of $\mathcal{T} \setminus \mathcal{F}$. Given P_1, we define the skeleton $\mathcal{S}_{\mathcal{T} \setminus \mathcal{F}}$ of $\mathcal{T} \setminus \mathcal{F}$ as follows. $\mathcal{S}_{\mathcal{T} \setminus \mathcal{F}}$ is obtained from $\mathcal{T}$ by repeatedly deleting nonportal clusters of degree 1 until all remaining clusters are either in P_1 or they have degree at least 2. Denote by P_2 all clusters of degree larger than 2 that are not primary portals; P_2 are the *secondary portals*. The set of all portal clusters now is $P := P_1 \cup P_2$. The skeleton $\mathcal{S}_{\mathcal{T} \setminus \mathcal{F}}$ is thus a forest consisting of a set of portals and paths connecting them, where all inner clusters of these paths have degree 2.

Given the skeleton $\mathcal{S}_{\mathcal{T} \setminus \mathcal{F}}$, let $\mathcal{P}$ be one of these portal-connecting paths. In the last step, we remove the edge with the smallest capacity from each such $\mathcal{P}$. In doing so, we split the forest into trees so that each tree contains exactly one portal. It is straightforward to bound the number of resulting trees in terms of $\mathcal{F}$.

LEMMA 6.8. *Let $\mathcal{H}(\mathcal{T}, \mathcal{F}) \in \mathbb{H}[j]$, i.e., $|\mathcal{F}| \le j$. Then, in the above construction, the total number of portal nodes is less than $4j$.*

Proof. Clearly, $|P_1| \le 2|\mathcal{F}| \le 2j$. As when computing the skeleton $\mathcal{S}_{\mathcal{T} \setminus \mathcal{F}}$, nonportal clusters of degree 1 are successively removed, and we obtain a forest whose leaves are primary portals. As the sum of the degrees in an N-node forest is at most $2(N-1)$, the number of nodes of degree at least 3 is upper bounded by the number of leaves minus 2. We conclude that $|P_2| < |P_1| \le 2j$, and hence $|P| < 4j$. ∎

Finally, we identify each of the resulting trees with its portal and logically move all edges between different trees to the portals. For each edge $e = \{c, c'\} \in \mathcal{E}_{\mathcal{H}} \setminus (\mathcal{T} \setminus \mathcal{F})$ (i.e., each nontree edge of $\mathcal{H}(\mathcal{T}, \mathcal{F})$), we add a virtual edge of capacity $\mathrm{cap}_{\mathcal{G}}(e)$ between the portals of the trees containing c and c', respectively. Further, let $\mathcal{D}$ be the set of edges that were deleted from the paths of degree-2 clusters connecting portals in the skeleton. For every edge $e \in \mathcal{D}$, we add a virtual edge of capacity $\mathrm{cap}_{\mathcal{H}}(e) = \mathrm{cap}_{\mathcal{T}}(e)$ between the two portals that were connected by the path from which e was deleted.

Let us summarize this part of the construction; see Figure 3 for an example of a possible result. Starting from a forest $\mathcal{T} \setminus \mathcal{F}$, do as follows:

1. Define P_1 as the endpoints of edges in $\mathcal{F}$;
2. iteratively delete degree-1 clusters that are not in P_1 until this process halts;
3. define P_2 as the clusters retaining degree larger than 2 that are not in P_1 and set $P := P_1 \cup P_2$;
4. delete from each (maximal) path without clusters from P the edge $e \in \mathcal{D}$ of minimum capacity and replace it by an edge of the same capacity between the endpoints of the path; and
5. for each edge $e \in \mathcal{E}_{\mathcal{H}}$ between different components of $\mathcal{T} \setminus (\mathcal{F} \cup \mathcal{D})$, add an edge of the same capacity between the unique portals in these components.

Hence, the resulting graph consists of the forest induced by $\mathcal{T} \setminus (\mathcal{F} \setminus \mathcal{D})$ and (possibly parallel) edges between the unique portals of the trees of the forest. By Lemma 6.8, the number of such portals is smaller than $4j$, implying that the resulting graph is a $4j$-tree. In the following, we denote this $4j$-tree by $\mathcal{J}$.

Mutual embeddability of $\mathcal{H}(\mathcal{T}, \mathcal{F})$ and $\mathcal{J}$. Before discussing how to efficiently construct (and represent) $\mathcal{J}$ in a distributed way, let us first show that $\mathcal{H}(\mathcal{T}, \mathcal{F})$ and $\mathcal{J}$ are $O(1)$-embeddable into each other.

The proofs of the following two lemmas are very similar to the corresponding result by Madry [21]. However, since our j-tree construction slightly deviates from Madry's, the claims do not readily follow from any lemma in [21, 22].

LEMMA 6.9. *$\mathcal{H}(\mathcal{T}, \mathcal{F})$ is $O(1)$-embeddable into $\mathcal{J}$.*

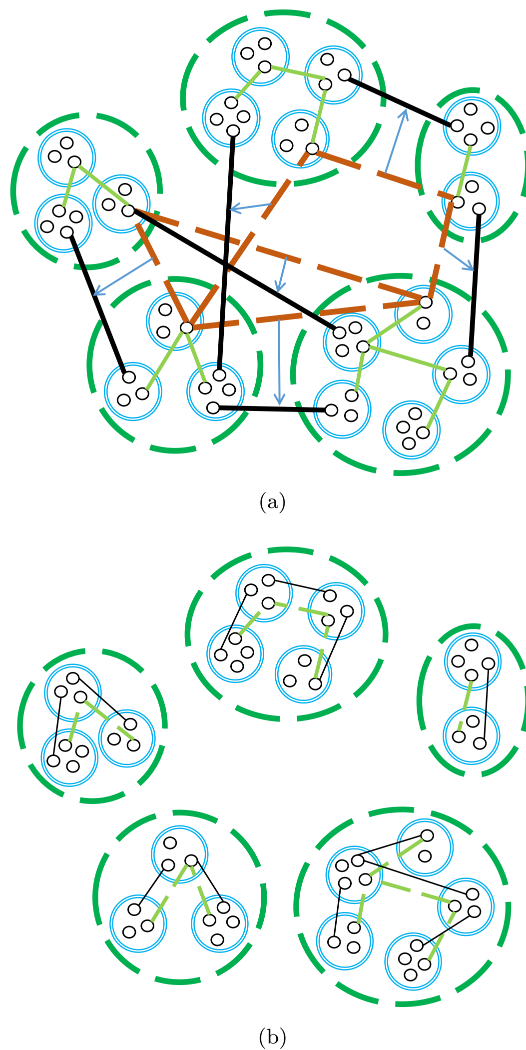
Proof. There are three types of edges of $\mathcal{H}(\mathcal{T}, \mathcal{F})$ to distinguish:

(a) edges in $(\mathcal{T} \setminus \mathcal{F}) \setminus \mathcal{D}$,
(b) edges in $\mathcal{D}$, and
(c) the remaining edges from $\mathcal{E}_{\mathcal{H}}$ connecting different trees of the forest $\mathcal{T} \setminus \mathcal{F}$.

Case (a) is the most straightforward, as all these edges are also present in $\mathcal{J}$ with the same capacity. Edges from $e \in \mathcal{D}$ were deleted from a path $\mathcal{P}$ connecting two portals in the skeleton. In $\mathcal{J}$, they can therefore be routed through the path $\mathcal{P}$ and the virtual edge with capacity $\mathrm{cap}_{\mathcal{T}}(e)$ connecting the portal nodes at the ends of $\mathcal{P}$. Because e is the lowest capacity edge of $\mathcal{P}$, this adds relative load at most 1 to each edge of $\mathcal{P}$.

Finally, let us consider one of the remaining edges $e \in \mathcal{E}_{\mathcal{H}}$. The edge $e = \{c_1, c_2\}$ connects two trees $T_1 \neq T_2$ of $\mathcal{J}$. Let us assume that $c_1 \in T_1$ and $c_2 \in T_2$. When routing from c_1 to c_2, we follow

1. the path from c_1 in T_1 to the first skeleton cluster $s_1 \in T_1$ on the path to the (unique) portal $p_1 \in T_1 \cap P$,
2. the skeleton path from s_1 to p_1,
3. the virtual edge corresponding to e between p_1 and the (unique) portal $p_2 \in T_2 \cap P$,
4. the skeleton path from p_2 to the last skeleton cluster $s_2 \in T_2$ when going from p_2 to c_2 in T_2, and



(a)

(b)

Fig. 3. *An example j-tree at the second level of recursion. In* (a)*, dashed circles indicate the components of the forest of this j-tree, which are each made of a number of* 1*-clusters, indicated by thick circles. Edges inside* 1*-clusters are not shown. Solid thin edges indicate virtual edges of level* 1 *that became edges of the level-*2 *forest. For each of these thin edges, there is a real edge between some two nodes of the same two level-*1 *components (not shown). Dashed edges represent (virtual) core edges, and they are mapped (by arrows) to corresponding real edges (thick lines). In* (b)*, real edges related to the virtual forest edges are represented as solid thin edges between two nodes of the same connected component.*

5. the path from s_2 to c_2 in T_2.

Let us compare the path from c_1 via s_1 to p_1 with the path on which e is routed on the spanning tree $\mathcal{T}$. The part from c_1 to s_1 is also used when routing in $\mathcal{T}$. If from s_1 we follow the same direction to p_1 as in $\mathcal{T}$, the two paths are, in fact, identical up the point when we reach p_1. In this case, the capacities on this path suffice by construction, as we defined $\text{cap}_{\mathcal{T}}(e') = |\boldsymbol{f}'(e')|$. Let us therefore consider the case in which we set out in the opposite direction from s_1 on the skeleton path $\mathcal{P} \ni s_1$ connecting p_1 to some other portal p_2 than we would in $\mathcal{T}$. In that case, routing on

$\mathcal{T}$ would cross the edge $e' \in \mathcal{D}$ that was deleted from $\mathcal{P}$. Because e' is the edge from $\mathcal{P}$ of smallest capacity, the contribution to the relative load of all edges crossed on $\mathcal{P}$ is upper bounded by the relative load contributed to e' when routing on $\mathcal{T}$. Again, summing over all edges from $\mathcal{E}_{\mathcal{H}}$ falling under case (c), this may increase their total relative loads only by an additive 1. Trivially, the third step causes relative load 1 on the virtual edge corresponding to e, since it is not used for routing any other edge. Reasoning symmetrically for Steps 4 and 5, we can conclude that embedding $\mathcal{H}(\mathcal{T},\mathcal{F})$ into $\mathcal{J}$ leads to constant relative load on all edges. ◻

LEMMA 6.10. *$\mathcal{J}$ is $O(1)$-embeddable into $\mathcal{H}(\mathcal{T},\mathcal{F})$.*

Proof. All the edges of the trees of $\mathcal{J}$ are also present in $\mathcal{H}(\mathcal{T},\mathcal{F})$ with the same capacity; they are hence straightforward to embed. Let us therefore consider the virtual edges connecting the portals of $\mathcal{J}$. There are two types of virtual edges—the ones representing edges from $\mathcal{D}$ and those that correspond to the nontree edges of $\mathcal{H}(\mathcal{T},\mathcal{F})$. We first have a look at a virtual edge e corresponding to an edge $e' \in \mathcal{D}$. The edge e is routed by following the path from which e' was removed. Since the capacity of each edge on the path is at least the capacity of e, this contributes at most 1 to the relative load of each edge.

Now consider a virtual edge $e = \{c, c'\}$ corresponding to an edge $e' \in \mathcal{E}_{\mathcal{H}}$ of $\mathcal{H}(\mathcal{F},\mathcal{T})$. The edge e is routed on the trees of $\mathcal{J}$ the clusters c and c' reside in and via e'. The latter causes relative load 1, as e and e' have the same capacity and no other edge uses e'. Similarly to the embedding of $\mathcal{H}(\mathcal{T},\mathcal{F})$ into $\mathcal{J}$, the tree parts of the routing path are subpaths of the path between c and c' in $\mathcal{T}$ and thus will not cause more than additive relative load 1 when summing over all edges of this type to embed. If we diverge from this path, this is because an edge from $\mathcal{D}$ lies on the routing path in $\mathcal{T}$; analogously to Lemma 6.9, following the skeleton path from which it was deleted to the respective portal increases the maximum relative load by at most an additional 1. ◻

Distributed implementation. Let us now move on to the distributed implementation of the above $4j$-tree construction. Recall that because $\mathcal{F}$ includes the random set of edges $\mathcal{R}$, by Lemma 6.5, w.h.p., all trees in $\mathcal{T} \setminus \mathcal{F}$ have depth $\tilde{O}(\sqrt{n})$. With this in mind, constructing the skeleton is fairly simple.

LEMMA 6.11. *Given are a spanning tree $\mathcal{T}$ of a distributed cluster graph and the set of tree edges $\mathcal{F}$ as computed above. We can determine the skeleton $\mathcal{S}_{\mathcal{T}\setminus\mathcal{F}}$, the set of portals P, and the set of edges $\mathcal{D}$ (i.e., for $\{c, c'\}_{uv} \in \mathcal{D}$, u and v will learn this) in time $O(\sqrt{n}\log n)$ in the CONGEST model on the underlying network graph. In the same time, we can also orient the trees rooted at the portals.*

Proof. Without loss of generality, consider a single tree T of the forest $\mathcal{T} \setminus \mathcal{F}$. By Lemma 6.5, the induced tree in G has depth $\tilde{O}(\sqrt{n})$. Perform the following steps:

- For each edge $e \in \mathcal{F}$, its incident clusters learn[10] that they are primary portals, i.e., are in P_1.
- Iteratively mark nonportal clusters with at most 1 marked neighboring cluster until this process stops. Unmarked clusters are in the skeleton.
- Unmarked clusters with more than two unmarked neighboring portals are secondary portals.
- The skeleton paths connecting portals find a minimum capacity edge and add

[10]A cluster for which edges to children are in $\mathcal{F}$ may not "know" about its incident edges in $\mathcal{F}$ as a whole, but determining whether there is at least one is trivial.

it to $\mathcal{D}$.

- Each tree of $\mathcal{T} \setminus (\mathcal{F} \cup \mathcal{D})$ is rooted at its unique portal, whose identifier is made known to all nodes in the induced tree in G (together with clusters' spanning trees).
- These identifiers are exchanged with all neighbors in G.

From the gathered information, for each edge $\{c, c'\}_{uv} \in \mathcal{J}$, u and v now can determine its membership and its capacity in $\mathcal{J}$. Observe that the bound of $\tilde{O}(\sqrt{n})$ on the depth of the spanning trees of G leveraged for communication in the above construction implies that all the above steps can be completed in $\tilde{O}(\sqrt{n})$ rounds, which completes the proof. ∎

The trees rooted at the portals now induce the clusters of the new cluster graph.

COROLLARY 6.12. *Given a graph $\mathcal{H}(\mathcal{T}, \mathcal{F}) \in \mathbb{H}[j/4]$ as computed above on a cluster graph whose clusters' spanning trees have maximum depth d, there is an $\tilde{O}(D + d + \sqrt{n})$-round distributed algorithm to compute*

- *a cluster graph whose clusters' spanning trees have depth $d + \tilde{O}(\sqrt{n})$; and*
- *a j-tree $\mathcal{J}$ on this cluster graph, i.e., for each edge $e \in \mathcal{J}$, there is a corresponding graph edge $\{u, v\} \in E$ whose constituent nodes know that $e \in \mathcal{J}$ as well as $\operatorname{cap}_{\mathcal{J}}(e)$; such that*
- *$\mathcal{H}(\mathcal{T}, \mathcal{F})$ is 1-embeddable into $\mathcal{J}$ and $\mathcal{J}$ is $O(1)$-embeddable into $\mathcal{H}(\mathcal{T}, \mathcal{F})$; and*
- *the new clusters are induced by the tree components of $\mathcal{J}$.*

Proof. This readily follows from Lemmas 6.5, 6.8, 6.9, 6.10, and 6.11. The only thing left to note is that clusters can learn the number of nodes they contain by a simple convergecast and broadcast operation on their spanning trees. ∎

6.4. Sampling from the recursively constructed distribution. We now have all pieces in place to efficiently sample from a distribution similar to Sherman's in a distributed fashion. The difference is that Theorem 3.2 and thus Lemma 6.7 merely give $\alpha \in 2^{O(\sqrt{\log n \log\log n})}$, implying that we must use fewer levels of recursion to ensure that the final approximation guarantee of the congestion approximator will remain in $n^{o(1)}$.

THEOREM 6.13. *With high probability, within $(\sqrt{n} + D)n^{o(1)}$ rounds of the* CONGEST *model, we can sample a tree $\mathcal{T}$ from a distribution of $n^{1+o(1)}$ (virtual) rooted spanning trees on G with the following properties.*

- *For any cut of G of capacity C, the capacity of the cut in $\mathcal{T}$ is at least C.*
- *For any cut of G of capacity C, the expected capacity of the cut in $\mathcal{T}$ is at most αC, where $\alpha \in n^{o(1)}$.*
- *The distributed representation of $\mathcal{T}$ is given by a hierarchy of cluster graphs $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i, \mathcal{L}_i, \mathfrak{T}_i, \psi_i)$, $i \in \{0, \dots, i_0\}$, $i_0 \in o(\log n)$, on network graph G, with the following properties.*
  - *The spanning trees of the clusters of $\mathcal{G}_i$ have depth $\tilde{O}(\sqrt{n})$.*
  - *$|\mathcal{V}_{i_0}| = n^{1/2+o(1)}$.*
  - *$\mathcal{G}_i$ is the (rooted) tree resulting from $\mathcal{T}$ by contracting the clusters of $\mathcal{G}_i$.*
  - *For $i > 0$, $\mathcal{G}_i$ is also a cluster graph on network graph $\mathcal{G}_{i-1}$.*
  - *For $i > 0$, each cluster $c_i \in \mathcal{V}_i$ of $\mathcal{G}_i$, interpreted as cluster graph on $\mathcal{G}_{i-1}$, contains a unique* portal cluster *$p(c_i) \in \mathcal{V}_{i-1}$ of $\mathcal{G}_{i-1}$ that is incident*[11] *to all edges of $\mathcal{G}_i$ containing c_i. That is, $\mathcal{G}_{i-1}$ is a $|\mathcal{V}_i|$-tree with core*

[11] Note that the corresponding physical edges in G may still connect to different subclusters of c_i.

$p(\mathcal{V}_i)$.

Proof. In the following, we will use w.h.p. statements as if they were deterministic; the result then follows by taking the union bound over all (polynomially many in n) such statements we use.

Set $\beta := 2^{\log^{3/4} n}$. To start the recursion, we will use G as a cluster graph of itself. Formally, $\tilde{\mathcal{G}}_0 := (V, E, V, \{(\{v\}, \emptyset)\}_{v \in V}, \mathrm{id})$, where id is the identity function. We execute the following algorithm until it terminates:

1. Sparsify $\tilde{\mathcal{G}}_{i-1}$ using Lemma 7.1 for some fixed constant ε, e.g., $\varepsilon = 1/2$. This takes $(\sqrt{n} + D)n^{o(1)}$ rounds. Multiply all edge capacities by $1/(1-\varepsilon)$ (so $\tilde{\mathcal{G}}_{i-1}$ can be 1-embedded into the sparser graph).
2. If $|\mathcal{V}_{i-1}| \notin \omega(\sqrt{n}\beta/\log n)$, set $i_0 := i$ and stop. This takes $O(D)$ rounds by communicating over a BFS tree of G.
3. Apply Lemma 6.7 for $j = |\mathcal{V}_{i-1}|/(4\beta)$ to the sparsified cluster graph; by the previous step, this choice of j is feasible. As $|\mathcal{V}_{i-1}|/j = 4\beta \in n^{o(1)}$, constructing the distribution requires $(\sqrt{n} + D)n^{o(1)}$ rounds in total.
4. Sample a cluster graph from the distribution. This is done in $O(D)$ rounds, letting some node broadcast $O(\log n)$ random bits over a BFS tree.
5. Apply Corollary 6.12 to extract a $|\mathcal{V}_{i-1}|/\beta$-tree of $\mathcal{G}_{i-1}$. The corollary also yields a cluster graph $\tilde{\mathcal{G}}_i$ (which is also a cluster graph on network graph $\mathcal{G}_{i-1}$) so that each of its clusters c_i contains exactly one portal cluster $p(c_i)$ of the $|\mathcal{V}_{i-1}|/\beta$-tree on $\mathcal{G}_{i-1}$. This step completes in $\tilde{O}(\sqrt{n} + D)$ rounds: there are fewer than $\log_\beta \sqrt{n} \ll \log n$ iterations of the overall construction, as $|\mathcal{V}_i| \leq |\mathcal{V}_{i-1}|/\beta$, implying that $d \in \tilde{O}(\sqrt{n})$ for each application of Corollary 6.12.
6. Recurse on $\tilde{\mathcal{G}}_i$, i.e., set $i := i + 1$ and go back to Step 1.

When the above construction halts, we have that $|\mathcal{V}_{i_0-1}| = O(\sqrt{n}\beta) = n^{1/2+o(1)}$. Thus, we can make the (sparsified) cluster graph $|\mathcal{G}_{i_0-1}|$ known to all nodes in $(\sqrt{n} + D)n^{o(1)}$ rounds via a BFS tree of G. We then continue the construction locally without controlling the size of components, which removes the constraint on j when applying Lemma 6.11, until the core becomes empty, i.e., we construct a tree.[12] We collapse the cluster graph hierarchy for all locally performed iterations $i \geq i_0$, which defines the tree $\mathcal{G}_{i_0}$ on clusters $\mathcal{C}_{i_0}$ (this is feasible as each $\mathcal{G}_i$, $i > 0$, is also a cluster graph on network graph $\mathcal{G}_{i-1}$).

This completes the description of the algorithm. Summing up the running times of the individual steps and using that $i_0 = o(\log n)$, we conclude that the construction takes $(\sqrt{n} + D)n^{o(1)}$ rounds. The construction also maintained the stated structural properties of the cluster hierarchy. Hence, it remains to show that (i) we sampled from a distribution of $n^{1+o(1)}$ trees and (ii) the stated cut approximation properties are satisfied.

Showing these properties now is straightforward. In each step $i > 0$ of the recursion, by Lemma 6.7 we constructed a distribution on $\tilde{O}(\beta)$ $|\mathcal{V}_{i-1}|$-trees. The total number of recursive steps (including the local ones) is bounded by $\lceil \log_\beta n \rceil = O(\log^{1/4} n)$, as $|\mathcal{V}_i| \leq |\mathcal{V}_{i-1}|/\beta$ for each $i > 0$. On each level of recursion, we compute a distribution on $2^{O(\sqrt{\log |\mathcal{V}_i| \log\log |\mathcal{V}_i|})}\beta \leq 2^{O(\sqrt{\log n \log\log n})}\beta$ graphs. Hence, the total number of virtual trees in the (implicit) distribution of virtual trees from which we sampled is bounded by

$$\left(2^{O(\sqrt{\log n \log\log n})}\beta\right)^{\lceil \log_\beta n \rceil} = n \cdot 2^{O(\sqrt{\log n \log\log n} \log^{1/4} n)} = n^{1+o(1)}.$$

[12] This is essentially Sherman's construction on the small constructed cluster graph.

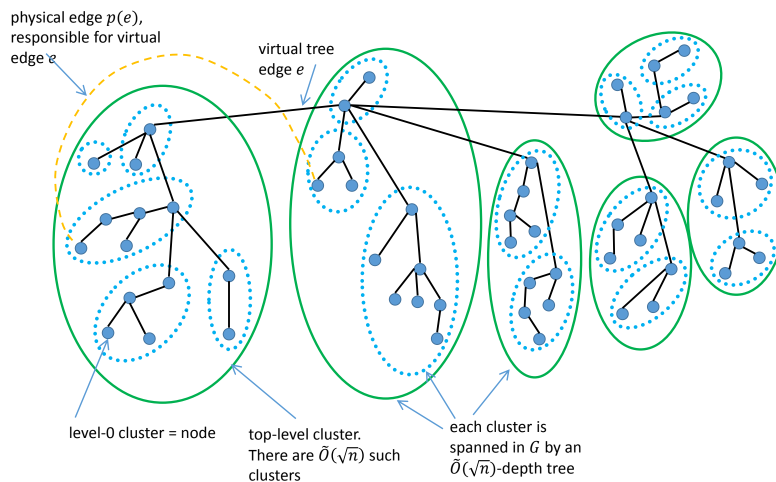


FIG. 4. *Hierarchical cluster decomposition of a virtual tree $\mathcal{T} \in \mathbb{T}$. Continuous edges are virtual tree edges, which are represented by a physical edge connecting the top-level clusters they connect (the dotted edge $p(e)$ corresponds to the edge labeled e). Each cluster is spanned by a tree in G of depth $\tilde{O}(\sqrt{n})$, which is not shown.*

Consider a cut of G of capacity C. By the properties of decompositions and the fact that we multiplied capacities by $1/(1-\varepsilon)$ whenever we sparsified, G is 1-embeddable into any of the trees we might construct, implying that the corresponding cut of the sampled tree has capacity at least C. As in each step, we (i) apply a $(1+\varepsilon)$-sparsifier and multiply capacities by $1/(1-\varepsilon)$ for constant ε, (ii) construct a $(2^{O(\sqrt{\log n \log\log n})}, \mathbb{H})$-decomposition (for some family $\mathbb{H}$) from which we sample, and (iii) transform the resulting graph into a j-tree which can be $O(1)$-embedded into the graph from which it is constructed; we overestimate the capacity of a given cut by an expected factor of $2^{O(\sqrt{\log n \log\log n})} \cdot O(1) = 2^{O(\sqrt{\log n \log\log n})}$ in each step. Using that this bound is uniform and the randomness on each level of recursion is independent, it follows that the expected capacity of a cut of G of capacity C in the sampled virtual tree is bounded by

$$\left(2^{O(\sqrt{\log n \log\log n})}\right)^{\lceil \log_\beta n \rceil} = 2^{O(\sqrt{\log n \log\log n} \log^{1/4} n)} = n^{o(1)}. \qquad \square$$

6.5. Congestion approximation. Our congestion approximator R is defined by the edge-induced cuts of a sample $\mathbb{T}$ of virtual trees $\mathcal{T}$ constructed according to Theorem 6.13. As stated in the theorem, the trees are represented distributively by a hierarchy of *cluster graphs* (see Figure 4 for an illustration and recall the formal definition of cluster graphs from section 5). Intuitively, a cluster graph partitions the nodes into clusters, each of which has a spanning tree rooted at a leader, and a collection of edges between clusters that are represented by corresponding graph edges between some nodes of the clusters they connect.

The first two properties of each $\mathcal{T}$ stated in the theorem imply that we can use them to construct a good congestion approximator R. More precisely, Lemma 3.3 implies the following corollary.

COROLLARY 6.14. *Sampling a collection $\mathbb{T}$ of $O(\log n)$ virtual trees given by Theorem* 6.13 *and using them as congestion approximator R in the way specified in sec-*

tion 3.2 *implies that the total number of iterations of Algorithm* 2 *is* $n^{o(1)}$.

All that remains now is to show that the distributed representation of each sampled $\mathcal{T} \in \mathbb{T}$ allows us to simulate a convergecast and a downcast on $\mathcal{T}$ in $(\sqrt{n}+D)n^{o(1)}$ rounds: then we can implement the key subroutines (1) and (2) (i.e., compute $\boldsymbol{y}$ and $\boldsymbol{\pi}$) outlined in section 3.2 with this time complexity, and by Corollary 6.14 the total number of rounds of the computation is bounded by $(\sqrt{n} + D)n^{o(1)}$.

Fortunately, the recursive structure of the decomposition is very specific. The cluster graphs of the different levels of recursion are nested; i.e., the clusters of the $(i-1)$th level of recursion are subdivisions of the clusters of the ith level. What is more, each cluster is a subtree of the virtual tree and is spanned by a tree of depth $\tilde{O}(\sqrt{n})$ in G (cf. Figure 4). Hence, while the physical graph edges representing the virtual tree edges are between arbitrary nodes within the clusters they connect, we can (i) identify each cluster on each hierarchy level with the root of the subtree induced by its nodes, (ii) handle such subtrees recursively (both for convergecasts and downcasts), (iii) on each level of recursion but the last, perform the relevant communication by broadcasting or upcasting on the underlying cluster spanning trees in G of depth $\tilde{O}(\sqrt{n})$, and (iv) communicate over a BFS tree of G on the final level of recursion, where merely $n^{1/2+o(1)}$ clusters/nodes of the virtual tree remain.

COROLLARY 6.15. *On each virtual tree* $\mathcal{T} \in \mathbb{T}$, *we can simulate convergecast and upcast operations in* $\tilde{O}(\sqrt{n} + D)$ *rounds.*

Theorem 1.1 now follows from Sherman's results on the number of iterations of the gradient descent algorithm [32], the discussion in section 3.2, and Corollaries 6.14 and 6.15.

7. Distributed construction of cut sparsifiers. In this section we specify the distributed implementation of cut sparsifiers. We start with the following lemma.

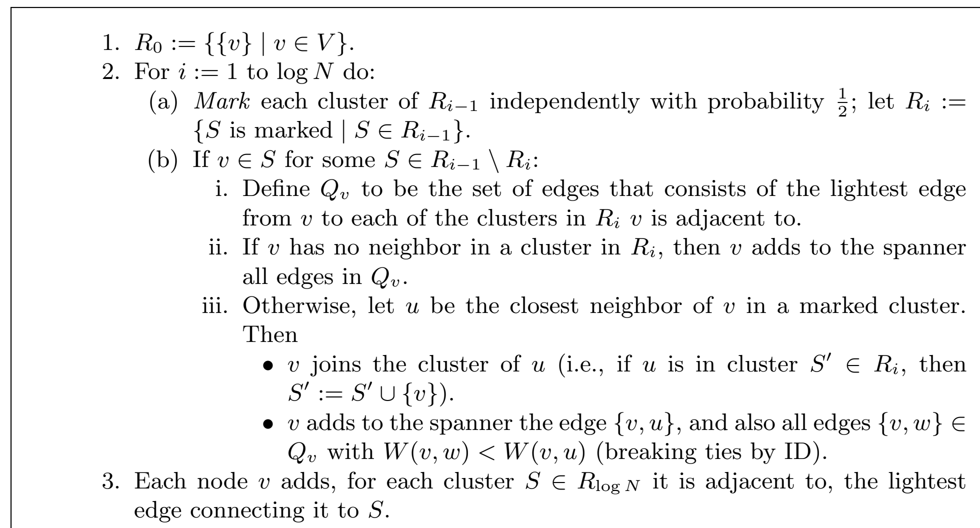
LEMMA 7.1. *In a weighted* N*-node distributed cluster graph of size* n, *for any* $\varepsilon > 0$, *it is possible to compute a* $(1+\varepsilon)$*-cut sparsifier with* $O(N \cdot (\varepsilon^{-1} \cdot \log N)^{O(1)})$ *edges (w.h.p.) in the* CONGEST *model in time* $O\big((D + \sqrt{n}) \cdot (\varepsilon^{-1} \cdot \log N)^{O(1)}\big)$.

Proof. We prove the lemma using the algorithm PARALLELSPARSIFY of Koutis [18] and then orient the edges. Koutis's algorithm relies on the $O(\log n)$-stretch spanner construction algorithm of Baswana and Sen [9], which we henceforth refer to as BS. See Figure 5 for a description of the BS algorithm.

We start by showing how to emulate a step of node v in BS by a depth-d tree (which we shall denote by T_v) in $O(d + \log N)$ time, w.h.p. We may assume w.l.o.g. the existence of a root in each tree (because we can select one in $O(d)$ time); Step 2a is carried out by the root and the result is broadcast over the tree. For Step 2b, we note that w.h.p., $|Q_v| = O(\log N)$ and hence making it known to all nodes of T_v takes $O(d + \log N)$ time using standard convergecast-broadcast. Step 3 is straightforward given that each node knows its cluster.

Next, given a tree $T(v)$ for each node, we assume that the depth of all trees is at least $c \log N$ for some appropriate constant $c > 0$. If this assumption does not hold, we extend $T(v)$ with a dummy path (i.e., pretend as if a path of nodes is attached to one of the leaves): clearly $T(v)$ can emulate the extended tree without any slowdown. However, this extension may increase the number of nodes by an $O(\log N)$ factor. Now, under this assumption and the emulation above, we may apply Lemma 5.3 to conclude that BS can be executed in time $O((D + \sqrt{N \log N}) \log N)$.

Going back to the algorithm of Koutis [18], we note that it consists of $(\log n/\varepsilon)^{O(1)}$



1. $R_0 := \{\{v\} \mid v \in V\}$.
2. For $i := 1$ to $\log N$ do:
   (a) *Mark* each cluster of R_{i-1} independently with probability $\frac{1}{2}$; let $R_i := \{S \text{ is marked} \mid S \in R_{i-1}\}$.
   (b) If $v \in S$ for some $S \in R_{i-1} \setminus R_i$:
      i. Define Q_v to be the set of edges that consists of the lightest edge from v to each of the clusters in R_i v is adjacent to.
      ii. If v has no neighbor in a cluster in R_i, then v adds to the spanner all edges in Q_v.
      iii. Otherwise, let u be the closest neighbor of v in a marked cluster. Then
         - v joins the cluster of u (i.e., if u is in cluster $S' \in R_i$, then $S' := S' \cup \{v\}$).
         - v adds to the spanner the edge $\{v, u\}$, and also all edges $\{v, w\} \in Q_v$ with $W(v, w) < W(v, u)$ (breaking ties by ID).
3. Each node v adds, for each cluster $S \in R_{\log N}$ it is adjacent to, the lightest edge connecting it to S.

Fig. 5. *The BS algorithm for $O(\log N)$ spanner construction given an N-node weighted graph $G = (V, E, W)$. The output is a subset of E.*

invocations of BS, and some independent random selection and reweighting of edges. The former is discussed above, and the latter is trivial to emulate locally. ∎

8. Distributed construction of low average-stretch spanning trees.

Theorem 8.1 (Theorem 3.2 restated and rephrased). *Suppose H is an N-node multigraph obtained from G by assigning arbitrary edge lengths in $2^{n^{o(1)}}$ to the edges of G (known to incident nodes) and performing an arbitrary sequence of contractions. Then we can compute a rooted spanning tree of H of expected stretch $2^{O(\sqrt{\log n \log\log n})}$ within $(\sqrt{n} + D)n^{o(1)}$ rounds, where the edges of the tree in H and their orientation is locally known to the endpoints of the corresponding edges in G.*

Proof. We follow [11]: the high-level algorithm is by Alon et al. [3], which uses Algorithm Partition (of [11]) for unweighted graphs. We describe the algorithm bottom-up. The main component in Algorithm Partition is Algorithm SplitGraph, reproduced in Figure 6. Algorithm SplitGraph assumes that H is given as an N-node cluster graph of the underlying graph G. In line 2(a) of Algorithm SplitGraph, we need to select a fixed a random subset S^t of V^t of a fixed size x_t. This can be done first assigning each node of V^t a random number from a sufficiently large domain such that the numbers are unique w.h.p. Then, the first x_t nodes can be determined in time $O(D \log N)$ in the CONGEST model on the underlying graph G by applying a standard distributed selection algorithm that can be run on top of a global BFS tree of G [19, 25].

The basic action of Algorithm SplitGraph is growing BFS trees, an action in which emulating a single node by a tree is trivial. In SplitGraph we may have contending BFS growths, but note that if two or more BFS traversals collide, only the winning ID needs to proceed, and hence there are no collisions because no edge needs to carry more than a single BFS traversal in each direction. Regarding the tree construction, we first note that the BFS growth naturally creates a spanning tree for each cluster. Moreover, we can make all nodes know the complete path to the root of their respective cluster in additional $O(\rho)$ steps, by letting each node send its ith ancestor to all its children in round i. Except for the sampling step in line 2(a), the running time of Algorithm SplitGraph is $O(\rho \log N)$ in the bounded-space CONGEST model, because

1. $H^1 = (V^1, E^1) := H$; $\mathcal{C} := \emptyset$.
2. For $t = 1$ to $2 \log N$ do:
   (a) Let S^t be a random subset of V^t of $x_t := 12 \frac{2^{t/2}}{N} |V^t|$ nodes; if V^t is smaller than $\frac{N}{12 \cdot 2^{t/2}}$, $S^t := V^t$.
   (b) $\mathcal{C} := \mathcal{C} \cup \big\{ \{s\} \mid s \in S^t \big\}$.
   (c) Each $s \in S^t$ draws a random delay δ_s^t uniformly from $[0, \lfloor \rho/(2 \log N) \rfloor]$.
   (d) Each $s \in S^t$ waits δ_s^t rounds and then initiates a BFS for $\rho(1 - \frac{t-1}{2 \log N}) - \delta_s^t$ rounds in H^t.
   (e) A node covered by a BFS is added to cluster C_s, where s is the source of the first BFS to visit it, breaking ties by ID.
   (f) $V^{t+1} := V \setminus \{v \mid v \in C \text{ for some } C \in \mathcal{C}\}$; $H^{t+1} := H^t[V^{t+1}]$.

FIG. 6. *Algorithm SplitGraph. The input is an unweighted N-node multigraph $H = (V, E)$ and a target radius ρ.*

each iteration takes $O(\rho)$ rounds and there are $O(\log N)$ iterations. Therefore, using Lemma 5.3, we conclude that we can run SplitGraph (including line 2(a)) in time $O(\rho \log^2 N (D + \sqrt{N}))$ in the CONGEST model.

Algorithm SplitGraph is called by Algorithm Partition, whose input is an unweighted graph with an arbitrary partition of the edges into K classes. Algorithm Partition applies Algorithm SplitGraph disregarding classes, and then checks whether there exists a class where too many edges were split in different clusters. If there is such an oversplit class, the algorithm is restarted. We can implement each checking and restart in the CONGEST model in $O(D + K)$ time using a global BFS tree. Since the number of restarts is bounded by $O(\log N)$ w.h.p. [11], and in our implementation we shall have $K = O(\sqrt{N})$, the overall time for running Algorithm Partition is $O(\rho \log^3 N (D + \sqrt{N}))$ in the CONGEST model.

The outermost algorithm is the one by Alon et al. [3], whose input is a *weighted* graph. The algorithm first partitions the edges into $O(\sqrt{\log N})$ classes by weight, where class E_i contains all edges whose weight is in $[z^{i-1}, z^i)$ for a certain value $z = \tilde{\Theta}(2^{\sqrt{6 \log N \cdot \log \log N}})$. Then the algorithm proceeds in iterations until the graph is a single node, where iteration j is as follows.

1. Call Algorithm Partition with edges $E_1, \ldots, E_j$ and target radius $\rho = z/4$. Obtain clusters $\{C_i\}$.
2. Output a BFS tree for each cluster C_i.
3. Contract each resulting cluster C_i to a single node. Remove all self loops, but leave parallel edges in place. The resulting multigraph, augmented with edge class E_{j+1}, is the input to iteration $j + 1$.

For the distributed implementation, note that edge contraction is trivial given that the endpoints know the identity of the cluster they belong to, and that edge classification is purely local given z (which can be communicated to all in $O(D)$ time units). It can be shown [11] that w.h.p., the number of iterations is $O(\log \Delta / \sqrt{\log N \log \log N})$, and hence the running time of the algorithm is $O(\rho \log \Delta \log^{O(1)} N (D + \sqrt{N})) = \log \Delta \cdot 2^{O(\sqrt{\log N \log \log N})}$ because $\rho = z/4 = \tilde{\Theta}(2^{\sqrt{6 \log N \cdot \log \log N}})$.

The claimed stretch follows from [11]. ◻

9. Conclusion. In this paper we have demonstrated that with the help of randomization, a network with capacitated links can compute the value of the maximal flow between two nodes to any desired accuracy in time close to the worst-case lower bound of $\tilde{\Omega}(\sqrt{n} + D)$ while using only small messages. Computing maximal flow has been a cornerstone of network optimization for more than half a century, and in many

cases it is given in contexts which are naturally distributed, which makes us believe that our result may be used to build more advanced distributed systems. However, our algorithm is not very simple, and a few questions present themselves immediately.

- Can the complexity be reduced to $\tilde{O}(\sqrt{n}+D)$?
- Is there a "competitive" algorithm for maximal flow, i.e., an algorithm whose running time is related to the network structure, so that it runs quickly on "easy" instances and slowly on harder ones?
- What is the complexity of a deterministic solution to max flow? We note that classical max-flow algorithms are deterministic, but they do not seem to parallelize readily.

More interestingly, consider using the link capacities directly: Suppose that a link of capacity c can transfer c bits per time step (analogously to water pipes). Does this model allow for faster computation of the maximal flow?

Appendix A. Coping with general capacities. In this appendix we consider the case that edge capacities may be nonpolynomial (in n) or nonintegral.

First, note that we may assume without any loss of generality that all edge weights are at least 1; otherwise we can normalize by dividing by $\min\{\text{cap}(e) \mid e \in E\}$. Next, we note the following straightforward consequence of the max-flow min-cut theorem.

COROLLARY A.1. *Denote by $G_{>c}$ the graph induced by all edges of capacity more than c. Then the following hold:*

1. *If s and t are connected in $G_{>c}$, deleting all edges of capacity smaller than $\varepsilon c/n^2$ decreases the value of a max flow by at most factor $1+O(\varepsilon)$.*
2. *If s and t are not connected in $G_{>c}$, setting all edge capacities to $\max\{\text{cap}(e), n^2 c\}$ decreases the value of a max flow by at most factor $1+O(\varepsilon)$.*

Proof. To see the first claim, observe that a minimum s-t cut has at least capacity c, even after deleting all edges of capacity at most c. As any cut contains fewer than n^2 edges, deleting all edges of capacity smaller than $\varepsilon c/n^2$ reduces its capacity by less than εc. By the max-flow min-cut theorem, the claim follows.

Concerning the second claim, observe that a minimum s-t cut has capacity smaller than $n^2 c$ before modifying the capacities. Capping capacities at $\lceil n^2 c \rceil$ does not decrease the capacity of any cut below this threshold, implying the claim by the max-flow min-cut theorem. ∎

Using this corollary and binary search to approximate the threshold c at which s and t become disconnected in $G_{>c}$, we can reduce to edge weights from a polynomial range.

LEMMA A.2. *If edge capacities are from $[1, C] \subseteq \mathbb{R}$, for any $\varepsilon > 0$ we can reduce to the case of edge weights $[1, O(n^6/\varepsilon)]$ in $\tilde{O}((\sqrt{n}+D)\log C)$ rounds, at the expense of factor $1+O(\varepsilon)$ in approximation.*

Proof. We perform a binary search to approximate the threshold c_0 at which $G_{>c}$ switches from connecting s and t to not connecting them, up to factor n. Such a threshold exists, as G is connected and $G_{>C}$ is empty. Clearly, $\lceil \log(C/n) \rceil \in O(\log C)$ iterations of binary search suffice to determine a suitable approximation $\hat{c} \in [c_0/n, c_0 n]$.

Thus, we need to show how, for a given $c \in \mathbb{R}$, to determine whether $G_{>c}$ connects s and t or not. Note that nodes can decide locally whether an edge is in $G_{>c}$ or not; i.e., we can trivially simulate any distributed algorithm in the CONGEST model on $G_{>c}$, without overhead. However, the diameter of (a component of) $G_{>c}$ may be

significantly larger than D, the diameter of G, so simple flooding could be too slow. Instead, we invoke the minimum spanning tree algorithm by Kutten and Peleg [20] on each component of the (unweighted) graph $G_{>c}$; as each node learns an identifier for its component in this algorithm, after completion we can check whether s and t are in the same component within D rounds by, e.g., flooding the respective identifiers for s and t through G. The algorithm by Kutten and Peleg has running time $\tilde{O}(\sqrt{n}+D)$, yielding the claimed round complexity.

Finally, we delete all edges with capacity smaller than $\varepsilon\hat{c}/n^3$, reduce all capacities larger than $n^3\hat{c}$ to this value, and then rescale all capacities by factor $n^3/(\varepsilon\hat{c})$ (this is reverted after the flow algorithm is complete by multiplying the output by $\varepsilon\hat{c}/n^3$). This requires no communication and, by Corollary A.1, decreases the value of a max flow by factor $1+O(\varepsilon)$. ∎

It remains to reduce to integral capacities.

LEMMA A.3. *If edge capacities are* $[1, C] \in \mathbb{R}$*, we can reduce this case to the one of integral capacities* $1, \ldots, O(C/\varepsilon)$ *at the expense of factor* $1+O(\varepsilon)$ *in approximation, for any choice of* $\varepsilon > 0$*.*

Proof. For each edge $e \in E$, replace $\text{cap}(e)$ by the capacity given by cutting off all but the $\lceil \log(1/\varepsilon) \rceil$ most significant bits of the mantissa of $\text{cap}(e)$ in floating point representation. This does not increase the value of a max flow and decreases it by at most factor $1+O(\varepsilon)$.

Next, multiply each edge capacity by $2^{\lceil \log(1/\varepsilon) \rceil}$, making it integral. The optimum flow value and corresponding flows scale accordingly. Thus, we can solve the resulting instance with edge capacities $1, \ldots, O(C/\varepsilon)$, scale down the output by factor $2^{\lceil \log(1/\varepsilon) \rceil}$, and obtain a $(1+O(\varepsilon))$-approximate flow for the original instance. ∎

REFERENCES

[1] I. Abraham, Y. Bartal, and O. Neiman, *Nearly tight low stretch spanning trees*, in Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS), IEEE, Washington, DC, 2008, pp. 781–790.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows*, Prentice–Hall, Engelwood Cliffs, NJ, 1993.

[3] N. Alon, R. M. Karp, D. Peleg, and D. West, *A graph-theoretic game and its application to the k-server problem*, SIAM J. Comput., 24 (1995), pp. 78–100, https://doi.org/10.1137/S0097539792224474.

[4] S. Arora, E. Hazan, and S. Kale, *The multiplicative weights update method: A meta-algorithm and applications*, Theory Comput., 8 (2012), pp. 121–164.

[5] B. Awerbuch, *Reducing complexities of the distributed max-flow and breadth-first-search algorithms by means of network synchronization*, Networks, 15 (1985), pp. 425–437.

[6] B. Awerbuch and R. Khandekar, *Stateless distributed gradient descent for positive linear programs*, SIAM J. Comput., 38 (2009), pp. 2468–2486, https://doi.org/10.1137/080717651.

[7] B. Awerbuch, R. Khandekar, and S. Rao, *Distributed algorithms for multicommodity flow problems via approximate steepest descent framework*, ACM Trans. Algorithms, 9 (2012), 3.

[8] B. Awerbuch and T. Leighton, *Improved approximation algorithms for the multi-commodity flow problem and local competitive routing for dynamic networks*, in Proceedings of the 26th Annual ACM Symposium on Theory of Computing, ACM, New York, 1994, pp. 487–496.

[9] S. Baswana and S. Sen, *A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs*, Random Structures Algorithms, 30 (2007), pp. 532–563.

[10] A. A. Benczúr and D. R. Karger, *Randomized approximation schemes for cuts and flows in capacitated graphs*, SIAM J. Comput., 44 (2015), pp. 290–319, https://doi.org/10.1137/070705970.

[11] G. E. Blelloch, A. Gupta, I. Koutis, G. L. Miller, R. Peng, and K. Tangwongsan, *Nearly-linear work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs*, Theory Comput. Syst., 55 (2014), pp. 521–554.

[12] P. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S.-H. Teng, *Electrical flows, Laplacian systems, and faster approximation of maximum flow in undirected graphs*, in Proceedings of the Annual ACM Symposium on Theory of Computing (STOC), ACM, New York, 2011, pp. 273–282.

[13] 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 Proceedings of the Annual ACM Symposium on Theory of Computing (STOC), ACM, New York, 2011, pp. 363–372.

[14] M. Ghaffari, A. Karrenbauer, C. Lenzen, and B. Patt-Shamir, *Near-optimal distributed maximum flow*, in Proceedings of the International Symposium on Principles of Distributed Computing (PODC), ACM, New York, 2015, pp. 81–90.

[15] A. V. Goldberg and S. Rao, *Beyond the flow decomposition barrier*, J. ACM, 45 (1998), pp. 783–797, https://doi.org/10.1145/290179.290181.

[16] A. V. Goldberg and R. E. Tarjan, *Efficient maximum flow algorithms*, Commun. ACM, 57 (2014), pp. 82–89, https://doi.org/10.1145/2628036.

[17] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford, *An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations*, in Proceedings of the 25th ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM, New York, SIAM, Philadelphia, 2014, pp. 217–226, https://doi.org/10.1137/1.9781611973402.16.

[18] I. Koutis, *Simple parallel and distributed algorithms for spectral graph sparsification*, in Proceedings of the Symposium on Parallel Algorithms and Architectures, ACM, New York, 2014, pp. 61–66.

[19] F. Kuhn, T. Locher, and R. Wattenhofer, *Tight bounds for distributed selection*, in Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), ACM, New York, 2007, pp. 145–153.

[20] S. Kutten and D. Peleg, *Fast distributed construction of k-dominating sets and applications*, in Proceedings of the International Symposium on Principles of Distributed Computing (PODC), ACM, New York, 1995, pp. 238–251.

[21] A. Madry, *Fast approximation algorithms for cut-based problems in undirected graphs*, in Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS), IEEE, Washington, DC, 2010, pp. 245–254.

[22] A. Madry, *Fast Approximation Algorithms for Cut-Based Problems in Undirected Graphs*, preprint, https://arxiv.org/abs/1008.1975, 2010.

[23] J. M. Marberg and E. Gafni, *An* $O(n^2 m^{1/2})$ *distributed max-flow algorithm*, in Proceedings of the International Conference on Parallel Processing (ICPP'87), Pennsylvania State University Press, University Park, PA, 1987, pp. 213–216.

[24] Y. Nesterov, *Introductory Lectures on Convex Optimization*, Appl. Optim. 87, Springer Science & Business Media, New York, 2004.

[25] B. Patt-Shamir, *A note on efficient aggregate queries in sensor networks*, Theoret. Comput. Sci., 370 (2007), pp. 254–264.

[26] D. Peleg, *Distributed Computing: A Locality-Sensitive Approach*, Discrete Math. Appl. 5, SIAM, Philadelphia, 2000, https://doi.org/10.1137/1.9780898719772.

[27] R. Peng, *Approximate Undirected Maximum Flows in* $O(m\,polylog(n))$ *Time*, https://arxiv.org/abs/1411.7631, 2014.

[28] S. A. Plotkin, D. B. Shmoys, and É. Tardos, *Fast approximation algorithms for fractional packing and covering problems*, Math. Oper. Res., 20 (1995), pp. 257–301.

[29] H. Räcke, *Optimal hierarchical decompositions for congestion minimization in networks*, in Proceedings of the Annual ACM Symposium on Theory of Computing (STOC), ACM, New York, 2008, pp. 255–264.

[30] A. Schrijver, *On the history of the transportation and maximum flow problems*, Math. Program., 91 (2002), pp. 437–445, https://doi.org/10.1007/s101070100259.

[31] A. Segall, *Decentralized maximum-flow protocols*, Networks, 12 (1982), pp. 213–230.

[32] J. Sherman, *Nearly maximum flows in nearly linear time*, in Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS), IEEE, Washington, DC, 2013, pp. 263–269.

[33] D. A. Spielman and S. Teng, *Nearly-Linear Time Algorithms for Preconditioning and Solving Symmetric, Diagonally Dominant Linear Systems*, https://arxiv.org/abs/cs/0607105, 2006

[34] N. E. Young, *Sequential and parallel algorithms for mixed packing and covering*, in Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS), IEEE, Washington, DC, 2001, pp. 538–546.