Near-Optimal Distributed Maximum Flow

Extended Abstract

Mohsen Ghaffari
Massachusetts Institute of Technology
02139 Cambridge, USA
ghaffari@csail.mit.edu

Andreas Karrenbauer*
MPI for Informatics
66123 Saarbrücken, Germany
karrenba@mpi-inf.mpg.de

Fabian Kuhn
University of Freiburg
79110 Freiburg, Germany
kuhn@cs.uni-freiburg.de

Christoph Lenzen
MPI for Informatics
66123 Saarbrücken, Germany
clenzen@mpi-inf.mpg.de

Boaz Patt-Shamir†
Tel Aviv University
Tel Aviv 6997801, Israel
boaz@tau.ac.il

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 $O(m)$ time bound, and it nearly matches the $\tilde{\Omega}(D+\sqrt{n})$ round complexity lower bound.

The algorithm contains two sub-algorithms of independent interest, both with running time $(D+\sqrt{n})\cdot n^{o(1)}$:

- Distributed construction of a spanning tree of average stretch $n^{o(1)}$.
- 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.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Network Problems*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Non-numerical Algorithms and Problems—*Computations on Discrete Structures*

*Supported by the Max Planck Center for Visual Computing and Communication (www.mpc-vcc.org).

†Supported in part by the Israel Science Foundation (grant no. 1444/14) and by a grant from the Israel Ministry of Science, Technology and Space.

PODC'15, July 21–23, 2015, Donostia-San Sebastián, Spain.

ACM 978-1-4503-3617-8 /15/07 ...$15.00.
http://dx.doi.org/10.1145/2767386.2767440.

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 $O(m)$, i.e., the time to collect the entire topology and solve locally. In this paper we improve this unsatisfying state of affairs to near-optimality:

THEOREM 1.1. *On undirected weighted graphs, a $(1+\varepsilon)$-approximation of a maximum s-t flow can be computed in $(D+\sqrt{n})\cdot n^{o(1)}\varepsilon^{-3}$ rounds of the CONGEST model with high probability.*

This round complexity almost matches the $\tilde{\Omega}(D+\sqrt{n})$ lower bound of Das Sarma et al. [11], which holds for any non-trivial approximation.

1.1 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 [22] (see, e.g., the classic book [2] and the recent survey [12]). 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, where m is the number of edges, by collecting all input at a single node, solving it there, and distributing the results back.

Early attempts focused, as customary in those days, on reducing the number of messages in asynchronous executions. For example, Segall [23] gives an $O(nm^2)$-messages, $O(n^2m)$-time algorithm for exact max flow, and Gafni and Marberg [16] give an 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 syn-

chronizer, giving rise to an algorithm whose time and message complexities are $O(n^3)$. With Leighton, in [8] they give 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 LP (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 running time $\tilde{O}(n^2)$.

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

1.2 The Model

We use the standard CONGEST model of synchronous computation [19]. We are given a connected, weighted graph $G = (V, E, \text{cap})$, where $\text{cap} : E \to \mathbb{N}$, $\text{cap}(e) \in \text{poly}\, n$, are the edge capacities.[1] By D, we denote the (hop) diameter of G. Each of the $n := |V|$ nodes hosts a processor with a unique identifier of $O(\log n)$ bits, and over each of the $m := |E|$ edges $O(\log n)$ bits can be sent in each synchronous round of communication. 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.[2] Initially, each node only knows its identifier, its incident edges, and their capacities.

1.3 The Problem

We fix an arbitrary orientation of the edges. In the following, we write $(u, v) \in E$ if $\{u, v\} \in E$ is directed from u to v. An instance of the (single-commodity) *max flow* problem is specified by a graph G, and by two designated nodes: a *source* denoted s and a *sink* denoted t . A (feasible) *flow* of *value* $F \in \mathbb{R}$ is a vector $\boldsymbol{f} \in \mathbb{R}^E$ satisfying:

1. capacity constraints (edges): $\forall e \in E : \ |f_e| \leq \text{cap}(e)$;
2. conservation constraints (nodes):

$$\forall v \in V \setminus \{s, t\} : \sum_{(u,v)\in E} f_e - \sum_{(v,u)\in E} f_e = 0; \text{ and}$$

3. value constraints (at source and sink):

$$\sum_{(s,u)\in E} f_e - \sum_{(u,s)\in E} f_e = \sum_{(u,t)\in E} f_e - \sum_{(t,u)\in E} f_e = F.$$

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 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 each of its incident edges $e = \{u, v\} \in E$.

[1] As merely an approximate flow is required, we can reduce the general case to this setting in $\tilde{O}((\sqrt{n}+D)\log C)$ rounds, where C is an upper bound on the ratio between the largest and smallest capacity.

[2] Taking the union bound over polynomially many events does not affect this property. We use this fact frequently and implicitly throughout the paper.

1.4 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 [24], as well as a few other contributions. Due to lack of space, in this paper we focus on the presentation of the key concepts relevant to our results (along with credit).

We start by carefully revisiting Sherman's approach [24] 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; for lack of space, complete proofs are deferred to the full version of the paper. Finally, in Section 4, we outline the distributed construction of an $n^{o(1)}$-*congestion approximator*, which 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 the algorithm.

2. OVERVIEW OF THE CENTRALIZED FRAMEWORK

Sherman's approach [24] is based on *gradient descent* (see, e.g., [17]) for *congestion minimization* with a clever dualization of the flow conservation constraints. The flow problem is re-formulated 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 negative b_s and positive 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

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

Note that given a general (i.e., unconstrained) flow vector $\boldsymbol{f} \in \mathbb{R}^m$, $(B\boldsymbol{f})_v$ is exactly the excess flow at node v. Hence, by the max-flow min-cut theorem, if we can solve problem (1), a simple binary search will find an approximate max flow.

Instead of directly solving this constrained system, Sherman allows for general vectors and adds a penalty term for any violation of flow constraints, 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 are chosen so that the optimum of this unconstrained optimization problem does not

violate the flow constraints. As we are interested in an approximate max flow, we can compute an approximate solution and argue that the violation of the flow constraints will be small, too. Then one simply re-routes the 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.

2.1 The Congestion Approximator R

The *congestion* of an edge e (for a given flow $\boldsymbol{f}$) is defined as the ratio $|f_e|/\mathrm{cap}(e)$. When referring to the congestion of a cut in a given flow, we mean the ratio between the net flow crossing the cut to the total capacity of the cut. Suppose for a moment that $\alpha = 1$ and R contains one row for each cut 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). The max-flow min-cut theorem and the factor of 2 in the second term of the objective function imply that it always improves the value of the objective function to route the demands arising from a violation of flow constraints optimally. Moreover, the gradient descent concentrates 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. 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 small violation of 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 unsatisfied demands improves the objective function. However, this implies that the second term of the objective function may dominate its gradient and thus emphasis is shifted rather to feasiblity than optimality. Sherman proves that this slows down the gradient descent by at most a factor of α^2, i.e., if $\alpha \in n^{o(1)}$, so is the number of iterations of the gradient descent algorithm that need to be performed.

2.2 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 on T, this cut contains only the single deleted edge, and 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 [21], Räcke showed that, using multiplicative weight updates (see e.g. [4, 20, 25]), 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 $O(\alpha C)$; here α is the approximation ratio of a *low average stretch spanning tree* algorithm Räcke's construction uses as subroutine. Note that this bound on the expectation implies that for any cut of 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.

2.3 Low Average Stretch Spanning Trees

In order to perform Räcke's construction, one requires an efficient algorithm for computing low average stretch spanning trees. 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 $\alpha \in O(\log n \log^2 \log n)$ 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 [10].

2.4 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 our point of view as well as Sherman's. Madry generalized Räcke's approach to a construction resulting in a distribution over $\tilde{O}(m/j)$ so-called *j-trees* [15], 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 Fig. 1).

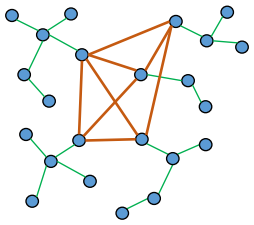

Figure 1: A 5-tree with core links depicted in brown.

The properties of the distribution are the same as for Räcke's: sampling from the distribution preserves cut capacities up to an expected $O(\alpha)$-factor, where α is the stretch of the utilized spanning tree algorithm. Likewise, using all (dominant) cuts of all j-trees in the distribution to construct R yields an $O(\alpha)$-congestion approximator. Note that any cut in a j-tree is *dominated* by either a cut induced by an edge of the forest, or by a cut of the core, in the following sense: Consider any demand vector and any "mixed" cut. 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.

2.5 Congestion Approximators: Combining Cut Sparsifiers and Madry's Construction

Using j-trees, Sherman derives a suitable congestion approxmiator, 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 [9]. 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 Sherman's 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 $\tilde{O}(\beta^{\log_\beta n}) = \tilde{O}(n)$, the number of nodes in a graph (i.e., a core from the previous level) on the i^{th} 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

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

2.6 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. 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. Fig. 1). In the forest component, routing demands 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 we recurse.

Sherman proves that his 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.[3] It is straightforward to see (cf. Section 5.1) 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: CONTRIBUTION AND KEY IDEAS

For a distributed implementation of Sherman's approach, many subproblems need to be solved (sufficiently fast) in the CONGEST model. We summarize them in the following list, where stars indicate that these components are readily available from prior work.

* 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 [14].
* Constructing cut sparsifiers. Koutis [13] provides a solution that completes in polylog n rounds of the CONGEST model. We prove a simulation result for use in the recursive construction.

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

[3] Sherman mentions that Nesterov's accelerated gradient descent method [18] could reduce this to $O(\varepsilon^{-2}\alpha \log^2 n)$ steps.

3.1 Low Average Stretch Spanning Trees

THEOREM 3.1. *Suppose H is a multigraph obtained from G by assigning arbitrary edge lengths in $\left[2^{n^{o(1)}}\right]$ to the edges of G 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 theorem, we translate a PRAM algorithm by Blelloch et al. [10] to the CONGEST model. The main issue when transitioning from the PRAM to the CONGEST model is that in the PRAM model, information about distant parts of the graph may be readily accessed. In the CONGEST model, we handle this by pipelining long-distance communication over a global BFS tree of G; communication over $O(\sqrt{n})$ hops is handled using the edges that have already been selected for inclusion into the spanning tree and spanning trees of the contracted regions of G.

3.2 Implementing Madry's Scheme

This is technically the most challenging part. Also here, we have to overcome the difficulty of potentially needing to communicate a large amount of information over many hops; doing this naively results in too much contention and thus slow algorithms. We approach this by modifying Madry's construction so that:

- Instead of "aggregating" edges so that the core becomes a graph, we admit a multigraph as core.
- We do not explicitly construct the core. Instead, we simulate both the sparsifier and the low average stretch spanning tree algorithm using the abstraction of *cluster graphs*.
- In doing so, we maintain that every core edge is also a graph edge. This enables to handle all communication over this edge by using the corresponding graph edge.
- The cluster hierarchy that is established during the construction permits a straightforward recursive evaluation of the corresponding congestion approximator.

3.3 Sampling from the Distribution

This is now straightforward, because for each sample, on each level of the recursion we need to construct only $n^{o(1)}$ different j-trees for some j.

THEOREM 3.2 (INFORMAL). *Within $\tilde{O}((\sqrt{n}+D)\cdot\beta)$ rounds of the CONGEST model, we can sample a virtual tree from the distribution used in Sherman's framework, where $\tilde{O}(\beta)$ is the number of j-trees in the distribution constructed when recursing on a core. The distributed representation allows to evaluate the dominant cuts of the tree when using it in a congestion approximator within $\tilde{O}(\sqrt{n}+D)$ rounds.*

3.4 Avoiding the Use of the entire Distribution for the Congestion Approximator

While Sherman can afford to use all trees in the (recursively constructed) distribution, the above theorem is not strong enough to allow for fast evaluation of all $\tilde{\Theta}(n)$ trees. As Madry points out [15], 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 expense 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 bound, 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 a max flow was translated to minimizing congestion for demands $-F$ and F at s and 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.

Unfortunately, applying the sampling strategy as indicated by Madry is infeasible in Sherman's framework. As the goal is a $(1+\varepsilon)$-approximation, applying it to the above problem directly will yield a too inaccurate approximation. 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 $\boldsymbol{b}\in\{-1,0,1\}^n$, 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 expense of squaring the approximation ratio of the resulting congestion approximator.

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

PROOF. Recall that cut approximators estimate the maximum congestion when optimally routing an arbitrary demand. Consider any demand vector and denote by C the capacity of the corresponding cut that is most congested when routing the demand. As sampling from the distribution yields approximation factor α in expectation, there must be *some* tree T in the distribution whose corresponding cut has capacity at most αC. However, this means that when routing the demand via T, there is some edge in T that experiences at least $1/\alpha$ times the maximum congestion when routing the demand optimally in G. As the capacity of the edge is at least that of the corresponding cut in G, it follows that the corresponding cut of G has congestion at least $1/\alpha$ of that of the min-cut when routing the demand.

As there are $\operatorname{poly} n$ trees, each of which has $n-1$ edges, this shows that for any demand vector there is one of polynomially many cuts of G that experience at least $1/\alpha$ times the maximum congestion when optimally routing the demand vector. By Markov's bound and the union bound, w.h.p. the congestion on each of these cuts will be approximated up to another factor of 2α when using $O(\log n)$ samples. □

3.5 Performing a Gradient Descent Step

Most of the high-level operations required for executing a gradient descent algorithm are straightforward to implement

using direct communication between neighbors or broadcast and convergecast operations on a BFS tree. The most involved part is multiplying the (implicitly constructed) congestion approximator R with an arbitrary demand vector $\boldsymbol{b}$, and multiplying the transposed of the approximator matrix, $R^\top$, with a given vector that specifies a *cost* for each edge of the trees.

Multiplying by R is done by exploiting that routing on trees is trivial and using standard techniques: during the construction, we already decomposed each tree into $O(\sqrt{n})$ components of strong diameter $O(\sqrt{n})$, which can be used to solve partially by contracting components, make the resulting tree of $O(\sqrt{n})$ nodes globally known, then determine modified demand vectors for the components out of the now locally computable partial solution, and finally resolve these remaining demands within each component. Multiplication with $R^\top$ is implemented using similar ideas. We refer to Section 5 for a detailed discussion of these procedures. Plugging the building blocks outlined in this section into this machinery, we obtain our main result Theorem 1.1.

4. THE DISTRIBUTED CONGESTION APPROXIMATOR CONSTRUCTION

In this section, we outline how to adapt Madry's construction to its recursive application in the distributed setting. We formally prove that we achieve the same guarantees as Madry's distribution [15] 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 a single step of the recursion.

4.1 Centralized Algorithm

As a starting point, let us summarize the main steps of one iteration of the centralized construction. We state a slightly simplified variant of Madry's construction, which offers the same worst-case performance and is a better starting point for what follows. From the previous step of constructing the distribution, 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 $\mathrm{cap}(e)$ units of a commodity com_e from v to w on (the unique path from v to w in) $\mathcal{T}$.[4] Denote by $\boldsymbol{f}$ the vector of the sum of absolute flows passing through the edges of $\mathcal{T}$. Recall that $\max_{e\in E}\{\mathrm{cap}(e)\} \in \mathrm{poly}\, n$ and thus $\|\boldsymbol{f}\|_\infty \in \mathrm{poly}(n)$.
3. For $e \in \mathcal{T}$, define the *relative load* of e as $\mathrm{rload}(e) := |f_e|/\mathrm{cap}(e) \in \mathrm{poly}\, n$. We decompose the edge set of $\mathcal{T}$ into $O(\log n)$ subsets $\mathcal{F}_i$, $i \in \{1, \ldots, \lceil \log(\|\boldsymbol{f}\|_\infty + 1)\rceil\}$, where $e \in \mathcal{T}$ is in $\mathcal{F}_i$ if $\mathrm{rload}(e) \in (R/2^i, R/2^{i-1}]$ for $R := \max_{e\in\mathcal{T}}\{\mathrm{rload}(e)\}$. As $\mathcal{T}$ has $n-1 \geq j$ edges, there must be some $\mathcal{F}_i$ with $\Omega(j/\log n)$ edges; let i_0 be minimal with this property. Define $\mathcal{F} := \{e \in \mathcal{T} \mid \mathrm{rload}(e) > 2^{i_0-1}\}$. Note that $|\mathcal{F}| \leq j$.
4. $\mathcal{T}\setminus\mathcal{F}$ is a spanning 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 between different components of $(V, \mathcal{T}\setminus\mathcal{F})$.
5. For components C and C' of $(V, \mathcal{T}\setminus\mathcal{F})$, pick arbitrary $v \in C$ and $w \in C'$ and denote by $p(C, C') \in C$ the last node from C on the v-w path in $\mathcal{T}$; note that $p(C, C')$ does not depend on the choice of v and w. Denote by P the set of such *portals*. Replace all edges between different components C, C' of $(V, \mathcal{T}\setminus\mathcal{F})$ by parallel edges $\{p(C, C'), p(C', C)\}$ (of the same capacity).
6. In the resulting multigraph, 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.
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 challenges, so we will focus on the distributed implementation of the above steps in this section.

4.2 Modifications to the Centralized Algorithm

Before we come to the distributed algorithm, let us first discuss a few changes we make to the algorithm in centralized terms. These do not affect the reasoning underlying the scheme, but greatly simplify its distributed implementation.

- We will omit the last step of the algorithm 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, as one can see by replacing 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 affecting the 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, we will immediately delete a subset of $\tilde{O}(\sqrt{n})$ edges to ensure that the new clusters will 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's arguments show that removing *any* subset of edges of $\mathcal{T}$ and replacing it this way can only improve the quality of cut approximation. The main point of his analysis is that choosing $\mathcal{F}$ in the way he does guar-

[4] The difference to a single commodity is simply that flows in opposing directions do not cancel out. This means that *any* given feasible (i.e., congestion-1) flow in G can be routed on T with at most the congestion of this multi-commodity flow.

antees that, in terms of constructing the final distribution of j-trees, progress proportional to the number of edges in $\mathcal{R}_{i_0}$ is made. We will apply the construction to cores of size $n' \gg \tilde{O}(\sqrt{n})$, which implies that removing the additional edges has asymptotically no effect on the progress guarantee.
- In the counterpart to Step 6 in Madry's routine, also nodes from P may be removed if their degree becomes 1. Also here, there is no asymptotic difference in the worst-case performance of our routine from Madry's.

To simplify the presentation, in this section we will 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. This approach was first used for MST construction [14]; we use a simpler randomized variant.

4.3 Cluster Graphs

Recall that we will recursively call (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 will use *cluster graphs*, in which G is decomposed into components that play the role of core nodes We maintain the following invariants during the recursion:

1. There is a one-to-one correspondence between nodes of the core 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 will refer to this multigraph as the core.
4. All edges in the (non-contracted) graph are also edges of G, and their endpoints know their lengths from the previous iteration of the recursion.

4.4 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 following sketches the main steps of the distributed implementation of our overall approach.

1. Compute a spanning tree $\mathcal{T}$ of stretch α of the core. This is done by the spanning tree algorithm of Theorem 3.1, which can operate on the cluster graph.
2. For each edge $e \in \mathcal{T}$, determine its absolute flow $|f_e|$ (and thus $\text{rload}(e) = |f_e|/\text{cap}(e)$) as follows (cf. Fig. 2).
   (*) For each cluster c, consider the cut induced by the edge to its parent. For each "side" of the cut, we want to determine the total capacity of all edges incident to nodes of c that connect to the respective side of the cut. Denote by c_+ the total "outgoing" capacity of cluster c towards the root's side and by c_- the "incoming" capacity.
   (a) Each cluster c learns its ancestor clusters in the spanning tree of C.
   (b) Observe that for a cluster c, an edge does contribute to c_- if and only if it connects to a node within its subtree $\mathcal{T}_c$. From the previous step, this information is known to one of the endpoints of the edge. We communicate this and determine in each cluster c the values c_+ and c_- by aggregation on its spanning tree.
   (c) Suppose $e \in \mathcal{T}$ is the edge from cluster c to its parent. Using aggregation on the spanning tree of C, we compute $|f_e| = \sum_{c' \in \mathcal{T}_c} c'_+ - c'_-$.
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 done in $\tilde{O}(D)$ rounds using binary search in combination with converge- and broadcasts on a BFS tree. We set $\mathcal{F} := \{e \in \mathcal{T} \mid \text{rload}(e) > 2^{i_0-1}\}$.
4. Define P as the set of clusters incident to edges in $\mathcal{F}$. A simple broadcast on the cluster spanning trees makes membership known to all nodes of each cluster $c \in P$.
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 2 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 not being 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.

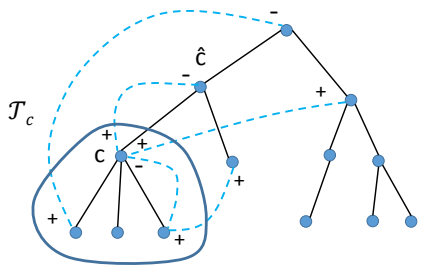

Figure 2: Illustration of the underlying idea of the aggregation scheme for the cut capacities. The cut corresponding to edge $(c, \hat{c})$ of the tree has a total capacity given by all graph edges leaving the subtree $\mathcal{T}_c$. By labeling the endpoint of graph edge by "+" if it leaves the subtree and by "−" if it connects to a descendant, the cut capacity is thus the sum of all capacities of edges labeled "+" minus all those of edges labeled "−" within $\mathcal{T}_c$.

5. THE HIGH-LEVEL ALGORITHM

The algorithm is a distributed implementation of Sherman's algorithm [24]. It consists of a logarithmic number of calls to algorithm AlmostRoute and one computation of a maximum-weight spanning tree and routing the left-over demand through this tree. Most of this section is dedicated to explaining how to implement the AlmostRoute algorithm.

Let us first quickly outline how we implement the final steps using standard techniques.

LEMMA 5.1. *Routing the leftover demands over an MST can be implemented in the CONGEST model in $\tilde{O}(D+\sqrt{n})$ rounds w.h.p.*

PROOF SKETCH. 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 [14]. To compute the flow, we use the following observation: if T was rooted at one of its nodes, then to route the demand over T, it would suffice for each node v to learn the total demand d_v in the subtree rooted at v. Then, 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 $1/\sqrt{n}$. W.h.p.,

(i) each connected component induced by the remaining edges contains 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 over a BFS tree in $\tilde{O}(\sqrt{n}+D)$ rounds due to (iii).

Moreover, in this time we can also assign unique identifiers to the components (e.g., the minimum identifier) and make the tree resulting from contracting components globally known. 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. □

5.1 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

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

is used as a differentiable approximation to the max-norm.

AlmostRoute performs $O(\alpha^2\varepsilon^{-3}\log n)$ iterations to compute a flow $\boldsymbol{f}$ optimizing the potential up to factor $(1+\varepsilon)$. Pseudocode for this algorithm is given in Algorithm 1.

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

Algorithm 1 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: $\quad \delta \leftarrow \sum_{e\in E}|\operatorname{cap}(e)\frac{\partial\phi}{\partial f_e}|$
7: $\quad$ **if** $\delta\ge\varepsilon/4$ **then**
8: $\quad\quad f_e \leftarrow f_e - \operatorname{sgn}\left(\frac{\partial\phi}{\partial f_e}\right)\cdot\operatorname{cap}(e)\frac{\delta}{1+4\alpha^2}$
9: $\quad$ **else**
10: $\quad\quad f_e\leftarrow f_e/k_f$ for all edges $e\in E$.
11: $\quad\quad b_v\leftarrow b_v/(k_bk_f)$ for all nodes $v\in V$
12: $\quad\quad$ **return**
13: **until** done

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 v is incident to, and the current demand at v (i.e., $(\boldsymbol{b}-B\boldsymbol{f})_v$). Let us break the potential function ϕ in two, i.e.,

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

We proceed as follows. First, we compute $\phi_1(\boldsymbol{f})$: to find $\operatorname{smax}(C^{-1}\boldsymbol{f})$, it suffices to sum both $\exp(f_e/\operatorname{cap}(e))$ and $\exp(-f_e/\operatorname{cap}(e))$ 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, and thus, encoding $\exp(\phi(\boldsymbol{f}))$ with sufficient accuracy requires $O(\varepsilon^{-1}\log n)$ bits, which is thereby also a bound on the encoding length of all individual terms in the sums for ϕ_1 and ϕ_2. The error introduced by rounding theses values to integers is small enough to not affect the asymptotics of the running time.

For determining $\phi_2(\boldsymbol{f})$, we first compute the vector $\boldsymbol{y} := 2\alpha R(\boldsymbol{b}-B\boldsymbol{f})$ and then do an aggregation on a BFS tree as for $\phi_1(\boldsymbol{f})$. Since $B\boldsymbol{f}$ can be computed instantly ($(B\boldsymbol{f})_v$ is exactly the net flow into v), this boils down to multiplying a locally known vector with R. Before we discuss how implement this operation, let us explain more about the structure of R and how we determine $\frac{\partial\phi}{\partial f_e}$, which is required in Lines 6 and 8.

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, denote by I the set of row indices of R. Observe that

$$\frac{\partial\phi}{\partial f_e} = \frac{\exp(f_e/\operatorname{cap}(e)) - \exp(-f_e/\operatorname{cap}(e))}{\operatorname{cap}(e)\exp(\phi_1)} + \frac{\partial\phi_2}{\partial f_e} \qquad (2)$$

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

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

where $\operatorname{cap}(i)$ is the capacity of cut i in the congestion approximator and $B_{i,e}\in\{-1,0,1\}$ denotes whether e is outgoing (-1), ingoing (1), or not crossing cut i.[5]

The cuts $i\in I$ are induced by the edges of a collection of (rooted, virtual, capacitated) spanning trees $\mathbb{T}$, where for

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

$\mathcal{T} \in \mathbb{T}$ 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. For each $\mathcal{T} \in \mathbb{T}$, each edge $(v, \hat{v}) \in \mathcal{T}$ now 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 by $\delta(\mathcal{T}_v)$. Let us also define

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

With this notation, we have that

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

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 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,

$$\frac{\partial \phi_2}{\partial f_e} = \pi_v - \pi_u , \tag{3}$$

and our task boils down to determining the value of the potential π_v at each node $v \in V$. For that, we use two subroutines to compute distributedly the following quantities:

(1) y_i for each cut i. Note that $\boldsymbol{b} - B\boldsymbol{f}$ is known distributedly, 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, i.e., simulate a convergecast on the virtual tree $\mathcal{T}$.

(2) π_v for each node v. Provided that each (virtual) tree edge knows its 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 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 the previous step.
3. Compute the potential $\boldsymbol{\pi}$ (local knowledge).
4. For each $e \in E$, its incident nodes determine $\frac{\partial \phi}{\partial f_e}$ (based on Equations 2 and 3, it suffices to exchange π_u and π_v over e).
5. Compute δ (aggregation on 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. Sherman proved [24] that AlmostRoute terminates after $\tilde{O}(\varepsilon^{-3}\alpha^2)$ iterations. As it is only called $O(\log n)$ times by the max-flow algorithm, Theorem 1.1 follows if we can compute $\boldsymbol{y}$ and $\boldsymbol{\pi}$ in $(\sqrt{n}+D)n^{o(1)}$ rounds for an α-congestion approximator with $\alpha = n^{o(1)}$; this is subject of the next subsection.

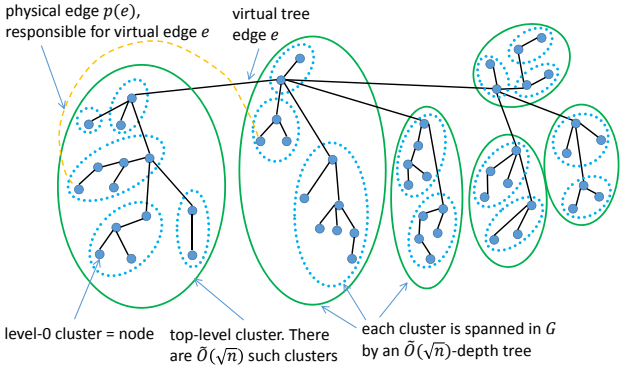

Figure 3: Hiearchical cluster decomposition of a virtual tree $\mathcal{T} \in \mathbb{T}$. Black edges are virtual tree edges represented by a physical edge connecting the top-level clusters they connect (the orange dotted edge $p(e)$ corresponds to the edge labeled e). Each cluster is spanned by a tree in G of depth $\tilde{O}(\sqrt{n})$, which is not shown.

5.2 Congestion Approximation

Our congestion approximator R is defined by the edge-induced cuts of a sample $\mathbb{T}$ of virtual trees $\mathcal{T}$ from a recursively constructed distribution. The trees are represented distributedly by a hierarchy of *cluster graphs* (see Fig. 3 for an illustration). 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.

THEOREM 5.2. *W.h.p., 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 $\mathcal{G}_i$ on cluster sets $\mathcal{V}_i$ (their "nodes"), $i \in \{0, \dots, i_0\}$, $i_0 \in o(\log n)$, on network graph G, with the following properties.*
  - *Spanning trees of clusters of $\mathcal{G}_i$ have depth $\tilde{O}(\sqrt{n})$.*
  - *$|\mathcal{V}_{i_0}| = n^{1/2+o(1)}$.*
  - *$\mathcal{G}_i$ (as graph on node set $\mathcal{V}_i$) 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 to all edges of $\mathcal{G}_i$ containing c_i. That is, $\mathcal{G}_{i-1}$ is a $|\mathcal{V}_i|$-tree with core $p(\mathcal{V}_i)$.*

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:

COROLLARY 5.3. *Sampling a collection $\mathbb{T}$ of $O(\log n)$ virtual trees given by Theorem 3.2 and using them as congestion approximator R in the way specified in Section 5.1 implies that the total number of iterations of Algorithm 1 is $n^{o(1)}$.*

All that remains now is to show that the distributed representation of each sampled $\mathcal{T} \in \mathbb{T}$ allows 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 5.1 with this time complexity, and by Corollary 5.3 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 i^{th} 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. Fig. 3) 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 5.4. *On each virtual tree $\mathcal{T} \in \mathbb{T}$, we can simulate convergecast and upcast 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 [24], the discussion in Section 5.1, and Corollaries 5.3 and 5.4.

6. REFERENCES

[1] I. Abraham, Y. Bartal, and O. Neiman. Nearly tight low stretch spanning trees. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 781–790. IEEE, 2008.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows*. Prentice-Hall, Engelwood Cliffs, New Jersey, 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 Journal on Computing*, 24(1):78–100, 1995.

[4] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[5] B. Awerbuch. Reducing complexities of the distributed max-flow and breadth-first-search algorithms by means of network synchronization. *Networks*, 15(4):425–437, Winter 1985.

[6] B. Awerbuch and R. Khandekar. Stateless distributed gradient descent for positive linear programs. *SIAM Journal on Computing*, 38(6):2468–2486, 2009.

[7] B. Awerbuch, R. Khandekar, and S. Rao. Distributed algorithms for multicommodity flow problems via approximate steepest descent framework. *ACM Transactions on Algorithms*, 9(1):3, 2012.

[8] B. Awerbuch and T. Leighton. Improved approximation algorithms for the multi-commodity flow problem and local competitive routing for dynamic networks. In *Proc. 26th Ann. ACM Symp. on Theory of Computing*, pages 487–496, 1994.

[9] A. A. Benczúr and D. R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *CoRR*, cs.DS/0207078, 2002.

[10] 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(3):521–554, 2014.

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

[12] A. V. Goldberg and R. E. Tarjan. Efficient maximum flow algorithms. *Commun. ACM*, 57(8):82–89, August 2014.

[13] I. Koutis. Simple parallel and distributed algorithms for spectral graph sparsification. In *the Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 61–66, 2014.

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

[15] A. Madry. Fast approximation algorithms for cut-based problems in undirected graphs. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 245–254, 2010.

[16] J. M. Marberg and E. Gafni. An $O(n^2m^{1/2})$ distributed max-flow algorithm. In *Int. Conf. on Parallel Processing, (ICPP'87)*, pages 213–216, 1987.

[17] Y. Nesterov. *Introductory lectures on convex optimization*, volume 87. Springer Science & Business Media, 2004.

[18] Y. Nesterov. Smooth minimization of non-smooth functions. *Mathematical programming*, 103(1):127–152, 2005.

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

[20] S. A. Plotkin, D. B. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, 20(2):257–301, 1995.

[21] H. Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 255–264, 2008.

[22] A. Schrijver. On the history of the transportation and maximum flow problems. *Mathematical Programming*, 91(3):437–445, 2002.

[23] A. Segall. Decentralized maximum-flow protocols. *Networks*, 12(3):213–230, Fall 1982.

[24] J. Sherman. Nearly maximum flows in nearly linear time. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 263–269, 2013.

[25] N. E. Young. Sequential and parallel algorithms for mixed packing and covering. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 538–546. IEEE, 2001.