Parallel Approximate Maximum Flows in Near-Linear Work and Polylogarithmic Depth

Arpit Agarwal* Sanjeev Khanna† Huan Li† Prathamesh Patil†
Chen Wang‡ Nathan White§ Peilin Zhong¶

November 6, 2023

Abstract

We present a parallel algorithm for the $(1-\varepsilon)$ -approximate maximum flow problem in capacitated, undirected graphs with n vertices and m edges, achieving $\mathrm{O}(\varepsilon^{-3}\operatorname{polylog} n)$ depth and $O(m\varepsilon^{-3}\operatorname{polylog} n)$ work in the PRAM model. Although near-linear time sequential algorithms for this problem have been known for almost a decade, no parallel algorithms that simultaneously achieved polylogarithmic depth and near-linear work were known.

At the heart of our result is a polylogarithmic depth, near-linear work recursive algorithm for computing congestion approximators. Our algorithm involves a recursive step to obtain a low-quality congestion approximator followed by a "boosting" step to improve its quality which prevents a multiplicative blow-up in error. Similar to Peng [SODA'16], our boosting step builds upon the hierarchical decomposition scheme of Räcke, Shah, and Täubig [SODA'14]. A direct implementation of this approach, however, leads only to an algorithm with $n^{o(1)}$ depth and $m^{1+o(1)}$ work. To get around this, we introduce a new hierarchical decomposition scheme, in which we only need to solve maximum flows on subgraphs obtained by *contracting* vertices, as opposed to vertex-induced subgraphs used in Räcke, Shah, and Täubig [SODA'14]. This in particular enables us to directly extract congestion approximators for the subgraphs from a congestion approximator for the entire graph, thereby avoiding additional recursion on those subgraphs. Along the way, we also develop a parallel flow-decomposition algorithm that is crucial to achieving polylogarithmic depth and may be of independent interest.

We extend our results to related graph problems such as sparsest and balanced sparsest cuts, fair and isolating cuts, approximate Gomory-Hu trees, and hierarchical clustering. All algorithms achieve polylogarithmic depth and near-linear work.

Finally, our PRAM results also imply the first polylogarithmic round, near-linear total space MPC algorithms for approximate undirected maximum flows, as well as all its aforementioned applications in the fully scalable regime where the local machine memory is $O(n^{\delta})$ for any constant $\delta > 0$.

1 Introduction.

The *maximum flow* problem, or equivalently, the *minimum congestion flow* problem is one of the oldest and most well-studied combinatorial optimization problems that finds numerous applications across

*Columbia University. email: arpit.agarwal@columbia.edu

†University of Pennsylvania. email: {sanjeev,huanli,pprath}@cis.upenn.edu. Research supported in part by NSF awards CCF-1934876 and CCF-2008305.

‡Rutgers University. email: wc497@cs.rutgers.edu, supported in part by NSF CAREER Grant CCF-2047061, a gift from Google Research, and a Rutgers Research Fulcrum Award.

§University of Pennsylvania. email: nathanlw@cis.upenn.edu. Research supported in part by NSF awards CCF-1934876 and CCF-2008305, and partially supported by the Department of Defense (DoD) through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program.

¶Google Research. email: peilinz@google.com

computer science and engineering. Formally, we are given a directed, capacitated flow network (graph) $G = (V, E, c)$ with $|V| = n$ vertices and $|E| = m$ edges with positive edge capacities $c \in \mathbb{R}^E$, along with a set of vertex demands $b \in \mathbb{R}^V$ specifying the desired excess flow at each vertex where $\sum_{v \in V} b_v = 0$. The objective is to find a flow $f \in \mathbb{R}^E$ that satisfies demands b, while minimizing the maximum congestion $|f_e|/c_e$ on any edge $e \in E$ in the network, i.e.

$$\min \ \|C^{-1}f\|_\infty \ \text{ s.t. } Bf = b \ \textit{(flow conservation constraints)}; \ f \geq 0, \tag{1.1}$$

where $C \in \mathbb{R}^{E \times E}$ is a diagonal matrix of edge capacities with $C_{e,e} = c_e$, and $B \in \{-1, 0, 1\}^{V \times E}$ is the vertex-edge incidence matrix with $B_{v,e}$ being 1 if $e = (u, v)$, -1 if $e = (v, u)$ and 0 otherwise. For the special case where the edges are undirected, the formulation is identical except that the edges are oriented arbitrarily, and the non-negativity constraint on the flows is dropped; the sign of the flow on an edge specifies its direction relative to the edge orientation.

The maximum flow problem has a rich history, starting with the work of [19] and [25], who first proposed algorithms with pseudo-polynomial running times of $O(mn^2U)$ and $O(m^2U)$ respectively, for networks with maximum edge capacity U. Since then, substantial effort has been devoted to designing increasingly efficient algorithms for this problem. This has resulted in a sequence of exciting developments, with initial improvements coming from primarily combinatorial ideas such as shortest augmenting paths and blocking flows [21, 23, 41, 22, 24, 9], push-relabeling [33, 30, 56], pseudo-flows [37, 10], and capacity scaling [2, 32], eventually culminating in the recent breakthrough $O(m^{1+o(1)})$ time result of [14] achieved through a combination of second-order continuous optimization techniques (interior point methods) and efficient dynamic graph data-structures. Meanwhile, for the weaker objective of finding an approximately maximum flow in undirected graphs, even faster algorithms have been developed using simpler first-order continuous optimization techniques [66, 57, 67, 68]. In particular, [57] showed that in undirected graphs, a $(1 - \varepsilon)$-approximate maximum flow can be computed in just $O(m \ \text{poly}(1/\varepsilon, \log n))$ time.

However, these aforementioned algorithmic results were largely developed in the sequential setting, and are not readily parallelizable. Moreover, the question of designing fast *parallel* algorithms for max-flows has remained surprisingly under-explored.

In the context of parallel algorithms, several models of computation have been proposed over the years. Amongst them, the parallel random-access machine (PRAM) is often considered as a standard, owing to its simplicity and its well-understood connections to other models of parallel computation. A generalization of the usual (sequential) RAM model, the PRAM is a synchronous, shared-memory, multi-processor model. Within PRAM, there are several variants depending on how concurrent operations in the shared memory are handled: (in decreasing order of restrictiveness) exclusive-read-exclusive-write, concurrent-read-exclusive-write, and concurrent-read-concurrent-write. However, due to the low-level nature of the PRAM, even the simplest algorithms designed for it involve tedious implementation details. In order to abstract away these specifics, we usually consider the equivalent work-depth paradigm, which measures the performance of a parallel algorithm using two parameters – *total work*, which is the total running time needed given only one processor, and *depth*, which is the total parallel time given a maximal number of processors. Moreover, in this framework, the different variants of PRAM are equivalent up to polylogarithmic factors in work and depth, and therefore, can also be abstracted.

For the question of designing parallel algorithms for max flows, the literature is sparse. For finding exact max flows, early results of [70, 62] achieved $\tilde{O}(n^2)$ depth and $\tilde{O}(mn)$ work. More recently, [59] improved the depth to $\tilde{O}(n)$ while retaining the same asymptotic work. For the weaker objective of finding an approximate max flow, the combined results of [54] and [58] imply an algorithm with $\tilde{O}(\sqrt{m})$ depth and $\tilde{O}(m^{1.5})$ work at the cost of a $O(1/\text{poly}(n))$ additive error in the flow value. A much earlier work of [65] also implies a $O(\text{poly}(\log n, \log(1/\varepsilon)))$ depth algorithm for finding a $(1 - \varepsilon)$-approximate max flow via a reduction to finding maximum matchings in bipartite graphs. However, this comes at the expense of an unspecified polynomial blow-up in work, a common issue with many of the early results for parallel algorithms. The question of whether it is possible to *simultaneously* have a polylogarithmic depth and near-linear work approximate max-flow algorithm remained open. In this paper, we answer this in the affirmative for undirected graphs. Namely, we show the following main result.

THEOREM 1.1. *There is a randomized* PRAM *algorithm that given an undirected capacitated graph $G = (V, E, c)$, $s, t \in V$, and precision $\varepsilon > 0$, computes both a $(1-\varepsilon)$-approximate s-t maximum flow and a $(1+\varepsilon)$-approximate s-t minimum cut with high probability in $O\left(\varepsilon^{-3} \operatorname{polylog} n\right)$ depth and $O\left(m\varepsilon^{-3} \operatorname{polylog} n\right)$ total work.*

A more recent model of parallel computation is the *massively parallel computation* (MPC) model, which is a common abstraction of many MapReduce-style computational frameworks (see i.e. [8, 4, 35]). In this model, the input data is partitioned across multiple machines that are connected together through a communication network. The computation proceeds in synchronous rounds where in each round, the machines can perform unlimited local computation on their local memory, but cannot communicate with other machines. Between rounds, machines can communicate, so long as the total size of messages sent and received by any machine does not exceed the size of its local memory. The performance of an MPC algorithm is measured by the number of rounds needed to complete the computation, the size of the local machine memory, as well as the total memory used across all machines. A simulation result of [40, 35] shows that any PRAM algorithm with D depth and W work can be simulated by an MPC algorithm with $O(D)$ rounds and $O(W)$ total memory even in the most stringent *fully-scalable* regime[1] where the local memory size is $O(n^\delta)$ for any constant $\delta > 0$. Using this simulation result, we obtain the following corollary of our main result.

COROLLARY 1.1. (OF THEOREM 1.1) *There is a randomized* MPC *algorithm that given an undirected capacitated graph $G = (V, E, c)$, $s, t \in V$ and precision $\varepsilon > 0$, computes both a $(1-\varepsilon)$-approximate s-t maximum flow and a $(1+\varepsilon)$-approximate s-t minimum cut with high probability in $O\left(\varepsilon^{-3} \operatorname{polylog} n\right)$ rounds, $O\left(m\varepsilon^{-3} \operatorname{polylog} n\right)$ total memory and $O\left(n^\delta\right)$ local memory, for any constant $\delta > 0$.*

Note that by standard reduction, our Results 1.1 and 1.1 extend to general vertex demands $b \in \mathbb{R}^V$ in the same form of Eq 1.1.

1.1 Applications. Our result for approximate max flows has broad implications to other related graph problems, where it implies either new or substantially improved parallel algorithms (both PRAM and MPC) for them.

Sparsest cut and balanced sparsest cut: The *sparsest cut* problem is an important subroutine for divide-and-conquer based approaches for several graph problems and has many applications including image segmentation, VLSI design, clustering and expander decomposition. In this problem, given a weighted undirected graph $G = (V, E, c)$, the objective is to find a cut $(S, \bar{S})$ with minimum sparsity $\phi(S)$ which is defined as $\phi(S) := c(E(S, \bar{S}))/\min\{|S|, |\bar{S}|\}$, where $c(E(S, \bar{S}))$ is the total weight of the edges going across the cut. The *balanced sparsest cut* problem is a variant of this problem where there is an additional requirement that the cut $(S, \bar{S})$ must be β-balanced, i.e. $\min\{|S|, |\bar{S}|\} \geq \beta n$ for some given parameter $\beta \in (0, 1/2)$. Balanced sparse cuts are useful in applications where one wants the divide-and-conquer tree to have low-depth.

The sparsest cut problem is NP-hard and the best known approximation factor of $O(\sqrt{\log n})$ is achieved by an SDP-based algorithm due to Arora, Rao and Vazirani [6]. However, this algorithm is highly sequential, and computationally expensive. The most efficient algorithms for sparsest cut problem are based on the cut-matching game framework of Khandekar, Rao and Vazirani [45], which effectively reduces the sparsest cut problem to a poly-logarithmic number of single commodity max-flow computations. In this framework, a cut-player and a matching-player play an alternating game; in each round, the former produces a bisection $(S, \bar{S})$ of vertices, and the latter produces a perfect matching between S and $\bar{S}$ that can be embedded in the underlying graph. The game ends when either the cut

[1] The dependence of the the local memory parameter (the constant $\delta > 0$) on the number of rounds and total memory is a multiplicative $O(1/\delta)$, and is usually omitted.

player produces a bisection for which the matching player cannot find a perfect matching (i.e. a sparse cut has been found), or the union of the perfect matchings produced thus far form an expander (i.e. an expander can be embedded in the underlying graph, certifying its expansion). [45] showed that there is a cut strategy with runtime $T_{\text{cut}} = \tilde{O}(n)$ such that this game terminates in $\alpha = O(\log^2 n)$ rounds, regardless of the matching player's strategy. Furthermore, the matching player's strategy can be implemented using a max-flow computation with the sources and sinks being the two sides of the bipartition. In combination, this framework produces an $O(\alpha)$-approximation to sparsest cut with a runtime of $O(\alpha \cdot (T_{\text{cut}} + T_{\text{flow}}))$, where α is an upper bound on the number of rounds of this game, T_{cut} is the time to implement cut player's strategy and T_{flow} is the time to compute a single commodity max-flow. This framework also generalizes to the problem of β-balanced sparsest cut for any constant β with the same approximation factor and running time.

Nanongkai and Saranurak [55] further showed that the matching player's strategy can be implemented using approximate max-flow computations rather than exact max-flows. Building on [55], we show that our parallel algorithm for max-flows (Result 1.1) can be used to implement the matching player's strategy, yielding a $O(\text{polylog}\, n)$ depth and $\tilde{O}(m)$ work algorithm for the matching player. Furthermore, we show that the cut player's strategy can also be implemented in $O(\text{polylog}\, n)$ depth and $\tilde{O}(m)$ work using our new *parallel flow decomposition* algorithm (see Section 2.2 for a discussion). In combination, this gives a $O(\text{polylog}\, n)$ depth and $\tilde{O}(m)$ work algorithm for sparsest cuts that has an approximation factor of $O(\log^3 n)$. These results also imply a bicriteria approximation algorithm for β-balanced sparsest cut with the same depth and work. Specifically, our algorithm finds a $(\beta/\log^2 n)$-balanced cut whose sparsity is at most $O(\log^3 n)$ times the sparsity of an optimal β-balanced cut.

Minimum cost hierarchical clustering: Hierarchical clustering is a fundamental problem in data analysis where the goal is to recursively partition data into clusters which results in a rooted tree structure. This problem has wide-ranging applications across different domains including phylogenetics, social network analysis, and information retrieval. While the study of hierarchical clustering goes back several decades, Dasgupta [20] initiated its study from an optimization viewpoint. Specifically, he proposed a minimization objective for hierarchical clustering on similarity graphs that measures the cost of a hierarchy as the sum of costs of its internal splits, which in turn is measured as the total edge-weight across the split scaled by the size of the cluster at that split. [20] and subsequent work by [17, 12] showed that the hierarchical clustering produced by recursively splitting the graph using a α-approximate sparsest cut subroutine results in a hierarchy that is an $O(\alpha)$-approximation for this minimization objective. Furthermore, using an α-approximate β-balanced sparsest cut subroutine results in the same approximation with an additional property that the depth of the tree is $O(\log n)$ for $\beta = \Omega(1)$. [12] also showed that a (α, β')-bicriteria approximation oracle for β-balanced min-cut is sufficient to achieve a $O(\alpha/\beta')$-approximation for minimum cost hierarchical clustering. As a consequence, using our parallel balanced min-cut algorithm as a subroutine, we get the first parallel algorithm that computes a tree that is a $O(\log^5 n)$-approximation to Dasgupta's objective with $\tilde{O}(m)$ work and $O(\text{polylog}\, n)$ depth.

Fair cuts and approximate Gomory-Hu trees: Li *et al.* [51] recently introduced the notion of fair cuts for undirected graphs which are a "robust" generalization of approximate min-cuts. Specifically, for $\alpha \geq 1$, a s-t-cut is α-*fair* if there exists an s-t flow that uses at least $1/\alpha$-fraction of the capacity of every edge in the cut. [51] showed that a near-linear time oracle for computing a fair cut is useful for obtaining near-linear time algorithms for several applications including computation of (approximate) all-pairs maxflow values (using approximate Gomory-Hu trees). They also showed that, given an unweighted graph, a fair cut can be computed in parallel using $m^{1+o(1)}$ work and $n^{o(1)}$ depth, implying a similar result for approximate Gomory-Hu trees. Our results improve upon their results by giving parallel algorithms for fair cuts and approximate Gomory-Hu trees that have $O(m\ \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth. The key technical tool that results in this improvement is our $O(\text{polylog}\, n)$ depth construction of a $O(\text{polylog}\, n)$-*congestion approximator* (see Section 2.1 for a discussion).

2 Technical Overview.

The starting point of our discussion is Sherman's framework [66, 68], which was initially proposed for computing fast approximate undirected max-flows in the *sequential* setting. Given the optimization problem defined in Equation 1.1, let $\mathsf{opt}_G(b) := \min_{f:Bf=b} \|C^{-1}f\|_\infty$ be the minimum congestion of any feasible flow routing given demands b in the graph G. Sherman showed that if we have a matrix Φ such that for *any* demand vector b, $\|\Phi b\|_\infty$ is always an α-approximation to $\mathsf{opt}(b)$, i.e., $\|\Phi b\|_\infty \leq \mathsf{opt}(b) \leq \alpha \cdot \|\Phi b\|_\infty$, then there is an iterative algorithm to compute a $(1+\varepsilon)$-approximate minimum congestion flow. The matrix Φ is called a *α-congestion approximator matrix*, and the algorithm requires $\mathrm{poly}(\alpha, \varepsilon^{-1}, \log n)$ iterations[2] where each iteration requires in total $O(m)$-time algorithmic operations plus computing the matrix-vector products Φx and $\Phi^T y$ for some arbitrary vectors x and y.

A line of work in the parallel transshipment (ℓ_1 flow) literature [5, 50, 64, 72] observed that Sherman's framework can be adapted to the *parallel* computing regime. In particular, they showed the following: given an α-congestion approximator matrix Φ, there is a PRAM algorithm which outputs a $(1+\varepsilon)$-approximate minimum congestion flow. The depth of the algorithm is $\mathrm{poly}(\alpha, \varepsilon^{-1}, \log n) \cdot (\mathrm{depth}(\Phi x) + \mathrm{depth}(\Phi^T y))$, and the work is $\mathrm{poly}(\alpha, \varepsilon^{-1}, \log n) \cdot (\mathrm{work}(\Phi x) + \mathrm{work}(\Phi^T y) + O(m))$, where $\mathrm{depth}(\cdot), \mathrm{work}(\cdot)$ denote the depth and work required to compute the corresponding sub-problem respectively, and $\Phi x, \Phi^T y$ denote the sub-problems of multiplication between Φ and an arbitrary vector and the multiplication between Φ^T and an arbitrary vector respectively. Therefore, the problem effectively reduces to efficiently computing a good congestion approximator matrix Φ that also allows efficient computation of Φx and $\Phi^T y$.

Räcke, Shah, and Täubig [61] gave an efficient algorithm for building a *tree-structured* congestion approximator. Specifically, they showed that given any graph $G = (V, E, c)$, one can construct an $O(\log n)$-depth tree $R = (V_R, E_R, c_R)$ supported on $V_R \supseteq V$ such that for any demand vector b, $\mathsf{opt}_R(b) \leq \mathsf{opt}_G(b) \leq O(\log^4 n) \cdot \mathsf{opt}_R(b)$. A useful property of the tree R is that the flow that goes from u to the parent of u is equal to the total demand within the subtree of u. This implies that we can write $\mathsf{opt}_R(b) = \|\Phi b\|_\infty$ for a matrix Φ, where each row of Φ corresponds to a node in R, such that the u-th entry of Φb equals the congestion of the tree edge connecting u to its parent:

$$(\Phi b)_u = \frac{\sum_{v \text{ in the subtree of } u} b_v}{c_R(u, \text{parent of } u)}.$$

Furthermore, computing Φx is equivalent to computing the sum of rescaled node weights in each subtree, and computing $\Phi^T y$ is equivalent to computing the sum of rescaled edge weights of the path from each node to the root. Thus, both computations of Φx and $\Phi^T y$ can be done in $\mathrm{polylog}(n)$ depth and $n \cdot \mathrm{polylog}(n)$ work using standard dynamic programming ideas. Hence, our goal becomes to compute such a congestion approximator tree (or hereafter simply *congestion approximator*) R.

The algorithm of [61] constructs such a tree by performing a hierarchical decomposition of the graph. At a high level, starting with the vertex set V, they recursively apply a subroutine to partition a given cluster S of vertices into smaller sub-clusters such that the *boundary edges* leaving each sub-cluster are *well-linked*, in the sense that one can route a product multicommodity flow between them with low congestion. However, this partitioning routine itself requires computing $(1+\varepsilon)$-approximate minimum congestion flows on the induced subgraph $G[S]$, which is the exact same problem (though on a subgraph) we set out to solve by constructing a congestion approximator!

In the *sequential setting*, this chicken-and-egg situation was resolved by Peng [57] using a clever recursive construction, which we summarize below:

1. Given an input graph $G = (V, E, c)$, the goal is to output an $O(\log^4 n)$-congestion approximator tree $R = (V_R, E_R, c_R)$.

[2] To be precise, Sherman's algorithm upon termination routes a flow that *almost* satisfies the desired demand b, and the negligible residual demand is such that it can be routed in the flow network with $1/\mathrm{poly}(m)$ congestion. We get a feasible flow by trivially routing this residual demand along a maximum spanning tree, which is known to admit an efficient parallel construction with $\tilde{O}(m)$ work and $\mathrm{polylog}(n)$ depth [60].

2. Simulate the algorithm of [61] on G. **Every time** a $(1+\varepsilon)$-approximate minimum congestion flow computation is required on some **vertex-induced subgraph** $G[S]$ of G, do the following:
   (a) Compute a sparsifier H of $G[S]$ such that for any demand b, $\mathsf{opt}_H(b) \leq \mathsf{opt}_{G[S]}(b) \leq \text{polylog}(n) \cdot \mathsf{opt}_H(b)$. In addition, H can be decomposed into a core graph C and a forest F such that (i) C only has $|S|/\text{polylog}(n)$ nodes and edges, and (ii) each connected component of F has exactly one vertex in C.
   (b) **Recursively** compute an $O(\log^4 n)$-congestion approximator tree R_C for the core graph C.
   (c) Let $R_{G[S]} = R_C \cup F$. It is easy to show that by composing, $R_{G[S]}$ is a polylog(n)-congestion approximator tree for $G[S]$.
   (d) Compute a $(1+\varepsilon)$-approximate minimum congestion flow on $G[S]$ by plugging $R_{G[S]}$ into Sherman's algorithm.
3. Output an $O(\log^4 n)$-congestion approximator tree R for G obtained from simulating the algorithm of [61].

Note that since C has significantly smaller size than $G[S]$, the computation of a congestion approximator tree for C is much faster than for $G[S]$. Leveraging this observation, and using a large enough polylog(n) factor in the size reduction of the sparsifier, [57] showed that the overall running time of their algorithm can be bounded by $m \cdot \text{poly}(\varepsilon^{-1}, \log(n))$.

However, there are two major challenges to convert Peng's algorithm into a small depth PRAM algorithm. The first challenge is that Peng's algorithm has multiple recursive calls and the input to each recursive call depends on the output of previous recursive calls. These dependent recursive calls result in long dependent computation paths and thus the algorithm has a large (i.e. super-logarithmic) depth. The second challenge comes from how the approximate maximum flows are used in [61]. In particular, given an approximate maximum s-t flow, [61] crucially requires a decomposition of the flow into s-t flow paths.

Our paper's *main technical contribution* is to address these two challenges. To address the second challenge, we propose a novel efficient parallel flow decomposition routine which we discuss in more detail later in Section 2.2 of this overview. With this routine, we are already able to implement the algorithm of [61] with $m^{1+o(1)}$ work and $n^{o(1)}$ depth. However, getting the depth down to polylog(n) leaves us with the considerably more difficult task of addressing the first challenge. To this end, we devise a variant of the algorithm in [61] for constructing congestion approximators. While our new algorithm also computes a hierarchical decomposition of the graph, it allows us to apply our partitioning routine by running maximum flows on subgraphs obtained by *contracting* vertices, as opposed to recursing on vertex-induced subgraphs as done in [61]. This in particular allows us to *extract* congestion approximators for the subgraphs from a given congestion approximator of the entire graph, thus avoiding any additional recursive calls on these subgraphs. As a result, we only have *one* recursive call, whose goal is to compute a congestion approximator of the core graph C of the sparsifier, enabling us to obtain a depth of polylog(n).

We adopt a similar recursive idea to that of Peng [57], with the key difference being our approach involves only one recursive call. Specifically, our construction consists of a *recursive step* to obtain a low-quality (though still polylog(n)-approximate) congestion approximator, and a *boosting step* to improve it to an $O(\log^9 n)$-congestion approximator, so as to avoid an accumulation of error over successive recursive calls. We summarize our new construction as follows:

1. Given input $G = (V, E, c)$, the goal is to output an $O(\log^9 n)$-congestion approximator tree $R = (V_R, E_R, c_R)$.
2. Compute a sparsifier H of G such that for any demand b, $\mathsf{opt}_H(b) \leq \mathsf{opt}_G(b) \leq \text{polylog}(n) \cdot \mathsf{opt}_H(b)$. In addition, H can be decomposed into a core graph C and a forest F such that (i) C only has $|S|/\text{polylog}(n)$ nodes and edges, and (ii) each connected component of F has exactly one vertex in C.

3. (Recursive step) **Only one recursive call:** Recursively compute an $O(\log^9 n)$-congestion approximator tree R_C for the core graph C.

4. Let $R_G = R_C \cup F$. Then R_G is a polylog(n)-congestion approximator tree for G.

5. (Boosting step) Simulate our variant of the algorithm in [61] on G to find an $O(\log^9 n)$-congestion approximator tree R, where every time when $(1+\varepsilon)$-approximate minimum congestion flow computation is required on some subgraph $G(S)$ obtained from G by contracting vertices $V \setminus S$ into a single supernode, do the following:

   (a) Extract from R_G a polylog(n)-congestion approximator tree R_S for $G(S)$.

   (b) Compute a $(1+\varepsilon)$-approximate minimum congestion flow required on $G(S)$ by plugging R_S into Sherman's algorithm.

Line 2 involves constructing ultrasparsifiers [71, 46], and subsequently transforming them into j-trees [53]. Though not particularly technically challenging, we are the first to give efficient constructions of these objects in the PRAM model. We refer the reader to Section 5 for their parallel implementations. Line 5, namely our variant of the algorithm in [61], is substantially more involved, as we must open the blackbox of [61] entirely. In the following section, we explain in more detail, our new construction of congestion approximators that is used in our boosting step.

2.1 Overview of New Congestion Approximator Construction. As mentioned above, the difficulty in parallelizing the construction of congestion approximators in [61] arises from the fact that we need to solve (approximate) maximum flows on vertex-induced subgraphs, which in turn requires congestion approximators for those subgraphs. However, recursing on these subgraphs necessarily blows up the depth of our algorithm to super-logarithmic. There are two possibilities of resolving this: (i) use a more aggressive size reduction than polylog(n) when computing sparsifiers, or (ii) reduce the number of dependent recursive calls. Here, (i) can be immediately ruled out, since a more aggressive size reduction implies a worse approximation for the sparsifier, which in turn results in a larger iteration count (and hence, depth) of Sherman's algorithm to compute maximum flows. This leaves us with (ii) as the only option. Notice that having even just *two* dependent recursive calls would result in a depth of $2^{O(\frac{\log n}{\log\log n})} = n^{o(1)}$, which makes (ii) even more imperative.

This raises the following question: can we reduce the number of dependent recursive calls by *reusing* congestion approximators? While it indeed seems plausible to combine the congestion approximators for the subgraphs into one for the entire graph, this does not suit the construction of [61] - in particular, the hierarchical decomposition tree there is computed in a *top-down manner*, and thus we do not know on what subgraphs we want to run maximum flows until we are done with the partitioning at higher levels. This suggests that we should instead consider *extracting* congestion approximators for the subgraphs *from* that of the entire graph. But unfortunately, this is in general impossible due to the information loss due to congestion approximation (which in turn is a form of sparsification that preserves both cuts and flows).

Our solution here is to open the blackbox of [61] entirely and propose a new framework for computing a hierarchical decomposition. Our new framework allows us to partition the graph by running (approximate) maximum flows on *contracted subgraphs*, each of which is obtained by contracting a subset of vertices into a single *supernode*, as opposed to vertex-induced subgraphs as in [61]. We show that for the contracted subgraphs we encounter in our construction, we are actually able to extract congestion approximators for them from a given congestion approximator of the entire graph by contracting its vertices[3], thereby avoiding *any* additional recursive calls. Crucially, the congestion approximators

[3]This is a vast simplification of our extraction process. In particular, since we need our congestion approximator to be a tree in order to run Sherman's algorithm, we can only do certain "partial" contractions that preserves the tree structure, which significantly complicates both our algorithm and its analysis. We refer the reader to Section 7 for more detail.

obtained for the contracted subgraphs have size proportional to the size of the corresponding subgraphs which prevents any substantial blow up in the total work of our algorithm.

We next highlight some additional ideas needed to make our new construction work. Since we run maximum flows on contracted subgraphs, the routing we get could use paths in the contracted subgraphs, which are not necessary valid in the entire graph. Therefore we need a "fixing" step in our routing - namely, whenever we obtain a routing in a contracted subgraph, we have to then fix it so that it becomes a valid routing (with low congestion) in the entire graph, while routing the exact same demands[4]. This fixing step is possible because our construction guarantees that the edges incident on a contracted supernode (which we call *boundary edges*) are always well-linked, in the sense that one could route in the entire graph a product multicommodity flow between them with low (polylog(n)) congestion. Thus, we can convert all the flow paths passing through a supernode in a contracted subgraph into flow paths in the entire graph using the well-linkedness of these edges while blowing up the congestion by at most polylog(n).

As stated thus far, it may seem that too many contracted subgraphs might end up using a same edge in the entire graph to route and thus overcongesting the edge by too much. We address this issue by fixing the routing level-by-level in our hierarchical decomposition in a bottom-up manner; at each level, we only partially fix the routing in a contracted subgraph, in the sense that it becomes valid (i.e. routes the exact same demand) in the slightly larger contracted subgraph obtained one level above. We aim to maintain the invariant that in each contracted subgraph, the edges incident on the contracted supernode (i.e. boundary edges) are never congested by a factor larger than polylog(n). This invariant guarantees that in the (partial) fixing step, the edges inside the contracted supernodes do not get congested by a factor larger than polylog(n) either: any routing that utilizes these edges has to go through the boundary edges in the first place, and the latter are guaranteed to have low congestion.

However, notice that naively performing this (partial) fixing operation level-by-level does not get us the desired invariant. This is because each time we use well-linkedness of the boundary edges to fix the routing, we get a multiplicative polylog(n) blowup in congestion, which accumulates across all $\Theta(\log n)$ levels to a very large value. To address this issue, we use a similar trick as in [61] where, when solving maximum flows in each contracted subgraph, we lower the weights of the boundary edges by a large enough polylog(n) factor. Therefore, the actual congestion we get on these edges is in fact polylog(n) times smaller than what we get in our maximum flow routing, canceling out the multiplicative blowup caused by the fixing step. Moreover, a polylog(n)-congestion approximator for the original contracted subgraph remains a polylog(n)-congestion approximator for the reweighted contracted subgraph, albeit with a worse polylog(n) approximation factor. Therefore we can still use Sherman's algorithm to solve maximum flows on the reweighted subgraph in polylog(n) depth. Leveraging all these ideas, we get a parallel algorithm for computing an $O(\log^9 n)$-congestion approximator.

2.2 Overview of Our Parallel Flow Decomposition Routine. We now summarize the ideas behind our parallel flow decomposition routine that addresses our second challenge. Here we are given an s-t flow represented by a flow network, with each edge carrying some non-negative amount of flow, and our goal is to decompose the flow into s-t flow paths. We will decompose the flow in an iterative manner where in each iteration, we *shortcut* flow paths of length two in the flow network by replacing them with a single edge having the same flow value. We then repeat until (almost) all remaining edges directly go from s to t (i.e. all s-t paths have length 1).

In order to achieve small depth, in every iteration, we need to find a large (in capacity) collection of edge-disjoint length two flow paths so that they can be shortcut in parallel without interfering with each other. We show that such a collection exists, and moreover, can be found efficiently by formulating the problem as b-matching instances. Specifically, for every (directed) edge $e = (u, v)$ that does not directly connect s and t, we independently and randomly assign e either as an "outgoing" edge to its head vertex u, or as an "incoming" edge to its tail vertex v. Now taking a local view at any vertex w in the flow

[4] In a contracted subgraph, we only ever route demands that are supported entirely on the "uncontracted" vertices. Thus there is no ambiguity in saying "same demands" in contracted subgraphs vs. in the entire graph.

network, the task of shortcutting reduces to a b-matching problem in a complete bipartite graph, where the two sides are vertices corresponding to incoming edges into w and outgoing edges from w, respectively, with demands equal to their flow values. This way, we never have to worry about an edge being matched more than once as it gets assigned to exactly one of its two endpoints.

We show that in expectation over the random incoming/outgoing assignments of the edges, the total weight of the maximum b-matchings across all vertices must be at least a constant fraction of the total remaining flow that does not directly go from s to t. Moreover, algorithmically, in each b-matching instance, we can utilize a natural greedy strategy to gather at least a constant fraction of the maximum matched value, which can further be implemented in parallel across all vertices. This guarantees that in each iteration, the total ℓ_1-norm of the remaining flow shrinks by a constant factor, and thus after $O(\log n)$ iterations we have decomposed a $(1 - 1/\text{poly}(n))$-fraction of the total flow into s-t paths, which suffices for our purpose since we only want approximate maximum flows. At the end, we output the entire shortcutting history represented by a DAG data structure from which we can recover our desired information.

2.3 Algorithms. Lastly, we present all the key algorithms outlines in our overview. Our main algorithm is a recursive PRAM algorithm for computing a $O(\log^9 n)$-congestion approximator in $O(\text{polylog}\, n)$ depth and $\tilde{O}(m)$ total work. We present our algorithms for the case where the ratio between the largest and smallest capacities in the graph is bounded by $\text{poly}(n)$, but by the reduction shown in Section E, this is without loss of generality.

ALGORITHM 2.1. `congestion-approx`(G)
Input: Graph $G = (V, E, c)$ with $n = |V|, m = |E|$
Output: A $O(\log^9 n)$-congestion approximator for G.
Procedure:

1: $G_s \leftarrow$ `ultrasparsifier`(G, κ) with $\kappa = 10 \log^4 n$. ▷ Section 5.1.1
2: $(\mathcal{C}, \mathcal{E}) \leftarrow j$-`tree`$(G_s)$ with $\mathcal{C}$ the core and $\mathcal{E}$ the envelope of the j-tree. ▷ Section 5.1.2
3: // This is the only recursive call to `congestion-approx`
4: $R_{\mathcal{C}} \leftarrow$ `congestion-approx`$(\mathcal{C})$.
5: // $R_{\mathcal{C}} \cup \mathcal{E}$ is an $O(\log^{13} n)$-congestion approximator for G
6: $R' \leftarrow$ `tree-hierarchical-decomp`$(R_{\mathcal{C}} \cup \mathcal{E})$ ▷ Section 5.2
7: // From the call to `tree-hierarchical-decomp`, R' is a binary tree with $O(\log n)$ depth
8: $R \leftarrow$ `improve-CA`(G, R') ▷ Algorithm 2.2
9: **return** R

Our major technical contributions are in the new framework for computing congestion approximators, the implementation specifics of which are described in the `improve-CA` subroutine and the details of which are discussed in Section 6.

ALGORITHM 2.2. `improve-CA`(G, R)
Input: Graph $G = (V, E, c)$ and $O(\log^d n)$-congestion approximator R for G, with $9 < d \leq 30$.
Output: An $O(\log^9 n)$-congestion approximator for G.
Procedure:

1: $A_1, A_2 \leftarrow$ `partition-A`(G) ▷ Section 6.1
2: $B_1, B_2 \leftarrow$ `partition-B`(G, A_1, A_2) ▷ Section 6.2
3: // Claim 7.1 guarantees the edges leaving each Z_i are $O(1/\log^9 n)$-well-linked
4: Set $Z_1 = A_1 \cap B_1, Z_2 = A_2 \cap B_1, Z_3 = A_1 \cap B_2, Z_4 = A_2 \cap B_2$.
5: **for all** Z_i **do**
6: // R_i has size $\tilde{O}(|Z_i|)$ by Lemma 7.1
7: $R_i \leftarrow$ `ca-contraction`(R, Z_i) ▷ Section 7.1
8: // $G(Z_i)$ is a contracted subgraph (Definition 4.3), reweighted as in Section 6.3
9: $T_i \leftarrow$ `improve-CA`$(G(Z_i), R_i)$

10: Set R' as the tree formed by the T_i, using lines 11 and 12 of Algorithm 6.1.
11: **return** R'

THEOREM 2.1. *Given an undirected capacitated graph $G = (V, E, c)$ with n nodes and m edges, Algorithm 2.1 is a $\text{polylog}\, n$-depth, $O(m\ \text{polylog}\, n)$ total work* PRAM *algorithm which outputs a hierarchical decomposition tree that, with high probability, is a $O(\log^9 n)$-congestion approximator for G.*

Proof. The depth, work and correctness of all subroutines besides `improve-CA` follow from the proofs in their respective sections. Specifically, these sections show that R' passed to `improve-CA` is a $O(\text{polylog}\, n)$-congestion approximator for G which is a binary tree with depth $O(\log n)$, and that R' can be computed using $O(\text{polylog}\, n)$ depth and $\tilde{O}(m)$ work. Then, the correctness of `improve-CA` follows from Theorem 6.1, as does the depth and work of all but the `ca-contraction` calls. To bound the work from contracting the congestion approximators, by Lemma 6.1 we have that A_1, A_2, B_1, B_2 are both partitionings of V, and so $\bigcup_i Z_i = V$. Let $G(Z_i)$ be as defined in Definition 4.3, and properly modifying the capacities of some edges as in Section 6.3. As the Z_i's are a partitioning of nodes $V(G)$, each edge in G can be present in a most two contracted graphs $G(Z_i)$. Thus, $\sum_i |E(G(Z_i))| = O(m)$ as well. Constructing all contracted graphs $G(Z_i)$ can therefore be done in $O(m)$ work (and $O(1)$ depth), and so by Lemma 7.1, computing the contracted congestion approximators for all $G(Z_i)$ takes work $\tilde{O}(m)$ and depth $O(\log n)$. The total work and depth of the algorithm then follow from Theorem 6.1. ∎

Proof. (Theorem 1.1) Using the congestion approximator from Theorem 2.1, we can run the parallel version of Sherman's algorithm (Theorem 4.1) to get both a $(1-\varepsilon)$-approximate maximum s-t flow and a $(1+\varepsilon)$-approximate minimum s-t cut. ∎

2.4 Organization. We first discuss additional related work in Section 3 and then preliminaries in Section 4. We give details of our construction of congestion approximators in three subsequent sections. Firstly, Section 5 presents the construction of low-quality congestion approximators (outlined in Algorithm 2.1). Secondly, Section 6 presents the boosting step where the quality of the congestion approximator is improved (outlined in Algorithm 2.2). Finally, Section 7 presents details on how we extract congestion approximators for contracted subgraphs and utilize them within Sherman's framework [66]. We give details of our parallel flow decomposition algorithm in Section 8. We finally discuss applications of our approximate max-flows result in Section 9.

3 Other Related Work.

In addition to the related work we covered in the introduction, parallel maximum flow has also been studied for restricted classes of graphs. For example, when the flow network is a DAG with depth r, [16] gives a PRAM algorithm with near linear work and $\text{poly}(r, \log n)$ depth. Another problem related to the parallel maximum flow is parallel global minimum cut. The global minimum cut problem has been studied by [39, 38, 29]. The current best known algorithm [29] has $\text{polylog}(n)$ depth and $\tilde{O}(m)$ work.

In the sequential setting, the maximum flow problem on *directed* graphs has been extensively studied over the past ∼70 years. Ford and Fulkerson gave the first maximum flow algorithm using the idea of augmenting paths [25]. Since then, there has been a long line of work giving increasingly more computationally efficient algorithms for this problem; a partial list includes [21, 23, 41, 22, 24, 28, 34, 33, 32, 9, 30, 18, 37, 10, 49, 31, 54, 52, 42, 14]. When supplies and demands are polynomially bounded integrals, [14] provides the current fastest maximum flow algorithm which achieves a running time of $m^{1+o(1)}$ by leveraging interior point methods, which are powerful second-order continuous optimization methods. When considering *undirected* maximum flow problem, continuous optimization techniques have also been used to obtain fast $(1-\varepsilon)$-approximate maximum flow algorithms including, e.g., [15, 44, 48, 66, 43, 57, 67, 68]. Instead of using IPMs, these algorithms use simpler first-order methods.

In the parallel computation setting, flow problems other than maximum flow/minimum congestion flow have also been widely studied. As we have discussed in previous sections, the goal of maximum

flow/minimum congestion flow problem is to minimize the ℓ_∞ norm of the (rescaled) flow vector while satisfying the flow conservation constraints. Alternatively, if the objective is to minimize the ℓ_1 norm of the (rescaled) flow vector while satisfying the same constraints, the problem is known as the uncapacitated minimum cost flow problem. This problem can be seen as a generalization of the shortest path problem. There is a line of work of parallel undirected uncapacitated minimum cost flow algorithms that are based on Sherman's framework [68] as well. In particular, [5, 50, 64] provide efficient procedures to compute a polylog(n)-approximate ℓ_1-oblivious routing scheme (ℓ_1 preconditioner), and thus they are able to apply Sherman's framework [68] to compute a $(1-\varepsilon)$-approximate uncapacitated minimum cost flow for undirected graphs using polylog(n) depth and $\tilde{O}(m)$ work. If the goal is to minimize the ℓ_2 norm of the (rescaled) flow vector while satisfying the flow conservation constraints, the problem is known as the electric flow problem. The efficient parallel electric flow (ℓ_2-flow) algorithm essentially reduces to an efficient (approximate) solver for symmetric, diagonally dominant (SDD) linear systems due to [58]. In particular, [58] provides a parallel algorithm to solve SDD systems with n dimensions and m non-zero entries in polylog(n) depth and $\tilde{O}(m)$ work, and thus the electric flow problem can be solved in the same complexity. Interestingly, by plugging the parallel SDD system solver into the framework of [54] to deal with the computation of each Newton step in the IPMs, it implicitly gives a parallel maximum flow algorithm for directed graphs with $\tilde{O}(\sqrt{m})$ depth and $\tilde{O}(m^{1.5})$ work (and with $1/\text{poly}(n)$ additive error).

Finally, we also note that a recent paper by Forster et al. [27] used a relevant technique in [16] to round fractional flows to integral ones in their distributed maximum flow algorithms. However, we emphasize that the scheme of [16] is for flow rounding rather than flow decomposition, i.e. rounding fractional flows to integer flows, as opposed to finding the actual s-t flow paths. The latter is crucial for our application in cut matching games, as the matching player needs to compute a (fractional) matching between the sources and sinks from a given s-t flow.

4 Preliminaries.

Graphs. We represent an n-vertex, m-edge undirected, capacitated graph as $G = (V, E, c)$ with V being the vertices (nodes), E being the edges, and c being the edge capacities (weights). We also use $V(G)$ and $E(G)$ to denote the vertices and edges of the graph G, respectively. We will interchangeably write the capacity of an edge e as $c(e)$ or c_e. Sometimes the same edge e appears in two different graphs with different capacities, thus we write $c^H(e)$ or c_e^H to denote the weight of e in graph H to avoid ambiguity. In Section E, we show how to transform a graph with arbitrary capacities into one with polynomial aspect ratio while only reducing the maximum s-t flow by a $(1-\varepsilon)$ factor, for any $s, t \in G$ and $\varepsilon > 0$. Thus, throughout the paper we assume that graphs have poly(n/ε) aspect ratio.

We usually need to solve maximum flows by adding a source and a sink, for which we set up the following notation.

Definition 4.1. (Graphs with Sources and Sinks) *Given a graph H with two (not necessarily disjoint) vertex subsets $V_1, V_2 \subseteq V(H)$, and a parameter $c_u \geq 0$ for each $u \in V_1$, and a parameter $d_u \geq 0$ for each $u \in V_2$, we write H_{st} to denote the graph obtained from H by doing the following:*

1. *Add a vertex s, and connect s to each $u \in V_1$ with capacity c_u;*
2. *Add a vertex t, and connect t to each $u \in V_2$ with capacity d_u.*

Congestion Approximators and Sherman's Algorithm. Given a graph $G = (V, E, c)$ and a demand vector b, let $\mathsf{opt}_G(b) = \min_{f: Bf=b} \|C^{-1}f\|_\infty$ be the optimal congestion of any flow routing a given demand vector b in the graph G, with B being the edge-vertex incidence matrix of G.

We first define congestion approximators. Note that in this paper, we *always* consider congestion approximators that are trees, which are readily plugged into Sherman's algorithm.

Definition 4.2. (Congestion Approximators) *Given a graph $G = (V, E, c)$, a tree $R = (V_R, E_R, c_R)$ with $V \subseteq V_R$ is an α-congestion approximator of G for some $\alpha \geq 1$ if for any demand vector b, we have*

$$\mathit{opt}_R(b') \leq \mathit{opt}_G(b) \leq \alpha \cdot \mathit{opt}_R(b'),$$

where b' is obtained by appending b with zeros on entries $V_R \setminus V$.

We next state the parallel version of Sherman's algorithm. While we believe it is known that Sherman's algorithm can be implemented with low depth *modulo the computation of a good congestion approximator*, we give a discussion of such an implementation in Appendix C for completeness.

THEOREM 4.1. (SHERMAN'S ALGORITHM [66], PARALLEL VERSION) *There is a* PRAM *algorithm that, given a graph $G = (V, E, c)$, an α-congestion approximator $R = (V_R, E_R, c_R)$ of G, a source $s \in V$ and a sink $t \in V$, and an $\varepsilon \in (0, 1)$, computes a $(1-\varepsilon)$-approximate maximum s-t flow as well as a $(1+\varepsilon)$-approximate minimum s-t cut in G. The total work of the algorithm is $O((|E|+|E_R|)\alpha^2\varepsilon^{-3}\operatorname{polylog}(n))$, and the depth of the algorithm is $O(\alpha^2\varepsilon^{-3}\operatorname{polylog}(n))$.*

Contracted Subgraphs. We give the definition of contracted subgraphs below. We use X to denote the subset of vertices that we contract, and use $S = V(G) \setminus X$ to denote the remaining vertices.

DEFINITION 4.3. (CONTRACTED SUBGRAPHS) *Given a graph $G = (V, E, c)$ and a subset of vertices $S \subseteq V(G)$, we denote by $G(S)$ the graph obtained from G by contracting $X := V \setminus S$ into a single supernode u_X, deleting any resulting self-loops, and keeping all resulting parallel edges.*

We now define reweighted contracted subgraphs which will be useful in our new congestion approximator construction in Section 6.

DEFINITION 4.4. (REWEIGHTED CONTRACTED SUBGRAPHS) *Given a graph $G = (V, E, c)$ and a pair $\mathcal{P} = (S, \omega)$, where $S \subseteq V(G)$ is a subset of vertices, and $\omega : E(S, V(G) \setminus S) \to (0, 1]$ is a real-valued* reweighting function*, we denote by $G(\mathcal{P})$ the graph obtained from G by doing the following operations:*

1. *Contract $X := V \setminus S$ into a single supernode u_X, deleting any resulting self-loops, and keeping all resulting parallel edges;*
2. *Scale the weight of each (parallel) edge e incident on u_X by a factor of $\omega(e)$.*

We call u_X the contracted vertex, and refer to other vertices (a.k.a. S) in $G(\mathcal{P})$ as uncontracted vertices. We refer to the edges in $G(\mathcal{P})$ that are incident on u_X the boundary edges.

Subdivision Graphs and Well-Linked Edges. We recall the definition of subdivision graphs from [61].

DEFINITION 4.5. (SUBDIVISION GRAPHS) *Given a capacitated graph $G = (V, E, c)$, we write $G' = (V', E', c')$ to denote the subdivision graph of G, defined as follows. To obtain G', we split each edge $e = (u, v)$ in G by introducing a new* split vertex *(or subdivision vertex) x_e and two split edges (u, x_e) and (x_e, v), both with capacity c_e. In words, we have $V' = V \cup X_E$ where X_E is the set of split vertices of the edges in E, $E' = \bigcup_{e=(u,v)\in E} \{(u, x_e), (x_e, v)\}$, and $c'_{(u,x_e)} = c'_{(v,x_e)} = c_e$ for every edge $e \in E$.*

For a subset of edges $F \subseteq E$, we write X_F to denote the set of split vertices of edges in E, and c_F (or equivalently, $c(F)$) to denote the total capacity of edges in F.

DEFINITION 4.6. *Given $G(\mathcal{P})$ and another reweighting function $\tilde{\omega}$ supported on the split edges of the boundary edges of $G(\mathcal{P})$ in $G'(\mathcal{P})$. Then we define the graph $G'(\tilde{\mathcal{P}} = (\mathcal{P}, \tilde{\omega}))$ to be the graph obtained from $G'(\mathcal{P})$ by reweighting the split edges of the boundary edges by $\tilde{\omega}$. That is, for an edge e that is a split edge of a boundary edge, we have $c_e^{G'(\tilde{\mathcal{P}}=(\mathcal{P},\tilde{\omega}))} = \tilde{\omega}(e) \cdot c_e^{G'(\mathcal{P})}$.*

We next recall the notion of well-linked edges as used in [61].

DEFINITION 4.7. (WELL-LINKED EDGES) *For a capacitated graph $G = (V, E, c)$, a set $F \subseteq E$ of edges of total capacity c_F is called* β-well-linked *for some $\beta \in (0, 1]$ if there is a feasible multicommodity flow in the subdivision graph G' that sends $(\beta c_e c_f / c_F)$ units of flow from x_e to x_f for all pairs $e, f \in F$ simultaneously.*

Any subset of well-linked edges are also well-linked.

PROPOSITION 4.1. *If set $F \subseteq E$ of edges is β-well-linked in $G = (V, E, c)$, then any subset $F' \subseteq F$ of F is β-well-linked.*

Proof. Given a product demand on F', we can first let each edge distribute its demands uniformly to all edges in F with each edge getting flow proportional to its capacity. This can be done with congestion $1/\beta$ since edges in F are β-well-linked. Then for each demand between two vertices s, t in F', we can compose the routings which distributed the flow from s and t to F. The proposition thus follows. ∎

PRAM Primitives. We state a few basic PRAM primitives, whose implementation is discussed in Appendix A for completeness.

PROPOSITION 4.2. (PREFIX SUMS [3]) *There is a* PRAM *algorithm that, given a list of values $a_1, \ldots, a_n$, computes all prefix sums $\sum_{i \le k} a_i$'s. The algorithm has $O(n)$ total work and $O(\log n)$ depth.*

PROPOSITION 4.3. (SUBTREE SUMS) *There is a* PRAM *algorithm that, given a tree $T = (V_T, E_T, w_T)$ where $w_T : V_T \to \mathbb{R}$ is a vertex weight function, a vertex $r \in V_T$, computes all subtree sums of the weight function w_T with the r being the root of tree. The algorithm has total work $O(|E_t|)$ and total depth $O(\log |E_T|)$.*

5 Recursive Computation of a Low-Quality Congestion Approximator.

In this section, we detail the steps of Algorithm 2.1 up to the call to `improve-CA` (which is detailed in Algorithm 2.2 and Section 6). That is, in this section we detail how to recursively compute a α'-congestion approximator for G which is a $O(\log n)$ depth binary hierarchical decomposition tree of G where $\alpha' = O(\text{polylog}\, n)$. Section 6 then details how to improve the quality of the congestion approximator to $O(\log^9 n)$.

5.1 Reducing the Size of the Graph.

5.1.1 Constructing an Ultrasparsifier. To reduce the size, our first step is to construct an ultrasparsifier of the graph. An ultrasparsifier, informally speaking, consists of a spanning tree plus some additional edges that approximately preserves all graph cuts.

DEFINITION 5.1. (ULTRASPARSIFIER) *Given a graph $G = (V, E, c)$ with n nodes and m edges, and any parameter $\kappa \ge 1$, a κ-ultrasparsifier $G_s = (V, E_s, c_s)$ is the union of a spanning tree T of G and a collection of edges E' such that $|E'| = O(m \log^2 n/\kappa)$. Moreover, with high probability, the capacity of every cut of G is preserved to within a κ factor in G_s.*

In Algorithm 2.1, we use $\kappa = 10 \log^4 n$ to ensure a polylog n reduction in size compared to the original graph.

LEMMA 5.1. *There exists a randomized* PRAM *algorithm that given an undirected, capacitated graph $G = (V, E, c)$, and any desired quality parameter $\kappa \ge 1$, constructs a κ-ultrasparsifier of G with $O(\log n)$ depth, and $O(m)$ total work.*

Proof. The algorithm first computes a maximum spanning tree using the result of [60], which uses $O(\log n)$ depth and $O(m)$ work. Then, scale up the weights on the tree edges by a factor of κ, and sample each remaining edge with probability $\Theta(\log^2 n/\kappa)$.

Sampling can easily be done in parallel, and the number of sampled edges in expectation is also immediately as desired. So, it remains to show that the final constructed graph preserves all cuts to within a $O(\kappa)$ factor with high probability.

Let T be the constructed maximum spanning tree of G. Multiplying the weight of each edge of T by κ results in a worst-case cut distortion factor of κ. So, after scaling, we wish to select additional edges in

order to approximate cuts (in the graph with newly scaled edges) to within an $O(1)$ factor. [26] shows that to approximate edges cuts to within a $(1+\varepsilon)$ factor, it is sufficient to sample each edge e with probability at least $Cc_e \log^2 n/(\rho_e \varepsilon^2)$, where ρ_e is the edge-connectivity between the endpoints of e, c_e is the capacity of edge e, and C is a suitably large constant. Here, the connectivity of an edge $e=(u,v)$ is defined as the capacity of the minimum cut separating u from v.

As our algorithm samples off-tree edges (that is, edges in G but not T) at rate $\Theta(\log^2 n/\kappa)$, to complete our analysis it suffices to show that the connectivity of any off-tree edge $e=(u,v)$ is at least $c_e\kappa$. Since e is off-tree, adding it to T creates a cycle. Moreover, T is the maximum weight spanning tree, so $c_e \le c_{e'}$ for all e' on the cycle created. Any cut separating u from v must also involve some tree edge e' from this cycle, so the capacity of any cut (after scaling) is at least $c_{e'}\kappa$. Thus, the connectivity of e is also at least $c_{e'}\kappa \ge c_e\kappa$. ∎

5.1.2 Constructing a j-tree. After constructing a $O(\text{polylog}(n))$-ultrasparsifier, we now have a graph G_s which is the union of a spanning tree T and collection of off-tree edges E'. We now transform the ultrasparsifer so that we only need to compute the congestion approximator for a smaller subgraph. Namely, we convert the ultrasparsifer into a j-tree. A j-tree, introduced in [53], is a graph consisting of a forest with j trees (called the *envelope*) connected together by an arbitrary graph (called the *core*) over j vertices, one from each tree.

DEFINITION 5.2. (j-TREE [53]) *A weighted graph $G=(V,E,c)$ is a j-tree if it is a union of a core graph H which is an arbitrary capacitated graph supported on at most j vertices of G, and a forest on V such that each connected component in the forest has exactly one vertex in H. We call this forest the envelope of the j-tree.*

Lemma 5.8 of [53] shows how to sequentially convert any ultrasparsifer to an $O(j)$-tree, where j is the number of nodes incident on off-tree edges in the ultrasparsifer. In this section, we show how to adapt this sequential process to the PRAM setting.

The algorithm of [53] for converting an ultrasparsifier to a j-tree first recursively removes degree 1 vertices. Let F be the subgraph consisting of these deleted nodes and edges; F is thus a forest, and all removed edges come from the spanning tree T of the ultrasparsifier. Unfortunately, this process is inherently sequential, so we instead show that an alternative process, which can easily be parallelized and identifies this forest of removed nodes.

LEMMA 5.2. *Let G_s be a $O(\text{polylog}(n))$-ultrasparsifier with spanning tree T and off-tree edges E'. Suppose T is rooted at an arbitrary node incident on some off-tree edge. Then, a node u is removed by recursive deletion of degree 1 vertices if and only if the subtree of T rooted at u contains no nodes incident on off-tree edges. Moreover, there is a* PRAM *algorithm with $O(\log n)$ depth and $O(m)$ work which removes all such nodes.*

Proof. Let r be the root of T, and let F be the set of nodes removed by recursively deleting degree 1 vertices (so F induces a forest). Define X as the set of nodes incident on some off-tree edge. If the subtree of u contains no nodes in X, then it immediately follows that u (and all of its descendants) must be in F because they induce a tree.

Now, suppose some node in the subtree of u is incident on an off-tree edge. Let p be the parent of u, let v be a child of u such that the subtree rooted at v also contains a node in X, and let H_u be the graph formed by removing u from G_s. As $r \in X$, both p and v are connected to X in H_u. If p, v are in the same connected component of H_u, then u is in a cycle, and so it will never be removed by the recursive deletion procedure.

Otherwise, p, v are connected in H_u to distinct elements of X; p is connected to r and v is connected to some $x \in X$ such that $x \neq r$. As p and v are both neighbors of u, there exists in G_s a $r \to x$ path P such that $u \in P$. Consider the rounds of the recursive deletion procedure, where in each round the procedure removes all degree 1 vertices currently present in the graph. If before round i, no nodes of P have been deleted, then after round i, none of P has been removed as well: the interior points have 2

neighbors in the path, thus having degree at least 2, and the endpoints are never removed as they are incident on off-tree edges. The entire path is trivially present before the procedure begins, and so no nodes in P are ever removed. Thus, as $u \in P$, $u \notin F$, as desired.

The algorithm to find F is as follows: first, root T at an arbitrary node incident on some off-tree edge. Construct node weights by setting $w_u = 1$ if u is incident on some off-tree edge, and $w_u = 0$ otherwise. Then, run the subtree sum algorithm of Theorem A.4, and output all nodes whose subtree sum is 0. Identifying which nodes are adjacent to off-tree edges requires at most $O(1)$ depth and $O(m)$ work, and the subtree sum algorithm has depth $O(\log n)$ and total work $O(n)$, giving the desired runtime bounds. ∎

Let G' be the graph after recursively deleting degree 1 vertices, or, equivalently by Lemma 5.2, removing all nodes whose subtree contains no nodes incident on off-tree edges. The next step is to find paths in G' involving entirely degree 2 vertices except for the end points u, v which have degree at least 3, and "move" capacity from the lowest weight edge into an edge between the endpoints of the path. In Lemma 5.8, [53] proves that after reconnecting the vertices removed by recusively deleting degree 1 vertices, this results in $(3j-2)$-tree, where j is the number of nodes incident on off-tree edges in G_s.

ALGORITHM 5.1. `transform-paths-and-cycles`(G')
Input: Graph G' with no degree 1 vertices, and j nodes incident on off-tree edges.
Output: A $O(j)$-tree G''.
Procedure:

1: Initialize $W \leftarrow \{u \mid \deg(u) = 2\}$, $\mathcal{P} = \emptyset, \mathcal{C} = \emptyset$, and $G'' = G'$.
2: Construct G_d by removing $V \setminus W$ (i.e. nodes of at least degree 3) from G'.
3: Run connectivity to identify connected components $C_1, \ldots C_q$ of G_d.
4: **for** each component C_i **do**
5: Find $U_1 \subseteq C_i$, the set of degree 1 vertices in C_i.
6: **if** $|U_1| = 0$ **then**
7: Add C_i to $\mathcal{C}$.
8: **else**
9: Let $U_1 = \{u_1, u_2\}$, and let $v_1, v_2 \in V \setminus W$ be the additional neighbors in G' of u_1, u_2.
10: **if** $v_1 = v_2$ **then**
11: Add the cycle formed by C_i and v_1 to $\mathcal{C}$.
12: **else**
13: Add the path from $v_1 \to v_2$ through C_i to $\mathcal{P}$.
14: **for** each $S \in \mathcal{P} \cup \mathcal{C}$ **do**
15: Find $e_{\min}$, the edge with minimum capacity in S.
16: For all other $e \in S$, update capacity $w(e) = w(e) + w(e_{\min})$ in G''.
17: If $S \in \mathcal{P}$ with endpoints (v_1, v_2) and $(v_1, v_2) \notin E(G'')$, create edge (v_1, v_2) with capacity $w(e_{\min})$.
18: Delete $e_{\min}$ from G''.
19: **return** G''.

THEOREM 5.1. (PARALLEL VERSION OF LEMMA 5.8 OF [53]) *Algorithm 5.1 uses $O(\log n)$ depth and $\tilde{O}(m)$ total work to construct a $O(j)$-tree, where j is the number of nodes incident on an off-tree edge of G_s.*

Proof. For runtime, in Line 3 we may use the connectivity algorithm shown in [69], which has $O(\log n)$ depth and $\tilde{O}(m)$ work. Finding the additional neighbors can be identified in $O(1)$ depth and $O(n)$ work, along with identifying whether they induce a path or a cycle, and a $O(\log n)$ depth algorithm can correctly update the capacities.

For correctness, all constructed paths have endpoints of degree 3. After removing all nodes of degree at least 3, the remaining nodes all have degree 2 in G', as G' has no degree 1 vertices by Lemma 5.2. So,

every degree 2 vertex and every edge between degree 2 vertices is in exactly one component, and thus every edge is in some path or cycle and all paths and cycles are edge disjoint.

The result then follows from the proof Lemma 5.8 in [53]. ∎

Once we have a $O(j)$-tree J, to identify the core on which we wish to construct a congestion approximator, we first find a spanning tree of J. We then invoke the procedure of Lemma 5.2 with respect to this spanning tree to remove any nodes not present in the core. Then by Lemma 5.8 in [53], the constructed j-tree preserves all cut values to within a polylog(n) factor.

Finally, we then recursively run Algorithm 2.1 on the core of the j-tree, and attach the trees in the envelope (where envelope is defined as in Definition 5.2).

5.2 Hierarchical Decomposition on Trees. After recursing on the core of the j-tree, we have a congestion approximator R' for G. However, the depth of R' may be very large, as there is no bound on the depth of trees which make up the envelope of the j-tree. Several parts of our algorithm, such as the procedure to reduce the size of the congestion approximator (Section 7.1), rely on R' having $O(\log n)$ depth. As such, before running the procedure to improve the quality of the congestion approximation of R', we emulate the hierarchical decomposition procedure of [61] *on the tree* R'. That is, we take as input the tree R', and output a hierarchical decomposition tree with depth $O(\log n)$ which is an $O(\log^4 n)$[5] congestion approximator for R'. Moreover, to bound the work of the contraction of congestion approximators in Section 7.1, we require the output congestion approximator to be a binary tree.

The goal of this section is to prove the following theorem:

THEOREM 5.2. (HIERARCHICAL DECOMPOSITION ON TREES) *Let T be a tree on n nodes. There is a $O(\log^3 n)$ depth, $\tilde{O}(n)$ work algorithm to compute an $O(\log^4 n)$-congestion approximator R for T which is a hierarchical decomposition with depth $O(\log n)$. Moreover, in the same depth and work, we can ensure that R is a binary tree.*

The algorithm makes use of a *tree separator node*, which is any node q of a tree T such that removing q results in a forest of components of size at most $|T|/2$.

DEFINITION 5.3. (TREE SEPARATOR NODE) *A node q in a tree T is called a tree separator node of T if the forest induced by $T \setminus \{q\}$ consists of trees with at most $|T|/2$ nodes.*

Lemma A.2 gives a $O(\log |T|)$ depth, $O(|T|)$ work PRAM algorithm to find a tree separator node for any tree T.

With this definition, we now describe the algorithm for computing a hierarchical decomposition on trees. Each level in the hierarchical decomposition algorithm of [61] consists of two partitioning steps. Given a set P of nodes, the first determines an $\Omega(1/\log^2 n)$-well-linked (see Definition 4.7) set of edges within the graph induced by P and for the second, it suffices to find an exact min-cut between these well-linked edges and the edges leaving P (see Lemma 3.9 of [61] for more details)[6].

DEFINITION 5.4. (PARTITION A) *Let P be a subset of nodes of a graph $G = (V, E, c)$, and let $G'[P]$ be the subdivision graph (see Definition 4.5) of the subgraph of G induced by P. Then, Partition A is a partitioning of P into clusters $P_1, \ldots, P_w$ such that the set of inter-cluster edges in $G'[P]$, namely $\{(u, v) \mid u \in P_i, v \in P_j, i \neq j\}$, is $\Omega(1/\log^2 n)$-well-linked. Moreover, we have $|P_i| \leq (1/2)|V(G)|$ for all i.*

For the definition of Partition B, we state it in terms of an exact min-cut, which is stronger than what [61] or Section 6 uses, since when the graph is a tree, we are able to compute exact min-cut efficiently in low depth (see Section D for more details).

[5] As this loss of $O(\log^4 n)$ suffices for our purposes, we do not attempt to improve it, and instead rely on the analysis of [61] in a near-black-box fashion.

[6] We reuse the names "Partition A" and "Partition B" used in Section 6, to highlight the similar goals of the procedures `partition-A` and `partition-B`, however we are not invoking or directly implementing these procedures.

Definition 5.5. (Min-Cut Partition B) *Let P be a subset of nodes of a graph $G = (V, E, c)$, let B_P be set of edges with one endpoint in P and one endpoint in $V \setminus P$. Construct a graph H as follows. Start with $G'[P]$, and for each edge e in B_P, add a node x_e to H. Finally, for each $e = (u, v)$ in B_P such that $v \in P$, add an edge (x_e, v) with capacity $c(e)/\log n$. Then, with W the set of nodes of H which corresponds to the set of subdivision nodes of the inter-cluster edges from Partition A, Partition B is a partitioning $(X, P \setminus X)$ such that $(X, P \setminus X)$ is a min-cut between B_P and W in the graph H.*

The analysis of [61] and Section 6 shows that to get a polylog(n) depth, $\tilde{O}(n)$ work PRAM algorithm to compute a hierarchical decomposition for a tree, it suffices to implement Partition A and Partition B in polylog(k) depth and $\tilde{O}(k)$ work on a set of k nodes. We now give a pair of lemmas showing that we can indeed do this.

Lemma 5.3. (Computing Partition A on Trees) *There is a $O(\log k)$ depth, $\tilde{O}(k)$ work PRAM algorithm to compute Partition A on a set of k nodes which induce a tree.*

Proof. Let T be the tree induced by the vertex set we wish to partition. Then, compute a tree separator node c of T, and set the partitions to be the connected components of $T \setminus \{c\}$ along with $\{c\}$. The depth and work are then $O(\log k)$ and $\tilde{O}(k)$, using the algorithms for finding a tree separator node and connected components.

The only edges cut are those incident on the center c. The set of edges incident on the center are 1-well-linked, since they share an endpoint, meeting the requirements for Partition A. ∎

Lemma 5.4. (Computing Partition B on Trees) *Given a tree T, there is a $O(\log^2 k)$ depth, $\tilde{O}(k)$ work PRAM algorithm to compute Partition B on the subgraph of T induced by a set K of k nodes.*

Proof. First, note that the subgraph induced by K is a forest, as T is a tree. Then, with $P = K$, define B_P, W, and H as in Definition 5.5. Note that as the subgraph induced by K in T is a forest, H is also a forest. Add to H a super-source s connected to each node of B_P with capacity ∞ and a super-sink t connect to each node of W with capacity ∞. Then, the result follows from computing a exact s-t min-cut on H (using Lemma D.2)[7]. ∎

These lemmas allow us to compute hierarchical decompositions on trees.

Lemma 5.5. *Given a tree T on n nodes, there is an $O(\log^3 n)$ depth, $\tilde{O}(n)$ work PRAM algorithm to compute an $O(\log^4 n)$-congestion approximator R for T which is a hierarchical decomposition with depth $O(\log n)$.*

Proof. We follow the algorithm of [61] to construct a congestion approximator for T. As each level of the constructed congestion approximator in [61] corresponds to a partitioning of the edges, computing Partition A and B for all sets at a given level can be done in $O(\log^2 n)$ depth and $\tilde{O}(n)$ work. The use of tree separator nodes leads to a constant factor size reduction from each Partition A step, and so there are at most $O(\log n)$ levels, leading to a $O(\log^3 n)$ depth, $\tilde{O}(n)$ work algorithm. Finally, the analysis of [61] shows that the constructed hierarchical decomposition is a $O(\log^4 n)$-congestion approximator for the original tree, and that the tree has $O(\log n)$ depth. ∎

It thus remains shows how to convert the congestion approximator R into a binary tree; we describe the procedure in Algorithm 5.2. In Algorithm 5.2, we use the term *first non-binary node* to refer to any node $u \in R$ such that none of the ancestors of u have more than two children, but u has at least three children. Since R has depth $O(\log n)$, a breadth-first traversal can find all first non-binary nodes in $O(\log n)$ depth and $O(n)$ work.

Finally, for any node u of a hierarchical decomposition tree R, we use the notation R_u to refer to the subtree of R rooted at u, and P_u the set represented by node u.

[7] Lemma D.2 is written for a tree with the addition of a source s and sink t, rather than a forest; as such, we first add a dummy node y that connects to one node of every component of the forest with a capacity 0 edge, which does not change the min-cut value and allows us to use Lemma D.2.

ALGORITHM 5.2. `convert-binary`(R)
Input: Hierarchical decomposition tree $R = (V_R, E_R, c_R)$ which is a α-congestion approximator for some tree T.
Output: Binary tree R' which is a α-congestion approximator for T.
Procedure:

1: Compute the set $\mathcal{B}$ of all first non-binary nodes in R
2: // R_u is the subtree of R rooted at u
3: Let $\mathcal{F} = \bigcup_{u \in \mathcal{B}} R_u \setminus \mathcal{B}$ be the forest of descendants of all first non-binary nodes
4: Initialize $R' \leftarrow R \setminus \mathcal{F}$
5: **for** each first non-binary node $u \in \mathcal{B}$ **do**
6: // P_{v_i} is the set represented by v_i in R, and $q \geq 3$ as $u \in \mathcal{B}$
7: Let $v_1, \ldots, v_q$ be the children of u such that $|P_{v_1}| \geq |P_{v_2}| \geq \ldots |P_{v_q}|$
8: Set s such that s is the smallest index such that $|P_{v_1} \cup \ldots \cup P_{v_s}| \geq |P_u|/4$
9: Add to R' new nodes z_1, z_2 as children of u with edges of capacity ∞. Moreover, in R', z_1 represents $P_{v_1} \cup \ldots \cup P_{v_s}$ and z_2 represents $P_{v_{s+1}} \cup \ldots \cup P_{v_q}$
10: **for** $1 \leq i \leq s$ **do**
11: Add R_{v_i} to R' by adding edge (z_1, v_i) with capacity $c_R((u, v_i))$
12: **for** $s + 1 \leq i \leq q$ **do**
13: Add R_{v_i} to R' by adding edge (z_2, v_i) with capacity $c_R((u, v_i))$
14: If z_1 (or z_2) has exactly one child v, remove z_1 (or z_2) and connect v to u with an edge of capacity $c_R((u, v))$.
15: If R' is binary, return R'. Otherwise, return `convert-binary`(R').

To prove the correctness of Algorithm 5.2, we first prove the following helper lemma.

LEMMA 5.6. *Let $R' =$ `convert-binary`(R), and consider x a node of R'. Let y be the parent of x, and let z be the parent of y. Then, $|P'_x| \leq (3/4)|P'_z|$, where for any $w \in R'$, P'_w is the set represented by w in R'.*

Proof. Every node in R is present in R', as each call to `convert-binary` adds nodes but does not remove any. As such, $V(R) \subseteq V(R')$. We again use the notation P_w to denote the set represented by a node w in R, and $P'_w = P_w$ for all $w \in R$. Call any node in $R' \setminus R$ an intermediary node. First, consider a node x in R' whose parent y is not an intermediary node; that is, $y \in R$. If $x \in R$ as well, then by Lemma 5.3, $|P_x| \leq (1/2)|P_y|$. So, suppose x is an intermediary node, and thus P'_x is a union of sets $P_{w_1}, \ldots, P_{w_q}$, where $w_1, \ldots, w_q$ are the children of y in R. As in Algorithm 5.2, set s to be the smallest index such that $|P_{w_1} \cup \ldots \cup P_{w_s}| \geq (1/4)|P_y|$, and let $S_1 = P_{w_1} \cup \ldots \cup P_{w_s}$ and $S_2 = P_{w_{s+1}} \cup \ldots \cup P_{w_q}$. From the fact that $|P_{w_i}| \leq (1/2)|P_y|$ (which is again from Lemma 5.3) and the setting of s, we have that $(1/4)|P_y| \leq |S_1| \leq (3/4)|P_y|$. Since $S_1 \cup S_2 = P_y$, the same inequality holds for $|S_2|$. Thus, as either $P'_x = S_1$ or $P'_x = S_2$, we have that $|P'_x| \leq (3/4)|P'_y|$. Since R' is a hierarchical decomposition, if z is the parent of y, $P'_y \subseteq P'_z$, and thus all nodes which are the child of a non-intermediary node satisfy the lemma.

Now, consider a node $x \in R'$ whose parent y is an intermediary node. Let z be the parent of y. As R' is a hierarchical decomposition by construction, there exists a set $S' \subseteq V(R)$ such that $P'_z = \bigcup_{w \in S'} P_w$ and a set $S \subset S'$ such that $P'_y = \bigcup_{w \in S} P_w$. If for all $w \in S$, $|P_w| \leq (1/2)|P'_z|$, then by the same argument used in the case where y was non-intermediary, $|P'_x| \leq |P'_y| \leq (3/4)|P'_z|$. Suppose, for contradiction, there exists a $w \in S$ such that $|P_w| > (1/2)|P_z|$. Since Algorithm 5.2 sorts the nodes in Line 7 in decreasing order of size of the sets they represent, if S contains an element w such that $|P_w| > (1/2)|P_z|$, w must be the only element of S. As such, since $S = \{w\}$, y is not an intermediary node, which is a contradiction. ∎

LEMMA 5.7. *Given an α-congestion approximator R from Lemma 5.5 for a tree T with n nodes, Algorithm 5.2 is a $O(\log^2 n)$ depth, $\tilde{O}(n)$ work* PRAM *algorithm that constructs a binary tree R' such*

that R' is a α-congestion approximator for T. Moreover, R' has depth $O(\log n)$.

Proof. Let $R' = \texttt{convert-binary}(R)$ be the output of Algorithm 5.2 on R. Lemma 5.6 immediately implies that R' has depth $O(\log n)$, and Line 15 guarantees that it is binary, so it remains to show that R' is a α-congestion approximator for T and that Algorithm 5.2 has $O(\log^2 n)$ depth and $\tilde{O}(n)$ total work. Note that for any edge (u, v) in R, there is a path from u to v in R' with the same minimum capacity as (u, v) in R; namely, all edges on the path from u to v in R' have capacity either ∞ or $c_R((u, v))$. Let P_{uv} be the path for (u, v) in R', and let P_{xy} be the path for some other edge $(x, y) \in E(R)$ in R'. By the setting of edge capacities in Algorithm 5.2, all edges along both P_{uv} and P_{xy}, if there are any, must have capacity ∞. As such, any routing in R' can be converted to a routing in R with the same congestion, and vice-versa, so R' is also a α-congestion approximator for T.

Let $R_0 = R, R_1, \ldots, R_L$ be the sequence of trees such that for all $i < L$, $\texttt{convert-binary}(R_i)$ makes a recursive call $\texttt{convert-binary}(R_{i+1})$ in Line 15, and the call $\texttt{convert-binary}(R_L)$ returns R' without further recursion. Each call to `convert-binary` only increases the number of nodes and depth of the tree: no nodes are removed, and each addition of a node z_i in Line 9 can only increase the depth. So, it follows that for any $i \in [L]$, the depth of R_i is no more than the depth of R'; by Lemma 5.6, we have that the depth of R_i is then $O(\log n)$. Moreover, by construction, each R_i is a hierarchical decomposition of T, so each level j of R_i represents a partitioning of $V(T)$. Thus, as there are $O(\log n)$ levels, each node in T can appear in the sets of at most $O(\log n)$ nodes of R_i, each at a different level of R_i. Since there are n nodes in T, it follows that there are $O(n \log n)$ nodes in R_i.

Each call $\texttt{convert-binary}(R_i)$ before the recursive call in Line 15 can be implemented in $O(\log |R_i|)$ depth and $\tilde{O}(|R_i|)$ work. Since $|R_i| = O(n \log n)$, each call can be implemented in $O(\log n)$ depth and $\tilde{O}(n)$ work. To bound the total depth and work, it thus suffices to bound the total number of recursive calls to `convert-binary` made during a call of $\texttt{convert-binary}(R)$. Define $B(R_i)$ to be the lowest depth (i.e. distance from the root) of any first non-binary node in R_i. In each call $\texttt{convert-binary}(R_i)$, after the `for` loop of Line 4, all first non-binary nodes have at most two children, and are thus no longer first non-binary nodes. In addition, the `for` loop can only reduce the number of children any node has. As such, it follows that $B(R_{i+1}) \geq B(R_i) + 1$ for all $i < L$. Then, as R_L has depth $O(\log n)$, we must have $B(R_i) = O(\log n)$ for all i, and thus $L = O(\log n)$ as well. This results in a complete depth of $O(\log^2 n)$ and total work of $\tilde{O}(n)$. ∎

The proof of Theorem 5.2 then follows from Lemma 5.5 and Lemma 5.7.

6 New Framework for Congestion Approximator Computation.

The goal of this section and the next (Section 7) is to give a full presentation and analysis of our Algorithm 2.2, whose goal is to boost the approximation quality of a given congestion approximator. Specifically, given any congestion approximator of a graph G of arbitrary $\text{polylog}(n)$ distortion, we will compute an $O(\log^9 n)$-congestion approximator of G in $O(m \,\text{polylog}(n))$ work and $O(\text{polylog}(n))$ depth.

Our presentation of Algorithm 2.2 will be given as two parts. First in this section, we show, assuming being able to solve approximate maximum flows on contracted subgraphs, how we can compute a congestion approximator of $O(\log^9 n)$ distortion. Then in the next section (Section 7), we show how we can directly extract congestion approximators for contracted subgraphs from a given congestion approximator of the entire graph, allowing us to then run Sherman's algorithm [66] to compute $(1 - \varepsilon)$-approximate maximum flows on these contracted subgraphs.

We now present in this section a variant of the framework in [61] that allows us to *boost* the approximation of a given congestion approximator in a manner that is both work-efficient and parallelizable. The key novelty in our new framework is to avoid running (approximate) maximum flows on subgraphs of G, which are obtained by removing vertices/edges from G. This is because running maximum flows on such subgraphs requires computing congestion approximators for them using additional recursions, which would have blown up the depth of our algorithm to at least $n^{o(1)}$. Rather, we run approximate maximum flows on *contracted subgraphs* of G, that are obtained from G by contracting subsets of vertices into single nodes. As we will show in Section 7, we can extract congestion approximators

for contracted subgraphs *directly* from a given congestion approximator of the entire graph, *without* having to recurse on the subgraphs.

Throughout this section, we assume that we have a PRAM algorithm $\mathcal{A}$, whose implementation we describe in the next section (Section 7), that can compute a $(1-\varepsilon)$-approximate maximum flow on any given contracted subgraph of G with work near-linear in the number of edges of the subgraph and depth $O(\text{polylog}(n))$. We separate the implementation of $\mathcal{A}$ from the presentation of the new framework here in an effort to make the latter cleaner and more intelligible.

REMARK 6.1. *Note that the specific implementation of algorithm $\mathcal{A}$ will depend on the execution of our framework in this section, and in particular the execution of our framework and that of $\mathcal{A}$ should alternate with each other. This is because the implementation of $\mathcal{A}$ involves extracting congestion approximators for the contracted subgraphs from the given congestion approximator of the entire graph. However, only after we have run $\mathcal{A}$ on a current cluster S do we know the sub-clusters on which we want to run $\mathcal{A}$ subsequently. The composition of the two algorithms is given in Algorithm 2.2.*

We state the performance of our assumed PRAM algorithm $\mathcal{A}$ below, and describe how to implement it in Section 7. Note that in our algorithm, we always set ζ to be at least $1/\text{polylog}(n)$ so $\mathcal{A}$ has near-linear total work and $\text{polylog}(n)$ depth.

PROPOSITION 6.1. (PERFORMANCE OF $\mathcal{A}$) *Let $G'(\mathcal{P}=(S,\omega))$ be the subdivison of a contracted subgraph of G with reweighting function ω such that the range of ω is within $[\zeta,1]$ for some $\zeta\in(0,1]$. Then given an arbitrary graph $G'_{st}(\mathcal{P})$ obtained from $G'(\mathcal{P})$ with maximum s-t flow value F^*, $\mathcal{A}$ computes with high probability*

1. *A feasible s-t flow of value at least $(1-\varepsilon)F^*$ in $G'_{st}(\mathcal{P})$.*
2. *An s-t cut $(T,\bar{T})$ in $G'_{st}(\mathcal{P})$ with capacity at most $(1+\varepsilon)F^*$.*

$\mathcal{A}$ has total work $O(|E(G'_{st}(\mathcal{P}))|\zeta^{-2}\varepsilon^{-3}\,\text{polylog}(n))$ and depth $O(\zeta^{-2}\varepsilon^{-3}\,\text{polylog}(n))$.

Plan for the Rest of the Section. We will obtain a hierarchical decomposition tree of the graph that serves as our congestion approximator. In each step of the decomposition, we use the maximum flow algorithm $\mathcal{A}$ to perform two partitioning steps on a current cluster S, and use the resulting partitions to obtain a two-level decomposition tree. Recursively applying this decomposition step to each sub-cluster obtained ends up giving us a hierarchical decomposition tree of the graph. As will be guaranteed by our partitioning steps, in each decomposition step we reduce the size of the cluster by a constant factor. As a result, we get a tree of $O(\log n)$ depth.

At a high level, the goals of the two partitioning steps are similar to those of [61]. Specifically, the first partitioning step is to find a subset of edges that (i) are well-linked in the sense that we can route product demands between them with low congestion, and (ii) separate the current cluster S into sub-clusters whose sizes are a constant factor smaller. The second partitioning step is then to find a bottleneck cut that separates the inter-cluster edges found in step one and the *boundary edges* that go from S to $V(G)\setminus S$. The difference between our partitioning steps and those of [61] is that we obtain these partitions in contracted subgraphs rather than vertex-induced subgraphs, which avoids additional recursions on these subgraphs when solving maximum flows on them, but on the other hand requires extra care so as to prevent the edges from getting over-congested, since after all an edge could implicitly appear in polynomially many contracted subgraphs as one inside the contracted vertices.

In the rest of this section, we will first present the two partitioning steps needed, and then show how to use them to obtain a hierarchical decomposition tree. In the following, we will interchangeably use the terms *partitioning* and *clustering*, and the terms *partitions* and *clusters*. For a given edge subset F, the partition *induced by* F is the partition of the vertices correponding to the connected components of the graph after the removal of F.

6.1 The First Partitioning Step. Given a contracted subgraph $G(\mathcal{P})$ with at least two uncontracted vertices, our first step is to partition the vertices $V(G(\mathcal{P}))$ into sets Z_1, Z_2 such that the edges between Z_1, Z_2 are well-linked in $G(\mathcal{P})$, and Z_1, Z_2 are size balanced. To this end, we first use the parallel implementation of the cut-matching game in [61] to get a balanced partition with *almost* all of the inter-cluster edges well-linked. To simplify our presentation, we say a partition $Z_1, \ldots, Z_z$ of a set Z is γ-balanced for some $\gamma \in (0, 1]$ if $|Z_i| \leq \gamma |Z|$ for all i.

By plugging $\mathcal{A}$ into the cut matching game of [61] (their Lemma 3.1), whose parallel version is described in Appendix B, we have the following lemma.

LEMMA 6.1. *There exists a* PRAM *algorithm* `partition-A1` *that given $G(\mathcal{P})$ for $\mathcal{P} = (S, \omega)$ with $X = V(G) \setminus S$ and the range of ω within $[\zeta, 1]$, and a set of edges $F \subseteq E$ that induces a 3/4-balanced partition of $V(G(\mathcal{P}))$, with high probability computes a set of new edges F_{new} such that F_{new} also induces a 3/4-balanced partition of $V(G(\mathcal{P}))$, and*

1. *either $c^{G(\mathcal{P})}(F_{\text{new}}) \leq \frac{7}{8} c^{G(\mathcal{P})}(F)$;*
2. *or $F_{\text{new}} = A \cup R$ with A, R disjoint, such that $c^{G(\mathcal{P})}(A) \leq c^{G(\mathcal{P})}(F)$, $c^{G(\mathcal{P})}(R) \leq \frac{2}{\log n} \cdot c^{G(\mathcal{P})}(A)$, and edges in A are β-well-linked in $G(\mathcal{P})$ for $\beta = \Omega(1/\log^2 n)$.*

The algorithm `partition-A1` *has $O(|E(G(\mathcal{P}))|\zeta^{-2} \operatorname{polylog}(n))$ total work and $O(\zeta^{-2} \operatorname{polylog}(n))$ depth.*

We also need the following lemma, whose proof is deferred to Section 6.2.

LEMMA 6.2. *There is a* PRAM *algorithm* `partition-A2` *that given $G(\mathcal{P})$ for $\mathcal{P} = (S, \omega)$ with $X = V(G) \setminus S$ and the range of ω within $[\zeta, 1]$, a set of edges $F = A \cup B$ inducing a 3/4-balanced partition of $V(G(\mathcal{P}))$ where A, B are disjoint with $c^{G(\mathcal{P})}(B) \leq 2c^{G(\mathcal{P})}(A)/\log n$,*

1. *either finds an edge set F_{new} with $c^{G(\mathcal{P})}(F_{\text{new}}) \leq \frac{3}{4} c^{G(\mathcal{P})}(F)$ that induces a 3/4-balanced partition;*
2. *or finds an edge set C disjoint from A such that $F_{\text{new}} := A \cup C$ induces a 3/4-balanced partition, and there exists a multicommodity flow in $G'(\mathcal{P})$ with congestion $O(\log n)$ from X_C to X_A such that (i) each $x_e \in X_C$ sends $c_e^{G(\mathcal{P})}$ units of flow, and (ii) each $x_f \in X_A$ receives $O(\log n) \cdot c_f^{G(\mathcal{P})}$ units of flow.*

The algorithm `partition-A2` *has $O(|E(G(\mathcal{P}))|\zeta^{-2} \operatorname{polylog}(n))$ total work and $O(\zeta^{-2} \operatorname{polylog}(n))$ depth.*

We now prove the main lemma about our first partitioning step.

LEMMA 6.3. *There is a* PRAM *algorithm* `partition-A` *that given $G(\mathcal{P})$ for $\mathcal{P} = (S, \omega)$ with $X = V(G) \setminus S$ and the range of ω within $[\zeta, 1]$, with high probability partitions the uncontracted vertices into Z_1, Z_2 such that*

1. *$Z_1 \cup Z_2 = V(G(\mathcal{P})) \setminus \{u_X\}$.*
2. *$|Z_i| \leq \frac{7}{8} |V(G(\mathcal{P}))|$ for each $i = 1, 2$.*
3. *The set of inter-cluster edges $F := \{(u, v) | u \in Z_1, v \in Z_2\}$ is β-well-linked in $G(\mathcal{P})$ for $\beta = \Omega(1/\log^3 n)$.*

The algorithm `partition-A` *has $O(|E(G(\mathcal{P}))|\zeta^{-2} \operatorname{polylog}(n))$ total work and $O(\zeta^{-2} \operatorname{polylog}(n))$ depth.*

Proof. By composing Lemmas 6.1 and 6.2 the same way as [61] compose their Lemmas 3.1 and 3.2, we get in desired total work and depth a partition $Z'_1, \ldots, Z'_z$ of $V(G(P))$ such that

1. $Z'_1 \cup \ldots \cup Z'_z = V(G(\mathcal{P}))$.
2. $|Z'_i| \leq \frac{3}{4} |V(G(\mathcal{P}))|$ for each $i \in [z]$.

3. The set of inter-cluster edges $F' = \{(u,v) | u \in Z'_i, v \in Z'_j, i \neq j\}$ is β'-well-linked in $G(\mathcal{P})$ for $\beta' = \Omega(1/\log^3 n)$.

We can then obtain Z_1, Z_2 by the following process. First, we remove the supernode u_X from $Z'_1, \ldots, Z'_z$ to obtain a partition $Z''_1, \ldots, Z''_z$ of the uncontracted vertices. Now the inter-cluster edges between $Z''_1, \ldots, Z''_z$ are still $\Omega(1/\log^3 n)$-well-linked in $G(\mathcal{P})$ since they are a subset of the inter-cluster edges between $Z'_1, \ldots, Z'_z$. Then to get a bi-partition, we merge the subsets $Z''_1, \ldots, Z''_z$ as follows. Let i be the largest integer s.t. $|Z''_1 \cup Z''_2 \cup \ldots \cup Z''_i| \leq \frac{7}{8} |V(G(\mathcal{P})|$. Then we merge the $Z''_1, \ldots Z''_i$ into one partition Z_1 and the remaining Z''_i's into another partition Z_2 to obtain our bi-partition Z_1, Z_2. Then because $Z''_1, \ldots, Z''_z$ is a partition of the uncontracted vertices, so is Z_1, Z_2. Since each Z''_i has size at most $\frac{3}{4} |V(G(\mathcal{P}))|$, by the definition of i, Z_1 has size between $[\frac{1}{8} |V(G(\mathcal{P}))|, \frac{7}{8} |V(G(\mathcal{P}))|]$. Therefore, Z_2 has size at most $\frac{7}{8} |V(G(\mathcal{P}))|$ since Z_1, Z_2 is a partition of the uncontracted vertices in $V(G(\mathcal{P}))$. Finally, inter-cluster edges (call them F) from Z_1 to Z_2 are still $\Omega(1/\log^3 n)$-well-linked in $G(\mathcal{P})$ since they are a subset of the inter-cluster edges between $Z''_1, \ldots, Z''_z$.

Note that the partitioning into Z_1, Z_2 can be done in parallel in $O(\log n)$ depth by computing a prefix sum and a binary search. The total work and depth then follows from the performance of the first partitioning step of [61]. ∎

6.2 The Second Partitioning Step. Let B denote the *boundary edges* in $G(\mathcal{P})$, namely those that go from the contracted vertex u_X to the uncontracted vertices. In other words, $B := E(u_X, V(G) \setminus X)$.

Our second step is to find a cut in $G(\mathcal{P})$ separating the boundary edges B from the inter-cluster edges F that we identified in the first partitioning step. Here, we want the property that there is a low-congestion routing from the cut edges we find to the boundary edges B, as well as from the cut edges to the inter-cluster edges F, such that each cut edge sends out flow equal to its capacity. Specifically, we prove the following lemma. Recall that for a graph H and an edge $e \in H$, we write c_e^H to denote the capacity of edge e in H.

LEMMA 6.4. *There is a* PRAM *algorithm* `partition-B` *that given $G(\mathcal{P})$ for $\mathcal{P} = (S, \omega)$ with $X = V(G) \setminus S$ and the range of ω being within $[\zeta, 1]$ with $\zeta \in (0, 1]$, two disjoint edge subsets $B, F \subset E(G(\mathcal{P}))$, and a parameter $\psi \in (0, 1]$, with high probability returns a subset of edges Y (potentially intersecting both B and F) in $G(\mathcal{P})$ such that*

1. *X_Y separates X_B from X_F in $G'(\mathcal{P})$, that is, in $G'(\mathcal{P})$ every path between a vertex in X_B and a vertex in X_F must contain a vertex in X_Y. Moreover, in $G(\mathcal{P})$, the total capacity of Y is at most twice the capacity of B and at most twice the capacity of F.*
2. *There is a multicommodity flow in $G'(\mathcal{P})$ from X_Y to X_B such that*
   (a) *The congestion on edges incident on u_X in $G'(\mathcal{P})$ is $O(\psi \log n)$, while the congestion on other edges in $G'(\mathcal{P})$ is $O(\log n)$.*
   (b) *Each $x_y \in X_Y$ sends $c_y^{G(\mathcal{P})}$ units of flow while each $x_b \in X_B$ receives at most $O(\log n) \cdot c_b^{G(\mathcal{P})}$ units of flow.*
3. *Similarly, there is a multicommodity flow in $G'(\mathcal{P})$ from X_Y to X_F such that*
   (a) *The congestion on edges incident on u_X in $G'(\mathcal{P})$ is $O(\psi \log n)$, while the congestion on other edges in $G'(\mathcal{P})$ is $O(\log n)$.*
   (b) *Each $x_y \in X_Y$ sends $c_y^{G(\mathcal{P})}$ units of flow, while each $x_f \in X_F$ receives at most $O(\log n) \cdot c_f^{G(\mathcal{P})}$ units of flow.*

The algorithm `partition-B` *has $O(|E(G(\mathcal{P}))| \zeta^{-2} \psi^{-2} \operatorname{polylog}(n))$ total work and $O(\zeta^{-2} \psi^{-2} \operatorname{polylog}(n))$ depth.*

Proof. We will do an iterative refinement process to find the desired set of cut edges Y. Initially, we start with Y being the smaller (in capacity) of B and F, which has the desired property that X_Y separates X_B from X_F in $G'(\mathcal{P})$. We will maintain this property, while "refine" the set Y. In particular, we will classify the edges in Y into *good* edges Y_{good} and *bad* edges Y_{bad}, and try to reduce the total capacity of the bad edges Y_{bad}. Initially, $Y_{\text{good}} = \emptyset$ and $Y_{\text{bad}} = Y$.

In each iteration, we do the following refinement step. Define a reweighting function $\tilde{\omega}$ on the split edges (in $G'(\mathcal{P})$) of the boundary edges of $G(\mathcal{P})$ such that

$$\tilde{\omega}(f) = \begin{cases} \psi & f \text{ incident on } u_X \\ 1 & \text{otherwise.} \end{cases}$$

and look at the graph $G'(\tilde{\mathcal{P}} = (\mathcal{P}, \tilde{\omega}))$. Let $\varepsilon = \log^{-1} n$.

Let $T_1 := B$ and $T_2 := F$. For $j \in \{1, 2\}$, we consider the graph $G'_{s_j t_j}(\tilde{\mathcal{P}})$ where we connect s_j to each $x_y \in X_{Y_{\text{bad}}}$ with capacity $c_y^{G(\mathcal{P})}$ and connect each $x_e \in X_{T_j}$ to t_j with capacity $c_e^{G(\mathcal{P})}$. We run $\mathcal{A}$ (as defined in Proposition 6.1) on $G'_{s_j t_j}(\tilde{\mathcal{P}})$ to find a $(1-\varepsilon)$-approximate maximum s_j-t_j flow f'_j and a $(1+\varepsilon)$-approximate s_j-t_j minimum cut with edges Y'_j.

We say an edge $y \in Y_{\text{bad}}$ *has become good*, if for both $j = 1, 2$, the edge (s_j, x_y) in $G'_{s_j t_j}(\tilde{\mathcal{P}})$ carries (in the direction from s_j to x_y) at least $c_y^{G(\mathcal{P})}/4$ units of flow in f'_j.

We then do a case analysis as follows:

Case 1 If for both $j = 1, 2$, we have

$$c^{G'_{s_j t_j}(\tilde{\mathcal{P}})}(Y'_j) \geq \frac{9}{10} c^{G(\mathcal{P})}(Y_{\text{bad}}),$$

then move the edges in Y_{bad} that have become good to Y_{good}.

Case 2 Else, let $j^* \in \{1, 2\}$ be such that

$$c^{G'_{s_{j^*} t_{j^*}}(\tilde{\mathcal{P}})}(Y'_{j^*}) < \frac{9}{10} c^{G(\mathcal{P})}(Y_{\text{bad}}).$$

Then let $Y_{\text{bad}} \leftarrow Y_{j^*}$, where Y_{j^*} is constructed as follows:

(a) Let Y''_{j^*} be Y'_{j^*} with edges incident on u_X removed.
(b) Include any edge $e \in E(G(\mathcal{P}))$ in Y_{j^*} if x_e is incident on at least one edge in Y''_{j^*}.

CLAIM 6.1. *In **Case 1**, the edges $y \in Y_{\text{bad}}$'s that have become good contribute at least $1/5$ of the total capacity of Y_{bad}.*

Proof. For $j \in \{1, 2\}$, consider the flow f'_j in $G'_{s_j t_j}(\tilde{\mathcal{P}})$. The flow value is at least $\frac{4}{5} c^{G(\mathcal{P})}(Y_{\text{bad}})$ given the conditions of **Case 1** and that f'_j is a $(1-\varepsilon)$-approximate maximum flow. Therefore at least $3/5$ (in capacity) of $y \in Y_{\text{bad}}$'s (call them $Y^{(j)}_{\text{good}}$) satisfies that the edge (s_j, x_y) in $G'_{s_j t_j}(\tilde{\mathcal{P}})$ carries (in the direction from s_j to x_y) at least $c_y^{G(\tilde{\mathcal{P}})}/4$ units of flow in f'_j, since otherwise the total flow value would be

$$\left(\frac{3}{5} + \frac{2}{5} \cdot \frac{1}{4}\right) c^{G(\mathcal{P})}(Y_{\text{bad}}) < \frac{4}{5} c^{G(\mathcal{P})}(Y_{\text{bad}}),$$

a contradiction. The claim then follows by taking the intersection of $Y^{(1)}_{\text{good}}$ and $Y^{(2)}_{\text{good}}$. ∎

CLAIM 6.2. *In **Case 2**, we have*

$$c^{G(\mathcal{P})}(Y_{j^*}) \leq c^{G'_{s_{j^*} t_{j^*}}(\tilde{\mathcal{P}})}(Y'_{j^*}).$$

Moreover, $X_{Y_{j^*} \cup Y_{\text{good}}}$ separates X_B from X_F.

Proof. For the capacity condition, notice that we can simply charge the capacity of every edge in $e \in Y_{j^*}$ to the capacity of an edge in Y''_{j^*} that is incident on x_e. Let Y'_{good} denote the set of all split edges of edges in Y_{good} in $G'(\mathcal{P})$. Define edge set Z_{j^*} by including edges $e \in E(G(\mathcal{P}))$ for which x_e is incident on at least on edge in Y'_{j^*}. Since Y'_{j^*} is an s_j-t_j cut in $G'_{s_j t_j}(\tilde{\mathcal{P}})$, $X_{Z_{j^*}}$ separates $X_{Y_{\text{bad}}}$ from $X_{T_{j^*}}$. Since $X_{Y_{\text{bad}}} \cup X_{Y_{\text{good}}}$ separates X_B and X_F, $X_{Z_{j^*}} \cup X_{Y_{\text{good}}}$ also separates X_B and X_F. Now notice that the only difference between Y_{j^*} and Z_{j^*} is that the former does not have the edges e for which the only incident edge of e in Y'_{j^*} is (u_X, x_e). However, these edges are only useful for moving between X_B through u_X, and thus does not affect whether or not X_B is separated from X_F. Hence $X_{Y_{j^*} \cup Y_{\text{good}}}$ also separates X_B from X_F. ∎

Therefore, if we repeat the above refinement process, each time we shrink the capacity of Y_{bad} by a constant factor. So after $O(\log n)$ iterations, we are done finding a desired set of cut edges Y that can route to both B and F with the desired congestion. Moreover, in each iteration, we can determine in parallel which edges in Y_{bad} have become good by simply examining our flow solution. The lemma thus follows. ∎

We now prove Lemma 6.2.

Proof. (Lemma 6.2) The proof is essentially the same as Lemma 3.2 of [61], with their second partitioning step replaced by ours. Using Lemma 6.4 with $\psi = 1$, we can find an edge set C such that X_C separates X_A and X_B in $G'(\mathcal{P})$ and there exists a desired multicommodity flow routing from X_C to X_A as guaranteed by Lemma 6.4. Moreover, the total capacity of C is at most twice of the total capacity of B. Using the fact that X_C separates X_A and X_B in $G'(\mathcal{P})$, and that $A \cup B$ induces a 3/4-balanced partition, due to Claim 1 in the Proof of Lemma 3.2 in [61], either $A \cup C$ or $B \cup C$ induces a 3/4-balanced partition. If it is $A \cup C$, then we have achieved the second case of the lemma. If it is $B \cup C$, we have achieved the first case of the lemma. ∎

6.3 Recursive Construction of Congestion Approximators. We now show how to recursively construct a congestion approximator for G using our two partitioning steps above. Note that imperatively, we use a lower reweighting factor for the boundary edges in `partition-A` than in `partition-B` ($\log^{-12} n$ vs. $\log^{-4} n$) to avoid a blowup in the congestion when doing the routing "fixing" phase, which is needed because we route on contracted subgraphs rather than vertex-induced subgraphs.

ALGORITHM 6.1. `hierarchical-decomp`$(G(\mathcal{P}))$
Input: Reweighted contracted subgraph $G(\mathcal{P})$, where $\mathcal{P} = (S, 1)$.
Output: A hierarchical tree decomposition tree $\boldsymbol{T}$ of G.
Procedure:

1: **if** $G(\mathcal{P})$ only has one uncontracted vertex, plus the supernode u_X **then**
2: Return the single uncontracted vertex as our congestion approximator and abort.
3: Run `partition-A` on $G(\mathcal{P}_A := (S, \log^{-12} n))$ to get a partition Z_1, Z_2 of the uncontracted vertices $V(G(\mathcal{P})) \setminus \{u_X\}$, with F being the inter-cluster edges between Z_1, Z_2.
4: Run `partition-B` on $G(\mathcal{P}_B := (S, \log^{-4} n))$ with $\psi = \log^{-6} n$ to obtain a set of cut edges Y separating boundary edges B from inter-cluster edges F.
5: // Recall that B is the set of edges incident to u_X in $G(\mathcal{P}_B)$.
6: Let L_1 denote the vertices in $V(G(\mathcal{P}))$ that cannot reach any edge in B after the removal of Y, and let L_2 denote the other vertices.
7: // L_2 are the vertices that can reach B after the removal of Y
8: Let $\mathcal{Z} := \{\mathcal{Z}_1 := Z_1 \cap L_1, \mathcal{Z}_2 := Z_2 \cap L_1, \mathcal{Z}_3 := Z_1 \cap L_2, \mathcal{Z}_4 := Z_2 \cap L_2\}$ be the partition obtained by taking the intersection of the partitions returned by `partition-A` and `partition-B`.
9: **for** $\mathcal{Z}_i \in \mathcal{Z}$ **do**
10: Recursively run `hierarchical-decomp` on $G(\mathcal{P}_i)$ to get a tree $\mathcal{T}_i$, where $\mathcal{P}_i = (\{S_i := \mathcal{Z}_i\}, 1)$

11: Create a new tree node r as the root of our tree, and two other nodes v_{L_1}, v_{L_2} as r's children corresponding to vertex subsets v_{L_1}, v_{L_2} respectively. We also create nodes v_1, v_2, v_3, v_4 corresponding to $\mathcal{Z}_1, \mathcal{Z}_2, \mathcal{Z}_3, \mathcal{Z}_4$ respectively, and let v_1, v_2 be the children of L_1 and let v_3, v_4 be the children of L_2.
12: For each v_i, let v_i be the root of the tree $\mathcal{T}_i$ that we computed using recursive calls to `hierarchical-decomp`. In other words, the children of L_1 are the roots of $\mathcal{T}_1$ and $\mathcal{T}_2$, and the children of L_2 are the roots of $\mathcal{T}_3$ and $\mathcal{T}_4$.
13: For each tree node corresponding to a subset of vertices $S \subset V(G)$, weight the tree edge that connects this tree node to its parent by the total capacity of the cut $(S, V(G) \setminus S)$ in G.
14: **return** The tree rooted at r constructed above.

Similar to [61], our congestion approximator will also be a hierarchical decomposition tree of the graph. For a tree node corresponding to a subset of vertices $S \subset V$, the tree edge that connects this tree node to its parent will have capacity equal to the total capacity of edges leaving S in G (i.e. the capacity of the cut $(S, V(G) \setminus S)$ in G).

We give the pseudocode of our construction of a congestion approximator of G below. As we highlighted at the beginning of this section, this algorithm implements our Algorithm 2.2 modulo being able to solve approximate maximum flows on contracted subgraphs, which we will show how to do in the next section (Section 7). During the execution of our algorithm, we do not reweight the graph that we recurse on, but only do reweighting when running the two partitioning steps. We slightly abuse notations by writing a number γ to denote a constant reweighting function that evaluates to γ on every boundary edge. Initially, we call the algorithm on the entire graph with $X = \emptyset$.

THEOREM 6.1. *The algorithm* `hierarchical-decomp`(G) *has* $O(m \operatorname{polylog}(n))$ *total work and* $O(\operatorname{polylog}(n))$ *depth. Moreover, the output tree* $\mathcal{T} =$ `hierarchical-decomp`(G) *is an* $O(\log^9 n)$*-congestion approximator of* G *with high probability.*

Proof. The total work and depth follows easily from the performance of the two partitioning steps above. We thus focus on proving that the returned tree $\mathcal{T}$ is an $O(\log^9 n)$-congestion approximator. Specifically, we show that

1. Any multicommodity flow demands that can be routed in G with congestion 1 can also be routed on the tree with congestion at most 1.

2. Any multicommodity flow demands that can be routed on the tree with congestion 1 can also be routed in G with congestion $O(\log^9 n)$.

Here 1 is clear, since the tree cut induced by each tree edge corresponds to a cut in the original graph with exactly the same capacity, and thus the set of these tree cuts are a subset of the cuts in the original graph. We then prove 2. Consider that there is a demand D_{st} between each vertex pair $s, t \in V(G)$ such that these demands D_{st}'s can be routed simultaneously on the tree with congestion at most 1. We then describe a routing scheme to show that they can also be simultaneously routed in G with congestion $O(\log^9 n)$. Our routing scheme will be invoking an essentially same routing routine in [61], plus an additional routing *fixing* phase, which is necessary due to the fact that we work in contracted subgraphs rather than vertex-induced subgraphs.

To illustrate a routing between s and t of flow value D_{st}, we let each of s, t send out a message of size D_{st}. We will thus use the terms *flow* and *message*, as well as *message passing* and *flow routing*, interchangeably. We will move these messages up along the hierarchical decomposition tree by repeatedly moving the messages from the boundary edges of the current cluster to the boundary edges of the parent cluster. If at some point, to some edge, we have routed the same amount of flow from s and t, then we can construct a routing from s to t of this flow amount by reversing the message trail of t; whenever this happens in our routing process, we discard these paired-up messages and never consider them in subsequent routing steps. Crucially, we move these messages for different s, t pairs *simultaneously*.

We will specify this message passing process inductively. Note that in each recursive call of the tree construction, we always do two steps of partitioning and obtain a two-level tree before recursing on the leaves of the two-level tree corresponding to smaller clusters. Let us call the two levels in each single recursive call a *level block*. We will route the messages level block by level block. For each level block with root being $S = V(G) \setminus X$, we specify a routing in the graph $G'(\mathcal{P} = (S, 1))$, the subdivision of the contracted subgraph. When we say we route some flow to an edge in $G(\mathcal{P})$ (the contracted subgraph without subdivision), we mean we route the flow to the split vertex of that edge. We will maintain the following invariant:

> **Invariant:** *After we are done with the routing for a level block with root being $S = V(G) \setminus X$, all messages originating from vertices in S either have been discarded, or reside on the boundary edges (i.e. the ones that go between S and u_X).* $(\star)$

We will also do the routing "fixing" inductively. Specifically, in each inductive step, even before we specify the routing, we first fix the routing for each smaller cluster $\mathcal{Z}_i$ that we specified in the smaller contracted subgraph $G'(\mathcal{P}_i = (\mathcal{Z}_i, 1))$, so that they become routings in the bigger contracted subgraph $G'(\mathcal{P} = (S, 1))$. Only after the fixing is done for each $\mathcal{Z}_i$ do we proceed with the routing of the messages in $G'(\mathcal{P})$.

As a result, our inductive step consists of two phases, namely a *fixing phase*, and a *routing phase*, where the routing phase will be essentially the same as in [61]. We now fully describe our inductive routing scheme below, as well as analyze the congestion caused. Notice that, throughout the proof, we always analyze the congestion of edges with respect to their *original* capacities in G (without considering the reweighting factors that we use in contracted subgraphs). We will highlight all the congestion we calculate along the way of presenting our routing scheme. Since we route in subdivision graphs, we will need to distinguish, for a boundary edge e, the congestion on its split edge incident on u_X, and the congestion on its other split edge incident on S. We will call the former outer congestion, and the latter inner congestion.

Note that for our goal of proving the obtained tree is a congestion approximator, it suffices to prove the existence of such a low-congestion routing. We will then plug in the tree into Sherman's algorithm to compute approximate maximum flows.

Base Case. Initially, as the base case of the inductive routing process, for each demand D_{st}, s and t will each distribute the total amount of flow D_{st} uniformly to their outgoing edges - a.k.a. the boundary edges of these singleton clusters - with each edge getting flow proportional to its capacity. Thus we have established the invariant $(\star)$.

This routing causes congestion at most 2 on the edges of G, since a node cannot send or receive a total flow amount greater than its weighted degree given that the demands are routable with congestion 1, and an edge is incident on two vertices.

Inductive Step: Fixing Phase. For $i = 1, 2, 3, 4$, given a routing in $G'(\mathcal{P}_i)$, we aim to obtain another routing in the bigger contracted subgraph $G'(\mathcal{P})$ that route the same demands. Recall that S denotes the set of nodes in consideration, i.e., $S = V \setminus X = \mathcal{Z}_1 \cup \mathcal{Z}_2 \cup \mathcal{Z}_3 \cup \mathcal{Z}_4$. Suppose first we take the exact same given routing in $G'(\mathcal{P}_i)$ and simply expand (namely, undo the contractions of) the vertices in $S \setminus \mathcal{Z}_i$, obtaining a routing in $G'(\mathcal{P})$. Due to the expanding we have done, we may have created extra deficits and excesses on the vertices in $(S \setminus \mathcal{Z}_i) \cup \{u_X\}$, which we now have to fix. To this end, it suffices to give for each demand a routing with the deficits/excesses on $(S \setminus \mathcal{Z}_i) \cup \{u_X\}$ switched. These routings should be simultaneously achievable with low congestion.

Let $\alpha_i \geq 0$ be the maximum outer congestion on the boundary edges of $G(\mathcal{P}_i = (\mathcal{Z}_i, 1))$, and let $\alpha := \max\{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$. Let $\gamma_i \geq 0$ be the maximum inner congestion on the boundary edges of $G(\mathcal{P}_i = (\mathcal{Z}_i, 1))$, and let $\gamma := \max\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$.

Claim 6.3. *We can fix the routings in $G(\mathcal{P}_i)$'s so that the become routings in $G(\mathcal{P})$ with the same demand while causing outer congestion of boundary edges $O(\alpha \log^{-3} n)$, inner congestion of boundary edges $O(\alpha \log n)$, and congestion on edges inside S $O(\alpha \log^8 n)$.*

Proof. For $\mathcal{Z}_3, \mathcal{Z}_4$, we fix the extra deficits/excesses by a routing from $Y \setminus B$ to B, followed by a mixing to uniform (with each edge getting flow proportional to its capacity) over B. The routing causes outer congestion of boundary edges $O(\alpha \log^{-9} n)$, inner congestion of boundary edges $O(\alpha \log^{-3} n)$, and congestion of edges inside S $O(\alpha \log n)$. The mixing causes congestion $O(\alpha \log^{-3} n)$. The last congestion is because each edge in B receives $O(\alpha \log^{-3} n)$ times its capacity units of flow,

For $\mathcal{Z}_1, \mathcal{Z}_2$, we fix the extra deficits/excesses by a routing from Y to F, followed by a mixing to uniform (with each edge getting flow proportional to its capacity) over F. The routing causes outer congestion of boundary edges $O(\alpha \log^{-5} n)$, inner congestion of boundary edges $O(\alpha \log n)$, and congestion of edges inside S $O(\alpha \log^5 n)$. The mixing causes congestion on boundary edges $O(\alpha \log^{-4} n)$, and congestion on edges inside S $O(\alpha \log^8 n)$. ∎

Inductive Step: Routing Phase. This phase is essentially the same as in [61], except that we do the routing in the contracted subgraph $G'(\mathcal{P})$ rather than in the vertex induced subgraph, and therefore we also have to analyze the congestion on the boundary edges B.

For the routing phase, we divide the messages originating in S that haven't yet been discarded (thus still need to be rounted) into following types:

Type 1 The messages corresponding to demands D_{st}'s for which both s, t lie in $L = \mathcal{Z}_1 \cup \mathcal{Z}_2$ (vertices not reachable from B after removing Y).

Type 2 The messages corresponding to demands D_{st}'s for which one of s, t lies in L and the other lies outside of L.

Type 3 The messages corresponding to demands D_{st}'s for which both s, t lie outside L (but at least one of s, t lies in $R = S \setminus L$, as otherwise they are not considered in this level block).

Recall that, by the invariant ($\star$) that we maintain, all three types of messages reside on the boundary edges of the smaller clusters $\mathcal{Z}_i$'s. Note that, by the construction of the $\mathcal{Z}$'s, the boundary edges of clusters $\mathcal{Z}_1, \mathcal{Z}_2$ are subsets of $F \cup Y$, while the boundary edges of $\mathcal{Z}_3, \mathcal{Z}_4$ are subsets of $Y \cup B$. The goal of our inductive routing step is as follows:

Goal 1 For **Type 1** messages, we show that we can simultaneously pair up the messages from s and the ones from t.

Goal 2 For **Type 2** and **Type 3** messages, we show how to route them to the boundary edges B of the bigger cluster S.

Here **Goal 1** means that we can discard all **Type 1** messages afterwards, since by doing so we have already found a simultaneous routing of the corresponding demands D_{st}'s; whereas **Goal 2** means that after our routing, **Type 2** and **Type 3** messages always reside on the boundary edges of S. These together imply that we have maintained our invariant ($\star$).

We now describe how we achieve these goals, as well as analyze the congestion caused. Throughout, let $\beta_i \geq 0$ be the maximum ratio of the amount of flow received to the (original) edge capacity over the boundary edges of $\mathcal{Z}_i$, when we finish routing in $G(\mathcal{P}_i)$. Then let $\beta = \max\{\beta_1, \beta_2, \beta_3, \beta_4\}$.

Inductive Step: Routing Type 1 Messages. By our invariant ($\star$), **Type 1** messages reside on the boundary edges of $\mathcal{Z}_1 \cup \mathcal{Z}_2$, which are $Y \cup F$. We first route **Type 1** messages that reside on $Y \setminus F$ to F. By the guarantee of `partition-B`, this can be done with outer congestion of boundary edges $O(\beta \log^{-5} n)$, inner congestion of boundary edges $O(\beta \log n)$, and congestion of edges inside S $O(\beta \log^5 n)$.

After the routing, all **Type 1** messages reside on F, and the flow that each edge in F carries is at most $2\beta + O(\beta \log^5 n) = O(\beta \log^5 n)$ of its capacity. We then simultaneously mix each of **Type 1** messages uniformly over F, in the sense that each edge in F gets an amount proportional to its capacity. After the mixing, we have successfully paired up all **Type 1** messages and thus can discard them all.

This mixing causes congestion on boundary edges $O(\beta \log^{-4} n)$, and congestion on edges inside S $O(\beta \log^{8} n)$. Combined this with the congestion we get in the $Y \setminus F \to F$ routing, the added outer and inner congestion of B in $G(\mathcal{P})$ is at most $O(\beta \log^{-4} n)$ and $O(\beta \log n)$, whereas the added congestion on edges inside S in $G(\mathcal{P})$ is at most $O(\beta \log^{8} n)$.

Inductive Step: Routing Type 2, Type 3 Messages. By our invariant ($\star$), both **Type 2** and **Type 3** messages reside on the edges in $Y \cup B \cup F$. We first route the messages that reside on $F \setminus Y$ to Y. This routing consists of first mixing the message uniformly on F and then reversing the routing from Y to a uniform distribution over F, whose existence is guaranteed by `partition-B`. After the routing, edges in $F \setminus Y$ carry no flow on them, and each edge in Y carries flow that is at most $2\beta+1$ of its capacity in $G(\mathcal{P})$.

By a similar analysis as in the routing of **Type 1** messages, the routing can be done with outer congestion of boundary edges $O(\beta \log^{-5} n)$, inner congestion of boundary edges $O(\beta \log n)$, and congestion of edges inside S $O(\beta \log^{5} n)$.

We then route message on $Y \setminus B$ to B. The routing causes outer congestion of boundary edges $O(\beta \log^{-9} n)$, inner congestion of boundary edges $O(\beta \log^{-3} n)$, and congestion of edges inside S $O(\beta \log n)$. After the routing, each edge in B carries flow that is at most $\beta + (2\beta + 1)O(\log^{-3} n) = (1 + O(1/\log^{3} n))\beta$ of its capacity in $G(\mathcal{P})$.

Total Congestion Analysis. We first obtain an upper bound on β, the total amount of flow carried by a boundary edge divided by its capacity in G. Initially, we have $\beta = O(1)$ in our base case. Then in each inductive step, β grows by $1 + O(1/\log^{3} n)$, as we have analyzed above when routing **Type 2**, **Type 3** messages, which is the only place where we route flows to B. Since the depth of the tree is $O(\log n)$, we have $\beta = O(1)$ throughout.

We next obtain an upper bound on α, the total outer congestion of any boundary edge, and γ, the total inner congestion of any boundary edge, In the fixing phase, the total outer congestion and inner congestion added on a boundary edge is $O(\alpha_i/\log^{3} n)$ and $O(\alpha \log n)$ respectively. In the routing phase, the total outer and inner congestion added on a boundary edge is $O(\beta \log^{-4} n)$ and $O(\beta \log n)$. Therefore, by a simple induction and the fact that the tree has $O(\log n)$ depth, the total outer and inner congestion accumulated over descendant routing steps on a boundary edge is bounded by $O(1)$ and $O(\log^{2} n)$, respectively.

Finally, in the routing phase and the fixing phase, the total congestion added on the edges inside S is bounded by $O(\beta \log^{8} n) + O(\alpha \log^{8} n) = O(\log^{8} n)$. Since each edge appears $O(\log n)$ times as inside S, the total congestion accumulated is $O(\log^{9} n)$. This finishes the proof of the theorem. ∎

7 Extraction of Congestion Approximators.

We finally describe the implementation details of the approximate maximum flow oracle $\mathcal{A}$ in Section 6 whose performance guarantees are summarized in Proposition 6.1. We would like to remind the reader that this oracle, at its heart, utilizes Sherman's framework to compute approximate maximum flows which requires as input polylog n-congestion approximators for the flow instance upon which it is invoked. While we defer the PRAM implementation details of Sherman's algorithm to Appendix C, we discuss in this section, three important routines for constructing congestion approximators required for the contracted subgraphs generated by our new framework for constructing high-quality congestion approximators. These are (i) (partially) compressing the global congestion approximator to obtain one for the contracted subgraphs, (ii) obtaining a near-linear work, low-depth implementation of Sherman's algorithm when given as input these (partially) compressed congestion approximators, and (iii) constructing congestion approximators for contracted subgraphs with an arbitrarily attached super-source and super-sink. We would crucially like to remind the reader that at all points in our overall approach, we build and maintain congestion approximators for *subdivision graphs* and *contracted subdivision graphs*, as these are precisely the flow instances upon which the max-flow oracle is invoked.

7.1 Compressing Congestion Approximators.

We start by describing the `ca-contraction` subroutine of Algorithm 2.2 that given a contracted subgraph and a congestion approximator for a much

larger graph, compresses it to have size proportional to that of the contracted subgraph. Specifically, we are given as input a $O(\text{polylog}\, n)$-congestion approximator R for some larger graph $G = (V, E, c)$ along with a subset $S \subset V$ of k vertices that have the following special property: all edges that leave the set S are well-linked in G. We prove such a property for the graphs we encounter:

CLAIM 7.1. *In any $G(\mathcal{P} = (S, 1))$ with $X = V(G) \setminus S$ and S being a cluster corresponding to a leaf of a level block of the tree constructed by* `hierarchical-decomp`*, the boundary edges incident on u_X are $\Omega(1/\log^9 n)$-well-linked in G.*

Proof. Note that in $G(\mathcal{P})$, these edges are 1-well-linked. We then consider the multicommodity flow routing between them in $G(\mathcal{P})$. We now fix this routing using our fixing step with performance guaranteed by Claim 6.3 in a bottom-up manner until we have converted the routing into a valid one in G with the same demands. Then by Claim 6.3 the total congestion is $O(\log^9 n)$, implying the claim. ∎

Our goal is to output a $O(\text{polylog}\, n)$-congestion approximator of size $O(k \log n)$ for the contracted subgraph $G(S)$ where $V \setminus S$ is compressed into a single node[8], and the aforementioned well-linkedness property will be crucial in achieving it. This compression step is critical in order to achieve near-linear work; while the original congestion approximator R of G is also a $O(\text{polylog}\, n)$-congestion approximator for the contracted graph $G(S)$, its size may be very large relative to that of $G(S)$, and naively using it for computing approximate max-flows in $G(S)$ would substantially blow up the total work. This subroutine precisely addresses this issue by reducing the size of the congestion approximator without degrading its quality substantially.

If we were simply to contract into a *single node* all the nodes in R which represent only subsets of $V \setminus S$, the resulting graph would not be a tree, and so we would not be able to use it for computing max-flows with Sherman's algorithm. As such, we must use an alternative approach to shrink the size of R that preserves the tree structure. We accomplish this with the following algorithm, which uses all three properties guaranteed by the transformation in Section 5.2: R is a hierarchical decomposition and also a binary tree of depth $O(\log n)$. As R is a hierarchical decomposition, for all $v \in S$, there is a leaf of R corresponding to the set containing only v. Assign a node weight of 1 to these leaves, and assign a weight of 0 to all other nodes in R. Then, run the subtree sum algorithm of Theorem A.4 and contract all subtrees whose subtree sum is 0. As before, for a graph G and set $S \subseteq V(G)$, let $G(S)$ be G with $V(G) \setminus S$ contracted into a single node. We use the fact that R is binary and has $O(\log n)$ depth to bound the size of resulting tree.

ALGORITHM 7.1. `ca-contraction`(R, S)
Input: α-congestion approximator R (for a graph $G = (V, E, c)$), which is a hierarchical decomposition and a binary tree of depth $O(\log n)$; a subset $S \subset V$ of $|S| = k$ uncontracted vertices.
Output: Tree R' with $O(k \log n)$ nodes that is an $(\alpha \cdot \text{polylog}\, n)$-congestion approximator for $G(S)$.
Procedure:

1: Assign a node weight of 1 to each leaf u of R such that $P_u = \{v\}$ for some $v \in S$, where $P_u \subseteq V$ is the partition of vertices the node u corresponds to in the hierarchical decomposition.
2: Assign a node weight of 0 to all other nodes of R.
3: Compute the subtree sums with respect to these weights using the algorithm of Theorem A.4.
4: From top-down, contract each subtree whose sum is 0 into a single supernode.

It is important to note that this algorithm does *not* require all leaves of the input R to correspond to single vertices, as is the case in a standard hierarchical decomposition tree. In Algorithm 2.2, we contract congestion approximators that themselves have been contracted from a previous tree, and so not all leaves may correspond to single vertices.

[8] with some edges properly reweighted; see Section 6.3. Since this does not affect the algorithm or analysis of this section, we assume this reweighting has been done prior to the call to `ca-contraction`.

LEMMA 7.1. *The* `ca-contraction`(R, S) *subroutine has depth* $O(\log n)$ *and* $O(|R| \log n)$ *total work, and outputs a tree* R' *with* $O(k \log n)$ *nodes.*

Proof. By Theorem A.4 and the fact that R has $O(\log n)$ depth, the algorithm has $O(\log n)$ depth and $O(|R| \log n)$ total work. To bound the size, first note by the fact that R is a hierarchical decomposition, if $u_1, \ldots, u_q$ are the nodes of R at some level i, the $P_{u_1}, \ldots, P_{u_q}$ are a partitioning of V. So, at each level of R, there can be at most k nodes u such that $P_u \cap S \neq \emptyset$, and so at most k nodes can remain uncontracted at each level after contraction. Moreover, since R is a binary tree, every uncontracted node can have at most two supernodes as children, and, by definition, all supernodes are leaves. Since R has depth $O(\log n)$, it thus follows that the size of the resulting tree R' is $O(k \log n)$, as desired. ∎

It thus remains to show that the contracted tree R' can still be used to route flow with low congestion in the graph with $V \setminus S$ contracted into a single node. R' is constructed by contracting some subtrees of R; call the root of these contracted subtrees (along with any remaining leaves corresponding to single nodes from $V \setminus S$) $u_1^*, \ldots, u_q^*$ and let $T_i = P_{u_i^*}$, ordering arbitrarily. So, $T_1 \cup \ldots \cup T_q = V \setminus S$. Let $\bar{G}(S)$ be the graph with each T_i contracted to a single node, but these contracted nodes are *not* further contracted. Since R' is constructed by contracting the nodes corresponding to each T_i in R, R' is a α-congestion approximator for $\bar{G}(S)$ (where α is the congestion achieved by R for routing on G). Our goal is thus to show that R' can be used as an $(\alpha \cdot \text{polylog}\, n)$-congestion approximator for $G(S)$.

Let x be the contracted node (i.e. the node formed by contracting $V \setminus S$ in G) in $G(S)$, and suppose we are given a demand vector b on $G(S)$. Furthermore, for each $i \in [q]$, let x_i be the node in $\bar{G}(S)$ formed by contracting T_i. If we were able to efficiently split the demand b_x into demands for $x_1, \ldots, x_q$ without inducing much additional congestion, then we would be able to use R' as a congestion approximator for $G(S)$. Namely, we first convert the demand on $G(S)$ into the corresponding demand on $\bar{G}(S)$, and then use R' to route the flow on $\bar{G}(S)$ which is also a routing on $G(S)$ (by replacing any x_i with x). So, it remains to show that we may indeed split the demand b_x into demands $b'_{x_1}, \ldots, b'_{x_q}$ such that $b_x = \sum_{i \in [q]} b'_{x_i}$ and b' can be routed on $\bar{G}(S)$ with low congestion. This follows from the fact that we are guaranteed the set of edges leaving S in G are $\Omega(1/\log^9 n)$-well-linked (Claim 7.1), and is formalized in the following lemma. Importantly, since R' remains an α-congestion approximator for $\bar{G}(S)$, for any $Q \subseteq S$, `ca-contraction`(R', Q) is also an α-congestion approximator for $\bar{G}(Q)$[9]. As such, the error does not accumulate when repeatedly contracting a congestion approximator, as we do in Algorithm 2.2.

LEMMA 7.2. *Let* b *be a demand vector on* $G(S)$ *which can be satisfied with congestion 1. Let* $W_x = \sum_{(u,x) \in E} c((u, x))$ *be the sum of capacities of edges incident on* x *in* $G(S)$*, and define* W_{x_i} *similarly for each* x_i *in* $\bar{G}(S)$*. Then, with* b' *the demand vector on* $\bar{G}(S)$ *such* $b'_u = b_u$ *for all* $u \in S$ *and* $b'_{x_i} = W_{x_i} b_x / W_x$*,* b' *can be satisfied with congestion* $O(\log^9 n)$*.*

Proof. Consider some flow f in $G(S)$ which satisfies b with congestion 1. Let $S = \{(u, x) \mid u \in S\}$ be the set of edges leaving S in $G(S)$, and similarly let $S_i = \{(u, x_i) \mid u \in S\}$ be the set of edges incident on x_i in $\bar{G}(S)$. Note that for each $(u, x) \in S$, there exists a corresponding edge (u, x_i) (for some x_i) in $\bar{G}(S)$ with the same capacity, by the construction of x and $x_1, \ldots, x_q$. So, we may convert f into a flow on $\bar{G}(S)$; call this f'. Define the flow vector b^* such that $b^*_{x_i}$ is the net incoming flow to x_i induced by f', and $b^*_u = b_u$ for all $u \in S$. f' has congestion 1, by assumption on b, and satisfies the demand vector b^*, so it follows that b^* can be satisfied in $\bar{G}(S)$ with congestion 1 as well. As S is $\Omega(1/\log^9 n)$-well-linked for each S on which we call `ca-contraction` in Algorithm 2.2 (as shown in Claim 7.1), each S_i is as well by Proposition 4.1. Thus, by definition of well-linked and the fact that b and b^* differ only on the x_i, it follows that b' can be satisfied on $\bar{G}(S)$ with congestion $O(\log^9 n)$. ∎

There is one last point to check: that this splitting procedure can be implemented in $O(\log n)$ depth and $O(|R|)$ work. This is not as simple as it may initially seem: since there could be potentially $\Theta(k \log n)$

[9] We slighly abuse notation to refer to $\bar{G}(Q)$ as the graph with the same components contracted as in the tree output of `ca-contraction`(R', Q)

contracted nodes x_i in $\bar{G}(S)$, naively summing the weight on all edges could result in $\Omega(k^2)$ work. Fortunately, given access to the contracted graph $G(S)$ (which are computed in Algorithm 2.2), we can use subtree sums to implement the demand splitting efficiently. Iterate through these edges, and for each $(u, v) \in B$, where B is the boundary edges of $G(S)$, such that $u \in S$ and $v \in V \setminus S$, assign a node weight of $c((u, v))$ to leaf of R which corresponds to v. Assigning these weights can be implemented in work $O(|B|)$; this suffices to prove that Algorithm 2.2 requires only $\tilde{O}(m)$ total work and does not affect the runtime of `ca-contraction`. Once the node weights have been assigned, compute the subtree sums for each node in R, which can be done in $O(\log n)$ depth and $O(|R|)$ work by Theorem A.4. The sum W_{x_i} used in Lemma 7.2 is then exactly the subtree sum of the node corresponding to the set T_i, allowing us to correctly distribute the demands.

7.2 Implementing Sherman's Algorithm on the Contracted Subgraph. Next, we discuss how we achieve a linear-work, low-depth implementation of Sherman's algorithm on the contracted subgraph $G(S)$, where $S \subset V$ is the set of $|S| = k$ uncontracted vertices, with all other vertices $V \setminus S$ being contracted into a single super-vertex x. We encourage the reader to familiarize themselves with the vanilla implementation of Sherman's algorithm outlined in Appendix C to obtain a better understanding of the discussion that follows.

Recall from the preceding section, that we are given access to an α'-congestion approximator R' for a slightly larger graph $\bar{G}(S)$, where the vertices $V \setminus S$ have been partitioned and contracted into multiple super-vertices $x_1, \ldots, x_q$. In order to use this congestion approximator R' for our desired contracted graph $G(S)$, we need to translate demands on vertices in $G(S)$ into demands on vertices in $\bar{G}(S)$. The optimization problem within the `AlmostRoute` subroutine of [66] therefore becomes a minimization problem over the new congestion potential

$$\phi(f) = \mathsf{lmax}(C^{-1}f) + \mathsf{lmax}(2\alpha' R' P(Bf - b)),$$

where B is the vertex-edge incidence matrix for $G(S)$, and P is a linear operator that projects any demands b supported over vertices of $G(S)$ to demands b' supported over vertices of $\bar{G}(S)$ as described in Lemma 7.2. Due to the fact that R' is a $O(k \log n)$ size, $O(\log n)$ depth tree, evaluating this potential is easy; its computation remains unchanged from the vanilla case described in Appendix C. The only major challenge here is efficiently computing the derivatives of this new potential, specifically, the derivative of the second term

$$\phi_2(f) := \mathsf{lmax}(2\alpha' R' P(Bf - b)).$$

First, observe that the operation $P \cdot B$ effectively constructs a new vertex-edge incidence matrix[10] B' for $\bar{G}(S)$ from the incidence matrix B in the following way: for every edge $e = (u_1, u_2)$ in $G(S)$ with only uncompressed vertices $u_1, u_2 \in S$ as its endpoints, this operation replicates the edge exactly in B', i.e. $B'_{v,e} = B_{v,e} \in \{-1, 1\}$, for $v \in \{u_1, u_2\}$ and 0 otherwise. However, for any edge $e = (u, x)$ (or (x, u)) with one of its endpoints being the contracted vertex x, this operation splits the edge e into fractional copies $e_1 = (u, x_1), \ldots, e_q = (u, x_q)$ with copy e_i having fractional value $\rho_{x_i} := W_{x_i}/W_x$ as defined in Lemma 7.2. i.e. for each $i \in [q]$, $B'_{x_i,e_i} = \rho_{x_i} \cdot B_{x,e}$, $B'_{u,e_i} = \rho_{x_i} \cdot B_{u,e}$, and 0 otherwise. Note that we never consider edges going between two partially contracted vertices x_i, x_j, as they are absent in $G(S)$. While we do not explicitly compute this new incidence matrix B' as doing so naively might exceed our linear (in size of $G(S)$) work requirement, it will serve as a useful intermediate object for analyzing the gradients.

Now let $\mathcal{I}'$ be the set of all cuts considered by our congestion approximator R', and for any cut $i = (S_i, \overline{S_i}) \in \mathcal{I}'$, let $y_i = 2\alpha'[R'P(Bf - b)]_i$ be the congestion induced by the residual demands across

[10] Note that this new vertex-edge incidence matrix B' may not be the same as the actual incidence matrix of $\bar{G}(S)$. However, this is how it effectively appears to Sherman's algorithm when invoked with R' as its input.

cut i. We have that for any edge $e \in G(S)$, the partial derivative

$$\frac{\partial \phi_2(f)}{\partial f_e} = \sum_{i \in \mathcal{I}'} \frac{\partial \phi_2(f)}{\partial y_i} \cdot \frac{\partial y_i}{\partial f_e} = \sum_{i \in I'} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha' B'_{S_i,e}}{c(S_i, \overline{S_i})},$$

where $c(S_i, \overline{S_i})$ is the capacity of cut $i = (S_i, \overline{S_i})$ in $\bar{G}(S)$ considered in our congestion approximator R', and (with some abuse of notation) $B'_{S_i,e}$ represents the total "fraction" of the edge e crossing the cut $(S_i, \overline{S_i})$; for edges $e = (u_1, u_2)$ with only uncompressed vertices as its endpoints, this quantity is $B'_{S_i,e} = \sum_{v \in S_i} B'_{v,e} = \sum_{v \in S_i \cap S} B_{v,e}$, and for edges $e = (u, x)$ (or (x, u)) with one of its endpoints being the compressed vertex, this quantity is $B'_{S_i,e} = \sum_{j \in [q]} B'_{S_i,e_j} = \sum_{j \in [q]} \sum_{v \in S_i} B'_{v,e_j} = \sum_{v \in S_i \cap S} B_{v,e} + \sum_{v \in S_i \setminus S} \rho_v \cdot B_{x,e}$, where the set $\{e_j\}_{j \in [q]}$ correspond to the fractional copies of edge e constructed by the operation $P \cdot B$.

In order to efficiently compute this gradient, we shall again exploit the fact that R' is represented by a rooted hierarchical decomposition tree T' of size $O(k \log n)$ and depth $O(\log n)$. To do so, we use the same node-potential trick as in Appendix C: for any internal node j in T' (which in turn corresponds to a cut $(S_j, \overline{S_j})$), we define the node potential π_j as

$$\pi_j := \sum_{i \in T'_{j,r}} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha'}{c(S_i, \overline{S_i})},$$

where $T'_{j,r}$ denotes the path in T' from node j to the root r of T'. Now observe that, following an identical calculation as in Appendix C, the gradient of $\phi_2(f)$ w.r.t. the flow f_e on any edge $e = (u_1, u_2)$ with only uncompressed vertices as its end-points remains unchanged from the vanilla case.

$$\frac{\partial \phi_2(f)}{\partial f_e} = \sum_{i \in T'_{u_1,u_2}} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha B'_{S_i,e}}{c(S_i, \overline{S_i})} = \pi_{u_2} - \pi_{u_1},$$

where T'_{u_1,u_2} denotes the unique path between u_1, u_2 in the tree T'. For any edge $e = (u, x)$ (or (x, u)) with one of its end points being the contracted vertex x that is in turn partitioned into fractional edges $e_1 = (u, x_1), \ldots, e_q = (u, x_q)$ in B', observe that this gradient

$$\begin{aligned}\frac{\partial \phi_2(f)}{\partial f_e} &= \sum_{j \in [q]} \sum_{i \in T'_{u,x_j}} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha B'_{S_i,e_j}}{c(S_i, \overline{S_i})} \\ &= \sum_{j \in [q]} \rho_{x_j}(\pi_{x_j} - \pi_u) = \left(\sum_{j \in [q]} \rho_{x_j} \pi_{x_j} \right) - \pi_u,\end{aligned}$$

where the final equality follows from the fact that $\sum_{j \in [q]} \rho_{x_j} = 1$. We can easily compute all these node potentials π_v (through a prefix sum over an Eulerian tour of the congestion approximator tree T' starting at its root) and precompute the quantity $\sum_{j \in [q]} \rho_{x_j} \pi_{x_j}$ in the PRAM model with $O(|R'|) = O(k \log n)$ work and $O(\log n)$ depth. Since this is all we need, namely, be able to efficiently evaluate congestions and compute gradients over the supplied congestion approximator, we have an efficient implementation of Sherman's algorithm on the contracted subgraph $\bar{G}(S)$.

7.3 Computing a Congestion Approximator for $G \cup \{s, t\}$. The final requirement is the following: in the cut-matching game of Section B, we need to compute $(1 - 1/\operatorname{polylog} n)$-approximate max flow on G with the addition of a source s and sink t which are arbitrarily connected to G. As such, we must convert our α-congestion approximator R for G into a $O(\alpha \operatorname{polylog} n)$-congestion approximator R' for $G \cup \{s, t\}$. In this section, we specify how this is achieved.

LEMMA 7.3. *Let R be a α-congestion approximator for G which is a hierarchical decomposition, and let s,t be two additional vertices connected arbitrarily (and with arbitrary capacities) to G. Then, there is a* $\operatorname{polylog} k$ *depth,* $\tilde{O}(k)$ *work* PRAM *algorithm which computes a* $O(\alpha \operatorname{polylog} n)$*-congestion approximator R' for $G \cup \{s,t\}$, where $k = |R|$.*

Note that $R \cup \{s,t\}$, with s,t connected to the leaves of R corresponding to their neighbors, can be used to route any feasible flow on $G \cup \{s,t\}$ with congestion α, since R is an α-congestion approximator for G. However, $R \cup \{s,t\}$ is not a tree, and thus cannot be readily plugged into Sherman's algorithm to compute maximum flows in $G \cup \{s,t\}$. It thus remains to obtain a congestion approximator (that is a tree) for $G \cup \{s,t\}$, which boils down to computing a hierarchical decomposition (as in [61]) for $R \cup \{s,t\}$. The algorithm is quite similar to the procedure of Section 5.2, but with a few key modifications to account for the nodes s and t.

Recall from Section 5.2 that it suffices to implement the two partitioning steps Partition A and Partition B; we use the same definitions as in Section 5.2.

LEMMA 7.4. *Partition A*[11] *can be implemented in $O(\log |Q|)$ depth and $\tilde{O}(|Q|)$ work on any subset Q of nodes of $R \cup \{s,t\}$.*

Proof. We use the following procedure:

1. If $s \in Q$, then output the partition $(\{s\}, Q \setminus \{s\})$
2. If $t \in Q$ and $s \notin Q$, then output the partition $(\{t\}, Q \setminus \{t\})$
3. If $s \notin Q$ and $t \notin Q$, then Q induces a subtree of R, and we run the procedure of Lemma 5.3

The depth and work are immediate from Lemma 5.3. It remains to show that the edges between the partition that is output are 1-well-linked. If $s \in Q$ or $t \in Q$, then all the edges between the outputted partitions are incident on s or t and are thus 1-well-linked. If $s \notin Q$ and $t \notin Q$, then the edges between the outputted partitions are 1-well-linked by Lemma 5.3. ∎

For Partition B, we only need to apply it on partitions without s or t, or on partitionings where all boundary edges are incident on either s or t, by the construction of Partition A. As such, Partition B reduces to finding the (exact) min-cut on a tree with a source and sink added, exactly as in Section 5.2. So, we may use the algorithm of Appendix D to compute Partition B and the analysis follows from Lemma 5.4.

The proof of Lemma 7.3 is then essentially identical to the proof of Lemma 5.5.

8 Parallel Flow Decomposition by Shortcutting.

We describe our PRAM flow decomposition subroutine in this section, and we will begin by specifying the relevant notation. Consider an s-t flow f specified by a weighted, directed graph $H = (V, E, f)$ containing a source vertex $s \in V$, and sink vertex $t \in V$, with the flow on any edge given by $f : E \to \mathbb{R}^+$. Note that this flow network H is restricted to only the subset of edges that carry *positive* flow, and will be iteratively updated by our algorithm as we make progress towards our flow-decomposition objective. Moreover, while the initial graph H specifying the flow f does not contain any parallel edges, such edges will inevitably end up being created in our flow-decomposition process. Therefore in this section, we will more generally deal with *multigraphs*, whose edges are assumed to be uniquely indexed. Let $|f| := \sum_{(s,v)\in E} f_{(s,v)} - \sum_{(v,s)\in E} f_{(v,s)}$ be the value of the s-t flow. Lastly, we use $S = N^H_{\text{out}}(s)$ to denote the out-neighbors of s corresponding to the "source-side" vertices, and $T = N^H_{\text{in}}(t)$ to denote the in-neighbors of t corresponding to the "sink-side" vertices.

[11] If $s \in Q$ or $t \in Q$, the partitions are not balanced, which does not meet the exact definition of Partition A. However, there are only be 2 such paritionings, and so this can only increase the depth of the final tree by 2.

Our goal is to determine how much flow in f is routed between each pair $x \in S, y \in T$. We do so by computing *pairwise* demands[12] $d : S \times T \to \mathbb{R}^+$ such that $\|d\|_1 = |f|$ and d can be routed in H exactly with f being the edge-capacity constraints, and as in the parallel setting, this objective more easily admits a small work and low-depth implementation. We will build a DAG data structure that implicitly encodes the necessary information.

DEFINITION 8.1. (FLOW DECOMPOSITION DAG) *An ℓ-layered directed graph $\mathcal{D}$ is a flow decomposition DAG of an s-t flow f specified by a weighted, directed graph H iff*

1. *Each node of $\mathcal{D}$ corresponds to a directed edge (not necessarily in H) between two vertices u, v in V associated with a flow value $h_{(u,v)} \in \mathbb{R}^+$. We use (u, v, h) to represent this node in $\mathcal{D}$, dropping the subscript (u, v) in the flow value $h_{(u,v)}$ when it is unambiguous for ease of exposition. Note that there can be multiple nodes of the form $(u, v, h_{(u,v)})$ at the same layer of $\mathcal{D}$ due to the existence of parallel (u, v) edges, each with potentially different flow values $h_{(u,v)}$. We assume that all nodes in $\mathcal{D}$ are uniquely indexed to avoid ambiguity.*

2. *The nodes (u, v, h) at the lowest layer 1 do not have any predecessors. In particular, each node (u, v, h) in layer 1 corresponds to a directed edge $(u, v) \in E$ in H, and has the same flow value $h_{(u,v)} = f_{(u,v)}$ as in H.*

3. *Each node (u, v, h) in any other layer $l \in \{2, \ldots, \ell\}$ has at most two predecessors in the previous layer $l-1$. If (u, v, h) has two predecessors, then they have the form (u, w, g), (w, v, g') corresponding to a length-two path between u, v; this means that (u, v, h) is obtained by "merging" (a part of the) flows (u, w, g) and (w, v, g'). If (u, v, h) has only one predecessor, it must have the form (u, v, g). This means that (u, v, g) had some residual flow $h \leq g$ that was not merged with any other node.*

4. *The nodes at each layer $l \in \{1, \ldots, \ell - 1\}$ satisfy flow conservation with the succeeding layer $l + 1$, i.e. for each node (u, v, h) at any layer l having successors $\Pi(u, v, h)$ at layer $l + 1$, we have $\sum_{(x,y,g)\in\Pi(u,v,h)} g_{(x,y)} = h_{(u,v)}$.*

5. *The nodes at the top-most layer ℓ are only of the form (s, t, h), and have total flow value equal to $|f|$, i.e. these nodes all correspond to parallel (s, t) edges, which together account for all of the s-t flow f.*

The size of this data-structure is measured in terms of the total number of nodes and edges it contains, where the edges denote successor-predecessor relationships between nodes across consecutive layers. If the size and the number of layers of a flow decomposition DAG are both low, we can efficiently perform various PRAM computations with low work and depth therein. Of particular importance to our approximate max-flow algorithm, for a η-size ℓ-layer flow decomposition DAG, we can compute the second vertex (which is a neighbor of s) and penultimate vertex (which is a neighbor of t) of every flow path in layer ℓ simultaneously with $O(\eta)$ work and $O(\ell + \log \eta)$ depth using a simple algorithm; we show this in Lemma 8.2.

We now show that a small-size and low-depth flow decomposition DAG can indeed be found efficiently by slightly relaxing property 5 in Definition 8.1; specifically, given a parameter $\delta \in (0, 1)$, we have that the nodes of the form $(s, t, \cdot)$ (i.e. parallel (s, t) edges) in layer ℓ together account for a $(1 - \delta)$ fraction of the s-t flow value $|f|$ (at a small cost to its size and depth). This is sufficient for all our applications, since we are only concerned with *approximate* max flows. The precise guarantees are given in the following main theorem of this section.

[12] For the objective of computing pairwise demands, we assume that the source and sink-side vertices are non-overlapping, i.e. $S \cap T = \emptyset$, since this will always be the case in our application which is the implementation of the cut-matching game. The parallel flow-decomposition result however, is more generally applicable.

THEOREM 8.1. (PARALLEL FLOW DECOMPOSITION) *There exists a* PRAM *algorithm* `flow-decomp` *that given any parameter* $\delta \in (0,1)$ *and a polynomially bounded* s-t *flow specified by a weighted, directed graph* $H = (V, E, f)$ *with flow value* $|f|$*, finds with high probability a flow decomposition DAG* $\mathcal{D}$ *of* $\ell = O(\log(n/\delta))$ *layers such that at the topmost layer* ℓ*, the total value of the flow captured by nodes* $(s, t, \cdot)$ *representing (parallel)* (s, t) *edges is at least* $(1-\delta)|f|$*. The algorithm* `flow-decomp` *has total work* $O(m \text{ polylog}(n) \log(n/\delta))$ *and the total depth is* $O(\text{polylog}(n) \log(n/\delta))$*. The total number of nodes, and edges representing successor-predecessor relationships between nodes across consecutive layers in* $\mathcal{D}$ *are both bounded by* $O(m \log(n/\delta))$*.*

The idea behind our proof of Theorem 8.1 can be illustrated by the following relatively intuitive process[13]: repeatedly "shortcut" the flow graph H by replacing a length-two flow path $u \to w \to v$ with a single edge (u, v) having maximal flow value $h_{(u,v)} = \min\{g_{(u,w)}, g_{(w,v)}\}$, and a residual edge $(x, y) \in \{(u, w), (w, v)\}$ having flow value $h_{(x,y)} = |g_{(u,w)} - g_{(w,v)}|$ iff there is any non-zero leftover flow not accounted for by this shortcut edge (u, v). Consequently in the flow-decomposition DAG, the nodes $(u, w, g_{(u,w)})$ and $(w, v, g_{(w,v)})$ become the predecessors of this "shortcut edge" $(u, v, h_{(u,v)})$, and the node $(x, y, g_{(x,y)})$ becomes the predecessor of the residual edge $(x, y, h_{(x,y)})$ if there is any leftover flow. In order to achieve low depth, our objective is to find a collection of length-two flow paths that together account for a large fraction of the (total ℓ_1 norm of the) flow that does not directly go from s to t which we can then shortcut in parallel. We show that we can find such a collection of flow paths efficiently with the following technical lemma.

LEMMA 8.1. *Let* f *be an* s-t *flow of value* $|f|$ *specified by a weighted, directed multigraph* $H = (V, E, f)$ *containing source vertex* $s \in V$*, sink vertex* $t \in V$ *with no edges directly connecting* s *to* t *and with no self-loops, and the flow on any edge being given by* $f : E \to \mathbb{R}^+$*. Then we can find in* $O(m \text{ polylog}(n))$ *work and* $O(\text{polylog}(n))$ *depth, a collection of "shortcut paths" represented as tuples* $\mathcal{P} := \left\{(e_i^{(1)}, e_i^{(2)}, h_i)\right\}$ *where* $e_i^{(1)} \neq e_i^{(2)} \in E$*,* $h_i \in \mathbb{R}^+$*, along with a collection of "residual edges" given by* $\mathcal{R} := \{(e_i, r_i)\}$ *where* $e_i \in E$ *are disjoint, and* $r_i \in \mathbb{R}^+$ *such that*

1. *(Length-two paths) For each* $i \in [|\mathcal{P}|]$*,* $e_i^{(1)}, e_i^{(2)}$ *form a path of length two.*
2. *(Flow constraints) For each edge* $e \in E$*, the total flow accounted for by the shortcut paths involving this edge, which is given by the summation of* h_i *over the tuples in* $\mathcal{P}$ *in which* e *appears along with its residual capacity* r_i *if any (if* e *is present in* $\mathcal{R}$*), is exactly* f_e*:*

$$\sum_{i \in [|\mathcal{P}|]: e \in \left\{e_i^{(1)},\ e_i^{(2)}\right\}} h_i + \sum_{i \in [|\mathcal{R}|]: e = e_i} r_i = f_e.$$

3. *(Large* ℓ_1*-norm) With probability at least* $1/15$*,* $\sum_{i \in [|\mathcal{P}|]} h_i \geq \|f\|_1 / 16$*.*
4. *(Non-increasing flow-support)* $|\mathcal{P}| + |\mathcal{R}| \leq |E|$*.*

Proof. (Theorem 8.1) Given a flow f specified by a weighted directed graph $H = H_1$, we begin by creating the lowest layer 1 of our flow decomposition DAG $\mathcal{D}$ from H_1 as specified in property 2 of Definition 8.1. It is easy to see that this can be achieved with $O(m)$ total work and $O(1)$ depth. We also create a set $\mathcal{S}$, initially empty, to track tuples of the form $(s, t, h_{(s,t)})$ (i.e. direct (s, t) edges) which correspond to "fully processed" s-t flow paths. If H_1 contains such a direct edge, we delete it from H_1 and add the tuple $(s, t, f_{(s,t)})$ to $\mathcal{S}$. The subsequent proof (and algorithmic procedure) then follows from a repeated application of Lemma 8.1; the $l > 1$-th iteration, given as input an s-t flow specified by a weighted, directed *multigraph* H_{l-1}, we have:

[13] This is only an illustration of our idea, see Section 8.1 and the proof of Proposition 8.1 for what we actually do.

1. Invoke Lemma 8.1 on H_{l-1} to find the desired collection of shortcut paths $\mathcal{P}_l = \left\{(e_i^{(1)}, e_i^{(2)}, h_i)\right\}$, and residual edges $\mathcal{R}_l = \{(e_i, r_i)\}$. Set $\mathcal{S}_l$ to be initially empty.

2. Construct a new flow graph H_l, initially empty, as follows: for each tuple $(e_i^{(1)}, e_i^{(2)}, h_i) \in \mathcal{P}_l$, add a directed edge (u_i, v_i) with flow value $f_{(u_i,v_i)} = h_i$, where u_i, v_i are the endpoints[14] of the length-two path $(e_i^{(1)}, e_i^{(2)})$. For each residual edge $(e_i, r_i) \in \mathcal{R}_l$, add a directed edge e_i with flow value r_i. Note that this may lead to the creation of parallel edges. Delete any edge (s, t) in H_l (may be multiple), as these correspond to "fully processed" flow paths that will be tracked separately in $\mathcal{S}_l$. Also delete any self-loops (edges with both endpoints being the same vertex) from H_l.

3. Construct layer l of the flow-decomposition DAG $\mathcal{D}$ as follows: for each tuple $\{e_i^{(1)}, e_i^{(2)}, h_i\} \in \mathcal{P}_l$, add to the l^{th} layer of $\mathcal{D}$ a node (u_i, v_i, h_i), where u_i, v_i are the endpoints[15] of the length-two path $(e_i^{(1)}, e_i^{(2)})$. We set the predecessors of (u_i, v_i, h_i) to be the nodes $(u_i^{(1)}, v_i^{(1)}, f_{(u_i^{(1)}, v_i^{(1)})})$ and $(u_i^{(2)}, v_i^{(2)}, f_{(u_i^{(2)}, v_i^{(2)})})$ from layer $l-1$, where $e_i^{(j)} = (u_i^{(j)}, v_i^{(j)})$ for $j \in \{1, 2\}$. For each residual edge $(e_i, r_i) \in \mathcal{R}_l$, add to the l^{th} layer of $\mathcal{D}$, a node (u_i, v_i, r_i) where $e_i = (u_i, v_i)$. We set the predecessor of (u_i, v_i, r_i) to be the node $(u_i, v_i, f_{(u_i,v_i)})$ from layer $l-1$. For each newly created node of the form (s, t, h) at layer l, add this node to $\mathcal{S}$. For each node $(s, t, h_i) \in \mathcal{S}_{l-1}$, add to the l^{th} layer of $\mathcal{D}$, a new node (s, t, h_i), and set its predecessor to be the corresponding (s, t, h_i) node from layer $l-1$. Finally, create $\mathcal{S}_l$ by adding all newly created $(s, t, \cdot)$ tuples in $\mathcal{S}$ to $\mathcal{S}_{l-1}$.

Observe that in each iteration, the updated flow specified by the multigraph H_l (prior to deleting (s, t) edges and self-loops if any) can trivially be routed in the preceding graph H_{l-1} by simply "undoing" the shortcutting that produced H_l. Moreover, after accounting for all the (s, t) edges specified by tuples in $\mathcal{S}_l$, the total flow value leaving s is preserved across all iterations in our procedure. Therefore, the "routability" property desired of the flow decomposition procedure is trivially satisfied by our aforementioned process.

We shall now prove the work and depth guarantees of this algorithm, along with the near linear-size and polylogarithmic depth of the resulting flow-decomposition DAG $\mathcal{D}$. Since in every iteration, the ℓ_1-norm of the flow f reduces by $\sum_{i \in \mathcal{P}} h_i$, which by property (3) of Lemma 8.1 is at least a constant-fraction of the ℓ_1-norm of the flow f at the start of the iteration with constant probability, we can deduce that after $\ell = \Theta(\log \frac{n\|f\|_1}{\delta|f|}) = \Theta(\log(n/\delta))$ iterations of the above process, with polynomially high probability (in n, follows by a straightforward Chernoff bound), the edges that remain in the flow graph H_ℓ (i.e. that do not directly go from s to t) contribute at most a δ fraction of the total (initial) amount of flow $|f|$ leaving s. Since in all our applications, we are only concerned with finding an approximate max-flow, we can safely ignore this residual flow for small enough δ. The second equality in the above bound follows by observing that $|f| \geq \min_{e \in E} w_e$, and $\|f\|_1 \leq \sum_{e \in E} w_e \leq m \cdot \max_{e \in E} w_e$, and the aspect ratio $\max_{e \in E} w_e / \min_{e \in E} w_e$ is assumed to be polynomially bounded.

In each iteration l, our procedure involves obtaining the relevant collection of tuples $\mathcal{P}_l, \mathcal{R}_l$ given an input flow multigraph H_{l-1}, which in turn are used to add a new layer l to the flow-decomposition DAG $\mathcal{D}$, and to update the flow multigraph $H_{l-1} \to H_l$. The total number of edges $|E_l|$ in the updated flow multigraph H_l is at most $|\mathcal{P}_l| + |\mathcal{R}_l| - |\mathcal{S}_l \setminus \mathcal{S}_{l-1}|$, which by property (4) of Lemma 8.1 is at most $|E_{l-1}|$, the total number of edges in the flow multigraph H_{l-1} from the previous iteration $l-1$. Moreover, the total number of nodes added to layer l in $\mathcal{D}$ is at most $|\mathcal{P}_l| + |\mathcal{R}_l| + |\mathcal{S}_{l-1}| = |\mathcal{P}_l| + |\mathcal{R}_l| - |\mathcal{S}_l \setminus \mathcal{S}_{l-1}| + |\mathcal{S}_l \setminus \mathcal{S}_{l-1}| + |\mathcal{S}_{l-1}| \geq |E_l| + |S_l|$. However, by property (4) of Lemma 8.1, we also have that $|\mathcal{P}_l| + |\mathcal{R}_l| \leq |E_{l-1}|$, due to which we have that $|\mathcal{P}_l| + |\mathcal{R}_l| + |\mathcal{S}_{l-1}| \leq |E_{l-1}| + |\mathcal{S}_{l-1}|$. Therefore, we can infer that the quantity $|E_{l'}| + |\mathcal{S}_{l'}|$ is non-increasing across l', due to which we can

[14] Since we are only guaranteed an approximate max-flow f, it may contain circulations that we may discover in this process, i.e. the two endpoints u_i, v_i are identical. In this case, we can simply delete all the self-loops.

[15] If $u_i = v_i$, then skip this tuple without adding any nodes/edges.

conclude that $|E_{l'}| + |\mathcal{S}_{l'}| \leq |E_1| + |\mathcal{S}_1| = |E|$. The above bounds, and the fact that the number of edges $|E|$ in the initial flow graph H provided as input to our algorithm is at most m combined with the computational guarantees of Lemma 8.1 gives us that any iteration of our algorithm (finding tuples, updating $H_{l-1} \to H_l$, creating a new layer in $\mathcal{D}$) can be implemented in $O(m \operatorname{polylog} n)$ work and $O(\operatorname{polylog} n)$ depth. Moreover, each new layer created in the flow decomposition DAG has at most m nodes, and at most $2m$ edges between it and the preceding layer (since every node has at most two predecessors from the previous layer). Since there are a total of $\Theta(\log \frac{n\|f\|_1}{\delta|f|}) = O(\log(n/\delta))$ iterations of our algorithm for polynomially bounded flows, we have that the total work and depth of our algorithm is $O(m \operatorname{polylog}(n) \log(1/\delta))$ and $O(\operatorname{polylog}(n) \log(1/\delta))$, respectively, and the size (number of nodes,edges) and depth of the flow decomposition DAG $\mathcal{D}$ produced is at most $O(m \log(n/\delta))$, and $O(\log(n/\delta))$, respectively as claimed. ∎

8.1 Proof of Lemma 8.1. One can formulate the problem as a (uncapacitated) b-matching problem. Specifically, we create a b-matching instance H_b from the flow multigraph $H = (V, E, f)$ (with no edge directly connecting s to t) as follows. For each edge $e \in E$, we create a vertex u_e with demand f_e. Then for every pair of edges $e^{(1)}, e^{(2)}$ that form a length-two path by sharing a vertex $v \in V$, we connect their corresponding vertices $u_{e^{(1)}}, u_{e^{(2)}}$ with an edge of infinite capacity. It suffices to find a large b-matching whose size is a constant fraction of $\|f\|_1$, the total ℓ_1-norm of the flow f.

First, we shall prove such a b-matching exists. To this end, let us consider a bipartite b-matching instance H'_b, constructed as follows. For each edge $e \in E$, we create two vertex copies $u_{e_{\text{in}}}$ and $u_{e_{\text{out}}}$ both with demand f_e. Then for every pair of edges $(e^{(1)}, e^{(2)})$ that form a length-two path by sharing a vertex $v \in V$, we add an edge between $u_{e^{(1)}_{\text{in}}}$ and $u_{e^{(2)}_{\text{out}}}$ with infinite capacity. We claim that there is a b-matching of size at least $\|f\|_1/2$, by the following simple construction. For each vertex $v \in V$ such that $v \neq s$ and $v \neq t$, we use a perfect b-matching between the vertices corresponding to the "in" copies of the incoming edges of v and the vertices corresponding to the "out" copies of the outgoing edges of v, whose existence is guaranteed by the flow conservation property. Then both vertices corresponding to every edge $e \in E$ is fully matched except for the vertices corresponding to the "out" copies of edges leaving the source s, and the vertices corresponding to the "in" copies of edges entering the sink t. By a simple charging argument that assigns the b-matching value for every vertex $v \in V$ to the "in" copies of the vertices corresponding to the incoming edges into v, we can bound the total demand of these aforementioned unmatched edges by at most $\|f\|_1/2$, which gives us our claim. Now consider for each edge $e \in E$, keeping either $u_{e_{\text{in}}}$ or $u_{e_{\text{out}}}$ uniformly at random and discarding the other vertex, letting the resulting subsampled graph be H_b. Then in expectation, observe that the maximum b-matching in the resulting subsampled graph is at least $\|f\|_1/8$, since each matched edge is kept with probability $1/4$ (i.e. if its two end points, one of which is a vertex corresponding to the "in" copy of an incoming edge, and the other is a vertex corresponding to the "out" copy of an outgoing edge are both sampled in the subsampled graph H_b). Therefore, by a standard Markov argument, we have that with probability at least $1/15$, the matching size in the subsampled graph is at least $\|f\|_1/16$.

We can in fact algorithmically compute such a matching efficiently in light of the above construction. As described above, we can subsample the vertices corresponding to the edge-copies in the graph H'_b and find the maximum b-matching in the resulting graph by locally finding, for each vertex $v \in V \setminus \{s, t\}$, the maximum b matching between the (subsampled) vertices corresponding to "in" copies of incoming edges into v, and (subsampled) vertices corresponding to "out" copies of outgoing edges from v in H. This is correct because after we keep either $u_{e_{\text{in}}}$ or $u_{e_{\text{out}}}$ and discard the other for each e in H, the graph becomes a union of disjoint connected components, where each component consists of the "in" copies of the incoming edges and the "out" copies of the outgoing edges of a single vertex. It then remains to show that we can efficiently find the maximum b-matching locally for every component, along with the non-increasing flow-support property for which it suffices to prove the following proposition.

PROPOSITION 8.1. *Given two sets of elements $A = \{a_1, \ldots, a_{n_{\text{in}}}\}$ and $B = \{b_1, \ldots, b_{n_{\text{out}}}\}$, along with a positive function $f : A \cup B \to \mathbb{R}^+$ we can find in $O((n_{\text{in}} + n_{\text{out}}) \operatorname{polylog}(n_{\text{in}} + n_{\text{out}}))$ total work and*

$O(\log(n_{\text{in}} + n_{\text{out}}))$ *depth, collections of tuples* $\mathcal{P} := \{(x_i, y_i, h_i)\}$ *where* $x_i \in A, y_i \in B, h_i \in \mathbb{R}^+$, *and* $\mathcal{R} := \{(z_i, r_i)\}$ *where* $z_i \in A \cup B$ *are disjoint and* $r_i \in \mathbb{R}^+$ *such that*

1. *(Flow constraints) For each* $a \in A$ *(and each* $b \in B$*), we have*

$$\sum_{i \in [|\mathcal{P}|]: x_i = a} h_i + \sum_{i \in [|\mathcal{R}|]: a = z_i} r_i = f_a, \quad \text{and}$$
$$\sum_{j \in [|\mathcal{P}|]: y_j = b} h_j + \sum_{j \in [|\mathcal{R}|]: b = z_j} r_j = f_b.$$

2. *(Non-increasing support)* $|\mathcal{P}| + |\mathcal{R}| \le n_{\text{in}} + n_{\text{out}}$

3. *(Maximality)* $\sum_{i \in [|\mathcal{P}|]} h_i = \min\left\{\sum_{a \in A} f_a, \sum_{b \in B} f_b\right\}$.

Proof. The proof is standard and it appeared in earlier work like [5], though we describe it explicitly for completeness. We begin by computing the prefix sum of the values in both A and B; with some abuse of notation, let $A_k = \sum_{i=1}^{k} f_{a_i}$ and $B_k = \sum_{i=1}^{k} f_{b_i}$. By a standard PRAM algorithm ([3]), this procedure takes $O(n_{\text{in}} + n_{\text{out}})$ work and $O(\max\{\log n_{\text{in}}, \log n_{\text{out}}\}) = O(\log(n_{\text{in}} + n_{\text{out}}))$ depth. Next, we rank (sort) all the values in $\{A_i\}_{i \in [n_{\text{in}}]} \cup \{B_j\}_{j \in [n_{\text{out}}]}$, which takes $O((n_{\text{in}} + n_{\text{out}}) \log(n_{\text{in}} + n_{\text{out}}))$ work and $O(\log(n_{\text{in}} + n_{\text{out}}))$ depth; let C_k be the k-th ranked value in the resulting sorted prefix sum values, with c_k denoting the element a_i or b_j depending on whether $C_k = A_i$ or $C_k = B_j$. If there are ties (i.e. the total value A_i of the first i elements of A is exactly equal to the total value B_j of the first j elements of B for some i, j), then set $c_k = a_i$, and drop the $C_{k'}$ term corresponding to B_j after setting b_j to be the successor of c_k, which we define next. For each element c_k, find its successor $c_{\text{succ-}k}$ to be the element from the other array corresponding to the smallest prefix sum that is at least as large as C_k, i.e., if $c_k = a_i$ for some i (i.e. $C_k = A_i$), then the successor $c_{\text{succ-}k} = b_j$ where $j = \arg\min_{\ell: B_\ell \ge C_k = A_i} B_\ell$. If there is no such value (i.e. $C_k = A_i > B_{n_{\text{out}}}$), then we set the successor $c_{\text{succ-}k}$ to be a dummy element $\perp$. The case where $c_k = b_j$ follows symmetrically. We also let $C_0 = 0$ so that the subsequent process is well defined. Finally let C be the resulting processed, sorted array, and let n_{tot} be its length (i.e. array C contains $C_0, C_1, \ldots, C_{n_{\text{tot}}-1}$). This procedure can also be done in $O((n_{\text{in}} + n_{\text{out}}) \cdot \log(n_{\text{in}} + n_{\text{out}}))$ work and $O(\log(n_{\text{in}} + n_{\text{out}}))$ depth by the standard doubling trick. Now starting with both $\mathcal{P}, \mathcal{R}$ being initially empty, do the following: for each $\ell \in \{1, \ldots, n_{\text{tot}} - 1\}$ in parallel: if $c_\ell = a_i$, and $c_{\text{succ-}\ell} \neq \perp$, then add the tuple $(c_\ell, c_{\text{succ-}\ell}, C_\ell - C_{\ell-1})$ to $\mathcal{P}$; if $c_\ell = b_j$, and $c_{\text{succ-}\ell} \neq \perp$, then add the tuple $(c_{\text{succ-}\ell}, c_\ell, C_\ell - C_{\ell-1})$ to $\mathcal{P}$; otherwise, add the tuple $(c_\ell, C_\ell - C_{\ell-1})$ to $\mathcal{R}$. This entire process requires just $O(n_{\text{in}} + n_{\text{out}})$ work and $O(1)$ depth. This proves our computational guarantees.

We shall now prove the flow constraint, non-increasing support, and maximality properties outlined in the proposition statement. To prove the flow constraint property, consider any fixed element b_j for some j. By nature of our algorithm, b_j appears in tuples due to one of two reasons: either (a) it was the successor $b_j = c_{\text{succ-}k'}$ for some elements $c_{k'}$ (the number of such elements is ≥ 0), or (b) when the element $C_k = B_j$ was processed by itself (exactly once if B_j was not tied with some A_i, in which case it was combined with its successor $c_{\text{succ-}k}$ to form a tuple and added to set $\mathcal{P}$ if $c_{\text{succ-}k} \neq \perp$, and to set $\mathcal{R}$ otherwise. If tied with some A_i, then this does not occur as there is no $k : C_k = B_j$). In the former, observe that b_j will be the successor of all elements $c_{k'} = a_i$ with value $C_{k'} = A_i$ where $B_{j-1} < C_{k'} = A_i \le B_j$. If we sum the $h_{k'}$ values corresponding to all such k', including the final h_k value (which occurs only if there is a $k : C_k = B_j$), we get $\sum_{k': B_{j-1} < C_{k'} \le B_j} h_{k'} + \mathbf{1}(\exists\, k : C_k = B_j) h_k = \sum_{k': B_{j-1} < C_{k'} \le B_j} (C_{k'} - C_{k'-1}) + \mathbf{1}(\exists\, k : C_k = B_j)(C_k - C_{k-1}) = B_j - B_{j-1} = f_{b_j}$, where the final equality follows by telescoping summation. The argument where $c_k = a_i$ follows symmetrically. The non increasing support property trivially follows by observing that the total number of tuples added to $\mathcal{P}, \mathcal{R}$ together is exactly $n_{\text{tot}} - 1 \le n_{\text{in}} + n_{\text{out}}$ by definition of C. Lastly, to prove maximality, observe that the largest term $C_{n_{\text{tot}}-1}$ must be achieved at either $A_{n_{\text{in}}}$ or $B_{n_{\text{out}}}$. Let us assume that it is $B_{n_{\text{out}}}$, in which case it must be the case that every C_k term corresponding to some A_i value will have a successor in B,

in which case the total capacity of $A_{n_{\text{in}}} = \sum_{i=1}^{n_{\text{in}}} f_{a_i}$ must be accounted for by the tuples in $\mathcal{P}$, since no a_i element will end up in the residual set $\mathcal{R}$. The argument for the other case where the largest term $C_{n_{\text{tot}}-1}$ is achieved at $A_{n_{\text{in}}}$ follows symmetrically. ∎

8.2 Computing Fractional Matching. It remains to show how, for each neighbor u of s, we may determine how much flow through u is routed through each neighbor of t. This allows us to use the flow to compute a (fractional) matching between the neighbors of s and the neighbors of t, which we use in the cut matching game (see Section B). Note also that in our use, the flow f that we want to decompose never contains any edges connecting s directly to t. The following lemma shows that this can be done using a flow-decomposition DAG; combined with Theorem 8.1, this lemma shows that this can be done in logarithmic depth and near-linear work.

LEMMA 8.2. *Given a graph H, and vertices $s, t \in V(H)$, let S and T be the set of neighbors of s and t in H, respectively. Given a flow f without edges connecting s directly to t and a flow-decomposition DAG $\mathcal{D}$ of size η and depth ℓ, let P be the set of flow paths corresponding to the nodes of the form $(s, t, \cdot)$ at the final layer ℓ. There exists a $O(\ell + \log \eta)$ depth, $O(\eta)$ work* PRAM *algorithm which computes $r_{x,y}$ for each $x \in S$ and $y \in T$ such that a total of $r_{x,y}$ units of flow is routed by the flow paths in P whose second vertex is x and penultimate vertex is y.*

Proof. For each node u of the form $(s, t, \cdot)$ at layer ℓ of the flow-decomposition DAG $\mathcal{D}$, our goal is to compute $\texttt{search}(u)$, which is the set $\{x, y\}$ such that $x \in S$ is the second node on the path represented by u and $y \in T$ is the penultimate node. To aid in presentation, for each DAG node v of the form $v = (s, a, \cdot)$ where $a \neq t$, we define $\texttt{search}(v)$ to be only the second node $x \in S$ along the path represented by v; we analogously define $\texttt{search}(v)$ to be only the penultimate node on the path when $v = (a, t, \cdot)$. Note that we are only interested in $\texttt{search}(v)$ of nodes $u = (v_1, v_2, \cdot)$ of $\mathcal{D}$ such that $v_1 = s$ or $v_2 = t$.

Starting from each node u of the form $(s, t, \cdot)$ at level ℓ, in parallel we recursively compute $\texttt{search}(u)$ as follows:

- If u has exactly one predecessor p, return $\texttt{search}(p)$.
- If u is at level 1 in $\mathcal{D}$, then either $u = (s, x, \cdot)$ for some $x \in S$ or $u = (y, t, \cdot)$ for some $y \in T$ (recall that the initial given flow f does not contain edges directly connecting s to t); return x (or y, respectively).
- Otherwise, u has two predecessors u_1 and u_2.
  - If $u = (s, t, \cdot)$, then $u_1 = (s, a, \cdot)$ and $u_2 = (a, t, \cdot)$, for some $a \in V(H)$. Return $\{\texttt{search}(u_1), \texttt{search}(u_2)\}$.
  - Otherwise, $u = (v_1, v_2, \cdot)$, where either $v_1 = s$ or $v_2 = t$ (but not both), and $u_1 = (v_1, a, \cdot)$ and $u_2 = (a, v_2, \cdot)$ for some $a \in V(H)$. If $v_1 = s$, return $\texttt{search}(u_1)$, and if $v_2 = t$, return $\texttt{search}(u_2)$.

The correctness follows by induction on the level of u. If u is at level 1, $\texttt{search}(u)$ is trivially correct, and if u has exactly one predecessor p, then $\texttt{search}(u) = \texttt{search}(p)$, which is correct by induction. Otherwise, suppose u has two predecessors u_1 and u_2. If $u = (s, t, \cdot)$, then by the inductive hypothesis, $\texttt{search}(u_1)$ is the second node on the path represented by u_1 and $\texttt{search}(u_2)$ is the penultimate node on the path represented by u_2; since the path represented by u is the union of the paths represented by u_1 and u_2, it then follows that $\texttt{search}(u)$ is also correct. The correctness of $\texttt{search}(u)$ when $u = (v_1, v_2, \cdot)$, with $v_1 = s$ or $v_2 = t$, follows similarly.

The depth of the algorithm is bounded by the depth of $\mathcal{D}$, which is ℓ. Similarly, for work, the work to compute any *one* $\texttt{search}(u)$ is at most $O(\ell)$, as $\mathcal{D}$ has ℓ layers. So, the total work is $O(\zeta\ell) = O(\eta)$, where ζ is the number of nodes at level ℓ, and $\zeta\ell = O(\eta)$ by property 4 of Lemma 8.1. The desired values $r_{x,y}$ can then be computed by, for each u at level ℓ, adding the flow value of u to $r_{\texttt{search}(u)}$ (which is initially set to 0), which takes $O(\eta)$ work $O(\log \eta)$ depth by parallel summation. ∎

9 Applications.

In this section, we discuss some notable applications of our parallel approximate max-flow algorithm. Namely, we show that our aforementioned result implies new or substantially improved parallel algorithms for (balanced) sparsest cuts, minimum-cost hierarchical clustering [20], fair-cuts [51] and approximate Gomory-Hu trees. The input instance to all these aforementioned problems are undirected, weighted graphs $G = (V, E, c)$ with positive edge-weights that are assumed to be polynomially-bounded (or alternatively, a polynomially bounded ratio of maximum to minimum edge weights).

9.1 Sparsest Cut and Balanced Min-cut. The sparsest cut problem is a classic problem in graph theory that informally asks to partition a given graph while removing as little edge-mass as possible. More precisely, given an undirected, weighted graph $G = (V, E, c)$ the objective is to find a cut $(X, V \setminus X)$ of minimum *sparsity*, which is formally defined as

$$\phi(X) = \frac{c(E(X, V \setminus X))}{\min\{|X|, |V \setminus X|\}},$$

where $c(E(X, V \setminus X))$ is the total weight of edges going across the cut $(X, V \setminus X)$. A closely related problem, the β-balanced minimum cut asks for a cut $(X, V \setminus X)$ such that the smaller side of the partition has at least βn vertices for a given parameter $\beta > 0$, and the total edge weight $c(E(X, V \setminus X))$ is minimized among all such partitions. The β-balanced sparsest cut is similarly defined as a cut whose i). smaller side has at least βn vertices and ii). sparsity is minimized among all such partitions.

All the aforementioned problems are known to be NP-hard, and the best-known polynomial-time algorithms achieve $O(\sqrt{\log n})$-approximation [6] to their corresponding objective, albeit at the expense of a large (sequential) polynomial running time. For the balance constrained variants of these cut problems, bicriteria approximations that allow multiplicative factors on both the balance parameter as well as the cut size are also commonly studied. This notion can be formally defined as follows.

DEFINITION 9.1. (BICRITERIA APPROXIMATION FOR BALANCED CUT PROBLEMS) *Let $\beta' < \beta \leq \frac{1}{2}$ be real numbers, and let $(X^*, V \setminus X^*)$ be an optimal β-balanced min-cut (resp. β-balanced sparsest cut of G). We say that a cut $(X, V \setminus X)$ is an (α, β')-bicriteria approximation of the β-balanced min-cut (resp. β-balanced sparsest cut) if*

1. *$(X, V \setminus X)$ is a β'-balanced cut, and;*
2. *Approximation guarantees:*
   (a) *For β-balanced sparsest cut, $\phi(X) \leq \alpha \cdot \phi(X^*)$.*
   (b) *For β-balanced min-cut, $c(E(X, V \setminus X)) \leq \alpha \cdot c(E(X^*, V \setminus X^*))$.*

While algorithms for sparsest and balanced-min cuts have been developed for the distributed CONGEST model of computation [47, 11], no parallel PRAM algorithms with nearly-linear work and polylogarithmic depth are known. We resolve this state-of-the-affair by designing the first PRAM $(\operatorname{polylog} n, \operatorname{polylog} n)$-bicriteria approximation for both problems using our parallel approximate max-flow algorithm. Formally,

THEOREM 9.1. *There is a randomized PRAM algorithm that given an undirected weighted graph $G = (V, E, c)$, computes with high probability a cut $(X, V \setminus X)$ that achieves*

- *An $O(\log^3 n)$-approximation for sparsest cut;*
- *An $\left(O\left(\log^3 n\right), O\left(\frac{\beta}{\log^2 n}\right)\right)$-bicriteria approximation for β-balanced sparsest cut;*
- *An $\left(O\left(\frac{\log^3 n}{\beta}\right), O\left(\frac{\beta}{\log^2 n}\right)\right)$-bicriteria approximation for β-balanced min-cut.*

This algorithm has $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth.

Our algorithm builds upon the cut-matching game framework developed by [45] that effectively reduces the computation of all of these aforementioned problems to polylogarithmically many single-commodity max-flow computations (as well as a flow-decomposition of the corresponding max-flow solutions). As a consequence, this framework provides algorithms for computing polylog n approximations to all these cut problems with work and depth that matching that of the max-flow (and flow-decomposition) oracle utilized in their implementations up to a polylog n factor. While the original idea of [45] required an *exact* max-flow oracle in its implementation, [55] showed in a fairly straightforward extension that an approximate max-flow oracle also suffices to achieve morally the same result.

Our algorithm, formally described in Algorithm 9.1, is a straightforward extension of this aforementioned result of [55] to the PRAM setting. By crucially utilizing our parallel approximate max-flow result in Theorem 1.1, the parallel flow-decomposition result in Lemma 8.1, and the near-linear work, logarithmic depth flow-rounding algorithm of [16], this algorithm achieves guarantees formally described in Lemma 9.1, which in turn imply the guarantees claimed in Theorem 9.1.

ALGORITHM 9.1. `Parallel cut-matching game for sparsest cuts`(G, α, β)
Input: Graph $G = (V, E, c)$ with $n = |V|, m = |E|$, a sparsity parameter α, a balance parameter β.
Output:
Either Cut Case: a cut $(X, V \setminus X)$ such that $\phi(X) \leq \alpha$ and $\beta n \leq |X| \leq \frac{n}{2}$;
Or Expander Case: a graph H embeddable in G with congestion at most $O(\log^3 n/\alpha)$ such that every $(\beta \log^2 n)$-balanced cut in H has sparsity at least $\Omega(1)$.
Procedure:

for round $t = 1, \ldots, c_1 \log^2 n$, where c_1 is a sufficiently large constant **do**
  // The cut player:
  Sample a random n-dimensional unit vector $\boldsymbol{r}$ orthogonal to $\mathbf{1}$.
  if $t > 1$ **then**
    $\boldsymbol{u} \leftarrow \mathsf{M}_{t-1}(\mathsf{M}_t(\ldots(\mathsf{M}_1 \boldsymbol{r})))$, where for any $t' < t$, the $(n \times n)$ (sparse) matrix $\mathsf{M}_{t'}$ is the probability transition matrix corresponding to perfect matching output by the matching player in round t'.
  else
    $\boldsymbol{u} \leftarrow \boldsymbol{r}$
  Return cut $X_t \leftarrow$ vertices corresponding to the smallest $n/2$ entries in $\boldsymbol{u}$.
  // Matching player:
  Fix $\varepsilon = 1/10$, and let $c_2 = c_2(\varepsilon)$ be a sufficiently large constant.
  for sub-round $j = 1, \ldots, c_2 \log n$ **do**
    Maintain Sl_t^j and Sr_t^j: at the beginning, let $Sl_t^1 = X_t$; $Sl_t^j = Sl_t^j \setminus V(\mathsf{M}_t^{j-1})$, where $V(\mathsf{M}_t^{j-1})$ is the matching computed in the $(j-1)$-th sub-round of the matching player. Update Sr_t^j analogously.
    // Note that the sizes of Sl_t^j and Sr_t^j always remain equal by construction across all sub-rounds.
    Connect a source s to Sl_t^j and a sink t to Sr_t^j with edge capacities 1, and scale all other edge capacities in G by α^{-1}.
    Compute a $(1-\varepsilon)$-approximate max-flow, and round the flow to be integral using the algorithm in [16]; also obtain the corresponding approximate min-cut $(C_t^j, V \setminus C_t^j)$.
    // by the dual variables of Sherman's framework.
    if $c(E(C_t^j, V \setminus C_t^j)) < \left|Sl_t^j\right| - \beta n$ **then**
      Output $(C_t^j, V \setminus C_t^j)$ as the desired α-sparse cut and terminate the game.
    else
      Compute a *partial* matching M_t^j between the vertex sets Sl_t^j, Sr_t^j given by the flow decomposition.
  Let $Sl_t^{J+1} = Sl_t^J \setminus V(\mathsf{M}_t^J)$, and $Sr_t^{J+1} = Sr_t^J \setminus V(\mathsf{M}_t^J)$, where $J = c_2 \log n$ is the final sub-round of the matching player.

22: **if** $\left|Sl_t^{J+1}\right| = \left|Sr_t^{J+1}\right| < \beta n$ **then**

23: Compute an arbitrary matching M'_t between Sl_t^{J+1}, Sr_t^{J+1}.

24: Return the perfect matching $\mathsf{M}_t = \left(\bigcup \mathsf{M}_t^j\right) \cup \mathsf{M}'_t$ to be the union of the partial matchings across all sub-rounds of the matching player.

25: If the matching player outputs a cut in any (sub)round, then it is the desired α-sparse cut; otherwise, $H = \bigcup \left(\mathsf{M}_t \setminus \mathsf{M}'_t\right)$ is the graph embeddable in G with low congestion.

LEMMA 9.1. (PARALLEL VERSION OF [55] LEMMA B.18, CF. [45]) *There is a randomized* PRAM *algorithm that given an undirected, weighted graph $G = (V, E, c)$, a sparsity parameter $\alpha > 0$, and a balance parameter $\beta = O(\frac{1}{\log^2 n})$, with high probability computes*

- *either an α-sparse cut X such that $\beta n \le |X| \le \frac{n}{2}$;*
- *or a graph H embeddable in G with congestion at most $O(\log^3/\alpha)$ such that every $(\beta \log^2 n)$-balanced cut in H has sparsity at least $\Omega(1)$.*

The algorithm has $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth.

Proof. These aforementioned guarantees are achieved by Algorithm 9.1, and its correctness follows directly from [55] with no change to the proofs. Specifically the correctness for an α-approximate (unconstrained) sparsest cut follows from Lemma B.16 and Corollary B.21, and for an α-approximate, β-balanced sparsest cut follows from Lemma B.18 and Corollary B.22.

We now show that this algorithm admits a $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth PRAM implementation. First, observe that in the t^{th} round of the cut-matching game, the cut player's strategy can implemented in $O(mt)$ work and $O(t \log n)$ depth as described in Lemma B.1. Now consider the matching player's strategy in the t^{th} round of the cut-matching game, which consists of $O(\log n)$ sub-rounds. We claim that each sub-round j of the matching player requires $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth. Observe that maintaining Sl and Sr and scaling edge weights can be done in $O(m)$ work and $O(1)$ depth. Since we pick $\varepsilon = O(1)$, the $(1-\varepsilon)$ max-flow algorithm runs in $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth as in Corollary 1.1. Rounding the flow to an integral solution can be done in $O(m)$ work and $O(\log n)$ depth by [16]. Finally, since the edge capacities are bounded polynomials, we can use our parallel flow decomposition algorithm in Lemma 8.1 with $\delta = \frac{\varepsilon}{\|c\|_1}$. By Lemma 8.1, this takes $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth, and it preserves flow decomposition for a $(1-\varepsilon)$ approximate max-flow. Since there are at most $O(\log n)$ sub-rounds of the matching player in any (outer) round of our cut-matching game, the cut-player's strategy in the t^{th} round can also be implemented in $O(m \,\text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth. Finally, since the number of rounds t of the cut-matching game is bounded by $O(\log^2 n)$, the total work and depth of our algorithm is bounded by $O(m \,\text{polylog}\, n)$ and $O(\text{polylog}\, n)$, respectively. ∎

The algorithm and analysis of Lemma 9.1. The algorithm follows from polylogarithmically-many parallel applications of Lemma 9.1. Concretely, when setting $\beta' = \beta/(10c \log^2 n)$ (or $\beta' = 0$ for the sparsest cut), we can guess the value of α as $(c_{\min}/n) \cdot 2^i$ for integer i and return the smallest guess value that returns a sparse cut. Any graph G whose eligible cuts have sparsity less than $\alpha/\log^3 n$ will not be embeddable for H, which gives us a cut whose sparsity is at least an $O(\log^3 n)$ approximation. Finally, note that by the balance of partition, an α-approximation of the β-balanced sparsest is always an $O(\alpha/\beta)$-approximation for the β-balanced min-cut.

To analyze the efficiency, under the assumption that the ratio between the maximum and the minimum capacities is polynomial, the geometrically-increasing guess can be done in parallel with an $O(\log n)$ multiplicative factor of work and $O(1)$ depth overhead. Each call to the algorithm of Lemma 9.1 takes $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth, and so the complete algorithm has total work $O(m \cdot \text{polylog}\, n)$ and depth $O(\text{polylog}\, n)$.

9.2 Minimum Cost Hierarchical Clustering. Hierarchical clustering is a fundamental data analysis tool used to organize data into a dendogram. Given data represented as an undirected weighted graph $G = (V, E, c)$, where the vertices represent datapoints and (positive) edge weights represent similarities between their corresponding end points, the goal is to build a hierarchy, represented as a rooted tree $\mathcal{T}$; the leaves of this tree correspond to individual datapoints (vertices V), and the internal nodes correspond to a *cluster* $S \subseteq V$ consisting of their descendent leaves. Intuitively, this tree can be viewed as clustering the vertices of G at multiple levels of granularity simultaneously, with the clustering becoming increasingly fine-grained at deeper levels. [20] initiated the study of this problem from an *optimization* perspective by proposing the following cost function for similarity-based hierarchical clustering-

$$\text{cost}_G(\mathcal{T}) = \sum_{(u,v)\in E} c_{uv} \cdot |\text{leaves}(\mathcal{T}_{uv})|, \tag{9.2}$$

where c_{uv} is the weight (similarity) of the edge $(u, v) \in E$ and $\mathcal{T}_{uv}$ is the subtree of $\mathcal{T}$ rooted at the least common ancestor of u and v, and $|\text{leaves}(\mathcal{T}_{uv})|$ is the number of descendent leaves in this subtree $\mathcal{T}_{uv}$. Intuitively, this objective imposes a large penalty for separating "similar" vertices at higher levels in the tree, thereby placing "similar" vertices closer together.

This minimization problem was shown to be NP-hard, and moreover, no polynomial time constant factor approximation factor was shown to be possible for this objective assuming the *small-set expansion* hypothesis [12]. Nevertheless, this objective is considered a reasonable one; [20] and a follow-up work by [17] show that it satisfies several properties desired of a "good" hierarchical clustering. As a consequence, this objective has been well studied in the literature (cf. [20, 12, 63, 13, 17, 7, 1], and references therein), where it was shown to have strong algorithmic connections to the sparsest cut-problem. Specifically, [20, 12, 17] show that recursively partitioning the subgraph induced at each internal node using any α-approximate sparsest cut oracle results in a cluster tree that is a $O(\alpha)$ approximation for Dasgupta's objective.

While our results do imply a near-linear work, polylogarithmic depth parallel algorithm for computing polylog-approximate sparsest cuts, this by itself does not suffice to give a low-depth, work-efficient parallel algorithm for (approximately minimum-cost) hierarchical clustering. The reason is precisely that the sparsest cut subroutine may produce highly imbalanced partitions, resulting in a cluster tree with super-logarithmic depth. Due to the dependent nature of the recursively generated subgraphs on which the sparsest cut oracle is invoked, the resulting clustering algorithm would have not just super-logarithmic depth, but also super-linear work.

Fortunately, the solution is relatively simple, which is to recursively partition the graph using balanced min-cuts (with $\beta = \Omega(1)$) instead, guaranteeing that the resulting hierarchy would have logarithmic depth. When utilizing such a balanced min-cut subroutine, [12, 7] show morally the same guarantees for the resultant hierarchical clustering as the unbalanced case. Specifically, they show that recursively partitioning the subgraph induced at each internal node using any (α, β')-bicriteria approximation oracle for β-balanced min-cuts ($0 < \beta' \leq \beta \leq 1/2$) results in a hierarchy that is an $O(\alpha/\beta')$ approximation for Dasgupta's cost function.

This result in combination with our parallel $O(\beta^{-1}\log^3 n, \beta\log^{-2} n)$-bicriteria approximation algorithm for β-balanced min-cuts presented in Theorem 9.1 directly gives us the first PRAM algorithm for minimum-cost hierarchical clustering. Formally,

THEOREM 9.2. *There is a randomized* PRAM *algorithm that given an undirected weighted graph $G = (V, E, c)$, computes with high probability, a hierarchical clustering tree $\mathcal{T}$ that is a $O(\log^5 n)$-approximation for Dasgupta's objective (Eq (9.2)). This algorithm has $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth.*

Combined with the simulation result of [40, 35], the above result implies the first *fully-scalable* MPC algorithm for this problem; our algorithm computes a $O(\log^5 n)$-approximate minimum cost hierarchical clustering in $O(\text{polylog}\, n)$ rounds, where each machine has local-memory $O(n^\delta)$ for any constant $\delta > 0$, and the total memory is $O(m\,\text{polylog}\, n)$. Prior to this work, the state-of-the-art MPC algorithm for this problem required $\Omega(n\ \text{polylog}\, n)$ local (per-machine) memory [1].

9.3 Fair Cuts and Approximate Gomory-Hu Trees. In a recent work, [51] introduced the notion of fair cuts and showed that it is useful in several applications, one of them being the constructing approximate Gomory-Hu trees. Formally, a fair cut is defined as follows.

DEFINITION 9.2. (FAIR CUT [51]) *Let $G = (V, E, c)$ be an undirected graph with edge capacities $c \in R^E_{>0}$. For any two vertices s and t, and parameter $\alpha \geq 1$, we say that a cut (S, T) is a α-fair (s, t)-cut if there exists a feasible (s, t)-flow such that $f(u, v) \geq \frac{1}{\alpha} \cdot c(e)$ for every $e \in E(S, T)$.*

Note that an α-fair (s, t)-cut is also an α-approximate (s, t)-min-cut but not vice-versa. We show that the results presented in our paper also extend to this stronger notion of approximate min-cuts, improving upon the algorithmic result of [51]. In particular, [51] give a PRAM algorithm for computing a $(1 + \varepsilon)$-fair cut with $n^{o(1)}/\text{poly}(\varepsilon)$ depth and $m^{1+o(1)}/\text{poly}(\varepsilon)$ work. Importantly, the computational bottleneck in their approach is the construction and resultant quality of a congestion approximator for the input graph, in the sense that their algorithm has depth and work $\text{poly}(\alpha, \varepsilon^{-1}, \log n)$, and m $\text{poly}(\alpha, \varepsilon^{-1}, \log n)$, respectively, plus the depth and work required to construct such an α-congestion approximator. The same bottleneck persists in the application of their fair cut idea for computing isolating cuts and approximate Gomory-Hu trees. In their work, [51] utilize a *boundary-linked expander decomposition* of [11, 36] as their congestion approximator, which requires $n^{o(1)}$ depth and $m^{1+o(1)}$ work to construct, producing a $n^{o(1)}$-congestion approximator which gives them their PRAM result[16]. Our new polylogarithmic depth, near-linear work construction of congestion approximators from Theorem 2.1 when used as a blackbox in the algorithm of [51] immediately imply the following improvements.

THEOREM 9.3. *There is a randomized* PRAM *algorithm that given as input an undirected, weighted graph $G = (V, E, c)$, vertices $s, t \in V$, and desired precision $\varepsilon > 0$, with high probability, computes a $(1+\varepsilon)$-fair (s, t)-cut in $O(m\ \text{poly}(\varepsilon^{-1}, \log n))$ work and $O(\text{poly}(\varepsilon^{-1}, \log n))$ depth.*

THEOREM 9.4. *There is a randomized* PRAM *algorithm that given as input an undirected, weighted graph $G = (V, E, c)$, and desired precision $\varepsilon > 0$, computes with high probability a $(1+\varepsilon)$-approximate Gomory-Hu tree in $O(m\ \text{poly}(\varepsilon^{-1}, \log n))$ work and $O(\text{poly}(\varepsilon^{-1}, \log n))$ depth.*

References

[1] Arpit Agarwal, Sanjeev Khanna, Huan Li, and Prathamesh Patil. Sublinear algorithms for hierarchical clustering. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2022 (NeuIPS 2022)*, 2022.

[2] Ravindra K Ahuja and James B Orlin. A capacity scaling algorithm for the constrained maximum flow problem. *Networks*, 25(2):89–98, 1995.

[3] S.G. Akl. *Parallel Computation: Models and Methods*. Prentice Hall, 1997.

[4] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*, page 574–583, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Parallel approximate undirected shortest paths via low hop emulators. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 322–335, 2020.

[6] Sanjeev Arora, Satish Rao, and Umesh V. Vazirani. Expander flows, geometric embeddings and graph partitioning. *J. ACM*, 56(2):5:1–5:37, 2009.

[7] Sepehr Assadi, Vaggos Chatziafratis, Jakub Lacki, Vahab Mirrokni, and Chen Wang. Hierarchical clustering in graph streams: Single-pass algorithms and space lower bounds. In Po-Ling Loh and Maxim Raginsky, editors, *Conference on Learning Theory, 2-5 July 2022, London, UK*, volume 178 of *Proceedings of Machine Learning Research*, pages 4643–4702. PMLR, 2022.

[16] [51] only show their results for unweighted graphs since their parallel construction of congestion approximators are only for unweighted graphs; though they mention in their paper that they believe known techniques imply their same results for weighted graphs. We give an explicit construction even for the weighted case.

[8] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6), oct 2017.

[9] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

[10] Bala G Chandran and Dorit S Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations research*, 57(2):358–376, 2009.

[11] Yi-Jun Chang and Thatchaphol Saranurak. Improved distributed expander decomposition and nearly optimal triangle enumeration. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 66–73. ACM, 2019.

[12] Moses Charikar and Vaggos Chatziafratis. Approximate hierarchical clustering via sparsest cut and spreading metrics. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 841–854. SIAM, 2017.

[13] Vaggos Chatziafratis, Rad Niazadeh, and Moses Charikar. Hierarchical clustering with structural constraints. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 773–782. PMLR, 2018.

[14] Li Chen, Rasmus Kyng, Yang P Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 612–623. IEEE, 2022.

[15] Paul Christiano, Jonathan A Kelner, Aleksander Madry, Daniel A Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 273–282, 2011.

[16] Edith Cohen. Approximate max-flow on small depth networks. *SIAM Journal on Computing*, 24(3):579–597, 1995.

[17] Vincent Cohen-Addad, Varun Kanade, Frederik Mallmann-Trenn, and Claire Mathieu. Hierarchical clustering: Objective functions and algorithms. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 378–397. SIAM, 2018.

[18] Samuel I Daitch and Daniel A Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 451–460, 2008.

[19] George B Dantzig. Application of the simplex method to a transportation problem. *Activity analysis and production and allocation*, 1951.

[20] Sanjoy Dasgupta. A cost function for similarity-based hierarchical clustering. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 118–127. Association for Computing Machinery, 2016.

[21] EA Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation, soviet math. doll. 11 (5), 1277-1280,(1970). *English translation by RF. Rinehart*, 1970.

[22] EA Dinic. Metod porazryadnogo sokrashcheniya nevyazok i transportnye zadachi (excess scaling and transportation problems). *Issledovaniya po Diskretnoi Matematike*, 1973.

[23] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[24] Shimon Even and R Endre Tarjan. Network flow and testing graph connectivity. *SIAM journal on computing*, 4(4):507–518, 1975.

[25] Lester Randolph Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956.

[26] Wai-Shing Fung, Ramesh Hariharan, Nicholas J. A. Harvey, and Debmalya Panigrahi. A general framework for graph sparsification. *SIAM Journal on Computing*, 48(4):1196–1223, 2019.

[27] Sebastian Forster, Gramoz Goranci, Yang P. Liu, Richard Peng, Xiaorui Sun, and Mingquan Ye. Minor sparsifiers and the distributed laplacian paradigm. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 989–999, 2022.

[28] Harold N Garbow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*,

31(2):148–168, 1985.

[29] Barbara Geissmann and Lukas Gianinazzi. Parallel minimum cuts in near-linear work and low depth. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 1–11, 2018.

[30] Andrew V. Goldberg. The partial augment–relabel algorithm for the maximum flow problem. In *Algorithms - ESA 2008*, pages 466–477. Springer Berlin Heidelberg, 2008.

[31] Andrew V Goldberg, Sagi Hed, Haim Kaplan, Pushmeet Kohli, Robert E Tarjan, and Renato F Werneck. Faster and more dynamic maximum flow by incremental breadth-first search. In *Algorithms-ESA 2015: 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 619–630. Springer, 2015.

[32] Andrew V Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM (JACM)*, 45(5):783–797, 1998.

[33] Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[34] Donald Goldfarb and Michael D Grigoriadis. A computational comparison of the dinic and network simplex methods for maximum flow. *Annals of Operations Research*, 13(1):81–123, 1988.

[35] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, volume 7074 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2011.

[36] Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2212–2228. SIAM, 2021.

[37] Dorit S Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations research*, 56(4):992–1009, 2008.

[38] David R Karger. Minimum cuts in near-linear time. *Journal of the ACM (JACM)*, 47(1):46–76, 2000.

[39] David R Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4):601–640, 1996.

[40] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 938–948. SIAM, 2010.

[41] Alexander V Karzanov. On finding maximum flows in networks with special structure and some applications. *Matematicheskie Voprosy Upravleniya Proizvodstvom*, 5:81–94, 1973.

[42] Tarun Kathuria, Yang P. Liu, and Aaron Sidford. Unit capacity maxflow in almost $O(m^{4/3})$ time. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 119–130, 2020.

[43] Jonathan A Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 217–226. SIAM, 2014.

[44] Jonathan A Kelner, Gary L Miller, and Richard Peng. Faster approximate multicommodity flow using quadratically coupled flows. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1–18, 2012.

[45] Rohit Khandekar, Satish Rao, and Umesh V. Vazirani. Graph partitioning using single commodity flows. In Jon M. Kleinberg, editor, *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*, pages 385–390. ACM, 2006.

[46] Ioannis Koutis, Gary L. Miller, and Richard Peng. Approaching optimality for solving SDD linear systems. *SIAM J. Comput.*, 43(1):337–354, 2014.

[47] Fabian Kuhn and Anisur Rahaman Molla. Distributed sparse cut approximation. In Emmanuelle Anceaume, Christian Cachin, and Maria Gradinariu Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, volume 46 of *LIPIcs*, pages 10:1–10:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[48] Yin Tat Lee, Satish Rao, and Nikhil Srivastava. A new approach to computing maximum flows using electrical flows. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 755–764, 2013.

[49] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in o (vrank) iterations and faster algorithms for maximum flow. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 424–433. IEEE, 2014.

[50] Jason Li. Faster parallel algorithm for approximate shortest path. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 308–321, 2020.

[51] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, and Thatchaphol Saranurak. Near-linear time approximations for cut problems via fair cuts. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 240–275. SIAM, 2023.

[52] Yang P Liu and Aaron Sidford. Faster energy maximization for faster maximum flow. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 803–814, 2020.

[53] Aleksander Madry. Fast approximation algorithms for cut-based problems in undirected graphs. In *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, page 245–254. IEEE Computer Society, 2010.

[54] Aleksander Madry. Computing maximum flow with augmenting electrical flows. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 593–602. IEEE, 2016.

[55] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and $O(n^{1/2-\varepsilon})$-time. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1122–1129. ACM, 2017.

[56] James B. Orlin and Xiao-Yue Gong. A fast maximum flow algorithm. *Networks*, 77(2):287–321, 2021.

[57] Richard Peng. Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1862–1867. SIAM, 2016.

[58] Richard Peng and Daniel A Spielman. An efficient parallel solver for sdd linear systems. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 333–342, 2014.

[59] Yossi Peretz and Yigal Fischler. A fast parallel max-flow algorithm. *Journal of Parallel and Distributed Computing*, 169:226–241, 2022.

[60] Seth Pettie and Vijaya Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM Journal on Computing*, 31(6):1879–1895, 2002.

[61] Harald Räcke, Chintan Shah, and Hanjo Täubig. Computing cut-based hierarchical decompositions in almost linear time. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 227–238. SIAM, 2014.

[62] Vijaya Ramachandran et al. Parallel algorithms for shared-memory machines. In *Algorithms and Complexity*, pages 869–941. Elsevier, 1990.

[63] Aurko Roy and Sebastian Pokutta. Hierarchical clustering via spreading metrics. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2316–2324, 2016.

[64] Václav Rozhoň, Christoph Grunau, Bernhard Haeupler, Goran Zuzic, and Jason Li. Undirected $(1+\varepsilon)$-shortest paths via minor-aggregates: near-optimal deterministic parallel and distributed algorithms. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 478–487, 2022.

[65] Maria Serna and Paul Spirakis. Tight rnc approximations to max flow. In *STACS 91: 8th Annual Symposium on Theoretical Aspects of Computer Science Hamburg, Germany, February 14–16, 1991 Proceedings 8*, pages 118–126. Springer, 1991.

[66] Jonah Sherman. Nearly maximum flows in nearly linear time. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 263–269. IEEE, 2013.

[67] Jonah Sherman. Area-convexity, ℓ_∞ regularization, and undirected multicommodity flow. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 452–460, 2017.

[68] Jonah Sherman. Generalized preconditioning and undirected minimum-cost flow. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 772–780. SIAM, 2017.

[69] Yossi Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. Technical report, Computer Science Department, Technion, 1980.

[70] Yossi Shiloach and Uzi Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

[71] Daniel A. Spielman and Shang-Hua Teng. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM J. Matrix Anal. Appl.*, 35(3):835–885, 2014.

[72] Goran Zuzic, Gramoz Goranci, Mingquan Ye, Bernhard Haeupler, and Xiaorui Sun. Universally-optimal distributed shortest paths and transshipment via graph-based l1-oblivious routing. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2549–2579. SIAM, 2022.

A PRAM Primitives.

A.1 Eulerian Tours and Subtree Sum.

Consider a tree T, and for each undirected edge (u, v) create two anti-parallel arcs (u, v) and (v, u); call this new graph T'. Then, T' admits an Eulerian tour, which can also be viewed as DFS traversal of T. This Eulerian tour can be used to compute a number of properties of T efficiently in parallel, and in this section we describe several of use to us. Most of these results are folklore and can be found in several textbooks on parallel algorithms, such as [3].

THEOREM A.1. (EULERIAN TOUR [3]) *There is a $O(1)$ depth, $O(n)$ total work* PRAM *algorithm which outputs an ordered list of nodes corresponding to an Eulerian tour of a tree T with n nodes starting at a root r.*

Eulerian tours are usually computed via a linked list representation, where each directed arc points to the next element in the tour. This can also be converted to a random access ordered array, which is useful in several results below.

It is also known that one can compute all prefix sums of an array in $O(\log n)$ depth.

THEOREM A.2. (PREFIX SUMS [3]) *Given a list of values $a_1, \ldots, a_n$, there is an $O(\log n)$ depth, $O(n)$ work* PRAM *algorithm which outputs a new list $b_1, \ldots, b_n$ such that $b_k = \sum_{i \leq k} a_i$*

Using prefix sums and Eulerian tours, we can compute several useful properties of nodes in a tree.

THEOREM A.3. ([3]) *Let T be a tree rooted at r. Then, there is a* PRAM *algorithm which in $O(\log n)$ depth and $O(n)$ total work computes*

1. *The first and last time each node is visited by an Eulerian tour starting at r*
2. *The parent of each node*
3. *The depth of each node*

whenever there are at least $\Omega(n)$ processors.

Proof. Note that the length of the Eulerian tour array is always $O(n)$, as each original edge is traversed exactly twice and all trees over n nodes have $n-1$ edges. To find the first time each node is visited by a tour starting at r, assign one processor to each pair of consecutive elements in the list. Let u, v be these elements, appearing at indices $i-1, i$. The processor assigned to the pair then sets $b[(u, v)] = i$, where b is a new lookup table for the ordering of edges. To determine the first time a node u is visited by the tour, it suffices to compute $\min_v b[(v, u)]$, and similarly the last time can be determined by $\max_v b[(u, v)]$. These minimums and maximums can then be computed in depth $O(\log n)$ [3]. The parent of any node first visited at position i in the list is the $i-1$ element in the Eulerian tour list.

For the depth each node, we say (u, v) is a "forward" arc if u is the parent of v, and a "reverse" edge if v is the parent of u. In the ordered list of arcs traversed by the Eulerian tour, assign each forward arc a value of $+1$ and each reverse arc a value of -1. Then, run prefix sum. Any cycle in the tour must have zero sum, since it must use both copies of each undirected edge it visits, and so the prefix sum whenever the tour vists v is the depth of v. Using the previously computed first time each node is visited, we may in parallel output the depth of every node. ∎

One of the most useful results of this this section is that we can compute *sumtree sums*.

THEOREM A.4. (SUBTREE SUM) *Let T be a tree where each node v is associated with a weight w_v. Then, there is a* PRAM *algorithm with $O(\log n)$ depth and $O(n)$ total work that computes, for every node u, the sum of the w_v in the subtree rooted at u.*

Proof. As in the proof of Theorem A.3, classify each arc in the Eulerian tour of T as either forward or reverse. Set the weight of each reverse arc to be 0, and the weight of each forward arc (u, v) to w_v. Then, run prefix sum on the Eulerian tour list of arcs. At the last occurrence of u, all arcs in the subtree of u must have already been traversed (since the Eulerian tour uses each arc exactly once, and does not visit u again), so the prefix sums includes the sum of all weights in this subtree. Moreover, between the first and last visit to u, the tour cannot leave the subtree of u; if it did, then one of the arcs between u and its parent must have been used at least twice. So, the sum of the subtree is exactly $w_u + p[u^{(-1)}] - p[u^{(1)}]$, where p is the array of prefix sums and $u^{(-1)}, u^{(1)}$ are the indices of the last and first occurrences of u in the tour, respectively. The prefix sum along with first and last occurrences of each node can be computed in $O(\log n)$ depth and $O(n)$ work, and the final sum requires only $O(1)$ depth and $O(n)$ work. ∎

For example, subtree sums can be used to find the LCA of two nodes in a tree, which we use in Section D.

THEOREM A.5. *Let T be a tree rooted at r. Then, for any $u, v \in T$, there is an $O(\log n)$ depth, $O(n)$ work* PRAM *algorithm to find the least-common ancestor of u and v.*

Proof. The algorithm first sets weights of 0 on all nodes besides u and v, and sets weight 1 on u and v before computing subtree sums. Using the algorithm of Theorem A.3, also compute the depth of each node. Then, output the lowest-depth (i.e. furthest from the root) node who has subtree sum 2.

Computing the subtree sum and depth of each node can be done in $O(\log n)$ work and $O(n)$ work. Identifying all nodes with sum 2 can be done in $O(1)$ depth and $O(n)$ work, and taking the max depth of these can be done in $O(\log n)$ depth and $O(n)$ work, so the entire algorithm has the desired runtime.

For correctness, the algorithm computes the lowest-depth node whose subtree contains both u and v, which is exactly the LCA of u and v. ∎

A.2 Tree Separators. As also defined in Section 5.2, a tree separator node of a tree is a node whose removal results is a forest with components of at most half the size of the original tree.

DEFINITION A.1. (TREE SEPARATOR NODE, DEFINITION 5.3) *A node q in a tree T is called a tree separator node of T if the forest induced by $T \setminus \{q\}$ consists of trees with at most $|T|/2$ nodes.*

A very useful folklore results shows that every tree contains at least one tree separator node.

LEMMA A.1. (FOLKLORE) *For all trees T, there exists a node $q \in T$ such that q is a tree separator node of T.*

Proof. Suppose for contradiction T does not contain a tree separator node. Define M_u for each node $u \in T$ to be the size of the largest component of $T \setminus \{u\}$, and let $u' = \arg\min_{u \in T} M_u$. By assumption, $M_{u'} > |T|/2$, and let C' be the component of $T \setminus \{u'\}$ with size $M_{u'}$. Since T is a tree, there is a unique neighbor of u' in C; call this node v. $|C'| > |T|/2$, so $|T \setminus C'| \leq |C'| + 1$ and $|T \setminus (C' \cup \{u'\})| \leq |C'|$. Thus, $C' \setminus \{v\}$ must be the largest component in $T \setminus \{v\}$, as the sum of all other components can be at most $|C'| - 1$. But then we have $M_v < M_{u'}$, contradicting that $M_{u'}$ is minimal. ∎

We now give an algorithm to find a tree separator node with logarithmic depth and linear work, and uses the subtree sum algorithm of Theorem A.4.

LEMMA A.2. *There is a $O(\log k)$ depth, $O(k)$ work* PRAM *algorithm to find a tree separator node of any tree with k nodes.*

Proof. Let T be a tree with k nodes, and arbitrarily root T. The algorithm first assigns every node weight 1, and uses the algorithm of Theorem A.4 to compute subtree sums. Let s_u be the value of this subtree sum at node u. Then, in parallel, for each $u \in T$, check if $s_v \leq k/2$ for all v children of u, and check that $k - s_u \leq k/2$. Output any such u as a tree separator node.

Removing any tree separator node, by definition, results in a forest with components of size at most $k/2$, which is the exact condition the algorithm checks. A tree separator node exists for all trees (Lemma A.1), so the algorithm always outputs a tree separator node. The depth of the algorithm comes from the fact that the subtree sums and examining all nodes can be done in $O(\log k)$ depth. For work, note that the value of every node u needs to be checked twice: once when determining if $k - s_u \leq k/2$, and once when its (unique) parent checks if $s_u \leq k/2$. So, combined with the $O(k)$ work for subtree sums, the total work is $O(k)$. ∎

B Parallel Implementation of the Cut-Matching Game of [61].

In this section, we discuss the implementation of `partition-A1` in Section 6.1, which in turn is built on top of the cut-matching game in [61]. In the remainder of this section, note that we always operate over the subdivision graph as defined in Definition 4.5, i.e., we have $V' = V \cup X_E$ where X_E is the set of split vertices of the edges in E, $E' = \bigcup_{e=(u,v)\in E} \{(u, x_e), (x_e, v)\}$, and $c'_{(u,x_e)} = c'_{(v,x_e)} = c_e$ for every edge $e \in E$. Moreover, for ease of exposition, whenever we refer to a set of edges Y, we in fact refer to the subdivision vertices X_Y corresponding to set Y. It is straightforward to see that constructing the subdivision graph only takes $O(m)$ work and $O(1)$ depth.

All the algorithms that are presented in this section are identical to that of [61]; we merely discuss them for the sake of completeness and showing that they admit a $O(m \operatorname{polylog} n)$ work and $O(\operatorname{polylog} n)$ depth PRAM implementation. This entire process requires a blackbox access to only two subroutines: a $(1-\varepsilon)$-approximate max-flow algorithm $\mathcal{F}_\varepsilon$ that also returns the $(1+\varepsilon)$-approximate minimum cut, and a $(1-\delta)$-approximate flow-path decomposition algorithm $\mathcal{D}_\delta$ (whose implementation is given in Section 8). We shall henceforth refer to the work and depth of these algorithms $\mathcal{A} \in \{\mathcal{F}_\varepsilon, \mathcal{D}_\delta\}$ on an m-edge graph as $T(\mathcal{A}, m)$, and $D(\mathcal{A}, m)$. To build intuition, we begin by discussing the implementation specifics for the uncapacitated case in Appendix B.1 through Appendix B.4, and extend our result to the general capacitated case in Appendix B.5.

Using a cut-matching game to find well-linked edges. Before discussing [61], we shall find it useful to first understand the cut-matching game framework of [45]. It is a technical tool connecting the problem of approximating sparsest cuts or alternatively, certifying expansion to max flows, and is vital in the aforementioned result of [61]. Conceptually, it is an alternating game between a cut-player and a matching-player, where the former produces a bisection of the vertices, and the latter responds by producing a perfect matching across this bisection that does not necessarily belong to the underlying graph, but can be embedded in it[17]. The game ends when either the cut player produces a bisection for which the matching player cannot find a perfect matching that can be embedded in it (i.e. a sparse cut has been found), or the union of the perfect matchings produced thus far form an expander (i.e. an expander can be embedded in the underlying graph, certifying its expansion). The objective of the cut player is to terminate this game as quickly as possible, whereas the matching player seeks to delay this. In their work, [45] showed that there is a near-linear time strategy for the cut-player that guarantees that this game terminates in $O(\log^2 n)$ rounds, regardless of the matching player's strategy, and it is easiest to understand when viewed as the following *multicommodity flow problem*: initially, every vertex starts off with a unit of its own unique commodity (which we can represent as a n-dimensional one-hot encoded vector, which we shall henceforth refer to as a *flow vector* held by the vertex), and the goal is to uniformly distribute these commodities across all vertices with low congestion. In each round of the game, every matched pair of vertices (according to the perfect matching produced by the matching player) mix (average) their currently held commodities with each other through the matching edge. Since the perfect matching could be embedded into the underlying graph, this operation is guaranteed to be feasible. Now supposing in some round, every vertex ended up with a (near) uniform spread over all unique commodities, then we are done; it implies that the union of the perfect matchings must induce an expander, which moreover can be embedded into the underlying graph. A good cut strategy therefore must find bisections that

[17] Roughly speaking, we say a matching can be embedded into the graph, if there exist flow paths connecting every pair of matched nodes such that the net flow does not exceed the capacity of any edge.

mix these commodities quickly, and the key to arguing its existence is a potential function that measures the total distance of each vertex's currently held commodities from uniform. In particular, one can show that there always exists a bipartition that guarantees a multiplicative $(1 - \Omega(1/\log n))$ factor reduction in this potential, regardless of the perfect matching produced by the matching-player, guaranteeing that the game terminates in just $O(\log^2 n)$ rounds. Moreover, this bisection can be found efficiently. While this sketches the *analysis*, this procedure would be far too inefficient to implement as it would require explicitly maintaining the vector of currently held commodities for every vertex (which would naively take $O(n^2)$ work). Fortunately, [45] showed that it suffices to consider a *sketch* that is a *random projection* of these flow vectors in every round and still obtain the same convergence guarantees with high probability. The matching players strategy is essentially a max-flow computation with the sources and sinks being the two sides of the bipartition. If the max flow value is $n/2$, then we can obtain a perfect matching via a flow-path decomposition. Otherwise, a perfect matching cannot be embedded into this cut.

The above cut-matching game provides the basis of the algorithm of [61] for finding well linked edges; intuitively, a set F of edges is well-linked if we can embed an *expander* between the subdivision vertices of these edges, and the cut-matching game is precisely the technical tool used to verify this. The algorithm of [61] starts off with a candidate set of edges F, and the cut-matching game is used to either certify well-linkedness of this set F, or to produce a witness of non-expansion (a highly congested cut), in which case F is updated to a new set F_{new} (intuitively, to the edges in the most congested cut). However, in the event that F is updated, the entire game would have to restart, which would be too expensive (unless the updated set is a constant factor smaller than the old candidate set, in which case it is okay because the number of restarts would be bounded by $O(\log n)$). The modified game of [61] circumvents this issue by *reusing* rounds of the old game in the event that the candidate set F does not change by much. This is achieved by *moving* flow vectors from the old candidate set F to the new set F_{new} along paths of constant congestion (found by the flow decomposition algorithm $\mathcal{D}$). However, not all edges in the new set F_{new} may end up receiving a flow vector from the old set F. Following the notation of [61], we shall therefore denote all candidate sets $F := A \uplus R$, where A refers to the set of *active* edges that have a flow vector, and R refers to the remaining edges in F. The exact details of this matching player are discussed in Appendix B.3. The cut-player is also slightly different from the one in [45], where instead of a strict bipartition, the player produces a disjoint edge sets corresponding to the "source" side and "sink" side satisfying certain necessary properties. The exact details of this cut player are discussed in Appendix B.2.

B.1 Computing the projections of flow vectors. We first discuss how to efficiently compute the *sketch*, i.e. random projection of current state of the flow vectors held by every active edge (which evolves starting with a one-hot encoded vector, mixing and moving in every round depending on the matching player's strategy in that round). We remind the reader that given a candidate set of active edges A, there are $|A|$ flow vectors, each in $\mathbb{R}^m$, and the naive way to average and move flow vector would take $O(|A| \cdot m) = O(m^2)$ work. However, we note that in the algorithmic process, the flow vectors are only used for cut player to find the sources and sinks, and the procedure is implemented by a *projection* onto a random unit vector which we show can be simulated in nearly-linear work. We give a subroutine for such an implementation.

LEMMA B.1. *Let $\boldsymbol{r}$ be any unit vector of dimension m, and let $\mathsf{M}_t = M_t M_{t-1} \cdots M_1$ be a $m \times m$ matrix such that each M_i is supported over at most m' non-zero coordinates, and let $\{\boldsymbol{v}_i\}_{i=1}^m$ be m standard basis vectors that represent indicators for each of the m unique commodities. Then, there exists a* PRAM *algorithm that computes $\mu_i^{(t)} = \boldsymbol{r}^T \mathsf{M}_t \boldsymbol{v}_i$ for every i, in $O(t \log n)$ depth and $O((m + m')t)$ work.*

Proof. We perform the multiplication from left to right: $\mu_i^{(t)} = ((\boldsymbol{r}^T M_t) M_{t-1}) M_{t-2} \ldots$, where each time we multiply a vector with a matrix of at most m' nonzeros, which takes $O(\log n)$ depth and $O(m')$ work. Summing over t iterations gives the desired result. ∎

As we will see shortly, the operations on the flow vectors in the t^{th} round of the cut-matching

game can all be simulated by matrix multiplication of $\boldsymbol{r}^T \mathsf{M}_t \boldsymbol{v}_i$ as prescribed in Lemma B.1. Moreover, in all applications where the above subroutine is invoked, we always have the sparsity property $m' = O(m \text{ polylog } n)$ for each matrix M_i since they correspond to fractional matrices that mix or move flows produced by the flow decomposition algorithm $\mathcal{D}$, and $t = O(\text{polylog } n)$ due to the convergence guarantees of the cut-matching game in [61]. We will refer to these matrices M_i's as mix-or-move matrices. We remark that the idea was implemented by both [45] and [61], albeit it was less explicitly stated.

B.2 Parallel implementation of the cut player. We now describe the parallel implementation of a single iteration of the cut player that achieves the guarantees in Lemma 3.3 in [61]. Recall that the cut player's strategy given a set of active edges A, is to find two different subsets $A_S \subseteq A$ and $A_T \subseteq A$ whose commodities are not *well-mixed*. This is done in the following way: for every edge $e \in A$, let $\mathbf{f}_e$ be the flow vector currently held by e. In round t of the game, this is precisely given by $\mathbf{f}_e = M_t M_{t-1} \dots M_1 \boldsymbol{v}_e$, where $\boldsymbol{v}_e$ is the indicator (one-hot encoded vector) of edge e's unique commodity, and each M_i represents the fractional mix-or-move matrix representing the matching player's strategy (i.e. mixing or moving) in round i. Let $\boldsymbol{\mu} := (1/|A|) \sum_{e \in A} \mathbf{f}_e$ represent the average flow vector in that round. Given these quantities (neither explicitly computed), and a random vector $\boldsymbol{r}$, for every edge $e \in A$, let $\mu_e = \langle \mathbf{f}_e , r \rangle$, and $\bar{\mu} := \langle \boldsymbol{\mu} , \boldsymbol{r} \rangle$ denote the projection of the flow vector of edge e, and the average flow vector onto the random vector, respectively. For any $B \subseteq A$, the potential P_B is defined as

$$P_B := \sum_{f \in B} (\mu_f - \bar{\mu})^2 .$$

Now given a set of active edges A, the cut player precisely seeks to identify a set of source edges A_S, a set of sink edges A_T, along with a value η such that (rf. Lemma 3.3 in [61])

1. η separates the sets, i.e. $\max_{e \in A_S} \mu_e \le \eta \le \min_{f \in A_T} \mu_f$ or $\min_{e \in A_S} \mu_e \ge \eta \ge \max_{f \in A_T} \mu_f$,
2. $|A_S| \le |A|/8$ and $|A_T| \ge |A|/2$
3. for every source edge $e \in A_S$, $|\mu_e - \eta|^2 \ge (1/9)|\mu_e - \bar{\mu}|^2$,
4. $\sum_{e \in A_S} |\mu_e - \bar{\mu}|^2 \ge (1/160) \sum_{e \in A} |\mu_e - \bar{\mu}|^2$,

Note that, unlike [45], A_S and A_T does not need to be an exact bi-partition of A. The following procedure describes implementation of the cut player's strategy that meets the above requirements in detail.

Subroutine `find-sources-and-sinks` [61]

Input: subdivision graph $G' = (V', E')$; a set of active edges $A \subseteq E$; a sequence of t mix-or-move matrices $M_t, \dots, M_1$ each of dimension $m \times m$ and supported over at most $O(m \text{ polylog } n)$ non-zero coordinates.

Output: a set of source edges $A_S \subset A$, a set of sink edges $A_T \subset A$, and a separation value η.

Procedure:

1. Sample a unit vector $\boldsymbol{r}$ uniformly at random, and project the flow vector of each $e \in A$ to $\boldsymbol{r}$ to get $\mu_e = \langle \mathsf{M}_t \boldsymbol{v}_e , \boldsymbol{r} \rangle$, where $\mathsf{M}_t = M_t M_{t-1} \dots M_1$. This step is executed by initializing length-m flow vectors with $\boldsymbol{v}_e$ the indicator vector of e, and then running the algorithm from Lemma B.1.
2. Assuming w.l.o.g that $|\{e \in A \mid \mu_e < \bar{\mu}\}| \le |\{e \in A \mid \mu_e \ge \bar{\mu}\}|$, pick $L = \{e \in A \mid \mu_e < \bar{\mu}\}$, and let $R = A \setminus L$.
3. Compute P_L and P_R. If $P_L \ge \frac{1}{20} P_A$, set A_S as the $|A|/8$ edges (or all edges in L if there are fewer than $|A|/8$ edges in L) with the smallest μ_e values from L, A_T as R, and η as $\bar{\mu}$.

4. Otherwise:
   (a) Let $\ell = \sum_{e \in L} |\mu_e - \bar{\mu}|$, and let $\eta = \bar{\mu} + 4\ell/|A|$.
   (b) Let A_T be the edges whose μ_e is at most η.
   (c) Construct new sets $R' = \{e \in A \mid \mu_e \geq \bar{\mu} + 6\ell/|A|\}$, and let A_S be the $|A|/8$ edges with the largest μ_e values from R'.

This subroutine is used repeatedly during the cut-matching game in the hierarchical decomposition procedure of [61]. We now show this subroutine can be implemented efficiently in the PRAM setting.

CLAIM B.1. *There is a* PRAM *implementation of the subroutine* `find-source-and-sinks` *that, given an arbitrary sequence of t matrices, each with support over $O(m \operatorname{polylog} n)$ nonzero entries, has $O(t \log n)$ depth and $O(tm \operatorname{polylog} n)$ work.*

Proof. By Lemma B.1, computing the projections μ_e's takes $O(\log n)$ depth and $O(m \operatorname{polylog} n)$ work since the support size of each mix-or-move matrix M_i is $O(m \operatorname{polylog} n)$. The steps that compute the average and the potential all take $O(\log n)$ depth and $O(m \operatorname{polylog} n)$ work. Finally, taking a subset from A (resp. L and R) by checking the μ values also takes $O(\log n)$ depth and $O(m)$ work. Summarizing the above steps gives the desired $O(\log n)$ depth and $O(m \operatorname{polylog} n)$ work. ∎

B.3 Parallel implementation of the matching player. We now discuss the parallel implementation of the matching player. The matching player takes as input the source edges A_S and the sink edges A_T computed by the cut player, and computes a partial fractional matching M between the subdivision nodes X_{A_S} and X_{A_T} in the subdivision graph G'. The matching player uses a $(1-\varepsilon)$-approximate max-flow algorithm $\mathcal{F}_\varepsilon$ [18] to ensure that the matching can be routed in G' with constant congestion, and uses the $(1-\delta)$-approximate flow-path decomposition algorithm $\mathcal{D}_\delta$ to construct a matching from the output of the flow algorithm. At this point, we remind the reader that the cut-matching game in [61] deviates from that of [45]. In [61], the matching player chooses to perform either a *matching* step or a *deletion* step and the choice is made by flipping a fair random coin. The matching step is similar to [45] where the flow vectors of the matched edges are *mixed* resulting in a reduction in potential. The deletion step deletes the flow vectors on some edges (potentially *moving* them to new edges) resulting in a new set of active edges which again leads to potential reduction. The following procedure describes the matching player's strategy from [61] in detail.

Subroutine `match-or-delete` [61]

Inputs: subdivision graph $G' = (V', E')$; a set of candidate edges $F \subseteq E$, active edges $A \subseteq E$, remaining edges $R \subseteq E$; source edges $A_S \subset A$; sink edges $A_T \subset A$; set of edges B.
Output: a $m \times m$ mix-or-move matrix M with support size at most $O(m \operatorname{polylog} n)$; new candidate edges F_{new}; new active edges A_{new}; new remaining edges R_{new}; set B_{new}.

Procedure:

1. Construct a capacitated graph G'_{st}: add a super-source s and connect s to all subdivision vertices $x \in X_{A_S}$ with capacity 1; add a super-sink t and connect t to all subdivision vertices $x \in X_{A_T}$ with capacity $\frac{1}{2}$; add capacity 2 to all edges in G' that are not incident on s or t.
2. Run the flow algorithm $\mathcal{F}_\varepsilon$ on G'_{st} with $\varepsilon = \frac{1}{3 \log^3 n}$ to obtain a flow f; let $C' \in E'$ be the set of

[18] We assume that $\mathcal{F}_\varepsilon$ is such that it also outputs an approximate min-cut. Note that Sherman's algorithm [66] satisfies this assumption.

edges that correspond to the approximate min *cut*, and let $C \in E$ be the corresponding set of cut edges in G.

3. Run the flow-decomposition algorithm $\mathcal{D}_\delta$ on f with $\delta = \frac{1}{3\log^3 n}$, and let G be the data-structure encoding the flow paths (see Lemma 8.1). For every edge (s,x) where $x \in X_{A_S}$ with flow $f_{(s,x)} \geq 1/2$, rescale the flow such that $f_{(s,x)} = 1$. For every flow path p that uses edge (s,x), rescale the flow on p by $1/f_{(s,x)}$ to make the flow path consistent, which can be done by propagating this rescaling top-down in G. Adjust capacities in G'_{st} to make this new flow feasible.

4. With probability $\frac{1}{2}$, enter the **matching case**:

   (a) Let M_0 be the matrix of the fractional matching between X_{A_S}, X_{A_T} induced at the top-level of G. If M_0 is a partial matching, then make it perfect by adding self loops. Then, the mix-or-move matrix is given by $M \leftarrow \frac{1}{2}M_0 + \frac{1}{2}I$ where I is the identity matrix.

   (b) **return** mix-or-move matrix M, candidate edges $F_{\text{new}} = F = A \uplus R$, set $B_{\text{new}} = B$

5. With probability $\frac{1}{2}$, enter the **deletion case**:

   (a) If $((A \cup R) \setminus A_T) \cup C$ induces a balanced clustering in G, **return** new candidate edges $F_{\textbf{new}} = ((A \cup R) \setminus A_T) \cup C$. In this case, $|F_{\text{new}}| \leq (7/8)|F|$ due to which the cut-matching game restarts from scratch, and the remaining return values are not useful.

   (b) Else set $F_{\textbf{new}} = ((A \cup R) \setminus A_S) \cup C$: Let M_0 be the fractional matching between X_{A_S}, X_{A_T} induced at the top level of G. For each matched pair $(x, x') \in (X_{A_S} \times X_{A_T})$ in M_0, identify the first edge y in C on the flow path $x \rightsquigarrow x'$ using the data-structure G and construct a mix-or-move matrix M which moves the flow vector (instead of mixing it) from x to y. If the total flow received by y from all $x \in A_S$ exceeds 1, rescale this total flow to have value 1. Otherwise, zero out this flow, and add this edge y to set C_B.

   (c) Set $A_{\text{new}} = (A \setminus A_S) \cup (C \setminus C_B)$[a], $R_{\text{new}} = F_{\text{new}} \setminus A_{\text{new}}$.

   (d) **return** mix-or-move matrix M, F_{new}, A_{new}, R_{new}, $B_{\text{new}} = B \cup C_B$.

[a] Note that this update is mentioned as $A_{\text{new}} = (A \setminus A_T) \cup (C \setminus C_B)$ in [61] which is a typo.

Note that although the flow decomposition is an approximation version, the total flow value induced by the decomposition is at least $(1 - \frac{1}{3\log^3 n})^2 \geq (1 - \frac{1}{\log^3 n})$ of the max-flow value (assuming $n \geq 6$), which satisfies the desired requirement as in [61]. We now show that the matching-or-deletion subroutine can be implemented under the PRAM setting efficiently.

CLAIM B.2. *There is a* PRAM *implementation of the subroutine* `match-or-delete` *using depth* $O\left(D(\mathcal{F}_\varepsilon, m) + D(\mathcal{D}_\delta, m) + \text{polylog}\, n\right)$ *and work* $O\big(T(\mathcal{F}_\varepsilon, m) + T(\mathcal{D}_\delta, m) + m \text{ polylog}\, n\big)$, *where* $D(\mathcal{A}, m)$ *and* $T(\mathcal{A}, m)$ *are the depth and work required by the algorithm* $\mathcal{A} \in \{\mathcal{F}_\varepsilon, \mathcal{D}_\delta\}$ *for* $\varepsilon, \delta = 1/(3\log^3 n)$, *respectively.*

Proof. The first step in the algorithm is the construction of the graph G'_{st} which can be done in $O(1)$ depth and $O(m)$ work. The set C' is returned by $\mathcal{F}_\varepsilon$, and C can be constructed from C' in $O(1)$ depth and $O(m \text{ polylog}\, n)$ work. There are at most $O(m \text{ polylog}\, n)$ flow paths. Hence, using the data structure G returned by the flow decomposition algorithm $\mathcal{D}_\delta$, re-scaling of the flows and adjusting capacities takes $O(\log n)$ depth and $O(m \text{ polylog}\, n)$ work. We now analyze the required depth and work for both the matching and the deletion cases:

- In the matching case, since the fractional matching matrix has support size $O(m \text{ polylog}\, n)$, adding

self-loops and computing the mixing matrix M takes $O(m \operatorname{polylog} n)$ work and $O(1)$ depth.

- In the deletion case, checking whether an induced clustering is balanced in Line 5a can be done by simply deleting the candidate edges and checking the sizes of the resulting connected components which takes $O(m)$ work and $O(\log n)$ depth. If we enter Line 5b, since there are at most $O(m \operatorname{polylog} n)$ matched pairs (x, x'), identifying the first edge in C on the flow path $x \rightsquigarrow x'$ for every matched pair requires $O(\operatorname{polylog}(n))$ depth and $O(m \operatorname{polylog}(n))$ work using the data structure G. The moving matrix M can then be computed in $O(m \operatorname{polylog} n)$ work and $O(1)$ depth. Rescaling and deleting flows can also be performed using $O(m \operatorname{polylog} n)$ work and $O(1)$ depth.

Taking the worst-case work and depth among the cases gives us the desired statement. ∎

B.4 Putting it together: `partition-A1`. In this section we combine the implementation of the cut and matching players to achieve the guarantees of Lemma 3.1 in [61]. This is also the algorithm that achieves the guarantees we claim in Lemma 6.1. Specifically, given a set of edges F that induces a 3/4-balanced clustering of the graph, the goal of Lemma 3.1 is to find a new set of edges F_{new} such that (rf. Lemma 3.1 in [61])

1. either $|F_{\text{new}}| \leq \frac{7}{8}|F|$;
2. or $F_{\text{new}} = A \cup R$ such that $|A| \leq |F|$, $|R| \leq \frac{2}{\log n} A$, and the edges in A are $\Omega(1/\log^2 n)$-well-linked.

The following presents the details of the algorithm that is used to prove Lemma 3.1 in [61].

Subroutine `partition-A1`

Input: Subdivision graph $G' = (V', E')$ of $G = (V, E)$; a set of edges $F \subseteq E$ that induce a 3/4-balanced partition of V.
Output: A new set of edges F_{new} that also induces a 3/4-balanced partition of V such that either

1. $|F_{\text{new}}| \leq (7/8)|F|$, or
2. $F_{\text{new}} = A \cup R$ with A, R disjoint such that edges in A are $\Omega(1/\log^2 n)$-well-linked, and $|R| \leq 2|A|/\log n$.

Procedure (cut-matching game):

1. Initialize matching matrix M_0 corresponding to self-loops; $A = F$; $R = \emptyset$; $B = \emptyset$.
2. For $t = 1, 2, \ldots, O(\log^2 n)$:
   (a) $(A_S, A_T, \eta) \leftarrow$ `find-source-and-sinks`$(G', A, \{M_i\}_{0 \leq i < t})$.
   (b) $(M_t, F_{\text{new}}, A_{\text{new}}, R_{\text{new}}, B_{\text{new}}) \leftarrow$`match-or-delete`$(G', F, A, R, A_S, A_T, B)$.
   (c) If $|F_{\text{new}}| \leq (7/8)|F|$: **return** F_{new} (return condition 1).
   (d) Else, update $B = B_{\text{new}}$, $A = A_{\text{new}}$, $R = R_{\text{new}}$.
   (e) Compute potential P_A of active edges A.
   (f) If $P_A \leq 1/(16n^2)$ (flow vectors in A have mixed):
      i. If $|B| \leq 2|A|/\log n$: **return** $F_{\text{new}} = A \cup B$ (return condition 2).
      ii. Else, we necessarily have $|A \cup R| \leq (7/8)|F|$: **return** $F_{\text{new}} = A \cup R$ (return condition 1).

With Claims B.1 and B.2, it is not difficult to check that the whole procedure of step 2 can be implemented in poly-logarithmic depth and nearly-linear work. We formalize this as follows.

LEMMA B.2. *There is a* PRAM *implementation of the above algorithm that has depth* $O\left((D(\mathcal{F}_\varepsilon, m) + D(\mathcal{D}_\delta, m)) \cdot \operatorname{polylog} n\right)$ *and work* $O\big((m + T(\mathcal{F}_\varepsilon, m) + T(\mathcal{D}_\delta, m)) \cdot \operatorname{polylog} n\big)$ *for* $\varepsilon, \delta = O(1/\log^3 n)$.

Proof. Firstly, the number of iterations of the cut-matching game is upper bounded by $O(\log^2 n)$. Then the lemma follows immediately from the work and depth calculations of the cut and matching steps in Claims B.1 and B.2, respectively. ∎

B.5 Extending to the capacitated case. Although [61] states that their results hold for capacitated graphs, they do not detail this extension. However, for completeness, we sketch the implementation of the cut matching game in [61] on capacitated graphs, and discuss how they can be implemented in PRAM. Throughout this section we assume graphs have integer capacities bounded by $\operatorname{poly}(n)$.

We note that the analysis of the cut matching game follows by viewing each edge e of capacity c_e as c_e uncapacitated parallel copies. It thus remains to show that the cut matching game can be implemented with $O(m \operatorname{polylog} n)$ work and $\operatorname{polylog} n$ depth, assuming a $(1-\varepsilon)$-approximate maxflow algorithm with the same work and depth. We first describe a key subroutine that we need throughout the cut matching game, namely *averaging of flow vectors*, and then describe the changes we make to the cut and matching players.

Averaging flow vectors. Note that naively, treating each capacitated edge as uncapacitated copies can potentially lead to a large number of edges, and maintaining flow vectors on them will be too costly. Therefore we will always maintain the invariant that the parallel copies of the same capacitated edge carries the same flow vector. Whenever this variant gets violated (e.g. after a flow mixing or deletion step), we restore it by averaging out the flow vectors across the parallel copies. Thereby, we never store more than $O(m \operatorname{polylog}(n))$ flow vectors at any point. Notice that in the actual implementation of the cut matching game, we never explicitly store the flow vectors but only store their projections; and thus we only need to average the projections of these vectors, which is equivalent to first averaging the flow vectors and then taking projections by linearity. The averaging of the projections can be done in $O(\log n)$ depth and near-linear work by parallel summation.

Moreover, as we show in the claim below, the potential function only decreases after the averaging of any collection of flow vectors. Recall that in a graph with m_0 uncapacitated edges $e_1, \ldots, e_{m_0}$ with flow vectors $f_{e_1}, \ldots, f_{e_{m_0}}$ on them, the potential function in [61] is defined to be

$$\Phi_f := \min_c \sum_{i=1}^{m_0} \|f_{e_i} - c\|^2 = \sum_{i=1}^{m_0} \|f_{e_i} - \mu\|^2,$$

where $\mu = \frac{1}{m_0} \sum_{i=1}^{m_0} f_{e_i}$ is the average of all flow vectors.

CLAIM B.3. *Let* $E' \subseteq E$ *be any collection of uncapacitated edges. Define new flow vectors* $f'_{e_1}, \ldots, f'_{e_{m_0}}$ *by averaging out the flow vectors on edges in* E'*, namely,*

$$f'_e = \begin{cases} \frac{1}{|E'|} \sum_{e \in E'} f_e & e \in E' \\ f_e & e \notin E' \end{cases}.$$

Then the potential function can only decrease going from f *to* f'*:*

$$\Phi_f \geq \Phi'_f.$$

Proof. Let $\mu = \frac{1}{m_0} \sum_{i=1}^{m_0} f_{e_i}$ be the average flow vector of all edges. Note that this is also the average flow vector with respect to f', since averaging the flow vectors of edges in E' does not change the total sum

of the flow vectors. Thus it suffices to compare the contribution of edges in E' to the potential function with respect to f, f' respectively. To this end, we write the contribution with respect to f' as

$$\begin{aligned}\sum_{e\in E'} \|f'_e - \mu\|^2 =& \sum_{e\in E'} \left\| \frac{1}{|E'|} \sum_{e\in E'} f_e - \mu \right\|^2 \\ =& |E'| \cdot \left\| \frac{1}{|E'|} \sum_{e\in E'} f_e - \mu \right\|^2 .\end{aligned}$$

Letting $\mathcal{D}$ denote the uniform distribution over edges in E', we can write the above contribution as

$$|E'| \cdot \|\mathbb{E}_{e\in\mathcal{D}}[f_e] - \mu\|^2 .$$

Notice that the function $f(x) = \|x - \mu\|^2$ is quadratic and hence also convex. Therefore we can apply Jensen's inequality and obtain

$$\begin{aligned}|E'| \cdot \|\mathbb{E}_{e\in\mathcal{D}}[f_e] - \mu\|^2 \leq& |E'| \cdot \mathbb{E}_{e\in\mathcal{D}}\left[\|f_e - \mu\|^2\right] \\ =& \sum_{e\in E} \|f_e - \mu\|^2 .\end{aligned}$$

That is, the contribution of edges in E' with respect to f' is at most that with respect to f, implying that the potentail of f' can only be smaller than that of f, as desired. ∎

The cut and matching players. The cut player uses the exact same strategy to find the sources and sinks, except that now they do not explicitly operate all parallel copies, but rather manipulate the parallel copies of the same edge together, by exploiting the fact that they all have the same flow vectors. For example, when computing projection of the flow vectors, we only need to do the computation once for the parallel copies of the same edge; when computing the average of the flow vectors, we just need to compute a weighted average where each flow vector is weighted by the capacity of the corresponding edge. Notice that when choosing A_S, we could end up only choosing a subset of the parallel copies of the same edge, but leaving the remaining parallel copies out. Then the matching player will create a flow graph by connecting super source to the split vertex of the edge with capacity being the number of parallel copies in A_S.

The matching player also adopts almost the same strategy as the uncapacitated case, with the following modifications:

1. Most notably, in both matching case and deletion case, where we mix or move flow vectors, we average the flow vectors of parallel copies of the same edge afterwards. This is because different copies of the same edge could be matched differently, resulting in different flow vectors after mixing/moving. However, since the matching we found has support size $O(m\,\mathrm{polylog}(n))$, the total number of distinct flow vectors is always bounded by $O(m\,\mathrm{polylog}(n))$. Notice that once again, we never explicitly average the flow vectors but only average their projections.

2. At Line 3 of `match-or-delete`, after we find a flow decomposition, we scale the flow paths as follows. For every edge (s, x) with flow $f_{(s,x)} \geq (1/2)c_{(s,x)}$, rescale the flow such that $f_{(s,x)} = c_{(s,x)}$; For every flow path p that uses edge (s, x), rescale the flow on p by $c_{(s,x)}/f_{(s,x)}$ to make the flow path consistent, which can be done by propagating this rescaling top-down in the flow decomposition DAG in $O(m\,\mathrm{polylog}(n))$ work and $\mathrm{polylog}(n)$ depth. Adjust capacities in G'_{st} accordingly.

3. At Line 5b of `match-or-delete`, if the total flow received by y exceeds c_y, rescale this total flow to have value c_y. Otherwise, zero out this flow, and add (the parallel copies of) this edge y to set C_B. Notice that the rescaling of the flow can again be done by propagating it top-down in the flow decomposition DAG in $O(m\,\mathrm{polylog}(n))$ work and $\mathrm{polylog}(n)$ depth.

C Parallel Implementation of Sherman's Algorithm.

In this section, we discuss the implementation details of vanilla Sherman's algorithm [66] in the PRAM model, which forms the basis of the near-linear work, polylogarithmic depth approximate max-flow subroutine invoked in our cutting-scheme in Section 6. This algorithm consists of an outer-algorithm that makes $O(\log m)$ many iterative calls to an inner procedure `AlmostRoute` that actually performs the gradient descent. At the end of these calls, we are left with a minimum congestion flow that is *almost* feasible, in the sense that there is a negligible residual demand that can be routed in the flow network with $O(1/\text{poly}(m))$ congestion. This outer-algorithm terminates by routing this residual demand along a maximum spanning tree, achieving feasibility of the superimposed flows (i.e. the resultant flow routes the desired demands b). This relatively simpler outer algorithm is described below, with the bulk of the technical detail being contained in the `AlmostRoute` subroutine that implements gradient descent.

ALGORITHM C.1. `approximate maximum flow`$(G, R, b, \alpha, \varepsilon)$
Input: Graph $G = (V, E, c)$; α-congestion approximator R that is a hierarchical decomposition of G; vertex demands $b \in \mathbb{R}^V$; quality of the congestion approximator $\alpha = O(\text{polylog}\, n)$; precision $\varepsilon > 0$.
Output: A $(1+\varepsilon)$-approximate minimum congestion flow $f \in \mathbb{R}^E$ that routes demands b; $(1-\varepsilon)$-approximate maximum congested cut S.
Procedure:

1: $b_0 \leftarrow b$; compute B, the vertex-edge incidence matrix of G.
2: $(f_0, S_0) \leftarrow$ `AlmostRoute`$(G, R, b_0, \alpha, \varepsilon)$
3: **for** $i \leftarrow 1 \ldots \log(2m)$ **do**
4: $\quad b_i \leftarrow b_{i-1} - Bf_{i-1}$
5: $\quad (f_i, S_i) \leftarrow$ `AlmostRoute`$(G, R, b_i, \alpha, 1/2)$. ▷ subsequent S_i are not needed
6: Let $t = \log(2m)$ be the final iteration counter of the above loop; set $b_T \leftarrow b_t - Bf_t$.
7: Compute the maximum spanning tree T of G.
8: Let f_T be the flow obtained by routing demands b_T on the tree T
9: **return** flow $f = f_T + \sum_{i=1}^{\log(2m)} f_i$; cut $S = (S_0, \overline{S_0})$.

It is easy to see that the above outer-algorithm admits an efficient PRAM implementation, assuming that the subroutine `AlmostRoute` admits a PRAM implementation with near-linear work and polylogarithmic depth; there are only $O(\log n)$ many iterations in the outer algorithm, and lines 1,4,6, and 8 can easily be implemented with $O(m)$ work and $O(1)$ depth, and the maximum spanning tree construction in line 7 has a known $O(m)$ work, $O(\text{polylog}\, n)$ depth PRAM algorithm [60].

We next discuss the implementation specifics of the subroutine `AlmostRoute` which actually performs the optimization and is more involved. The key idea behind this subroutine is to transform the constrained optimization problem (for undirected graphs) given in Eqn. 1.1 into an unconstrained one using the following *congestion potential* function

$$\phi(f) = \mathsf{lmax}(C^{-1}f) + \mathsf{lmax}(2\alpha R(b - Bf)),$$

where for any $x \in \mathbb{R}^k$,

$$\mathsf{lmax}(x) := \log\left(\sum_{i=1}^{k}(e^{x_i} + e^{-x_i})\right)$$

is the symmetric softmax function, a differentiable approximation of $||\cdot||_\infty$. Algorithmically, this is achieved via a standard gradient descent which finds a $(1-\varepsilon)$-approximate maximum flow f after at most $O(\varepsilon^{-3}\alpha^2 \log n)$ iterations. Therefore, an efficient PRAM implementation of this algorithm effectively reduces to finding an efficient implementation of a single iteration of the descent step within this algorithm, a formal description of which is given below.

ALGORITHM C.2. `AlmostRoute`$(G, R, b, \alpha, \varepsilon)$
Input: Graph $G = (V, E, c)$; α-congestion approximator R that is a hierarchical decomposition of G;

vertex demands $b \in \mathbb{R}^V$; quality of the congestion approximator $\alpha = O(\text{polylog}\, n)$; precision $\varepsilon > 0$.
Output: A $(1+\varepsilon)$-approximate minimum congestion flow $f \in \mathbb{R}^E$ that routes demands b; $(1-\varepsilon)$-approximate maximum congested cut S.
Procedure:

1: Initialize $f \leftarrow 0$; compute $k_b \leftarrow (16 \log n)/(2\alpha\varepsilon ||Rb||_\infty)$; scale $b \leftarrow k_b \cdot b$.
2: **repeat**
3: Set $k_f \leftarrow 1$; scaling factor $s \leftarrow 17/16$.
4: **while** $\phi(f) < 16\varepsilon^{-1} \log n$ **do**
5: Scale $k_f \leftarrow s \cdot k_f$; $f \leftarrow s \cdot f$; $b \leftarrow s \cdot b$.
6: Set $\delta \leftarrow \sum_{e \in E} \left| c_e \cdot \frac{\partial \phi(f)}{\partial f_e} \right|$.
7: **if** $\delta \geq \varepsilon/4$ **then**
8: For each edge $e \in E$, update $f_e \leftarrow f_e - \text{sign}\left(\frac{\partial \phi(f)}{\partial f_e}\right) \cdot \frac{\delta c_e}{1+4\alpha^2}$.
9: **else**
10: Undo scaling $f \leftarrow f/k_f$, $b \leftarrow b/(k_b k_f)$.
11: Compute the maximum congested cut $(S, \overline{S})$ from the $(n-1)$ threshold cuts of vertex potentials $\{\pi_v\}_{v \in V}$ induced due to $\partial\phi(f)/\partial f_e$ (described shortly).
12: **return** flow f, cut S .
13: **until termination**

Observe that implementing the above algorithm requires us to compute (i) the value of the potential function $\phi(f)$, and (ii) the partial derivatives $\partial\phi(f)/\partial f_e$ of the potential with respect to the flow on each edge in the graph. Additionally, we also need to bound the total number of iterations of the above algorithm (lines 2, 4), for which we directly leverage the result of Sherman (Lemma 2.5 in [66]), which shows that the total number of iterations until termination (line 2) is $O(\alpha^2 \varepsilon^{-3} \log n)$, and within each iteration, the total number of times we scale the flow and demands (line 4) is $O(\log \alpha)$. We now show how to compute the value of the potential, and its partial derivatives. In order to do so, we shall find it instructive to understand the structure of the congestion approximator R.

The congestion approximator $R \in \mathbb{R}^{x \times n}$ is a matrix with each row $i \in [x]$ corresponding to a cut in the graph, and each column corresponding to a vertex. For any cut $i = (S_i, \overline{S_i})$ considered by the congestion approximator, entry $R_{i,v} \in \{0, 1\}$ indicates whether vertex v lies on the S_i side of the cut, normalized by the total capacity $c(S_i, \overline{S_i})$ of the cut, i.e. the sum of capacities of all edges crossing this cut. Therefore, the product $[Rb']_i$ of this row of the congestion approximator with any demand vector b' gives the congestion that would be induced by routing these demands across the cut $(S_i, \overline{S_i})$. However, we cannot explicitly construct this matrix due to work and depth constraints, and instead shall use the specific structure of the congestion approximator to efficiently compute these congestion values for all cuts explicitly considered by the approximator.

The congestion approximator in our case is a $O(\log n)$ depth rooted tree T corresponding to a hierarchical decomposition of the flow instance $G = (V, E)$ upon which Sherman's algorithm is invoked, with the leaves corresponding to the vertices $v \in V$ in the flow network, and the internal nodes corresponding to a cluster consisting of the leaf vertices in the subtree rooted at that internal node. This hierarchical decomposition tree T can equivalently be viewed as a set of cuts in the input graph; each node $i \in T$ in this tree corresponds to a cut $(S_i, \overline{S_i})$, where S_i is the set of vertices corresponding to the leaves in the subtree rooted at node i in the tree T. In the following analysis, we shall leverage this view of the congestion approximator in order to efficiently compute the value of the congestion potential, as well as its partial derivatives.

We begin by decomposing the congestion potential into its two components

$$\phi(f) = \phi_1(f) + \phi_2(f); \text{ where } \phi_1(f) = \mathsf{lmax}(C^{-1} f), \text{ and } \phi_2(f) = \mathsf{lmax}(2\alpha R(b - Bf)).$$

To compute the first component $\phi_1(f)$, we can simply compute the congestion f_e/c_e of every edge $e \in E$ in parallel, followed by an aggregation step, which can be done with $O(m)$ total work and $O(\log n)$ depth.

To compute the second component $\phi_2(f)$, we can compute the residual demands $b_v - \sum_{e\in\delta(v)} B_{v,e} f_e$ for every vertex $v \in V$ where $\delta(v)$ corresponds to the set of edges incident on vertex v. This also requires $O(m)$ total work and $O(\log n)$ depth (with every vertex first reading the flow values of incident incoming edges followed by those of incident outgoing edges in two separate passes to avoid read collisions). The total demand of any subset of vertices in the congestion approximator (rooted-tree) can then be computed with $\widetilde{O}(n)$ total work and $O(\log n)$ depth using subtree sums. Given the capacity $c(S_i, \overline{S_i})$ of every cut $i = (S_i, \overline{S_i})$ represented by the internal nodes in our congestion approximator, the second term can then be computed via an aggregation, which can be done with $O(m)$ total work and $O(\log n)$ depth.

To compute the partial derivatives, we first consider the component $\phi_1(f)$ in our congestion potential. Then we have that for any edge $e \in E$, the partial derivative

$$\frac{\partial \phi_1(f)}{\partial f_e} = \frac{\exp(f_e/c_e) - \exp(-f_e/c_e)}{c_e \cdot \exp(\phi_1(f))}$$

which can easily be computed with $O(m)$ work and $O(1)$ depth when the potential $\phi_1(f)$ is known (its computation is described above).

To compute the partial derivative of the second component $\phi_2(f)$, let $\mathcal{I}$ be the set of all cuts (rows) considered by our congestion approximator, and for any cut $i = (S_i, \overline{S_i}) \in \mathcal{I}$, let $y_i = 2\alpha[R(b - Bf)]_i$ be the congestion induced by the residual demands across cut $i = (S_i, \overline{S_i})$. Then we have for any edge $e \in E$, the partial derivative

$$\frac{\partial \phi_2(f)}{\partial f_e} = \sum_{i\in\mathcal{I}} \frac{\partial \phi_2(f)}{\partial y_i} \cdot \frac{\partial y_i}{\partial f_e} = \sum_{i\in\mathcal{I}} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha B_{S_i,e}}{c(S_i, \overline{S_i})},$$

where $c(S_i, \overline{S_i})$ is the capacity of the cut $i = (S_i, \overline{S_i})$ considered in our congestion approximator, and (with some abuse of notation) $B_{S_i,e} = \sum_{v\in S_i} B_{v,e} \in \{-1, 0, 1\}$ is an indicator of whether in cut i, edge e is an incoming edge (1), outgoing edge (-1) or does not cross it (0). The cuts $\mathcal{I}$ are not arbitrary. Rather, they are induced by a single rooted hierarchical decomposition tree T, which we can use to efficiently compute this partial derivative for every edge. For an edge $e = (u, v) \in E$, let $T_{u,v}$ be the unique path between u, v in T. Then we have that

$$\frac{\partial \phi_2(f)}{\partial f_e} = \sum_{i\in T_{u,v}} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha B_{S_i,e}}{c(S_i, \overline{S_i})}.$$

Now observe that for any internal node i (corresponding to some cut $(S_i, \overline{S_i})$) that is encountered on the path $T_{u,x}$ between u and the *least-common-ancestor* x of u, v in the rooted tree T, we have $B_{S_i,e} = -1$, and for any internal node i that is encountered on the path $T_{x,v}$ between v and x in $\mathcal{T}$, we have $B_{S_i,e} = +1$. Now for any internal node j in T, let $T_{j,r}$ denote the unique path in T from the root r of T to node j. Then we can define for every internal node j in T, a node potential π_j as

$$\pi_j := \sum_{i\in T_{j,r}} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha}{c(S_i, \overline{S_i})},$$

which is easy to compute with $O(n)$ total work and $O(\log n)$ depth through a prefix sum on an Eulerian tour of T that starts and ends at the root r of T. This is achieved by setting the weight of the forward edge entering an internal node i from its parent to be $+(\exp(y_i) - \exp(-y_i))/(\exp(\phi_2(f))) \cdot (2\alpha/c(S_i, \overline{S_i}))$ and the reverse edge leaving the internal node i going to its parent to be $-(\exp(y_i) - \exp(-y_i))/(\exp(\phi_2(f))) \cdot (2\alpha/c(S_i, \overline{S_i}))$. Therefore, sum corresponding to subtrees in the prefix sums evaluate to 0, leaving just the sum of the root r to node i path. Given these node potentials, it is now easy to compute the partial derivatives of any edge $e = (u, v)$ as

$$\frac{\partial \phi_2(f)}{\partial f_e} = \pi_v - \pi_u,$$

which requires just $O(m)$ total work and $O(\log n)$ depth.

Lastly, Sherman shows that these vertex potentials π_v induced by the flow when it is approximately optimal (i.e. when the subroutine terminates) also allow us to efficiently recover the approximate minimum cut (equivalently, the approximate maximum congested cut). Specifically, one of the threshold cuts with respect to the vertex potentials is an approximate min-cut, and this can be computed efficiently in $\widetilde{O}(m)$ total work and $O(\log n)$ depth by sorting the vertices by their potential values and returning the most congested cut from the resulting $n-1$ threshold cuts. Therefore, we have that Sherman's algorithm admits an efficient implementation in the PRAM model.

D Computing Min Cut on Trees.

To compute a hierarchical decomposition on trees in Section 5.2, we need to compute an *exact* s-t min-cut cut on a congestion approximator tree with the addition of a super-source s and super-sink t. In this section, we show how to compute the exact min-cut when the tree has $O(\log n)$ depth, which is simpler, before extending it to trees with arbitrary depth.

The easy case: $O(\log n)$ depth. Let T be a tree rooted at r with $O(\log n)$ depth, let $s \notin T$ be a super-source, and $t \notin T$ be a super-sink; s and t may be connected arbitrarily to T and these edges may have arbitrary capacity. For a s-t min-cut $(S, \bar{S})$ of $T \cup \{s,t\}$, without loss of generality $s \in S$ and $t \in \bar{S}$. As such, it remains to determine which nodes of T are in S and which are in $\bar{S}$, and so we consider finding a s-t min-cut as a tree problem. To account for the capacities of edges incident on s or t, for each $u \in T$, we set node weights $w_u^s = c_{us}$ and $w_u^t = c_{ut}$, where c_{us}, c_{ut} are the capacities of the (u,s) and (u,t) edges, respectively, or 0 if no such edge exists.

To find the min-cut capacity, we can use dynamic programming. For each $u \in T$, define $\texttt{cut}^s(u)$ and $\texttt{cut}^t(u)$ as the s-t min-cut capacity of the subtree rooted at u with the restriction that u is on the s side of the cut (i.e. $u \in S$) or u is on the t side of the cut (i.e. $u \in \bar{S}$), respectively. The recurrence relations are then $\texttt{cut}^s(u) = w_u^t + \sum_{v \in D(u)} \min\{\texttt{cut}^s(v), c_{uv} + \texttt{cut}^t(v)\}$ and $\texttt{cut}^t(u) = w_u^s + \sum_{v \in D(u)} \min\{c_{uv} + \texttt{cut}^s(v), \texttt{cut}^t(v)\}$, where $D(u)$ is the set of children of u. Since we assume T has $O(\log n)$ depth, standard dynamic programming techniques allow us to compute $\min\{\texttt{cut}^s(r), \texttt{cut}^t(r)\}$, which is the s-t min-cut capacity, in $O(\log n)$ depth and $O(n)$ work. For brevity, we have presented computing only the capacity of the min-cut, but the actual cut may be found by storing the argmin for each minimum taken in the recurrence relations.

Before continuing to the arbitrary depth case, we introduce a (slightly) generalized problem where some vertices are constrained to be in S, or $\bar{S}$, an extension useful when extending to trees of arbitrary depth.

DEFINITION D.1. ((F_s, F_t)-RESTRICTED s-t MIN-CUT) *Let T be a tree, let s be a super-source and let t be a super-sink. Then, given disjoint subsets F_s, F_t of nodes of T, a (F_s, F_t)-Restricted s-t Min-Cut of $T \cup \{s,t\}$ is a minimum s-t cut $(S, \bar{S})$ under the restriction that $F_s \cup \{s\} \subseteq S$ and $F_t \cup \{t\} \subseteq \bar{S}$.*

The DP presented before can be easily modified to also solve this extended version when T has $O(\log n)$ depth, using recurrence relations

$$\texttt{res\_cut}^s(u) = \begin{cases} w_u^t + \sum_{v \in D(u)} \min\{\texttt{res\_cut}^s(v), c_{uv} + \texttt{res\_cut}^t(v)\} & \text{if } u \notin F_t \\ \infty & \text{if } u \in F_t \end{cases} \tag{D.1}$$

$$\texttt{res\_cut}^t(u) = \begin{cases} w_u^s + \sum_{v \in D(u)} \min\{c_{uv} + \texttt{res\_cut}^s(v), \texttt{res\_cut}^t(v)\} & \text{if } u \notin F_s \\ \infty & \text{if } u \in F_s \end{cases} \tag{D.2}$$

where again $D(u)$ is the set of children of $u \in T$.

LEMMA D.1. *Let T be a tree on n nodes rooted at r with depth $O(\log n)$, and let $\texttt{res\_cut}^s$ and $\texttt{res\_cut}^t$ be defined as in (D.1) and (D.2). Then, given disjoint subsets F_s, F_t of vertices of T,*

$\min\{\texttt{res\_cut}^s(r), \texttt{res\_cut}^t(r)\}$ *is the capacity of an* (F_s, F_t)*-Restricted* s*-*t *Min-Cut on* $T \cup \{s, t\}$*. Moreover, this value can be computed using an* $O(\log n)$ *depth and* $O(n)$ *total work PRAM algorithm.*

Proof. By straightforward dynamic programming, since T has $O(\log n)$ depth, $\texttt{res\_cut}^s(r)$ and $\texttt{res\_cut}^t(r)$ can be computed in $O(\log n)$ depth and $O(n)$ work. For correctness, first note that the base cases are correct: the cost of any solution where $x \in F_s$ is placed in $\bar{S}$ is infinite (and analogously for F_t and S) and, by construction, w_u^s is the capacity of the edge between u and s, if it exists (and similarly for w_u^t). The correctness then follows by induction. ∎

Extending to arbitrary depth trees. When T has super-logarithmic depth, we modify the DP and divide into subproblems based on tree separator nodes (see Definition 5.3) rather than children. For a DP subproblem to find the s-t min-cut on a tree T', we compute a tree separator node q and recurse on each tree of $T' \setminus \{q\}$. From the definition of a tree separator node, this results in subproblems on trees which are at most half the size of T'.

When combining subproblems, we use the recursive calls to determine the min-cut capacity when $q \in S$ and the min-cut capacity with $q \in \bar{S}$, and return the lower value. To do this, we must also determine for each $u \in \delta(q)$ (i.e. each neighbor of q in T') whether $u \in S$ or $u \in \bar{S}$, in order to determine whether to add the capacity of the (u, q) edge to the cut. As such, for every neighbor u of q, we compute 2 subproblems on the connected component of $T' \setminus \{q\}$ containing u: one where we constrain $u \in S$, and one where we constrain $u \in \bar{S}$. Due to recursion, in any given subproblem there might be multiple vertices which are constrained to be in S or $\bar{S}$; we call these the *tracked* vertices. Each subproblem then takes as input a tree T', a set of tracked vertices $\mathcal{A}$, and subset $\mathcal{S} \subseteq \mathcal{A}$, where the goal of the subproblem is to compute a s-t min-cut $(S, \bar{S})$ on T' under the restriction that $\mathcal{S} \subseteq S$ and $(\mathcal{A} \setminus \mathcal{S}) \subseteq \bar{S}$.

However, there are $2^{|\mathcal{A}|}$ possible subsets $\mathcal{S} \subseteq \mathcal{A}$, and so the number of possible subproblems (and thus total work) is exponential in the number of tracked vertices. As such, we must bound the number of tracked vertices in any subproblem. To do this, if we ever have a subproblem with 3 tracked vertices, rather than recursing on the components formed by removing a tree separator node, we recurse on the trees formed by removing the LCA of 2 tracked vertices. By rooting every subtree at a tracked vertex, this results in new subproblems with at most 2 tracked vertices, and so all subproblems have at most 3 tracked vertices. Importantly, these additional steps to reduce the number of tracked vertices at most double the depth of the recursion, leading to $O(\log n)$ levels of recursion. Since each level of recursion can be implemented in $O(\log n)$ depth and $O(n)$ work via dynamic programming, we obtain a $O(\log^2 n)$ depth, $\tilde{O}(n)$ work PRAM algorithm.

Below we present the full algorithm for general trees. For brevity, we present computing the capacity of the min-cut; the actual cut may be found by storing the argmin for each min taken in the recurrence relation. We reuse the notation from the algorithm for $O(\log n)$ depth, with S denoting the side of the cut containing s and w_u^s, w_u^t denoting the capacity of the (u, s) or (u, t) edge, respectively, and 0 if no such edge exists.

Min Cut on Trees Subroutine
Inputs: Tree T with root r, super-source $s \notin T$ and super-sink $t \notin T$, both connected arbitrarily to T.

Goal: For T' a subtree of T, $\mathcal{A} \subseteq T'$ a set of tracked vertices and $\mathcal{S} \subseteq \mathcal{A}$, recursively compute $\texttt{cut}(T', \mathcal{A}, \mathcal{S})$, which is the capacity of a s-t min-cut $(S, \bar{S})$ on $T' \cup \{s, t\}$ such that $\mathcal{S} \cup \{s\} \subseteq S$ and $(\mathcal{A} \setminus \mathcal{S}) \cup \{t\} \subseteq \bar{S}$.

Output: $\texttt{cut}(T, \emptyset, \emptyset)$ as the s-t min-cut capacity.

Computing $\texttt{cut}(T', \mathcal{A}, \mathcal{S})$**:**
If the depth of T' is at most $10 \log n$, run the algorithm of Lemma D.1 on $T' \cup \{s, t\}$, with $F_s = \mathcal{S}$ and $F_t = \mathcal{A} \setminus \mathcal{S}$ (and T' rooted arbitrarily).

Otherwise, with T' rooted at any tracked vertex when $|\mathcal{A}| \neq \emptyset$ and arbitrarily otherwise:

1. Compute a split vertex q:
   (a) If $|\mathcal{A}| \leq 2$, set q to be a tree separator node of T', using the algorithm of Lemma A.2.
   (b) If $|\mathcal{A}| = 3$, set q as the LCA of 2 non-root tracked vertices, using the algorithm of Theorem A.5.
2. Compute the connected components of $T' \setminus \{q\}$, and for each $u \in \delta(q)$[a] define C^u as the component containing u.
3. For each $u \in \delta(q)$, update tracked vertices $\mathcal{A}^u = (\mathcal{A} \cap C^u) \cup \{u\}$ and $\mathcal{S}^u = \mathcal{S} \cap \mathcal{A}^u$.
4. Compute the min-cut capacity conditioned on $q \in S$:
$$\texttt{cut}_{q \in S} = w_q^t + \sum_{u \in \delta(q)} \min\{w_u^t + \texttt{cut}(C^u, \mathcal{A}^u, \mathcal{S}^u \cup \{u\}),\ w_u^s + c_{uq} + \texttt{cut}(C^u, \mathcal{A}^u, \mathcal{S}^u)\}$$
5. Similarly, compute the min-cut capacity conditioned on $q \in \bar{S}$:
$$\texttt{cut}_{q \in \bar{S}} = w_q^s + \sum_{u \in \delta(q)} \min\{w_u^t + c_{uq} + \texttt{cut}(C^u, \mathcal{A}^u, \mathcal{S}^u \cup \{u\}),\ w_u^s + \texttt{cut}(C^u, \mathcal{A}^u, \mathcal{S}^u)\}$$
6. If $q \notin \mathcal{A}$, return
$$\texttt{cut}(T', \mathcal{A}, \mathcal{S}) = \min\{\texttt{cut}_{q \in S}, \texttt{cut}_{q \in \bar{S}}\}$$
7. If $q \in \mathcal{A}$ and $q \in \mathcal{S}$, return $\texttt{cut}(T', \mathcal{A}, \mathcal{S}) = \texttt{cut}_{q \in S}$.
8. If $q \in \mathcal{A}$ and $q \notin \mathcal{S}$, return $\texttt{cut}(T', \mathcal{A}, \mathcal{S}) = \texttt{cut}_{q \in \bar{S}}$.

[a] $\delta(q)$ is the set of neighbors of q in T'

LEMMA D.2. (MIN CUT ON TREES) *The algorithm Min Cut on Trees computes the s-t min-cut capacity of a tree T with the addition of a super-source s and super-sink t in $O(\log^2 n)$ depth and $\tilde{O}(n)$ work.*

Proof. To bound the depth and work of the algorithm, we first show that the number of tracked vertices in any subproblem is always at most 3. The algorithm begins with a call that has no tracked vertices, and each recursive call adds at most one additional tracked vertex to each subproblem. So, to bound the number of tracked vertices, it suffices to show that if a subproblem has 3 tracked vertices, the number of tracked vertices in each recursive call is at most 2. Consider the call $\texttt{cut}(T', \mathcal{A}, \mathcal{S})$ with $|\mathcal{A}| = 3$, and let q be the LCA computed in Line 1. By the definition of an LCA and the fact that we always root T' at a tracked vertex, it follows that the path in T' between any 2 tracked vertices passes through q. As such, after removing q from T', each element of $\mathcal{A}$ is in a separate connected component. Thus, the tracked sets used in the recursive calls each have at most 2 elements: at most one element from $\mathcal{A}$, and one neighbor of q.

We claim there are at most $O(\log n)$ recursive levels of the algorithm, where recursive level i consists of all subproblems resulting from i consecutive recursive calls to $\texttt{cut}(T, \emptyset, \emptyset)$. Consider a subproblem to compute $\texttt{cut}(T', \mathcal{A}, \mathcal{S})$ with at most 2 tracked vertices (i.e. $|\mathcal{A}| \leq 2$). In this case, we recurse on subtrees which are connected components after the removal of a tree separator node. By the definition of a tree separator node (Definition 5.3), this removal results in subtrees which are at most half the size of T'. So, there can be at most $O(\log n)$ such steps before the recursion terminates. Now, suppose $|\mathcal{A}| = 3$ and the subproblem $\texttt{cut}(T', \mathcal{A}, \mathcal{S})$ occurs at level i. The resulting subproblems in level $i + 1$ have at most 2

tracked vertices, and so in level $i+2$, the size of the resulting subtrees is at most half the size of T'. Thus, there are $O(\log n)$ total levels. Each recursive level can be processed in parallel, in $O(\log n)$ depth (to compute connected components, subtree sums, and combining recursive calls), making the total depth of the algorithm $O(\log^2 n)$.

The subtrees (i.e. all distinct T' from subproblems) at each level of recursion form a partition of the nodes of T. Moreover, by rooting trees and computing the split vertex q deterministically, every subproblem on T' has the same set of tracked vertices. So, since there are at most 3 tracked vertices in any subproblem, there are at most 8 possible sets $\mathcal{S} \subseteq \mathcal{A}$, and thus at most 8 subproblems using any one subtree. The work to compute a k node subtree, outside of recursion, is at most $O(k)$. So, the total work at each level is $O(n)$, and as there are $O(\log n)$ levels, the complete work is $\tilde{O}(n)$.

For subtrees with depth $O(\log n)$, the correctness follows from Lemma D.1. For trees with larger depth, we must have either $q \in S$ or $q \in \bar{S}$, where q is the split vertex computed in Line 1. By induction and the setting of the edge weights w^s_u, w^t_u, $\texttt{cut}_{q\in S}$ is the min-cut with the restrictions on $\mathcal{A}$ imposed by $\mathcal{S}$ when $q \in S$ (and similarly for $\texttt{cut}_{q\in\bar{S}}$). So, as we take the min of $\texttt{cut}_{q\in S}, \texttt{cut}_{q\in\bar{S}}$ when $q \notin \mathcal{A}$, $\texttt{cut}(T', \mathcal{A}, \mathcal{S})$ is the correct min-cut capacity. ∎

E Ensuring Polynomial Aspect Ratio.

In this section, we give a $O(\log n)$ depth, $\tilde{O}(m)$ work PRAM algorithm which, given s, t and ε converts any capacitated graph G into one with with $\text{poly}(n/\varepsilon)$ aspect ratio that preserves the s-t maximum flow up to a $(1-\varepsilon)$ factor.

ALGORITHM E.1. `lower-aspect-ratio`(G, s, t, ε)
Input: Graph G with arbitrary capacities, $s, t \in V(G)$, error parameter ε.
Output: Graph G' with $\text{poly}(n/\varepsilon)$ aspect ratio. Moreover, with f the max s-t flow value in G and f' the max s-t flow value in G', $(1-\varepsilon)f \le f' \le f$.
Procedure:

1: Initialize $G' \leftarrow G$.
2: Let T be a maximum spanning tree of G, computed using the algorithm of [60].
3: Compute c' as the capacity of the lowest capacity edge on the unique s-t path in T.
4: For any edge e in G' with capacity larger than mc', reduce its capacity to mc'.
5: For any edge e in G' with capacity less than $\varepsilon c'/m$, delete e.
6: **return** G'

The algorithm of [60] has $O(\log n)$ depth and $O(m)$ work and finding all edges on the s-t path in T can be done in $O(\log n)$ depth and $\tilde{O}(m)$ work using subtree sums, so computing c' and modifying the capacities can be done in $O(\log n)$ depth and $\tilde{O}(m)$ work. Moreover, by construction, the modified graph G' has aspect ratio $m^2/\varepsilon = \text{poly}(n/\varepsilon)$ and no capacities have been increased, so it remains to show that the maximum s-t flow does not reduce by more than a $(1-\varepsilon)$ factor.

For this, we first need the following simple lemma about maximum spanning trees.

LEMMA E.1. *Let G be a capacitated graph and let T be a maximum capacity spanning tree of G. Let P be the unique s-t path in T (which is also a path in G), and let P' be any other s-t path in G. Then,*

$$\min_{e\in P} c_e \ge \min_{e\in P'} c_e$$

where c_e is the capacity of edge e.

Proof. Let $c' = \min_{e\in P} c_e$, and suppose for contradiction there exists in G a s-t path P' such that for all $e \in P'$, $c_e > c'$. Let e' be an edge on P with capacity c'. Removing e' from T results in exactly two connected components S and $V \setminus S$ (with $s \in S$ and $t \in V \setminus S$). Since P' is an s-t path, it must contain an edge $h = (u, v)$ such that $u \in S$ and $v \in V \setminus S$, so removing e' from T while adding h results in a spanning tree T'. But $c_h > c_{e'}$ by assumption, and so the capacity of T' is greater than that of T, contradicting that T is a maximum spanning tree of G. ∎

This then allows us to show the procedure decreases the maximum flow by at most a ε factor.

LEMMA E.2. *Let G be a capacitated graph and let $G' = \texttt{lower-aspect-ratio}(G, s, t, \varepsilon)$. Suppose it possible to route f units of flow from s to t in G. Then, it is possible to route $(1-\varepsilon)f$ units of flow from s to t in G'.*

Proof. Let T be a maximum spanning tree of G, and let c' be the capacity of the minimum capacity edge on the unique s-t path in T. By Lemma E.1, all other s-t paths in G have minimum capacity at most c'; thus, every s-t path can support at most c' units of flow. There can be at most m disjoint paths from s to t, and so it follows that $f \leq c'm$. Thus, reducing the capacity of all edges with capacity greater than mc' to mc' does not affect the maximum flow. Similarly, clearly $f \geq c'$, as there is a s-t path with minimum capacity c'. Thus, as the sum flow on edges of capacity at most $\varepsilon c'/m$ is at most $\varepsilon c'$, deleting edges of capacity at most $\varepsilon c'/m$ reduces the flow by at most εf. It then follows that the max flow in G' must have value at least $(1-\varepsilon)f$. ∎