Parallel Minimum Cost Flow in Near-Linear Work and Square Root Depth for Dense Instances

Jan van den Brand
Georgia Institute of Technology
Atlanta, Georgia, USA
vdbrand@gatech.edu

Hossein Gholizadeh
Karlsruhe Institute of Technology
Karlsruhe, Germany
hgholizadeh8@gmail.com

Yonggang Jiang
Max Planck Institute for Informatics and Saarland University
Saarbücken, Germany
yjiang@mpi-inf.mpg.de

Tjin de Vos
TU Graz
Graz, Austria
tijn.devos@tugraz.at

ABSTRACT

For n-vertex m-edge graphs with integer polynomially-bounded costs and capacities, we provide a randomized parallel algorithm for the minimum cost flow problem with $\widetilde{O}(m + n^{1.5})$ work and $\widetilde{O}(\sqrt{n})$ depth[1]. On moderately dense graphs ($m > n^{1.5}$), our algorithm is the first one to achieve both near-linear work and sub-linear depth. Previous algorithms are either achieving almost optimal work but are highly sequential [CKL+22], or achieving sub-linear depth but use super-linear work [LS14, OS93]. Our result also leads to improvements for the special cases of max flow, bipartite maximum matching, shortest paths, and reachability. Notably, the previous algorithms achieving near-linear work for shortest paths and reachability all have depth $n^{o(1)} \cdot \sqrt{n}$ [JLS19, FHL+25].

Our algorithm consists of a parallel implementation of [BLL+21]. One important building block is a parallel batch-dynamic expander decomposition, which we show how to obtain from the recent parallel expander decomposition of [CMGS25].

CCS CONCEPTS

• Theory of computation → Shared memory algorithms; Network flows.

KEYWORDS

Maximum Flow, Bipartite Matching, Parallel Algorithm, Expander Decomposition

ACM Reference Format:
Jan van den Brand, Hossein Gholizadeh, Yonggang Jiang, and Tjin de Vos. 2025. Parallel Minimum Cost Flow in Near-Linear Work and Square Root Depth for Dense Instances. In 37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28–August 1, 2025, Portland, OR, USA. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3694906.3743356

[1]We use $\widetilde{O}(\cdot)$ to hide polylogarithmic factors.

SPAA '25, July 28–August 1, 2025, Portland, OR, USA

ACM ISBN 979-8-4007-1258-6/25/07
https://doi.org/10.1145/3694906.3743356

1 INTRODUCTION

Minimum cost flow is one of the most fundamental questions in algorithm design. It has been widely studied in the sequential setting, but relatively few results are known in the parallel setting. All previous results for min-cost flow, including its important special cases max flow and bipartite maximum matching, either achieve optimal work but are highly sequential [CKL+22, BLL+21], or achieve sublinear depth of non-optimal work: Lee and Sidford [LS14] have $\widetilde{O}(m\sqrt{n})$ work and $\widetilde{O}(\sqrt{n})$ depth, and the matrix multiplication technique which gives $\widetilde{O}(m^{\omega+4})$ work[2] and $\widetilde{O}(1)$ depth [OS93]. This brings us to the natural question.

Question: Can we achieve both linear work and sublinear depth for min-cost flow?

Although parallel flow has been studied for decades (see, e.g. [SV82, KUW86, R+90]), only very recently breakthrough results answer this question positively for the special cases of reachability [Fin20, JLS19], shortest path [RGH+22, CF23] and negative weight SSSP [ABC+24], where the current record is near-linear work and $n^{1/2+o(1)}$-depth. However, the question of getting linear work and sublinear depth still remains widely open for bipartite maximum matching, max flow, and the most general min-cost flow.

In this paper, we resolve the most general problem for moderately dense graphs: for $m \geq n^{1.5}$, we give a near-linear work and $\widetilde{O}(\sqrt{n})$-depth algorithm for min-cost flow, which matches the current progress on shortest path related problems, where it even improves the small $n^{o(1)}$ factor.

Theorem 1.1 (Informal). There exists a randomized algorithm that computes exact min-cost flow in $\widetilde{O}(m+n^{1.5})$ work and $\widetilde{O}(\sqrt{n})$ depth.

[2]Here, $\omega \approx 2.37$ [DWZ23, VXXZ24] denotes the exponent of current matrix multiplication.

Techniques. To obtain this result, we use an interior point method based on the sequential algorithm by v.d.Brand et al. [BLL+21]. Their interior point method contains $\widetilde{O}(\sqrt{n})$ iterations, and they show how to perform each iteration in amortized $\widetilde{O}(m/\sqrt{n}+n)$ time by using sequential data structures maintaining the information needed for each iteration. For an overview, see Section 2.2.

Our contribution is to parallelize the data structures so that they cost $\widetilde{O}(m/\sqrt{n}+n)$ work and $\widetilde{O}(1)$ depth per iteration, which then implies a $\widetilde{O}(m+n^{1.5})$ work and $\widetilde{O}(\sqrt{n})$ depth flow algorithm. In particular, most of the operations used by the data structures involve multiplying matrices and are thus easily parallelizable. The bottleneck is the following data structure involving parallel batch-dynamic expander decomposition: given edge deletions (or insertions) in batches, maintaining an expander decomposition of the graph in work proportional to the batch size and depth $\widetilde{O}(1)$.

Previous work shows dynamic expander decomposition can be done in sequential time proportional to the batch size [SW19]. However, their algorithm involving push-relabel is highly sequential. A more recent work [CMGS25] shows how to use parallel push-relabel to get a parallel static expander decomposition. Our main contribution is adapting their algorithms to get a parallel batch-dynamic expander decomposition that supports low-depth updates. We provide a detailed overview in Section 2.3.

In the next section, we present our main result and its corollaries more formally, also comparing it in detail to the state of the art.

1.1 Our Results

Min-Cost Flow and Max Flow. In the minimum cost flow problem (min-cost flow), we are given a connected directed graph $G=(V,E,u,c)$ with edges capacities $u\in\mathbb{R}^E_{\geq 0}$ and costs $c\in\mathbb{R}^E$. Given $s,t\in V$, we call $x\in\mathbb{R}^E$ an s-t flow if $x_e\in[0,u_e]$ for all $e\in E$ and for each vertex $v\in V\setminus\{s,t\}$ the amount of flow entering v equals the amount of flow leaving v, i.e., $\sum_{(u,v)\in E} e_{(u,v)}=\sum_{(v,u)\in E} e_{(v,u)}$. The value of the flow is the amount of flow leaving s (or equivalently, entering t): $\sum_{(s,u)\in E} e_{(s,u)}=\sum_{(u,t)\in E} e_{(u,t)}$. The maximum flow problem (max flow) is to compute an s-t flow of maximum value. In the minimum cost maximum flow problem (min-cost flow), the goal is to compute a maximum s-t flow x of minimum cost, $x^\top c$.

The min-cost flow problem generalizes many important graph problems, among which the max flow problem, maximum bipartite matching, the negative-weight shortest path problem, and reachability.

Theorem 1.2. There exists an algorithm that, given a directed graph $G=(V,E,u,c)$ with capacities $u\in\mathbb{Z}^E_{\geq 0}$ and costs $c\in\mathbb{Z}^E$, and $s,t\in V$, computes with high probability the exact minimum cost maximum s-t flow in $\widetilde{O}((m+n^{1.5}\log(CW))\log(CW))$ work and $\widetilde{O}(\sqrt{n}\log(CW))$ depth, where $W:=||u||_\infty$ and $C:=||c||_\infty$.

This improves upon the work by Lee and Sidford [LS14], which runs in $\widetilde{O}(\sqrt{n})$ depth and $\widetilde{O}(m\sqrt{n})$ work.

Corollaries. We note that this also gives new results for some of the problems that reduce to min-cost flow. Firstly, we obtain an algorithm for max flow. Here, we again improve upon the result by Lee and Sidford [LS14]. We provide further related work in Section 1.2. For other special cases, improvements on [LS14] are already known or additional work-depth trade-offs are possible. We compare our results for the cases of bipartite maximum matching, negative-weight shortest paths and reachability.

Bipartite Maximum Matching. For the case of bipartite maximum matching, we obtain the following result.

Corollary 1.3. There exists an algorithm that, given a bipartite graph $G=(V,E)$, computes with high probability a maximum matching in $\widetilde{O}(m+n^{1.5})$ work and $\widetilde{O}(\sqrt{n})$ depth.

Again, this improves upon the $\widetilde{O}(m\sqrt{n})$-work and $\widetilde{O}(\sqrt{n})$-depth algoritm of Lee and Sidford [LS14]. An alternative remains the algorithm by Mulmuley, Vazirani, and Vazirani [MVV87], which has more work, $\widetilde{O}(n^\omega)$, but lower depth, $\widetilde{O}(1)$.

Negative-Weight Shortest Paths. We also obtain the following corollary for the negative-weight shortest path problem.

Corollary 1.4. There exists an algorithm that, given a directed graph $G=(V,E,w)$ with negative edge weights and a source $s\in V$, computes SSSP from s with $\widetilde{O}(m+n^{1.5})$ work and $\widetilde{O}(\sqrt{n})$ depth.

For dense graph ($m\geq n^{1.5}$), this improves with a $n^{o(1)}$ factor over the state of the art: Fisher et al. [FHL+25] provide an algorithm with $\widetilde{O}(m)$ work and $n^{0.5+o(1)}$ depth.

Reachability. Even for the special case of reachability, we obtain a similar improvement.

Corollary 1.5. There exists an algorithm that, given a directed graph $G=(V,E)$ and a source $s\in V$, computes reachability from s with $\widetilde{O}(m+n^{1.5})$ work and $\widetilde{O}(\sqrt{n})$ depth.

Again, for dense graphs, we improve upon the state of the art by a $n^{o(1)}$-factor: Liu, Jambulapati, and Sidford [JLS19] provide an algorithm with $\widetilde{O}(m)$ work and $n^{0.5+o(1)}$ depth. For other trade-offs between work and depth, see Section 1.2 and Table 1.

Our algorithm is based on an interior point method for linear programs. Previous work on near-linear work reachability is limited. There is a folklore algorithm consisting of a parallel BFS, giving $\widetilde{O}(n)$ depth. The more recent algorithm [Fin20, JLS19] are based on shortcuts. With our paper, we show a different approach for the case of dense graphs.

Expander Decompositions. Expander decompositions have become an essential tool in algorithm design. They have been used in the first almost-linear time algorithms for many fundamental questions, including but not limited to, max flow

[KLOS14], min-cost flow [CKL+22], electrical flow [ST04], Gomory-Hu trees [ALPS23], and vertex/edge connectivity [KT19, Li21, LNP+21]. In many such applications, a dynamic expander decomposition is used as a subroutine.

In the parallel setting, the first expander decompositions are given by Chang and Saranurak [CS19, CS20], with the state of the art given by Chen et al. [CMGS25] (see Section 2.3 for more details). We show how to use the latter to obtain a parallel batch-dynamic expander decomposition.

Lemma 1.6 (Informal version of Lemma 3.1). There exists a randomized data structure that maintains a parallel ϕ-expander decomposition[3], where a batch of E' updates uses $\widetilde{O}\left(1/\phi^4\right)$ depth and amortized $\widetilde{O}\left(|E'|/\phi^5\right)$ work.

1.2 Related Work

The Laplacian Paradigm. Many of the results on flow in the last two decades make use of techniques from the Laplacian paradigm. This line of research was initiated by the seminal work of Spielman and Teng [ST04], who showed that linear equations in the Laplacian matrix of a graph can be solved in near-linear time. The Laplacian matrix of a weighted graph G is defined as $L(G) := \mathrm{Deg}(G) - A(G)$, where $\mathrm{Deg}(G)$ is the diagonal weighted degree matrix: $\mathrm{Deg}(G)_{uu} := \sum_{(u,v)\in E} w(u,v)$ and $\mathrm{Deg}(G)_{uv} := 0$ for $u \neq v$, and $A(G)$ is the adjacency matrix: $A(G)_{uv} := w(u,v)$.

More efficient linear system solvers have been presented since [ST04], see, e.g., [KOSZ13, KMP14, KMP11, CKM+14, KS16, KLP+16]. The Laplacian paradigm has booked many successes, including but not limited to flow problems [Mad13, She13, KLOS14, Mad16, Pen16, CMSV17, LS20a, LS20b, AMV20], and bipartite matching [BLN+20]

Later, it was also shown that linear systems can be solved efficiently in the parallel setting [KM07, BGK+14, PS13, KX16, LPS15, SZ23]. Again, this has had many application, including but not limited to flow [LS14, AKL+24] and shortest paths [Li20, ASZ20].

Parallel Min-Cost Flow and Max Flow. The first algorithms for parallel max flow had $\widetilde{O}(mn)$ work and $\widetilde{O}(n^2)$ depth [SV82, R+90], or unspecified polynomial work and $\widetilde{O}(1)$ depth [KUW86]. The latter comes from a reduction to maximum matching and only holds for uncapacitated graphs. Recently, a (relatively) simple combinatorial algorithm was given by Peretz and Fischler [PF22], with $\widetilde{O}(mn)$ work and $\widetilde{O}(n)$ depth. A significantly faster algorithm was already provided by Lee and Sidford [LS14], who provided an algorithm with $\widetilde{O}(m\sqrt{n})$ work and $\widetilde{O}(\sqrt{n})$ depth by using an interior point method. This algorithm also holds for the more general problem of min-cost flow.

For approximate flow, faster algorithms are known. An early result by Serna and Spirakis [SS91] showed how to obtain a $(1+\varepsilon)$-approximation of max flow in $\widetilde{O}(\log(1/\varepsilon))$ depth, via a reduction to maximum matching. This algorithm comes with unspecified polynomial work. Within the Laplacian paradigm, the interior point method of Madry [Mad13] combined with the SDD solver of Peng and Spielman [PS13] gives an algorithm for max flow with $\widetilde{O}(m^{10/7})$ work and $\widetilde{O}(m^{3/7})$ depth at the cost of a $1/\mathrm{poly}(n)$ additive error. More recently, Agarwal et al. [AKL+24] gave a $(1+\varepsilon)$-approximation of max flow for undirected graphs with $\widetilde{O}(m\varepsilon^{-3})$ work and $\widetilde{O}(\varepsilon^{-3})$ depth.

The algorithm of Madry [Mad13] is based on an interior point method with $\widetilde{O}(m^{3/7})$ iterations. We note that other efficient sequential flow solvers based on interior point methods, e.g., [Mad16, CMSV17, KLS24], also have the potential to be implemented using a parallel SDD solver. However, these algorithms provide approximate solutions. Depending on which version of the min-cost or max flow LP they solve, this does not give an exact solution. To be precise, [LS14, BLL+21] can round their approximate solution directly to obtain an exact solution. On the other hand, [Mad16, CMSV17, KLS24] need a polynomial number of augmenting paths to derive an exact solution. In the parallel setting, computing an augmenting path currently uses $\widetilde{O}(\sqrt{n})$ depth, making it an expensive subroutine. Although we believe such results provide sublinear depth, we are not aware of any works specifying the exact trade-off. We note however, that our solver Theorem 1.2 is as fast as solving the augmenting path subroutine. Since this is called polynomially many times, our solver will have a polynomial advantage.

Iteration Count and Depth. Recently, Chen et al. [CKL+22] gave an almost-linear time algorithm for min-cost flow. This algorithm (and also its deterministic version [BCP+23]) uses $\Omega(m)$ iterations, which seems to render it hard to implement it efficiently in a parallel setting, as any intuitive implementation uses at least one round per iteration. The algorithms with lowest iteration counts have either $\widetilde{\Theta}(\sqrt{n})$ iterations [LS14, BLL+21], which is the lowest in terms of n, or $\widetilde{\Theta}(m^{1/3})$ iterations [KLS24, AMV20], which is lowest in terms of m. This means that without significant improvements in the iteration count of the max flow interior point methods, a depth of $\widetilde{\Theta}(\sqrt{n})$ is currently the best we can hope for in the parallel setting.

Bipartite Maximum Matching. Early on, it was shown that bipartite maximum matching can be computed with polylogarithmic depth [Lov79, KUW86]. Later, it was shown by Mulmuley, Vazirani, and Vazirani [MVV87] how to reduce the problem to matrix inversion and hence to matrix multiplication. This rendered an algorithm with $\widetilde{O}(n^\omega)$ work and $\widetilde{O}(1)$ depth. For sparser graphs, different work-depth tradeoffs were given by the reduction to max flow or min-cost flow. In particular, the algorithm of Lee and Sidford [LS14] gives $\widetilde{O}(m\sqrt{n})$ work and $\widetilde{O}(\sqrt{n})$ depth.

Negative Weight-Shortest Paths. Even in the sequential setting, the negative shortest path problem admitted no algorithms with less than $\widetilde{O}(m\sqrt{n})$ time until recently. Up until then, the scaling framework from Goldberg [Gab85, GT89, Gol95] provided the state of the art. In 2022, Bernstein,

[3] Here we decompose the edge set of G into ϕ-expanders, such that each vertex is in at most $\widetilde{O}(1)$ many expanders.

Work	Depth	
$m^{1+o(1)}$	$m^{1+o(1)}$	[CKL+22]
$\widetilde{O}(m+n^{1.5})$	$\widetilde{O}(m+n^{1.5})$	[BLL+21]
$\widetilde{O}(m^{\omega+4})$	$\widetilde{O}(1)$	[OS93]
$\widetilde{O}(m\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	[LS14]
$\widetilde{O}(m+n^{1.5})$	$\widetilde{O}(\sqrt{n})$	This paper

Work	Depth	
$O(m)$	$\widetilde{O}(n)$	Parallel BFS
$\widetilde{O}(n^{\omega})$	$\widetilde{O}(1)$	Parallel Trans. Closure
$\widetilde{O}(m+n\rho^2)$	$\widetilde{O}(n/\rho)$	[Spe97]
$\widetilde{O}(m\rho+\rho^4/n)$	$\widetilde{O}(n/\rho)$	[UY91]
$\widetilde{O}(m)$	$n^{0.5+o(1)}$	[JLS19]
$\widetilde{O}(m+n^{1.5})$	$\widetilde{O}(\sqrt{n})$	This paper

Table 1: On the left, an overview of parallel min-cost flow. On the right, an overview of the special case of parallel reachability. The latter is adapted from [JLS19].

Nanongkai, and Wulff-Nilsen provided an algorithm with $\widetilde{O}(m)$ time [BNW25].

In the parallel setting, a parallel version of Goldberg's algorithm was given by Cao, Fineman, and Russell [CFR22] with $\widetilde{O}(m\sqrt{n})$ work and $n^{0.5+o(1)}$ depth. Later, a parallel version of [BNW25] was given by Ashvinkumar et al. [ABC+24] with $m^{1+o(1)}$ work and $n^{1+o(1)}$ depth. This was further improved by Fisher et al. [FHL+25], who provide an algorithm with $\widetilde{O}(m)$ work and $n^{0.5+o(1)}$ depth.

Reachability. Two folklore algorithms for reachability are running a parallel breadth first search (BFS) or computing the transitive closure. The former has $O(m)$ work and $\widetilde{O}(n)$ depth, and the latter has $\widetilde{O}(n^{\omega})$ work and $\widetilde{O}(1)$ depth. Spencer [Spe97] and Ullman and Yannakakis [UY91] provide algorithms with parameterized work-depth trade-offs. However, for near-linear work, they do not improve on the $\widetilde{O}(n)$ depth of parallel BFS. More recently, the breakthrough paper of Fineman [Fin20] provides $\widetilde{O}(m)$ work and $\widetilde{O}(n^{2/3})$ depth. This paper uses a tool called hopsets or shortcuts. These are edges added to the graph that decrease the diameter of the graph, without impacting reachability. Building on these concepts, Liu, Jambulapati, and Sidford [JLS19] improved this to $\widetilde{O}(m)$ work and $n^{0.5+o(1)}$ depth. For an overview of these results, also see Table 1.

Distributed Min-Cost Flow. Exact min-cost flow has also been studied in the related CONGEST model [FGL+21, dV23] and the (broadcast) congested clique [FdV22, FdV23]. However, in these models, each node has infinite computing capacities. This means that the total work is not taken into account and hence those results are orthogonal to the contributions of this paper.

2 OVERVIEW

This overview consists of two main parts. In Section 2.2, we give a summary of previous work and show how the interior point method is set-up to obtain our main result, Theorem 1.2. We describe which subroutines are necessary. In particular, we show that the main technical ingredient is a parallel batch-dynamic expander decomposition. We describe how we obtain the latter in Section 2.3. Some necessary notation and definitions are provided in Section 2.1.

2.1 Notations & Definitions

We write $[n] := \{1, 2, ..., n\}$. We use the $\widetilde{O}(\cdot)$ notation to hide $\mathrm{poly}(\log \varepsilon^{-1}, \log n)$ factors, where n denotes the number of vertices.

Norms. We write $\|\cdot\|_p$ for the ℓ_p-norm, i.e., for any vector $v \in \mathbb{R}^d$, $\|v\|_p := (\sum_i |v_i|^p)^{1/p}$, $\|v\|_\infty = \max_i |v_i|$, and $\|v\|_0$ denotes the number of non-zero entries of v. Further, for a vector $\tau \in \mathbb{R}^d$, we define $\|v\|_\tau := \left(\sum_i \tau_i v_i^2\right)^{1/2}$ and the mixed norm $\|v\|_{\tau+\infty} := \|v\|_\infty + C\log(4m/n)\|v\|_\tau$ for a large constant C.

Matrix and Vector Operations. Given vectors $u, v \in \mathbb{R}^d$ for some d, we perform arithmetic operations $\cdot, +, -, /, \sqrt{\cdot}$ elementwise. For example, $(u \cdot v)_i = u_i \cdot v_i$ or $(\sqrt{v})_i = \sqrt{v_i}$. For a vector $v \in \mathbb{R}^d$ and a scalar $\alpha \in \mathbb{R}$, we let $(\alpha v)_i = \alpha v_i$ and $(v+\alpha)_i = v_i + \alpha$. Additionally, given a vector $v \in \mathbb{R}^m$, we use capitalized $\mathbf{V} \in \mathbb{R}^{m\times m}$ as the diagonal matrix whose diagonal entries are the elements of v, i.e., $\mathbf{V}_{i,i} = v_i$ for all $i \in [m]$. For a function $\phi : \mathbb{R}^m \to \mathbb{R}^m$ and a vector $\tau \in \mathbb{R}^m$, the diagonal matrices $\Phi(x)$ and $\mathbf{T}$ are defined analogously.

For $\varepsilon > 0$, we write $\mathbf{A} \approx_\varepsilon \mathbf{B}$ to denote matrix $\mathbf{A}$ being a $\exp(\pm\varepsilon)$-spectral approximation of $\mathbf{B}$. Similarly, we extend this notation for vectors, letting $u \approx_\varepsilon v$ if and only if $\exp(-\varepsilon)v_i \le u_i \le \exp(\varepsilon)v_i$ for all i. Observe that $\exp(\pm\varepsilon)$ is close to $(1\pm\varepsilon)$ for small $\varepsilon > 0$.

Leverage Scores. For a full-rank matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, let $\sigma(\mathbf{A}) \in \mathbb{R}^m$ denote its leverage scores, defined as $\sigma(\mathbf{A})_i \stackrel{\text{def}}{=} a_i^\top (\mathbf{A}^\top \mathbf{A})^{-1} a_i$ for each $i \in [m]$.

Expander Graphs. For an undirected graph $G = (V, E)$, we use $E(A, B)$ for two disjoint vertex sets A, B to denoted the edge set between A and B. We use $\deg_G(A)$ to denote $\sum_{v\in A} \deg_G(v)$ where $\deg_G(v)$ is the degree of v in G. We say G is a ϕ-expander if for every $S \subseteq V$ with $S \neq \emptyset, S \neq V$, we have

$$\frac{|E(S, V\backslash S)|}{\min(\deg_G(S), \deg_G(V\backslash S))} \ge \phi$$

In this overview, to simplify notations, when we say a graph is an expander, we mean a ϕ-expander for some $\phi = \frac{1}{\mathrm{polylog}(n)}$.

(Edge-partitioned) expander decomposition. For an undirected graph $G = (V, E)$, (edge-partitioned) ϕ-expander decomposition is a partition $E = E_1 \cup E_2 \ldots \cup E_z$ such that E_i is a ϕ-expander for every i (as a subgraph), and every vertex is contained in at most $\widetilde{O}(1/\phi)$ many different E_i's (as subgraphs).

We mostly care about the case when $\phi = 1/\text{polylog}(n)$, so to simplify notations, in this overview, when we say expander decomposition, we mean an edge-partitioned ϕ-expander decomposition for some $\phi = 1/\text{polylog}(n)$.

Note that the expander decomposition operates by ignoring edge directions and is therefore defined on undirected graphs. This simplification is appropriate because the decomposition is primarily used to construct the HeavyHitter data structure, which maintains edge weights and supports weight-based queries, rather than relying on edge directions.

Parallel batch-dynamic expander decomposition. The problem of parallel batch-dynamic expander decomposition asks for maintaining the expander decomposition of an undirected graph $G = (V, E)$ where G undergoes edge updates (both insertions and deletions). Moreover, the edge updates are given as batches: in each batch, a set of edges is given to the algorithm to be deleted (or inserted), and the algorithm needs to perform this update and give the expander decomposition after this batch update in small work (ideally proportional to the number of updated edges) and small depth (ideally $\text{polylog}(n)$).

Flows. A flow f on an undirected graph $G = (V, E)$ assigns non-negative real values to simple paths in G, meaning the total value of flow over this path. The size of the flow is the total value among all paths. A source (sink) demand vector Δ assigns a real value to each vertex. We say a flow routes a source demand Δ to a sink demand ∇ if at least $\Delta(v)$ flows are starting from v for every v and at most $\nabla(v)$ flows are ending at v for every v.

Note that the notion of flow is defined for undirected graphs in the context of the expander decomposition algorithm, where it is exclusively used.

2.2 A (Parallel) Interior Point Method

In this section we recap interior point methods for solving minimum cost flows. The interior point method presented here stems from previous work on sequential algorithms. We outline the framework and the computational tasks that must be solved in the parallel setting. Readers familiar with this framework may skip ahead to Section 2.3, where we outline our contributions, i.e., solving the necessary subroutines in a parallel setting.

Consider the linear programming definition of minimum cost flow: for an $m \times n$ incidence matrix $\mathbf{A} \in \{-1, 0, 1\}^{m \times n}$, edge-cost vector $c \in \mathbb{R}^m$, demand vector $b \in \mathbb{R}^n$, and edge capacities $u \in \mathbb{R}^m_{\geq 0}$, the following linear program models minimum-cost flows.

$$\min_{x \in \mathbb{R}^E} c^\top x \;\text{ subject to }\; \mathbf{A}^\top x = b,\; 0 \leq x \leq u$$

A common approach for solving linear programs are central path methods. These reduce solving linear programs to solving a sequence of linear systems. The idea is to define a potential function f_μ for $\mu \in \mathbb{R}_{>0}$ such as

$$f_\mu : \{x \mid \mathbf{A}^\top x = b\} \to \mathbb{R} \;\text{ where}$$
$$f_\mu(x) = c^\top x - \mu \cdot \sum_{i=1}^{m} \log x_i + \log(u_i - x_i). \tag{1}$$

Observe that the log-terms go to $-\infty$ as x_i approaches 0 or capacity u_i. Hence to minimize $f_\mu(x)$, we need that x stays far from the boundaries $0 \leq x_i \leq u$. However, as $\mu \to 0$, the first term $c^\top x$ starts to dominate, so minimizing $c^\top x$ becomes more important. In particular, for $\mu \to 0$, the minimizer x converges towards the optimal solution of the linear program. The curve of minimizers $c(\mu) = \text{argmin}_{\mathbf{A}^\top x = b} f_\mu(x)$, is referred to as "the central path" as it traces a path from the center of the feasible space towards the optimal solution of the linear program.

The idea of central path methods is to start with an initial point x that is the minimizer of f_μ for large μ, and then follow the central path by repeatedly decreasing μ and moving x closer to the new minimizer of f_μ via Newton steps.

Once μ is small enough (i.e., some $1/\text{poly}(m, \|u\|_\infty, \|c\|_\infty)$) then x is close to the optimal solution of the linear program. In particular for min-cost flow, where the optimal solution is guaranteed to be integral, we can simply round all coordinates of x_i to the nearest integer to obtain the optimal min-cost flow.

Solving min-cost flow in $\sqrt{n}$ depth. In [LS19], it was shown that for a slightly different choice of f_μ, the number of iterations is $\widetilde{O}(\sqrt{n})$. They use a variation of (1) where the log-terms are weighted by Lewis-weights $\tau \in \mathbb{R}^m_{\geq 0}$.

$$\begin{aligned} f_\mu(x) &= c^\top x + \mu \cdot \sum_{i=1}^{m} \tau_i \cdot \phi(x)_i \\ \phi(x)_i &= -\log(x_i) - \log(u_i - x_i), \\ \phi'(x)_i &= -\frac{1}{x_i} + \frac{1}{u_i - x_i}, \\ \phi''(x)_i &= \frac{1}{x_i^2} + \frac{1}{(u_i - x_i)^2} \\ \tau &= \sigma(\mathbf{T}^{1/2-1/p} \Phi''(x)^{-1/2} \mathbf{A}) + \frac{n}{m} \end{aligned} \tag{2}$$

Here the τ that satisfy the recursive equation (2) for $p = 1 - 1/(4 \log(4m/n))$ are referred to as ℓ_p Lewis-weights [CP15, LS19].

To measure the distance of x towards the central path, it is useful to introduce a variable $s \in \mathbb{R}^m$ of form $s = c - \mathbf{A}y$ for $y \in \mathbb{R}^n$. The optimality condition for $f_\mu(x)$ is existence of $y \in \mathbb{R}^n$ with $0 = \nabla f_\mu(x) - \mathbf{A}y$ (using that the gradient ∇f_μ must be orthogonal to the feasible space $\{x \mid \mathbf{A}^\top x = b\}$), and thus $0 = c + \mu\tau\phi'(x) - \mathbf{A}y = s + \mu\tau\phi'(x)$. So the distance to the central path can also be measured via the length of the vector $s + \mu\tau\phi'(x)$ rather than the value of $f_\mu(x)$. In [LS19], it was shown that for certain potential function Ψ, measuring the length of $s + \mu\tau\phi'(x)$, the following iterative steps allow convergence towards the optimal solution of the linear program in only

$\widetilde{O}(\sqrt{n})$ iterations.

$$
\begin{aligned}
&x \leftarrow x + \delta_x,\ s \leftarrow s + \delta_s \\
&\delta_x = \Phi''(x)^{-1/2} g \\
&\qquad - \mathbf{T}^{-1}\Phi''(x)^{-1}\mathbf{A}\ (\mathbf{A}^\top \mathbf{T}^{-1}\Phi''(x)^{-1}\mathbf{A})^{-1}\ \mathbf{A}^\top \Phi''(x)^{-1/2} g \\
&\delta_s = \mu \mathbf{A}\ (\mathbf{A}^\top \mathbf{T}^{-1}\Phi(x)^{-1}\mathbf{A})^{-1}\ \mathbf{A}^\top \Phi''(x)^{-1/2} g \\
&g = \nabla\Psi\left(\frac{s + \mu\tau\phi'(x)}{\mu\tau\sqrt{\phi''(x)}}\right)^{\flat(\tau)}
\end{aligned}
\tag{3}
$$

where we define for vectors $v, \tau \in \mathbb{R}^m$

$$v^{\flat(\tau)} := \operatorname*{argmax}_{\|w\|_{\tau+\infty} \le 1} \langle w, v \rangle.$$

Essentially, g is the largest step we can take in direction of the gradient $\nabla\Psi$ while the step is still bounded in the $\|\cdot\|_{\tau+\infty}$ norm.

Performing steps as in (3), then decreasing μ by some $1 - O(1/\sqrt{n})$) factor, and repeating, takes $\widetilde{O}(\sqrt{n})$ iterations to solve the linear program. In particular, it implies a $\widetilde{O}(\sqrt{n})$ depth algorithm for min-cost flow, but with large work to calculate the steps in each iteration. Thus the next task is to reduce the total amount of work without substantially increasing the depth.

Robust Interior Point Method. Observe that even when ignoring the time for calculation, just writing down the vectors δ_x, δ_s of (3) takes $\Theta(m)$ work per iteration. To reduce time per iteration, sequential work [CLS21, Bra20, LSZ19, JSWZ21, BLL+21, BLN+20] has shown that computing the Newton steps approximately suffices. These methods use only approximations of x, s, τ and the matrix inverse $(\mathbf{A}\mathbf{T}^{-1}\Phi''(x)^{-1}\mathbf{A})^{-1}$. Concretely, [BLL+21] showed the following method converges within $\widetilde{O}(\sqrt{n})$ iterations despite crude approximations (here $\varepsilon = O(1/\log m)$):

$$
\begin{aligned}
&\text{Pick } \overline{x} \approx_\varepsilon x,\ \overline{s} \approx_\varepsilon s,\ \overline{\tau} \approx_\varepsilon \tau(\Phi''(x)^{-1/2}\mathbf{A}),\ \overline{\mu} \approx_\varepsilon \mu \\
&g = -\gamma\nabla\Psi\left(\frac{\overline{s} + \overline{\mu\tau}\phi'(\overline{x})}{\overline{\mu\tau}\sqrt{\phi''(\overline{x})}}\right)^{\flat(\overline{\tau})} \\
&\mathbf{H} \approx_\varepsilon \mathbf{A}^\top \overline{\mathbf{T}}^{-1}\Phi''(x)^{-1}\mathbf{A} \text{ (spectral sparsifier)} \\
&\delta_c = \mathbf{H}^{-1}(\mathbf{A}^\top x - b),\ \delta_y = \mathbf{H}^{-1}\mathbf{A}^{\top\prime\prime}(x)^{-1/2} g \\
&\mathbf{R} = \text{random } m \times m \text{ diagonal matrix,} \\
&\mathbf{R}_{i,i} = \begin{cases} 1/p_i & \text{with probability } p_i \\ 0 & \text{else} \end{cases} \\
&\text{where } p_i \ge \min\left(1,\ \frac{m}{\sqrt{n}} \cdot \frac{((\overline{\mathbf{T}}\Phi''(\overline{x}))^{-1}\mathbf{A}(\delta_y + \delta_c))_i^2}{\|(\overline{\mathbf{T}}\Phi''(\overline{x}))^{-1}\mathbf{A}(\delta_y + \delta_c)\|_2^2} + \frac{1}{\sqrt{n}} + \overline{\tau}_i\right) \\
&\delta_x = \Phi''(x)^{-1/2} g - \mathbf{R}\overline{\mathbf{T}}^{-1}\Phi''(x)^{-1}\mathbf{A}(\delta_y + \delta_c),\ \delta_s = \mu\mathbf{A}\delta_y \\
&x \leftarrow x + \delta_x,\ s \leftarrow s + \delta_s,\ \mu \leftarrow \mu(1 - \widetilde{O}(1/\sqrt{n})) \\
&\text{Repeat from (4)}
\end{aligned}
\tag{4, 5}
$$

Unlike (3), here the additional term δ_c is needed, because by using the spectral approximation $\mathbf{H}$, we no longer have $\mathbf{A}^\top\delta_x = 0$. Thus $\mathbf{A}^\top(x + \delta_x) \neq b$ which is corrected with the δ_c step. To support fast calculation of $\mathbf{A}^\top x - b = \mathbf{A}x^{(\text{old})} + \mathbf{A}^\top\delta_x$, part of the vector δ_x is sparsified via the random matrix $\mathbf{R}$ in (5).

It is important for (4) that x, s, τ are not actually computed and only defined as reference point for the approximation $\overline{x}, \overline{s}, \overline{\tau}$. Only these approximate values are computed by the algorithm. This allows for (amortized) sublinear-work per iteration, because one can prove that the vectors $\overline{x}, \overline{s}, \overline{\tau}$ require only a few changed entries per iteration to stay valid approximations. So there is no need to recompute m entries of $\overline{x}, \overline{s}, \overline{\tau}$ from scratch in each iteration. By developing data structures tailored to this task of updating entries of $\overline{x}, \overline{s}, \overline{\tau}$, [BLL+21] obtained a sequential algorithm that solves min-cost flow in $\widetilde{O}(m + n^{1.5})$ time. However, their result does not imply an $\widetilde{O}(\sqrt{n})$-depth algorithm, because their data structures need up to $\Theta(m)$ depth in some iterations. Our main contribution is the development of low-depth equivalents of their data structures. The major bottlenecks we need to solve are as follows.

Sampling $\mathbf{R}$. Computing the sampling probabilities for $\mathbf{R}$ as in (5) would require a matrix vector product of form $\mathbf{W}\mathbf{A}h$ for some vector $h \in \mathbb{R}^n$ and $m \times m$ diagonal matrix $\mathbf{W}$. This calculation would require $O(m)$ time per iteration. However, let us assume for simplicity that $\mathbf{A}$ is incidence matrix of an expander graph and $\mathbf{W}_{i,i} = \mathbf{W}_{j,j}$ for all i, j. We will argue that for this case, the sampling task is simple.

Assume without loss of generality that $h \perp \mathbf{D}\mathbf{1}$ where $\mathbf{D} \in \mathbb{R}^{V \times V}$ is a diagonal matrix with $\mathbf{D}_{v,v} = \deg(v)$ (we can add a multiple of the all-1-vector to h to satisfy this. This does not change the task as $\mathbf{A}\mathbf{1} = 0$). Iterate over the vertices, and for each $v \in V$ sample each incident edge $\{u, v\}$ independently with probability proportional to $h_v^2 > |h_u - h_v|^2/2 = (\mathbf{A}h)_{uv}$. This sampling can be implemented to take time proportional to the number of returned edges, i.e., $\sum_{v \in V} 1 + \deg(v) \cdot h_v^2 \le \widetilde{O}(n + \|\mathbf{A}h\|_2^2)$ in expectation by Cheeger-inequality (which states $\phi_G^2/2 \le \lambda_2(\mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2})$, and thus bounds $\sum_{v \in V} \deg(v) \cdot h_v^2 = \|\mathbf{D}^{1/2}h\|_2 \le \widetilde{O}((\mathbf{D}^{1/2}h^\top)(\mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2})(\mathbf{D}^{1/2}h)) = \widetilde{O}(h^\top\mathbf{L}h) = \widetilde{O}(\|\mathbf{A}h\|_2^2)$ for $1/\text{polylog}(n)$ expander graph and $h \perp \mathbf{D}\mathbf{1}$, i.e., $\mathbf{D}^{-1/2}h$ is orthogonal to the (λ_1=0)-eigenspace of $\mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2}$).

Thus, if we can decompose the weighted graph represented by $\mathbf{W}\mathbf{A}$ into a collection of expanders such that each expander has (almost) uniform edge weights, then sampling can be done efficiently without computing $\mathbf{W}\mathbf{A}h$ in $O(m)$ time. Hence the main problem is to maintain an expander decomposition for the graph where edge weights are $w := (\overline{\tau}\phi''(\overline{x}))^{-1}$, and thus change from one iteration to the next. In previous work, this expander decomposition could not be done in low depth. We solve this issue by developing a parallel batch-dynamic expander decomposition that is efficient in the parallel setting, see Section 2.3.

Maintaining $\overline{s}, \overline{\tau}$. We also need to compute $\overline{s}$ and $\overline{\tau}$ in each iteration. Since they are m-dimensional vectors, we cannot afford to recompute them in each iteration. Instead, we aim to only update a few of their entries per iteration. We outline the idea here for $\overline{s}$ but it extends to $\overline{\tau}$ as well.

If we already have some approximation $\bar{s} \approx s$, and perform one more iteration $s \leftarrow s + \delta_s$, then $\bar{s}_i$ is still a good approximation of s_i if $(\delta_s)_i \ll s_i$ was sufficiently small. In particular, instead of computing the entire m-dimensional vector δ_s in each iteration, it suffices to only calculate information about entries $(\delta_s)_i$ where $(\delta_s)_i \gg \varepsilon \cdot s_i$ for some threshold ε.

We thus reduces the problem to a data structure task where, given a vector $\delta_y \in \mathbb{R}^n$, we must detect all indices i where $(\bar{S}^{-1}\delta_s)_i = (\bar{S}^{-1}\mathbf{A}\delta_y)_i > \varepsilon$. This too can be solved easily when incidence matrix $\bar{S}^{-1}\mathbf{A}$ represents an expander graph. E.g., it could be solved by repeatedly sampling indices proportional to $\bar{S}^{-1}\mathbf{A}\delta_y$, which reduces the task to the previously outlined sampling problem. In summary, for an efficient parallel algorithm for min-cost flow, all that is left is to develop a dynamic expander decomposition of low-depth and work.

2.3 Expander Decomposition

In this section, we provide an overview of our algorithm for parallel batch-dynamic expander decomposition (Section 3), i.e., we want to maintain an expander decomposition[4] of an undirected graph while this graph undergoes edge insertions and deletions in batches, and for each batch update of edge set E', we wish to maintain the expander decomposition in $\widetilde{O}(1)$ depth and amortized $\widetilde{O}(|E'|)$ work. See Lemma 3.1 for a detailed definition.

Reduction to decremental setting. It is known that in the sequential setting, fully dynamic expander decomposition (edges can be both inserted and deleted) can be reduced to two subroutines, (i) the static setting (compute an expander decomposition of an undirected graph once) and (ii) the decremental setting (edges can only be deleted). The reduction can be found in [BBG+22]. This reduction can be naturally implemented in the PRAM model under batch updates (see Section 3, Proof of Lemma 3.1 for more details). In a nutshell, we partition the edge set E into $O(\log n)$ edge sets $E_1, ..., E_{O(\log n)}$ where $|E_i| < 2^i$, and the algorithm will maintain expander decomposition for every E_i. When a batch of edges E' is inserted into the graph, the algorithm will find the smallest i such that $2^i > |E'| + \sum_{j \le i} |E_i|$ and use the static algorithm to recompute an expander decomposition of $\cup_{j \le i} E_i \cup E'$ and make it the new E_i. Notice that the static algorithm takes work proportional to the number of edges inserted. Thus, we only need to handle the static case and the decremental case.

For the static computation of expander decomposition, a recent paper [CMGS25] gives a near-linear time and polylogarithmic depth algorithm, so it is done. For the rest of this section, we focus on giving the decremental algorithm.

[4]As defined in Section 2.1, in this overview, for simplicity, we use expander decomposition to refer to decomposing the edge set of a graph into subgraphs where each subgraph is an expander, and expander refers to a ϕ-expander for some $\phi = 1/\text{polylog}(n)$. It is guaranteed that each vertex appears in at most $\widetilde{O}(1)$ subgraphs.

Reduction to expander pruning. It is known from [SW19] that decremental expander decomposition can be reduced to the following expander pruning task.

Input. An expander graph H.

Updates. An online sequence of batch deletions $E_1, ..., E_k$. Denote $H_i = H - \cup_{j \le i} E_i$ the graph after the i-th deletion.

Outputs. After each update, a set of pruned vertices V_i such that after deleting $\cup_{j \le i} V_i$ from H_i (and all their adjacent edges), the graph becomes an expander again. It is required that the number of adjacent edges in V_i in H_i is at most $\widetilde{O}(|E_i|)$, i.e., at most $\widetilde{O}(|E_i|)$ edges can be pruned out after the i-th update.

We also wish to perform the i-th update in $\widetilde{O}(|E_i|)$ amortized work and $\widetilde{O}(1)$ depth.

In a nutshell, the reduction works as follows. In the decremental setting, i.e., deleting an edge set E' from an expander decomposition, we 'prune out' sets of vertices (and adjacent edges) from each decomposed expander of total size $\widetilde{O}(|E'|)$ so that the remaining subgraphs are still expanders, where the 'pruned out' edges are inserted into the graph again which is handled by edge insertion in the previous paragraph (remember that edge insertion can be handled by static computation of expander decomposition).

For the rest of this section, we only need to consider parallel expander pruning, the formal definition is in Lemma 3.3.

Trimming. In [SW19], expander pruning is based on modifying a procedure called trimming, which can be viewed as a static version of expander pruning: a set of edges E' is deleted from an expander H, and we want to delete another set of edges of size $\widetilde{O}(|E'|)$[5] so that the remaining subgraph is still an expander. Notice the difference: expander pruning requires many batches of deletions where expander trimming only requires one batch of deletions.

However, the trimming algorithm in [SW19] is highly sequential. In a more recent paper [CMGS25], a parallel trimming algorithm is provided, with $\widetilde{O}(m)$ work and $\widetilde{O}(1)$ depth. The challenge is to use it to get expander pruning, i.e., instead of just deleting one batch of edges from H, it needs to support deleting an online sequence of edge sets $E_1, E_2, ..., E_k$, where each deletion only uses $\widetilde{O}(|E_i|)$ work and $\widetilde{O}(1)$ depth. This is not so obvious to do as the algorithmic ideas from [CMGS25] are quite different from [SW19].

Algorithm overview of [CMGS25]. We first give a brief overview of the trimming algorithm in [CMGS25]. Given an expander H and an edge set E' to be deleted from H, it outputs an edge set of size $\widetilde{O}(|E'|)$ deletions such that after these deletions H is an expander again. The algorithm uses $\widetilde{O}(m)$ work and $\widetilde{O}(1)$ depth.

The intuition of the algorithm comes from a structural lemma (see Lemma 3.9), for which we first define a certificate. We call a flow f supported by $H - E'$ a certificate if: (i) f routes the source demand $\Delta(u) = \widetilde{O}\left(\deg_{E'}(u)\right)$[6], i.e.,

[5]In the previous paragraph, we mentioned deleting a set of vertices, but that is equivalent to deleting all the adjacent edges and for convenience here we write deleting a set of edges.

[6]Here we use $\deg_{E'}(u)$ for the edges in E' that are adjacent to u.

each edge $(u,v) \in E'$ contributes roughly $\widetilde{O}(1)$ source demands on both (u,v), (ii) f routes the sink demand $\nabla(u) = \deg_H(u)/\text{polylog}(n)$. Lemma 3.9 shows that if a certificate exists, then $H - E'$ is an expander.

A natural idea is to solve a max flow problem to try to find a certificate in case $H - E'$ is already an expander and we do not need to delete any further. One worry is that we might not be able to solve the max flow problem in $\widetilde{O}(m)$ work and $\widetilde{O}(1)$ depth. However, [CMGS25] shows that we do not need to solve the flow problem perfectly but just need to use the push relabel algorithm up to $\widetilde{O}(1)$ layers. After which, either we route all the demand, which certifies that $H - E'$ is an expander, so we are done, or we can find a set of edges to be deleted from $H - E'$, denoted by E'', of size $\widetilde{O}(|E'|)$. It is guaranteed that if we substitute E' by $E' \cup E''$, we can set up the flow problem again and the source demand size reduces by a constant factor. Thus, $\widetilde{O}(1)$ iterations suffice to finally find a certificate, and the remaining graph is an expander. The flow algorithm is described in detail in Section 3.2.

A better analysis of the algorithm. Although in [CMGS25], the stated work is $\widetilde{O}(m)$, we can actually give a better analysis to make the work $\widetilde{O}(|E'|)$. The intuition is that the sink demand for every vertex u is $\deg_H(u)/\text{polylog}(u)$, so the flow cannot go out of u unless roughly $\deg_H(u)$ amount of flow is absorbed by u. In other words, if an edge (u,v) is in the support of the flow, then either u or v is saturated so that we can charge the size of the support by the total source demand size, which is $\widetilde{O}(|E'|)$. The algorithm has work proportional to the support of the flow as it is using push-relabel.

Supporting decremental updates. Now, we explain how we convert the trimming algorithm into an pruning algorithm supporting many batches of edge deletions. Suppose that after the batch of edge deletions E_0 arrives, we call the trimming algorithm on H, E_0 and get a set $E_0' = \widetilde{O}(|E_0|)$ such that $H - E_0 - E_0'$ is an expander. A natural idea is to substitute H by $H_1 := H - E_0 - E_0'$ and run the algorithm again for H_1, E_1. However, there is a technical issue that makes this simple idea wrong: the trimming algorithm has an inherent property that the expansion of H_1 decreases by a constant factor compared to H, which means after k updates, the expansion becomes exponentially smaller in terms of k. To avoid this, instead of replacing H by H_1 and feeding it into the trimming algorithm, we will try to run the trimming algorithm again on $H, E_0 \cup E_1$. This results in high work since the work for the second update is $\widetilde{O}(|E_0 \cup E_1|)$ instead of $\widetilde{O}(|E_1|)$. We resolve this in a similar way to [SW19], i.e., by reusing the flow information from the first trimming algorithm on H, E_0 so that we only need to route the flow contributed by E_1. Let us call the flow instance routing the contribution of E_i by f_i. The certificate after deleting $E_0, ..., E_i$ from H will be $f_0 + ... + f_i$.

However, this idea has an additional issue in the parallel setting. The parallel trimming algorithm by [CMGS25] has good parallel work and depth mainly because each time the sink demand for flow f_i is relatively large, i.e., at least $\deg(u)/\text{polylog}(n)$. That means that the flow cannot go out from u unless it gets absorbed by an amount proportional to the degree of u. Also, remember that we compute each f_i using the idea from [CMGS25], which means f_i must have sink demand at least $\deg(u)/\text{polylog}(n)$ for every i. Thus, the total flow summing over all f_i is polynomially larger than $\deg(u)$ if the number of updates are polynomial. This violates the definition of a certificate flow. In other words: we can only support the batch updates up to $\text{polylog}(n)$ times.

In order to support an arbitrary number of batch updates, we prove a lemma of batch number boosting (see Lemma 3.5 for more details). Similar ideas appear in [NSW17, JS21]. The boosting is simple: every time the algorithm undergoes 2^i batch updates, it rolls back to the beginning and combines all the 2^i updates into a single batch update, and does the 2^i to 2^{i+1} updates using similar rolling back techniques. Finally, we can support polynomially many decremental batch updates.

Vertex Decomposition. Although we presented our algorithm for maintaining an (edge-partitioned) expander decomposition, the same algorithm should work for the more prevalent vertex-partitioned expander decomposition, where the vertex set is partitioned into $V_1, ..., V_z$ such that each induced subgraph is an expander, and there are at most $\widetilde{O}(1/\phi) \cdot m$ inter-cluster edges. This is because expander pruning will give a pruned vertex set instead of and edge set, and all the arguments above should work. Note that since our result specifically concerns edge-partitioned decompositions, the vertex-partitioned variant is not directly relevant to our main result.

3 PARALLEL BATCH-DYNAMIC EXPANDER DECOMPOSITION

In this section, we will prove the following lemma. It is a parallel version of Theorem 4.3 in [BBG+22].

Lemma 3.1. There exists a randomized data structure against an adaptive adversary that, given an undirected graph $G = (V, E)$ (initially empty) and $\phi < 1/\log^C n$ for sufficiently large constant C, maintains subgraphs $G_1, ..., G_t$ of G. It is guaranteed that $G_1, ..., G_t$ partitions the edge set of G, G_i is a ϕ-expander for every i with high probability, and $\sum_i |V(G_i)| = \widetilde{O}(n)$. The data structure supports batch updates, i.e., given a set of edges E' to be deleted or added to G, the update uses $\widetilde{O}(1/\phi^4)$ depth and amortized $\widetilde{O}(|E'|/\phi^5)$ work.

Notice that by following the convention, expander decomposition means decomposing the vertex set instead of the edge set, see the theorem below.

Theorem 3.2 (Parallel Expander Decomposition, Theorem 1 of [CMGS25]). Given a graph $G = (V, E)$ of m edges and a parameter $\phi \in (0, 1)$, there is a randomized parallel algorithm that with high probability finds a partition of V into ϕ-expanders $V_1, .., V_k$ such that $\sum_{i<j} |E_G(V_i, V_j)| = \widetilde{O}(\phi m)$. The total work of the algorithm is $\widetilde{O}(m/\phi^2)$ with depth $\widetilde{O}(1/\phi^4)$.

By repeatedly applying Theorem 3.2, we can get a static algorithm for Lemma 3.1. To get the dynamic version, we

need the following parallel expander pruning lemma, it is the parallel version of Theorem 3.1 in [SW19].

Lemma 3.3 (Parallel expander pruning). There is a deterministic data structure that, given an undirected graph $G = (V, E)$ which is a ϕ-expander, and an online sequence of edge set deletions $E_1, E_2, ..., E_k$, maintains a pruned set $P \subseteq V$ such that the following property holds. Let G_i and P_i be the graph G and the set P after the i-th deletion. We have, for all i,

(1) $P_0 = \emptyset$ and $P_i \subseteq P_{i+1}$,
(2) $\mathbf{deg}(P_i) \leq \widetilde{O}\left(\sum_{j<i} |E_j|/\phi\right)$, and
(3) $G_i[V - P_i]$ is a $\phi/(6 \log n)$-expander.

There is no initialization required. Assuming the graph is given by an adjacency list, the work and depth for updating each P_i is $\widetilde{O}\left(|E_i|/\phi^4\right)$ and $\widetilde{O}\left(1/\phi^3\right)$ for every i.

Proving Lemma 3.3 is the main technical contribution of this section. The algorithm in [SW19] is highly sequential. We will adjust the parallel trimming algorithm in [CMGS25] to get our result, see Section 3.1. For completeness, let us first see how we can combine Theorem 3.2 and Lemma 3.3 to get Lemma 3.1, by following the same algorithm as in [BBG+22].

To prove Lemma 3.1, let us first prove the static version:

Lemma 3.4. There exists a randomized algorithm that given an undirected graph $G = (V, E)$ and $\phi < 1/\log^C n$ for sufficiently large constant C, outputs subgraphs of G denoted by $G_1, ..., G_t$. It is guaranteed that $G_1, ..., G_t$ partitions the edge set of G, G_i is a ϕ-expander for every i with high probability, and $\sum_i |V(G_i)| = \widetilde{O}(n)$. The algorithm runs in $\widetilde{O}\left(m/\phi^2\right)$ work and $\widetilde{O}\left(1/\phi^4\right)$ depth.

Proof. We use Theorem 3.2 on G and ϕ to get $V_1, ..., V_k$, and let $G[V_i]$ for every i to be the output subgraphs. Then we delete the edges in $G[V_i]$ for every i from the graph, and we are guaranteed that the remaining number of edges is at most $\widetilde{O}(\phi m) < m/2$, according to Theorem 3.2 and $\phi < 1/\log^C m$ for sufficiently large C. We repeat the same procedure on the remaining graph, after $O(\log n)$ iterations the graph will be empty and we have successfully partitioned the edge set into ϕ-expanders. Moreover, each vertex is contained in at most $O(\log n)$ many subgraphs since in each iteration, the vertex sets of the subgraphs generated in this iteration are disjoint. □

Proof of Lemma 3.1. The data structure maintains $O(\log n)$ graphs $G_1, G_2, ...$ so that they are subgraphs of the dynamic graph G, their union is equal to G, and $|E(G_i)| \leq 2^i$. For each G_i, we will maintain a partition of the edge set of G_i into expanders $G_{i,1}, .., G_{i,k_i}$ such that each vertex appears in $O(\log n)$ many of them. If we can maintain that, we get the desired partition of the whole graph G.

When an edge set is inserted into the graph, we will first insert them into G_1 and perform some operations on G_1; when an edge set is deleted, we will split the deleted edge set into $O(\log n)$ sets and delete the corresponding edge set in each G_i. In the next two paragraphs, we describe how we handle the insertion or deletion of an edge set from a specific G_i.

If some set of edges I is to be inserted into G_i, then we consider two cases: (i) If $|E(G_i) \cup I| > 2^i$, then we set G_i to be an empty graph and insert $E(G_i) \cup I$ into G_{i+1}. (ii) If on the other hand $|E(G_i) \cup I| \leq 2^i$, then we perform the algorithm of Lemma 3.4 on G_i to obtain a partition of G_i into $(6 \log n\phi)$-expanders (which is also a ϕ-expander), and re-initialized the pruning algorithm for each partitioned subgraph.

If some set of edges I is to be deleted from G_i, we use the pruning algorithm of Lemma 3.3 on each of the $(6 \log n\phi)$-expander of the partitioned subgraphs of G_i. In that way, we delete some nodes from G_i, along with their adjacent edges. For all of those edges, we insert them into G_1 again. Notice that we can run the pruning algorithm on G_i is because we can guarantee that there will only be deletions on G_i without insertions: once an insertion happens in G_i, according to the last paragraph, either the whole G_i will be renewed into another partition of expanders, or will be deleted.

The correctness and depth of each update follow from Lemmas 3.3 and 3.4 and the definition of the algorithm. Next, we analyze the amortized work.

For each G_i, we define a life-time of G_i to start from one insertion into G_i and to end at the next insertion into G_i. We first show that the number of edge updates during a life-time of G_i is at least $2^i/(\phi \log^C n)$ for some sufficiently large constant C: if G_i gets some edges to be inserted, that must be from the edges in G_{i-1}, when G_{i-1} becomes larger than 2^{i-1} according to the definition of insertion. Analogously, this is due to G_{i-2}, down to G_1. So all G_1 to $G_{2^{i-1}}$ must be empty for each insertion to G_i, thus, between two insertions, at least Ω^{2^i} edges are inserted. These insertions can be from a direct insertion to G or from the pruning procedure, which according to Lemma 3.3 must correspond to at least $\widetilde{O}(1/\phi)$ multiplied by the number of deleted edges. Thus, the number of edge update in a life-time is $2^i/(\phi \log^C n)$. Next, we show that the work done during a life-time is small: it corresponds to one call to Lemma 3.4 on 2^i edges or simply deleting these edges, and then multiple calls to Lemma 3.3 which together cost at most $\widetilde{O}\left(2^i/\phi^4\right)$ work. Thus, the amortized work is at most $\widetilde{O}\left(1/\phi^5\right)$. □

3.1 Parallel Expander Pruning (Proof of Lemma 3.3)

Later we will show that we can only get expander pruning for $\widetilde{O}(1)$ batch updates at most, instead of supporting an arbitrary number of updates. Luckily, this is not a problem according to the following batch number boosting lemma. Similar ideas were used in [NSW17, JS21].

First we define some notation. A data structure (decremental, incremental, or fully dynamic) requires preprocessing, then can undergo a bunch of batch updates where each update is guaranteed to have some complexity, in our case work and depth. The data structure has batch number b if it can only support b batch updates, after which the data structure cannot support any updates.

Lemma 3.5 (Batch number boosting). Let G be a graph undergoing batch updates (decremental, incremental, or fully dynamic). Assume there is a data structure $\mathfrak{D}$ maintaining some graph properties of G with batch number b, preprocessing time t_{pre}, amortized update work w_{amor}, update depth d. Here $b, t_{\text{pre}}, w_{\text{amor}}$ and d are functions that map the upper bounds of some graph measures throughout the update, e.g. maximum number of edges, to non-negative numbers.

Then, for an arbitrary function $\overline{b}$, there is a data structure (decremental, incremental, or fully dynamic, correspondingly) undergoing batch updates with batch number $\overline{b}$, preprocessing time t_{pre}, amortized update work $O(b \cdot (\overline{b})^{1/b} \cdot w_{\text{amor}})$ and update depth $O(d \cdot b + \log_b \overline{b})$, maintaining the same graph properties of G as $\mathfrak{D}$.

Proof. The case for $\overline{b} \le b$ is trivial. Let us consider the instance where $\overline{b} > b$. The data structure first initializes the data structure $\mathfrak{D}$. When we update the data structure $\mathfrak{D}$ we will memorize our updates so that in the future we can make an operation called rollback, which will restore the data structure to a previous state where the updates have not been made.

Define $D_k = (\overline{b})^{k/b}$. When the i-th batch update arrives, the data structure checks for every integer $k > 0$ if there exist integers a and b such that $i = a \cdot D_k + b \cdot D_{k-1}$ with $b < D_1 = D_k / D_{k-1}$ or not. It finds the largest such $k > 0$ (it must exist since when $k = 1$, $D_{k-1} = 1$ and a, b are simply the module of D_1) and rollback the data structure to the point where the $a \cdot D_k$-th update has just been made. Then it updates the batch combining all the updates from $a \cdot D_k + 1$ to i.

Correctness. Notice that after the arriving of the i-th update, the data structure maintains the correct $\mathfrak{D}$ which undergoes the updates combined by all the 1 to i-th updates, by induction on the correctness. Moreover, $\mathfrak{D}$ has always been updated by at most b batches: according to the definition of the data structure, if a batch update is generated by rolling back to the $a \cdot D_k$-th update, then the previous batch update must be rolling back to the $a' \cdot D_{k+1}$ for some a', which means the batch size goes up by D_1 at each time, so that the maximum batch number is at most $\log_{D_1}(\overline{b}) = b$.

Complexity. Finding the k and finding a, b costs $O(\log_b \overline{b})$ work and $O(\log_b \overline{b})$ depth. Then for the largest k, rolling back the data structure costs depth $O(d \cdot b)$ since as we previously proved, the batch number for the data structure is at most b and rolling back one batch costs d depth as it requires to reverse all the operations made by $\mathfrak{D}$. Now we show the amortized work. Notice that the rolling back procedure can be charged to the update procedure, so we only need to analyze the updates. We only need to prove that for every i, the i-th update batch gets involved in at most $O(b \cdot (\overline{b})^{1/b})$ many batch updates for $\mathfrak{D}$. For that, we only need to prove that for every k (at most b different k's), it is involved in at most $O(\overline{b})^{1/b})$ updates. Now fix k and let a be the largest integer such that $a \cdot D_k < i$. The i-th update can only be involved in the batch update from $a \cdot D_k + 1$ to $a \cdot D_k + b \cdot D_{k-1}$ for $b < D_1 = (\overline{b})^{1/b}$, so we are done. □

Now we give the data structure for a small number of batch updates, captured by the following lemma.

Lemma 3.6. There is a deterministic data structure that, given an undirected graph $G = (V, E)$ which is a ϕ-expander, and an online sequence of edge set deletions $E_1, E_2, ..., E_b$ with $b < (\log n)/2$, maintains a pruned set $P \subseteq V$ such that the following properties holds. Let G_i and P_i be the graph G and the set P after the i-th deletion. We have, for all i,

(1) $P_0 = \emptyset$ and $P_i \subseteq P_{i+1}$,
(2) $\mathbf{deg}(P_i) \le \widetilde{O}\left(\sum_{j<i} |E_j|/\phi\right)$, and
(3) $G_i[V - P_i]$ is a $\phi/6 \log n$-expander.

There is no initialization required. Assuming the graph is given by an adjacency list, the work and depth for updating each P_i is $\widetilde{O}\left(|E_i|/\phi^4\right)$ and $\widetilde{O}\left(1/\phi^3\right)$ for every i.

Our proof of Lemma 3.6 will make use of the following Lemma 3.7, proven in Section 3.3, which is adapted from the trimming algorithm in [CMGS25].

Lemma 3.7. The algorithm Trimming$(G = (V, E), A, \phi)$ with inputs

(1) a graph $G = (V, E)$ accessed by adjacency list,
(2) a set $A \subseteq V$ accessed by identity query, i.e., given a vertex, it answers whether it is in A or not, a parameter $\phi \in \mathbb{R}_{\ge 0}$,
(3) explicitly given $E(A, V \setminus A)$, such that $|E(A, V \setminus A)| \le \phi \cdot m$,

outputs explicitly the set $V - A'$ and a flow $\boldsymbol{f}$ with non-zero entries explicitly given, such that

(1) $\boldsymbol{f}$ restricted in A' routes source $\frac{2}{\phi}(\mathbf{deg}_G(v) - \mathbf{deg}_{G[A']}(v))$ to sinks $\nabla(v) \le \mathbf{deg}_G(v)/\log n$,
(2) $\mathbf{deg}_G(A - A') = \widetilde{O}\left(\frac{1}{\phi}\right)|E(A, V \setminus A)|$,

in deterministic $\widetilde{O}\left(|E(A, V \setminus A)|/\phi^4\right)$ work and $\widetilde{O}\left(1/\phi^3\right)$ depth.

Proof of Lemma 3.6. Notice that once we have $\sum_{j<i} |E_i| > \phi \cdot m/\log n$, we can let $P_i = V$ and we are done. So throughout the algorithm, we will assume $\sum_{j<i} |E_i| \le \phi \cdot m/\log n$.

Recall that G_i and P_i are the graph G and the set P after the i-th deletion. Define $V_i := V - P_i$. When E_i is deleted, the data structure sets up the virtual graph G'_i by starting at the induced subgraph $G_{i-1}[V_{i-1}]$, and inserting a node in the middle of each edge $(u, v) \in E_i \cap E(G_{i-1}[V_{i-1}])$, i.e., it builds a new node e with edges $(u, e), (e, v)$ and deletes the original edge (u, v). Intuitively, this step is to ensure that $G'_i[V_{i-1}]$ does not contain any edges in E_i, so $G'_i[V_{i-1}]$ is the same as $G_i[V_{i-1}]$, the difference is that $G'_i[V_{i-1}]$ has more adjacent edges in G'_i than $G_i[V_{i-1}]$.

Now we can apply Lemma 3.7. We can call Trimming on G'_i, V_{i-1}, ϕ, which satisfies all input requirements: the graph G'_i is given as an adjacency list as we know E_i explicitly; A is accessed by identity query; and $E_{G'_i}(A, V \setminus A)$ is explicitly given as we know explicitly E_i (more specifically, $E_{G'_i}(A, V \setminus A)$ contains all two-edges split by inserting a node in the middle of each edge in $E_i \cap E(G_{i-1}[V_{i-1}])$). The reader can now see

more intuition on why we define G_i' in the above way: the additional adjacent edges of $G_i'[V - P_{i-1}]$ will contribute to the source capacity of $\boldsymbol{f}$.

From Lemma 3.7, we get the set $V_{i-1} - A'$ and the flow $\boldsymbol{f}$. We let $P_i = P_{i-1} \cup (V_{i-1} - A')$. The first point of Lemma 3.3 follows by the definition. The second point of Lemma 3.3 follows by the following argument: $\mathbf{deg}_G(P_i)$ contains two parts. One part is from the deleted edges, which is bounded by $\sum_{j<i} |E_j|$. The other part is from increasing by $|\mathbf{deg}_{G_i'}(V_{i-1} - A')|$ for each run of Trimming, which according to Lemma 3.7 is bounded by $\widetilde{O}\left(|E_{G_i}(V_{i-1}, V(G_i) \setminus V_{i-1})|/\phi\right) \leq \widetilde{O}\left(|E_i|/\phi\right)$ according to the definition of G_i.

The complexity follows as well since

$$\widetilde{O}\left(|E_{G_i}(V_{i-1}, V(G_i) \setminus V_{i-1})|\right) = \widetilde{O}\left(|E_i|\right).$$

It remains to prove that $G_i[V - P_i]$ is a $\phi/6\log n$-expander. We will first prove the following lemma.

Lemma 3.8. For every $i < (\log n)/2$, there exists a flow $\boldsymbol{f}_i$ supported by the graph $G_i[V_i]$ that routes the source $\frac{2}{\phi}(\mathbf{deg}_G(v) - \mathbf{deg}_{G_i[V_i]}(v))$ to sinks $\nabla(v) \leq 2i\mathbf{deg}_G(v)/\log n$, with the capacity of each edge bounded by $2i/\phi$.

Proof. We prove it by induction on i. Initially, when $i = 0$, the source vector is zero because $V_0 = V$ and $G_i = G$, so we are done. Now suppose we have a flow $\boldsymbol{f}_{i-1}$, we will construct the flow $\boldsymbol{f}_i$ as follows. The flow $\boldsymbol{f}_{i-1}$ is supported by $G_{i-1}[V_{i-1}]$, we first restrict it to $G_i'[V_i]$, which is the same as $G_i[V_i]$. After this restriction, for every edge (u, v) in $G_{i-1}[V_{i-1}]$ that goes from V_i to $V_{i-1} - V_i$, u will receive at most $2/\phi$ many redundant flows since flow of $\boldsymbol{f}_{i-1}$ is cut out here. Moreover, for every edge (u, v) in G that $u \in V_i$, u gets an extra source capacity from $\boldsymbol{f}_i$ of $2/\phi$ compared to $\boldsymbol{f}_{i-1}$ if either $(u, v) \in E_i$ (newly deleted edge) or $v \in V_{i-1} - V_i$ and $(u, v) \in E(G_{i-1})$. The idea is to use the flow $\boldsymbol{f}$ returned by Trimming in the i-th round to route all the new demand compared to $\boldsymbol{f}_{i-1}$: consider the flow $\boldsymbol{f} + \boldsymbol{f}$, each $\boldsymbol{f}$ routes the demand $\frac{2}{\phi} \cdot (\mathbf{deg}_{G_i'} - \mathbf{deg}_{G_i'[V-P_i]})$ according to Lemma 3.7, so the first $\boldsymbol{f}$ can route the cut-off demand of $\boldsymbol{f}_{i-1}$ back into V_i (remember that the boundary of V_i in G_i' contains the same number of edge as the boundary of G_{i-1} although some edges are deleted by E_i, because we put a node in the middle of that edge to construct G_i'), the second $\boldsymbol{f}$ can route the extra source capacity for edges (u, v) with $u \in V_i$ and either $(u, v) \in E_i$, in which case it contributes to $\mathbf{deg}_{G_i'} - \mathbf{deg}_{G_i'[V-P_i]}$ according to the definition of G_i', or $v \in V_{i-1} - V_i$ and $(u, v) \in E(G_{i-1})$.

Notice that the sink capacity of every node increases by $2\mathbf{deg}_G(v)\log n$ every round, and the edge capacity increases by $2/\phi$ every round, so the lemma follows. □

The following lemma shows that the flow in Lemma 3.8 certifies that $G_i[V_i]$ is an expander. It is similar to Proposition 3.2 of [SW19]. For its formulation, we say that $G' \subseteq G$ is a ϕ-nearly expander if for all $A' \subseteq V(G')$ with $\mathbf{deg}_G(A') \leq \mathbf{deg}_G(G)/2$ we have $|E_G(A', V(G) - A')| \geq \mathbf{deg}_G(A')$.

Lemma 3.9. Given a graph $G = (V, E)$ and a subgraph $G' \subseteq G$ such that G' is a ϕ-nearly expander and there exists a flow f supported on the graph G' that routes source $\Delta(v) \stackrel{\text{def}}{=} 2/\phi \cdot (\mathbf{deg}_G(v) - \mathbf{deg}_{G'}(v))$ to sinks $\nabla(v) \stackrel{\text{def}}{=} \mathbf{deg}_G(v)$ for $v \in V(G')$ with uniform edge capacity $2\log n/\phi$. Then G' is a $\frac{\phi}{6\log n}$-expander.

Proof. If G' is not a $\frac{\phi}{6\log n}$-expander, then there exists $A' \subseteq V(G')$, such that more than $1 - \frac{\phi}{3\log n}$ fraction of the edges in $E_G(A', V \setminus A')$ are not in G', in which case those edges contribute $\frac{1.9}{\phi}\mathbf{deg}_G(A') \cdot \phi$ many flows to the source capacity of f since G' is a ϕ-nearly expander. Moreover, the sink capacity of all nodes in A' is at most $\mathbf{deg}_G(A')$, so at least $0.9\mathbf{deg}_G(A')$ capacity of flow should go out of A' to $V(G') - A'$. However, the capacity of all the edges in $E_{G'}(A', V(G') - A')$ is at most $\frac{\phi}{3\log n} \cdot \mathbf{deg}_G(A') \cdot 2\log n/\phi < 0.9\mathbf{deg}_G(A')$, a contradiction. □

By combining Lemma 3.8 and Lemma 3.9, we get that $G_i[V_i]$ is always a $\phi/(6\log n)$-expander. □

Lemma 3.3 follows from combining Lemmas 3.5 and 3.6: we let the data structure $\mathfrak{D}$ be the data structure of Lem. 3.6, which has batch number $b = \log n/2$, then we get a data structure of batch size m without losing the update work and depth by a polylog factor. Batch size m suffices for Lemma 3.3 since after m updates we can simply set $P_i = V$.

3.2 Parallel Unit Flow

Before proving the trimming lemma, Lemma 3.7, we need an important subroutine, see the following lemma.

Algorithm 1: ParallelUnitFlow$(G, \boldsymbol{c}, \Delta, \nabla, h)$

$\boldsymbol{f}_0 \leftarrow \mathbf{0}$; $\nabla_0 \leftarrow \mathbf{0}$: $\forall v \in V : l(v) = 0$
for $i = 1, \ldots, 8 \cdot \log_2 n$ do
  $x_i \leftarrow \sum_{v \in V : l(v) \neq h+1} \mathbf{ex}^G_{\boldsymbol{f}_{i-1}, \Delta, \nabla_{i-1}}(v)$ /* Non settled excess */
  $\boldsymbol{f}_i' \leftarrow \mathbf{0}$; $\nabla_i \leftarrow \frac{1}{8\log_2 n}\nabla$
  while $\sum_{v \in V : l(v) \neq h+1} \mathbf{ex}^G_{\boldsymbol{f}_{i-1} + \boldsymbol{f}_i', \Delta, \nabla_i}(v) \geq x_i/2$ do
    $(\boldsymbol{f}_i', l) \leftarrow$ PushThenRelabel$(G, \boldsymbol{c}_{\boldsymbol{f}_{i-1}}, \boldsymbol{f}_i', \mathbf{ex}^G_{\boldsymbol{f}_{i-1}, \Delta, \nabla_{i-1}}, \nabla_i, h, l)$
  $\boldsymbol{f}_i \leftarrow \boldsymbol{f}_{i-1} + \boldsymbol{f}_i'$
$\forall v \in V$ s.t. $l(v) = h + 1$: $l(v) \leftarrow h$
return $(\boldsymbol{f}_{8\cdot\log_2 n}, l)$

Since we do not change the algorithm compared to [CMGS25], the correctness directly follows, as shown in the following lemma.

Lemma 3.10. Given a height parameter h and a residual flow instance $\Pi = (G, \boldsymbol{c}, \Delta, \nabla)$ where $\nabla(v) \geq \gamma \cdot \mathbf{deg}(v)$ for all vertices $v \in V$ for some $0 < \gamma \leq 1$, $\|\Delta\|_1 \leq 2m$ and $\Delta(v) \leq \eta \cdot \mathbf{deg}_G(v)$ for all $v \in V$, and $\|\boldsymbol{c}\|_\infty \leq \eta$, the algorithm ParallelUnitFlow$(G, \boldsymbol{c}, \Delta, \nabla, h)$ produces a flow $\boldsymbol{f}$ and labeling $l : V \longrightarrow \{0, ..., h\}$ such that:

Algorithm 2: PushThenRelabel($G, \boldsymbol{c}, \boldsymbol{f}, \Delta, \nabla, h, l$)

1 for j = h ..., 1 do
2 In parallel, push all flow from all vertices v with $l(v) = j$ that have excess flow to vertices u with level $l(u) = j - 1$ until there either is no flow left or all the edges to such vertices are saturated. Update $\boldsymbol{f}$ accordingly.
3 For all vertices v that only have saturated edges going to level $l(v) - 1$ and have no remaining sink capacity, increase their level $l(v) \leftarrow \min(l(v) + 1, h + 1)$.
4 return $(\boldsymbol{f}, l)$

(i) If $l(u) > l(v) + 1$ where $\{u, v\}$ is an edge, then $\{u, v\}$ is saturated in the direction from u to v, i.e. $\boldsymbol{f}(u, v) = \boldsymbol{c}(u, v)$.
(ii) If $l(u) \geq 1$, then u's sink is nearly saturated, i.e. $\boldsymbol{f}(u) \geq \nabla(u)/(8 \cdot \log_2 n)$.
(iii) If $l(u) < h$, then there is no excess mass at u, i.e. $\mathbf{ex}^G_{\Delta,\nabla,\boldsymbol{f}}(u) = 0$.

Now we analyze the complexity of the algorithm in a more fine-grained way.

Lemma 3.11. Given a height parameter h and a residual flow instance $\Pi = (G, \boldsymbol{c}, \Delta, \nabla)$ where $\nabla(v) \geq \gamma \cdot \mathbf{deg}(v)$ for all vertices $v \in V$ for some $0 < \gamma \leq 1$, $\|\Delta\|_1 \leq 2m$ and $\Delta(v) \leq \eta \cdot \mathbf{deg}_G(v)$ for all $v \in V$, and $\|\boldsymbol{c}\|_\infty \leq \eta$, ParallelUnitFlow($G$,$\boldsymbol{c}$, Δ, ∇, h) can be implemented to output the flow $\boldsymbol{f}$ and labeling $l : V \longrightarrow \{0, ..., h\}$ implicitly, i.e., only non-zero entries of $\boldsymbol{f}$ and l are stored in the output, in $\|\Delta\|_0 \cdot \widetilde{O}\left(\eta h^2/\gamma^2\right)$ work and $\widetilde{O}\left(\eta h^2/\gamma\right)$ depth.

Proof of Lemma 3.11. Since we only give implicit output, the initialization of $\boldsymbol{f}$, Δ, l can be skipped. Now we look into the for-loop. Throughout the algorithm, we will have the following invariance.

Claim 1. The total number of edges adjacent to a vertex without remaining sink capacity is at most $\widetilde{O}\left(\|\Delta\|_0/\gamma\right)$. This implies that the total number of vertices u with excess and the total number of vertices with $l(u) > 0$ are both bounded by $\widetilde{O}\left(\|\Delta\|_0/\gamma\right)$.

Proof. A vertex can have excess only if it has no remaining sink capacity, which means it absorbs at least $\widetilde{\Omega}\left(\gamma \cdot \mathbf{deg}(v)\right)$ flows. All vertices can absorb at most $\|\Delta\|_0$ flows in total. Thus, if we sum up $\gamma \cdot \mathbf{deg}(v)$ for all v with excess, it is at most $\|\Delta\|_0$, this implies that the total number of edges adjacent to a vertex with excess is at most $\widetilde{O}\left(\|\Delta\|_0/\gamma\right)$.

According to PushThenRelabel, a vertex has a non-zero level only if it has no sink capacity. So the claim follows. □

According to Claim 1, we can always compute x_i in $\widetilde{O}(1)$ depth and $\widetilde{O}\left(\|\Delta\|_0/\gamma\right)$ work.

We can skip initializing $\boldsymbol{f}'_i, \nabla_i$ since they can be accessed entry-wise when in query.

Now we look at the while-loop: the condition is checked in $\widetilde{O}\left(\|\Delta\|_0/\gamma\right)$ work and $\widetilde{O}(1)$ depth according to Claim 1. Inside PushThenRelabel, for each j, we only look at v with $l(v) > 0$, and push flows accordingly. According to Claim 1, the push operations can be performed in $\widetilde{O}\left(\|\Delta\|_0/\gamma\right)$ work and $\widetilde{O}(1)$ depth for each j. The relabel operations are only on vertices that have no remaining sink capacity, so it can be done in $\widetilde{O}\left(\|\Delta\|_0/\gamma\right)$ work and $\widetilde{O}(1)$ depth according to Claim 1.

Now we argue the number of times that the while-loop can be ran as the following claim. It is implicitly implied by the proof of Claim 4.1 in [CMGS25].

Claim 2. In ParallelUnitFlow, for each i, the while-loop terminates in $\widetilde{O}\left(\eta h/\gamma\right)$ loops.

Proof. Loot at PushThenRelabel, after the push operation, we claim that all the nodes with excess must have their level increase unless it is at level $h + 1$. This is because such a node cannot have remaining sink capacity as otherwise it cannot have excess; also it cannot have non-saturated edges going to level $l(v)-1$, as otherwise the excess would be pushed through that edge in the push operations (during the push operations, if a node has all edges going to level $l(v) - 1$ saturated, those edges cannot be un-saturated in the following push operations since push only happens from an upper level to a lower level). Thus, for each run of PushThenRelabel, we should think of all the remaining excess on a vertex less than $h + 1$ level being "raised" by one level.

Now we count the total number of flows that can be raised. A vertex v can raise at most $h\eta \cdot \mathbf{deg}_G(v)$ flows, since the edge capacities are bounded by η and $\mathbf{deg}_G(v)$ edges going into this node, and it can increase its level at most h times. Thus, the total number of flows that can be raised should be $h\eta \cdot \sum_{v \text{ has excess}} \mathbf{deg}_G(v)$. Notice that $\sum_{v \text{ has excess}} \mathbf{deg}_G(v)$ can be bounded by $\widetilde{O}\left(x_i/\gamma\right)$ according to the same reason as in Claim 1: if v has excess then v has absorbed $\widetilde{\Omega}\left(\gamma \cdot \mathbf{deg}_G(v)\right)$ flows, but the total flow that can be absorbed is x_i. Thus, the total number of flows that can be raised is bounded by $\widetilde{O}\left(h\eta \cdot x_i/\gamma\right)$.

Now if each while-loop raises $x_i/2$ flows, the number of while-loops is bounded by

$$\widetilde{O}\left(h\eta \cdot x_i/\gamma\right)/(x_i/2) = \widetilde{O}\left(\eta h/\gamma\right). \qquad \square$$

The above claim also implies that the support of $\boldsymbol{f}'_i$ is bounded by $\widetilde{O}\left(\eta h/\gamma\right) \cdot h \cdot \widetilde{O}\left(\|\Delta\|_0/\gamma\right)$, so $\boldsymbol{f}_i$ can be updated in such work for each iteration. In the end we update l with non-zero entries.

To summarize, the dominating term of complexity is the push operations, which costs in total $\widetilde{O}\left(\eta h/\gamma\right) \cdot h \cdot \widetilde{O}\left(\|\Delta\|_0/\gamma\right) = \|\Delta\|_0 \cdot \widetilde{O}\left(\eta h^2/\gamma^2\right)$ work and $\widetilde{O}\left(\eta h^2/\gamma\right)$ depth. □

3.3 Trimming

Our goal in this section is to prove Lemma 3.7. Just for readability, we restate the lemma here.

Lemma 3.7. The algorithm Trimming($G = (V, E), A, \phi$) with inputs

(1) a graph $G = (V, E)$ accessed by adjacency list,

(2) a set $A \subseteq V$ accessed by identity query, i.e., given a vertex, it answers whether it is in A or not, a parameter $\phi \in \mathbb{R}_{\geq 0}$,
(3) explicitly given $E(A, V \setminus A)$, such that $|E(A, V \setminus A)| \leq \phi \cdot m$,

outputs explicitly the set $V - A'$ and a flow $\boldsymbol{f}$ with non-zero entries explicitly given, such that

(1) $\boldsymbol{f}$ restricted in A' routes source $\frac{2}{\phi}(\mathbf{deg}_G(v) - \mathbf{deg}_{G[A']}(v))$ to sinks $\nabla(v) \leq \mathbf{deg}_G(v)/\log n$,
(2) $\mathbf{deg}_G(A - A') = \widetilde{O}\left(\frac{1}{\phi}\right)|E(A, V \setminus A)|$,

in deterministic $\widetilde{O}\left(|E(A, V \setminus A)|/\phi^4\right)$ work and $\widetilde{O}\left(1/\phi^3\right)$ depth.

Proof. We first need two lemmas almost identical to Claim 3.7 and Claim 3.8 in [CMGS25].

Lemma 3.12. The while-loop at Line 11 terminates in less than h steps.

Proof. The proof is identical to the proof for Claim 3.7 in [CMGS25] since we do not change Line 11. □

Lemma 3.13. The main loop at Line 4 of Algorithm 3 terminates after at most $\log n$ steps.

Proof. The proof is almost identical to the proof of Claim 3.8 in [CMGS25] except that we have an extra $\log n$ factor on both h and ∇_i which cancel out. To be precise, by following the same argument, let the total excess at the end of the iteration k by X^k, we have that

$$X^{i-1} \geq \mathbf{deg}_G(S_j)/8 \log^3 n$$

because every node in S_j absorbs at least a $1/8\log^2 n$ fraction of its degree many flows (according to Lemma 3.10 point (ii)). Moreover, similarly, we have that $X^i \leq \frac{4}{\phi} \cdot \frac{5 \ln m}{h} \cdot \mathbf{deg}_G(S_j) \leq 1/32 X^{i-1}$, and the algorithm terminates after $\log n$ iterations. □

Now we can prove the first point of our lemma. Almost identical to the proof of Claim 3.5 in [CMGS25], we have that $\boldsymbol{f}$ routes the source $\frac{2}{\phi}(\mathbf{deg}_G(v) - \mathbf{deg}_{G[A']}(v))$ to sinks $\nabla(v) = i \cdot \frac{\mathbf{deg}_G(v)}{\log^2 n}$ according to the definition of the algorithm (it terminates when the excess is 0 according to Line 9), since we have Lemma 3.13, which means $i \leq \log n$, we are done.

Now we prove the second point, which is almost identical to the proof of Claim 3.9 in [CMGS25]. Notice that every node v in $A - A'$ absorbs at least $(1/8\log^3 n)\mathbf{deg}_G(v)$ many flows according to point (ii) of Lemma 3.10, and according to the proof of Lemma 3.13, the total amount of flow that ever exists in the algorithm can be bounded by $X^0 + X^1 + \ldots$ which is bounded by $2X^0 = \frac{4}{\phi} \cdot |E(A, V \setminus A)|$ since $X_i \leq X_{i-1}/32$. Thus, we have that $2X^0 \geq (1/8\log^3 n)\mathbf{deg}_G(A - A')$, and we are done. □

Lemma 3.14. Given a graph $G = (V, E)$ accessed by adjacency list, a parameter $\phi \in \mathbb{R}_{\geq 0}$ and a set $A \subseteq V$ along with $E(A, V \setminus A)$ such that $G[A]$ is a ϕ-nearly expander and $|E(A, V \setminus A)| \leq \phi \cdot m$, the algorithm Trimming$(G = (V, E), A, \phi)$ can be implemented to output $A - A'$ (which implicitly tells what is A') in $\widetilde{O}\left(|E(A, V \setminus A)|/\phi^4\right)$ work and $\widetilde{O}\left(1/\phi^3\right)$ depth.

Algorithm 3: Trimming$(G = (V, E), A, \phi)$

1 $h \stackrel{\text{def}}{=} \frac{5120}{\phi} \cdot \log_2^3 n \cdot \ln m$
2 $\boldsymbol{c} \leftarrow \frac{1}{\phi} \cdot \mathbf{1}$
3 $A_0 \leftarrow A$; $\boldsymbol{f}_0 \leftarrow \mathbf{0}$, $\Delta_0 \leftarrow \frac{2}{\phi}(\mathbf{deg}_G[A] - \mathbf{deg}_{G[A]})$; $\nabla_0 \leftarrow \mathbf{0}$; $i \leftarrow 0$
4 while true do /* While we do not have a feasible flow. */
5 $\quad i \leftarrow i + 1$
6 $\quad \nabla_i \leftarrow \nabla_{i-1} + \frac{1}{\log^2 n}\mathbf{deg}_G(v)[A_{i-1}]$
7 $\quad (\boldsymbol{f}'_i, l_i) \leftarrow$ ParallelUnitFlow$(G[A_{i-1}], \boldsymbol{c}_{\boldsymbol{f}_{i-1}}[A_{i-1}], \mathbf{ex}^{G[A_{i-i}]}_{\boldsymbol{f}_{i-1}, \Delta_{i-1}, \nabla_{i-1}}, \frac{\mathbf{deg}_G(v)[A_{i-1}]}{\log_2 n}, h)$
8 $\quad \boldsymbol{f}_i \leftarrow \boldsymbol{f}_{i-1} + \boldsymbol{f}'_i$
9 $\quad$ if $\mathbf{ex}^{G[A_i]}_{\boldsymbol{f}_i, \Delta_i, \nabla_i} = \mathbf{0}$ then break
10 $\quad j \leftarrow 0$; $S_0 \leftarrow \{v \in A_{i-1} : l_i(v) = h\}$
11 $\quad$ while $|E_{\boldsymbol{f}_i}(S_j, A_{i-1} \setminus S_j)| \geq \frac{5 \ln m}{h} \cdot \mathbf{deg}_G(S_j)$ do
12 $\quad\quad j \leftarrow j + 1$; $S_j \leftarrow \{v \in A_{i-1} : l_i(v) \geq h - j\}$
13 $\quad A_i \leftarrow A_{i-1} \setminus S_j$.
14 $\quad \Delta_i \leftarrow \frac{2}{\phi}(\mathbf{deg}_G[A_i] - \mathbf{deg}_{G[A_i]})$; $\nabla_i \leftarrow \nabla_i[A_i]$
15 return $A' = A_{i-1}$, $\boldsymbol{f} = \boldsymbol{f}_i$

Proof of Lemma 3.14. We skip the explicit initialization of $h, \boldsymbol{c}, A_0, \boldsymbol{f}_0, \nabla_0$ as they are all zero and can be updated entry-wise when needed. We initialize Δ_0 by memorizing all its non-zero entries in $O(|E(A, V \setminus A)|)$ work and $O(1)$ depth.

According to Lemma 3.13, the main loop on Line 4 terminates after $\widetilde{O}(1)$ steps, so let us focus on each step of the loop.

For the updates on Δ_i, we will only update the entries which are non-trivial, i.e., which have absorbed some flows so that the entry is less than $(i/\log^2 n)\mathbf{deg}_G(v)[A_{i-1}]$. The work for updating Δ_i thus can be charged to computing the flow $\boldsymbol{f}_i$.

Now we compute the work and depth for the call to ParallelUnitFlow. According to Lemma 3.11, initially we have that $\gamma = \Omega(1/\log^2 n)$, $\eta = \widetilde{O}(1/\phi)$, and $\|\Delta\|_0 \leq \frac{4}{\phi}|E(A, V \setminus A)|$, so that the work is $\widetilde{O}\left(|E(A, V \setminus A)|/\phi^4\right)$ and depth is $\widetilde{O}\left(1/\phi^3\right)$. Notice that according to the proof of Lemma 3.12, the total source flow at each iteration X^i is only decreasing, and γ and η remain unchanged, so we have that the work over all iterations is bounded by $\widetilde{O}\left(|E(A, V \setminus A)|/\phi^4\right)$.

Both the update of $\boldsymbol{f}_i$ and the check of excess can be charged to the work of ParallelUnitFlow as just performed.

The while-loop in Line 11 contains less than h iterations according to Lemma 3.12. The work for each iteration depends on $\sum_{l_i(v)>0} \mathbf{deg}_G(v)$, which is bounded by $\widetilde{O}\left(|E(A, V \setminus A)|/\phi\right)$ according to Claim 1.

The update on A_i should be implemented by adding the set S_j to $V \setminus A_{i-1}$ since we are maintaining $V \setminus A_i$. The update

of Δ_i can be done easily since it only contains edges adjacent to $V \setminus A_i$; the update on ∇ should not be explicitly maintained as we only need to store the non-trivial sink capacities.

To summarize, the work and depth are dominated by the flow computation, which takes $\widetilde{O}\left(|E(A, V \setminus A)|/\phi^4\right)$ work and $\widetilde{O}\left(1/\phi^3\right)$ depth. □

4 ACKNOWLEDGMENT

We thank Thatchaphol Saranurak for pointing out the paper [CMGS25] at the early stage of this project.

Jan van den Brand was supported by NSF Award CCF-2338816.

This research was funded in whole or in part by the Austrian Science Fund (FWF) https://doi.org/10.55776/P36280. For open access purposes, the author has applied a CC BY public copyright license to any author-accepted manuscript version arising from this submission.

REFERENCES

[ABC+24] Vikrant Ashvinkumar, Aaron Bernstein, Nairen Cao, Christoph Grunau, Bernhard Haeupler, Yonggang Jiang, Danupon Nanongkai, and Hsin-Hao Su. Parallel, distributed, and quantum exact single-source shortest paths with negative edge weights. In Timothy M. Chan, Johannes Fischer, John Iacono, and Grzegorz Herman, editors, 32nd Annual European Symposium on Algorithms, ESA 2024, September 2-4, 2024, Royal Holloway, London, United Kingdom, volume 308 of LIPIcs, pages 13:1–13:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[AKL+24] Arpit Agarwal, Sanjeev Khanna, Huan Li, Prathamesh Patil, Chen Wang, Nathan White, and Peilin Zhong. Parallel approximate maximum flows in near-linear work and polylogarithmic depth. In David P. Woodruff, editor, Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024, pages 3997–4061. SIAM, 2024.

[ALPS23] Amir Abboud, Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. All-pairs max-flow is no harder than single-pair max-flow: Gomory-hu trees in almost-linear time. In 64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023, pages 2204–2212. IEEE, 2023.

[AMV20] Kyriakos Axiotis, Aleksander Madry, and Adrian Vladu. Circulation control for faster minimum cost flow in unit-capacity graphs. In Sandy Irani, editor, 61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020, pages 93–104. IEEE, 2020.

[ASZ20] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Parallel approximate undirected shortest paths via low hop emulators. In Proc. of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, pages 322–335. ACM, 2020.

[BBG+22] Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-dynamic graph sparsifiers against an adaptive adversary. In ICALP, volume 229 of LIPIcs, pages 20:1–20:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[BCP+23] Jan van den Brand, Li Chen, Richard Peng, Rasmus Kyng, Yang P. Liu, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. A deterministic almost-linear time algorithm for minimum-cost flow. In 64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023, pages 503–514. IEEE, 2023.

[BGK+14] Guy E. Blelloch, Anupam Gupta, Ioannis Koutis, Gary L. Miller, Richard Peng, and Kanat Tangwongsan. Nearly-linear work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs. Theory Comput. Syst., 55(3):521–554, 2014. Announced at SPAA'11.

[BLL+21] Jan van den Brand, Yin Tat Lee, Yang P. Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and ℓ_1-regression in nearly linear time for dense instances. In Samir Khuller and Virginia Vassilevska Williams, editors, STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021, pages 859–869. ACM, 2021.

[BLN+20] Jan van den Brand, Yin Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS), pages 919–930, 2020.

[BNW25] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. Negative-weight single-source shortest paths in near-linear time. Commun. ACM, 68(2):87–94, 2025. Announced at FOCS'22.

[Bra20] Jan van den Brand. A deterministic linear program solver in current matrix multiplication time. In Shuchi Chawla, editor, Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020, pages 259–278. SIAM, 2020.

[CF23] Nairen Cao and Jeremy T. Fineman. Parallel exact shortest paths in almost linear work and square root depth. 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 4354–4372. SIAM, 2023.

[CFR22] Nairen Cao, Jeremy T. Fineman, and Katina Russell. Parallel shortest paths with negative edge weights. In Kunal Agrawal and I-Ting Angelina Lee, editors, SPAA '22: 34th ACM Symposium on Parallelism in Algorithms and Architectures, Philadelphia, PA, USA, July 11 - 14, 2022, pages 177–190. ACM, 2022.

[CKL+22] 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 63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022, pages 612–623. IEEE, 2022.

[CKM+14] Michael B. Cohen, Rasmus Kyng, Gary L. Miller, Jakub W. Pachocki, Richard Peng, Anup B. Rao, and Shen Chen Xu. Solving SDD linear systems in nearly $m\log^{1/2}n$ time. In Proc. of the Symposium on Theory of Computing, STOC 2014, pages 343–352. ACM, 2014.

[CLS21] Michael B. Cohen, Yin Tat Lee, and Zhao Song. Solving linear programs in the current matrix multiplication time. J. ACM, 68(1):3:1–3:39, 2021.

[CMGS25] Daoyuan Chen, Simon Meierhans, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Parallel and distributed expander decomposition: Simple, fast, and near-optimal. In SODA, pages 1705–1719. SIAM, 2025.

[CMSV17] Michael B. Cohen, Aleksander Madry, Piotr Sankowski, and Adrian Vladu. Negative-weight shortest paths and unit capacity minimum cost flow in $\widetilde{o}(m^{10/7}\log w)$ time (extended abstract). In Proc. of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, pages 752–771. SIAM, 2017.

[CP15] Michael B. Cohen and Richard Peng. L_p row sampling by lewis weights. In Rocco A. Servedio and Ronitt Rubinfeld, editors, Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015, pages 183–192. ACM, 2015.

[CS19] 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.

[CS20] Yi-Jun Chang and Thatchaphol Saranurak. Deterministic distributed expander decomposition and routing with applications in distributed derandomization. In Sandy Irani, editor, 61st IEEE Annual Symposium on Foundations of

Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020, pages 377–388. IEEE, 2020.

[dV23] Tijn de Vos. Minimum cost flow in the CONGEST model. In Sergio Rajsbaum, Alkida Balliu, Joshua J. Daymude, and Dennis Olivetti, editors, Structural Information and Communication Complexity - 30th International Colloquium, SIROCCO 2023, Alcalá de Henares, Spain, June 6-9, 2023, Proceedings, volume 13892 of Lecture Notes in Computer Science, pages 406–426. Springer, 2023. Announced at PODC'23.

[DWZ23] Ran Duan, Hongxun Wu, and Renfei Zhou. Faster matrix multiplication via asymmetric hashing. In 64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023, pages 2129–2138. IEEE, 2023.

[FdV22] Sebastian Forster and Tijn de Vos. The laplacian paradigm in the broadcast congested clique. In Alessia Milani and Philipp Woelfel, editors, PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022, pages 335–344. ACM, 2022.

[FdV23] Sebastian Forster and Tijn de Vos. Brief announcement: The laplacian paradigm in deterministic congested clique. In Rotem Oshman, Alexandre Nolin, Magnús M. Halldórsson, and Alkida Balliu, editors, Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023, pages 75–78. ACM, 2023.

[FGL+21] Sebastian Forster, Gramoz Goranci, Yang P. Liu, Richard Peng, Xiaorui Sun, and Mingquan Ye. Minor sparsifiers and the distributed laplacian paradigm. In 62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022, pages 989–999. IEEE, 2021.

[FHL+25] Nick Fischer, Bernhard Haeupler, Rustam Latypov, Antti Roeyskoe, and Aurelio L. Sulser. A simple parallel algorithm with near-linear work for negative-weight single-source shortest paths. In Proc. of the ACM-SIAM Symposium on Simplicity in Algorithms, SOSA 2025. SIAM, 2025.

[Fin20] Jeremy T. Fineman. Nearly work-efficient parallel algorithm for digraph reachability. SIAM J. Comput., 49(5), 2020. Announced at STOC'18.

[Gab85] Harold N. Gabow. Scaling algorithms for network problems. J. Comput. Syst. Sci., 31(2):148–168, 1985. Announced at FOCS'83.

[Gol95] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. SIAM J. Comput., 24(3):494–504, 1995. Announced at SODA'93.

[GT89] Harold N. Gabow and Robert Endre Tarjan. Faster scaling algorithms for network problems. SIAM J. Comput., 18(5):1013–1036, 1989.

[JLS19] Arun Jambulapati, Yang P. Liu, and Aaron Sidford. Parallel reachability in almost linear work and square root depth. In David Zuckerman, editor, 60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019, pages 1664–1686. IEEE Computer Society, 2019.

[JS21] Wenyu Jin and Xiaorui Sun. Fully dynamic s-t edge connectivity in subpolynomial time (extended abstract). In FOCS, pages 861–872. IEEE, 2021.

[JSWZ21] Shunhua Jiang, Zhao Song, Omri Weinstein, and Hengjie Zhang. A faster algorithm for solving general lps. In Samir Khuller and Virginia Vassilevska Williams, editors, STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021, pages 823–832. ACM, 2021.

[KLOS14] 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 Proc. of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, pages 217–226. SIAM, 2014.

[KLP+16] Rasmus Kyng, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Daniel A. Spielman. Sparsified cholesky and multigrid solvers for connection laplacians. In Proc. of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, pages 842–850. ACM, 2016.

[KLS24] Tarun Kathuria, Yang P. Liu, and Aaron Sidford. Unit capacity maxflow in almost $m^{4/3}$ time. SIAM J. Comput., 53(6):S20–175, 2024. Announced at FOCS'20.

[KM07] Ioannis Koutis and Gary L. Miller. A linear work, $o(n^{1/6})$ time, parallel algorithm for solving planar laplacians. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007, pages 1002–1011. SIAM, 2007.

[KMP11] Ioannis Koutis, Gary L. Miller, and Richard Peng. A nearly-m log n time solver for SDD linear systems. In Proc. of the IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, pages 590–598. IEEE Computer Society, 2011.

[KMP14] Ioannis Koutis, Gary L. Miller, and Richard Peng. Approaching optimality for solving SDD linear systems. SIAM J. Comput., 43(1):337–354, 2014. Announced at FOCS 2010.

[KOSZ13] Jonathan A. Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen Zhu. A simple, combinatorial algorithm for solving sdd systems in nearly-linear time. In Proc. of the 45th Annual ACM Symposium on Theory of Computing (STOC 2013), pages 911–920, 2013.

[KS16] Rasmus Kyng and Sushant Sachdeva. Approximate gaussian elimination for laplacians - fast, sparse, and simple. In Proc. of the IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, pages 573–582. IEEE Computer Society, 2016.

[KT19] Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic edge connectivity in near-linear time. J. ACM, 66(1):4:1–4:50, 2019.

[KUW86] Richard M. Karp, Eli Upfal, and Avi Wigderson. Constructing a perfect matching is in random NC. Comb., 6(1):35–48, 1986. Announced at STOC'85.

[KX16] Ioannis Koutis and Shen Chen Xu. Simple parallel and distributed algorithms for spectral graph sparsification. ACM Trans. Parallel Comput., 3(2):14:1–14:14, 2016. Announced at SPAA'14.

[Li20] Jason Li. Faster parallel algorithm for approximate shortest path. In Proc. of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, pages 308–321. ACM, 2020.

[Li21] Jason Li. Deterministic mincut in almost-linear time. In Samir Khuller and Virginia Vassilevska Williams, editors, STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021, pages 384–395. ACM, 2021.

[LNP+21] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Vertex connectivity in poly-logarithmic max-flows. In Samir Khuller and Virginia Vassilevska Williams, editors, STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021, pages 317–329. ACM, 2021.

[Lov79] László Lovász. On determinants, matchings, and random algorithms. In Lothar Budach, editor, Fundamentals of Computation Theory, FCT 1979, Proceedings of the Conference on Algebraic, Arthmetic, and Categorial Methods in Computation Theory, Berlin/Wendisch-Rietz, Germany, September 17-21, 1979, pages 565–574. Akademie-Verlag, Berlin, 1979.

[LPS15] Yin Tat Lee, Richard Peng, and Daniel A. Spielman. Sparsified cholesky solvers for SDD linear systems. CoRR, abs/1506.08204, 2015.

[LS14] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in $\widetilde{O}(\sqrt{\text{rank}})$ iterations and faster algorithms for maximum flow. In Proc. of the 55th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2014), pages 424–433. IEEE Computer Society, 2014.

[LS19] Yin Tat Lee and Aaron Sidford. Solving linear programs with sqrt(rank) linear system solves, 2019.

[LS20a] Yang P. Liu and Aaron Sidford. Faster divergence maximization for faster maximum flow. CoRR, abs/2003.08929, 2020.

[LS20b] Yang P. Liu and Aaron Sidford. Faster energy maximization for faster maximum flow. In Proc. of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, pages 803–814. ACM, 2020.

[LSZ19] Yin Tat Lee, Zhao Song, and Qiuyi Zhang. Solving empirical risk minimization in the current matrix multiplication time. In Alina Beygelzimer and Daniel Hsu, editors, Conference on Learning Theory, COLT 2019, 25-28 June 2019, Phoenix, AZ, USA, volume 99 of Proceedings of Machine Learning Research, pages 2140–2157. PMLR, 2019.

[Mad13] Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In Proc. of the 54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, pages 253–262. IEEE Computer Society, 2013.

[Mad16] Aleksander Madry. Computing maximum flow with augmenting electrical flows. In Proc. of the IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, pages 593–602. IEEE Computer Society, 2016.

[MVV87] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. Comb., 7(1):105–113, 1987. Announced at STOC'87.

[NSW17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In FOCS, pages 950–961. IEEE Computer Society, 2017.

[OS93] James B. Orlin and Clifford Stein. Parallel algorithms for the assignment and minimum-cost flow problems. Oper. Res. Lett., 14(4):181–186, 1993.

[Pen16] Richard Peng. Approximate undirected maximum flows in O(mpolylog(n)) time. In Proc. of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, pages 1862–1867. SIAM, 2016.

[PF22] Yossi Peretz and Yigal Fischler. A fast parallel max-flow algorithm. J. Parallel Distributed Comput., 169:226–241, 2022.

[PS13] Richard Peng and Daniel A. Spielman. An efficient parallel solver for sdd linear systems, 2013.

[R+90] Vijaya Ramachandran et al. Parallel algorithms for shared-memory machines. In Algorithms and Complexity, pages 869–941. Elsevier, 1990.

[RGH+22] Václav Rozhon, 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 Stefano Leonardi and Anupam Gupta, editors, STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022, pages 478–487. ACM, 2022.

[She13] Jonah Sherman. Nearly maximum flows in nearly linear time. In Porc. of the 54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, pages 263–269. IEEE Computer Society, 2013.

[Spe97] Thomas H. Spencer. Time-work tradeoffs for parallel algorithms. J. ACM, 44(5):742–778, 1997. Announced at SODA'91.

[SS91] Maria J. Serna and Paul G. Spirakis. Tight RNC approximations to max flow. In Christian Choffrut and Matthias Jantzen, editors, STACS 91, 8th Annual Symposium on Theoretical Aspects of Computer Science, Hamburg, Germany, February 14-16, 1991, Proceedings, volume 480 of Lecture Notes in Computer Science, pages 118–126. Springer, 1991.

[ST04] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In Proc. of the 36th Annual ACM Symposium on Theory of Computing (STOC 2004), pages 81–90. ACM, 2004.

[SV82] Yossi Shiloach and Uzi Vishkin. An $o(n^2 \log n)$ parallel MAX-FLOW algorithm. J. Algorithms, 3(2):128–146, 1982.

[SW19] Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In SODA, pages 2616–2635. SIAM, 2019.

[SZ23] Sushant Sachdeva and Yibin Zhao. A simple and efficient parallel laplacian solver. In Kunal Agrawal and Julian Shun, editors, Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2023, Orlando, FL, USA, June 17-19, 2023, pages 315–325. ACM, 2023.

[UY91] Jeffrey D. Ullman and Mihalis Yannakakis. High-probability parallel transitive-closure algorithms. SIAM J. Comput., 20(1):100–125, 1991. Announced at SPAA'90.

[VXXZ24] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In David P. Woodruff, editor, Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024, pages 3792–3835. SIAM, 2024.