Parallel and Distributed Expander Decomposition: Simple, Fast, and Near-Optimal

Daoyuan Chen
ETH Zurich
chenda@student.ethz.ch

Simon Meierhans*
ETH Zurich
mesimon@inf.ethz.ch

Maximilian Probst Gutenberg*
ETH Zurich
maximilian.probst@inf.ethz.ch

Thatchaphol Saranurak†
University of Michigan
thsa@umich.edu

Abstract

Expander decompositions have become one of the central frameworks in the design of fast algorithms. For an undirected graph $G = (V, E)$, a near-optimal ϕ-expander decomposition is a partition $V_1, V_2, \ldots, V_k$ of the vertex set V where each subgraph $G[V_i]$ is a ϕ-expander, and only an $\widetilde{O}(\phi)$-fraction of the edges cross between partition sets.

In this article, we give the first near-optimal *parallel* algorithm to compute ϕ-expander decompositions in near-linear work $\widetilde{O}(m/\phi^2)$ and near-constant span $\widetilde{O}(1/\phi^4)$. Our algorithm is very simple and likely practical. Our algorithm can also be implemented in the distributed Congest model in $\tilde{O}(1/\phi^4)$ rounds.

Our results surpass the theoretical guarantees of the current state-of-the-art parallel algorithms [CS19, CS20], while being the first to ensure that only an $\tilde{O}(\phi)$ fraction of edges cross between partition sets. In contrast, previous algorithms [CS19, CS20] admit at least an $O(\phi^{1/3})$ fraction of crossing edges, a polynomial loss in quality inherent to their random-walk-based techniques. Our algorithm, instead, leverages flow-based techniques and extends the popular sequential algorithm presented in [SW19].

1 Introduction

Over the past two decades, expander decompositions have emerged as a central framework in graph algorithms. At a high level, these decompositions partition a graph into a collection of well-conditioned subgraphs and a small set of crossing edges. Formally, a ϕ-expander G is a graph in which, for any subset $S \subseteq V$, the number of edges leaving S is large compared to the volume of the smaller side of the cut $(S, V \setminus S)$. Specifically, $|E_G(S, V \setminus S)| \geq \phi \cdot \min\{\mathbf{deg}_G(S), \mathbf{deg}_G(V \setminus S)\}$. A ϕ-expander decomposition of a graph $G = (V, E)$ is a partition of the vertex set V into clusters $V_1, V_2, \ldots, V_k$ such that each subgraph $G[V_i]$ is a ϕ-expander. The error of an expander decomposition is the number of edges in E that cross between clusters V_i and V_j for $i \neq j$. Intuitively, ϕ-expanders are well-connected clusters where the 'well-connectedness' increases with ϕ. When

*The research leading to these results has received funding from grant no. 200021 204787 of the Swiss National Science Foundation. Simon Meierhans is supported by a Google PhD Fellowship.

†Supported by NSF grant CCF-2238138.

$\phi \approx 1$, the graph is very well-connected, has low diameter, and exhibits spectral properties similar to those of a complete graph. Conversely, every simple connected graph is a $\phi = 1/|V|^2$ expander.

Expander decompositions have been pivotal in developing the first deterministic and randomized almost-linear time algorithms for various fundamental graph problems such as maximum and min-cost flow [KLOS14, CKL+22], electrical flows [ST04], Gomory-Hu trees [ALPS23], and edge- and vertex-connectivity [KT18, Li21, LNP+21], and many more [WN17, NS17, NSWN17, CK19, BGS20, BvdBPG+22, CS21, GRST21, Chu21, BGS22, BGS22, JS22, KMG24, CKL+24, JST24, CK24, vdBCK+24].

This research program seeking fast almost-linear time algorithms is driven by the increasing volume of data, resulting in large-scale graphs for which it is prohibitive to spend more than almost-linear time relative to the input size.

In a first effort to translate these new almost-linear time algorithms from theory to practice, [GPPG24] gave a first practical implementation of an algorithm to compute expander decompositions for the important regime of $\phi = \tilde{\Omega}(1)$[1]. These implementations demonstrate that expander decompositions can be computed in reasonable time (under or roughly one day on a high-performance computer) for medium-sized graphs with roughly 400,000 to 35,000,000 edges.[2] This is achieved by implementing the framework suggested in [SW19], the simplest and theoretically fastest algorithm to compute expander decompositions, augmented by various practical optimizations and heuristics.

While a promising first step, the result from [GPPG24] strongly suggests that either the algorithm has to be significantly improved, or that parallelization has to be harnessed in order to be able to handle large-sized graphs, meaning graphs that are of the order of a hundred-million or even a billion edges.

1.1 Our Contribution

In this paper, we present a new parallel algorithm to compute expander decompositions.

Theorem 1.1 (Parallel Expander Decomposition)**.** *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[3] finds a ϕ-expander decomposition $V_1, .., V_k$ with error $\sum_{i<j} |E_G(V_i, V_j)| = \widetilde{O}(\phi m)$. The total work of the algorithm is $\widetilde{O}(m/\phi^2)$ with span $\widetilde{O}(1/\phi^4)$.*

Our algorithm is also implementable in the distributed CONGEST model.

Theorem 1.2 (Distributed Expander Decomposition)**.** *Given a graph $G = (V, E)$ of m edges and a parameter $\phi \in (0, 1)$, there is a randomized distributed algorithm in the* CONGEST *model that with high probability finds a ϕ-expander decomposition $V_1, .., V_k$ with error $\sum_{i<j} |E_G(V_i, V_j)| = \widetilde{O}(\phi m)$. The number of rounds are $\widetilde{O}(1/\phi^4)$.*

Distributed expander decompositions are useful in particular because many parallel algorithms can be directly ported to the distributed model on expander graphs [GKS17, GL18, CPZ19]. In this article, we only prove the parallel expander decomposition result. The algorithm can be ported to the distributed model directly.

[1] In this article we use $\tilde{O}(\cdot)$ and $\tilde{\Omega}(\cdot)$ to hide $\operatorname{polylog}|V|$ and $1/\operatorname{polylog}|V|$ factors respectively.

[2] In their experiments, they used a 2x8-core Intel Xeon Gold 6144 Skylake CPU, clocked at 3.5GHz with 24.75MB L3 cache and 192GB DDR4 RAM (2666 MHz).

[3] In this article we say with high probability to mean that for every constant $C > 0$, there is such an algorithm that succeeds with probability at least $1 - n^{-C}$.

Our algorithm yields expander decompositions of nearly-optimal error in quality and has extremely low span for the important setting where $\phi = \tilde{\Omega}(1)$. At the same time, our algorithm is still very simple and likely practical to the extent that it can most likely be integrated with the implementation from [GPPG24] with relatively little overhead as it extends the framework of [SW19].

Our result also simultaneously surpasses all theoretical guarantees obtained by previous parallel and distributed algorithms [CS19, CS20]. In particular, our algorithm is the first to obtain near-optimal error for any value of ϕ improving from the currently best error of $\tilde{O}(\phi^{1/3}m)$ achieved by [CS20]. Our algorithm matches the work bound of the best sequential algorithm [SW19] up to a $\tilde{O}(1/\phi)$ factor. We refer the reader to Appendix B for a review of sequential, parallel and distributed expander decomposition algorithms.

We further believe that our newly developed techniques are of general interest in the design of other parallel algorithms in general and in the design of faster, stronger and more robust sequential and dynamic algorithms for expander decompositions in particular.

1.2 Our Techniques

Our algorithm takes the framework of [SW19] as a starting point. In their framework, the popular cut-matching algorithm [KRV09] is used to find balanced sparse cuts, that is cuts $(S, V \setminus S)$ where $|E_G(S, V \setminus S)| < \phi \cdot \min\{\mathbf{deg}_G(S), \mathbf{deg}_G(V \setminus S)\}$ and the degree incident to S and $V \setminus S$ is $\tilde{\Omega}(m)$, respectively; or certify that no such balanced sparse cut exists. If a sparse cut is found, it is used to refine the partition of the vertex set.

Conversely, if no balanced sparse cut exists the algorithm returns a vertex set $A \subseteq V$ of large volume such that $G[A]$ is a nearly expander. Formally, we have $\mathbf{deg}_G(V \setminus A) = O(m/\text{ polylog}(m))$ and every $S \subseteq A$, $|E_G(S, V \setminus S)| \geq \phi \cdot \min\{\mathbf{deg}_G(S), \mathbf{deg}_G(V \setminus S)\}$. From an algorithmic perspective, it would be desirable to turn A into a partition set and recurse on $V \setminus A$, but unfortunately $G[A]$ isn't quite a ϕ-expander. Therefore, a trimming algorithm is used to find a large subset $A' \subseteq A$ such that $G[A']$ is a $\Omega(\phi)$-expander, and the algorithm recurses on $V \setminus A'$ instead.

Trimming with Few Iterations. In [SW19], a trimming algorithm is obtained by constructing a flow problem on $G[A]$ where every edge has capacity $2/\phi$, every vertex v is a source with $2/\phi \cdot (\mathbf{deg}_G(v) - \mathbf{deg}_{G[A]}(v))$ mass and $\mathbf{deg}_G(v)$ sink capacity. It is then shown that the min-cut in this flow problem yields a relatively sparse cut in $G[A]$ and that the larger side A' of this cut is a $\Omega(\phi)$-expander.

Since current fast maximum flow algorithms are far from practical and have large sub-polynomial overhead, the state-of-the-art trimming algorithms use the unit flow algorithm from [HRW20], a height-constrained version of the famous push-relabel algorithm, that heavily exploits the uniform sinks of the flow problem above. They set up said flow problem for sets $A_0 = A, A_1, \ldots, A_k$ where $A_{i+1} \subseteq A_i$. The i-th flow problem is then used to determine which vertices need to be removed to arrive at A_{i+1} and eventually the algorithm will solve the flow problem and decide that $G[A_k]$ is a $\Omega(\phi)$-expander.

But while the authors cleverly re-use information across the flow problems to obtain fast runtime, the algorithm is inherently sequential as it needs to solve the i-th flow problem before it even knows A_{i+1}. In our work, we observe that a rather simple technique can be used to bound the number of flow problems that ever need to be solved by $\log_2 n$. To do so, we show that it suffices to grow the sink capacity over the flow problems slowly, i.e. instead of directly admitting $\mathbf{deg}_G(v)$ sink

capacity, our algorithm admits only $\frac{i \cdot \mathbf{deg}_G(v)}{\log_2 n}$ sink capacity in the i-th flow problem. We show that this speeds up convergence exponentially. The technique is described and analyzed in detail in Section 3.

Parallel Unit Flow. Our second contribution is to parallelize the unit flow algorithm that was given in [HRW20]. As a starting point, we exploit the simple insight from [GT88] for the underlying push-relabel algorithm that one can work with rounds where in even rounds, levels remain fixed and all possible pushes are executed, and in odd rounds all vertices that still have excess can be relabelled in parallel. We note that via the analysis of [GT88], this only yields a trivial upper bound on the number of rounds.

However, we show that as long as at most half of the original source mass is settled (meaning absorbed or placed as excess at an 'inactive' vertex of maximum admissible level), we can carry out an $\tilde{\Omega}(1/\phi^2)$-fraction of the total work in every round. Further, by exploiting the structure of the flow problem, we show that we can implement each round with span $\tilde{O}(1/\phi)$. Thus, we can obtain a span of $\tilde{O}(1/\phi^3)$.

Clearly, this technique only works until half the original source mass is settled. However, we again exploit that growing the sink capacity keeps the algorithm efficient. Concretely, each parallel unit flow problem has for every vertex v a sink capacity of at least $\Delta(v) = \mathbf{deg}_G(v)/\log_2(n)$. We start solving the flow problem while only admitting $\frac{\Delta(v)}{8 \cdot \log_2(n)}$ capacity to each sink and then grow the capacity by $\frac{\Delta(v)}{8 \cdot \log_2(n)}$ whenever the unsettled source mass drops by a factor of $1/2$. This ensures that the span remains $\tilde{O}(1/\phi^3)$ for each flow problem, and therefore the total span is at most $\tilde{O}(\log n/\phi^3)$, as desired.

Our parallel unit-flow algorithm is described and analyzed in Section 4.

A Distributed Algorithm. Since we employ the distributed framework of [CS19, CS20] for implementing the cut matching game, it suffices to realize that all our flow algorithms and trimming procedures can be implemented in the distributed CONGEST model as well. Our parallel unit flow algorithm is completely local since the only operations we perform is locally adding sink, and pushing one unit of flow across an edge. Our trimming procedure in Section 3 uses parallel unit flow and ball growing to identify a sparse cut. Growing a ball to the adequate diameter can be implemented in the CONGEST model directly. Finally, the parallel matching algorithm used to implement the cut matching game in Appendix A uses a combination of our unit flow algorithm and ball growing, which is again implementable in a distributed fashion directly.

1.3 Roadmap

For the rest of this article, we focus on obtaining a fast parallel trimming routine. The main technical result is stated in Lemma 3.2. We defer the proof of Theorem 1.1 to Appendix A since the parallelization of the framework of [SW19] is rather straightforward given a parallel trimming algorithm and has already been done previously in [CS20].

2 Preliminaries

Graphs. We denote graphs as tuples $G = (V, E)$ and refer to the number of vertices and edges with n and m respectively. For an undirected graph $G = (V, E)$ and a subset $A \subset V$, we use $G[A]$

to denote the induced graph and $E[A]$ to denote the set of edges with both endpoints in A, i.e. $E[A] = \{\{u, v\} \in E, u, v \in A\}$. For disjoint subsets $A, B \subset V$, we denote by $E(A, B)$ edges in E with exactly one endpoint in A and B. We denote with $\mathbf{deg}_G$ the degree vector of graph G, and for a set $S \subseteq V$ we let $\mathbf{deg}_G(S) \stackrel{\text{def}}{=} \sum_{v \in S} \mathbf{deg}_G(v)$. We let $\boldsymbol{B} \in \mathbb{R}^{E \times V}$ denote the edge-vertex incidence matrix for an arbitrary but fixed orientation of the edges of graph G.

We sometimes use X_G to refer to a variable X associated with graph G to remove ambiguity.

Flows. A *(residual) flow instance* $\Pi = (G, \boldsymbol{c}, \boldsymbol{f}, \boldsymbol{\Delta}, \boldsymbol{\nabla})$ consists of an undirected graph $G = (V, E)$, a capacity vector $\boldsymbol{c} \in \mathbb{R}^E_{\geq 0}$, a feasible flow vector $\boldsymbol{f} \in \mathbb{R}^E$, a source vector $\boldsymbol{\Delta} \in \mathbb{R}^V_{\geq 0}$ and a sink vector $\boldsymbol{\nabla} \in \mathbb{R}^V_{\geq 0}$. We use *mass* to refer to the substance routed.

For a vertex $v \in V$, $\boldsymbol{\Delta}(v)$ specifies the amount of mass initially placed on v, and $\boldsymbol{\nabla}(v)$ specifies the capacity of v as a sink, i.e., the amount of mass that v can absorb. For an edge $\boldsymbol{c}(e)$ specifies the amount of mass that can be routed along e in either direction. A flow $\boldsymbol{f}$ is said to be feasible if $-\boldsymbol{c} \leq \boldsymbol{f} \leq \boldsymbol{c}$.

We say $e \in E$ is *saturated* if $|\boldsymbol{f}(e)| = \boldsymbol{c}(e)$. Given a flow $\boldsymbol{f}$ on the undirected graph G we let $G_{\boldsymbol{f}} = (V, E_{\boldsymbol{f}})$ denote the residual graph where the edge $e = (u, v)$ has residual capacity $\boldsymbol{c}(e) - \boldsymbol{f}(e)$ in direction of (u, v), and $\boldsymbol{c}(e) + \boldsymbol{f}(e)$ in direction (v, u). We use $E_{\boldsymbol{f}}(A, B)$ to denote the edges in $E(A, B)$ with nonzero residual capacity going from A to B. For convenience, we let $\boldsymbol{f}(u, v) \stackrel{\text{def}}{=} \boldsymbol{f}(e)$ for $e = (u, v)$ and $\boldsymbol{f}(v, u) \stackrel{\text{def}}{=} -\boldsymbol{f}(u, v)$ denote the flow in the other direction. Finally, we let $\boldsymbol{c}_{\boldsymbol{f}}$ be the directed residual capacity vector for initial capacities $\boldsymbol{c}$ and a flow $\boldsymbol{f}$.

For a given flow $\boldsymbol{f}$ on a graph G with source $\boldsymbol{\Delta}$ and sink $\boldsymbol{\nabla}$, we let $\mathbf{ex}^G_{\boldsymbol{f}, \boldsymbol{\Delta}, \boldsymbol{\nabla}} \stackrel{\text{def}}{=} \max(\boldsymbol{B}^T_G \boldsymbol{f} + \boldsymbol{\Delta} - \boldsymbol{\nabla}, \mathbf{0})$ denote the excess (source) left over.

Linear Algebra. Given a vector $\boldsymbol{x} \in \mathbb{R}^A$, we let $\boldsymbol{x}[B]$ for $B \subseteq A$ denote the vector in $\mathbb{R}^B$ with $\boldsymbol{x}[B](i) = \boldsymbol{x}(i)$ for $i \in B$.

3 The Trimming Step

In this section, we present our parallel algorithm for trimming nearly expanders to expander graphs. Given a graph G, a set A is said to induce a nearly expander if it expands in the context of the whole graph G.

Definition 3.1 (Nearly Expander)**.** *Given $G = (V, E)$ and $A \subseteq V$, $G[A]$ is a ϕ-nearly expander in G if for all $S \subseteq A$ such that $\mathbf{deg}_G(S) \leq \mathbf{deg}_G(A \setminus S)$, we have $|E_G(S, V \setminus S)| \geq \phi \cdot \mathbf{deg}_G(S)$.*

Notice that Definition 3.1 would correspond to the definition of expander graphs if the edge set $E_G(S, V \setminus S)$ was restricted to edges internal to $G[A]$. Therefore, every induced expander is also a nearly-expander, but the contrary is not the case in general. Historically, nearly expanders play a crucial role in the development of efficient algorithms for computing expander decompositions, and the first near-linear time algorithms only guaranteed that each component be a nearly expander [ST13]. To obtain the stronger and significantly easier to work with guarantee that each cluster be an expander graph, expander trimming algorithms output a set $A' \subseteq A$ such that $G[A']$ is a $\Omega(\phi)$-expander and A' has nearly the same size as A.

We state the main lemma we prove in this section.

Lemma 3.2 (Parallel Trimming). *Given a graph $G = (V, E)$, a parameter $\phi \in \mathbb{R}_{\geq 0}$ and a set $A \subseteq V$ such that $G[A]$ is a ϕ-nearly expander and $|E(A, V \setminus A)| \leq \phi \cdot m$ the algorithm* Trimming$(G = (V, E), A, \phi)$ *outputs a set A' such that*

1. *$G[A']$ is a $\phi/6$ expander*
2. *$\mathbf{deg}_G(A') \geq \mathbf{deg}_G(A) - \frac{4\log^2 n}{\phi}|E(A, V \setminus A)|$*
3. *$|E(A', V \setminus A')| \leq 2 \cdot |E(A, V \setminus A)|$*

in total work $\tilde{O}(m/\phi^2)$ and total span $\tilde{O}(1/\phi^3)$.

Certifying Expansion via Flows. In [SW19] (inspired by [OZ14, NS17, NSWN17]), a flow problem that both guides the pruning and certifies expansion is introduced. We first observe that for a ϕ-nearly expander $G[A]$, we have that $G[A']$ is also a ϕ-nearly expander for all sets $A' \subseteq A$.

Lemma 3.3 (See Proposition 3.2 in [SW19]). *Given a graph $G = (V, E)$ and a set $A \subseteq V$ such that $G[A]$ is a ϕ-nearly expander and there exists a flow f supported on the graph $G[A]$ that routes source $\Delta(v) \stackrel{\text{def}}{=} 2/\phi \cdot (\mathbf{deg}_G(v) - \mathbf{deg}_{G[A]}(v))$ to sinks $\nabla(v) \stackrel{\text{def}}{=} \mathbf{deg}_G(v)$ for $v \in A$ on $G[A]$ with uniform edge capacity $2/\phi$. Then, $G[A]$ is a $\frac{\phi}{6}$-expander.*

Proof. Consider an arbitrary cut $S \subset A$ such that $\mathbf{deg}_G(S) \leq \mathbf{deg}_G(A \setminus S)$. We aim to show that the flow f certifies $|E_{G[A]}(S, A \setminus S)| \geq \frac{\phi}{6} \cdot \mathbf{deg}_G(S)$. To show the lemma by contradiction, we assume

$$|E_{G[A]}(S, A \setminus S)| < \frac{\phi}{6} \cdot \mathbf{deg}_G(S). \tag{1}$$

Since the cut S also satisfies the weaker condition $\mathbf{deg}_G(S) \leq \mathbf{deg}_G(V \setminus S)$, we have $E(S, V \setminus S) \geq \phi \cdot \mathbf{deg}_G(S)$. Therefore, we have that

$$\begin{aligned}
\sum_{v \in S} \Delta(v) &= \frac{2}{\phi} \cdot \sum_{v \in S} (\mathbf{deg}_G(v) - \mathbf{deg}_{G[A]}(v)) \\
&= \frac{2}{\phi} \cdot (|E(S, V \setminus S)| - |E_{G[A]}(S, A \setminus S)|) \\
&\stackrel{a)}{\geq} \frac{2}{\phi} \cdot (\phi \cdot \mathbf{deg}_G(S) - \frac{\phi}{6} \cdot \mathbf{deg}_G(S)) = \frac{5}{3} \cdot \mathbf{deg}_G(S)
\end{aligned}$$

where a) follows from our assumption (1) and Definition 3.1. Therefore the total source mass originating inside S is at least $\frac{5}{3} \cdot \mathbf{deg}_G(S)$. Since the total amount of source capacity inside S is $\sum_{v \in S} \nabla(v) = \mathbf{deg}_G(S)$, at least $\frac{2}{3} \cdot \mathbf{deg}_G(S)$ units of flow have to be routed across the cut $E_{G[A]}(S, A \setminus S)$. Every edge has capacity $\frac{2}{\phi}$, and therefore there are at most $\frac{1}{3} \cdot \mathbf{deg}(S)$ units of flow can be routed by (1) which leads contradiction. This concludes the proof. □

3.1 The Algorithm

For our trimming algorithm, we crucially rely on the following parallel implementation of the unit-flow algorithm suggested in [HRW17, SW19] for structured flow problems. More precisely, we use the following result whose proof we defer to Section 4.

Lemma 3.4. *Given a height parameter h and a residual flow instance $\Pi = (G, \boldsymbol{c}, \boldsymbol{\Delta}, \boldsymbol{\nabla})$ where $\boldsymbol{\nabla}(v) \geq \gamma \cdot \mathbf{deg}(v)$ for all vertices $v \in V$ for some $0 < \gamma \leq 1$, $\|\boldsymbol{\Delta}\|_1 \leq 2m$ and $\boldsymbol{\Delta}(v) \leq \eta \cdot \mathbf{deg}_G(v)$ for all $v \in V$, and $\|\boldsymbol{c}\|_\infty \leq \eta$.*

Then, there is a parallel algorithm ParallelUnitFlow$(G, \boldsymbol{c}, \boldsymbol{\Delta}, \boldsymbol{\nabla}, h)$*, that requires work $\tilde{O}(mh\eta/\gamma)$ and span $\tilde{O}(h^2\eta/\gamma)$, and produces a flow $\boldsymbol{f}$ and labeling $l : V \longrightarrow \{0, ..., h\}$ such that:*

(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. $\boldsymbol{f}(u) \geq \boldsymbol{\nabla}(u)/(8 \cdot \log_2 n)$.

(iii) If $l(u) < h$, then there is no excess mass at u, i.e. $\mathbf{ex}^G_{\boldsymbol{\Delta}, \boldsymbol{\nabla}, \boldsymbol{f}}(u) = 0$.

Next, we describe how we obtain a parallel trimming algorithm from the ParallelUnitFlow() subroutine. We refer the reader to detailed pseudo-code given in Algorithm 1. This algorithm is given a graph $G = (V, E)$, a subset A and a parameter ϕ such that $G[A]$ is a ϕ-nearly expander. We let $A_0 \leftarrow A$ and we give each edge in the graph $G[A] = (V, E)$ capacity $\frac{2}{\phi}$ throughout.

After initialization, the algorithm enters the main-loop. Each iteration i of said loop produces a flow $\boldsymbol{f}_i$. To compute the said flow, we call ParallelUnitFlow() on the induced graph $G_{[A_{i-1}]}$ with residual capacities $\boldsymbol{c}_{\boldsymbol{f}_{i-1}}$. We set the source and sink function to the excess of the previous iteration and $\frac{\mathbf{deg}_G(v)}{\log_2(n)}$ respectively. Then we, call ParallelUnitFlow() on this instance with a total of $h \stackrel{\text{def}}{=} \frac{5120}{\phi} \cdot \log_2^2 n \cdot \ln m$ levels to obtain the flow $\boldsymbol{f}_i$. If the excess after this call is 0 the algorithm returns the flow $\boldsymbol{f}_i$ and the level vector l_i and terminates. Otherwise, we aim to find a cut that further reduces excess. To do so, we initialize $S_0 = \{v \in A_{i-1} : l_i(v) \geq h\}$ to be the subset of A_{i-1} that still has excess flow to be routed. Then, in the j-th iteration of a sub-loop we check if $|E_{\boldsymbol{f}_i}(S_j, A_{i-1} \setminus S_j)| \geq \frac{5 \ln m}{h} \cdot \mathbf{deg}_G(S_j)$. If not, we let $A_i \leftarrow A_{i-1} \setminus S_j$ and continue with the next iteration of the main loop. Otherwise, we continue with $S_j = \{v \in A_{i-1} : l_i(v) \geq h - j\}$.

Algorithm 1: Trimming$(G = (V, E), A, \phi)$

$h \stackrel{\text{def}}{=} \frac{5120}{\phi} \cdot \log_2^2 n \cdot \ln m$
$\boldsymbol{c} \leftarrow \frac{2}{\phi} \cdot \mathbf{1}$
$A_0 \leftarrow A$; $\boldsymbol{f}_0 \leftarrow \mathbf{0}$, $\boldsymbol{\Delta}_0 \leftarrow \frac{2}{\phi}(\mathbf{deg}_G[A] - \mathbf{deg}_{G[A]})$; $\boldsymbol{\nabla}_0 \leftarrow \mathbf{0}$; $i \leftarrow 0$
while ***true*** **do** `/* While we do not have a feasible flow. */`
  $i \leftarrow i + 1$
  $\boldsymbol{\nabla}_i \leftarrow \boldsymbol{\nabla}_{i-1} + \frac{1}{\log_2 n}\mathbf{deg}_G(v)[A_{i-1}]$
  $(\boldsymbol{f}'_i, l_i) \leftarrow$ ParallelUnitFlow$(G[A_{i-1}], \boldsymbol{c}_{\boldsymbol{f}_{i-1}}[A_{i-1}], \mathbf{ex}^{G[A_{i-i}]}_{\boldsymbol{f}_{i-1}, \boldsymbol{\Delta}_{i-1}, \boldsymbol{\nabla}_{i-1}}, \frac{\mathbf{deg}_G(v)[A_{i-1}]}{\log_2 n}, h)$
  $\boldsymbol{f}_i \leftarrow \boldsymbol{f}_{i-1} + \boldsymbol{f}'_i$
  if $\mathbf{ex}^{G[A_i]}_{\boldsymbol{f}_i, \boldsymbol{\Delta}_i, \boldsymbol{\nabla}_i} = \mathbf{0}$ **then return** $A' = A_{i-1}$
  $j \leftarrow 0$; $S_0 \leftarrow \{v \in A_{i-1} : l_i(v) = h\}$
  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**
    $j \leftarrow j + 1$; $S_j \leftarrow \{v \in A_{i-1} : l_i(v) \geq h - j\}$
  $A_i \leftarrow A_{i-1} \setminus S_j$.
  $\boldsymbol{\Delta}_i \leftarrow \frac{2}{\phi}(\mathbf{deg}_G[A_i] - \mathbf{deg}_{G[A_i]})$; $\boldsymbol{\nabla}_i \leftarrow \boldsymbol{\nabla}_i[A_i]$

3.2 Correctness

In this section, we argue that $G[A']$ is a $\phi/6$-expander when the algorithm terminates. This almost immediately follows from Lemma 3.3.

Claim 3.5. *If Algorithm 1 terminates for $i \leq \log_2(n)$, then $G[A']$ is a $\phi/6$-expander.*

Proof. If the algorithm terminates in iteration i, then the final flow $\boldsymbol{f}_i$ routes the source $\mathbf{deg}_G(v) - \mathbf{deg}_{G[A']}(v)$ to sinks $\boldsymbol{\nabla}_i(v) = \frac{i \cdot \mathbf{deg}_G(v)}{\log_2 n} \leq \mathbf{deg}_G(v)$ for $v \in A'$ by the definition of our algorithm and the assumption that $i \leq \log_2 n$. Therefore, Lemma 3.3 applies and we conclude that $G[A']$ is a $\phi/6$-expander. □

In the remainder of this section, we show that the algorithm indeed terminates quickly, and that the set A' is not much smaller than A.

3.3 Runtime

We first argue that the calls to ParallelUnitFlow() are of the desired form.

Claim 3.6. *Whenever* ParallelUnitFlow() *is called in Line 7 of Algorithm 1, it requires work $\tilde{O}(m/\phi^2)$ and span $\tilde{O}(1/\phi^3)$.*

Proof. Since we add $\frac{1}{\log_2 n} \cdot \mathbf{deg}_G(v)$ sink before calling the algorithm, Lemma 3.4 applies with $\gamma = 1/\log_2 n$. The desired work and span bounds follow from the definition of $h = \tilde{O}(\phi^{-1})$ and the fact that the residual capacities are never larger than $4/\phi$. □

Then, we show that the while loop at Line 11 terminates in less than h steps.

Claim 3.7. *The while loop at Line 11 terminates in less than h steps.*

Proof. Consider the j-th iteration of the while loop. We observe that by item 1 in Lemma 3.4, all edges leaving S_j in the residual graph have both endpoints in the set S_{j+1}. Therefore, we obtain that $\mathbf{deg}_G(S_{j+1}) \geq \left(1 + \frac{5 \ln m}{h}\right) \mathbf{deg}_G(S_j)$ for every j until the algorithm terminates. Assume for the sake of a contradiction that the algorithm reaches iterate h. Then we have that $\mathbf{deg}_G(S_h) \geq \left(1 + \frac{5 \ln m}{h}\right)^h$ since $\mathbf{deg}_G(S_0) \geq 1$ because the algorithm would have terminated otherwise. But $\left(1 + \frac{5 \ln m}{h}\right)^h \geq n^3$ which is a contradiction since it exceeds the volume of the graph G. Therefore, the while loop at Line 11 terminates in less than h steps. □

To conclude correctness and runtime analysis, we argue that the main loop at Line 4 of the algorithm converges in $\log_2 n$ steps. To do so, we show that the remaining excess is reduced sufficiently in each round of the algorithm.

Claim 3.8. *The main loop at Line 4 of Algorithm 1 terminates after at most $\log_2 n$ steps.*

Proof. The initial excess is at most $\frac{2}{\phi} \cdot m\phi = 2m \leq n^4$. We next show that the remaining excess is reduced by a factor $\frac{1}{32}$ in each iteration.

Let's fix an iteration i, denote the total excess at the end of iteration k by X^k, and let S_j denote the final set the algorithm settles on when the while loop at Line 11 terminates. We have $j < h$ by Claim 3.7. Furthermore, by Lemma 3.4 all vertices in S_j have nearly saturated sinks, and since we

set the sink to $\mathbf{deg}_G(v)/\log_2 n$ for every vertex $v \in A_{i-1}$ we obtain that the total excess X^{i-1} at the end of the previous iteration was at least

$$X^{i-1} \geq \mathbf{deg}_G(S_j)/(8\log_2^2 n) \tag{2}$$

By the termination condition, we obtain that the total number of edges between S_j and $A_{i-1} \setminus S_j$ in the residual graph are at most $\frac{5\ln m}{h} \cdot \mathbf{deg}_G(S_j)$. Only these edges can contribute residual demand, since the flow that the other edges add is already routed away. Each such edge can add at most $4/\phi$ units of flow. Therefore, we have that $X^i \leq \frac{4}{\phi} \cdot \frac{5\ln m}{h} \cdot \mathbf{deg}_G(S_j) = \frac{1}{256\log_2^2 n} \cdot \mathbf{deg}_G(S_j)$. Together with (2) we obtain that $X^i \leq X^{i-1}/32$. Therefore, the algorithm terminates after $\log n$ iterations, since the total remaining excess at this point is at most $n^4/32^{\log_2 n} = n^4/2^{5\log_2 n} = 1/n < 1$. Since all the flows we consider are integral, this means that the flow routes the demand and therefore the loop terminates. This concludes the proof of this claim. □

3.4 Proof of Lemma 3.2

Before we conclude with the proof of Lemma 3.2, we show that the set A' returned by the algorithm is still large as a function of $E(A, V \setminus A)$.

Claim 3.9. $\mathbf{deg}_G(A') \geq \mathbf{deg}_G(A) - \frac{4\log^2 n}{\phi}|E(A, V \setminus A)|$

Proof. The proof follows the template of the proof of Claim 3.8. Every vertex v in $A \setminus A'$ absorbs at least $\mathbf{deg}_G(v)/(8\log_2^2 n)$ flow. Initially, the total source mass is exactly $\frac{2}{\phi} \cdot |E(A, V \setminus A)|$. Whenever we remove a set S_j from A we introduce at most $\frac{4}{\phi} \cdot \frac{5\ln m}{h} \cdot \mathbf{deg}(S_j) = \frac{1}{256\log_2^2 n} \cdot \mathbf{deg}_G(S_j)$ extra source flow. But at the same time, at least $\mathbf{deg}_G(S_j)/(8 \cdot \log_2^2 n)$ flow was absorbed within the removed part S_j. Therefore, the charged amount of flow in the graph got reduced by a factor at least $1/4$. The claim follows from a geometric sum. □

Claim 3.10. $|E(A', V \setminus A')| \leq 2 \cdot |E(A, V \setminus A)|$

Proof. We first notice that the total flow that is initially in the system is exactly $\frac{2}{\phi} \cdot |E(A, V \setminus A)|$. We now show that the total number of new crossing edges when going from sets A_{i-1} to A_i is small. If an edge between S_j and $A_{i-1} \setminus S_j$ carries flow originating inside the pruned part S_j, then the number of edges in the cut stays the same because it was placed there due to an edge between S_j and $V \setminus A$. The total number of other crossing edges is at most twice the number of edges that cross in the residual graph because no more edges can carry flow out that doesn't originate inside. But these edges are therefore at most $\phi\mathbf{deg}_G(S_j)/(64\log_2^2 n)$ in total. On the other hand, at least $\mathbf{deg}_G/(8\log_2^2 n)$ flow was absorbed. Since the initial flow is $\frac{2}{\phi} \cdot |E(A, V \setminus A)|$ we can charge a decline in flow of $2/\phi$ for each such edge and obtain our claimed bound. □

We conclude the section with a proof of the main lemma.

Proof of Lemma 3.2. The first item directly follows from Claim 3.5, the second item directly follows from Claim 3.9 and the third item follows from Claim 3.10. Therefore, it remains to argue that the work and span bounds are correct. By Claim 3.6, the work and span bound is achieved for the individual oracle calls to ParallelUnitFlow. By Claim 3.8, there are at most $\log_2$ such calls, which adds a $\log_2 n$ factor to both work and depth. Since an individual step of ball growing

in Line 11 can be implemented in parallel, determining the final set S_j at iteration i can be implemented in depth $\tilde{O}(1/\phi)$ and work $\tilde{O}(m)$. We recall that the main loop runs for at most $\log_2 n$ iterations by Claim 3.8 and observe that all other operations can be implemented in parallel directly. This yields the desired bounds for work and span. □

4 Parallel Unit Flow

In this section, we show Lemma 3.4, whose proof was deferred in Section 3. We restate the lemma for the readers convenience.

Lemma 3.4. *Given a height parameter h and a residual flow instance $\Pi = (G, \boldsymbol{c}, \boldsymbol{\Delta}, \boldsymbol{\nabla})$ where $\boldsymbol{\nabla}(v) \geq \gamma \cdot \mathbf{deg}(v)$ for all vertices $v \in V$ for some $0 < \gamma \leq 1$, $\|\boldsymbol{\Delta}\|_1 \leq 2m$ and $\boldsymbol{\Delta}(v) \leq \eta \cdot \mathbf{deg}_G(v)$ for all $v \in V$, and $\|\boldsymbol{c}\|_\infty \leq \eta$.*

Then, there is a parallel algorithm ParallelUnitFlow$(G, \boldsymbol{c}, \boldsymbol{\Delta}, \boldsymbol{\nabla}, h)$*, that requires work $\tilde{O}(mh\eta/\gamma)$ and span $\tilde{O}(h^2\eta/\gamma)$, and produces a flow $\boldsymbol{f}$ and labeling $l : V \longrightarrow \{0, ..., h\}$ such that:*

(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. $\boldsymbol{f}(u) \geq \boldsymbol{\nabla}(u)/(8 \cdot \log_2 n)$.

(iii) If $l(u) < h$, then there is no excess mass at u, i.e. $\mathbf{ex}^G_{\boldsymbol{\Delta}, \boldsymbol{\nabla}, \boldsymbol{f}}(u) = 0$.

Algorithm Description. The parallel unit-flow algorithm roughly follows the template already presented in [GT88], and we use the structure of our flow instance to show our work and span bounds. In particular, we exploit that every vertex is a sink proportional to its degree to relate the work to the amount of excess. Furthermore, we note that while our trimming algorithm operates on an undirected graph, unit-flow algorithm operates on a directed residual graph. See Algorithm 2 and Algorithm 3 for pseudo-code.

We are given a directed graph G with capacities $\boldsymbol{c}$, a source vector $\boldsymbol{\Delta}$ and a sink vector $\boldsymbol{\nabla} \geq \gamma \cdot \mathbf{deg}_G$ where $\mathbf{deg}_G$ is an upper bound on both the in- and out-degree of G. At the start a zero flow $\boldsymbol{f}_0 \leftarrow \mathbf{0}$ and a level function $l : V \mapsto \{0, \ldots, h\}$ to $l(v) \leftarrow 0$ for every vertex $v \in V$ are initialized.

Our algorithm goes through stages $i = 1, \ldots, 8 \cdot \log_2 n$. At the i-th stage, we let $\boldsymbol{\nabla}_i \stackrel{\text{def}}{=} \frac{i}{8 \log_2 n} \cdot \boldsymbol{\nabla}$, and let $x_i = \left\| \mathbf{ex}^G_{\boldsymbol{f}_{i-1}, \boldsymbol{\Delta}, \boldsymbol{\nabla}_{i-1}} \right\|_1$ denote the amount of excess flow that has not yet been routed. Then, the stage will attempt to compute a good unit-flow $\boldsymbol{f}'_i$ until it either succeeds and the algorithm terminates, or the remaining excess not at level $h + 1$ is less than $\boldsymbol{x}_i/2$ and the algorithm sets $\boldsymbol{f}_i \leftarrow \boldsymbol{f}_{i-1} + \boldsymbol{f}'_i$ and proceeds with the next stage.

To construct the flow $\boldsymbol{f}'_i$, the algorithm runs the following algorithm on the residual graph G with residual capacities $\boldsymbol{c}_{\boldsymbol{f}_{i-1}}$. It initializes the source to $\mathbf{ex}^G_{\boldsymbol{f}_{i-1}, \boldsymbol{\Delta}, \boldsymbol{\nabla}_{i-1}}$, sinks to $\boldsymbol{\nabla}/8 \log_2 n$. It then goes over the levels $j = h, \ldots, 1$ in a top to bottom order, and pushes all the flow from vertices at level j to vertices at level $j - 1$ until for every vertex v such that $l(v) = j$ either there is no excess left, or all edges from v to vertices at level $j - 1$ are saturated. Finally, after having processed each level, all vertices $v \in V$ for which all edges to vertices at level $l(v) - 1$ are saturated increase there level by one if they are not yet at level $h + 1$. Once the amount of excess not at level $h + 1$ dropped by a factor 2 and we continue with the next stage as described above.

Finally it returns the accumulated flow $\boldsymbol{f}_{8\log_2 n}$ and the level function l.

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

1 $\boldsymbol{f}_0 \leftarrow \mathbf{0}$; $\boldsymbol{\nabla}_0 \leftarrow \mathbf{0}$: $\forall v \in V : l(v) = 0$
2 **for** $i = 1, \ldots, 8 \cdot \log_2 n$ **do**
3 $\quad x_i \leftarrow \sum_{v \in V : l(v) \neq h+1} \mathbf{ex}^G_{\boldsymbol{f}_{i-1}, \boldsymbol{\Delta}, \boldsymbol{\nabla}_{i-1}}(v)$ `/* Non settled excess */`
4 $\quad \boldsymbol{f}'_i \leftarrow \mathbf{0}$; $\boldsymbol{\nabla}_i \leftarrow \frac{1}{8\log_2 n}\boldsymbol{\nabla}$
5 $\quad$ **while** $\sum_{v \in V : l(v) \neq h+1} \mathbf{ex}^G_{\boldsymbol{f}_{i-1}+\boldsymbol{f}'_i, \boldsymbol{\Delta}, \boldsymbol{\nabla}_i}(v) \geq x_i/2$ **do**
6 $\quad\quad (\boldsymbol{f}'_i, l) \leftarrow$ PushThenRelabel$(G, \boldsymbol{c}_{\boldsymbol{f}_{i-1}}, \boldsymbol{f}'_i, \mathbf{ex}^G_{\boldsymbol{f}_{i-1}, \boldsymbol{\Delta}, \boldsymbol{\nabla}_{i-1}}, \boldsymbol{\nabla}_i, h, l)$
7 $\quad \boldsymbol{f}_i \leftarrow \boldsymbol{f}_{i-1} + \boldsymbol{f}'_i$
8 $\forall v \in V$ s.t. $l(v) = h + 1$: $l(v) \leftarrow h$
9 **return** $(\boldsymbol{f}_{8\cdot\log_2 n}, l)$

Algorithm 3: PushThenRelabel$(G, \boldsymbol{c}, \boldsymbol{f}, \boldsymbol{\Delta}, \boldsymbol{\nabla}, h, l)$

1 **for** $j = h \ldots, 1$ **do**
2 $\quad$ 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** $(\boldsymbol{f}, l)$

Proof of Lemma 3.4. We first show that the algorithm runs in the desired work and span.

Claim 4.1 (Runtime)**.** *Given $\boldsymbol{\nabla} \geq \gamma \cdot \mathbf{deg}_G$, $\boldsymbol{\Delta}(v) \leq \eta \cdot \mathbf{deg}_G(v)$ for all $v \in V$, and $\|\boldsymbol{\Delta}\|_1 \leq 2m$ the algorithm* ParallelUnitFlow$(G, \boldsymbol{c}, \boldsymbol{\Delta}, \boldsymbol{\nabla}, h)$ *described in Algorithm 2 can be implemented with total work $\tilde{O}(mh\eta/\gamma)$ and span $\tilde{O}(h^2\eta/\gamma)$ where $\eta \geq \|\boldsymbol{c}\|_\infty$.*

Proof. The outer loop of Algorithm 2 has $\tilde{O}(1)$ iterations by the definition of the algorithm. To show that the number of steps of the while loop at Line 5 is also suitably bounded, we will refer to flow at vertices of level $h + 1$ as settled, and we observe that such flow never leaves the vertex again by the description of our algorithm. In every iteration of the loop at Line 5, every unit of unsettled excess gets both moved and raised by a level. Every individual vertex v can raise at most $\eta \cdot \mathbf{deg}_G(v)(h + 1)$ units of excess flow throughout. This is because it can contain at most $\eta \cdot \mathbf{deg}_G(v)$ excess at any point since the capacities are bounded by η, and it is raised at most $h+1$ times. But every vertex also absorbs $\gamma \mathbf{deg}_G(v)/(8\log_2 n)$ flow before it is every raised. Therefore the total units of flow that raise in level can be at most $O(x_i \cdot \eta \cdot h \cdot \frac{\log_2 n}{\gamma})$ before all the flow is settled in iteration i of the main loop at Line 2. But since each iteration of the inner while loop at Line 5 raises at least $x_i/2$ units of flow until half the flow is settled, the total number of iterations until half the flow is settled are bounded by $O(\eta \cdot h \cdot \frac{\log_2 n}{\gamma})$. Finally each call to Algorithm 3 causes another multiplicative factor of h in the span by the description of the algorithm. Therefore, the total span of the algorithm is $\tilde{O}(h^2\eta/\gamma)$.

Since all work can be charged to flow going down one level the total work of the algorithm is $\tilde{O}(mh\eta/\gamma)$ as claimed, again by the fact that the total increase in level is bounded by $O(x_i \cdot \eta \cdot h \cdot \frac{\log_2 n}{\gamma})$ and $x_i \leq \|\boldsymbol{\Delta}\|_1 \leq 2m$.

This concludes the runtime analysis since all other parts of the algorithm can directly be implemented in work $\tilde{O}(m)$. □

Then, we show correctness.

Claim 4.2 (Correctness)**.** *Given $\|\boldsymbol{\Delta}\|_1 \leq 2m$, the algorithm Algorithm 2 computes a flow $\boldsymbol{f}$ and labeling l such that*

(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. $\boldsymbol{f}(u) \geq \boldsymbol{\nabla}(u)/(8 \cdot \log_2 n)$.

(iii) If $l(u) < h$, then there is no excess mass at u, i.e. $\mathbf{ex}_{\Delta,\nabla,f}(u) = 0$.

Proof. We show the three items separately.

(i) The first item is maintained as an invariant throughout the execution of the algorithm, which follows from the description of the algorithm in Algorithm 3 since we only ever increase the levels of vertices that only have saturated edges to vertices of lower level.

(ii) The second item also directly follows from the description of the algorithm since for a vertex to be asssigned a non-zero level its, sink has to be saturated for at least one iterate i.

(iii) Since the amount of flow that is not yet settled reduces by a factor of two in each round, there is no non-settled flow left over after $8 \log_2 n$ rounds because $2m/2^{8 \log_2 n} < 1$ which directly implies the third item since all settled flow is either routed or at level h after the algorithm terminates.

This concludes our proof. □

We conclude with the proof of the main lemma.

Proof of Lemma 3.4. Follow directly from Claim 4.1 and Claim 4.2. □

References

[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 *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 2204–2212. IEEE, 2023. 2

[BGS20] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1123–1134. IEEE, 2020. 2

[BGS22] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental sssp and approximate min-cost flow in almost-linear time. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1000–1008. IEEE, 2022. 2

[BvdBPG+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 *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022. 2

[CGL+20] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1158–1167. IEEE, 2020. 20

[Chu21] Julia Chuzhoy. Decremental all-pairs shortest paths in deterministic near-linear time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 626–639, 2021. 2

[CK19] Julia Chuzhoy and Sanjeev Khanna. A new algorithm for decremental single-source shortest paths with applications to vertex-capacitated flow and cut problems. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 389–400, 2019. 2

[CK24] Julia Chuzhoy and Sanjeev Khanna. Maximum bipartite matching in $n^{2+o(1)}$ time via a combinatorial algorithm. *STOC'24*, 2024. 2

[CKGS23] Li Chen, Rasmus Kyng, Maximilian Probst Gutenberg, and Sushant Sachdeva. A simple framework for finding balanced sparse cuts via apsp. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 42–55. SIAM, 2023. 18

[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 *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 612–623. IEEE, 2022. 2

[CKL+24] Li Chen, Rasmus Kyng, Yang P Liu, Simon Meierhans, and Maximilian Probst Gutenberg. Almost-linear time algorithms for incremental graphs: Cycle detection, sccs, s-t shortest path, and minimum-cost flow. *STOC'24*, 2024. 2

[CPZ19] Yi-Jun Chang, Seth Pettie, and Hengjie Zhang. Distributed triangle detection via expander decomposition. In *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 821–840, 2019. 2, 20

[CS19] Yi-Jun Chang and Thatchaphol Saranurak. Improved distributed expander decomposition and nearly optimal triangle enumeration. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 66–73, 2019. 1, 3, 4, 20

[CS20] Yi-Jun Chang and Thatchaphol Saranurak. Deterministic distributed expander decomposition and routing with applications in distributed derandomization. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 377–388. IEEE, 2020. 1, 3, 4, 16, 17, 18, 20

[CS21] Julia Chuzhoy and Thatchaphol Saranurak. Deterministic algorithms for decremental shortest paths via layered core decomposition. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2478–2496. SIAM, 2021. 2

[GKS17] Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed mst and routing in almost mixing time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 131–140, New York, NY, USA, 2017. Association for Computing Machinery. 2

[GL18] Mohsen Ghaffari and Jason Li. New Distributed Algorithms in Almost Mixing Time via Transformations from Parallel Algorithms. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 2

[GPPG24] Lars Gottesbüren, Nikos Parotsidis, and Maximilian Probst Gutenberg. Practical expander decomposition. In *ESA (Track B) 2024*, 2024. 2, 3

[GRST21] Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2212–2228. SIAM, 2021. 2

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, oct 1988. 4, 10

[HRW17] Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. *CoRR*, abs/1704.01254, 2017. 6

[HRW20] Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. *SIAM Journal on Computing*, 49(1):1–36, 2020. 3, 4

[JS22] Wenyu Jin and Xiaorui Sun. Fully dynamic st edge connectivity in subpolynomial time. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 861–872. IEEE, 2022. 2

[JST24] Wenyu Jin, Xiaorui Sun, and Mikkel Thorup. Fully dynamic min-cut of superconstant size in subpolynomial time. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2999–3026. SIAM, 2024. 2

[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 *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 217–226. SIAM, 2014. 2

[KMG24] Rasmus Kyng, Simon Meierhans, and Maximilian Probst Gutenberg. A dynamic shortest paths toolbox: Low-congestion vertex sparsifiers and their applications. *STOC'24*, 2024. 2

[KRV09] Rohit Khandekar, Satish Rao, and Umesh Vazirani. Graph partitioning using single commodity flows. *Journal of the ACM (JACM)*, 56(4):1–15, 2009. 3, 16, 17, 20

[KT18] Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic edge connectivity in near-linear time. *Journal of the ACM (JACM)*, 66(1):1–50, 2018. 2

[KVV00] Ravi Kannan, Santosh S. Vempala, and Adrian Vetta. On clusterings - good, bad and spectral. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 367–377. IEEE Computer Society, 2000. 20

[Li21] Jason Li. Deterministic mincut in almost-linear time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 384–395, 2021. 2

[LNP+21] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Vertex connectivity in poly-logarithmic max-flows. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 317–329, 2021. 2

[NS17] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and o (n1/2-ε)-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1122–1129, 2017. 2, 6, 20

[NSWN17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 950–961. IEEE, 2017. 2, 6, 20

[OZ14] Lorenzo Orecchia and Zeyuan Allen Zhu. Flow-based algorithms for local graph clustering. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 1267–1286. SIAM, 2014. 6

[ST04] Daniel A Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 81–90, 2004. 2, 20

[ST13] Daniel A. Spielman and Shang-Hua Teng. A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning. *SIAM Journal on Computing*, 42(1):1–26, 2013. 5, 20

[SW19] Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2616–2635. SIAM, 2019. 1, 2, 3, 4, 6, 16, 20

[vdBCK+24] Jan van den Brand, Li Chen, Rasmus Kyng, Yang P Liu, Simon Meierhans, Maximilian Probst Gutenberg, and Sushant Sachdeva. Almost-linear time algorithms for decremental graphs: Min-cost flow and more via duality. *to appear at FOCS'24*, 2024. 2

[WN17] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1130–1143, 2017. 2, 20

A Proof of Theorem 1.1

A.1 Expander Decompositions by Recursing on Balanced Cuts

In this section, we show how to derive Theorem 1.1 by parallelizing the framework given by [SW19] that was also previously used in the parallel algorithm in [CS20].

The missing key component for this framework is a parallel implementation of the cut-matching game algorithm from [KRV09]. In the next section, we give such an implementation by speeding up an adaption by [CS20] with our techniques from Section 4.

Theorem A.1. *Let $G = (V, E)$ be an m-edge graph, and let $0 < \phi < 1$ be any parameter. There is a randomized algorithm with work $\tilde{O}(m/\phi^2)$ and span $\tilde{O}(1/\phi^4)$ that finds a cut $C \subseteq V$ satisfying $0 \leq \mathbf{deg}_G(C) \leq m$ and $|E_G(C, V \setminus C)| < \frac{\phi \cdot m}{64 \log^2 n}$ and if $\mathbf{deg}_G(C) \leq m/100$ then we also have that $G[V \setminus C]$ is a $\tilde{\Omega}(\phi)$-nearly expander. The algorithm succeeds with high probability.*

Given this algorithm and our trimming procedure from Lemma 3.2, we can now straightforwardly compute an expander decomposition via the algorithm ComputeExpDecomp(G, ϕ) presented in Algorithm 4. The algorithm either finds a balanced sparse cut and then recurses on both sides of the cuts separately, or it identifies a large expander and only needs to recurse on the (smallish) remainder of the graph.

The proof of Theorem 1.1 is now rather straightforward.

We first observe that the recursion depth of the algorithm is $O(\log n)$ since if the algorithm enters the if-statement the subgraph $G[C], G[V \setminus C]$ both have at most $(1 - 1/100)m$ edges by Theorem A.1. If the algorithm enters the else-statement, it only recurses on $G[V \setminus A']$ which

Algorithm 4: ComputeExpDecomp(G, ϕ)

1 $C \leftarrow$ Cut-Matching(G, ϕ).
2 **if** $\deg_G(C) > m/100$ **then**
3 | **return** ComputeExpDecomp$(G[C], \phi) \cup$ ComputeExpDecomp$(G[V \setminus C], \phi)$.
4 **else**
5 | $A' \leftarrow$ Trimming$(G, V \setminus C, \phi)$.
6 | **return** $\{A'\} \cup$ ComputeExpDecomp$(G[V \setminus A'], \phi)$.

consists of at most $\frac{51}{100} \cdot m$ edges as can be derived from combining the guarantees of Theorem A.1 and Lemma 3.2. This yields immediately that the runtime is $\tilde{O}(m/\phi^2)$ and span is $\tilde{O}(1/\phi^4)$.

For correctness, it suffices to see that each set A' that is finally put into the expander hierarchy is placed there in Line 6. It can be seen from Theorem A.1 and Lemma 3.2 that each such set has the property that $G[A']$ is an $\tilde{\Omega}(\phi)$-expander. To bound the total error, observe that the error is upper bounded by the number of edges leaving the clusters on which we recurse in each step. But in each invocation of ComputeExpDecomp(G, ϕ) on an m-edge graph there are at most $O(\phi m)$ edges crossing this cut. Since the number of edges inputted to all such invocations is $O(m \log n)$ by the bound on the recursion depth and the fact that we recurse on disjoint subgraphs, we obtain an error bound of $\tilde{O}(\phi m)$, as desired.

Finally, we observe that since Theorem A.1 succeeds with high probability, and is only called at most n times, we have that the algorithm succeeds with high probability overall.

A.2 Implementing the Cut-Matching Algorithm

In this section, we prove Theorem A.1. We note that this theorem is obtained by parallelizing the cut-matching algorithm from [KRV09]. This was previously done also in the work of [CS20] (see Lemma 3.1), however, their proof loses heavily in terms of work, span, and error.

However, following the proof of Lemma 3.1 in [CS20], one can verify that the only bottleneck is the computations of the flows in the cut-matching problem. In each of the $O(\log^2 n)$ iterations of the cut-matching algorithm that is used internally, one has to solve a flow problem that puts at most one unit of source or one unit of sink at each vertex and were edge capacities are again of order $\tilde{O}(1/\phi)$.

But we observe that our algorithm from Section 4 can also be used to solve this flow problem efficiently. We can prove the following theorem which tightens the key flow result from [CS20] to achieve much better work, span and error. We defer the proof to Appendix A.3.

Theorem A.2 (Alternative cut-or-match, see Lemma D.7 of [CS20])**.** *Consider a graph $G = (V, E)$ with degree bounded by 16 and a parameter $0 < \phi < 1$. Given a set of source vertices S and a set of sink vertices T with $|S| \leq |T|$, there is an algorithm that finds a cut C and a set of S-T paths $\mathcal{P}$ embedding a (possibly non-perfect) matching M between S and T such that:*

- *every edge in G appears on at most $\tilde{O}(1/\phi)$ paths in $\mathcal{P}$, and*
- *C contains all vertices in S that are not the starting vertex of a path in $\mathcal{P}$, and*
- *C does not contain any vertex in T that is not the last vertex of a path in $\mathcal{P}$, and*

- *either* $C = \emptyset$, *or* $|E_G(C, V \setminus C)| \leq \phi \cdot \mathbf{deg}_G(C) + 2 \cdot \phi \cdot m$.

The algorithm requires $\tilde{O}(m/\phi^2)$ *work and span* $\tilde{O}(1/\phi^4)$.

Using the above flow lemma in-lieu of Lemma D.7 of [CS20] in the proof of Lemma 3.1 in [CS20], we derive a parallel cut-matching algorithm that requires us to run the above algorithm $\tilde{O}(1)$ times and implements the rest of the framework in $\tilde{O}(m/\phi)$ work and $O(1/\phi)$ span. We point out that Theorem A.2 has an additive term of $2\phi m$ on the right-hand side of the inequality in the last bullet point, that is not present in Lemma D.7. However, it can be verified in the proof of Lemma 3.1 in [CS20] that this additive term does not change any of the claim guarantees asymptotically.

We note that technically speaking, the Lemma 3.1 in [CS20] is only proven for bounded-degree graphs, however, a transformation between general graphs and bounded-degree graphs that lifts this result (at the loss of constant factors) is folklore and can be implemented with $\tilde{O}(m)$ work and $\tilde{O}(1)$ span (see for example [CKGS23]).

A.3 Implementing the Alternative Cut-Or-Match Lemma

Finally, we prove the following lemma using our techniques from Section 4.

Theorem A.2 (Alternative cut-or-match, see Lemma D.7 of [CS20])**.** *Consider a graph* $G = (V, E)$ *with degree bounded by* 16 *and a parameter* $0 < \phi < 1$. *Given a set of source vertices* S *and a set of sink vertices* T *with* $|S| \leq |T|$, *there is an algorithm that finds a cut* C *and a set of* S-T *paths* $\mathcal{P}$ *embedding a (possibly non-perfect) matching* M *between* S *and* T *such that:*

- *every edge in* G *appears on at most* $\tilde{O}(1/\phi)$ *paths in* $\mathcal{P}$, *and*
- C *contains all vertices in* S *that are not the starting vertex of a path in* $\mathcal{P}$, *and*
- C *does not contain any vertex in* T *that is not the last vertex of a path in* $\mathcal{P}$, *and*
- *either* $C = \emptyset$, *or* $|E_G(C, V \setminus C)| \leq \phi \cdot \mathbf{deg}_G(C) + 2 \cdot \phi \cdot m$.

The algorithm requires $\tilde{O}(m/\phi^2)$ *work and span* $\tilde{O}(1/\phi^4)$.

We prove Theorem A.2 via a simple adaptation of the algorithm presented in Section 4. See Algorithm 5 for pseudo-code. Our algorithm initializes a flow instance with source $\mathbf{1}_S$ and sink $\mathbf{1}_T$. The algorithm roughly follows the template of the trimming algorithm described in Section 3 and Algorithm 5. We again introduce a level function for $h \stackrel{\text{def}}{=} 100 \cdot \log_2(m)/\phi$ pointing to levels $0, \ldots, h+1$, and we say flow is settled when it is at level $h+1$ since we never move it from there. Then, we run a parallel unit flow algorithm as long as $\phi \cdot m$ flow remains not settled, and we show that this implies that every round makes a lot of progress. Finally, once most of the flow is settled, we output the set C as a combination of the origin of all unsettled flow, and a sparse cut in the level graph which we show to be a sparse cut in the original graph as well.

Algorithm 5: ParallelMatching(G, S, T)

$\boldsymbol{f} \leftarrow \mathbf{0}$; $\forall v \in V : l(v) \leftarrow 0$; $\boldsymbol{c} \leftarrow \frac{2}{\phi} \cdot \mathbf{1}$; $h \leftarrow 100 \cdot \log_2(m)/\phi$

while $\sum_{v \in V : l(v) \neq h+1} \mathbf{ex}^G_{\boldsymbol{f}, \mathbf{1}_S, \mathbf{1}_T}(v) \geq \phi \cdot m/16$ **do**

  $\boldsymbol{t} \leftarrow \max(-\boldsymbol{B}^T \boldsymbol{f} - \mathbf{1}_S + \mathbf{1}_T, \mathbf{0})$

  $(\boldsymbol{f}, l) \leftarrow$ PushThenRelabel$(G, \boldsymbol{c}_{\boldsymbol{f}_{i-1}}, \boldsymbol{f}, \mathbf{ex}^G_{\boldsymbol{f}, \mathbf{1}_S, \mathbf{1}_T}, \boldsymbol{t}, h, l)$

$\forall v \in V$ such that $l(v) = h + 1$: $l(v) \leftarrow h$.

$j \leftarrow 0$

if $S_0 \neq \emptyset$ **then**

  $S_0 \leftarrow \{v \in V : l(v) = h\}$;

  while $|E_{\boldsymbol{f}}(S_j, V \setminus S_j)| \geq \frac{\phi}{4} \mathbf{deg}_G(S_j)$ **do**

    $j \leftarrow j + 1$; $S_j \leftarrow \{v \in V : l(v) \geq h + 1 - j\}$

else

  $S_j \leftarrow \emptyset$

Let $\mathcal{P}$ be the path decomposition of all but a ϕ fraction of $\boldsymbol{f}$.

Let $U \subseteq S$ be the set of all the vertices that don't have a path leaving in $\mathcal{P}$.

$C \leftarrow S_j \cup U$

return $(\mathcal{P}, C)$

Proof of Theorem A.2. We first show that the while loop at Line 2 terminates in $\tilde{O}(1/\phi^3)$ iterations. Initially there are at most m units of unsettled excess in the graph, where we will refer to excess at level $h + 1$ as settled since it never gets routed away again. We notice that each round of PushThenRelabel() increases the level of every unit of unsettled flow by one. But, every vertex can increase at most $\tilde{O}(1/\phi)$ units of flow whenever it increases its level, and therefore the total amount of flow increase is at most $O(m/\phi^2)$. Since every iteration increases at least ϕm units of flow, this process converges after $\tilde{O}(1/\phi^3)$ iterations. Therefore, the depth of the while loop at Line 2 is at most $\tilde{O}(\phi^{-4})$ and the total work is $\tilde{O}(m/\phi^2)$ since each iteration of *Algorithm 3* can be implemented in $\tilde{O}(\phi^{-1})$ depth and we can pay for work with a level decrease, and there are at most h.

We then consider the second while loop. We show that this loop has to converge in less than h iterations. Since the set S_0 is not empty and the graph is connected, we have that $\mathbf{deg}_G(S_0) \geq 1$, and because the residual graph contains all edges between S_j and S_{j+1} which follows directly from the description of the PushThenRelabel() routine described in Algorithm 3, we have that $\mathbf{deg}_G(S_{j+1}) \geq (1 + \phi/4)\mathbf{deg}_G(S_j)$ and therefore $\mathbf{deg}_G(S_{j+1}) > m$ which is a contradiction. Therefore, this part can be implemented with total work $\tilde{O}(m)$ and depth $\tilde{O}(\phi^{-1})$ as well.

If we were to compute the flow decomposition naively, we would obtain a congestion bound of $\tilde{O}(\phi^{-2})$ since an edge can be routed through in opposing directions up to h times. But we can remedy this issue by arbitrarily pairing up paths whenever they cross in both directions over an edge and swapping their tails. This can cause some paths to be very long, but at most a ϕ fraction of the m paths can be longer than $\tilde{O}(\phi^{-2})$ by Markov's inequality. Therefore, we can just explore all paths in parallel and drop the paths that are too short.

Finally, we show a bound on the cut $E(C, V \setminus C)$. The U component of C has volume at most $2\phi m$ since there is at most ϕm unsettled flow and at most ϕm paths that are dropped, and can therefore never pose an issue. Therefore, we just bound the size of the cut $E(S_j, V \setminus S_j)$. We notice that the size of the cut in the residual graph is at most $E_{\boldsymbol{f}}(S_j, V \setminus S_j) \leq \frac{\phi}{4} \mathbf{deg}_G(S_j)$ when the

algorithm terminates. All the other edges are saturated, but the total flow inside S_j is at most $\frac{1}{2}\mathbf{deg}_G(S_j)$ that could get routed in and $\mathbf{deg}_G(S_j)$ flow that originated inside. But this amount of flow can saturate at most $\frac{3}{2} \cdot \frac{\phi}{2} \cdot \mathbf{deg}_G(S_j) = \frac{3\phi}{4}\mathbf{deg}_G(S_j)$. Therefore, $|E(S_j, V \setminus S_j) \leq \phi \cdot \mathbf{deg}_G(S_j)$ as desired. This concludes the proof □

B Review of Algorithms for Expander Decompositions

Sequential Algorithms for Expander Decompositions. Expander decompositions were first introduced as a graph clustering framework in [KVV00]. The first nearly-linear time sequential algorithm for computing nearly-expander decompositions was presented in [ST13] as part of their seminal result on solving Laplacian linear equations [ST04]. While sufficient for their purposes, the algorithm of [ST13] had the defect that clusters were only guaranteed to be contained in a possibly larger unknown expander graph, i.e. nearly-expanders. This defect was remedied by the works of [WN17, NS17, NSWN17], but they instead suffered an almost-linear runtime meaning that the sub-polynomial overhead of their algorithm is larger than poly-logarithmic and a subpolynomial loss in the error of the expander decomposition. Finally, [SW19] presented a sequential algorithm that runs in time $\tilde{O}(m/\phi)$, achieves quality $\tilde{O}(\phi m)$ and ensures that each component is a ϕ-expander. All of the above results however were randomized due to the internal use of the cut-matching algorithm [KRV09]. In [CGL$^+$20], a deterministic cut-matching algorithm was given, effectively derandomizing all previously mentioned algorithms, however, again at the expense of subpolynomial losses in overhead and quality.

Parallel and Distributed Algorithms for Expander Decompositions. In the parallel and distributed setting, [CPZ19] showed that a variant of expander decompositions can be computed quickly, which allowed them to get improved triangle detection algorithms. This culminated in the state of the art parallel algorithms in [CS19, CS20], which nearly achieve the bound of [SW19] but lose a third root in the quality.