Near Linear-Work Parallel SDD Solvers, Low-Diameter Decomposition, and Low-Stretch Subgraphs[*]

Guy E. Blelloch   Anupam Gupta   Ioannis Koutis[†]   Gary L. Miller
Richard Peng   Kanat Tangwongsan

Carnegie Mellon University and [†]University of Puerto Rico, Rio Piedras

Abstract

We present the design and analysis of a near linear-work parallel algorithm for solving symmetric diagonally dominant (SDD) linear systems. On input of a SDD n-by-n matrix A with m non-zero entries and a vector b, our algorithm computes a vector $\tilde{x}$ such that $\|\tilde{x} - A^{+}b\|_A \leq \varepsilon \cdot \|A^{+}b\|_A$ in $O(m \log^{O(1)} n \log \frac{1}{\varepsilon})$ work and $O(m^{1/3+\theta} \log \frac{1}{\varepsilon})$ depth for any fixed $\theta > 0$.

The algorithm relies on a parallel algorithm for generating low-stretch spanning trees or spanning subgraphs. To this end, we first develop a parallel decomposition algorithm that in polylogarithmic depth and $\widetilde{O}(|E|)$ work[1], partitions a graph into components with polylogarithmic diameter such that only a small fraction of the original edges are between the components. This can be used to generate low-stretch spanning trees with average stretch $O(n^{\alpha})$ in $O(n^{1+\alpha})$ work and $O(n^{\alpha})$ depth. Alternatively, it can be used to generate spanning subgraphs with polylogarithmic average stretch in $\widetilde{O}(|E|)$ work and polylogarithmic depth. We apply this subgraph construction to derive a parallel linear system solver. By using this solver in known applications, our results imply improved parallel randomized algorithms for several problems, including single-source shortest paths, maximum flow, minimum-cost flow, and approximate maximum flow.

1 Introduction

Solving a system of linear equations $Ax = b$ is a fundamental computing primitive that lies at the core of many numerical and scientific computing algorithms, including the popular interior-point algorithms. The special case of symmetric diagonally dominant (SDD) systems has seen substantial progress in recent years; in particular, the ground-breaking work of Spielman and Teng showed how to solve SDD systems to accuracy ε in time $\widetilde{O}(m \log(\frac{1}{\varepsilon}))$, where m is the number of non-zeros in the n-by-n-matrix A.[2] This is algorithmically significant since solving SDD systems has implications to computing eigenvectors, solving flow problems, finding graph sparsifiers, and problems in vision and graphics (see [Spi10, Ten10] for these and other applications).

In the sequential setting, the current best SDD solvers run in $O(m \log n (\log \log n)^2 \log(\frac{1}{\varepsilon}))$ time [KMP11]. However, with the exception of the special case of planar SDD systems [KM07], we know of no previous

[*]Contact Address: 5000 Forbes Ave. Computer Science Department. Pittsburgh, PA 15213. E-mail: {`guyb, anupamg, i.koutis, glmiller, yangp, ktangwon`}`@cs.cmu.edu`.

[1]The $\widetilde{O}(\cdot)$ notion hides polylogarithmic factors.

[2]The Spielman-Teng solver and all subsequent improvements are randomized algorithms. As a consequence, all algorithms relying on the solvers are also randomized. For simplicity, we omit standard complexity factors related to the probability of error.

parallel SDD solvers that perform near-linear[3] work and achieve non-trivial parallelism. This raises a natural question: *Is it possible to solve an SDD linear system in $o(n)$ depth and $\widetilde{O}(m)$ work?* This work answers this question affirmatively:

Theorem 1.1 *For any fixed $\theta > 0$ and any $\varepsilon > 0$, there is an algorithm* `SDDSolve` *that on input an $n \times n$ SDD matrix A with m non-zero elements and a vector b, computes a vector $\tilde{x}$ such that $\|\tilde{x} - A^{+}b\|_A \leq \varepsilon \cdot \|A^{+}b\|_A$ in $O(m \log^{O(1)} n \log \frac{1}{\varepsilon})$ work and $O(m^{1/3+\theta} \log \frac{1}{\varepsilon})$ depth.*

In the process of developing this algorithm, we give parallel algorithms for constructing graph decompositions with strong-diameter guarantees, and parallel algorithms to construct low-stretch spanning trees and low-stretch ultra-sparse subgraphs, which may be of independent interest. An overview of these algorithms and their underlying techniques is given in Section 3.

Some Applications. Let us mention some of the implications of Theorem 1.1, obtained by plugging it into known reductions.

— *Construction of Spectral Sparsifiers.* Spielman and Srivastava [SS08] showed that spectral sparsifiers can be constructed using $O(\log n)$ Laplacian solves, and using our theorem we get spectral and cut sparsifiers in $\tilde{O}(m^{1/3+\theta})$ depth and $\tilde{O}(m)$ work.

— *Flow Problems.* Daitsch and Spielman [DS08] showed that various graph optimization problems, such as max-flow, min-cost flow, and lossy flow problems, can be reduced to $\widetilde{O}(m^{1/2})$ applications[4] of SDD solves via interior point methods described in [Ye97, Ren01, BV04]. Combining this with our main theorem implies that these algorithms can be parallelized to run in $\widetilde{O}(m^{5/6+\theta})$ depth and $\widetilde{O}(m^{3/2})$ work. This gives the first parallel algorithm with $o(n)$ depth which is work-efficient to within $\operatorname{polylog}(n)$ factors relative to the sequential algorithm for all problems analyzed in [DS08]. In some sense, the parallel bounds are more interesting than the sequential times because in many cases the results in [DS08] are not the best known sequentially (e.g. max-flow)—but do lead to the best know parallel bounds for problems that have traditionally been hard to parallelize. Finally, we note that although [DS08] does not explicitly analyze shortest path, their analysis naturally generalizes the LP for it.

Our algorithm can also be applied in the inner loop of [CKM+10], yielding a $\tilde{O}(m^{5/6+\theta}\text{poly}(\varepsilon^{-1}))$ depth and $\tilde{O}(m^{4/3}\text{poly}(\varepsilon^{-1}))$ work algorithm for finding $1-\varepsilon$ approximate maximum flows and $1+\varepsilon$ approximate minimum cuts in undirected graphs.

2 Preliminaries and Notation

We use the notation $\widetilde{O}(f(n))$ to mean $O(f(n) \operatorname{polylog}(f(n)))$. We use $A \uplus B$ to denote disjoint unions, and $[k]$ to denote the set $\{1, 2, \ldots, k\}$. Given a graph $G = (V, E)$, let $dist(u, v)$ denote the *edge-count distance* (or hop distance) between u and v, ignoring the edge lengths. When the graph has edge lengths $w(e)$ (also denoted by w_e), let $d_G(u, v)$ denote the *edge-length distance*, the shortest path (according to these edge lengths) between u and v. If the graph has unit edge lengths, the two definitions coincide. We drop subscripts when the context is clear. We denote by $V(G)$ and $E(G)$, respectively, the set of nodes and the set of edges, and use $n = |V(G)|$

[3]i.e. linear up to polylog factors.

[4]here $\tilde{O}$ hides $\log U$ factors as well, where it's assumed that the edge weights are integers in the range $[1 \ldots U]$

and $m = |E(G)|$. For an edge $e = \{u, v\}$, the stretch of e on G' is $\mathsf{str}_{G'}(e) = d_{G'}(u, v)/w(e)$. The *total stretch* of $G = (V, E, w)$ with respect to G' is $\mathsf{str}_{G'}(E(G)) = \sum_{e \in E(G)} \mathsf{str}_{G'}(e)$.

Given $G = (V, E)$, a distance function δ (which is either $dist$ or d), and a partition of V into $C_1 \uplus C_2 \uplus \ldots \uplus C_p$, let $G[C_i]$ denote the induced subgraph on set C_i. The *weak diameter* of C_i is $\max_{u,v \in C_i} \delta_G(u, v)$, whereas the *strong diameter* of C_i is $\max_{u,v \in C_i} \delta_{G[C_i]}(u, v)$; the former measures distances in the original graph whereas the latter measures distances within the induced subgraph. The strong (or weak) diameter of the partition is the maximum strong (or weak) diameter over all the components C_i's.

Graph Laplacians. For a fixed, but arbitrary, numbering of the nodes and edges in a graph $G = (V, E)$, the Laplacian L_G of G is the $|V|$-by-$|V|$ matrix given by

$$L_G(i, j) = \begin{cases} -w_{ij} & \text{if } i \neq j \\ \sum_{\{j,i\} \in E(G)} w_{ij} & \text{if } i = j \end{cases},$$

When the context is clear, we use G and L_G interchangeably. Given two graphs G and H and a scalar $\mu \in \mathbb{R}$, we say $G \preceq \mu H$ if $\mu L_H - L_G$ is positive semidefinite, or equivalently $x^\top L_G x \leq \mu x^\top L_H x$ for all vector $x \in \mathbb{R}^{|V|}$.

Matrix Norms, SDD Matrices. For a matrix A, we denote by A^+ the Moore-Penrose pseudoinverse of A (i.e., A^+ has the same null space as A and acts as the inverse of A on its image). Given a symmetric positive semi-definite matrix A, the A-*norm* of a vector x is defined as $\|x\|_A = \sqrt{x^\top A x}$. A matrix A is symmetric diagonally dominant (SDD) if it is symmetric and for all i, $A_{i,i} \geq \sum_{j \neq i} |A_{i,j}|$. Solving an SDD system reduces in $O(m)$ work and $O(\log^{O(1)} m)$ depth to solving a graph Laplacian (a subclass of SDD matrices corresponding to undirected weighted graphs) [Gre96, Section 7.1].

Parallel Models. We analyze algorithms in the standard PRAM model, focusing on the work and depth parameters of the algorithms. By *work*, we mean the total operation count—and by *depth*, we mean the longest chain of dependencies (i.e., parallel time in PRAM).

Parallel Ball Growing. Let $B_G(s, r)$ denote the ball of edge-count distance r from a source s, i.e., $B_G(s, r) = \{v \in V(G) : dist_G(s, v) \leq r\}$. We rely on an elementary form of parallel breadth-first search to compute $B_G(s, r)$. The algorithm visits the nodes level by level as they are encountered in the BFS order. More precisely, level 0 contains only the source node s, level 1 contains the neighbors of s, and each subsequent level $i + 1$ contains the neighbors of level i's nodes that have not shown up in a previous level. On standard parallel models (e.g., CRCW), this can be computed in $O(r \log n)$ depth and $O(m' + n')$ work, where m' and n' are the total numbers of edges and nodes, respectively, encountered in the search [UY91, KS97]. Notice that we could achieve this runtime bound with a variety of graph (matrix) representations, e.g., using the compressed sparse-row (CSR) format. Our applications apply ball growing on r roughly $O(\log^{O(1)} n)$, resulting in a small depth bound. We remark that the idea of small-radius parallel ball growing has previously been employed in the context of approximate shortest paths (see, e.g., [UY91, KS97, Coh00]). There is an alternative approach of repeatedly squaring a matrix, which can give a better depth bound for large r *at the expense* of a much larger work bound (about n^3).

Finally, we state a tail bound which will be useful in our analysis. This bound is easily derived from well-known facts about the tail of a hypergeometric random variable [Chv79, Hoe63, Ska09].

Lemma 2.1 (Hypergeometric Tail Bound) *Let H be a hypergeometric random variable denoting the number of red balls found in a sample of n balls drawn from a total of N balls of which M are red. Then, if $\mu =$*

$\mathbf{E}[H] = nM/N$, *then*

$$\mathbf{Pr}[H \geq 2\mu] \quad \leq \quad e^{-\mu/4}$$

Proof: We apply the following theorem of Hoeffding [Chv79, Hoe63, Ska09]. For any $t > 0$,

$$\mathbf{Pr}[H \geq \mu + tn] \leq \left(\left(\frac{p}{p+t}\right)^{p+t}\left(\frac{1-p}{1-p-t}\right)^{1-p-t}\right)^n,$$

where $p = \mu/n$. Using $t = p$, we have

$$\begin{aligned}\mathbf{Pr}[H \geq 2\mu] &\leq \left(\left(\frac{p}{2p}\right)^{2p}\left(\frac{1-p}{1-2p}\right)^{1-2p}\right)^n \\ &\leq \left(e^{-p\ln 4}\left(1+\frac{p}{1-2p}\right)^{1-2p}\right)^n \\ &\leq \left(e^{-p\ln 4}\cdot e^{p}\right)^n \\ &\leq e^{-\frac{1}{4}pn},\end{aligned}$$

where we have used the fact that $1 + x \leq \exp(x)$. ■

3 Overview of Our Techniques

In the general solver framework of Spielman and Teng [ST06, KMP10], near linear-time SDD solvers rely on a suitable preconditioning chain of progressively smaller graphs. Assuming that we have an algorithm for generating low-stretch spanning trees, the algorithm as given in [KMP10] parallelizes under the following modifications: (i) perform the partial Cholesky factorization in parallel and (ii) terminate the preconditioning chain with a graph that is of size approximately $m^{1/3}$. The details in Section 6 are the primary motivation of the main technical part of the work in this chapter, a parallel implementation of a modified version of Alon et al.'s low-stretch spanning tree algorithm [AKPW95].

More specifically, as a first step, we find an algorithm to embed a graph into a spanning tree with average stretch $2^{O(\sqrt{\log n \log\log n})}$ in $\widetilde{O}(m)$ work and $O(2^{O(\sqrt{\log n \log\log n})}\log\Delta)$ depth, where Δ is the ratio of the largest to smallest distance in the graph. The original AKPW algorithm relies on a parallel graph decomposition scheme of Awerbuch [Awe85], which takes an unweighted graph and breaks it into components with a specified diameter and few crossing edges. While such schemes are known in the sequential setting, they do not parallelize readily because removing edges belonging to one component might increase the diameter or even disconnect subsequent components. We present the first near linear-work parallel decomposition algorithm that also gives strong-diameter guarantees, in Section 4, and the tree embedding results in Section 5.1.

Ideally, we would have liked for our spanning trees to have a polylogarithmic stretch, computable by a polylogarithmic depth, near linear-work algorithm. However, for our solvers, we make the additional observation that we do not really need a spanning *tree* with small stretch; it suffices to give an "ultra-sparse" graph with small stretch, one that has only $O(m/\operatorname{polylog}(n))$ edges more than a tree. Hence, we present a parallel algorithm in Section 5.2 which outputs an ultra-sparse graph with $O(\operatorname{polylog}(n))$ average stretch, performing $\widetilde{O}(m)$ work with $O(\operatorname{polylog}(n))$ depth. Note that this removes the dependence of $\log\Delta$ in the depth, and reduces both the stretch and the depth from $2^{O(\sqrt{\log n \log\log n})}$ to $O(\mathrm{polylog}(n))$.[5] When combined with the aforementioned routines for constructing a SDD solver presented in Section 6, this low-stretch spanning subgraph construction yields a parallel solver algorithm.

4 Parallel Low-Diameter Decomposition

In this section, we present a parallel algorithm for partitioning a graph into components with low (strong) diameter while cutting only a few edges in each of the k disjoint subsets of the input edges. The sequential version of this algorithm is at the heart of the low-stretch spanning tree algorithm of Alon, Karp, Peleg, and West (AKPW) [AKPW95].

For context, notice that the outer layer of the AKPW algorithm (more details in Section 5) can be viewed as bucketing the input edges by weight, then partitioning and contracting them repeatedly. In this view, a number of edge classes are "reduced" simultaneously in an iteration. Further, as we wish to output a spanning subtree at the end, the components need to have low strong-diameter (i.e., one could not take "shortcuts" through other components). In the sequential case, the strong-diameter property is met by removing components one after another, but this process does not parallelize readily. For the parallel case, we guarantee this by growing balls from multiple sites, with appropriate "jitters" that conceptually delay when these ball-growing processes start, and assigning vertices to the first region that reaches them. These "jitters" terms are crucial in controlling the probability that an edge goes across regions. But this probability also depends on the number of regions that could reach such an edge. To keep this number small, we use a repeated sampling procedure motivated by Cohen's (β, W)-cover construction [Coh93].

More concretely, we prove the following theorem:

Theorem 4.1 (Parallel Low-Diameter Decomposition) *Given an input graph $G = (V, E_1 \uplus \ldots \uplus E_k)$ with k edge classes and a "radius" parameter ρ, the algorithm* `Partition`(G, ρ)*, upon termination, outputs a partition of V into components $\mathcal{C} = (C_1, C_2, \ldots, C_p)$, each with center s_i such that*

1. *the center $s_i \in C_i$ for all $i \in [p]$,*
2. *for each i, every $u \in C_i$ satisfies $dist_{G[C_i]}(s_i, u) \le \rho$, and*
3. *for all $j = 1, \ldots, k$, the number of edges in E_j that go between components is at most $|E_j| \cdot \frac{c_1 \cdot k \log^3 n}{\rho}$, where c_1 is an absolute constant.*

Furthermore, `Partition` *runs in $O(m \log^2 n)$ expected work and $O(\rho \log^2 n)$ expected depth.*

4.1 Low-Diameter Decomposition for Simple Unweighted Graphs

To prove this theorem, we begin by presenting an algorithm `splitGraph` that works with simple graphs with only *one* edge class and describe how to build on top of it an algorithm that handles multiple edge classes.

The basic algorithm takes as input a simple, unweighted graph $G = (V, E)$ and a radius (in hop count) parameter ρ and outputs a partition V into components $C_1, \ldots, C_p$, each with center s_i, such that

[5]As an aside, this construction of low-stretch ultra-sparse graphs shows how to obtain the $\widetilde{O}(m)$-time linear system solver of Spielman and Teng [ST06] without using their low-stretch spanning trees result [EEST05, ABN08].

(P1) Each center belongs to its own component. That is, the center $s_i \in C_i$ for all $i \in [p]$;

(P2) Every component has radius at most ρ. That is, for each $i \in [p]$, every $u \in C_i$ satisfies $dist_{G[C_i]}(s_i, u) \leq \rho$;

(P3) Given a technical condition (to be specified) that holds with probability at least $3/4$, the probability that an edge of the graph G goes between components is at most $\frac{136}{\rho} \log^3 n$.

In addition, this algorithm runs in $O(m \log^2 n)$ expected work and $O(\rho \log^2 n)$ expected depth. (These properties should be compared with the guarantees in Theorem 4.1.)

Consider the pseudocode of this basic algorithm in Algorithm 4.1. The algorithm takes as input an unweighted n-node graph G and proceeds in $T = O(\log n)$ iterations, with the eventual goal of outputting a partition of the graph G into a collection of sets of nodes (each set of nodes is known as a component). Let $G^{(t)} = (V^{(t)}, E^{(t)})$ denote the graph at the beginning of iteration t. Since this graph is unweighted, the distance in this algorithm is always the hop-count distance $dist(\cdot, \cdot)$. For iteration $t = 1, \dots, T$, the algorithm picks a set of starting centers $S^{(t)}$ to grow balls from; as with Cohen's (β, W)-cover, the number of centers is progressively larger with iterations, reminiscent of the doubling trick (though with more careful handling of the growth rate), to compensate for the balls' shrinking radius and to ensure that the graph is fully covered.

Still within iteration t, it chooses a random "jitter" value $\delta_s^{(t)} \in_R \{0, 1, \dots, R\}$ for each of the centers in $S^{(t)}$ and grows a ball from each center s out to radius $r^{(t)} - \delta_s^{(t)}$, where $r^{(t)} = \frac{\rho}{2 \log n}(T - t + 1)$. Let $X^{(t)}$ be the union of these balls (i.e., the nodes "seen" from these starting points). In this process, the "jitter" should be thought of as a random amount by which we delay the ball-growing process on each center, so that we could assign nodes to the first region that reaches them while being in control of the number of cross-component edges. Equivalently, our algorithm forms the components by assigning each vertex u reachable from one of these centers to the center that minimizes $dist_{G^{(t)}}(u, s) + \delta_s^{(t)}$ (ties broken in a consistent manner, e.g., lexicographically). Note that because of these "jitters," some centers might not be assigned any vertex, not even itself. For centers that are assigned some nodes, we include their components in the output, designating them as the components' centers. Finally, we construct $G^{(t+1)}$ by removing nodes that were "seen" in this iteration (i.e., the nodes in $X^{(t)}$)—because they are already part of one of the output components—and adjusting the edge set accordingly.

Analysis. Throughout this analysis, we make reference to various quantities in the algorithm and assume the reader's basic familiarity with our algorithm. We begin by proving properties (P1)–(P2). First, we state an easy-to-verify fact, which follows immediately by our choice of radius and components' centers.

Fact 4.2 *If vertex u lies in component $C_s^{(t)}$, then $dist^{(t)}(s, u) \leq r^{(t)}$. Moreover, $u \in B_s^{(t)}$.*

We also need the following lemma to argue about strong diameter.

Lemma 4.3 *If vertex $u \in C_s^{(t)}$, and vertex $v \in V^{(t)}$ lies on any u-s shortest path in $G^{(t)}$, then $v \in C_s^{(t)}$.*

Proof: Since $u \in C_s^{(t)}$, Fact 4.2 implies u belongs to $B_s^{(t)}$. But $dist^{(t)}(v, i) < dist^{(t)}(u, i)$, and hence v belongs to $B_s^{(t)}$ and $X^{(t)}$ as well. This implies that v is assigned to *some* component $C_j^{(t)}$; we claim $j = s$.

For a contradiction, assume that $j \neq s$, and hence $dist^{(t)}(v, j) + \delta_j^{(t)} \leq dist^{(t)}(v, s) + \delta_s^{(t)}$. In this case $dist^{(t)}(u, j) + \delta_j^{(t)} \leq dist^{(t)}(u, v) + dist^{(t)}(v, j) + \delta_j^{(t)}$ (by the triangle inequality). Now using the assumption,

Algorithm 4.1 `splitGraph` $(G = (V, E), \rho)$ — Split an input graph $G = (V, E)$ into components of hop-radius at most ρ.

```
Let $G^{(1)} = (V^{(1)}, E^{(1)}) \leftarrow G$. Define $R = \rho/(2\log n)$. Create empty collection of components $\mathcal{C}$.
Use $dist^{(t)}$ as shorthand for $dist_{G^{(t)}}$, and define $B^{(t)}(u, r) \overset{\text{def}}{=} B_{G^{(t)}}(u, r) = \{v \in V^{(t)} \mid dist^{(t)}(u, v) \le r\}$.
For $t = 1, 2, \ldots, T = 2\log_2 n$,
    1. Randomly sample $S^{(t)} \subseteq V^{(t)}$, where $|S^{(t)}| = \sigma_t = 12n^{t/T-1}|V^{(t)}|\log n$, or use $S^{(t)} = V^{(t)}$ if $|V^{(t)}| < \sigma_t$.
    2. For each "center" $s \in S^{(t)}$, draw $\delta_s^{(t)}$ uniformly at random from $\mathbb{Z} \cap [0, R]$.
    3. Let $r^{(t)} \leftarrow (T - t + 1)R$.
    4. For each center $s \in S^{(t)}$, compute the ball $B_s^{(t)} = B^{(t)}(s, r^{(t)} - \delta_s^{(t)})$.
    5. Let $X^{(t)} = \cup_{s \in S^{(t)}} B_s^{(t)}$.
    6. Create components $\{C_s^{(t)} \mid s \in S^{(t)}\}$ by assigning each $u \in X^{(t)}$ to the component $C_s^{(t)}$ such that $s$ minimizes
       $dist_{G^{(t)}}(u, s) + \delta_s^{(t)}$ (breaking ties lexicographically).
    7. Add non-empty $C_s^{(t)}$ components to $\mathcal{C}$.
    8. Set $V^{(t+1)} \leftarrow V^{(t)} \setminus X^{(t)}$, and let $G^{(t+1)} \leftarrow G^{(t)}[V^{(t+1)}]$. Quit early if $V^{(t+1)}$ is empty.
Return $\mathcal{C}$.
```

this expression is at most $dist^{(t)}(u, v) + dist^{(t)}(v, s) + \delta_s^{(t)} = dist^{(t)}(u, s) + \delta_s^{(t)}$ (since v lies on the shortest u-s path). But then, u would be also assigned to $C_j^{(t)}$, a contradiction. ■

Hence, for each non-empty component $C_s^{(t)}$, its center s lies within the component (since it lies on the shortest path from s to any $u \in C_s^{(t)}$), which proves (P1). Moreover, by Fact 4.2 and Lemma 4.3, the (strong) radius is at most TR, proving (P2). It now remains to prove (P3), and the work and depth bounds.

Lemma 4.4 *For any vertex $u \in V$, with probability at least $1 - n^{-6}$, there are at most $68\log^2 n$ pairs*[6] (s, t) *such that $s \in S^{(t)}$ and $u \in B^{(t)}(s, r^{(t)})$,*

We will prove this lemma in a series of claims.

Claim 4.5 *For $t \in [T]$ and $v \in V^{(t)}$, if $|B^{(t)}(v, r^{(t+1)})| \ge n^{1-t/T}$, then $v \in X^{(t)}$ w.p. at least $1 - n^{-12}$.*

Proof: First, note that for any $s \in S^{(t)}$, $r^{(t)} - \delta_s \ge r^{(t)} - R = r^{(t+1)}$, and so if $s \in B^{(t)}(v, r^{(t+1)})$, then $v \in B_s^{(t)}$ and hence in $X^{(t)}$. Therefore,

$$\textbf{Pr}\left[v \in X^{(t)}\right] \ge \textbf{Pr}\left[S^{(t)} \cap B^{(t)}(v, r^{(t+1)}) \ne \emptyset\right],$$

which is the probability that a random subset of $V^{(t)}$ of size σ_t hits the ball $B^{(t)}(v, r^{(t+1)})$. But,

$$\textbf{Pr}\left[S^{(t)} \cap B^{(t)}(v, r^{(t+1)}) \ne \emptyset\right] \ge 1 - \left(1 - \frac{|B^{(t)}(v, r^{(t+1)})|}{|V^{(t)}|}\right)^{\sigma_t},$$

which is at least $1 - n^{-12}$. ■

Claim 4.6 *For $t \in [T]$ and $v \in V$, the number of $s \in S^{(t)}$ such that $v \in B^{(t)}(s, r^{(t)})$ is at most $34\log n$ w.p. at least $1 - n^{-8}$.*

[6] In fact, for a given s, there is a unique t—if this s is ever chosen as a "starting point."

Proof: For $t = 1$, the size $\sigma_1 = O(\log n)$ and hence the claim follows trivially. For $t \geq 2$, we condition on all the choices made in rounds $1, 2, \ldots, t-2$. Note that if v does not survive in $V^{(t-1)}$, then it does not belong to $V^{(t)}$ either, and the claim is immediate. So, consider two cases, depending on the size of the ball $B^{(t-1)}(v, r^{(t)})$ in iteration $t-1$:

— *Case 1.* If $|B^{(t-1)}(v, r^{(t)})| \geq n^{1-(t-1)/T}$, then by Claim 3.5, with probability at least $1 - n^{-12}$, we have $v \in X^{(t-1)}$, so v would *not* belong to $V^{(t)}$ and this means **no** $s \in S^{(t)}$ will satisfy $v \in B^{(t)}(s, r^{(t)})$, proving the claim for this case.

— *Case 2.* Otherwise, $|B^{(t-1)}(v, r^{(t)})| < n^{1-(t-1)/T}$. We have

$$|B^{(t)}(v, r^{(t)})| \leq |B^{(t-1)}(v, r^{(t)})| < n^{1-(t-1)/T}$$

as $B^{(t)}(v, r^{(t)}) \subseteq B^{(t-1)}(v, r^{(t)})$. Now let X be the number of s such that $v \in B^{(t)}(s, r^{(t)})$, so $X = \sum_{s \in S^{(t)}} \mathbf{1}_{\{s \in B^{(t)}(v, r^{(t)})\}}$. Over the random choice of $S^{(t)}$,

$$\mathbf{Pr}\left[s \in B^{(t)}(v, r^{(t)})\right] = \frac{|B^{(t)}(v, r^{(t)})|}{|V^{(t)}|} \leq \frac{1}{|V^{(t)}|} n^{1-(t-1)/T},$$

which gives

$$\mathbf{E}\left[X\right] = \sigma_t \cdot \mathbf{Pr}\left[s \in B^{(t)}(v, r^{(t)})\right] \leq 17 \log n.$$

To obtain a high probability bound for X, we will apply the tail bound in Lemma 2.1. Note that X is simply a hypergeometric random variable with the following parameters setting: total balls $N = |V^{(t)}|$, red balls $M = |B^{(t)}(v, r^{(t)})|$, and the number balls drawn is σ_t. Therefore, $\mathbf{Pr}\left[X \geq 34 \log n\right] \leq \exp\{-\frac{1}{4} \cdot 34 \log n\}$, so $X \leq 34 \log n$ with probability at least $1 - n^{-8}$.

Hence, regardless of what choices we made in rounds $1, 2, \ldots, t-2$, the conditional probability of seeing more than $34 \log n$ different s's is at most n^{-8}. Hence, we can remove the conditioning, and the claim follows. ■

Lemma 4.7 *If for each vertex $u \in V$, there are at most $68 \log^2 n$ pairs (s, t) such that $s \in S^{(t)}$ and $u \in B^{(t)}(s, r^{(t)})$, then for an edge uv, the probability that u belongs to a different component than v is at most $68 \log^2 n / R$.*

Proof: We define a center $s \in S^{(t)}$ as "separating" u and v if $|B_s^{(t)} \cap \{u, v\}| = 1$. Clearly, if u, v lie in different components then there is some $t \in [T]$ and some center s that separates them. For a center $s \in S^{(t)}$, this can happen only if $\delta_s = R - dist(s, u)$, since $dist(s, v) \leq dist(s, u) - 1$. As there are R possible values of δ_s, this event occurs with probability at most $1/R$. And since there are only $68 \log^2 n$ different centers s that can possibly cut the edge, using a trivial union bound over them gives us an upper bound of $68 \log^2 n / R$ on the probability. ■

To argue about (P3), notice that the premise to Lemma 4.7 holds with probability exceeding $1 - o(1) \geq 3/4$. Combining this with Lemma 4.4 proves property (P3), where the technical condition is the premise to Lemma 4.7.

Finally, we consider the work and depth of the algorithm. These are randomized bounds. Each computation of $B^{(t)}(v, r^{(t)})$ can be done using a BFS. Since $r^{(t)} \leq \rho$, the depth is bounded by $O(\rho \log n)$ per iteration, resulting in $O(\rho \log^2 n)$ after $T = O(\log n)$ iterations. As for work, by Lemma 4.4, each vertex is reached by at most $O(\log^2 n)$ starting points, yielding a total work of $O(m \log^2 n)$.

4.2 Low-Diameter Decomposition for Multiple Edge Classes

Extending the basic algorithm to support multiple edge classes is straightforward. The main idea is as follows. Suppose we are given a unweighted graph $G = (V, E)$, and the edge set E is composed of k edge classes $E_1 \uplus \cdots \uplus E_k$. So, if we run `splitGraph` on $G = (V, E)$ and ρ treating the different classes as one, then property (P3) indicates that each edge—regardless of which class it came from—is separated (i.e., it goes across components) with probability $p = \frac{136}{\rho} \log^3 n$. This allows us to prove the following corollary, which follows directly from Markov's inequality and the union bounds.

Corollary 4.8 *With probability at least $1/4$, for all $i \in [k]$, the number of edges in E_i that are between components is at most $|E_i|\frac{272k\log^3 n}{\rho}$.*

The corollary suggests a simple way to use `splitGraph` to provide guarantees required by Theorem 4.1: as summarized in Algorithm 4.2, we run `splitGraph` on the input graph treating all edge classes as one and repeat it if any of the edge classes had too many edges cut (i.e., more than $|E_i|\frac{272k\log^3 n}{\rho}$). As the corollary indicates, the number of trials is a geometric random variable with with $p = 1/4$, so in expectation, it will finish after 4 trials. Furthermore, although it could go on forever in the worst case, the probability does fall exponentially fast.

Algorithm 4.2 `Partition` $(G = (V, E = E_1 \uplus \cdots \uplus E_k), \rho)$ — Partition an input graph G into components of radius at most ρ.

```
1. Let $\mathcal{C}$ = splitGraph($(V, \uplus E_i), \rho$).
2. If there is some $i$ such that $E_i$ has more than $|E_i|\frac{272 \cdot k\log^3 n}{\rho}$ edges between components, start over. (Recall
   that $k$ was the number of edge classes.)
Return $\mathcal{C}$.
```

Finally, we note that properties (P1) and (P2) directly give Theorem 4.1(1)–(2)—and the validation step in `Partition` ensures Theorem 4.1(3), setting $c_1 = 272$. The work and depth bounds for `Partition` follow from the bounds derived for `splitGraph` and Corollary 4.8. This concludes the proof of Theorem 4.1.

5 Parallel Low-Stretch Spanning Trees and Subgraphs

This section presents parallel algorithms for low-stretch spanning trees and for low-stretch spanning subgraphs. To obtain the low-stretch spanning tree algorithm, we apply the construction of Alon et al. [AKPW95] (henceforth, the AKPW construction), together with the parallel graph partition algorithm from the previous section. The resulting procedure, however, is not ideal for two reasons: the depth of the algorithm depends on the "spread" Δ—the ratio between the heaviest edge and the lightest edge—and even for polynomial spread, both the depth and the average stretch are super-logarithmic (both of them have a $2^{O(\sqrt{\log n \cdot \log\log n})}$ term). Fortunately, for our application, we observe that we do not need spanning trees but merely low-stretch sparse graphs. In Section 5.2, we describe modifications to this construction to obtain a parallel algorithm which computes sparse subgraphs that give us only polylogarithmic average stretch and that can be computed in polylogarithmic depth and $\widetilde{O}(m)$ work. We believe that this construction may be of independent interest.

5.1 Low-Stretch Spanning Trees

Using the AKPW construction, along with the `Partition` procedure from Section 4, we will prove the following theorem:

Theorem 5.1 (Low-Stretch Spanning Tree) *There is an algorithm* `AKPW`(G) *which given as input a graph* $G = (V, E, w)$, *produces a spanning tree in* $O(\log^{O(1)} n \cdot 2^{O(\sqrt{\log n \cdot \log\log n})} \log \Delta)$ *expected depth and* $\widetilde{O}(m)$ *expected work such that the total stretch of all edges is bounded by* $m \cdot 2^{O(\sqrt{\log n \cdot \log\log n})}$.

Algorithm 5.1 `AKPW` $(G = (V, E, w))$ — a low-stretch spanning tree construction.

```
i.   Normalize the edges so that $\min\{w(e) : e \in E\} = 1$.
ii.  Let $y = 2^{\sqrt{6 \log n \cdot \log\log n}}$, $\tau = \lceil 3\log(n)/\log y \rceil$, $z = 4c_1 y\tau \log^3 n$. Initialize $T = \emptyset$.
iii. Divide $E$ into $E_1, E_2, \ldots$, where $E_i = \{e \in E \mid w(e) \in [z^{i-1}, z^i)\}$.
     Let $E^{(1)} = E$ and $E_i^{(1)} = E_i$ for all $i$.
iv.  For $j = 1, 2, \ldots$, until the graph is exhausted,
        1. $(C_1, C_2, \ldots, C_p) = \texttt{Partition}((V^{(j)}, \uplus_{i \le j} E_i^{(j)}), z/4)$
        2. Add a BFS tree of each component to $T$.
        3. Define graph $(V^{(j+1)}, E^{(j+1)})$ by contracting all edges within the components and removing all self-loops (but maintaining parallel edges). Create $E_i^{(j+1)}$ from $E_i^{(j)}$ taking into account the contractions.
v.   Output the tree $T$.
```

Presented in Algorithm 5.1 is a restatement of the AKPW algorithm, except that here we will use our parallel low-diameter decomposition for the partition step. In words, iteration j of Algorithm 5.1 looks at a graph $(V^{(j)}, E^{(j)})$ which is a minor of the original graph (because components were contracted in previous iterations, and because it only considers the edges in the first j weight classes). It uses `Partition`$((V, \uplus_{j \le k} E_j), z/4)$ to decompose this graph into components such that the hop radius is at most $z/4$ and each weight class has only $1/y$ fraction of its edges crossing between components. (Parameters y, z are defined in the algorithm and are slightly different from the original settings in the AKPW algorithm.) It then shrinks each of the components into a single node (while adding a BFS tree on that component to T), and iterates on this graph. Adding these BFS trees maintains the invariant that the set of original nodes which have been contracted into a (super-)node in the current graph are connected in T; hence, when the algorithm stops, we have a spanning tree of the original graph—hopefully of low total stretch.

We begin the analysis of the total stretch and running time by proving two useful facts:

Fact 5.2 *The number of edges* $|E_i^{(j)}|$ *is at most* $|E_i|/y^{j-i}$.

Proof: If we could ensure that the number of weight classes in play at any time is at most τ, the number of edges in each class would fall by at least a factor of $\frac{c_1 \tau \log^3 n}{z/4} = 1/y$ by Theorem 4.1(3) and the definition of z, and this would prove the fact. Now, for the first τ iterations, the number of weight classes is at most τ just because we consider only the first j weight classes in iteration j. Now in iteration $\tau + 1$, the number of surviving edges of E_1 would fall to $|E_1|/y^\tau \le |E_1|/n^3 < 1$, and hence there would only be τ weight classes left. It is easy to see that this invariant can be maintained over the course of the algorithm. ■

Fact 5.3 *In iteration* j, *the radius of a component according to edge weights (in the expanded-out graph) is at most* z^{j+1}.

Proof: The proof is by induction on j. First, note that by Theorem 4.1(2), each of the clusters computed in any iteration j has edge-count radius at most $z/4$. Now the base case $j = 1$ follows by noting that each edge in E_1 has weight less than z, giving a radius of at most $z^2/4 < z^{j+1}$. Now assume inductively that the radius in iteration $j - 1$ is at most z^j. Now any path with $z/4$ edges from the center to some node in the contracted graph will pass through at most $z/4$ edges of weight at most z^j, and at most $z/4 + 1$ supernodes, each of which adds a distance of $2z^j$; hence, the new radius is at most $z^{j+1}/4 + (z/4 + 1)2z^j \leq z^{j+1}$ as long as $z \geq 8$. ■

Applying these facts, we bound the total stretch of an edge class.

Lemma 5.4 *For any $i \geq 1$, $\mathsf{str}_T(E_i) \leq 4y^2|E_i|(4c_1\tau \log^3 n)^{\tau+1}$.*

Proof: Let e be an edge in E_i contracted during iteration j. Since $e \in E_i$, we know $w(e) > z^{i-1}$. By Fact 5.3, the path connecting the two endpoints of e in F has distance at most $2z^{j+1}$. Thus, $\mathsf{str}_T(e) \leq 2z^{j+1}/z^{i-1} = 2z^{j-i+2}$. Fact 5.2 indicates that the number of such edges is at most $|E_i^{(j)}| \leq |E_i|/y^{j-i}$. We conclude that

$$\begin{aligned}\mathsf{str}_T(E_i) &\leq \sum_{j=i}^{i+\tau-1} 2z^{j-i+2}|E_i|/y^{j-i} \\ &\leq 4y^2|E_i|(4c_1\tau\log^3 n)^{\tau+1}\end{aligned}$$

■

Proof: [of Theorem 5.1] Summing across the edge classes gives the promised bound on stretch. Now there are $\lceil \log_z \Delta \rceil$ weight classes E_i's in all, and since each time the number of edges in a (non-empty) class drops by a factor of y, the algorithm has at most $O(\log \Delta + \tau)$ iterations. By Theorem 4.1 and standard techniques, each iteration does $O(m \log^2 n)$ work and has $O(z \log^2 n) = O(\log^{O(1)} n \cdot 2^{O(\sqrt{\log n \cdot \log\log n})})$ depth in expectation. ■

5.2 Low-Stretch Spanning Subgraphs

We now show how to alter the parallel low-stretch spanning tree construction from the preceding section to give a low-stretch spanning *subgraph* construction that has no dependence on the "spread," and moreover has only polylogarithmic stretch. This comes at the cost of obtaining a sparse subgraph with $n - 1 + O(m/\operatorname{polylog} n)$ edges instead of a tree, but suffices for our solver application. The two main ideas behind these improvements are the following: Firstly, the number of surviving edges in each weight class decreases by a logarithmic factor in each iteration; hence, we could throw in all surviving edges after they have been whittled down in a constant number of iterations—this removes the factor of $2^{O(\sqrt{\log n \cdot \log\log n})}$ from both the average stretch and the depth. Secondly, if Δ is large, we will identify certain weight-classes with $O(m/\operatorname{polylog} n)$ edges, which by setting them aside, will allow us to break up the chain of dependencies and obtain $O(\operatorname{polylog} n)$ depth; these edges will be thrown back into the final solution, adding $O(m/\operatorname{polylog} n)$ extra edges (which we can tolerate) without increasing the average stretch.

5.2.1 The First Improvement

Let us first show how to achieve polylogarithmic stretch with an ultra-sparse subgraph. Given parameters $\lambda \in \mathbb{Z}_{>0}$ and $\beta \geq c_2 \log^3 n$ (where $c_2 = 2 \cdot (4c_1(\lambda+1))^{\frac{1}{2}(\lambda-1)}$), we obtain the new algorithm $\texttt{SparseAKPW}(G, \lambda, \beta)$

by modifying Algorithm 5.1 as follows:

(1) use the altered parameters $y = \frac{1}{c_2}\beta/\log^3 n$ and $z = 4c_1 y(\lambda+1)\log^3 n$;

(2) in each iteration j, call `Partition` with at most $\lambda+1$ edge classes—keep the λ classes $E_j^{(j)}, E_{j-1}^{(j)}, \ldots, E_{j-\lambda+1}^{(j)}$, but then define a "generic bucket" $E_0^{(j)} := \cup_{j' \le j-\lambda} E_{j'}^{(j)}$ as the last part of the partition; and

(3) finally, output not just the tree T but the subgraph $\widehat{G} = T \cup (\cup_{i \ge 1} E_i^{(i+\lambda)})$.

Lemma 5.5 *Given a graph G, parameters $\lambda \in \mathbb{Z}_{>0}$ and $\beta \ge c_2 \log^3 n$ (where $c_2 = 2 \cdot (4c_1(\lambda+1))^{\frac{1}{2}(\lambda-1)}$) the algorithm* `SparseAKPW`$(G, \lambda, \beta)$ *outputs a subgraph of G with at most $n-1+m(c_2(\log^3 n/\beta))^\lambda$ edges and total stretch at most $m\beta^2 \log^{3\lambda+3} n$. Moreover, the expected work is $\widetilde{O}(m)$ and expected depth is $O((c_1\beta/c_2)\lambda \log^2 n(\log\Delta + \log n))$.*

Proof: The proof parallels that for Theorem 5.1. Fact 5.3 remains unchanged. The claim from Fact 5.2 now remains true only for $j \in \{i, \ldots, i+\lambda-1\}$; after that the edges in $E_i^{(j)}$ become part of $E_0^{(j)}$, and we only give a cumulative guarantee on the generic bucket. But this does hurt us: if $e \in E_i$ is contracted in iteration $j \le i+\lambda-1$ (i.e., it lies within a component formed in iteration j), then $\mathsf{str}_{\widehat{G}}(e) \le 2z^{j-i+2}$. And the edges of E_i that survive till iteration $j \ge i+\lambda$ have stretch 1 because they are eventually all added to $\widehat{G}$; hence we do not have to worry that they belong to the class $E_0^{(j)}$ for those iterations. Thus,

$$\mathsf{str}_{\widehat{G}}(E_i) \le \sum_{j=i}^{i+\lambda-1} 2z^{j-i+2} \cdot |E_i|/y^{j-i} \le 4y^2(\frac{z}{y})^{\lambda-1}|E_i|.$$

Summing across the edge classes gives $\mathsf{str}_{\widehat{G}}(E) \le 4y^2(\frac{z}{y})^{\lambda-1}m$, which simplifies to $O(m\beta^2\log^{3\lambda+3} n)$. Next, the number of edges in the output follows directly from the fact T can have at most $n-1$ edges, and the number of extra edges from each class is only a $1/y^\lambda$ fraction (i.e., $|E_i^{(i+\lambda)}| \le |E_i|/y^\lambda$ from Fact 5.2). Finally, the work remains the same; for each of the $(\log\Delta + \tau)$ distance scales the depth is still $O(z\log^2 n)$, but the new value of z causes this to become $O((c_1\beta/c_2)\lambda\log^2 n)$. ■

5.2.2 The Second Improvement

The depth of the `SparseAKPW` algorithm still depends on $\log\Delta$, and the reason is straightforward: the graph $G^{(j)}$ used in iteration j is built by taking $G^{(1)}$ and contracting edges in each iteration—hence, it depends on all previous iterations. However, the crucial observation is that if we had τ consecutive weight classes E_i's which are empty, we could break this chain of dependencies at this point. However, there may be no empty weight classes; but having weight classes with relatively few edges is enough, as we show next.

Fact 5.6 *Given a graph $G = (V, E)$ and a subset of edges $F \subseteq E$, let $G' = G \setminus F$ be a potentially disconnected graph. If $\widehat{G}'$ is a subgraph of G' with total stretch $\mathsf{str}_{\widehat{G}'}(E(G')) \le D$, then the total stretch of E on $\widehat{G} := \widehat{G}' \cup F$ is at most $|F| + D$.*

Consider a graph $G = (V, E, w)$ with edge weights $w(e) \ge 1$, and let $E_i(G) := \{e \in E(G) \mid w(e) \in [z^{i-1}, z^i)\}$ be the weight classes. Then, G is called (γ, τ)-*well-spaced* if there is a set of *special* weight classes

$\{E_i(G)\}_{i \in I}$ such that for each $i \in I$, (a) there are at most γ weight classes before the following special weight class $\min\{i' \in I \cup \{\infty\} \mid i' > i\}$, and (b) the τ weight classes $E_{i-1}(G), E_{i-2}(G), \ldots, E_{i-\tau}(G)$ preceding i are all empty.

Lemma 5.7 *Given any graph $G = (V, E)$, $\tau \in \mathbb{Z}_+$, and $\theta \leq 1$, there exists a graph $G' = (V, E')$ which is $(4\tau/\theta, \tau)$-well-spaced, and $|E' \setminus E| \leq \theta \cdot |E|$. Moreover, G' can be constructed in $O(m)$ work and $O(\log n)$ depth.*

Proof: Let $\delta = \frac{\log \Delta}{\log z}$; note that the edge classes for G are $E_1, \ldots, E_\delta$, some of which may be empty. Denote by E_J the union $\cup_{i \in J} E_i$. We construct G' as follows: Divide these edge classes into disjoint groups $J_1, J_2, \ldots \subseteq [\delta]$, where each group consists of $\lceil \tau/\theta \rceil$ consecutive classes. Within a group J_i, by an averaging argument, there must be a range $L_i \subseteq J_i$ of τ *consecutive* edge classes that contains at most a θ fraction of all the edges in this group, i.e., $|E_{L_i}| \leq \theta \cdot |E_{J_i}|$ and $|L_i| \geq \tau$. We form G' by removing these the edges in all these groups L_i's from G, i.e., $G' = (V, E \setminus (\cup_i E_{L_i}))$. This removes only a θ fraction of all the edges of the graph.

We claim G' is $(4\tau/\theta, \tau)$-well-spaced. Indeed, if we remove the group L_i, then we designate the smallest $j \in [\delta]$ such that $j > \max\{j' \in L_i\}$ as a special bucket (if such a j exists). Since we removed the edges in E_{L_i}, the second condition for being well-spaced follows. Moreover, the number of buckets between a special bucket and the following one is at most

$$2\lceil \tau/\theta \rceil - (\tau - 1) \leq 4\tau/\theta.$$

Finally, these computations can be done in $O(m)$ work and $O(\log n)$ depth using standard techniques [JáJ92, Lei92]. ■

Lemma 5.8 *Let $\tau = 3 \log n / \log y$. Given a graph G which is (γ, τ)-well-spaced,* `SparseAKPW` *can be computed on G with $\widetilde{O}(m)$ work and $O(\frac{c_1}{c_2}\gamma\lambda\beta \log^2 n)$ depth.*

Proof: Since G is (γ, τ)-well-spaced, each special bucket $i \in I$ must be preceded by τ empty buckets. Hence, in iteration i of `SparseAKPW`, any surviving edges belong to buckets $E_{i-\tau}$ or smaller. However, these edges have been reduced by a factor of y in each iteration and since $\tau > \log_y n^2$, all the edges have been contracted in previous iterations—i.e., $E_\ell^{(i)}$ for $\ell < i$ is empty.

Consider any special bucket i: we claim that we can construct the vertex set $V^{(i)}$ that `SparseAKPW` sees at the beginning of iteration i, without having to run the previous iterations. Indeed, we can just take the MST on the entire graph $G = G^{(1)}$, retain only the edges from buckets $E_{i-\tau}$ and lower, and contract the connected components of this forest to get $V^{(i)}$. And once we know this vertex set $V^{(i)}$, we can drop out the edges from E_i and higher buckets which have been contracted (these are now self-loops), and execute iterations $i, i+1, \ldots$ of `SparseAKPW` without waiting for the preceding iterations to finish. Moreover, given the MST, all this can be done in $O(m)$ work and $O(\log n)$ depth.

Finally, for each special bucket i in parallel, we start running `SparseAKPW` at iteration i. Since there are at most γ iterations until the next special bucket, the total depth is only $O(\gamma z \log^2 n) = O(\frac{c_1}{c_2}\gamma\lambda\beta \log^2 n)$. ■

Theorem 5.9 (Low-Stretch Subgraphs) *Given a weighted graph G, $\lambda \in \mathbb{Z}_{>0}$, and $\beta \geq c_2 \log^3 n$ (where $c_2 = 2 \cdot (4c_1(\lambda+1))^{\frac{1}{2}(\lambda-1)}$), there is an algorithm* `LSSubgraph`(G, β, λ) *that finds a subgraph $\widehat{G}$ such that*

1. $|E(\widehat{G})| \leq n - 1 + m\left(c_{LS} \frac{\log^3 n}{\beta}\right)^\lambda$

2. *The total stretch (of all $E(G)$ edges) in the subgraph $\widehat{G}$ is at most by $m\beta^2 \log^{3\lambda+3} n$,*

where c_{LS} $(= c_2 + 1)$ is a constant. Moreover, the procedure runs in $\widetilde{O}(m)$ work and $O(\lambda\beta^{\lambda+1}\log^{3-3\lambda} n)$ depth. If $\lambda = O(1)$ and $\beta = \mathrm{polylog}(n)$, the depth term simplifies to $O(\log^{O(1)} n)$.

Proof: Given a graph G, we set $\tau = 3\log n/\log y$ and $\theta = (\log^3 n/\beta)^\lambda$, and apply Lemma 5.7 to delete at most θm edges, and get a $(4\tau/\theta, \tau)$-well-spaced graph G'. Let $m' = |E'|$. On this graph, we run `SparseAKPW` to obtain a graph $\widehat{G}'$ with $n - 1 + m'(c_2(\log^3 n/\beta))^\lambda$ edges and total stretch at most $m'\beta^2\log^{3\lambda+3} n$; moreover, Lemma 5.8 shows this can be computed with $\widetilde{O}(m)$ work and the depth is

$$O\left(\frac{c_1}{c_2}(4\tau/\theta)\lambda\beta\log^2 n\right) = O(\lambda\beta^{\lambda+1}\log^{3-3\lambda} n).$$

Finally, we output the graph $\widehat{G} = \widehat{G}' \cup (E(G) \setminus E(G'))$; this gives the desired bounds on stretch and the number of edges as implied by Fact 5.6 and Lemma 5.5. ■

6 Parallel SDD Solver

In this section, we derive a parallel solver for symmetric diagonally dominant (SDD) linear systems, using the ingredients developed in the previous sections. The solver follows closely the line of work of [ST03, ST06, KM07, KMP10]. Specifically, we will derive a proof for the main theorem (Theorem 1.1), the statement of which is reproduced below.

Theorem 1.1. For any fixed $\theta > 0$ and any $\varepsilon > 0$, there is an algorithm `SDDSolve` that on input an SDD matrix A and a vector b computes a vector $\tilde{x}$ such that $\|\tilde{x} - A^+b\|_A \leq \varepsilon \cdot \|A^+b\|_A$ in $O(m\log^{O(1)} n \log\frac{1}{\varepsilon})$ work and $O(m^{1/3+\theta}\log\frac{1}{\varepsilon})$ depth.

In proving this theorem, we will focus on Laplacian linear systems. As noted earlier, linear systems on SDD matrices are reducible to systems on graph Laplacians in $O(\log(m+n))$ depth and $O(m+n)$ work [Gre96]. Furthermore, because of the one-to-one correspondence between graphs and their Laplacians, we will use the two terms interchangeably.

The core of the near-linear time Laplacian solvers in [ST03, ST06, KMP10] is a "preconditioning" chain of progressively smaller graphs $\langle A_1 = A, A_2, \ldots, A_d\rangle$, along with a well-understood recursive algorithm, known as recursive preconditioned Chebyshev method—`rPCh`, that traverses the levels of the chain and for each visit at level $i < d$, performs $O(1)$ matrix-vector multiplications with A_i and other simple vector-vector operations. Each time the algorithm reaches level d, it solves a linear system on A_d using a direct method. Except for solving the bottom-level systems, all these operations can be accomplished in linear work and $O(\log(m+n))$ depth. The recursion itself is based on a simple scheme; for each visit at level i the algorithm makes at most κ'_i recursive calls to level $i+1$, where $\kappa'_i \geq 2$ is a fixed system-independent integer. Therefore, assuming we have computed a chain of preconditioners, the total required depth is (up to a log) equal to the total number of times the algorithm reaches the last (and smallest) level A_d.

6.1 Parallel Construction of Solver Chain

The construction of the preconditioning chain in [KMP10] relies on a subroutine that on input a graph A_i, constructs a slightly sparser graph B_i which is spectrally related to A_i. This "incremental sparsification" routine is in turn based on the computation of a low-stretch tree for A_i. The parallelization of the low-stretch tree is actually the main obstacle in parallelizing the whole solver presented in [KMP10]. Crucial to effectively applying our result in Section 5 is a simple observation that the sparsification routine of [KMP10] only requires a low-stretch spanning subgraph rather than a tree.Then, with the exception of some parameters in its construction, the preconditioning chain remains essentially the same.

The following lemma is immediate from Section 6 of [KMP10].

Lemma 6.1 *Given a graph G and a subgraph $\widehat{G}$ of G such that the total stretch of all edges in G with respect to $\widehat{G}$ is $m \cdot S$, a parameter on condition number κ, and a success probability $1 - 1/\xi$, there is an algorithm that constructs a graph H such that*

1. $G \preceq H \preceq \kappa \cdot G$, *and*
2. $|E(H)| = |E(\widehat{G})| + (c_{IS} \cdot S \log n \log \xi)/\kappa$

in $O(\log^2 n)$ depth and $O(m \log^2 n)$ work, where c_{IS} is an absolute constant.

Although Lemma 6.1 was originally stated with $\widehat{G}$ being a spanning tree, the proof in fact works without changes for an arbitrary subgraph. For our purposes, ξ has to be at most $O(\log n)$ and that introduces an additional $O(\log \log n)$ term. For simplicity, in the rest of the section, we will consider this as an extra $\log n$ factor.

Lemma 6.2 *Given a weighted graph G, parameters λ and η such that $\eta \geq \lambda \geq 16$, we can construct in $O(\log^{2\eta\lambda} n)$ depth and $\widetilde{O}(m)$ work another graph H such that*

1. $G \preceq H \preceq \frac{1}{10} \cdot \log^{\eta\lambda} n \cdot G$
2. $|E(H)| \leq n - 1 + m \cdot c_{PC}/\log^{\eta\lambda - 2\eta - 4\lambda}(n)$,

where c_{PC} is an absolute constant.

Proof: Let $\widehat{G} = \texttt{LSSubgraph}(G, \lambda, \log^{\eta} n)$. Then, Theorem 5.9 shows that $|E(\widehat{G})|$ is at most

$$n - 1 + m\left(\frac{c_{LS} \cdot \log^3 n}{\beta}\right)^{\lambda} = n - 1 + m\left(\frac{c_{LS}}{\log^{\eta-3} n}\right)^{\lambda}$$

Furthermore, the total stretch of all edges in G with respect to $\widehat{G}$ is at most

$$S = m\beta^2 \log^{\lambda+3} n \leq m \log^{2\eta+3\lambda+3} n.$$

Applying Lemma 6.1 with $\kappa = \frac{1}{10}\log^{\eta\lambda} n$ gives H such that $G \preceq H \preceq \frac{1}{10}\log^{\eta\lambda} n \cdot G$ and $|E(H)|$ is at most

$$\begin{aligned} & n - 1 + m \cdot \left(\frac{c_{LS}^{\lambda}}{\log^{\lambda(\eta-3)} n} + \frac{10 \cdot c_{IS} \log^{2\eta+3\lambda+5} n}{\log^{\eta\lambda} n}\right) \\ \leq \quad & n - 1 + m \cdot \frac{c_{PC}}{\log^{\eta\lambda - 2\lambda - 3k - 5} n} \\ \leq \quad & n - 1 + m \cdot \frac{c_{PC}}{\log^{\eta\lambda - 2\eta - 4\lambda} n}. \end{aligned}$$

∎

We now give a more precise definition of the preconditioning chain we use for the parallel solver by giving the pseudocode for constructing it.

Definition 6.3 (Preconditioning Chain) *Consider a chain of graphs*

$$\mathcal{C} = \langle A_1 = A, B_1, A_2, \dots, A_d \rangle,$$

and denote by n_i and m_i the number of nodes and edges of A_i respectively. We say that $\mathcal{C}$ is preconditioning chain *for A if*

1. $B_i =$ `IncrementalSparsify`(A_i).
2. $A_{i+1} =$ `GreedyElimination`(B_i).
3. $A_i \preceq B_i \preceq 1/10 \cdot \kappa_i A_i$*, for some explicitly known integer* κ_i*.* [7]

As noted above, the `rPCh` algorithm relies on finding the solution of linear systems on A_d, the bottom-level systems. To parallelize these solves, we make use of the following fact which can be found in Sections 3.4. and 4.2 of [GVL96].

Fact 6.4 *A factorization $LL^\top$ of the pseudo-inverse of an n-by-n Laplacian A, where L is a lower triangular matrix, can be computed in $O(n)$ time and $O(n^3)$ work, and any solves thereafter can be done in $O(\log n)$ time and $O(n^2)$ work.*

Note that although A is not positive definite, its null space is the space spanned by the all 1s vector when the underlying graph is connected. Therefore, we can in turn drop the first row and column to obtain a semi-definite matrix on which LU factorization is numerically stable.

The routine `GreedyElimination` is a partial Cholesky factorization (for details see [ST06] or [KMP10]) on vertices of degree at most 2. From a graph-theoretic point of view, the routine `GreedyElimination` can be viewed as simply recursively removing nodes of degree one and splicing out nodes of degree two. The sequential version of `GreedyElimination` returns a graph with no degree 1 or 2 nodes. The parallel version that we present below leaves some degree-2 nodes in the graph, but their number will be small enough to not affect the complexity.

Lemma 6.5 *If G has n vertices and $n - 1 + m$ edges, then the procedure* `GreedyElimination`(G) *returns a graph with at most $2m - 2$ nodes in $O(n + m)$ work and $O(\log n)$ depth* ***whp.***

Proof: The sequential version of `GreedyElimination`(G) is equivalent to repeatedly removing degree 1 vertices and splicing out 2 vertices until no more exist while maintaining self-loops and multiple edges (see, e.g., [ST03, ST06] and [Kou07, Section 2.3.4]). Thus, the problem is a slight generalization of parallel tree contraction [MR89]. In the parallel version, we show that while the graph has more than $2m - 2$ nodes, we can efficiently find and eliminate a "large" independent set of degree two nodes, in addition to all degree one vertices.

We alternate between two steps, which are equivalent to `Rake` and `Compress` in [MR89], until the vertex count is at most $2m - 2$:
Mark an independent set of degree 2 vertices, then

[7]The constant of $1/10$ in the condition number is introduced only to simplify subsequent notation.

1. Contract all degree 1 vertices, and
2. Compress and/or contract out the marked vertices.

To find the independent set, we use a randomized marking algorithm on the degree two vertices (this is used in place of maximal independent set for work efficiency): Each degree two node flips a coin with probability $\frac{1}{3}$ of turning up heads; we mark a node if it is a heads and its neighbors either did not flip a coin or flipped a tail.

We show that the two steps above will remove a constant fraction of "extra" vertices. Let G is a multigraph with n vertices and $m+n-1$ edges. First, observe that if all vertices have degree at least three then $n \leq 2(m-1)$ and we would be finished. So, let T be any fixed spanning tree of G; let a_1 (resp. a_2) be the number of vertices in T of degree one (resp. two) and a_3 the number those of degree three or more. Similarly, let b_1, b_2, and b_3 be the number vertices in G of degree 1, 2, and at least 3, respectively, where the degree is the vertex's degree in G.

It is easy to check that in expectation, these two steps remove $b_1+\frac{4}{27}b_2 \geq b_1+\frac{1}{7}b_2$ vertices. In the following, we will show that $b_1 + \frac{1}{7}b_2 \geq \frac{1}{7}\Delta n$, where $\Delta n = n - (2m-2) = n - 2m + 2$ denotes the number of "extra" vertices in the graph. Consider non-tree edges and how they are attached to the tree T. Let m_1, m_2, and m_3 be the number of attachment of the following types, respectively:

(1) an attachment to x, a degree 1 vertex in T, where x has at least one other attachment.
(2) an attachment to x, a degree 1 vertex in T, where x has no other attachment.
(3) an attachment to a degree 2 vertex in T.

As each edge is incident on two endpoints, we have $m_1 + m_2 + m_3 \leq 2m$. Also, we can lower bound b_1 and b_2 in terms of m_i's and a_i's: we have $b_1 \geq a_1 - m_1/2 - m_2$ and $b_2 \geq m_2 + a_2 - m_3$. This gives

$$\begin{aligned} b_1 + \tfrac{1}{7}b_2 &\geq \tfrac{2}{7}(a_1 - m_1/2 - m_2) + \tfrac{1}{7}(m_2 + a_2 - m_3) \\ &= \tfrac{2}{7}a_1 + \tfrac{1}{7}a_2 - \tfrac{1}{7}(m_1 + m_2 + m_3) \\ &\geq \tfrac{2}{7}a_1 + \tfrac{1}{7}a_2 - \tfrac{2}{7}m. \end{aligned}$$

Consequently, $b_1 + \frac{1}{7}b_2 \geq \frac{1}{7}(2a_1 + a_2 - 2m) \geq \frac{1}{7} \cdot \Delta n$, where to show the last step, it suffices to show that $n + 2 \leq 2a_1 + a_2$ for a tree T of n nodes. WLOG, we may assume that all nodes of T have degree either one or three, in which case $2a_1 = n + 2$. Finally, by Chernoff bounds, the algorithm will finish with high probability in $O(\log n)$ rounds. ■

6.2 Parallel Performance of Solver Chain

Spielman and Teng [ST06, Section 5] gave a (sequential) time bound for solving a linear SDD system given a preconditioner chain. The following lemma extends their Theorem 5.5 to give parallel runtime bounds (work and depth), as a function of κ_i's and m_i's. We note that in the bounds below, the m_d^2 term arises from the dense inverse used to solve the linear system in the bottom level.

Lemma 6.6 *There is an algorithm that given a preconditioner chain $\mathcal{C} = \langle A_1 = A, A_2, \ldots, A_d\rangle$ for a matrix A, a vector b, and an error tolerance ε, computes a vector $\tilde{x}$ such that*

$$\|\tilde{x} - A^{+}b\|_A \;\leq\; \varepsilon \cdot \|A^{+}b\|_A,$$

with depth bounded by

$$\left(\sum_{1\le i\le d}\prod_{1\le j<i}\sqrt{\kappa_j}\right)\log n\log\left(\tfrac{1}{\varepsilon}\right) \;\le\; O\left(\left(\prod_{1\le j<d}\sqrt{\kappa_j}\right)\log n\log\left(\tfrac{1}{\varepsilon}\right)\right)$$

and work bounded by

$$\left(\sum_{1\le i\le d-1} m_i\cdot\prod_{j\le i}\sqrt{\kappa_j}+m_d^2\prod_{1\le j<d}\sqrt{\kappa_j}\right)\log\left(\tfrac{1}{\varepsilon}\right).$$

To reason about Lemma 6.6, we will rely on the following lemma about preconditioned Chebyshev iteration and the recursive solves that happen at each level of the chain. This lemma is a restatement of Spielman and Teng's Lemma 5.3 (slightly modified so that the $\sqrt{\kappa_i}$ does not involve a constant, which shows up instead as constant in the preconditioner chain's definition).

Lemma 6.7 *Given a preconditioner chain of length d, it is possible to construct linear operators solve_{A_i} for all $i\le d$ such that*

$$(1-e^{-2})A_i^{+}\preceq solve_{A_i}\preceq(1+e^{2})$$

and solve_{A_i} is a polynomial of degree $\sqrt{\kappa_i}$ involving $\mathsf{solve}_{A_{i+1}}$ and 4 matrices with m_i non-zero entries (from `GreedyElimination`*).*

Armed with this, we state and prove the following lemma:

Lemma 6.8 *For $\ell\ge 1$, given any vector b, the vector $\mathsf{solve}_{A_\ell}\cdot b$ can be computed in depth*

$$\log n\sum_{\ell\le i\le d}\prod_{\ell\le j<i}\sqrt{\kappa_j}$$

and work

$$\sum_{\ell\le i\le d-1} m_i\cdot\prod_{\ell\le j\le i}\sqrt{\kappa_j}+m_d^2\prod_{\ell\le j<d}\sqrt{\kappa_j}$$

Proof: The proof is by induction in decreasing order on ℓ. When $d=\ell$, all we are doing is a matrix multiplication with a dense inverse. This takes $O(\log n)$ depth and $O(m_d^2)$ work.

Suppose the result is true for $\ell+1$. Then since solve_{A_ℓ} can be expressed as a polynomial of degree $\sqrt{\kappa_\ell}$ involving an operator that is $\mathsf{solve}_{A_{\ell+1}}$ multiplied by at most 4 matrices with $O(m_\ell)$ non-zero entries. We have that the total depth is

$$\begin{aligned}&\log n\sqrt{\kappa_\ell}+\sqrt{\kappa_\ell}\cdot\left(\log n\sum_{\ell+1\le i\le d}\prod_{\ell+1\le j<i}\sqrt{\kappa_j}\right)\\ =\;&\log n\sum_{\ell\le i\le d}\prod_{\ell\le j<i}\sqrt{\kappa_j}\end{aligned}$$

and the total work is bounded by

$$\begin{aligned}&\sqrt{\kappa_\ell}m_\ell\;+\;\sqrt{\kappa_\ell}\cdot\left(\sum_{\ell+1\le i\le d-1} m_i\cdot\prod_{\ell+1\le j\le i}\sqrt{\kappa_j}+m_d^2\prod_{\ell+1\le j<d}\sqrt{\kappa_j}\right)\\ =\;&\sum_{\ell\le i\le d-1} m_i\cdot\prod_{\ell\le j\le i}\sqrt{\kappa_j}+m_d^2\prod_{\ell\le j<d}\sqrt{\kappa_j}.\end{aligned}$$

■

Proof: [of Lemma 6.6] The ε-accuracy bound follows from applying preconditioned Chebyshev to solve_{A_1} similarly to Spielman and Teng's Theorem 5.5 [ST06], and the running time bounds follow from Lemma 6.8 when $\ell = 1$. ■

6.3 Optimizing the Chain for Depth

Lemma 6.6 shows that the algorithm's performance is determined by the settings of κ_i's and m_i's; however, as we will be using Lemma 6.2, the number of edges m_i is essentially dictated by our choice of κ_i. We now show that if we terminate chain earlier, i.e. adjusting the dimension A_d to roughly $O(m^{1/3}\log\varepsilon^{-1})$, we can obtain good parallel performance. As a first attempt, we will set κ_i's uniformly:

Lemma 6.9 *For any fixed $\theta > 0$, if we construct a preconditioner chain using Lemma 6.2 setting λ to some proper constant greater than 21, $\eta = \lambda$ and extending the sequence until $m_d \le m^{1/3-\delta}$ for some δ depending on λ, we get a solver algorithm that runs in $O(m^{1/3+\theta}\log(1/\varepsilon))$ depth and $\widetilde{O}(m\log 1/\varepsilon)$ work as $\lambda \to \infty$, where ε is the accuracy precision of the solution, as defined in the statement of Theorem 1.1.*

Proof: By Lemma 6.1, we have that m_{i+1}—the number of edges in level $i+1$—is bounded by

$$O(m_i \cdot \frac{c_{PC}}{\log^{\eta\lambda-2\eta-4\lambda}}) = O(m_i \cdot \frac{c_{PC}}{\log^{\lambda(\lambda-6)}}),$$

which can be repeatedly apply to give

$$m_i \le m \cdot \left(\frac{c_{PC}}{\log^{\lambda(\lambda-6)} n}\right)^{i-1}$$

Therefore, when $\lambda > 12$, we have that for each $i < d$,

$$\begin{aligned} m_i \cdot \prod_{j\le i}\sqrt{\kappa(n_j)} &\le m \cdot \left(\frac{c_{PC}}{\log^{\lambda(\lambda-6)} n}\right)^{i-1} \cdot \left(\sqrt{\log^{\lambda^2} n}\right)^{i} \\ &= \tilde{O}(m) \cdot \left(\frac{c_{PC}}{\log^{\lambda(\lambda-12)/2} n}\right)^{i} \\ &\le \tilde{O}(m) \end{aligned}$$

Now consider the term involving m_d. We have that d is bounded by

$$\left(\frac{2}{3}+\delta\right)\log m / \log\left(\frac{1}{c_{PC}}\log n^{\lambda(\lambda-6)}\right).$$

Combining with the $\kappa_i = \log^{\lambda^2} n$, we get

$$\begin{aligned}
&\prod_{1\le j\le d} \sqrt{\kappa(n_j)} \\
&\quad = \left(\log n^{\lambda^2/2}\right)^{(\frac{2}{3}+\delta)\log m/\log\left(c\log n^{\lambda(\lambda-6)}\right)} \\
&\quad = \exp\left(\log\log n \frac{\lambda^2}{2}(\frac{2}{3}+\delta)\frac{\log m}{\lambda(\lambda-6)\log\log n - \log c_{PC}}\right) \\
&\quad \le \exp\left(\log\log n \frac{\lambda^2}{2}(\frac{2}{3}+\delta)\frac{\log m}{\lambda(\lambda-7)\log\log n}\right) \\
&\qquad\quad (\text{since } \log c_{PC} \ge -\log n) \\
&\quad = \exp\left(\log n \frac{\lambda}{\lambda-7}(\frac{1}{3}+\frac{\delta}{2})\right) \\
&\quad = O(m^{(\frac{1}{3}+\frac{\delta}{2})\frac{\lambda}{\lambda-7}})
\end{aligned}$$

Since $m_d = O(m^{\frac{1}{3}-\delta})$, the total work is bounded by

$$O(m^{(\frac{1}{3}+\frac{\delta}{2})\frac{\lambda}{\lambda-7}+\frac{2}{3}-2\delta}) = O(m^{1+\frac{7}{\lambda-7}-\delta\frac{\lambda-14}{\lambda-7}})$$

So, setting $\delta \ge \frac{7}{\lambda-14}$ suffices to bound the total work by $\widetilde{O}(m)$. And, when δ is set to $\frac{7}{\lambda-14}$, the total parallel running time is bounded by the number of times the last layer is called

$$\begin{aligned}
\prod_j \sqrt{\kappa(n_j)} &\le O(m^{(\frac{1}{3}+\frac{1}{2(\lambda-14)})\frac{\lambda}{\lambda-7}}) \\
&\le O(m^{\frac{1}{3}+\frac{7}{\lambda-14}+\frac{\lambda}{2(\lambda-14)(\lambda-7)}}) \\
&\le O(m^{\frac{1}{3}+\frac{7}{\lambda-14}+\frac{7}{\lambda-14}}) \\
&\le O(m^{\frac{1}{3}+\frac{14}{\lambda-14}}) \quad \text{when } \lambda \ge 21
\end{aligned}$$

Setting λ arbitrarily large suffices to give $O(m^{1/3+\theta})$ depth. ■

To match the promised bounds in Theorem 1.1, we improve the performance by reducing the exponent on the $\log n$ term in the total work from λ^2 to some large fixed constant while letting total depth still approach $O(m^{1/3+\theta})$.

Proof: [of Theorem 1.1] Consider setting $\lambda = 13$ and $\eta \ge \lambda$. Then,

$$\eta\lambda - 2\eta - 4\lambda \ge \eta(\lambda-6) \ge \frac{7}{13}\eta\lambda$$

We use c_4 to denote this constant of $\frac{7}{13}$, namely c_4 satisfies

$$c_{PC}/\log^{\eta k-2\eta-4\lambda} n \le c_{PC}/\log^{c_4\eta\lambda} n$$

We can then pick a constant threshold L and set κ_i for all $i \le L$ as follows:

$$\kappa_1 = \log^{\lambda^2} n, \kappa_2 = \log^{(2c_4)\lambda^2} n, \cdots, \kappa_i = \log^{(2c_4)^{i-1}\lambda^2} n$$

To solve A_L, we apply Lemma 6.9, which is analogous to setting $A_L, \dots, A_d$ uniformly. The depth required in constructing these preconditioners is $O(m_d + \sum_{j=1}^{L}(2c_4)^{j-1}\lambda^2)$, plus $O(m_d)$ for computing the inverse at the last level—for a total of $O(m_d) = O(m^{1/3})$.

As for work, the total work is bounded by

$$\begin{aligned}
&\sum_{i\leq d} m_i \prod_{1\leq j\leq i}\sqrt{\kappa_j} + \prod_{1\leq j\leq d}\sqrt{\kappa_j} m_d^2 \\
&\quad= \sum_{i<L} m_i \prod_{1\leq j\leq i}\sqrt{\kappa_j} \\
&\qquad + \left(\prod_{1\leq j<L}\sqrt{\kappa_j}\right)\cdot\left(\sqrt{\kappa_j}\sum_{i\geq L} m_i \prod_{L\leq j\leq i}\sqrt{\kappa_j} + m_d^2 \prod_{L\leq j\leq d}\sqrt{\kappa_j}\right) \\
&\quad\leq \sum_{i<L} m_i \prod_{1\leq j\leq i}\sqrt{\kappa_j} + \left(\prod_{1\leq j<L}\sqrt{\kappa_j}\right) m_L\sqrt{\kappa_L} \\
&\quad= \sum_{i\leq L} m_i \prod_{1\leq j\leq i}\sqrt{\kappa_j} \\
&\quad\leq \sum_{i\leq L} \frac{m}{\prod_{j<i}\kappa_i^{c_4}} \prod_{1\leq j\leq i}\sqrt{\kappa_j} \\
&\quad= m\sum_{i\leq L} \frac{\sqrt{\kappa_1}\prod_{2\leq j\leq i}\sqrt{\kappa_{j-1}^{2c_4}}}{\prod_{j<i}\kappa_i^{c_4}} \\
&\quad= mL\sqrt{\kappa_1}
\end{aligned}$$

The first inequality follows from the fact that the exponent of $\log^n$ in κ_L can be arbitrarily large, and then applying Lemma 6.9 to the solves after level L. The fact that $m_{i+1} \leq m_i \cdot O(1/\kappa_i^{c_4})$ follows from Lemma 6.2.

Since L is a constant, $\prod_{1\leq j\leq L} \in O(\operatorname{polylog} n)$, so the total depth is still bounded by $O(m^{1/3+\theta})$ by Lemma 6.9. ■

7 Conclusion

We presented a near linear-work parallel algorithm for constructing graph decompositions with strong-diameter guarantees and parallel algorithms for constructing $2^{O(\sqrt{\log n \log\log n})}$-stretch spanning trees and $O(\log^{O(1)} n)$-stretch ultra-sparse subgraphs. The ultra-sparse subgraphs were shown to be useful in the design of a near linear-work parallel SDD solver. By plugging our result into previous frameworks, we obtained improved parallel algorithms for several problems on graphs.

We leave open the design of a (near) linear-work parallel algorithm for the construction of a low-stretch tree with polylogarithmic stretch. We also feel that the design of (near) work-efficient $O(\log^{O(1)} n)$-depth SDD solver is a very interesting problem that will probably require the development of new techniques.

Acknowledgments

This work is partially supported by the National Science Foundation under grant numbers CCF-1018463, CCF-1018188, and CCF-1016799, by an Alfred P. Sloan Fellowship, and by generous gifts from IBM, Intel, and Microsoft.

References

[ABN08] Ittai Abraham, Yair Bartal, and Ofer Neiman. Nearly tight low stretch spanning trees. In *FOCS*, pages 781–790, 2008. 5

[AKPW95] Noga Alon, Richard M. Karp, David Peleg, and Douglas West. A graph-theoretic game and its application to the k-server problem. *SIAM J. Comput.*, 24(1):78–100, 1995. 4, 5, 9

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *J. Assoc. Comput. Mach.*, 32(4):804–823, 1985. 4

[BV04] S. Boyd and L. Vandenberghe. *Convex Optimization*. Camebridge University Press, 2004. 2

[Chv79] V. Chvátal. The tail of the hypergeometric distribution. *Discrete Mathematics*, 25(3):285–287, 1979. 3, 4

[CKM$^+$10] Paul Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. 2010. 2

[Coh93] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. In *Proceedings of the 1993 IEEE 34th Annual Foundations of Computer Science*, pages 648–658, Washington, DC, USA, 1993. IEEE Computer Society. 5

[Coh00] Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *J. ACM*, 47(1):132–166, 2000. 3

[DS08] Samuel I. Daitch and Daniel A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. *CoRR*, abs/0803.0988, 2008. 2

[EEST05] Michael Elkin, Yuval Emek, Daniel A. Spielman, and Shang-Hua Teng. Lower-stretch spanning trees. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 494–503, New York, NY, USA, 2005. ACM Press. 5

[Gre96] Keith Gremban. *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, October 1996. CMU CS Tech Report CMU-CS-96-123. 3, 14

[GVL96] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins Press, 3rd edition, 1996. 16

[Hoe63] Wassily Hoeffding. Probability Inequalities for Sums of Bounded Random Variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963. 3, 4

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. 13

[KM07] Ioannis Koutis and Gary L. Miller. A linear work, $O(n^{1/6})$ time, parallel algorithm for solving planar laplacians. In *SODA*, pages 1002–1011, 2007. 1, 14

[KMP10] Ioannis Koutis, Gary L. Miller, and Richard Peng. Approaching optimality for solving SDD linear systems. In *FOCS*, pages 235–244, 2010. 4, 14, 15, 16

[KMP11] Ioannis Koutis, Gary L. Miller, and Richard Peng. A nearly $m \log n$ time solver for SDD linear systems. In *FOCS*, page (to appear), 2011. 1

[Kou07] Ioannis Koutis. *Combinatorial and algebraic algorithms for optimal multilevel algorithms*. PhD thesis, Carnegie Mellon University, Pittsburgh, May 2007. CMU CS Tech Report CMU-CS-07-131. 16

[KS97] Philip N. Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algorithms*, 25(2):205–220, 1997. 3

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992. 13

[MR89] Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. In Silvio Micali, editor, *Randomness and Computation*, pages 47–72. JAI Press, Greenwich, Connecticut, 1989. Vol. 5. 16

[Ren01] James Renegar. *A mathematical view of interior-point methods in convex optimization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001. 2

[Ska09] Matthew Skala. Hypergeometric tail inequalities: ending the insanity, 2009. 3, 4

[Spi10] Daniel A. Spielman. Algorithms, Graph Theory, and Linear Equations in Laplacian Matrices. In *Proceedings of the International Congress of Mathematicians*, 2010. 1

[SS08] Daniel A. Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. In *STOC*, pages 563–568, 2008. 2

[ST03] Daniel A. Spielman and Shang-Hua Teng. Solving sparse, symmetric, diagonally-dominant linear systems in time $O(m^{1.31})$. In *FOCS*, pages 416–427, 2003. 14, 16

[ST06] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *CoRR*, abs/cs/0607105, 2006. 4, 5, 14, 16, 17, 19

[Ten10] Shang-Hua Teng. The Laplacian Paradigm: Emerging Algorithms for Massive Graphs. In *Theory and Applications of Models of Computation*, pages 2–14, 2010. 1

[UY91] Jeffrey D. Ullman and Mihalis Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM J. Comput.*, 20(1):100–125, 1991. 3

[Ye97] Y. Ye. *Interior point algorithms: theory and analysis*. Wiley, 1997. 2