Theory Comput Syst (2014) 55:521–554
DOI 10.1007/s00224-013-9444-5

Nearly-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

Published online: 1 March 2013

Abstract We present the design and analysis of a nearly-linear work parallel algorithm for solving symmetric diagonally dominant (SDD) linear systems. On input an SDD n-by-n matrix A with m nonzero 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 $\theta > 0$, where A^{+} denotes the Moore-Penrose pseudoinverse of A.

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 $O(m \log^{O(1)} n)$ work and polylogarithmic depth, partitions a graph with n nodes and m edges 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 $O(m \log^{O(1)} n)$ work and $O(n^{\alpha})$ depth for any $\alpha > 0$. Alternatively, it can be used to

G.E. Blelloch · A. Gupta · G.L. Miller · R. Peng (✉)
Carnegie Mellon University, Pittsburgh, PA 15213, USA
e-mail: yangp@cs.cmu.edu

G.E. Blelloch
e-mail: guyb@cs.cmu.edu

A. Gupta
e-mail: anupamg@cs.cmu.edu

G.L. Miller
e-mail: glmiller@cs.cmu.edu

I. Koutis
University of Puerto Rico, Rio Piedras, Puerto Rico
e-mail: ioannis.koutis@upr.edu

K. Tangwongsan
1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA
e-mail: ktangwon@cs.cmu.edu

generate spanning subgraphs with polylogarithmic average stretch in $O(m\log^{O(1)} n)$ work and polylogarithmic depth. We apply this subgraph construction to derive a parallel linear 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.

Keywords Parallel algorithms · Linear systems · Low-stretch spanning trees · Low-stretch subgraphs · Low-diameter decomposition

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 $O(m\log^{O(1)} n\log(1/\varepsilon))$, where m is the number of nonzeros in the n-by-n-matrix A.[1] This is algorithmically significant since solving SDD systems has implications to computing eigenvectors, solving flow problems, finding graph sparsifiers, and problems in computer vision and graphics (see [27, 30] for these and other applications).

In the sequential setting, the current best SDD solvers run in

$$O\big(m\log n(\log\log n)^2\log(1/\varepsilon)\big)$$

time [20]. However, with the exception of the special case of planar SDD systems [18], we know of no previous parallel SDD solvers that perform nearly-linear[2] 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 nearly-linear work?* This work answers this question affirmatively:

Theorem 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 nonzero elements and a vector b, computes with high probability a vector $\tilde{x}$ such that $\|\tilde{x} - A^+b\|_A \le \varepsilon \cdot \|A^+b\|_A$ in $O(m\log^{O(1)} n\log\frac{1}{\varepsilon})$ work and $O(m^{1/3+\theta}\log\frac{1}{\varepsilon})$ depth where the exponent on $\log n$ is an absolute constant.*

In the process of developing this algorithm, we devise parallel algorithms for constructing graph decompositions with strong-diameter guarantees, and parallel algorithms to construct low-stretch spanning trees and low-stretch ultrasparse subgraphs.

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

[2]I.e. linear up to polylogarithmic factors.

These results may be of independent interest. An overview of these algorithms and their underlying techniques is given in Sect. 3.

The solver construction in Sect. 6 is based on an earlier sequential algorithm that runs in time $O(m \log^2 n(\log\log n)^4 \log(1/\varepsilon))$ [19]. Following their analysis, we only consider two adjacent levels of the preconditioning chain at a time. We believe that the more global analysis of Koutis et al. [20] can reduce the exponent of $\log n$ in the total work. But we currently do not attempt to optimize for polylogarithmic factors in the solver bounds. Instead, we focus on making the analysis simple while obtaining $O(m \log^{O(1)})$ work bound. At the present stage, the major bottleneck in our parallel algorithm is the polylogarithmic dependencies in the graph decomposition routines; this must be addressed before we have hopes to obtain a work bound comparable to that of the state-of-the-art sequential algorithms.

Some Applications Let us mention some of the implications of Theorem 1, obtained by plugging it into known reductions.

– *Construction of Spectral Sparsifiers*. Spielman and Srivastava [28] showed that spectral sparsifiers can be constructed using $O(\log n)$ Laplacian solves, and using our theorem we get spectral and cut sparsifiers in $O(m^{1/3+\theta})$ depth and $O(m \log^{O(1)} n)$ work.
– *Cut, Flow, and Shortest Path*. Daitsch and Spielman [9] showed that various graph optimization problems, such as max-flow, min-cost flow, and lossy flow problems, can be reduced to $O(m^{1/2} \log^{O(1)}(nU))$ applications of SDD solves via interior point methods described in [4, 24, 32], where U is the ratio of the largest edge weight to the smallest nonzero edge weight in the graph. Combining this with our main theorem implies that these algorithms can be parallelized to run in $O(m^{5/6+\theta})$ depth and $O(m^{3/2} \log^{O(1)}(nU))$ work. This gives the first parallel algorithm with $o(n)$ depth which is work-efficient to within polylog(n) factors of the sequential algorithm for all problems analyzed in [9]. In some sense, the parallel bounds are more interesting than the sequential times because in many cases the results in [9] 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 [9] 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 Christiano et al. [5], yielding a $O(m^{5/6+\theta}\text{poly}(\log n, \varepsilon^{-1}))$ depth and $O(m^{4/3}\text{poly}(\log n, \varepsilon^{-1}))$ work algorithm for finding $(1-\varepsilon)$-approximate maximum flows and $(1+\varepsilon)$-approximate minimum cuts in undirected graphs.

2 Preliminaries and Notation

We use $A \uplus B$ 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 $n = |V(G)|$ and $m = |E(G)|$. For an edge $e = \{u, v\}$, the stretch of e on H is

$$\mathsf{str}_H(e) = d_H(u, v)/w(e).$$

The *total stretch* of $G = (V, E, w)$ with respect to H is

$$\mathsf{str}_H\big(E(G)\big) = \sum_{e \in E(G)} \mathsf{str}_H(e),$$

and the *average stretch* is simply $\frac{1}{|E(G)|}\mathsf{str}_H(E(G))$.

Given $G = (V, E)$, a distance function δ (which is either *dist* or d), and a partition of V into $C_1 \uplus C_2 \uplus \cdots \uplus C_p$, 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.

Matrix Norms and SDD Matrices Let A be a symmetric matrix. As is standard in the literature, we say that A is *positive semidefinite* if for all vector x, $x^\top Ax \geq 0$. For symmetric matrices A and B, we write

$$A \preceq B \quad \text{if } B - A \text{ is positive semidefinite,}$$

or $x^\top Ax \geq x^\top Bx$ for all vector x. This defines a partial order on the symmetric positive semidefinite matrices.

Often, we would like to be able to talk about the inverse of a matrix even when it is not full rank. For a matrix A, we denote by A^+ the Moore-Penrose *pseudoinverse* of A. That is, A^+ has the same null space as A and acts as the inverse of A on its image. Given a symmetric positive semidefinite matrix A, the A-*norm* of a vector x is defined as

$$\|x\|_A = \sqrt{x^\top Ax}.$$

A matrix A is *symmetric diagonally dominant* (SDD) if it is symmetric and for all i, $A_{i,i} \geq \sum_{j \neq i} |A_{i,j}|$. We now discuss a class of SDD matrices closely related to undirected graphs.

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 specified by

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

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 $x \in \mathbb{R}^{|V|}$.

While the class of symmetric diagonally dominant matrices encompasses more than just the Laplacians, a solver for Laplacian systems implies a general SDD solver, as the following lemma indicates:

Lemma 2 [12, Lemma 7.3], [29, Appendix A] *Let A be an n-by-n symmetric diagonally dominant matrix with m nonzero entries. Let $\varepsilon \geq 0$. There is a $2n$-by-$2n$ Laplacian $\widehat{A}$ with at most $2m$ nonzeros such that*

$$\left\| \begin{pmatrix} u \\ w \end{pmatrix} - \widehat{A}^+ \widehat{b} \right\| \leq \varepsilon \left\| \widehat{A}^+ \widehat{b} \right\| \quad \textit{implies} \quad \left\| \frac{1}{2}(u-w) - A^+ b \right\| \leq \varepsilon \left\| A^+ b \right\|,$$

where $u, w \in \mathbb{R}^n$. Furthermore, the Laplacian $\widehat{A}$ can be constructed in $O(n+m)$ work and $O(\log^2(n+m))$ depth.

With this lemma, solving an SDD linear system reduces to solving a graph Laplacian system, a subclass of SDD matrices corresponding to undirected weighted graphs.

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., the parallel time in PRAM). Since we are not concerned with polylogarithmic factors in work or depth, the particular variant of the PRAM (CRCW, CREW or EREW) does not matter.

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, respectively, the total numbers of edges and nodes encountered in the search [16, 31]. 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 $r = O(\log^c n)$, 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., [8, 16, 31]). There is an alternative approach of repeatedly squaring a matrix, which yields 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 [6, 13, 26].

Lemma 3 (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*

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

Proof We apply the following theorem of Hoeffding [6, 13, 26]. 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

We design a parallel linear SDD solver algorithm in the general framework of Spielman and Teng [19, 29]. Our solver is modeled after the algorithm of Koutis et al. [19]. The sequential solver builds a preconditioning chain of progressively smaller graphs via a combination of partial Cholesky factorization and a sparsification step, which relies on an algorithm for generating low-stretch spanning trees (LSSTs). Unfortunately, existing LSST algorithms do not readily parallelize and depend on low-diameter decomposition algorithms, which do not readily parallelize either.

Our first challenge is to devise a *parallel* LSST algorithm that has nearly-linear work and low depth. Building on the algorithm of Alon et al. [2], we give an algorithm for generating low-stretch spanning trees with average stretch $2^{O(\sqrt{\log n \log\log n})}$ in $O(m \log^{O(1)} n)$ 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 algorithm relies on a parallel graph decomposition scheme of Awerbuch [3], which takes an unweighted graph and breaks it into components with a specified diameter and a few crossing edges. While such schemes are known in the sequential setting, they do not parallelize well because removing edges belonging to one component might increase the diameter or even disconnect subsequent components.

We devise a parallel algorithm for partitioning a graph into components with low strong diameter while cutting only a small number of the input edges. The key challenge here lies in meeting the strong diameter requirement (i.e., one *cannot* take "shortcuts" through other components). Sequentially, the strong-diameter property is trivially met by removing components one after another. But this process is inherently sequential. In the parallel case, we grow balls from multiple sites but (conceptually) delay the starting appropriately. We assign vertices to the first region that reaches them. By carefully controlling the delay amount and other parameters, we have the first nearly-linear work, polylogarithmic depth algorithm for low-diameter decomposition (Sect. 4).

Ideally, we would have liked for our spanning trees to have a polylogarithmic stretch, computable by a polylogarithmic depth, nearly-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 "ultrasparse" graph with small stretch, one that has only $O(m/\text{polylog}(n))$ edges more than a tree. Hence, we present a parallel algorithm in Sect. 5.2 which outputs an ultrasparse graph with $O(\text{polylog}(n))$ average stretch, performing $O(m \log^{O(1)} n)$ work with $O(\text{polylog}(n))$ 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 $O(\text{polylog}(n))$.[3]

With the low-stretch subgraph result, the algorithm as given in Koutis et al. [19] parallelizes under the following modifications: (i) perform the partial Cholesky factorization in parallel and (ii) terminate the preconditioning chain when the graph has about $m^{1/3}$ nodes. In Sect. 6, we describe how these components fit together.

4 Parallel Low-Diameter Decomposition

In this section, we describe a parallel algorithm for partitioning a graph into components with low (strong) diameter while cutting only a small number of the input edges. This routine is a fundamental building block for our algorithms for constructing low-stretch spanning trees and subgraphs in Sect. 5. We begin by developing a partitioning algorithm for a special case of the problem. Then, we will show how to bootstrap it to obtain an algorithm for the general case. The main result of this section is as follows:

Theorem 4 (Parallel Low-Diameter Decomposition) *Given an input graph $G = (V, E_1 \uplus \cdots \uplus E_k)$ with k edge classes and a "radius" parameter ρ, the algorithm* $\texttt{Partition}(G, \rho)$, *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 $\mathrm{dist}_{G[C_i]}(s_i, u) \le \rho$, 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.*

[3]As an aside, this construction of low-stretch ultrasparse graphs shows how to obtain the nearly-linear time sequential linear system solver from [29] without the more intricate, polylog average stretch constructions [1, 10].

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

This theorem provides a nearly-linear work algorithm that takes a graph with multiple edge classes and produces a partition of vertices into components such that each component has a small strong diameter and the number of edges that go across the components is small in every edge class. These properties are necessary in our low-stretch embedding algorithms in the next section, as we briefly discuss next.

Our algorithms are based on the low-stretch spanning tree algorithm of Alon, Karp, Peleg, and West (AKPW) [2]. In broad strokes, such an algorithm (more details in Sect. 5) buckets the input edges by weight into edge classes and in a series of iterations, partitions the graph into components and contracts them. For each contracted component, it adds to the resulting tree a spanning tree connecting the component's nodes. While this process always outputs a spanning tree, the quality of the output tree and the efficiency of the algorithm depend on two important parameters: the diameter bound of the components and the number of edges that go across components. Intuitively, in every iteration, each cross edge is further stretched by an amount roughly proportional to the diameter bound, until it becomes part of a component. To ensure that the output tree has low stretch, the algorithm therefore looks for small-diameter components with a small number of cross edges.

Parallelizing such an algorithm requires designing an algorithm that finds the components in parallel. The key challenge here lies in meeting the requirement that in the APKW construction, the spanning tree for a component *cannot* take "shortcuts" through other components; this is known as the strong-diameter property (see Sect. 2). In the sequential case, the strong-diameter property is trivially met by removing components one after another. This process, however, 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 [7].

4.1 Low-Diameter Decomposition for Simple Unweighted Graphs

As a first step, we consider a special case where the input is a simple graph with only *one* edge class. We design an algorithm `splitGraph` for this special case and in the section that follows, we describe how to build on top of it an algorithm that can handle multiple edge classes.

The algorithm `splitGraph` works as follows. It 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 $C_1, \dots, C_p$, each with center s_i, such that

(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 $dist_{G[C_i]}(s_i, u) \le \rho$;

(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 $\frac{600}{\rho}\log^3 n$.

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.)

Algorithm 4.1 shows the pseudocode of the algorithm. It 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 each 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.

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

```
// initialization
Let $G^{(1)} = (V^{(1)}, E^{(1)}) = G$. 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 $t = 1, 2, \dots, T = 2\log_2 n$,
  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 $X^{(t)} = \bigcup_{s\in S^{(t)}} B_s^{(t)}$.
  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 $\mathcal{C}$.
```

After choosing the starting centers, the algorithm grows a ball from each center $s \in S^{(t)}$ to radius $r^{(t)} - \delta_s^{(t)}$, where $\delta_s^{(t)} \in_R \{0, 1, \dots, R\}$ is a random "jitter" value and

$r^{(t)} = \frac{\rho}{2\log n}(T - t + 1)$. The union of these balls forms $X^{(t)}$, the set of nodes "seen" from these starting point. In this process, the "jitter" value is conceptually a random amount by which we delay the ball-growing process on each center, so that we can 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 a starting center to the center that minimizes $dist_{G^{(t)}}(u, s) + \delta_s^{(t)}$ (ties broken in a consistent manner, e.g., lexicographically). Some centers may not have any vertices assigned to them; we discard them, retaining only non-empty components. 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, forming an induced graph on the remaining vertices.

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

Fact 5 *If vertex u lies in component $C_s^{(t)}$, then $dist^{(t)}(s, u) \le r^{(t)}$. Moreover, $u \in B_s^{(t)}$.*

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

Lemma 6 *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 5 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 $j \ne s$, and hence $dist^{(t)}(v, j) + \delta_j^{(t)} \le dist^{(t)}(v, s) + \delta_s^{(t)}$. In this case, we have $dist^{(t)}(u, j) + \delta_j^{(t)} \le dist^{(t)}(u, v) + dist^{(t)}(v, j) + \delta_j^{(t)}$, by the triangle inequality. Now using the assumption, 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). If all inequalities hold with equality, we have $j < s$ for v to be assigned to $B_j^{(t)}$. This means u would be assigned to $C_j^{(t)}$, giving 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 5 and Lemma 6, the (strong) radius is at most TR, proving (P2). It now remains to prove (P3), and the work and depth bound.

In a series of claims below, we will show that the number of starting centers that can reach a particular node is small. This number tells us how likely an edge is going to be cut, allowing us to prove (P3).

Claim 7 *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 $s \in S^{(t)}$, $r^{(t)} - \delta_s \geq r^{(t)} - R = r^{(t+1)}$, and so if $s \in B^{(t)}(v, r^{(t+1)})$, then $v \in B_s^{(t)}$ and hence in $X^{(t)}$. Therefore,

$$\mathbf{Pr}\big[v \in X^{(t)}\big] \geq \mathbf{Pr}\big[S^{(t)} \cap B^{(t)}\big(v, r^{(t+1)}\big) \neq \emptyset\big],$$

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,

$$\mathbf{Pr}\big[S^{(t)} \cap B^{(t)}\big(v, r^{(t+1)}\big) \neq \emptyset\big] \geq 1 - \left(1 - \frac{|B^{(t)}(v, r^{(t+1)})|}{|V^{(t)}|}\right)^{\sigma_t},$$

which is at least $1 - n^{-12}$. □

Claim 8 *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 $1 - n^{-8}$.*

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

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

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 $S^{(t)}$,

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

which gives

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

To obtain a high probability bound for X, we will apply the tail bound in Lemma 3. 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, $\mathbf{Pr}[X \geq 34 \log n] \leq \exp\{-\frac{1}{4} \cdot 34 \log n\}$, 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 9 *Suppose that for every vertex $u \in V$ and for every $t \in [T]$, there are at most λ pairs (s, t) such that $s \in S^{(t)}$ and $u \in B^{(t)}(s, r^{(t)})$; that is, u is reached from at most λ starting centers in each iteration. Then, for any edge $uv \in E$, the probability that u belongs to a different component than v (i.e., uv is cut) is at most $4\lambda T/R$.*

Proof Fix an edge $uv \in E$. We will upper bound the probability that it is cut. There are two possibilities u and v are separated by the algorithm:

(1) There is an iteration where either u or v is reached (i.e., it belongs to $X^{(t)}$) but not the other; or
(2) There is an iteration where both of them are reached but they are put into different components.

We begin by bounding the probability of the former case. Assume without loss of generality that u is reached but v is not. Consider a center s in iteration t. For u to be reached from s in this iteration, it must be the case that $\delta_s^{(t)} = r^{(t)} - \mathit{dist}^{(t)}(s, u)$ because $\mathit{dist}^{(t)}(s, v) \le \mathit{dist}^{(t)}(s, u) + 1$. Now there are R possible choices of $\delta_s^{(t)}$, so this happens with probability at most $1/R$ for a particular pair (t, s). But across T iterations, there are at most λT different centers that can possibly cut the edge in this way. Taking a union bound over them gives us an upper bound of $\lambda T/R$.

We now turn to bounding the probability for the latter case. Consider an iteration t. For notational convenience, let the centers in this iteration (i.e., the elements of $S^{(t)}$) be $s_1, s_2, \ldots, s_k$; the indices are ordered arbitrarily. Let $\mathcal{E}_{uv}$ be the event that u belongs to a center s_i and v belongs to a center $s_{i'} \neq s_i$ in this iteration. In the rest of this proof, the distance function is the distance function in iteration t. Since $\mathit{dist}(s_i, v) \le \mathit{dist}(s_i, u) + 1$, we have $\mathit{dist}(s_{i'}, v) - \delta_{s_{i'}} \le \mathit{dist}(s_i, u) + 1 - \delta_{s_i}$, which, in turn, implies $\mathit{dist}^{(t)}(s_{i'}, u) - \delta_{s_{i'}} \le \mathit{dist}(s_i, u) + 2$. Thus, the event $\mathcal{E}_{uv}$ happens only if there exist $i \neq i'$ such that

$$\max(\mathit{dist}_i - \delta_{s_i}, \mathit{dist}_{i'} - \delta_{s_{i'}}) \le 2 + \min_j(\mathit{dist}_j - \delta_{s_j}),$$

where dist_i is a shorthand for $\mathit{dist}^{(t)}(s_i, u)$.

To proceed, we call a center s_i *critical* if $\mathit{dist}_i - \delta_{s_i} \le 2 + \min_{j \le i}(\mathit{dist}_j - \delta_{s_j})$. We will show that the event $\mathcal{E}_{uv}$ is unlikely by bounding the number of critical centers over settings of $\delta_{s_1}, \delta_{s_2}, \ldots, \delta_{s_k}$. For this, define χ_i to be the number of nodes s_j, $j \le i$ that are critical. We will show by induction that $\mathbf{E}[\chi_i] \le 1 + \frac{5(i-1)}{R}$.

The base case follows trivially. Suppose the hypothesis is true for $i - 1$. If the minimum so far is x—that is, $x = \min_{j \le i}(\mathit{dist}_j - \delta_{s_j})$—and there are t critical nodes so far—that is, $\chi_{i-1} = t$, then there are three cases to consider depending on the value of $\mathit{dist}_i - \delta_{s_i}$:

1. $|\mathit{dist}_i - \delta_{s_i} - x| \le 2$;
2. $\mathit{dist}_i - \delta_{s_i} < x - 2$; or
3. $\mathit{dist}_i - \delta_{s_i} > x + 2$

Since δ_i is chosen independently of $\delta_1, \ldots, \delta_{i-1}$, the first case happens with probability at most $5/R$. In this case, the number of critical nodes becomes at most $t + 1$.

In the remaining cases, the number of critical nodes does not increase and remains bounded by t. Therefore,

$$\mathbf{E}[\chi_i] \le \mathbf{E}[\chi_{i-1}] + \frac{5}{R} \le 1 + \frac{5(i-2)}{R} + \frac{5}{R} = 1 + \frac{5(i-1)}{R},$$

where the second inequality follows from the inductive hypothesis.

This calculation shows that in expectation, the number of critical nodes in this iteration is at most $1 + 5(k-1)/R$. The minimizer of $\min_j dist_j - \delta_{s_j}$ is critical by definition, so $\chi_k \ge 1$ and we can apply Markov's inequality to bound the probability that $\chi_k \ge 2$. Since $k \le \lambda$, this probability, and in turn the probability of event $\mathcal{E}_{uv}$ happens is at most $\frac{1}{2}(1 + \frac{5(\lambda-1)}{R}) \le 3\lambda/R$ as long as $R \ge 5$. Therefore, taking a union bound over T iterations, the probability that uv is separated in this way is at most $3\lambda T/R$.

Combining these two cases, we conclude that the probability that uv is separated by the algorithm is at most $4\lambda T/R$. □

We now reason about (P3). Notice that by Claim 8, the premise to Lemma 9, with $\lambda = 34 \log n$, holds with probability at least $1 - o(1) \ge 3/4$. Therefore, property (P3) follows directly from Lemma 9, where the technical condition is the premise to the lemma.

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 $r^{(t)} \le \rho$, 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, each vertex can be reached from at most $O(\log n)$ starting centers per iteration (Claim 8); therefore, across $T = O(\log n)$ iterations, we have 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 $E_1 \uplus \cdots \uplus E_k$. Now 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 $p = \frac{600}{\rho} \log^3 n$. This allows us to prove the following corollary, which follows directly from Markov's inequality and the union bounds.

Corollary 10 *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* $|E_i| \frac{800k \log^3 n}{\rho}$.

The corollary suggests a simple way to use `splitGraph` to provide guarantees required by Theorem 4: 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 $|E_i| \frac{800k \log^3 n}{\rho}$). Hence, the number of trials

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 $|E_i|\frac{800\cdot k\log^3 n}{\rho}$ edges between components,
   start over. (Recall that $k$ was the number of edge classes.)
Return $\mathcal{C}$.
```

is a geometric random variable with $p = 1/4$, so in expectation, it will finish after 4 trials.

Finally, we note that properties (P1) and (P2) directly give Theorem 4(1)–(2)—and the validation step in `Partition` ensures Theorem 4(3), setting $c_1 = 800$. The work and depth bounds for `Partition` follow from the bounds derived for `splitGraph` and Corollary 10. This concludes the proof of Theorem 4.

5 Parallel Low-Stretch Spanning Trees and Subgraphs

This section presents parallel algorithms for low-stretch spanning trees (LSSTs) and for low-stretch spanning subgraphs (LSSGs). These algorithms take as input a weighted graph $G = (V, E, w)$ and produce either a tree $T \subseteq G$ in the case of the LSST algorithm or a subgraph $H \subseteq G$ in the case of the LSSG algorithm. The subgraph produced has a special form: it comprises a tree T and a subset of "extra" edges $\bar{E}$ such that the edges in $E \setminus \bar{E}$ have small stretch on average.

In addition, the algorithms will return an upper bound of the stretch value of each edge in $E \setminus \bar{E}$ (in the case of LSST, $\bar{E} = \emptyset$). That is, the algorithms will return a vector $\widehat{\mathsf{str}} : E \setminus \bar{E} \to \mathbb{R}_+$, where $\widehat{\mathsf{str}}_e$ is an upper bound of the stretch of the edge e. In general, the exact values of stretches with respect to a tree T can be computed in parallel using tree contraction [22], but having these upper bounds allows us to simplify our solver presentation.

To this end, we first derive a parallel LSST algorithm by applying the construction of Alon et al. [2] (henceforth, the AKPW construction), together with the parallel graph partition algorithm from the previous section. This results in an algorithm with $O(m \log^{O(1)} n)$ work and $O(\text{polylog}(n) \cdot 2^{O(\sqrt{\log n \cdot \log\log n})} \cdot \log \Delta)$ depth, producing a spanning subtree with an average stretch $O(2^{O(\sqrt{\log n \cdot \log\log n})})$. The algorithm also outputs a vector $\widehat{\mathsf{str}}$ upper-bounding the stretch values although in this particular case, since the output is a spanning tree, the exact stretch for every edge can be computed using a parallel lowest common ancestor algorithm [25] in $O(m \log^{O(1)} n)$ work and $O(\text{polylog}(n))$ depth.

This LSST algorithm, however, is less than ideal for two reasons: the depth of the algorithm depends on the "spread" Δ term—the ratio between the heaviest edge and the lightest edge—and even for polynomial spread, both the depth and the average stretch are more than poly-logarithmic (both of them have a $2^{O(\sqrt{\log n \cdot \log\log n})}$ term). For our solver application, we observe that we do not need spanning trees but merely low-stretch sparse graphs which have $O(m/\text{polylog}(n))$ more edges than a tree. With this extra flexibility, we show in Sect. 5.2 that the average stretch can be reduced to

$O(\text{polylog}(n))$ and the sparse graph can be constructed in $O(m \log^{O(1)} n)$ work and $O(\text{polylog}(n))$ depth. We give an LSSG algorithm that achieves this by conceptually setting aside a small number of edges and constructing a low-stretch spanning tree on the remaining graph; then, the output simply combines the spanning tree and the edges that we set aside. Notice that every edge that was set aside has stretch 1 and the stretch of the remaining edges is upper bounded by the stretch of these edges going through the spanning tree T. To accomplish this, we modify our LSST algorithm to identify and set aside roughly $O(m/\text{polylog}(n))$ edges in such a way that on the remaining edges a tree with average stretch $O(\text{polylog}(n))$ can be built. We believe this construction may be of independent interest.

5.1 Low-Stretch Spanning Trees

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

Theorem 11 (Low-Stretch Spanning Tree) *There is an algorithm* AKPW(G) *which given as input a graph $G = (V, E, w)$, produces a spanning tree T and a vector $\widehat{\mathsf{str}}$ such that $\mathsf{str}_T(e) \le \widehat{\mathsf{str}}(e)$ for all $e \in E$ and*

$$\sum_{e \in E} \widehat{\mathsf{str}}_e \le O\big(m \cdot 2^{O(\sqrt{\log n \cdot \log\log n})}\big).$$

Furthermore, the algorithm has $O(\log^{O(1)} n \cdot 2^{O(\sqrt{\log n \cdot \log\log n})} \log \Delta)$ expected depth and $O(m \log^{O(1)} n)$ expected work.

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`$((V, \uplus_{j \le k} E_j), z/4)$

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

```
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. $(C_1, C_2, \ldots, C_p) =$ Partition$((V^{(j)}, \uplus_{i \le j} E_i^{(j)}), z/4)$
      2. Add a BFS tree of each component to $T$.
      3. For all $e \in E_i^{(j)}$ contained in a component, set $\widehat{\mathsf{str}}_e$ to $2z^{j-i+2}$.
      4. 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 $T$ and a vector upperbounding the stretch values $\widehat{\mathsf{str}}$.
```

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. For some intuition on the stretch bound, notice that in this construction, every time an edge participates in `Partition` but is not cut, it is further stretched by about a factor of z, so the stretch of an edge $e \in E_i$ is a function of how many times the edge participates in `Partition` before it is cut. The algorithm uses $2z^{j-i+2}$ as an upperbound, where j is the iteration when e is cut.

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

Fact 12 *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, then 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(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. By induction, the same holds true for subsequent iterations. □

Fact 13 *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(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 $z^{j+1}/4 + (z/4 + 1)2z^j \le z^{j+1}$ as long as $z \ge 8$. □

Combining these facts, we will show that the vector $\widehat{\mathsf{str}}$ gives an upperbound on the edge stretch:

Lemma 14 *For any edge e,* $\mathsf{str}_T(e) \le \widehat{\mathsf{str}}(e)$.

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 13, the path connecting the two endpoints of e in F has distance at most $2z^{j+1}$. Thus, $\mathsf{str}_T(e) \le 2z^{j+1}/z^{i-1} = 2z^{j-i+2}$. □

We can then bound the total stretch of an edge class.

Lemma 15 *For any $i \geq 1$,*

$$\sum_{e \in E_i} \widehat{\mathsf{str}}(e) \leq 4y^2 |E_i| \bigl(4c_1 \tau \log^3 n\bigr)^{\tau+1}.$$

Proof Fact 12 indicates that the number of edges is from E_i contracted during iteration j at most $|E_i^{(j)}| \leq |E_i|/y^{j-i}$.

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

completing the proof. □

Proof of Theorem 11 Summing the bound in Lemma 15 across the edge classes gives the promised bound on the total of all upper bounds. As for work/depth bounds, 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 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 modify the parallel low-stretch spanning tree construction from the preceding section to give a low-stretch spanning *subgraph* whose depth does not depend on the "spread," and moreover has only polylogarithmic stretch. This comes at the cost of obtaining a sparse subgraph with $n - 1 + O(m/\text{polylog} n)$ edges instead of a tree, but suffices for our solver application. The two main ideas behind these improvements are the following:

– First, 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.
– Second, if Δ is large, we will identify certain weight-classes with $O(\frac{m}{\text{polylog}} n)$ edges, which by setting them aside, will allow us to break up the chain of dependencies and obtain $O(\text{polylog} n)$ depth; these edges will be thrown back into the final solution, adding $O(m/\text{polylog} n)$ 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 by setting aside a small fraction of the edges. 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 SparseAKPW(G, λ, β) 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" $E_0^{(j)} := \bigcup_{j' \le j-\lambda} E_{j'}^{(j)}$ as the last part of the partition; and
(3) finally, output T along with all the edges that were put into the generic bucket, $\bar{E} = \bigcup_{i \ge 1} E_i^{(i+\lambda)} \setminus T$.

Conceptually, we can think of $\bar{E}$ as the edges that we set aside, so then T is the spanning subtree that we build for $G \setminus \bar{E}$. Therefore, with respect to the subgraph $T \cup \bar{E}$ that we generate, the edges $\bar{E}$ that we set aside have stretch 1 and the remaining edges have stretch at most the stretch routing through T.

We now state and prove the guarantees of our modified algorithm:

Lemma 16 *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, λ, β) *outputs a subset of edges $\bar{E} \subseteq E$, a tree T and for all edges in $E \setminus \bar{E}$, upper bounds for stretches with respect to T. $\bar{E}$ has size at most $m(c_2 \log^3 n/\beta)^\lambda$, and the sum of upper bounds is at most $O(m\beta^2 \log^{3\lambda+3} n)$. Moreover, the expected work is $O(m \log^{O(1)} n)$ and expected depth is $O((c_1\beta/c_2)\lambda \log^2 n(\log \Delta + \log n))$.*

Proof The proof parallels that of Theorem 11. Fact 13 remains unchanged. The claim from Fact 12 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.

Summing across the edge classes gives $\sum_{e \in E \setminus \bar{E}} \widehat{\mathsf{str}}(e) \le 4y^2(\frac{z}{y})^{\lambda-1}m$, which simplifies to $O(m\beta^2 \log^{3\lambda+3} n)$. Next, the number of edges in the output follows from the fact that 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 12). 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.

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 $E_{i-1}(G), E_{i-2}(G), \dots, E_{i-\tau}(G)$ preceding i are all empty.

Lemma 17 *Given any graph $G = (V, E)$, $\tau \in \mathbb{Z}_+$, and $\theta \le 1$, there exists a graph $G' = (V, E')$ which is $(4\tau/\theta, \tau)$-well-spaced, and $|E \setminus E'| \le \theta \cdot |E|$. 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 $E_1, \dots, E_\delta$, some of which may be empty. Denote by E_J the union $\bigcup_{i \in J} E_i$. We construct G' as follows: Divide these edge classes into disjoint groups $J_1, J_2, \dots \subseteq [\delta]$, 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., $|E_{L_i}| \le \theta \cdot |E_{J_i}|$ and $|L_i| \ge \tau$. We form G' by removing these the edges in all these groups L_i's from G, i.e., $G' = (V, E \setminus (\bigcup_i E_{L_i}))$. 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

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

Finally, these computations can be done in $O(m)$ work and $O(\log n)$ depth using standard techniques [14, 21]. □ □

Lemma 18 *Let $\tau = 3\log n/\log y$. Given a graph G which is (γ, τ)-well-spaced,* `SparseAKPW` *can be computed on G with $O(m \log^{O(1)} n)$ 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 $i, i+1, \dots$ 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 $O(\gamma z \log^2 n) = O(\frac{c_1}{c_2}\gamma\lambda\beta \log^2 n)$. □

We conclude this section with a theorem that summarizes our result for low-stretch subgraphs:

Theorem 19 (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 spanning tree T, a subset of edges $\bar{E} \subseteq E(G)$ along with upper bounds on stretch $\widehat{\mathsf{str}}$ for all edges $e \in E(G) \setminus \bar{E}$ such that*

1. $|\bar{E}| \leq O(m(c_{LS}\frac{\log^3 n}{\beta})^\lambda)$;
2. *For each edge $e \in E(G) \setminus \bar{E}$, $\mathsf{str}_T(e) \leq \widehat{\mathsf{str}}_e$; and*
3. $\sum_{e\in E(G)\setminus\bar{E}} \widehat{\mathsf{str}}(e) \leq O(m\beta^2 \log^{3\lambda+3} n)$,

where c_{LS} $(= c_2 + 1)$ is a constant. Moreover, the procedure runs in $O(m \log^{O(1)} n)$ 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 17 to delete at most θm edges to form G'. This leaves us with a $(4\tau/\theta, \tau)$-well-spaced graph G', and let $m' = |E(G')|$. On this graph, we run `SparseAKPW` to obtain a spanning tree T and a subset of edges $\bar{E}' \subseteq E(G')$ with $O(m'(c_2 \log^3 n/\beta)^\lambda)$ edges, along with upper bounds for stretches for all these edges that sum to at most $m'\beta^2 \log^{3\lambda+3} n$ (by Lemma 16). Moreover, Lemma 18 shows this can be computed with $O(m \log^{O(1)} n)$ work and the depth is

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

Finally, the number of edges in $\bar{E} = \bar{E}' \cup (E(G) \setminus E(G'))$ is at most $m \cdot (\log^3 n/\beta)^\lambda + O(m'(c_2 \log^3 n/\beta)^\lambda)$, giving the desired bound. □

6 Parallel SDD Linear System Solver

In this section, we show how the ingredients developed in the previous sections can be used to derive a parallel algorithm for solving SDD linear systems. In particular, we show that any graph Laplacian system can be solved to an arbitrary accuracy in nearly-linear work and roughly $O(m^{1/3})$ depth:

Theorem 20 *For any fixed $\theta > 0$ and any $\varepsilon > 0$, there is an algorithm that on input an $n \times n$ Laplacian matrix A with m nonzero elements and a vector b, computes with high probability 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 where the exponent of the $\log n$ term is an absolute constant.*

By Lemma 2, which describes a reduction from any SDD linear system to a Laplacian system, this theorem directly implies a solver with the same guarantees for the general SDD case, proving Theorem 1.

At the current stage, however, this result is mainly of theoretical interest: the exponent of $\log n$ in the work term is somewhat large as a direct consequence of Theorem 19, which accumulates these factors from the techniques used in Sects. 4 and 5. For this reason, we do not focus on optimizing for polylogarithmic factors in the solver as we believe more efficient parallel routines for these are needed before one should attempt to do so. For this reason also, we follow a conceptually simpler solver algorithm [19] instead of the faster algorithm given in [20].

Next, we motivate the technical discussion in this section by providing a high-level description of SDD linear system solvers.

6.1 Background and Overview

All known nearly-linear time solvers for SDD linear systems to date follow a "template" developed in the seminal work of Spielman and Teng [29]. The algorithm we will derive is a parallelization of this template and essentially mirrors previous work on parallel solvers for SDD linear systems related to planar graphs [17, 18].

The algorithms in this framework are recursive in nature. Each recursive step attempts to solve a given linear system in A. First, it constructs a graph B, known as an ultrasparsifier, which has fewer edges than A but approximates A spectrally up to a certain bound. Then, it solves the linear system in A using a preconditioned iterative method, where each iteration of the method involves solving a linear system in B. To solve a system in B, the algorithm applies a partial Cholesky factorization to reduce the number of variables and proceeds to call the solver routine recursively on the rest of B. Eventually, when the system becomes sufficiently small, it is solved directly.

In the above template, the number of linear systems in B that need to be recursively solved depends on the quality of B with respect to A. Previous works showed how to construct an ultrasparsifier such that for a parameter $C = C(n, m)$, the algorithm solves an n-by-n Laplacian with m nonzeros by making C recursive calls to solves on systems of size at most $\frac{m}{2C}$. In these algorithms, C is about $\log^c n$ for some constant c. This leads to the following recurrence: the sequential running time to solve a system of with m nonzeros, $T(m)$, is

$$T(m) \le C \cdot T\left(\frac{m}{2C}\right) + O(C \cdot m)$$

The $C \cdot T(\frac{m}{2C})$ term is the cost of the preconditioned iterative method, corresponding to recursively solving C systems, each of size at most $\frac{m}{2C}$. The $O(C \cdot m)$ term is the cost of incorporating the solutions from each of those solves. This pattern of recursive calls is shown in Fig. 1. In terms of the total work, since the size of each subsequent layer is geometrically decreasing, the recurrence solves to $T(m) = O(C \cdot m)$.

From this high-level description, developing a parallel solver in this framework requires, at minimal, answering the following questions:

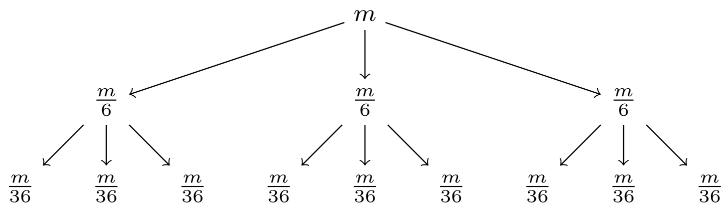

Fig. 1 Example of 3 layers of the call structure with $C = 3$. Total sizes at the 3 layers are m, $\frac{m}{2}$ and $\frac{m}{4}$ respectively

1. How can we construct ultrasparsifiers in parallel?
2. How can we compute a partial Cholesky factorization in parallel?

We will first focus on ultrasparsifiers and how they interact with other solver components. These objects play a critical role in controlling the size of the recursive subproblems.

Ultrasparsifiers An ultrasparsifier of a graph G is a graph with a fraction of the edges of G that approximates G spectrally. The following definition is borrowed from Spielman and Teng [29, Sect. 1.2] but is specialized to graph Laplacians with modified constants:

Definition 21 (Ultrasparsifier) A (κ, h)-ultra-sparsifier of a graph G is a graph H such that

1. $H \preceq G \preceq \kappa H$; and
2. H has at most $n - 1 + h \cdot m/\kappa$ edges.

In the sequential setting, we can generate an ultrasparsifier using the `IncrementalSparsify` routine from Koutis et al. [19]. Combined with subsequent improvements due to Kelner and Levin [15], its guarantees can be stated as follows:

Lemma 22 [19, Theorem 6.2], [15] *Let G be a graph with n vertices and m edges. Let a spanning tree T of G and an upperbound vector on the stretch values $\widehat{\mathsf{str}} : E(G) \to \mathbb{R}_+$ be also given. Then, for a parameter κ,* `IncrementalSparsify` *computes with high probability a graph H such that*

1. $H \preceq G \preceq \kappa H$
2. *H has at most $n - 1 + O(S \log n/\kappa)$ edges where $S = \sum_e \widehat{\mathsf{str}}(e)$.*

We parallelize this construction in Sect. 6.2, using the low-stretch subgraphs algorithm (Theorem 19) as a subroutine.

The Solver Chain The recursive solver algorithm can be viewed as working with a chain of progressively smaller matrices. This chain must be carefully constructed to ensure the right tradeoffs between how fast the number of nonzeros drops and the number of iterations needed to solve each system to a desired accuracy. The following definition quantifies this tradeoff in terms of condition numbers:

Definition 23 (Solver Chain) A *solver chain* for A is a sequence of matrices $\langle A_1 = A, B_1, A_2, B_2, \dots, A_d \rangle$, along with condition numbers $\kappa_1 \dots \kappa_{d-1}$, satisfying

1. $B_i \preceq A_i \preceq \frac{1}{10}\kappa_i B_i$;
2. $m_i \le m_{i+1}/2\sqrt{\kappa_i}$, where m_i is the number of nonzeros in A_i; and
3. A_{i+1} is generated from B_i by applying a partial Cholesky factorization.

This definition deserves further discussion. First, the definition does not specify how the partial Cholesky factorization must be done; it only requires that B_i can be written as

$$B_i = P_i U_i^\top \begin{pmatrix} D & \mathbf{0}^\top \\ \mathbf{0} & A_{i+1} \end{pmatrix} U_i P_i^\top,$$

where U_i is an upper-triangular matrix with ones in the diagonal and P_i is a permutation matrix. This affords us flexibility in performing the factorization. In the sequential setting, the factorization is typically performed using a `GreedyElimination` procedure, which amounts to removing all degree-1 and 2 nodes using Gaussian elimination. Our parallel algorithm will be less aggressive and may leave a small number of degree-2 nodes.

Second, in the view of our template, B_i is essentially an ultrasparsifier of A_i. Therefore, the algorithm solves a system in A_i using the preconditioned Chebyshev iteration with B_i as a preconditioner. Each iteration involves a matrix-vector multiplication with the matrix A_i, other simple vector-vector operations, and solving one linear system in B_i. The iterative method maintains an approximate solution which becomes progressively more accurate. Under a suitable norm, the error decreases by a constant factor every $O(\sqrt{\kappa_i})$ iterations.

The B_i's, however, are still not easy to solve directly. Therefore, we solve a system in B_i by first partially solving it (via a partial Cholesky factorization) and calling the solver routine recursively on the remaining system (in A_{i+1}). Spielman and Teng show that one needs at most $\sqrt{\kappa_i}$ recursive calls to solves involving A_{i+1} to obtain a fixed accuracy. The following lemma is a restatement of Lemma 5.3 from [29] modified so that the $\sqrt{\kappa_i}$ term does not involve a constant.

Lemma 24 [29, Lemma 5.3] *Given a solver chain of length d, it is possible to construct linear operators solve_{A_i} for all $i < d$ such that*

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

and solve_{A_i} is a polynomial of degree $\sqrt{\kappa_i} - 1$ involving $\mathsf{solve}_{A_{i+1}}$ and 4 matrices-vector multiplies involving matrices with $O(m_i)$ nonzero entries obtained from the partial Cholesky factorization.

Deriving a Parallel Solver Several minor modifications to the solver chain will be needed to obtain a parallel solver. It is already clear that the total work/depth will heavily depend on the choice of κ_i's. To shed more light on the multilevel solver and understand its parallel complexity, let us assume for a moment that all κ_i are equal to a fixed κ. This choice of parameters will be addressed in detail in Lemma 30, and improvements of it will lead to Theorem 20. Below, we provide an informal version of its analysis.

With this parameter, level i of the solver is visited $t_i = (\sqrt{\kappa})^i$ times. These visits need to be performed *sequentially*; this is the main bottleneck in the parallelization of the solver. Other than that, each visit at level i performs $O(1)$ matrix-vector multiplications with A_i and other vector-vector operations; all these require $O(m_i)$ work and $O(\log n)$ depth. In addition, vectors are mapped between levels i and $i+1$ using matrices obtained via the partial Cholesky procedure; as we will show in Sect. 6.3, this can be done in $O(m_i)$ work and $O(\log n)$ depth.

For a desired precision $\varepsilon > 0$, the accuracy at $A_1 = A$ (our input system) can be boosted to the desired accuracy by iterating the whole multilevel procedure $\log(1/\varepsilon)$ times. Therefore, the overall work is in the order of $\sum_{i=1}^{d} m_i t_i \log(1/\varepsilon)$, which is bounded by $O(m \log(1/\varepsilon))$ because the size reduction in m_i's is faster than the growth in t_i's. Specifically, we have $m_i t_i \le m/2^i$, which means that about half of the total work is performed at the first level of the chain.

As for depth, the overall depth of this multilevel procedure is

$$O\left(\sum_{i=1}^{d} (\sqrt{\kappa})^i \log n \log(1/\varepsilon)\right).$$

We control the depth of the solver by "shortening" the chain in the following ways:

First, it is always possible to terminate the chain when the matrix A_d has size roughly $O(m^{1/3})$. Then, we can in $O(m \log^{O(1)} n)$ work and $O(m^{1/3})$ depth factorize the inverse of A_d as $LL^\top$, and use the factorization to solve each system in A_d in $O(\log n)$ depth and $O(m^{2/3})$ work. A side-effect of this is that it increases the depth of constructing the chain to $O(m^{1/3})$.

Second, we can aim for a size reduction rate which is as high as possible for a fixed κ and not a mere $\sqrt{c \cdot \kappa}$ required to keep the total work bounded. This allows the reduction rate of problem sizes, m_i/m_{i+1} to approach $\sqrt{\kappa_i}$. On the other hand, it leads to an increase in total work since the work at first level depends on $\sqrt{\kappa_i} m_1 = \sqrt{\kappa} m$.

Third, we vary the settings κ_i's to accelerate the size reduction towards the bottom of the chain, allowing us to upper bound the exponent of $\log n$ in the work bound by an absolute constant while keeping the same depth.

These calculations are formalized in Sect. 6.4. Before that, we discuss the parallelization of incremental sparsification and the parallel Cholesky factorization.

6.2 Parallel Incremental Sparsification

The first step to parallelize is the construction of ultrasparsifiers. To begin, we briefly outline the sequential solution to `IncrementalSparsify` described in Lemma 22; a detailed description can be found in Section 6 of Koutis et al. [19]. Let $G = (V, E, w)$ be a weighted graph, and T be a (low-stretch) spanning subtree of G. Each edge of the graph is assigned a probability p_e proportional to its stretch with respect to the tree T. We then take $t = O(Sm \log n)/\kappa$ independent samples with replacement according to the probabilities p_e's. Each sample of e is added to the output with weight $w_e/(tp_e)$. The output also consists κ copies of the tree T.

The parallelization of `IncrementalSparsify` has been mostly achieved in Theorem 19; we only need to modify the sequential routine slightly. First, note that procedure `LSSubgraph` of Theorem 19 returns a low-stretch spanning tree T for a subgraph $G_s = (V, E - \bar{E})$ of G, and upper bounds on the stretches of the edges of G_s over T. Because the samples are taken independently, it is then easy to implement the sampling procedure in $O(\log n)$ depth. This leads to an ultrasparsifier H_s of G_s such that

1. $H_s \preceq G_s \preceq \kappa H_s$; and
2. H_s has $n - 1 + O(S \log n)/\kappa$ edges,

where S is the sum of the upper bounds on stretches (i.e., $S = \sum_e \widehat{\mathsf{str}}(e)$). To obtain an incremental sparsifier for G, we will simply add the edges in $\bar{E}$ back to H_s. If we let $\bar{G}$ denote the graph formed by these edges, the incremental sparsifier becomes $H = H_s \cup \bar{G}$ and has $n + O(mS \log n)/\kappa + |\bar{E}|$ edges.

Formally, we are able to derive the following parallelization of the incremental sparsification algorithm.

Lemma 25 *Given a weighted graph G, parameters λ and η such that $\eta \geq \lambda \geq 16$, there is a procedure* `parIncrementalSparsify`(G, λ, η) *that in $O(\log^{2\eta\lambda} n)$ depth and $O(m \log^{O(1)} n)$ work returns with high probability 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 Consider running `LSSubgraph`$(G, \log^\eta n, \lambda)$ and let the output be T, $\bar{E}$, and $\widehat{\mathsf{str}}$. We will pick $\kappa = \frac{1}{10} \cdot \log^{\eta\lambda} n$. Let $G_s = (V, E \setminus \bar{E})$ and let H_s be the graph returned by running the independent sampling process given in IncrementalSparsify$(G_s, T, \widehat{\mathsf{str}}, \kappa)$ in parallel. Then, Lemma 22 gives that with high probability H_s satisfies $H_s \preceq G_s \preceq \kappa H_s$

Thus, setting $H = H_s + \bar{G}$ gives $H_s + \bar{G} \preceq G_s + \bar{G} = G$ and

$$\begin{aligned} G &= G_s + \bar{G} \\ &\preceq \frac{1}{10}\kappa H_s + \bar{G} \\ &\preceq \frac{1}{10}\kappa H_s + \frac{1}{10}\kappa \bar{G} \\ &= \frac{1}{10}\kappa H. \end{aligned}$$

This gives the first required condition.

We now bound the numbers of edges in H. Theorem 19 shows that the stretch upper bounds sum to at most $O(m\beta^2 \log^{3\eta+3} n) = O(m \log^{2\lambda+3\eta+3} n)$. Then, after sampling, the number of edges in H_s is bounded by

$$n-1+m\left(\frac{10C_{IS}\log^{2\eta+3\lambda+4} n}{\log^{\eta\lambda} n}\right),$$

where c_{IS} is the hidden constant in the sampling procedure. Also, Theorem 19 guarantees that the number off "extra" edges in the set $\bar{E}$ is at most

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

Hence, the total number of edges in the final sparsifier H which consists of the edges in H_s as well as $\bar{E}$ is bounded by

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

□

6.3 Parallel Greedy Elimination

When solving a linear system via exact operations, a key step is the sequential elimination of variables/nodes from the system. Algebraically, we can exploit the symmetry and obtain a partial Cholesky factorization of the form

$$L_B=PU^{\top}\begin{pmatrix} D & \mathbf{0}^{\top} \\ \mathbf{0} & L_{A'} \end{pmatrix}UP^{\top},$$

where U is an upper-triangular matrix with ones in the diagonal and P is a permutation matrix. Observe that for Laplacians, the lower block of the central factor is still a Laplacian.

Direct solvers typically reorder the matrix B, at least conceptually, for efficiency in the elimination. In particular, these algorithms almost always begin by eliminating nodes of degrees 1 and 2; this can be accomplished in $O(m_B)$ time, where m_B is the number of nonzeros in B. In our case, since we are only interested in partial elimination, this is the only type of variable elimination that we use.

To apply the elimination algebraically, the corresponding vertices must be moved to the top coordinates of the matrix. This amounts to computing a permutation matrix P so that we actually compute the partial factorization

$$P^{\top}L_BP=U^{\top}\begin{pmatrix} D & \mathbf{0}^{\top} \\ \mathbf{0} & L_{A'} \end{pmatrix}U,$$

where P is the permutation matrix corresponding to a permutation π. In practice, we only need to store π since an application of P on a vector x is equivalent to a permutation of its coordinates according to π.

The computation of π and the factorization of $P^\top L_B P$ can be done in $O(m_B)$ time as noted above, via the standard procedure `GreedyElimination` which can be found, for example, in the solver constructions by Spielman and Teng [29] or Koutis et al. [19, 20]. From a graph-theoretic point of view, this procedure greedily removes degree-1 nodes and splices out degree-2 nodes (i.e., it replaces them with an edge joining the two neighbors).

The first step towards parallelizing the solver is the parallelization of greedy elimination. The sequential version of `GreedyElimination` returns a graph with no degree-1 or degree-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 running time of the solver.

Lemma 26 *If G has n vertices and $n-1+m$ edges, then there is a procedure* `parGreedyElimination`(G) *that runs in $O(n+m)$ work and $O(\log n)$ depth, and with high probability returns permutation matrices $P, P^\top$ along with a factorization*

$$P^\top L_G P = U^\top \begin{pmatrix} I & \mathbf{0}^\top \\ \mathbf{0} & L_{G'} \end{pmatrix} U,$$

where G' has at most $2m-2$ nodes and matrix-product vectors with U and $U^\top$ can be computed in $O(m)$ work and $O(\log n)$ depth.

Proof We first take a graph-theoretic look at the elimination. The sequential `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., [29] and [17, Sect. 2.3.4]). Thus, the problem is a slight generalization of parallel tree contraction [22]. 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-2 nodes, in addition to all degree-1 vertices.

We loop over two steps until the vertex count is at most $2m-2$:

1. Mark an independent set of degree-2 vertices.
2. Contract all degree-1 vertices.
3. Compress and/or contract out the marked vertices.

The contract and compress operations in steps 2 and 3 are equivalent to `Rake` and `Compress` in [22], and so they can be performed with the desired work and depth.

To find the independent set in step 1, we use a randomized marking algorithm on the degree-2 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, i.e. vertices in excess of $2m-2$. Let G be a multigraph with n vertices and $m+n-1$ edges. First, observe that if all vertices have degree at least 3 then $n \le 2(m-1)$ and we would be finished. So, let T be any fixed spanning tree of G.

Let a_1, a_2 and a_3 the number of vertices in T of degree $1, 2$ and at least 3 respectively. Similarly, let b_1, b_2, and b_3 be the number of vertices in G of degree 1, 2, and at least 3, respectively; here 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 the 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

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

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. Without loss of generally, 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.

The identities of vertices that get eliminated in every round of the above iteration are now known: they are the degree-1 vertices or the marked degree-2 vertices. Given these, it is fairly easy to compute P, $P^\top$, and the factorization in $O(m)$ work and $O(\log n)$ depth. Details on this can be found in [23]. □

6.4 Parallel Performance of Solver Chain

It can be observed that as long as κ_i are set to polylog$n$, `parGreedyElimination` and `parIncrementalSparsify` allows the parallel construction of a preconditioning chain in polylog$n$ depth and $O(m \log^{O(1)} n)$ work with high probability. As a result, with high probability we obtain a solver chain as specified in Definition 23.

As noted above, the recursive Preconditioned Chebyshev 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 Sects. 3.4. and 4.2 of [11].

Fact 27 *A factorization $LL^\top$ of the inverse of an n-by-n positive definite matrix A, where L is a lower triangular matrix, can be computed in $O(n)$ depth and $O(n^3)$ work, and any solves thereafter can be done in $O(\log n)$ time and $O(n^2)$ work.*

Although A_d is not positive definite when A_d is a Laplacian, its null space is precisely known: it is the space spanned by the constant vector when the underlying graph is connected. Based on this, we can drop the first row and column of A_d to obtain a definite matrix A'_d on which the above factorization is possible. Then, to find a solution of $A_d x = b$ we instead solve the system $A'_d y = b'$ where b' is obtained by b by dropping its last coordinate. We recover a solution x' by padding y with a 0 in the last coordinate. To obtain a solution vector x orthogonal to the null space, it suffices to subtract from x' a copy of the constant vector.

Spielman and Teng [29, Sect. 5] gave a (sequential) time bound for solving a linear SDD system given a preconditioner chain. The following lemma extends Theorem 5.5 from their paper to give parallel runtime bounds (work and depth), as a function of κ_i's and m_i's. Note that in this bound, the m_d^2 term arises from the dense inverse used to solve the linear system in the bottom level.

Lemma 28 *For $\ell \geq 1$, given any vector b, the vector $\mathsf{solve}_{A_\ell} \cdot b$ can be computed in depth*

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

and work

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

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 Lemma 24 gives that 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 $O(m_\ell)$ nonzero entries. This polynomial leads to a sequence of $\sqrt{\kappa_\ell}$ calls, giving a total depth of:

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

and the total work can be bounded by:

$$\sqrt{\kappa_\ell} O(m_\ell) + \sqrt{\kappa_\ell} \cdot O\left(\sum_{\ell+1 \le i \le d-1} m_i \cdot \prod_{\ell+1 \le j \le i} \sqrt{\kappa_j} + m_d^2 \prod_{\ell+1 \le j < d} \sqrt{\kappa_j}\right)$$

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

□

Corollary 29 *Given a preconditioner chain $\mathcal{C}$ for a matrix A, a vector b, and an error tolerance ε, we can compute a vector $\tilde{x}$ such that*

$$\left\|\tilde{x} - A^{+}b\right\|_A \le \varepsilon \cdot \left\|A^{+}b\right\|_A,$$

with depth bounded by

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

and work bounded by

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

Proof The ε-accuracy bound follows from applying preconditioned Chebyshev to solve_{A_1} similarly to Theorem 5.5 of [29], and the work/depth bounds follow from Lemma 28 when $\ell = 1$. □

Corollary 29 shows that the algorithm's performance is determined by the settings of κ_i's and m_i's; however, as we will use Lemma 25, 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 30 *For any fixed $\theta > 0$, if we construct a preconditioner chain using Lemma 25 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 $O(m \log^{O(\lambda^2)} \log 1/\varepsilon)$ work as $\lambda \to \infty$, where ε is the accuracy precision of the solution, as defined in the statement of Theorem 1.*

Proof By Lemma 22, we have that m_{i+1}—the number of edges in level $i + 1$—is bounded by

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

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 $i < d$,

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

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

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

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

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

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

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

So, setting $\delta \ge \frac{7}{\lambda-14}$ suffices to bound the total work by $O(m\log^{O(1)} n)$. 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

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

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

This gives us an algorithm that runs in $O(m^{1/3+\theta})$ depth. However, the exponent on in $\log^n$ also increases along with θ. In order to obtain bounds in Theorem 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 keeping the total depth at $O(m^{1/3+\theta})$.

Proof of Theorem 20 Consider setting $\lambda = 13$ and $\eta \geq \lambda$. Then,

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

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

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

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

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

To solve A_L, we apply Lemma 30, which is analogous to setting $A_L, \ldots, A_d$ uniformly. The depth required in constructing these preconditioners is $O(m_d + \sum_{j=1}^{d} O(\kappa_i))$ where $O(m_d)$ is the depth of computing the inverse of inverse at the last level. This gives a total of $O(m_d) = O(m^{1/3})$ for constructing the solver chain.

As $L \to \infty$, κ_L becomes arbitrarily large. A consequence of Lemma 30 is then $\prod_{L\leq j\leq d}\sqrt{\kappa_j} \leq O(m^{1/3+\theta})$. On the other hand, as L is a constant, $\prod_{1\leq j\leq L}\sqrt{\kappa_j} \in O(\operatorname{polylog} n)$. So as L is set to arbitrarily large constants, the total depth can be bounded by $O(m^{1/3+\theta})$.

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 30 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 25. □

7 Conclusion

We presented a nearly-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 ultrasparse subgraphs. The ultrasparse subgraphs were shown to be useful in the design of a nearly-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 (nearly) 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 require the development of new techniques.

Acknowledgements This work is partially supported by the National Science Foundation under grant numbers CCF-1018463, CCF-1018188, and CCF-1016799, CCF-1149048, by an Alfred P. Sloan Fellowship, and by generous gifts from IBM, Intel, and Microsoft.

References

1. Abraham, I., Bartal, Y., Neiman, O.: Nearly tight low stretch spanning trees. In: FOCS, pp. 781–790 (2008)
2. Alon, N., Karp, R.M., Peleg, D., West, D.: A graph-theoretic game and its application to the k-server problem. SIAM J. Comput. **24**(1), 78–100 (1995)
3. Awerbuch, B.: Complexity of network synchronization. J. Assoc. Comput. Mach. **32**(4), 804–823 (1985)
4. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge University Press, Cambridge (2004)
5. Christiano, P., Kelner, J.A., Madry, A., Spielman, D.A., Teng, S.H.: Electrical flows, Laplacian systems, and faster approximation of maximum flow in undirected graphs. In: STOC, pp. 273–282 (2011)
6. Chvátal, V.: The tail of the hypergeometric distribution. Discrete Math. **25**(3), 285–287 (1979)
7. Cohen, E.: Fast algorithms for constructing t-spanners and paths with stretch t. In: Proceedings of the 1993 IEEE 34th Annual Foundations of Computer Science, pp. 648–658. IEEE Computer Society, Washington (1993). doi:10.1109/SFCS.1993.366822. http://portal.acm.org/citation.cfm?id=1398517.1398961
8. Cohen, E.: Polylog-time and near-linear work approximation scheme for undirected shortest paths. J. ACM **47**(1), 132–166 (2000)
9. Daitch, S.I., Spielman, D.A.: Faster approximate lossy generalized flow via interior point algorithms. CoRR (2008). arXiv:0803.0988
10. Elkin, M., Emek, Y., Spielman, D.A., Teng, S.H.: Lower-stretch spanning trees. In: Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing, pp. 494–503. ACM Press, New York (2005). doi:10.1145/1060590.1060665
11. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. Johns Hopkins University Press, Baltimore (1996)
12. Gremban, K.: Combinatorial preconditioners for sparse, symmetric, diagonally dominant linear systems. Ph.D. thesis, Carnegie Mellon University, Pittsburgh (1996). CMU CS Tech Report CMU-CS-96-123
13. Hoeffding, W.: Probability inequalities for sums of bounded random variables. J. Am. Stat. Assoc. **58**(301), 13–30 (1963). doi:10.2307/2282952

14. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Reading (1992)
15. Kelner, J.A., Levin, A.: Spectral sparsification in the semi-streaming setting. In: Schwentick, T., Dürr, C. (eds.) 28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011). Leibniz International Proceedings in Informatics (LIPIcs), vol. 9, pp. 440–451. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl (2011)
16. Klein, P.N., Subramanian, S.: A randomized parallel algorithm for single-source shortest paths. J. Algorithms **25**(2), 205–220 (1997)
17. Koutis, I.: Combinatorial and algebraic algorithms for optimal multilevel algorithms. Ph.D. thesis, Carnegie Mellon University, Pittsburgh (2007). CMU CS Tech Report CMU-CS-07-131
18. Koutis, I., Miller, G.L.: A linear work, $O(n^{1/6})$ time, parallel algorithm for solving planar Laplacians. In: SODA, pp. 1002–1011 (2007)
19. Koutis, I., Miller, G.L., Peng, R.: Approaching optimality for solving SDD linear systems. In: FOCS, pp. 235–244 (2010)
20. Koutis, I., Miller, G.L., Peng, R.: A nearly $m \log n$ time solver for SDD linear systems. In: FOCS, pp. 590–598 (2011). doi:10.1109/FOCS.2011.85
21. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes. Morgan Kaufmann Publishers, San Francisco (1992)
22. Miller, G.L., Reif, J.H.: Parallel tree contraction part 1: fundamentals. In: Micali, S. (ed.) Randomness and Computation, vol. 5, pp. 47–72. JAI Press, Greenwich (1989)
23. Pan, V.Y., Reif, J.H.: Fast and efficient parallel solution of sparse linear systems. SIAM J. Comput. **22**(6), 1227–1250 (1993)
24. Renegar, J.: A Mathematical View of Interior-Point Methods in Convex Optimization. Society for Industrial and Applied Mathematics, Philadelphia (2001)
25. Schieber, B., Vishkin, U.: On finding lowest common ancestors: simplification and parallelization. SIAM J. Comput. **17**(6), 1253–1262 (1988). doi:10.1137/0217079
26. Skala, M.: Hypergeometric tail inequalities: ending the insanity (2009). http://ansuz.sooke.bc.ca/professional/hypergeometric.pdf
27. Spielman, D.A.: Algorithms, graph theory, and linear equations in Laplacian matrices. In: Proceedings of the International Congress of Mathematicians (2010)
28. Spielman, D.A., Srivastava, N.: Graph sparsification by effective resistances. In: STOC, pp. 563–568 (2008)
29. Spielman, D.A., Teng, S.H.: Nearly-linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. CoRR (2006). arXiv:cs/0607105
30. Teng, S.H.: The Laplacian paradigm: emerging algorithms for massive graphs. In: Theory and Applications of Models of Computation, pp. 2–14 (2010)
31. Ullman, J.D., Yannakakis, M.: High-probability parallel transitive-closure algorithms. SIAM J. Comput. **20**(1), 100–125 (1991). doi:10.1137/0220006
32. Ye, Y.: Interior Point Algorithms: Theory and Analysis. Wiley, New York (1997). http://www.worldcat.org/oclc/36746523