SIAM J. SCI. COMPUT.
Vol. 43, No. 6, pp. C411–C438




# RCHOL: RANDOMIZED CHOLESKY FACTORIZATION FOR SOLVING SDD LINEAR SYSTEMS*

CHAO CHEN†, TIANYU LIANG†, AND GEORGE BIROS†

**Abstract.** We introduce a randomized algorithm, namely, `rchol`, to construct an approximate Cholesky factorization for a given Laplacian matrix (a.k.a., graph Laplacian). From a graph perspective, the exact Cholesky factorization introduces a clique in the underlying graph after eliminating a row/column. By randomization, `rchol` only retains a sparse subset of the edges in the clique using a random sampling developed by Spielman and Kyng [*private communication*, 2020]. We prove `rchol` is breakdown free and apply it to solving large sparse linear systems with symmetric diagonally dominant matrices. In addition, we parallelize `rchol` based on the nested-dissection ordering for shared-memory machines. We report numerical experiments that demonstrate the robustness and the scalability of `rchol`. For example, our parallel code scaled up to 64 threads on a single node for solving the three-dimensional Poisson equation, discretized with the 7-point stencil on a $1024 \times 1024 \times 1024$ grid, a problem that has *one billion* unknowns.

**Key words.** randomized numerical linear algebra, incomplete Cholesky factorization, sparse matrix, symmetric diagonally dominant matrix, graph Laplacian, random sampling

**AMS subject classifications.** 65F08, 65F50, 62D05

**DOI.** 10.1137/20M1380624

## 1. Introduction.

We consider the solution of a large sparse linear system

$$A\,x = b, \tag{1.1}$$

where $A = (a_{ij}) \in \mathbb{R}^{N\times N}$ is a symmetric diagonally dominant (SDD) matrix, i.e.,

$$A = A^\top, \quad \text{and} \quad a_{ii} \geq \sum_{j\neq i} |a_{ij}| \quad \text{for } i = 1, 2, \ldots, N. \tag{1.2}$$

Note we require the diagonal of an SDD matrix to be nonnegative.[1] The linear system (1.1) appears in many scientific and engineering domains, e.g., the discretization of a partial differential equation (PDE) using finite difference or finite elements, spectral graph partitioning, and learning problems on graphs.

The essential ingredient of our method is the randomized Cholesky factorization (`rchol`). When $A$ has only negative nonzero off-diagonal entries , `rchol` computes an

*Submitted to the journal's Software and High-Performance Computing section November 16, 2020; accepted for publication (in revised form) August 31, 2021; published electronically December 21, 2021.

https://doi.org/10.1137/20M1380624

**Funding:** This material is based upon work supported by NIH award 5R01NS042645-11A1; by NSF award CCF-1817048; by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC0019393, by the U.S. Department of Energy, National Nuclear Security Administration Award Number DE-NA0003969; and by the Portugal Foundation for Science and Technology (FCT) and the UT Austin-Portugal program. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the AFOSR, DOE, NIH, and NSF. Computing time on the Texas Advanced Computing Centers Stampede 2 and Frontera systems were provided by an allocation from TACC and the NSF.

†University of Texas at Austin, Austin, TX 78705 USA (chenchao.nk@gmail.com, liangty1998@hotmail.com, biros@oden.utexas.edu).

[1] A relaxed definition requires $|a_{ii}| \geq \sum_{j\neq i} |a_{ij}|$ allowing negative diagonal entries. This relaxed definition is *not* what we use in this paper.

approximate Cholesky factorization

$$P^\top AP \approx GG^\top, \tag{1.3}$$

where $P$ is a permutation matrix and $G$ is a lower triangular matrix. Using $GG^\top$ as the preconditioner, we can solve (1.1) with the PCG method [36]. Generally, $A$ also has positive off-diagonal entries. In some cases (section 3.2.1), we can find a diagonal matrix $D$ with $+1$ or $-1$ on the diagonal such that $DAD$ has only negative nonzero off-diagonal entries; otherwise, we solve an equivalent linear system that has only negative nonzero off-diagonal entries but is twice larger.

**1.1. Related work.** Direct solvers compute exact factorizations of $A$ and generally require $\mathcal{O}(N^3)$ work and $\mathcal{O}(N^2)$ storage. Although matrix $A$ is sparse, a naive direct method may introduce excessive new nonzero entries (a.k.a., fill-in) during the factorization. To minimize fill-in, sparse-matrix reordering schemes, such as nested dissection [12] and approximate minimum degree (AMD) [2], are usually employed in state-of-the-art methods, namely, sparse direct solvers [9]. One notable example is the nested-dissection multifrontal method [11,28], where the elimination ordering and the data flow follow a special hierarchy of *separator fronts*. When applied to matrix $A$ from the discretization of PDEs in three-dimensional (3D) space, the multifrontal method generally reduces the computation and memory complexities to $\mathcal{O}(N^2)$ and $\mathcal{O}(N^{4/3})$, respectively. However, such costs, dominated by those for factorizing the largest separator front of size $\mathcal{O}(N^{2/3})$, are still prohibitive for large-scale problems.

Preconditioned iterative methods are often preferred for large-scale problems [36]. A key design decision in iterative solvers is the preconditioner. State-of-the-art methods such as domain decomposition and multigrid methods work efficiently for a large class of problems, including SDD matrices. A cheaper and simpler alternative is to use an approximate factorization as in (1.3), and one popular strategy to compute such a factorization is the incomplete factorization [32]. An incomplete factorization permits fill-in at only specified locations in the resulting factorization. These locations can be computed in two ways: statically, based on the sparsity structure of $A$ with a level-based strategy, or dynamically, generated during the factorization process with a threshold-based strategy [35] or its variants [17,37]. Because of its importance, an incomplete Cholesky factorization is often parallelized on single-node shared-memory machines, and this type of parallel algorithm has been studied extensively [3,7,22,34]. Incomplete factorizations are widely used in computational science and engineering, especially when the underlying physics of a problem are difficult to exploit. Besides being used as a stand-alone preconditioner, an incomplete factorization is also an important algorithmic primitive in more sophisticated methods. For example, it can be used to precondition subdomain solves in domain decomposition schemes or as a smoother in multigrid methods. In this paper, we focus on a randomized scheme for constructing incomplete factorizations. Although we compare our method directly with other solvers, we would like to emphasize that we envision it as an algorithmic primitive in more complex solvers.

More recently, a class of methods known as the Laplacian paradigm have been developed specifically for solving SDD linear systems as in (1.1). In a breakthrough [39], Spielman and Teng proved in 2004 that (1.1) can be solved in nearly-linear time. Despite the progress with asymptotically faster and simpler algorithms [21, 23, 25, 26], practical implementations of these methods that are able to compete with state-of-

the-art linear solvers are limited [24, 29]. A notable recent effort is `Laplacians.jl`,[2] a Julia package containing linear solvers for Laplacian matrices, but no results have been reported for solving problems related to PDEs, the target application of our work. In this paper, we build on two established ideas: the SparseCholesky algorithm in [25] and a random sampling scheme implemented in `Laplacians.jl`. In the SparseCholesky algorithm, the Schur-complement update is written as a diagonal matrix plus the graph Laplacian of a clique. Then edges in the clique are sampled and reweighted, so the graph Laplacian of sampled edges equals that of the clique in expectation. In `Laplacians.jl`, Spielman and Kyng [38] proposed another sampling strategy, which empirically performed better but has not been analyzed, according to our knowledge and the software documentation.

**1.2. Contributions.** In this work, we focus on solving SDD linear systems arising from the discretization of PDEs, and the main ingredient of our approach is an approximate Cholesky factorization constructed via random sampling. In particular, we introduce a randomized Cholesky factorization for Laplacian matrices building on top of previous work by Spielman and Kyng [25, 38]. As observed in [25], eliminating a row/column in the matrix is equivalent to subtracting the graph Laplacian of a *star* and adding the graph Laplacian of a *clique*. Following [38], we sample a sparse subset of the edges instead of keeping the full clique. Our specific contributions include the following:

- We prove that the sampled edges form a spanning tree on the clique, and consequently, `rchol` is breakdown free for an irreducible Laplacian matrix. We also extend `rchol` to compute approximate factorizations for subclasses of SDD matrices that are not Laplacian matrices. For the rest of SDD matrices that we cannot apply `rchol` directly, we clarify how to obtain an approximate solution of (1.1) under a given tolerance through solving an extended problem using PCG.
- We introduce a high-performance parallel algorithm for `rchol` based on the nested-dissection ordering and the multifrontal method. We implemented the parallel algorithm using a task-based approach for shared-memory multicore machines. Our software offering C++/MATLAB/Python interfaces is available at https://github.com/ut-padas/rchol.
- We benchmarked our code on various problems: Poisson's equation, variable-coefficient Poisson's equation, anisotropic Poisson's equation, and problems from the SuiteSparse Matrix Collection.[3] With our benchmark results, we demonstrated the importance of using fill-reducing orderings, the stability and the scalability of our method. We also compared our method to the well-established incomplete Cholesky factorization with threshold dropping.

Our results highlight several features of the new method that are distinct from existing deterministic incomplete Cholesky factorizations: (1) Fill-reducing ordering (as opposed to natural/lexicographical ordering), such as AMD and nested-dissection, improved the performance of our method; (2) the number of iterations required by PCG increased approximately logarithmically with the problem size for discretized 3D Poisson equation; and (3) the performance of our parallel algorithm is hardly affected by the number of threads used.

[2] https://github.com/danspielman/Laplacians.jl.
[3] https://sparse.tamu.edu/.

**1.3. Outline and notations.** The remainder of this paper is organized as follows. Section 2 introduces `rchol` with analysis. Section 3 focuses on solving SDD linear systems and the parallel algorithm for `rchol`. Section 5 presents numerical experiments, and section 6 discusses generalizations and draws conclusions.

Throughout this paper, matrices are denoted by capital letters with their entries given by the corresponding lowercase letter in the usual way, e.g., $A = (a_{ij}) \in \mathbb{R}^{N\times N}$. We adopt the MATLAB notation to denote a submatrix; e.g., $A(i,:)$ and $A(:,i)$ stand for the $i$th row and $i$th column in matrix $A$, respectively.

**2. Randomized Cholesky factorization for Laplacian matrix.** In this section, we focus on irreducible Laplacian matrices, which can be viewed as weighted undirected graphs that have only one connected component. Then we introduce Cholesky factorization and give the first formal statement of the clique sampling scheme by Spielman and Kyng [38] in the Laplacians.jl package. Finally, we provide analysis on the resulting randomized Cholesky factorization.

DEFINITION 2.1 (Laplacian matrix [25]). *Matrix $A \in \mathbb{R}^{N\times N}$ is a Laplacian matrix if* (1) $A = A^\top$, (2) $\sum_{j=1}^{N} a_{ij} = 0$ *for* $i = 1, 2, \ldots, N$, *and* (3) $a_{ij} \le 0$ *when* $i \ne j$.

DEFINITION 2.2 (irreducible matrix [40]). *Matrix $A$ is irreducible if there does not exist a permutation matrix $P$ such that $P^\top AP$ is a block triangular matrix.*

LEMMA 2.3 (irreducible Laplacian matrix). *Suppose $A \in \mathbb{R}^{N\times N}$ is an irreducible Laplacian matrix. If $N > 1$, then $a_{ii} > 0$ for all $i = 1, 2, \ldots, N$; otherwise, $A$ is a scalar zero.*

Note a Laplacian matrix is always positive semidefinite, and the null space is span$\{\mathbf{1}\}$ if it is irreducible. Below we state a well-known result that there exists a bijection between the class of Laplacian matrices and the class of weighted undirected graphs to prepare for the sampling algorithm.

DEFINITION 2.4 (graph Laplacian). *Let $\mathcal{G} = (V, E)$ be a weighted undirected graph, where $V = (v_1, v_2, \ldots, v_N)$ and an edge $e_{ij} = (v_i, v_j) \in E$ carries weight $w_{ij} > 0$. The graph Laplacian of $\mathcal{G}$ is*

$$L = \sum_{e_{ij}\in E} w_{ij}\, \boldsymbol{b}_{ij}\boldsymbol{b}_{ij}^\top, \tag{2.1}$$

*where $\boldsymbol{b}_{ij} = \boldsymbol{e}_i - \boldsymbol{e}_j$, the difference of two standard bases $\boldsymbol{e}_i, \boldsymbol{e}_j \in \mathbb{R}^N$ (the order of difference does not affect $L$).*

*Remark* 2.5. For completeness, we also mention another equivalent definition of graph Laplacian. Given a weighted undirected graph $\mathcal{G} = (V, E)$, the graph Laplacian of $\mathcal{G}$ is

$$L = D - W,$$

where $W$ is the weighted adjacency matrix; i.e., $-w_{ij}$ is the weight associated with edge $e_{ij} \in E$, and $D$ is the weighted degree matrix, i.e., $d_{ii} = -\sum_{j\ne i} w_{ij}$ for all $i$.

THEOREM 2.6. *Definitions* 2.1 *and* 2.4 *are equivalent: Matrix $L$ in* (2.1) *is a Laplacian matrix, and there exists a weighted undirected graph of which the graph Laplacian is equal to a given Laplacian matrix.*

*Proof.* Note that

$$\begin{array}{c} \\ i \\ \\ j \\ \\ \end{array}\!\!\begin{array}{c} \begin{array}{cccc} \quad & i & \quad & j \end{array} \\ \begin{pmatrix} \ddots & & & & \\ & 1 & \dots & -1 & \\ & \vdots & & \vdots & \\ & -1 & \dots & 1 & \\ & & & & \ddots \end{pmatrix} \end{array} = \boldsymbol{b}_{ij}\, \boldsymbol{b}_{ij}^\top,$$

and it is straightforward to verify that $L$ in (2.1) is a Laplacian matrix. In the other direction, for a given Laplacian matrix $A$, we can construct a weighted undirected graph $\mathcal{G}$ based on the weighted adjacency matrix $D-A$, where $D$ contains the diagonal of $A$. According to Remark 2.5, $A$ is the graph Laplacian of $\mathcal{G}$. ∎

**2.1. Cholesky factorization and clique sampling.** Consider applying the Cholesky factorization to an irreducible Laplacian matrix $L \in \mathbb{R}^{N\times N}$ for $N-1$ steps as shown in Algorithm 2.1. It is straightforward to verify that $L$ is always a Laplacian matrix inside the for-loop (line 4). Furthermore, the Schur complement at the $k$th step, i.e., $L(k{+}1{:}N, k{+}1{:}N)$, is an irreducible Laplacian matrix for $k = 1, 2, \dots, N-1$. According to Lemma 2.3, we know that $\ell_{kk} > 0$ at line 3 and $\ell_{NN} = 0$ after the for-loop. An irreducible Laplacian matrix corresponds to a connected graph, and the zero Schur complement, which stands for an isolated vertex, would not occur earlier until the other $N-1$ vertices have been eliminated.

**Algorithm 2.1** Classical Cholesky factorization for Laplacian matrix.

**Input:** irreducible Laplacian matrix $L \in \mathbb{R}^{N\times N}$
**Output:** lower triangular matrix $G \in \mathbb{R}^{N\times N}$
1: $G = \mathbf{0}_{N\times N}$
2: **for** $k = 1$ **to** $N-1$ **do**
3: $\quad G(:,k) = L(:,k)/\sqrt{\ell_{kk}}$ // $\ell_{kk} > 0$ for an irreducible Laplacian input
4: $\quad L = L - \frac{1}{\ell_{kk}}\, L(:,k)\, L(k,:)$ // dense Schur-complement update
5: **end for**

At the $k$th step in Algorithm 2.1, the elimination (line 4) leads to a dense submatrix in the Schur complement. Next, we use the idea of random sampling to reduce the amount of fill-in. At the $k$th step, we define the neighbors of $k$ as

$$\mathcal{N}_k \triangleq \{i : \ell_{ki} \neq 0, i \neq k\}, \tag{2.2}$$

corresponding to vertices connected to vertex $k$ in the underlying graph. We also define the graph Laplacian of the subgraph consisting of $k$ and its neighbors as

$$L^{(k)} \triangleq \sum_{i\in\mathcal{N}_k} (-\ell_{ki})\ \boldsymbol{b}_{ki}\boldsymbol{b}_{ki}^\top. \tag{2.3}$$

It is observed in [25] that the elimination at line 4 in Algorithm 2.1 can be written as the sum of two Laplacian matrices:

$$L - \frac{1}{\ell_{kk}}\, L(:,k)\, L(k,:) = \underbrace{L - L^{(k)}}_{\text{Laplacian matrix}} + \underbrace{L^{(k)} - \frac{1}{\ell_{kk}} L(:,k)L(k,:)}_{\text{Laplacian matrix}}.$$

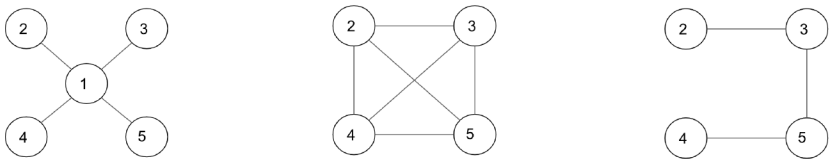


FIG. 1. *An example. (Left): Graph of $L$ before vertex 1 is eliminated. (Middle): Graph of the Schur complement after vertex 1 is eliminated. (Right): A randomly sampled subset of the clique.*

The first term is the graph Laplacian of the subgraph consisting of all edges except the ones connected to $k$. Since

$$L(:,k) - L^{(k)}(:,k) = 0, \quad L(k,:) - L^{(k)}(k,:) = 0,$$

we know $L - L^{(k)}$ zeros out the $k$th row/column in $L$ and updates the diagonal entries in $L$ corresponding to $\mathcal{N}_k$.

The second term

$$L^{(k)} - \frac{1}{\ell_{kk}} L(:,k)L(k,:) = \frac{1}{2} \sum_{i,j \in \mathcal{N}_k} \frac{\ell_{ki}\,\ell_{kj}}{\ell_{kk}}\, \boldsymbol{b}_{ij}\boldsymbol{b}_{ij}^{\top} \tag{2.4}$$

is the graph Laplacian of the clique among neighbors of $k$, where the edge between neighbor $i$ and neighbor $j$ carries weight $\ell_{ki}\,\ell_{kj}/\ell_{kk}$. Denote the number of neighbors of $k$ as $n$, i.e.,

$$n \triangleq |\mathcal{N}_k|.$$

Note (2.4) is a dense matrix with $n^2$ entries or a clique with $\mathcal{O}(n^2)$ edges. The idea of randomized Cholesky factorization is to sample $\mathcal{O}(n)$ edges from the clique (and assign new weights), corresponding to $\mathcal{O}(n)$ fill-in entries. The randomized algorithm is shown in Algorithm 2.2, and the difference from Algorithm 2.1 is shown pictorially with an example in Figure 1.

**Algorithm 2.2** Randomized Cholesky factorization for Laplacian matrix.

**Input:** irreducible Laplacian matrix $L \in \mathbb{R}^{N\times N}$
**Output:** lower triangular matrix $G \in \mathbb{R}^{N\times N}$

```
G = 0_{N×N}
for k = 1 to N − 1 do
    G(:,k) = L(:,k)/√ℓ_kk                    // ℓ_kk > 0 according to Corollary 2.8
    L = L − L^(k) + SampleClique(L,k)        // sparse Schur-complement update
end for
```

The pseudocode of the sampling algorithm is shown in Algorithm 2.3, which selects $n-1$ edges from a clique among $n$ vertices as follows. Before sampling, the neighbors of $k$ are sorted in ascending order based on their weights $|\ell_{ki}|$. For every $i \in \mathcal{N}_k$, we sample $j \in \mathcal{N}_k$ such that $|\ell_{kj}| > |\ell_{ki}|$ with a probability proportional to $|\ell_{kj}|$. Then an edge between $i$ and $j$ is created with an appropriate weight (so the graph Laplacian of the sampled edges is equal to (2.4) in expectation; see Theorem 2.10). Figure 2 shows an example of the sampling process step-by-step.

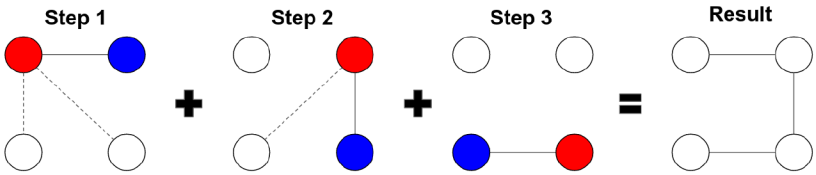


FIG. 2. *An instance of Algorithm* 2.3 *for the example in Figure* 1. *At every step, the red vertex stands for* $i \in \mathcal{N}$ *at line* 5 *in Algorithm* 2.3, *the blue vertex stands for* $j \in \mathcal{N}$ *at line* 8, *the solid line is the sampled edge, and the dashed lines are other potential candidates for sampling.*

**Algorithm 2.3** Sample clique (by Spielman and Kyng [38]).

**Input:** Laplacian matrix $L \in \mathbb{R}^{N\times N}$ and elimination index $k$
**Output:** graph Laplacian of sampled edges $C \in \mathbb{R}^{N\times N}$

1: $C = \mathbf{0}_{N\times N}$
2: Sort $\mathcal{N}_k$ in ascending order based on $|\ell_{ki}|$ for $i \in \mathcal{N}_k$ // $\mathcal{N}_k$ defined in (2.2)
3: $S = \ell_{kk}$ // $\ell_{kk} = -\sum_{i\in\mathcal{N}_k} \ell_{ki}$
4: **while** $|\mathcal{N}_k| > 1$ **do**
5: Let $i$ be the first element in $\mathcal{N}_k$ // loop over neighbors
6: $\mathcal{N}_k = \mathcal{N}_k/\{i\}$ // remove $i$ from the set
7: $S = S + \ell_{ki}$ // $S = -\sum_{j\in\mathcal{N}_k} \ell_{kj}$
8: Sample $j$ from $\mathcal{N}_k$ with probability $|\ell_{kj}|/S$
9: $C = C - \frac{S\,\ell_{ki}}{\ell_{kk}} \boldsymbol{b}_{ij}\boldsymbol{b}_{ij}^\top$ // pick edge $(i,j)$; assign weight $S\,|\ell_{ki}|/\ell_{kk}$
10: **end while**

**2.2. Analysis of randomized Cholesky factorization.** In this section, we prove the robustness and the scalability of `rchol`. The following theorem shows that the edges sampled by Algorithm 2.3 form a spanning tree, and consequently, Algorithm 2.2 never breaks down.

THEOREM 2.7 (spanning tree on clique). *The sampled edges in Algorithm* 2.3 *form a spanning tree of the clique on neighbors of* $k$.

*Proof.* Suppose $k$ has $n$ neighbors. Observe that $n-1$ edges are sampled and that all neighbors are included in the graph formed by these edges. It remains to be proved that this graph is connected.

Suppose the set of neighbors $\mathcal{N}_k$ is sorted in ascending order. We can find a path between any $i \in \mathcal{N}_k$ and the last/"heaviest" element in $\mathcal{N}_k$ with the following rationale:

1. Start from any $i \in \mathcal{N}_k$. Suppose a sampled edge goes from $i$ to a "heavier" neighbor $j \in \mathcal{N}_k$ ($|\ell_{ki}| < |\ell_{kj}|$).
2. Move to $j$, and repeat the previous process. It follows that we will reach the "heaviest" neighbor after a finite number of steps. ∎

COROLLARY 2.8 (breakdown free). *In Algorithm* 2.2, $\ell_{kk} > 0$ *at line* 3, *and* $\ell_{NN} = 0$ *after the for-loop.*

*Proof.* Since Algorithm 2.3 returns a graph Laplacian of a connected graph among the neighbors of $k$ at line 4 in Algorithm 2.2, it is straightforward to verify that the Schur complement at the $k$th step (i.e., $L(k{+}1{:}N, k{+}1{:}N)$) is an irreducible Laplacian matrix. Therefore, this corollary holds according to Lemma 2.3. ∎

The next theorem addresses the time complexity and the storage of `rchol` employing a random elimination ordering, which follows the argument in [25] closely. (We prove this in Appendix A.)

THEOREM 2.9 (running time and storage). *Suppose that an irreducible Laplacian matrix $L \in \mathbb{R}^{N\times N}$ has $M$ nonzeros and that a random row/column is eliminated at every step in Algorithm* 2.2*. Then the expected running time of Algorithm* 2.2 *is upper bounded by $\mathcal{O}(M \log N)$, and the expected number of nonzeros in the output triangular matrix $G$ is upper bounded by $\mathcal{O}(M \log N)$.*

The next theorem shows that Algorithm 2.3 returns an unbiased estimator at every step in Algorithm 2.2.

THEOREM 2.10 (unbiased estimator). *At the $k$th step in Algorithm* 2.2*, the expectation of $C$ = SampleClique(L,k) is equal to the result of exact elimination, as defined in* (2.4)*.*

*Proof.* Suppose $i, j \in \mathcal{N}_k$ and $0 < |\ell_{ki}| < |\ell_{kj}|$. The probability that edge $(i, j)$ being sampled is $P_{ij} = |\ell_{kj}|/S$, according to line 8 in Algorithm 2.3. Therefore, we have

$$\mathbb{E}[C] = \sum_{i,j\in\mathcal{N}_k \text{ and } |\ell_{ki}|<|\ell_{kj}|} P_{ij} \frac{S\,(-\ell_{ki})}{\ell_{kk}}\, \boldsymbol{b}_{ij}\boldsymbol{b}_{ij}^{\top} = \sum_{i,j\in\mathcal{N}_k \text{ and } |\ell_{ki}|<|\ell_{kj}|} \frac{\ell_{kj}\,\ell_{ki}}{\ell_{kk}}\, \boldsymbol{b}_{ij}\boldsymbol{b}_{ij}^{\top}. \qquad \square$$

**2.3. Relation to approximate Cholesky factorizations in [25] and [38].** While both `rchol` and the method in [25] follow the same template of Algorithm 2.2, they differ in two manners. The first difference is that the algorithms of clique sampling are different. In [25] the authors propose to sample $n$ edges from a clique at every step in Algorithm 2.1. To sample an edge, a neighbor $i$ is sampled uniformly from $\mathcal{N}_k$, and a neighbor $j$ is sampled from $\mathcal{N}_k$ with probability $|\ell_{kj}|/\ell_{kk}$; then an edge between $i$ and $j$ is created with weight $\ell_{ki}\ell_{kj}/|\ell_{ki} + \ell_{kj}|$ if $i \neq j$. With such a sampling strategy, an edge can be sampled repeatedly, and there is a probability that no edge is created (when $i$ and $j$ are identical). So Algorithm 2.3 can be viewed as a derandomized variant of the sampling in [25].

The other difference is that there is an extra initialization step before entering Algorithm 2.2 in [25]. For a Laplacian matrix, the initialization is to split every edge in the associated graph into $\rho = \mathcal{O}(\log^2 N)$ copies with $1/\rho$ of the original weight. Then the resulting *multigraph* becomes the input of Algorithm 2.2. It was proven that the norm of the *normalized graph Laplacian* associated with every edge in the multigraph is upper bounded by $1/\rho$ throughout the factorization with the aforementioned sampling algorithm. As a result, a nearly-linear time solver was obtained as the following theorem states.

THEOREM 2.11 (approximate Cholesky factorization in [25]). *Let $L \in \mathbb{R}^{N\times N}$ be an irreducible Laplacian matrix with $M$ nonzeros and $P \in \mathbb{R}^{N\times N}$ be a random permutation matrix. If we perform the above initialization step on $P^{\top}LP$ and apply Algorithm* 2.2 *with the above sampling algorithm, then the expected running time is $\mathcal{O}(\rho M \log N) = \mathcal{O}(M \log^3 N)$, and the expected number of nonzeros in the output triangular matrix $G$ is $\mathcal{O}(\rho M \log N) = \mathcal{O}(M \log^3 N)$. In addition, with high probability,*

$$\frac{1}{2}L \preceq (PG)(PG)^{\top} \preceq \frac{3}{2}L.$$

*(For two symmetric matrices $A$ and $B$, the notation $A \preceq B$ means that $B - A$ is a positive semidefinite matrix.)*

Overall, the algorithm in [25] requires a more expensive factorization than `rchol` (the extra $\log^2 N$ factor in the running time can be significant in practice), but it produces an approximation of better quality.

Compared to [38], `rchol` computes a mathematically equivalent operator if the same elimination ordering is used. (`rchol` by default uses the AMD ordering [2] in practice; see section 5.1.) Hence, our analysis for `rchol` also applies to the method in [38]. While `rchol` represents the output as an approximate Cholesky factorization, [38] uses a row-operation representation.

**3. Randomized preconditioner for SDD matrix.** In this section, we consider an SDD linear system $Ax = b$, where $A$ is irreducible as defined in Definition 2.2 but not a Laplacian matrix. In section 3.1, we consider the case when $A$ is an SDDM matrix, which can be viewed as the sum of a Laplacian matrix and a nonnegative diagonal matrix with at least one positive diagonal entry. In section 3.2.1, we introduce bipartite SDD matrices, a subclass of SDD matrices that contain positive off-diagonal entries but can be converted to either a Laplacian matrix or an SDDM matrix through diagonal scaling.

When $A$ is either an SDDM matrix or a bipartite SDD matrix, we can compute an approximate Cholesky factorization of $A$ and use it as a preconditioner to solve for $x$. Otherwise, it is well known in the literature [15] that $x$ can be obtained through solving a twice larger linear system $\tilde{A}y = \tilde{b}$ *in exact arithmetic*. In section 3.2.2, we show how to retrieve an approximate solution $x$ that has the same relative residual as a given *approximate solution* $y$ for the larger system.

### 3.1. SDDM matrix.

DEFINITION 3.1. *Matrix $A \in \mathbb{R}^{N\times N}$ is a symmetric diagonally dominant M-matrix if $A$ is* (1) *SDD,* (2) *positive definite, and* (3) $a_{ij} \le 0$ *when* $i \neq j$.

Our goal is to compute an approximate Cholesky factorization for an SDDM matrix $A$:

$$A \approx GG^\top. \tag{3.1}$$

The factorization can be used as a preconditioner for solving $Ax = b$. To obtain (3.1), our approach is applying Algorithm 2.2 to the following extended matrix that initially appeared in [15]:

$$\tilde{A} \triangleq \begin{pmatrix} A & -A\mathbf{1} \\ -\mathbf{1}^\top A & \mathbf{1}^\top A\mathbf{1} \end{pmatrix}, \quad \tilde{A} \in \mathbb{R}^{(N+1)\times(N+1)}, \tag{3.2}$$

where $\mathbf{1} \in \mathbb{R}^N$ stands for the all-ones vector. The reason we can apply Algorithm 2.2 is the following lemma.

LEMMA 3.2. *Given an irreducible SDDM matrix $A$, the extended matrix $\tilde{A}$, defined in* (3.2)*, is an irreducible Laplacian matrix.*

*Proof.* Since $A$ is SDD and positive definite, the row-sum vector $A\mathbf{1}$ has nonnegative entries and at least one positive entry. Therefore, it is straightforward to verify that $\tilde{A}$ is an irreducible Laplacian matrix. ∎

Suppose the output of Algorithm 2.2 is the following:

$$\texttt{rchol}(\tilde{A}) \triangleq \tilde{G} = \begin{pmatrix} \tilde{G}_{11} & \\ \tilde{G}_{21} & \tilde{g}_{22} \end{pmatrix}, \tag{3.3}$$

where $\tilde{G}_{11} \in \mathbb{R}^{N\times N}, \tilde{G}_{21} \in \mathbb{R}^{1\times N}$, and $\tilde{g}_{22} \in \mathbb{R}$. We know that $\tilde{g}_{22} = 0$ according to

Corollary 2.8. In other words, we have the approximation

$$\tilde{A} = \begin{pmatrix} A & -A\mathbf{1} \\ -\mathbf{1}^\top A & \mathbf{1}^\top A\mathbf{1} \end{pmatrix} \approx \tilde{G}\tilde{G}^\top = \begin{pmatrix} \tilde{G}_{11} & \\ \tilde{G}_{21} & 0 \end{pmatrix} \begin{pmatrix} \tilde{G}_{11}^\top & \tilde{G}_{21}^\top \\ & 0 \end{pmatrix},$$

from which we see that

$$A \approx \tilde{G}_{11}\tilde{G}_{11}^\top$$

in the leading principle block. We summarize the above algorithm in Algorithm 3.1.

**Algorithm 3.1** Randomized Cholesky factorization for SDDM matrix.

**Input:** irreducible SDDM matrix $A \in \mathbb{R}^{N\times N}$
**Output:** lower triangular matrix $G \in \mathbb{R}^{N\times N}$
1: Construct $\tilde{A}$ defined in (3.2).
2: Compute

$$\begin{pmatrix} \tilde{G}_{11} & \\ \tilde{G}_{21} & 0 \end{pmatrix} = \textsc{RandomizedCholesky}(\tilde{A}) \qquad \text{// call Algorithm 2.2}$$

where $\tilde{G}_{11} \in \mathbb{R}^{N\times N}$ and $\tilde{G}_{21} \in \mathbb{R}^{1\times N}$.
3: **return** $G = \tilde{G}_{11}$.

*Remark* 3.3 (reducible SDDM matrix). In general, Algorithm 3.1 can be applied to an SDDM matrix $A$ that is reducible because (3.2) is still an irreducible Laplacian matrix. However, it may be more efficient to apply Algorithm 3.1 to each irreducible component for solving a linear system with $A$.

Before ending this section, we justify using $\tilde{G}_{11}\tilde{G}_{11}^\top$ as a preconditioner through the following classical result.

THEOREM 3.4 ( [15, Lemma 4.2, page 56]). *Solving an irreducible SDDM linear system $Ax = b$ is equivalent to solving the following irreducible Laplacian linear system*

$$\tilde{A}\, y = \begin{pmatrix} b \\ -\mathbf{1}^\top b \end{pmatrix}. \tag{3.4}$$

*Proof.* It can be verified that the solution of (3.4) is

$$y = \begin{pmatrix} x \\ 0 \end{pmatrix} + \operatorname{span}\{\mathbf{1}\}. \tag{3.5}$$

Therefore, we can solve (3.4) to obtain $x$ and vice versa. ∎

To solve (3.4) and obtain $x$, we first apply PCG with the preconditioner $\tilde{G}\tilde{G}^\top$ in (3.3). Then we orthogonalize the PCG solution with respect to $\operatorname{span}\{\mathbf{1}\}$. This process turns out to be equivalent to using $\tilde{G}_{11}\tilde{G}_{11}^\top$ as the preconditioner (note $\tilde{G}_{11}$ is nonsingular) for solving $Ax = b$ with PCG directly, without going through the extended problem.

**3.2. SDD matrix.** Given an irreducible SDD matrix $A \in \mathbb{R}^{N\times N}$, let

$$A \triangleq A_d + A_n + A_p,$$

where $A_d, A_n, A_p \in \mathbb{R}^{N\times N}$ contain the diagonal, the negative off-diagonal, and the positive off-diagonal entries of $A$, respectively. In this section, we focus on the case when $A_p \neq \mathbf{0}$; i.e., $A$ contains at least two positive off-diagonal entries (due to symmetry).

**3.2.1. Bipartite SDD matrix.** We introduce bipartite SDD matrices and give three equivalent definitions below (proof is in Appendix B).

DEFINITION 3.5. *A bipartite SDD matrix $A$ can be defined in any of the following three equivalent ways:*

(a) *Let $\hat{A}$ be an SDD matrix defined by the off-diagonal part of $A$,*

$$\hat{A} \triangleq \operatorname{diag}\left((A_p - A_n)\mathbf{1}\right) + A_p + A_n, \tag{3.6}$$

*where* $\operatorname{diag}(\cdot)$ *maps a vector to a diagonal matrix. If* $\operatorname{rank}(\hat{A}) = N-1$*, then $A$ is a bipartite SDD matrix.*

(b) *Let $D$ be a diagonal matrix whose diagonal entries are either $1$ or $-1$. If there exists such a matrix $D$ that $DAD$ has only nonpositive off-diagonal entries, then $A$ is a bipartite SDD matrix.*

(c) *Let $\mathcal{G} = (V, E)$ be a undirected graph, where $V = (v_1, v_2, \ldots, v_N)$ has $N$ vertices; an edge $e_{ij} = (v_i, v_j) \in E$ exists if $a_{ij} \neq 0$ and carries weight $w_{ij} = -a_{ij}$. If the graph $\mathcal{G}$ is 2-colorable (bipartite) in the sense that*
   - *$v_i$ and $v_j$ have the same color if $w_{ij} > 0$,*
   - *$v_i$ and $v_j$ have different colors if $w_{ij} < 0$,*

   *then $A$ is a bipartite SDD matrix.*

*Example* 3.6. The following shows three $3\times 3$ SDD matrices with positive off-diagonal entries, where a symbol $\times$ denotes any value greater than or equal to 2. Among the three matrices, $A_1$ is a bipartite SDD matrix, and the other two are not:

$$A_1 = \begin{pmatrix} \times & 1 & 1 \\ 1 & \times & -1 \\ 1 & -1 & \times \end{pmatrix} \quad A_2 = \begin{pmatrix} \times & 1 & -1 \\ 1 & \times & -1 \\ -1 & -1 & \times \end{pmatrix} \quad A_3 = \begin{pmatrix} \times & 1 & 1 \\ 1 & \times & 1 \\ 1 & 1 & \times \end{pmatrix}.$$

*Remark* 3.7. Whether $A$ is a bipartite SDD matrix or not depends on only its off-diagonal part according to Definition 3.5(a). When $A_p \neq \mathbf{0}$, we have $\operatorname{rank}(\hat{A}) = N$ if $A$ is not a bipartite SDD matrix. Otherwise, when $A_p = \mathbf{0}$ ($A$ is either a Laplacian matrix or an SDDM matrix), we have $\operatorname{rank}(\hat{A}) = N-1$.

Our goal is to compute an approximate (generalized) Cholesky factorization of an irreducible bipartite SDD matrix. In the following, we show that it takes linear time to find the matrix $D$ in Definition 3.5(b), and thus we can apply `rchol` to $DAD$, which is either a Laplacian matrix or an SDDM matrix. Given an irreducible SDD matrix, Algorithm 3.2 tries to find the matrix $D$ by traversing the graph $\mathcal{G}$ defined in Definition 3.5(c). Algorithm 3.2 is based on the breadth-first search and can also be implemented in the depth-first search. With the matrix $D$, we obtain an approximate (generalized) Cholesky factorization $A \approx GG^\top$, where $G$ has both positive and negative diagonal entries.

**3.2.2. General SDD matrix.** We consider solving $Ax = b$, where $A_p \neq \mathbf{0}$ and $A$ is not a bipartite SDD matrix ($A$ is nonsingular according to Remark 3.7). Our goal is to find $x$ such that the relative residual is smaller than a prescribed tolerance $\epsilon$, i.e.,

$$\|b - Ax\|/\|b\| < \epsilon, \tag{3.7}$$

**Algorithm 3.2** Check bipartite SDD matrix.

**Input:** irreducible SDD matrix $A \in \mathbb{R}^{N\times N}$ (not necessarily bipartite)
**Output:** flag `BSDD_or_not` and diagonal matrix $D \in \mathbb{R}^{N\times N}$ (if $A$ is bipartite)

```
1: Let BSDD_or_not = true and d_11 = 1.
2: Mark index 1 as visited; and queue.push(1).
3: while queue is not empty do
4:     i = queue.pop()
5:     for k : a_ik ≠ 0, k ≠ i do
6:         if index k has not been visited then
7:             if a_ik < 0 then
8:                 Let d_kk = d_ii.
9:             else
10:                Let d_kk = −d_ii.
11:            end if
12:            Mark index k as visited; and queue.push(k).
13:        else
14:            if a_ik d_kk d_ii > 0 then                    // see lines 7–11
15:                Let BSDD_or_not = false and return .
16:            end if
17:        end if
18:    end for
19: end while
```

**Algorithm 3.3** Randomized Cholesky factorization for bipartite SDD matrix.

**Input:** irreducible bipartite SDD matrix $A$
**Output:** lower triangular matrix $G$

```
1: D = CheckBipartiteSDDMatrix(A)
2: G̃ = RandomizedCholesky(DAD)          // Algorithm 2.2 or Algorithm 3.1
3: G = DG̃                                // A ≈ DG̃G̃^⊤D
```

a common stopping criteria for iterative solvers such as PCG. Our approach is to solve the extended system $\tilde{A}y = \tilde{b}$ as initially proposed in [15], where

$$\tilde{A} \triangleq \begin{pmatrix} A_d + A_n & -A_p \\ -A_p & A_d + A_n \end{pmatrix}, \quad \tilde{b} \triangleq \begin{pmatrix} b \\ -b \end{pmatrix}, \tag{3.8}$$

and we seek to find $y$ satisfying

$$\|\tilde{b} - \tilde{A}y\|/\|\tilde{b}\| < \epsilon. \tag{3.9}$$

Before discussing how to solve the extended system, we state our main result in the following theorem.

THEOREM 3.8. *Given* $y = \left(\begin{smallmatrix} y_1 \\ -y_2 \end{smallmatrix}\right)$ *such that* (3.9) *holds, where* $y_1, y_2 \in \mathbb{R}^N$, *the vector*

$$x = \frac{y_1 + y_2}{2} \tag{3.10}$$

*satisfies* (3.7).

*Proof.* According to (3.9), we have

$$\begin{aligned}\|\tilde{b}-\tilde{A}y\|^2 &= \left\| \begin{pmatrix} b-(A_d+A_n)y_1-A_py_2 \\ b-A_py_1-(A_d+A_n)y_2 \end{pmatrix} \right\|^2 \\ &= \|b-(A_d+A_n)y_1-A_py_2\|^2+\|b-A_py_1-(A_d+A_n)y_2\|^2 \\ &< \epsilon^2\|\tilde{b}\|^2,\end{aligned}$$

where $\|\tilde{b}\|^2 = 2\|b\|^2$. We obtain (3.7) as follows:

$$\begin{aligned}\|b-Ax\|^2 &= \frac{1}{4}\|2b-(A_d+A_n+A_p)(y_1+y_2)\|^2 \\ &= \frac{1}{4}\|\left(b-(A_d+A_n)y_1-A_py_2\right)+\left(b-A_py_1-(A_d+A_n)y_2\right)\|^2 \\ &\le \frac{1}{2}\|\left(b-(A_d+A_n)y_1-A_py_2\right)\|^2+\frac{1}{2}\|\left(b-A_py_1-(A_d+A_n)y_2\right)\|^2 \\ &< \epsilon^2\|b\|^2.\end{aligned}$$

∎

A similar result on the relative errors also holds [40],

$$\left\|y-\tilde{A}^\dagger\tilde{b}\right\| \le \epsilon\left\|\tilde{A}^\dagger\tilde{b}\right\| \quad \text{implies} \quad \left\|x-A^{-1}b\right\| \le \epsilon\left\|A^{-1}b\right\|,$$

where $\tilde{A}^\dagger$ denotes the pseudoinverse of $\tilde{A}$. ($\tilde{A}$ may be singular, i.e., a Laplacian matrix.) In addition, if we seek for the exact solution, i.e., $\epsilon = 0$, then (3.10) is indeed the solution of $Ax = b$ [31, 40].

Next, we focus on solving the extended system $\tilde{A}y = \tilde{b}$. It is easy to see that $\tilde{A}$ is an SDD matrix with nonpositive off-diagonal entries, i.e., a Laplacian matrix or an SDDM matrix. In addition, $\tilde{A}$ is irreducible as the following theorem states (proof is in Appendix C).

THEOREM 3.9. *If an irreducible SDD matrix $A$ contains positive off-diagonal entries ($A_p \neq \mathbf{0}$) and is not a bipartite SDD matrix, then the matrix $\tilde{A}$ defined in* (3.8) *is irreducible.*

Therefore, we can construct an approximate Cholesky factorization of $\tilde{A}$, solve the extended system with PCG, and obtain $x$ according to Theorem 3.8. To summarize, Algorithm 3.4 shows the pseudocode of solving a general irreducible SDD linear system.

**4. Sparse matrix reordering and parallel algorithm.** In this section, we discuss two techniques for improving the practical performance of Algorithm 2.2 including reordering the input sparse matrix and parallelizing the computation.

Sparse matrix reordering is a mature technique that is used in sparse direct solvers to speed up factorization and to reduce the memory footprint. Since Algorithm 2.2 keeps a subset of fill-in at every step, it is intuitive that Algorithm 2.2 can also benefit from an appropriate ordering. The challenge, however, is that the fill-in pattern as a result of the random sampling algorithm is not deterministic and thus is impossible to predict beforehand. We resort to using the AMD ordering [2], a fill-in reducing heuristic for the (exact) Cholesky factorization. The advantage is that the AMD can be precomputed quickly and applied to the input sparse matrix before Algorithm 2.2. In practice, we find the AMD working well with `rchol`, although the fill-in behavior

**Algorithm 3.4** General SDD linear solver.

**Input:** irreducible SDD matrix $A \in \mathbb{R}^{N\times N}$, right-hand side $b \in \mathbb{R}^N$, and tolerance $\epsilon$
**Output:** $x \in \mathbb{R}^N$ satisfying (3.7).

1: Construct $\tilde{A}$ and $\tilde{b}$ as defined in (3.8).
2: Compute

$$\tilde{G} = \textsc{RandomizedCholesky}(\tilde{A}).$$

// Algorithm 2.2 or Algorithm 3.1

3: Compute

$$\begin{pmatrix} x_1 \\ -x_2 \end{pmatrix} = \mathrm{PCG}(\tilde{A}, \tilde{b}, \epsilon, \tilde{G}, \tilde{G}^\top), \quad x_1, x_2 \in \mathbb{R}^N.$$

// PCG with preconditioner $\tilde{G}\tilde{G}^\top$

4: **return** $x = (x_1 + x_2)/2$.

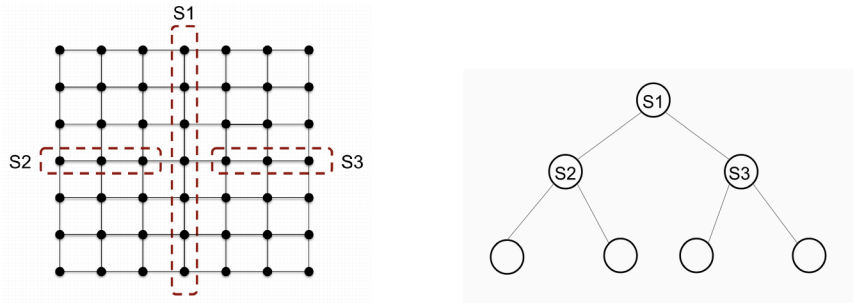


FIG. 3. *(Left): An example of the graph of A and its nested-dissection partitioning. S*1 *is the top separator, S*2 *and S*3 *are two decoupled separators at the second level, and the remaining four parts are decoupled from each other. (Right): Nested-dissection tree and task graph. Task dependency: Every node depends on its children (if they exist) and some descendants, and nodes at the same level can execute in parallel.*

of Algorithm 2.2 is quite different from that of the (exact) Cholesky factorization. We present comparisons between the AMD and other popular reordering strategies used in sparse direct solvers in section 5.1.

Next, we introduce a parallel algorithm for Algorithm 2.2 based on the nested-dissection scheme [12]. Consider the underlying graph associated with a given sparse matrix. If we split it into two disconnected components separated by a vertex separator, then we can apply Algorithm 2.2 on the two disconnected pieces using two threads in parallel. When more than two threads are available, we apply the same partitioning recursively on the two independent partitions to obtain more disconnected parts of the graph; see Figure 3 (left) for a pictorial illustration. Technically, the above procedure is known as the nested dissection and can be computed algebraically using METIS/ParMETIS [19, 20]. Moreover, we employ the AMD ordering within each independent region at the leaf level. The pseudocode of our ordering strategy is shown in Algorithm 4.1, which can be parallelized in a straightforward way.

The nested-dissection partitioning is naturally associated with a tree structure, where leaf nodes correspond to disconnected regions and the other nodes correspond to separators at different levels; see Figure 3 (right). This tree maps to the task graph of a parallel algorithm: Every tree node/task stands for applying Algorithm 2.2 to associated rows/columns in the sparse matrix. It is obvious that tasks at the same level

**Algorithm 4.1** Compute ordering.

**Input:** irreducible Laplacian matrix $L \in \mathbb{R}^{N\times N}$ and number of threads $p$
**Output:** the nested-dissection tree $\mathcal{T}$

```
1: ℓ = log2(p)                                          // assume p is a power of 2
2: Create a full binary tree T of ℓ levels              // initialize output
3: ComputeOrdering(T →root, L, ℓ)                       // start recursion

4: function ComputeOrdering(node, L, ℓ)
5:     if ℓ > 0 then
           // partition graph/indices into "left," "right," and "seperator"
6:         I_l, I_r, I_s = PartitionGraph(L)            // call METIS
7:         node→store_indices(I_s)
8:         ComputeOrdering(node→left, L(I_l, I_l), ℓ − 1)
9:         ComputeOrdering(node→right, L(I_r, I_r), ℓ − 1)
10:    else
11:        I = ComputeAMD(L)                            // AMD ordering at leaf level
12:        node→store_indices(I)
13:    end if
14: end function
```

can execute in parallel. Notice a task depends on not only its children but also some of their descendants. We employ a multifrontal type of approach [28] in our parallel algorithm, where a task receives the Schur-complement updates from its two children and sends necessary updates to its parent. In other words, a task communicates with only its children and parent. The pseudocode is shown in Algorithm 4.2, where we traverse the task tree in postorder to generate all tasks.

We have implemented Algorithm 4.2 with both OpenMP[4] tasks and the C++ thread library,[5] and we found the latter delivered slightly better performance in our numerical tests. Specifically, we use `std::async` to launch an asynchronous task at line 4 on a new thread and store the results in an `std::future` object. Synchronization is achieved by calling the `get()` method on the previous `future` object at line 7. One advantage of our approach is that we are able to pin threads on cores for locality via `sched_setaffinity()` in `sched.h`.

**5. Numerical results.** In this section, we refer to our randomized preconditioner as `rchol`. Recall our goal is solving $Ax = b$, and our approach is constructing a preconditioner $GG^\top$, where $G$ is a lower triangular matrix.

Besides problems from the SuiteSparse Matrix Collection, we generate test matrices from discretizing Poisson's equation, variable-coefficient Poisson's equation, and anisotropic Poisson's equation:

$$-\nabla\cdot(a(x)\,\nabla u(x)) = f, \quad x \in \Omega = [0,1]^3, \quad u(x) = 0 \text{ on } \partial\Omega. \tag{5.1}$$

- Poisson's equation: $a(x) = 1$.
- Variable-coefficient Poisson's (VC-Poisson) equation: We generate a high-contrast coefficient field $a(x)$ following [5, 6, 16]. First, we generate $\{a_i\}$ from

[4] https://www.openmp.org/.
[5] https://en.cppreference.com/w/cpp/thread.

**Algorithm 4.2** Parallel randomized Cholesky factorization.

**Input:** irreducible Laplacian matrix $L \in \mathbb{R}^{N\times N}$ and the nested-dissection tree $\mathcal{T}$
**Output:** matrix $G \in \mathbb{R}^{N\times N}$ (lower triangular if reordered according to $\mathcal{T}$)

ParRchol($\mathcal{T}$ →root, $L$, $G$) // start recursion; $L$ and $G$ modified in place

**function** ParRchol(node, $L$, $G$) // postorder tree traversal
  **if** node→not_leaf() **then**
    // recursive task generation
    $S_l$ = ParRchol(node→left, $L$, $G$)
    $S_r$ = ParRchol(node→right, $L$, $G$)
  **end if**
  // wait until child tasks finish
  $L = L + S_l + S_r$ // merge updates from children (reduction)
  $\mathcal{I}$ = node→get_indices()
  $S$ = RcholBlock($\mathcal{I}$, $L$, $G$) // apply `rchol` to a block of indices
  **return** $S$
**end function**

**function** RcholBlock($\mathcal{I}$, $L$, $G$)
  $S = \mathbf{0}_{N\times N}$
  **for** $k \in \mathcal{I}$ **do**
    // $\ell_{kk} = 0$ at the last index in the top separator according to Corollary 2.8
    $G(:,k) = \begin{cases} L(:,k)/\sqrt{\ell_{kk}} & \ell_{kk} \neq 0 \\ \mathbf{0} & \ell_{kk} = 0 \end{cases}$
    $C$ = SampleClique($L, k$)
    $C_1, C_2$ = SeparateEdges($\mathcal{I}, C$) // $C_1 + C_2 = C$
    $L = L - L^{(k)} + C_1$
    $S = S + C_2$ // cumulate updates and send to parent
  **end for**
  **return** $S$
**end function**

**function** SeparateEdges($\mathcal{I}$, $C$)
  $C_1 = \mathbf{0}_{N\times N}, C_2 = \mathbf{0}_{N\times N}$
  // suppose $C = \sum_{e_{ij}\in E} w_{ij}\, \boldsymbol{b}_{ij}\boldsymbol{b}_{ij}^\top$ since $C$ is a graph Laplacian
  **for** $e_{ij} \in E$ **do**
    **if** $i \in \mathcal{I}$ **or** $j \in \mathcal{I}$ **then**
      $C_1 = C_1 + w_{ij}\, \boldsymbol{b}_{ij}\boldsymbol{b}_{ij}^\top$ // needed by the current node
    **else**
      $C_2 = C_2 + w_{ij}\, \boldsymbol{b}_{ij}\boldsymbol{b}_{ij}^\top$ // needed by ancestors
    **end if**
  **end for**
  **return** $C_1, C_2$
**end function**

standard uniform distribution on a regular grid and compute the median $\mu$. Then we convolve $\{a_i\}$ with an isotropic Gaussian of width $4h$, where $h$ is the grid spacing. Finally, we quantize $\{a_i\}$ by setting

$$a_i = \begin{cases} \rho^{1/2} & \text{if } a_i \geq \mu, \\ \rho^{-1/2} & \text{if } a_i < \mu. \end{cases} \tag{5.2}$$

See Appendix D for an example of the random coefficients.

- Anisotropic Poisson's (Aniso-Poisson) equation: $a(x) = \mathrm{diag}(\delta^{1/2}, 1, \delta^{-1/2})$, where the coefficients are constant along each dimension.

In particular, we discretize the above elliptic PDE using the standard 7-point finite difference stencil over a uniform $n \times n \times n$ grid. Let $h = 1/n$, $x_j = h(j_1, j_2, j_3)$, where $j$ is the index of the triplet $(j_1, j_2, j_3)$ for $1 \leq j_1, j_2, j_3 \leq n$. The discretized PDE reads

$$\begin{aligned}(a_{j-e_1/2} + a_{j+e_1/2} + a_{j-e_2/2} + a_{j+e_2/2} + a_{j-e_3/2} + a_{j+e_3/2})u_j \\ - a_{j-e_1/2}u_{j-e_1} + a_{j+e_1/2}u_{j+e_1} - a_{j-e_2/2}u_{j-e_2} + a_{j+e_2/2}u_{j+e_2} \\ - a_{j-e_3/2}u_{j-e_3} + a_{j+e_3/2}u_{j+e_3} = h^2 f_j,\end{aligned}$$

where $e_1 = (1,0,0), e_2 = (0,1,0), e_3 = (0,0,1)$, and $u_j \approx u(x_j)$ is to be solved.

Experiments were performed on a node from Frontera.[6] Results in subsections 5.1 and 5.2 were obtained using a single thread on an Intel Xeon Platinum 8280 (Cascade Lake), and results in subsection 5.4 were obtained using multiple threads/cores on an Intel Xeon Platinum 8280M. Below are the notations we use to report results (all timing results are in seconds):

- $N$: matrix size of $A$.
- $p$: number of threads/cores.
- nnz: number of nonzeros in $A$.
- fill: *twice* the number of nonzeros in $G$.
- $t_p$: time for computing a permutation/reordering for $A$.
- $t_f$: time for computing the factorization/preconditioner.
- $t_s$: *total PCG time* for solving a standard-uniform random $b$.
- $n_{it}$: number of the PCG iterations with tolerance 1e$-$10. In cases where PCG stagnated before convergence, we report the iteration number to stagnation and the corresponding relative residual (relres) $\|b - Ax\|_2/\|b\|_2$.

**5.1. Reordering and stability.** We present results for five commonly used reordering strategies in Table 1. The test problem is the standard 7-point finite-difference discretization of Poisson's equation in a unit cube with the Dirichlet boundary condition. We have also tested the five strategies on other problems including VC-Poisson, Aniso-Poisson, and problems from the SuiteSparse Matrix Collection (see section 5.2.1), and the following observations generally apply:

1. Natural ordering (a.k.a., lexicographic ordering)/no reordering leads to significant amount of fill-in. Although PCG required a small number of iterations, the total solve time is significant with a relatively dense preconditioner.
2. Reverse Cuthill–McKee ordering aims at a small bandwidth of the reordered matrix, which helps reduce fill-in for some applications. But results showed that it is was not effective for `rchol`.

[6] https://frontera-portal.tacc.utexas.edu/user-guide/.

TABLE 1

*Sparse matrix reordering. The matrix is from discretizing Poisson's equation on a 3D regular grid of size $256^3$ using the standard 7-point finite difference. The orderings are computed using MATLAB commands in parentheses.*

| Ordering | fill/nnz | $t_p$ | $t_f$ | $t_s$ | $n_{it}$ |
|---|---|---|---|---|---|
| no reordering | 10.2 | 0 | 139 | 173 | 39 |
| reverse Cuthill–McKee (symrcm) | 7.9 | 5 | 97 | 138 | 41 |
| random ordering (randperm) | 3.3 | 0.8 | 76 | 362 | 55 |
| nested dissection (dissect) | 3.3 | 206 | 66 | 132 | 65 |
| approximate minimum degree (amd) | 3.5 | 38 | 50 | 126 | 60 |

(a) AMD reordering: $2.1\mathrm{e}+8$ nonzeros. (b) random reordering: 1.9e+8 nonzeros.

FIG. 4. *Sparsity pattern of triangular factors computed by* `rchol` *corresponding to the AMD ordering and the random ordering in Table* 1*, respectively. (The full spy plot for random ordering is quite large, and* (b) *corresponds to the leading principle submatrix of size* $3\mathrm{e}+5$*.)*

TABLE 2

*Variance of* `rchol` *(minimums and maximums among* 10 *independent trials). The matrices are from discretizing Poisson, VC-Poisson ($\rho = 1\mathrm{e}+5$), and Aniso-Poisson ($\delta = 1\mathrm{e}+4$) on a 3D regular grid of size $256^3$ using the standard 7-point finite difference. (PCG tolerance is $1\mathrm{e}-6$ for VC-Poisson, a highly ill-conditioned problem; see section* 5.2.2*.)*

| Ordering | fill/nnz | $t_f$ | $t_s$ | $n_{it}$ |
|---|---|---|---|---|
| Poisson | 3.538 - 3.542 | 48 - 54 | 117 - 128 | 57 - 62 |
| VC-Poisson | 4.074 - 4.078 | 56 - 65 | 257 - 303 | 120 - 141 |
| Aniso-Poisson | 2.556 - 2.557 | 38 - 43 | 79 - 80 | 44 - 44 |

3. Random ordering as suggested in [25] is effective in fill-in reduction. However, it results in widely scattered sparsity pattern in the triangular factor as shown in Figure 4, hampering practical performance of triangular solves at every iteration.
4. Nested-dissection ordering is effective in fill-in reduction but requires significant time to compute.
5. AMD ordering [2] is also effective in fill-in reduction and can be computed quickly. The fill-in pattern of `rchol` is not deterministic and is different from the (exact) Cholesky factorization. Although the AMD is designed as a greedy strategy for minimizing the fill-in of the (exact) Cholesky factorization, it also performs well when used with `rchol`. Among the five reordering strategies considered here, the AMD leads to the minimum running time consistently for all of our test problems, so we use the AMD by default.

Although `rchol` uses randomness in the algorithm, the resulting preconditioner delivers extremely consistent performance as Table 2 shows.

**5.2. Comparison with incomplete Cholesky.** We compare `rchol` to the incomplete Cholesky preconditioner with thresholding dropping (`ichol`) in MATLAB

TABLE 3

*SPD matrices from the SuiteSparse Matrix Collection. With* no preconditioner*, CG converged extremely slow, and the relative residuals were still quite large after* $2,500$ *iterations except for the second problem.*

| | Name | $N$ | nnz | Property | $n_{it}$ | relres |
|---|---|---|---|---|---|---|
| # 1 | ecology2 | 1.0e+5 | 5.0e+6 | SDDM | 2500 | 1e-01 |
| # 2 | parabolic_fem | 5.3e+5 | 3.7e+6 | SDD | 2500 | 2e-07 |
| # 3 | apache2 | 7.2e+5 | 4.8e+6 | not SDD | 2500 | 1e-02 |
| # 4 | G3_circuit | 1.6e+6 | 7.7e+6 | not SDD | 2500 | 5e-01 |

TABLE 4

*Comparison between* `rchol` *preconditioner and* `ichol` *preconditioner on matrices from the SuiteSparse Matrix Collection. AMD ordering is applied with* `rchol`*. Based on our experiments, the vanilla* `ichol` *preconditioner without any reordering performs slightly better than with a reordering.*

| | `rchol` | | | | | | `ichol` | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | fill/nnz | $t_p$ | $t_f$ | $t_s$ | $n_{it}$ | relres | fill/nnz | $t_f$ | $t_s$ | $n_{it}$ | relres |
| # 1 | 2.41 | 0.4 | 1.4 | 6.3 | 89 | 1e-08 | 2.72 | 0.2 | 68 | 798 | 3e-08 |
| # 2 | 2.27 | 0.4 | 1.0 | 2.8 | 65 | 8e-11 | 2.29 | 0.2 | 15 | 411 | 2e-10 |
| # 3 | 2.93 | 0.6 | 1.5 | 4.1 | 63 | 3e-10 | 2.96 | 0.2 | 18 | 322 | 4e-10 |
| # 4 | 2.68 | 1.5 | 2.8 | 9.6 | 90 | 9e-11 | 2.75 | 0.3 | 40 | 379 | 2e-10 |

R2020a. In particular, we manually tuned the drop tolerance in `ichol` to obtain preconditioners with slightly more fill-in. For both preconditioners, the construction time is usually much smaller than the time spent in PCG. For every PCG iteration, we expect similar running time because both preconditioners have approximately the same amount of fill-in. Therefore, the performance depends mostly on the numbers of PCG iterations. We used the AMD ordering in `rchol`. Based on our experiments, `ichol` performed better without any reordering, which is consistent with empirical results observed in the literature [10].

**5.2.1. Matrices from the SuiteSparse Matrix Collection.** We first compare `rchol` with `ichol` on four SPD matrices from the SuiteSparse Matrix Collection[7] that are not necessarily SDD. The first is an SDDM matrix, the second is an SDD matrix, and the last two are SPD (but not SDD) matrices. All matrices have only negative off-diagonal entries except for the second matrix. The second matrix is SDD, but approximately a third of the off-diagonal entries are as small as $3.2\mathrm{e}-7$. Since these entries are quite small relative to the remaining entries, we simply ignored these positives when applying `rchol`. The last two matrices are not SDD, and some of the diagonals are smaller than the sum of the absolute value of off-diagonals. But we were able to run `rchol` in a "black-box" fashion, which is equivalent to adding diagonal compensations to make the original matrix SDD.

Without any preconditioner, CG converged extremely slowly as shown in Table 3. As Table 4 shows, although the highly optimized `ichol` (in MATLAB) delivers faster factorization than our implementation of `rchol`, the `rchol`-PCG took much less time than the `ichol`-PCG due to significantly fewer iterations. In particular, PCG took about $9\times$ more iterations with `ichol` for "ecology2." For all cases with `ichol`, PCG stagnated before the $1\mathrm{e}-10$ tolerance was reached. With `rchol`, the relative residuals decreased to below $1\mathrm{e}-10$ for the second and the last problems. We also tested `ichol` with no fill-in, and the total times were greater than those in Table 4.

[7] https://sparse.tamu.edu.

TABLE 5

*Comparison between* `rchol` *preconditioner and* `ichol` *preconditioner on matrices from discretizing variable-coefficient Poisson's equation on a regular grid of size* $128^3$ *using the standard 7-point finite difference* ($N = 2.0\text{e}+6, nnz = 1.4\text{e}+7$). *The coefficients have contrast ratio* $\rho$*; see* (5.2). *When* $\rho \geq 1\text{e}+3$, *PCG stagnated before reaching tolerance* $1\text{e}-10$.

| $\rho$ | `rchol` | | | | | `ichol` | | | |
|---|---|---|---|---|---|---|---|---|---|
| | fill/nnz | $t_p$ | $t_f$ | $t_s$ | $n_{it}$ | fill/nnz | $t_f$ | $t_s$ | $n_{it}$ |
| 1e+0 | 3.23 | 3.8 | 5.3 | 12 | 51 | 3.40 | 0.7 | 21 | 102 |
| 1e+1 | 3.42 | 3.8 | 5.6 | 13 | 53 | 3.46 | 0.8 | 37 | 175 |
| 1e+2 | 3.57 | 3.8 | 5.7 | 19 | 83 | 3.63 | 0.8 | 50 | 235 |
| 1e+3 | 3.62 | 3.8 | 5.7 | 28 | 115 | 3.72 | 0.9 | 57 | 260 |
| 1e+4 | 3.62 | 3.9 | 5.7 | 29 | 126 | 3.78 | 0.9 | 57 | 254 |
| 1e+5 | 3.62 | 3.9 | 5.8 | 32 | 144 | 3.78 | 0.9 | 63 | 272 |

**5.2.2. Variable-coefficient Poisson's equation.** We compare the `rchol` preconditioner with the `ichol` preconditioner on a sequence of SDDM matrices that become gradually more ill-conditioned. The discretization of VC-Poisson on a regular grid using the standard 7-point finite-difference stencil has a condition number $\mathcal{O}(\rho N^{2/3})$.

The results are similar to above, where `ichol` required at least twice as many iterations. As a result, the total time taken with the `rchol` preconditioner is much less than with the `ichol` preconditioner in all cases. In Table 5, when the condition number is large, PCG stopped progressing before reaching the tolerance $1\text{e}-10$. Consequently, the relative residual with the solution returned from PCG decreased from approximately $1\text{e}-11$ to approximately $1\text{e}-8$ as $\rho$ increases from 1 to $1\text{e}+5$. Both preconditioners suffer from this performance deterioration.

**5.3. Comparison to multigrid methods.** We compared `rchol` to three multigrid methods including the combinatorial multigrid (CMG) [24],[8] the Ruge–Stuben (classical) AMG (RS-AMG), and the smoothed aggregation AMG (SA-AMG). The RS-AMG and the SA-AMG are from the pyamg package [33].[9] We ran `rchol` through the C++ interface.

The test matrices include the four problems from the SuiteSparse Matrix Collection (see subsection 5.2.1) and three matrices of size $128^3$ from discretizing the three Poisson equations, respectively. The results of comparison are shown in Table 6, which shows that our method is the fastest for two of the problems, CMG is the fastest for one problem, and the classical AMG is the fastest for the other four problems.

As is well accepted by the scientific computing community, the performance of linear solvers may depend on the input matrices, and there is no single best solver for all problems. As a result, there exist different solvers/preconditioners, including incomplete factorizations, multigrid, sparse direct solvers, etc. As Table 6 shows, multigrid methods usually perform well on matrices corresponding to regular grids.

**5.4. Parallel scalability.** In this section, we show the speedup of running `rchol` with multiple threads and the stability of the resulting preconditioner in terms of the fill-in ratio and the PCG iteration. The test problem is solving the 3D Poisson's equation with the Dirichlet boundary condition in the unit cube, which is discretized using the 7-point stencil on regular grids. We ran `rchol` in single-precision floating-point arithmetic to reduce memory footprint and computation time, and we ran PCG

[8] http://www.cs.cmu.edu/~jkoutis/cmg.html.

[9] https://github.com/pyamg/pyamg.

TABLE 6

*Comparison to multigrid methods. Highlighted rows are the fastest solve time among all methods. See section 5.2.1 for the first four matrices. The remaining three matrices are discretizations of Poisson, VC-Poisson ($\rho = 1e+5$), and Aniso-Poisson ($\delta = 1e+4$) equations on a 3D regular grid of size $128 \times 128 \times 128$.*

| matrix | rchol | | | CMG | | |
|---|---|---|---|---|---|---|
| | $t_f$ | $t_s$ | $n_{it}$ | $t_f$ | $t_s$ | $n_{it}$ |
| ecology2 | 0.9 | 4.63 | 90 | 1.0 | 4.27 | 58 |
| parabolic_fem | 0.9 | 2.08 | 67 | 2.59 | 3.20 | 45 |
| apache2 | 1.4 | 2.91 | 64 | - | - | - |
| G3_circuit | 2.5 | 7.96 | 90 | 5.67 | 9.59 | 73 |
| Poisson | 6.1 | 8.07 | 53 | 7.51 | 7.60 | 43 |
| VC-Poisson | 6.6 | 20.7 | 131 | 9.25 | 10.88 | 62 |
| Aniso-Poisson | 3.71 | 4.88 | 36 | 6.20 | 8.90 | 67 |

| matrix | RS-AMG | | | SA-AMG | | |
|---|---|---|---|---|---|---|
| | $t_f$ | $t_s$ | $n_{it}$ | $t_f$ | $t_s$ | $n_{it}$ |
| ecology2 | 1.44 | 3.00 | 21 | 3.30 | 2.54 | 19 |
| parabolic_fem | 1.08 | 1.15 | 14 | 1.42 | 1.48 | 27 |
| apache2 | 1.17 | 13.38 | 101 | 2.91 | 6.68 | 49 |
| G3_circuit | 2.38 | 10.82 | 39 | 7.29 | 23.43 | 67 |
| Poisson | 6.15 | 5.34 | 13 | 10.20 | 7.80 | 17 |
| VC-Poisson | 6.55 | 15.68 | 38 | 9.74 | 14.20 | 32 |
| Aniso-Poisson | 3.34 | 4.09 | 9 | 9.46 | 44.14 | 101 |

in double precision. The use of single precision in the construction of preconditioners has been studied in the literature [1,13,27,30], which may lead to an increase of PCG iterations for difficult problems. Here, our results show that the use of single precision in `rchol` does not impact the number of PCG iterations for solving the discretized Poisson's equation.

With $p = 1$ thread, we used the AMD reordering; otherwise, when $p > 1$, we used a $\log_2 p$-level nested-dissection ordering combined with the AMD ordering at the leaf level. All experiments were performed on an Intel Xeon Platinum 8280M (Cascade Lake), which has 112 cores on four sockets (28 cores/socket), and every thread is bound to a different core in a scattered fashion (e.g., the first four threads are each bound to one of the four sockets). We used the scalable memory allocator in the Intel TBB library.[10]

Table 7 shows the results of three increasing problem sizes—the largest one being *one billion* unknowns and the factorization time scaled up to 64 threads in each case. (Results of parallel sparse triangular solves are given in Appendix E.) For $N = 1024^3$, the sequential factorization took nearly 42 minutes, while it took approximately 3 minutes using 64 threads (cores), a 13.7× speedup. Table 7 also shows that the fill-in ratio and the PCG iteration are extremely stable regardless of the number of threads used. For the three problems, the memory footprints of the preconditioners are about 1.7 GB, 15 GB, and 130 GB, respectively, in single precision, where we stored only a triangular factor for every symmetric preconditioner.

Figure 5 shows the time spent on leaf tasks and separator tasks in strong- and weak-scaling experiments, respectively; recall the task graph in Figure 3. When $p$ doubles in strong scaling, the task tree increases by one level; in other words, every

[10] https://software.intel.com/content/www/us/en/develop/documentation/tbb-documentation/top/intel-threading-building-blocks-developer-guide/package-contents/scalable-memory-allocator.html.

TABLE 7

*Parallel scalability on an Intel Cascade Lake that has* 112 *cores on four sockets. We applied* `rchol` *to solving the* 3*D Poisson's equation (discretized with the* 7*-point stencil on regular grids). We used single-precision floating-point arithmetic in* `rchol`.

| $p$ | $N = 256^3$ | | | $N = 512^3$ | | | $N = 1024^3$ | | |
|---|---|---|---|---|---|---|---|---|---|
| | fill/nnz | $t_f$ | $n_{it}$ | fill/nnz | $t_f$ | $n_{it}$ | fill/nnz | $t_f$ | $n_{it}$ |
| 1 | 3.56 | 19.9 | 57 | 3.93 | 226 | 65 | 4.31 | 2523 | 78 |
| 2 | 3.60 | 10.7 | 59 | 3.98 | 113 | 68 | 4.37 | 1279 | 79 |
| 4 | 3.61 | 5.7 | 57 | 3.98 | 58 | 65 | 4.39 | 664 | 75 |
| 8 | 3.63 | 3.3 | 61 | 3.99 | 35 | 65 | 4.38 | 388 | 75 |
| 16 | 3.66 | 2.3 | 59 | 4.00 | 23 | 65 | 4.38 | 258 | 76 |
| 32 | 3.66 | 1.9 | 57 | 4.02 | 18 | 64 | 4.39 | 197 | 71 |
| 64 | 3.66 | 1.7 | 57 | 4.02 | 16 | 67 | 4.38 | 184 | 75 |

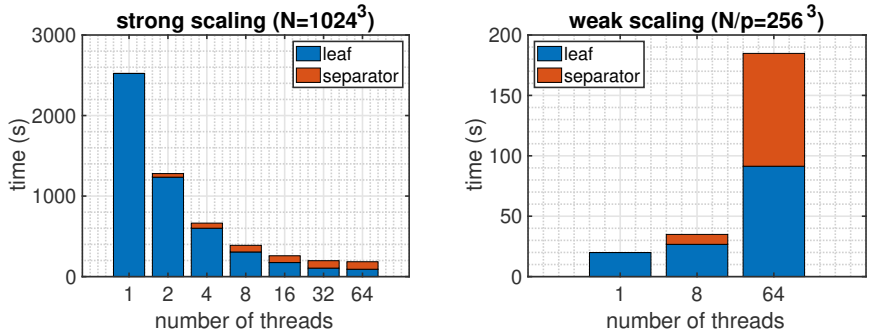


FIG. 5. *Strong and weak scalability of the* `rchol` *factorization/construction time on an Intel Cascade Lake. The input matrices are discretization of the* 3*D Poisson's equation using the* 7*-point stencil on regular grids. We used single-precision floating-point arithmetic in* `rchol`. *"leaf" denotes the maximum time of all leaf tasks executing in parallel, and "separator" denotes the remaining time spent on all separators. (Recall the task graph in Figure* 3*.)*

leaf task is decomposed into two smaller leaf tasks plus a separator task. In addition, this decomposition computed algebraically by graph partitioning can hardly avoid load imbalance. Therefore, the time reduction shrinks as $p$ increases in strong scaling. When $p$ increases by $8\times$ in weak scaling, the task tree increases by three levels, while the problem size associated with every leaf task remains the same if the partitioning is ideally uniform. In reality, however, load imbalance among leaf tasks becomes more and more significant as $p$ increases. The other reason for the increasing maximum running time of leaf tasks is that these tasks are memory-bound and suffer from memory bandwidth saturation if $p$ is large. The other bottleneck in weak scaling comes from the three extra levels of separator tasks when $p$ increases by $8\times$. Indeed, the top separator has size $\mathcal{O}(N^{2/3})$, but the corresponding task runs in sequential in our parallel algorithm. Parallelizing such tasks for separators at top levels is left as future work.

Table 8 shows the effectiveness of the `rchol` preconditioner computed with multiple threads, where the PCG iteration increases logarithmically with respect to the problem size $N$. By contrast, the PCG iteration with the `ichol` preconditioner increases by approximately $2\times$ when the problem size $N$ increases by $8\times$ (the mesh is refined by $2\times$ in every dimension).

TABLE 8

*Comparison of PCG iterations for solving the 3D Poisson equation discretized with the 7-point stencil on regular grids. We did not run* `ichol` *for* $N = 1024^3$ *limited by our computation budget. (We manually tuned the drop tolerance in* `ichol` *to obtain preconditioners with slightly more fill-in. See Table 7 for the fill-in of* `rchol` *preconditioners.)*

| $N$ | $128^3$ | $256^3$ | $512^3$ | $1024^3$ |
|---|---|---|---|---|
| `ichol` | 100 | 185 | 341 | - |
| `rchol` | 50 | 57 | 67 | 75 |

**6. Conclusions and generalizations.** In this paper, we have introduced a preconditioner named `rchol` for solving SDD linear systems. To that end, we construct a closely related Laplacian linear system and apply the randomized Cholesky factorization. Two essential ingredients for achieving practical performance include a heuristic for sampling a clique and a fill-reducing reordering before factorization. The resulting sparse factorization is shown to outperform `ichol` when both have roughly the same amount of fill-in. We view `rchol` as a variant of standard incomplete Cholesky factorization. But unlike classical threshold-based dropping and level-based dropping, the sampling scheme in `rchol` is an unbiased estimator: It randomly selects a subset of a clique and assigns them new weights. Interestingly, fill-reducing orderings are critical for the practical performance of `rchol` but are generally not effective for `ichol`. In addition, the nested-dissection decomposition used in our parallel algorithm does not affect the performance of `rchol` but generally degrades the preconditioner quality of `ichol`.

The described algorithm extends to the following two cases. The first is that $A$ is an SPD matrix that has only nonpositive off-diagonals (a.k.a., M-matrix). For such a matrix, there exists a positive diagonal matrix $D$ such that $DAD$ is SDDM [18], and then `rchol` can be applied to $DAD$. The other is that $A$ is the finite-element discretization of (5.1) in a bounded open region with positive conductivity, i.e., $a(x) > 0$. Such a matrix is generally SPD but not necessarily SDD, but there exists an analytical way to construct an SDD matrix whose preconditioner remains effective for $A$ [4].

Three important directions for future research include the following:

- Investigating variants of Algorithm 2.3 to sample more edges in a clique, which leads to approximate Cholesky factorizations with more fill-in than the one computed by `rchol`. Such approximations can potentially be more effective preconditioners for hard problems where the preconditioner based on `rchol` converges slowly.
- Parallelizing tasks for separators, especially for those at top levels. As Figure 5 shows, such tasks become the bottleneck of the parallel factorization time when a large number of threads are used. A naive method is to apply the current parallel algorithm recursively on the (sparse) frontal matrices associated with those top separators.
- Extending the current framework combining Gaussian elimination with random sampling to unsymmetric matrices, which leads to an approximate LU factorization. See [8] for some progress in this direction.

## Appendix A. Proof of Theorem 2.9.

*Proof.* Consider the matrix/graph after an elimination step in Algorithm 2.2; the number of nonzeros/edges decreases by 1. The reason is that at every step $n$ edges

are eliminated and $n-1$ edges are added/sampled, where $n = |\mathcal{N}_k|$ is the number of neighbors or the number of nonzeros in the eliminated row/column excluding the diagonal. Since a random row/column is eliminated at every step, we have

$$\mathbb{E}[n] = \frac{M-k+1}{N-k+1}$$

at the $k$th step. It is obvious to see that the computational cost and storage required by Algorithm 2.3 is $\mathcal{O}(n)$ at every step. Therefore, the expected running time and the expected storage are both bounded by

$$\sum_{k=1}^{N} \frac{M-k+1}{N-k+1} < \sum_{k=1}^{N} \frac{M}{N-k+1} < M \log N.$$

∎

## Appendix B. Proof of equivalence in Definition 3.5.

### B.1. Lemma.

LEMMA B.1. *If matrix $A \in \mathbb{R}^{N\times N}$ is an irreducible SDD matrix, then* $\operatorname{rank}(\hat{A}) \geq N-1$, *where matrix $\hat{A}$ is defined in* (3.6).

*Proof.* Consider the following quadratic form given a nonzero $x \in \mathbb{R}^N$:

$$x^\top \hat{A}\, x = \sum_{i,j} -a_{ij}^n (x_i - x_j)^2 + \sum_{i,j} a_{ij}^p (x_i + x_j)^2 \geq 0,$$

where $a_{ij}^n$ and $a_{ij}^p$ denote negative and positive off-diagonal entries in $A$, respectively. Suppose $x$ lies in the null space of $A$. We know that $x_i = x_j$ corresponding to every $a_{ij}^n$ and $x_i = -x_j$ corresponding to every $a_{ij}^p$. In addition, we know $x$ is entrywise nonzero because $A$ is irreducible (underlying graph is connected). Therefore, we can find at most one such $x$ (up to a scalar multiplication), which implies that $\operatorname{rank}(\hat{A}) \geq N-1$. ∎

### B.2. Formal proof.

*Proof.* Assuming (a) holds, we derive (c). There exists a nonzero $x \in \mathbb{R}^N$ such that $\hat{A}x = 0$. Consider the quadratic form

$$x^\top \hat{A}\, x = \sum_{i,j} -a_{ij}^n (x_i - x_j)^2 + \sum_{i,j} a_{ij}^p (x_i + x_j)^2 = 0,$$

where $a_{ij}^n$ and $a_{ij}^p$ denote negative and positive off-diagonal entries in $A$, respectively. Hence, we know that $x_i = x_j$ corresponding to every $a_{ij}^n$ and $x_i = -x_j$ corresponding to every $a_{ij}^p$. In addition, we know $x$ is entrywise nonzero because $A$ is irreducible (underlying graph is connected). Therefore, $x$ implies that the graph $\mathcal{G}$ is 2-colorable in that all vertices $v_i$ corresponding to $x_i > 0$ have the same color, while all vertices $v_i$ corresponding to $x_i < 0$ have the other color.

Assuming (b) holds, we derive (a) and (c) as follows. Without loss of generality, suppose $D = \operatorname{diag}(\underbrace{1,\ldots,1}_{n_1}, \underbrace{-1,\ldots,-1}_{n_2})$ and the matrix $A$ is partitioned as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

where $A_{11} \in \mathbb{R}^{n_1\times n_1}$ and $A_{22} \in \mathbb{R}^{n_2\times n_2}$. Since $DAD$ has only nonpositive off-diagonal entries, $A_{11}$ and $A_{22}$ have nonpositive off-diagonal entries, while $A_{12}$ and $A_{21}$ have nonnegative entries. Hence, we know the following:

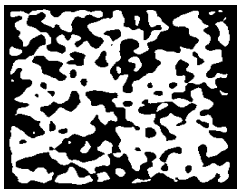

FIG. 6. *Example of the high-contrast coefficients for the variable-coefficient Poisson's equation in the unit square with a* $2D$ *grid* $256 \times 256$.

- The vector $D\mathbf{1}$ is in the null space of $\hat{A}$, which is thus rank deficient. According to Lemma B.1, we know $\operatorname{rank}(\hat{A}) = N - 1$.
- The graph $\mathcal{G}$ is 2-colorable in that $v_1, v_2, \ldots, v_{n_1}$ have the first color and $v_{n_1+1}, v_{n_1+2}, \ldots, v_N$ have the other color.

Assuming (c) holds, we derive (b). Without loss of generality, suppose $v_1, v_2, \ldots, v_{n_1}$ have the same color, which is different from the color that $v_{n_1+1}, v_{n_1+2}, \ldots, v_N$ have. In other words, matrix $A$ can be partitioned into

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

where $A_{11} \in \mathbb{R}^{n_1 \times n_1}$ and $A_{22} \in \mathbb{R}^{n_2 \times n_2}$ have nonpositive off-diagonal entries and $A_{12}$ and $A_{21}$ have nonnegative entries. Therefore, the diagonal rescaling $D$ given by

$$\operatorname{diag}(\underbrace{1, \ldots, 1}_{n_1}, \underbrace{-1, \ldots, -1}_{n_2})$$

satisfies that $DAD$ has only nonpositive off-diagonal entries. ∎

**Appendix C. Proof of Theorem 3.9.**

*Proof.* Without loss of generality, assume $A = \hat{A} \in \mathbb{R}^{N \times N}$; in other words, every diagonal entry is equal to the sum of the absolute value of off-diagonal entries on the same row/column. Suppose there exists a nonzero vector in the null space of $\tilde{A} \in \mathbb{R}^{2N \times 2N}$, i.e.,

$$\begin{pmatrix} A_d + A_n & -A_p \\ -A_p & A_d + A_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = 0,$$

where $x_1, x_2 \in \mathbb{R}^N$. It is easy to see that

$$(A_d + A_n + A_p)(x_1 - x_2) = 0,$$
$$(A_d + A_n - A_p)(x_1 + x_2) = 0.$$

Since $A = A_d + A_n + A_p$ is an irreducible nonbipartite SDD matrix, we know $\operatorname{rank}(\hat{A}) = \operatorname{rank}(A) = N$. Hence, $x_1 = x_2$. It is straightforward to verify that $A_d + A_n - A_p$ is a Laplacian matrix, and thus $x_1 = x_2 \in \operatorname{span}\{\mathbf{1}\}$. Therefore, we know $\operatorname{rank}(\tilde{A}) = 2N - 1$, which implies that Laplacian matrix $\tilde{A}$ is irreducible. ∎

**Appendix D. High-contrast coefficients for VC-Poisson.** One instance of the random coefficients constructed in (5.2) is shown in Figure 6.

**Appendix E. Results of parallel sparse triangular solve.** Table 9 shows parallel timing results of the parallel sparse triangular solve. The Cholesky factor $G$

TABLE 9

*Parallel sparse triangular solve (per iteration) on an Intel Cascade Lake that has* 112 *cores on four sockets. The matrices are from discretizing Poisson's equation on a* 3*D regular grid with the standard* 7*-point stencil.*

| $p$ | $N = 128^3$ | | | $N = 256^3$ | | | $N = 512^3$ | | |
|---|---|---|---|---|---|---|---|---|---|
| | $t_{\text{lower}}$ | $t_{\text{upper}}$ | $n_{it}$ | $t_{\text{lower}}$ | $t_{\text{upper}}$ | $n_{it}$ | $t_{\text{lower}}$ | $t_{\text{upper}}$ | $n_{it}$ |
| 1 | 0.0400 | 0.0430 | 50 | 0.409 | 0.409 | 57 | 5.59 | 4.49 | 64 |
| 2 | 0.0499 | 0.0536 | 50 | 0.333 | 0.348 | 57 | 2.93 | 2.69 | 67 |
| 4 | 0.0423 | 0.0446 | 50 | 0.199 | 0.197 | 58 | 1.51 | 1.31 | 65 |
| 8 | 0.0280 | 0.0301 | 53 | 0.157 | 0.161 | 54 | 0.962 | 0.814 | 64 |
| 16 | 0.0177 | 0.0200 | 49 | 0.136 | 0.136 | 59 | 0.730 | 0.536 | 65 |
| 32 | 0.0123 | 0.0140 | 49 | 0.113 | 0.121 | 55 | 0.603 | 0.404 | 64 |
| 64 | 0.0126 | 0.0107 | 50 | 0.104 | 0.104 | 57 | 0.653 | 0.429 | 67 |

was stored in the compressed sparse column format. Therefore, the upper triangular solve involving $G^\top$ was implemented in a straightforward way by a preorder traversal of the tree data structure used in `rchol`; see section 4. The lower triangular solve was implemented using a postorder traversal of our tree data structure. We implemented the parallel lower solve using an asynchronous approach, where the two child nodes updates the data owned by their parent asynchronously following ideas in [7, 14].

**Acknowledgments.** We would like to thank Rasmus Kyng and Daniel Spielman for private communication about the clique sampling algorithm. We also thank the anonymous referees for their thoughtful remarks, which have helped us improve the paper tremendously.

REFERENCES

[1] A. ABDELFATTAH, H. ANZT, E. G. BOMAN, E. CARSON, T. COJEAN, J. DONGARRA, M. GATES, T. GRÜTZMACHER, N. J. HIGHAM, S. LI, N. LINDQUIST, Y. LIU, J. LOE, P. LUSZCZEK, P. NAYAK, S. PRANESH, S. RAJAMANICKAM, T. RIBIZEL, B. SMITH, K. SWIRYDOWICZ, S. THOMAS, S. TOMOV, Y. M. TSAI, I. YAMAZAKI, U. M. YANG, *A Survey of Numerical Methods Utilizing Mixed Precision Arithmetic*, preprint, arXiv:2007.06674, 2020.

[2] P. R. AMESTOY, T. A. DAVIS, AND I. S. DUFF, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 886–905.

[3] H. ANZT, E. CHOW, AND J. DONGARRA, *ParILUT—A new parallel threshold ILU factorization*, SIAM J. Sci. Comput., 40 (2018), pp. C503–C519.

[4] E. G. BOMAN, B. HENDRICKSON, AND S. VAVASIS, *Solving elliptic finite element systems in near-linear time with support preconditioners*, SIAM J. Numer. Anal., 46 (2008), pp. 3264–3284.

[5] L. CAMBIER, C. CHEN, E. G. BOMAN, S. RAJAMANICKAM, R. S. TUMINARO, AND E. DARVE, *An algebraic sparsified nested dissection algorithm using low-rank approximations*, SIAM J. Matrix Anal. Appl., 41 (2020), pp. 715–746.

[6] C. CHEN, H. POURANSARI, S. RAJAMANICKAM, E. G. BOMAN, AND E. DARVE, *A distributed-memory hierarchical solver for general sparse linear systems*, Parallel Comput., 74 (2018), pp. 49–64.

[7] E. CHOW AND A. PATEL, *Fine-grained parallel incomplete LU factorization*, SIAM J. Sci. Comput., 37 (2015), pp. C169–C193.

[8] M. B. COHEN, J. KELNER, R. KYNG, J. PEEBLES, R. PENG, A. B. RAO, AND A. SIDFORD, *Solving directed Laplacian systems in nearly-linear time through sparse LU factorizations*, in Proceedings of the 2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS), IEEE, New York, 2018, pp. 898–909.

[9] T. A. DAVIS, S. RAJAMANICKAM, AND W. M. SID-LAKHDAR, *A survey of direct methods for sparse linear systems*, Acta Numer., 25 (2016), pp. 383–566.

[10] I. S. DUFF AND G. A. MEURANT, *The effect of ordering on preconditioned conjugate gradients*, BIT, 29 (1989), pp. 635–657.

[11] I. S. DUFF AND J. K. REID, *The multifrontal solution of indefinite sparse symmetric linear*, ACM Trans. Math. Software, 9 (1983), pp. 302–325.

[12] A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.
[13] L. Giraud, A. Haidar, and L. T. Watson, *Mixed-precision preconditioners in parallel domain decomposition solvers*, in Domain Decomposition Methods in Science and Engineering 17, Springer-Verlag, Berlin, 2008, pp. 357–364.
[14] C. Glusa, E. G. Boman, E. Chow, S. Rajamanickam, and D. B. Szyld, *Scalable asynchronous domain decomposition solvers*, SIAM J. Sci. Comput., 42 (2020), pp. C384–C409.
[15] K. D. Gremban, *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems*, Ph.D. thesis, Carnegie Mellon University, 1996.
[16] K. L. Ho and L. Ying, *Hierarchical interpolative factorization for elliptic operators: Differential equations*, Commun. Pure Appl. Math., 69 (2016), pp. 1415–1451.
[17] J. Hook, J. Scott, F. Tisseur, and J. Hogg, *A max-plus approach to incomplete Cholesky factorization preconditioners*, SIAM J. Sci. Comput., 40 (2018), pp. A1987–A2004.
[18] R. A. Horn, R. A. Horn, and C. R. Johnson, *Topics in Matrix Analysis*, Cambridge University Press, Cambridge, 1994.
[19] G. Karypis and V. Kumar, *A fast and highly quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1999), pp. 359–392.
[20] G. Karypis, K. Schloegel, and V. Kumar, *Parmetis: Parallel Graph Partitioning and Sparse Matrix Ordering Library*, Version 1.0, University of Minnesota, Minneapolis, 1997.
[21] J. A. Kelner, L. Orecchia, A. Sidford, and Z. A. Zhu, *A simple, combinatorial algorithm for solving SDD systems in nearly-linear time*, in Proceedings of the 45th Annual ACM Symposium on Theory of Computing, ACM, New York, 2013, pp. 911–920.
[22] K. Kim, S. Rajamanickam, G. Stelle, H. C. Edwards, and S. L. Olivier, *Task Parallel Incomplete Cholesky Factorization Using 2D Partitioned-Block Layout*, preprint, arXiv:1601.05871, 2016.
[23] I. Koutis, G. L. Miller, and R. Peng, *A nearly-m log n time solver for SDD linear systems*, in Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science, IEEE, New York, 2011, https://doi.org/10.1109/focs.2011.85.
[24] I. Koutis, G. L. Miller, and D. Tolliver, *Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing*, Comput. Vis. Image Underst., 115 (2011), pp. 1638–1646.
[25] R. Kyng and S. Sachdeva, *Approximate Gaussian elimination for Laplacians–Fast, sparse, and simple*, in Proceedings of the 2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS), IEEE, New York, 2016, pp. 573–582.
[26] Y. T. Lee and A. Sidford, *Efficient accelerated coordinate descent methods and faster algorithms for solving linear systems*, in Proceedings of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science, IEEE, New York, 2013, pp. 147–156.
[27] N. Lindquist, P. Luszczek, and J. Dongarra, *Improving the Performance of the GMRES Method Using Mixed-Precision Techniques*, in Smoky Mountains Computational Sciences and Engineering Conference, Springer-Verlag, Berlin, 2020, pp. 51–66.
[28] J. W. Liu, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Rev., 34 (1992), pp. 82–109.
[29] O. E. Livne and A. Brandt, *Lean algebraic multigrid (LAMG): Fast graph Laplacian linear solver*, SIAM J. Sci. Comput., 34 (2012), pp. B499–B522.
[30] J. A. Loe, C. A. Glusa, I. Yamazaki, E. G. Boman, and S. Rajamanickam, *Experimental Evaluation of Multiprecision Strategies for GMRES on GPUs*, preprint, arXiv:2105.07544, 2021.
[31] B. M. Maggs, G. L. Miller, O. Parekh, R. Ravi, and S. L. M. Woo, *Finding effective support-tree preconditioners*, in Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, ACM, New York, 2005, pp. 176–185.
[32] J. A. Meijerink and H. A. Van Der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp., 31 (1977), pp. 148–162.
[33] L. N. Olson and J. B. Schroder, *PyAMG: Algebraic multigrid solvers in Python v*4.0, Release 4.0, https://github.com/pyamg/pyamg, 2018.
[34] P. Raghavan and K. Teranishi, *Parallel hybrid preconditioning: Incomplete factorization with selective sparse approximate inversion*, SIAM J. Sci. Comput., 32 (2010), pp. 1323–1345.
[35] Y. Saad, *ILUT: A dual threshold incomplete LU factorization*, Numer. Linear Algebra Appl., 1 (1994), pp. 387–402.
[36] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, 2003.

[37] J. Scott and M. Tůma, *HSL_MI28: An efficient and robust limited-memory incomplete Cholesky factorization code*, ACM Trans. Math. Software, 40 (2014), pp. 1–19.

[38] D. A. Spielman and R. Kyng, *A Modification of the Sampling Solvers by Kyng and Sachdeva*, private communication, https://github.com/danspielman/Laplacians.jl/blob/master/docs/src/usingSolvers.md#sampling-solvers-of-kyng-and-sachdeva, 2020.

[39] D. A. Spielman and S.-H. Teng, *Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems*, in Proceedings of the 36th Annual ACM Symposium on Theory of Computing, ACM, New York, 2004, pp. 81–90.

[40] D. A. Spielman and S.-H. Teng, *Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems*, SIAM J. Matrix Anal. Appl., 35 (2014), pp. 835–885.