

Tackling Parallelization Challenges of Randomized Preconditioners With Dependency Tracking

Tianyu Liang
University of California, Berkeley
Berkeley, California, USA
tianyul@berkeley.edu

Chao Chen
North Carolina State University
Raleigh, North Carolina, USA
cchen49@ncsu.edu

Yotam Yaniv
Lawrence Berkeley National Lab
Berkeley, CA, USA
yotamy@lbl.gov

Hengrui Luo
Rice University
Houston, Texas, USA
hl180@rice.edu

David Tench
Lawrence Berkeley National Lab
Berkeley, California, USA
dtench@lbl.gov

Xiaoye S. Li
Lawrence Berkeley National Lab
Berkeley, California, USA
XSLi@lbl.gov

Aydin Buluç
Lawrence Berkeley National Lab
Berkeley, California, USA
abuluc@lbl.gov

James Demmel
University of California, Berkeley
Berkeley, California, USA
demmel@berkeley.edu

Jie Chen[†]
Amazon Web Services
Seattle, Washington, USA
jiechenj@amazon.com

Abstract—Incomplete Cholesky (IC) preconditioners are the standard black-box, memory-efficient first resort for large graph Laplacian systems, but classical IC can break down on irregular graphs. Approximate Cholesky (AC) addresses this with randomized dropping that is provably breakdown-free. Its stochastic fill-in pattern, however, resists static parallelization, creating a “setup bottleneck.” We present **ParAC**, which parallelizes the sequential `rchol` algorithm on CPUs and GPUs using dynamic dependency tracking and architecture-aware memory management, without altering its numerics. Total time-to-solution speedups over deterministic IC reach $57.3\times$ on CPU and $52.2\times$ on GPU. Algebraic multigrid (AMG) remains preferable on structured PDE discretizations; on irregular graph Laplacians where AMG struggles, **ParAC** attains up to $7.2\times$ and $6.0\times$. On 200 adversarial SDD systems, **ParAC** converges on every instance, whereas the AMG solvers fail on 15 and 19, on CPU and GPU, respectively.

I. INTRODUCTION

Large-scale, sparse linear systems involving symmetric diagonally dominant (SDD) matrices are foundational to modern scientific and engineering computing. Efficiently solving these systems is a prerequisite for spectral graph partitioning [1], graph-based machine learning [2]–[4], and numerical solutions of partial differential equations (PDEs) [5]. In this paper, we focus on solving the linear system

$$Lx = b, \quad (1)$$

where L is a **graph Laplacian**. This focus is well-justified, as solving general SDD linear systems can be reduced to solving Laplacian systems with minimal overhead [6].

Solver Landscape: Direct vs. Iterative. State-of-the-art methods for solving (1) generally fall into two categories: (sparse) direct and iterative solvers. Sparse direct methods [7], [8], such as Cholesky factorization, are valued for their numerical reliability. However, they frequently encounter a “memory wall” due to **fill-in**, that is, the generation of new non-zero entries during Gaussian elimination. This leads to prohibitive computational and spatial overheads for solving large-scale systems. Iterative methods offer a memory-efficient alternative but require high-quality preconditioners to achieve competitive convergence rates [9]. For graph Laplacians (and more generally, symmetric positive semi-definite matrices), a general, algebraic preconditioning approach is Incomplete Cholesky (IC) factorization [10], which constructs an approximation

$$L \approx GG^\top, \quad (2)$$

where G is a lower triangular matrix. Traditional IC drops fill-ins during elimination based on either a pre-determined sparsity pattern (level-based) [11] or a prescribed numerical threshold [12], [13]. While simple to implement, deterministic IC can be numerically unstable; dropping entries may destroy the positive semi-definiteness of intermediate Schur complements, causing the factorization to break down [14].

Breakthrough of Approximate Cholesky (AC). Recent breakthroughs have shown that IC using **randomized dropping rules** produces significantly more robust preconditioners. We refer to this family of methods collectively as **Approximate Cholesky (AC)** [15]–[17]. Every AC method is unbiased, satisfying $\mathbb{E}[GG^\top] = L$. Unlike deterministic IC, it is also provably **breakdown-free** on any connected graph Laplacian [16], [17]. The reason is that the randomized update always retains a spanning tree over the eliminated

[†] Work was done when affiliated with MIT-IBM Watson AI Lab, IBM Research.

vertex’s neighborhood. Empirically, AC also converges significantly faster than deterministic dropping rules across a wide variety of problems [16], [17]. The sequential algorithm that `ParAC` parallelizes is `rchol` (randomized Cholesky factorization) [16]; Section II-B reviews the method and its provenance in detail.

Parallelism Gap and Setup Bottleneck. Despite its numerical superiority, AC faces a significant hurdle on modern high-performance architectures, particularly GPUs. In the preconditioned iterative phase, triangular solves are now highly optimized and iteration counts are kept low with a high-quality AC preconditioner. Consequently, **the factorization (setup) phase frequently emerges as the primary computational bottleneck**. If the factorization remains serial or poorly parallelized, it negates the speedup of the subsequent solve phase.

To harness the full potential of randomized preconditioning, we need parallel algorithms that maintain AC’s numerical robustness while achieving high performance. However, AC’s stochastic nature makes this difficult: unlike level-based IC, where fill-in positions are known *a priori*, AC’s fill-in pattern is determined “online,” rendering static parallel schedules ineffective.

A. Existing Parallel Methods and Limitations

A substantial body of work has addressed the parallelization of level-based incomplete Cholesky (IC) factorizations on both CPUs [18]–[22] and GPUs [23]–[26]. These methods exploit the fact that the dependency graph is known *a priori*, before numerical factorization begins. In contrast, threshold-based IC and AC produce data-dependent sparsity patterns, leading to dynamically evolving dependencies.

Another strategy, applicable to both level-based and threshold-based IC, is domain decomposition [27], [28], which primarily targets distributed-memory settings. The closest prior work, `rchol` [16], applies this idea to AC on multi-core CPUs: a nested-dissection ordering partitions the graph so that subdomains are eliminated concurrently while the separators are eliminated sequentially. Such static schemes share an inherent drawback: partitioning carries a nontrivial preprocessing cost. `rchol` adds two more. The separator work remains serial, and the scheme does not map onto the fine-grained parallelism a GPU requires. `ParAC` instead tracks dependencies dynamically, which removes the partitioning step entirely, eliminates the serial separator phase, and admits an efficient GPU implementation.

To move beyond strict dependency constraints, Chow and Patel [29] proposed a fine-grained asynchronous approach that computes factor entries in parallel. This idea was later extended to a threshold-based GPU algorithm [30]. However, it remains unclear how to adapt this strategy effectively to AC.

Finally, recent theoretical work [31] analyzed the critical-path depth of a parallel AC algorithm that processes a maximum independent set of vertices (columns) at each step. Although this approach is theoretically appealing, translating it into an efficient parallel implementation remains challenging.

Dynamic graph data structures and runtime scheduling are not themselves new to sparse factorization; they appear, for instance, in left-looking LU implementations [32]. What is specific to AC is that the fill is *random*. The dependency structure is not merely discovered at runtime; it is generated by the algorithm as it proceeds. This rules out the symbolic preprocessing those classical schemes rely on. Our contribution is therefore not dynamic scheduling *per se*, but its AC-specific realization: the readiness invariant of Section III, the lock-free data structures of Section IV, and their mapping onto CPU and GPU memory systems.

B. Contributions: `ParAC`

In this paper, we introduce `ParAC`, a practical high-performance execution framework built on **dynamic dependency tracking** and **architecture-aware memory management**. `ParAC` parallelizes the sequential `rchol` algorithm [16] without altering its numerics. It produces the same output distribution, and therefore inherits the breakdown-free guarantee. At the same time, it exposes far more parallelism than `rchol`’s nested-dissection scheme, and it extends to GPUs for the first time. Our aim is not to displace algebraic multigrid (AMG). IC and AMG are complementary tools [33]. AMG converges near-optimally on structured PDE discretizations, but it relies on heuristic coarsening that can fail on irregular graphs. IC is a parameter-free, memory-efficient black box for any SDD system. `ParAC` advances the IC family by making it parallel while keeping it breakdown-free. Our contributions are as follows:

- **The `ParAC` Framework:** We introduce a dynamic dependency-tracking execution model that identifies the minimal dependencies of each vertex (column) at runtime. This allows `ParAC` to schedule elimination tasks with near-zero overhead, avoiding the conservative bottlenecks of static elimination trees [7], [34], which the stochastic fill-in of AC renders unusable.
- **Architecture-Aware Design:** We realize this model on both multicore CPUs and massively parallel GPUs, using lock-free dependency updates and pre-allocated memory structures to avoid the severe penalties of dynamic allocation inside parallel CUDA kernels. On the GPU we further adopt a persistent-kernel model, computing the entire factorization within a single launch so that kernel launch overhead remains negligible even for large systems.
- **Empirical Evaluation:** We evaluate `ParAC` on the NERSC Perlmutter supercomputer against deterministic IC and AMG solvers [35], [36]. An ordering study shows that fill-reducing AMD [37] remains preferable on CPUs, whereas lightweight orderings that expose structural parallelism win on GPUs, reversing the sequential guidance. Against deterministic IC, `ParAC` attains end-to-end speedups of up to **57.3×** on CPU and **52.2×** on GPU, measured where the baseline itself converges; on irregular graph Laplacians where AMG’s coarsening struggles, it attains up to **7.2×** and **6.0×**. It also remains

robust on massive real-world graphs where classical IC fails to converge and AMG exhausts GPU memory.

Our code is available at <https://github.com/Tianyu-Liang/P-parallel-Randomized-Cholesky>.

II. BACKGROUND

In this section, we establish the foundations of the Approximate Cholesky (AC) algorithm. We focus on the relationship between the algebraic elimination steps and their corresponding graph-theoretic interpretations, which dictate the data dependencies explored in this work.

A. Graph Laplacians and Graph-Matrix Duality

We begin with the graph Laplacian, defined as follows:

Definition 1 (Graph Laplacian). A matrix $L \in \mathbb{R}^{N \times N}$ is a graph Laplacian if (1) L is symmetric, (2) the row sums are zero ($\sum_j \ell_{ij} = 0$), and (3) all off-diagonal entries are non-positive ($\ell_{ij} \leq 0$ for $i \neq j$).

There is a one-to-one mapping between a Laplacian L and an undirected, weighted graph $\mathcal{G} \triangleq (V, E, w)$. Each row/column index corresponds to a vertex, and each non-zero off-diagonal entry ℓ_{ij} represents an edge between vertices i and j with weight $w_{ij} \triangleq -\ell_{ij}$. Alternatively, L can be expressed as a sum of outer products representing individual edges:

$$L = \sum_{(i,j) \in E} w_{ij} (\mathbf{e}_i - \mathbf{e}_j)(\mathbf{e}_i - \mathbf{e}_j)^\top, \quad (3)$$

where $\mathbf{e}_i$ is the i -th standard basis vector in $\mathbb{R}^N$.

For the remainder of this paper, we assume $\mathcal{G}$ is connected (i.e., L is irreducible). Under this assumption, L has a single zero eigenvalue with a null space spanned by the all-ones vector. As shown in [16], AC is robust to this singularity; pivots only vanish at the final step, and Krylov solvers remain stable provided the system $Lx = b$ is consistent ($b \perp \mathbf{1}$).

B. From Clique Formation to Clique Sampling

Consider applying one step of Cholesky factorization (Gaussian elimination) to a Laplacian matrix on row/column k . Algebraically, this forms a Schur complement. Graphically, the neighbors of k form a **clique** [34], [38], that is every neighbor is now connected to every other neighbor. This “clique formation” is the source of **fill-in**. In standard Cholesky, a vertex with d neighbors generates $O(d^2)$ new edges. In large-scale systems, these cliques quickly become dense, creating a “memory wall” and complex dependency chains that serialize the factorization process.

AC, summarized in Algorithms 1 and 2, belongs to the IC family: it computes a factorization of the form (2), and differs from classical IC only in how fill-in is discarded. In place of the $O(d^2)$ clique update it inserts a stochastically sampled spanning tree of only $O(d)$ edges.

Algorithm 1 Approximate Cholesky (AC) for Graph Laplacian Matrix [15], in the practical form of `rchol` [16]

Input: Graph Laplacian matrix $L \in \mathbb{R}^{N \times N}$
Output: Lower triangular factor $G \in \mathbb{R}^{N \times N}$ ($L \approx GG^\top$)

```

1:  $G = \mathbf{0}_{N \times N}$ 
2: for  $k = 1$  to  $N - 1$  do  $\triangleright N - 1$  iterations
3:   if  $L(:, k)$  is empty then
4:     continue
5:   end if
6:    $G(:, k) = L(:, k) / \sqrt{\ell_{kk}}$   $\triangleright \ell_{kk} > 0$  [16]
7:    $L = L - L^{(k)} + \text{SampleClique}(L, k)$   $\triangleright$  see (4)
8: end for
```

Algorithm 2 `SampleClique`: the sorted dropping rule of `rchol` [16]; [17] publishes the same rule without the sort

Input: Graph Laplacian matrix $L \in \mathbb{R}^{N \times N}$, index k
Output: Graph Laplacian $C \in \mathbb{R}^{N \times N}$ for sampled edges

```

1:  $C = \mathbf{0}_{N \times N}$ 
2:  $\mathcal{N}_k = \{i \mid i \neq k, \ell_{ki} \neq 0\}$   $\triangleright$  neighbors of  $k$ 
3: Sort  $\mathcal{N}_k$  in ascending order based on  $|\ell_{ki}|$  for  $i \in \mathcal{N}_k$ 
4:  $S = \ell_{kk}$ 
5: while  $|\mathcal{N}_k| > 1$  do  $\triangleright$  loop over neighbors
6:   Let  $i$  be the first element in  $\mathcal{N}_k$ 
7:    $\mathcal{N}_k = \mathcal{N}_k \setminus \{i\}$   $\triangleright$  remove  $i$  from the set
8:    $S = S + \ell_{ki}$ 
9:   Sample  $j$  from  $\mathcal{N}_k$  with probability  $|\ell_{kj}|/S$ 
10:   $C = C - \frac{S \ell_{ki}}{\ell_{kk}} \mathbf{b}_{ij} \mathbf{b}_{ij}^\top$   $\triangleright$  select weighted edge  $(i, j)$ 
11: end while
```

a) *Provenance:* AC originates with Kyng and Sachdeva [15], who introduced the randomized dropping rule that reweights the fill-in it retains, and proved that the resulting factorization is a spectral approximation of L . Two later variants, `rchol` [16] and [17], trade that guarantee for practical efficiency. Both adopt an improved dropping rule, differing only in whether the neighbors of k are sorted before sampling. The rule used by `rchol` is due to Spielman and Kyng (private communication, 2020), implemented in `Laplacians.jl` [39] and later published in [17]. `ParAC` parallelizes `rchol`, using its sorted variant of the rule.

The critical challenge for parallelism lies in Line 7 of Algorithm 1: the edges drawn by `SampleClique` are **stochastic**, so the dependency structure is generated online (Section III).

To formalize the mechanics of AC, we first define the local Laplacian $L^{(k)}$, which represents the star graph centered at vertex k and its neighbors $\mathcal{N}_k \triangleq \{i \mid i \neq k, \ell_{ki} \neq 0\}$:

$$L^{(k)} \triangleq \sum_{i \in \mathcal{N}_k} (-\ell_{ki}) \mathbf{b}_{ki} \mathbf{b}_{ki}^\top, \quad (4)$$

where $\mathbf{b}_{ki} \triangleq \mathbf{e}_k - \mathbf{e}_i$ denotes the difference between two standard basis vectors. The symmetry of the outer product $(\mathbf{e}_k - \mathbf{e}_i)(\mathbf{e}_k - \mathbf{e}_i)^\top$ ensures that the edge orientation does not affect the resulting Laplacian.

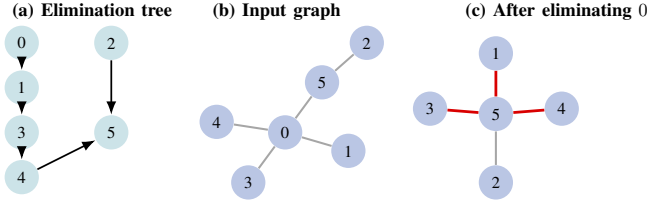


Fig. 1: Dynamic dependency tracking. (a) Elimination tree, which pessimistically serializes $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$. (b) Input graph. (c) After eliminating 0, a sampled spanning tree (red) replaces the clique on its neighbors.

By replacing the quadratic fill-in of a full clique with $O(|\mathcal{N}_k|)$ sampled edges, AC dramatically reduces the computational footprint. The sampling is not arbitrary: the $|\mathcal{N}_k| - 1$ edges drawn by Algorithm 2 always form a **spanning tree** over the neighborhood $\mathcal{N}_k$, and their weights are chosen so that the sampled update is itself a graph Laplacian. Because the active graph therefore stays a connected Laplacian at every step, the factorization is breakdown-free [16, Thm. 2.7, Cor. 2.8], [17, Claim 4.5].

b) Theoretical Qualification: Since ParAC changes only the elimination schedule, not SampleClique (Section III), the caveats here concern the AC numerics rather than our parallelization. Each sampled update is conditionally unbiased: given the current active graph, its expectation is the exact clique Schur complement, which telescopes into $\mathbb{E}[GG^\top] = L$. Unbiasedness is a first-moment property only. The variant we parallelize omits the multi-edge splitting and the uniformly random elimination order that together supply the per-step variance bounds of the matrix-martingale analysis in [15], so it falls outside that analysis, and no comparable spectral-approximation bound is established for [16], [17]. Empirically, however, the spectral property appears to be approximately preserved. On refined three-dimensional Poisson discretizations, the preconditioned iteration counts reported in [16, Tab. 7–8] grow as $O(\log N)$, corresponding to a preconditioned condition number of $O(\log^2 N)$. This is close to, though short of, the $O(1)$ that AMG attains on such problems. The work bounds proven for AC [15]–[17] likewise assume a uniformly random elimination order, whereas the fill-reducing orderings we adopt in practice fall outside that hypothesis and prove faster still (Section V).

III. DYNAMIC DEPENDENCY TRACKING

Dynamic dependency tracking is the key mechanism to expose parallelism despite AC’s stochastic fill-ins. Instead of relying on a static symbolic schedule, ParAC tracks the active dependency structure online and schedules a vertex as soon as its dependencies vanish.

A. Dependency Structure

Figure 1 illustrates why AC can expose more parallelism than complete Cholesky. Panel (a) is the elimination tree of the graph in panel (b), and a static schedule derived from it

eliminates $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ in sequence. ParAC instead finds 1, 3, and 4 ready at the same time once 0 has been eliminated, as the trace later in this section shows. If vertex 0 were eliminated by complete Cholesky, its neighbors would form a clique, inducing broad reachability and many dependencies. In contrast, AC forms only a sampled spanning tree, which reduces reachability and allows more vertices to proceed in parallel.

For *exact* Cholesky on a statically known graph, the dependencies of a vertex are characterized by reachability.

Lemma 1 ([40]). *Define the dependencies of $i \in V$ to be the vertices that must be eliminated before i is eliminated. Then i depends on j if and only if $j < i$ and there exists a path*

$$i \rightarrow p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n \rightarrow j$$

such that every internal vertex satisfies $p_1, \dots, p_n < i$.

Lemma 1 predicts fill from the *input* graph alone. That is useful for exact factorization but inapplicable here, because under AC the active graph evolves stochastically, so the transitive paths it quantifies over are not known in advance. ParAC sidesteps the prediction entirely by reading readiness off the current active graph, where a single scalar suffices. Let

$$\mathcal{G}^{(t)} \triangleq (V^{(t)}, E^{(t)}, m^{(t)})$$

be the active multigraph after t elimination steps, where $m^{(t)}(i, j)$ denotes edge multiplicity. For an active vertex $i \in V^{(t)}$, define

$$d_t(i) \triangleq \sum_{j < i} m^{(t)}(i, j),$$

namely, the number of incident edges from lower-labeled active vertices, counted with multiplicity.

Readiness then follows directly from the invariant, with no appeal to reachability. Eliminating a pivot k modifies exactly the vertices in $\mathcal{N}_k$, so an active vertex i can still be modified only by the elimination of an active vertex $j < i$ that is *currently* adjacent to i . If $d_t(i) = 0$ there is no such j : every remaining coupling of i is to a higher-labeled vertex, which will be eliminated after i , so i may be eliminated immediately. If $d_t(i) > 0$ then some active $j < i$ is adjacent to i , and eliminating j will update i , so i must wait. Hence an active vertex is ready exactly when $d_t(i) = 0$.

The saving is that ParAC needs only the scalar counter $d_t(i)$, updated locally as edges are inserted and deleted, rather than any transitive reachability information. Because the criterion is evaluated on the active graph rather than predicted from the input, it remains exact under AC’s stochastic fill-in, where a static schedule derived from Lemma 1 would be conservative.

B. Proposed Tracking Strategy

ParAC tracks dependencies online rather than estimating them statically. At any time, the ready set is

$$Q_t \triangleq \{i \in V^{(t)} \mid d_t(i) = 0\},$$

so vertices in Q_t can be eliminated immediately and, subject to hardware availability, in parallel.

Initially (Algorithm 3, lines 2–4), every input edge has multiplicity one, so

$$d_0(i) = |\{j \mid e_{ij} \neq 0, j < i\}|.$$

This is exactly the initialization of $\text{dp}[i]$ in line 2 of Algorithm 3. Vertices with $d_0(i) = 0$ are inserted into the initial job queue in line 3, and the CPU data structures are prepared in line 4.

Each thread repeatedly polls its assigned queue locations and retrieves a ready pivot k (Algorithm 3, lines 5–6). This is the implementation-level realization of the ready set Q_t .

After gathering and sorting the active neighbors of k (Algorithm 3, lines 13–16), ParAC samples Schur-complement edges among those neighbors. Whenever a sampled edge is formed between active vertices a and b , the larger-labeled endpoint gains one new lower-labeled incident edge, so $d(\max(a, b)) \leftarrow d(\max(a, b)) + 1$. Then the sampled edge is then inserted into the linked-list structure.

When the pivot k is eliminated, all active edges incident to k are removed. Thus, for every neighbor $i > k$, $d(i) \leftarrow d(i) - m(i, k)$, which appears in aggregate form in line 18 as $\text{dp}[i] \text{--} \text{N_k}(i) \cdot \text{multiplicity}$ (Algorithm 3, line 18). After these insertion and deletion updates, any vertex whose dependency count becomes zero is inserted into the job queue. This is exactly the readiness condition

$$d_t(i) = 0 \iff i \in Q_t.$$

Figure 1 shows one example of the elimination process, following the order in which Algorithm 3 performs the updates: insertions first, then deletions, with enqueueing triggered only by a positive-to-zero transition. Before eliminating vertex 0, the input graph (panel (b)) gives initial dependency counts $d = (d_0, d_1, d_2, d_3, d_4, d_5) = (0, 1, 0, 1, 1, 2)$. First, the sampled Schur-complement edges $(1, 5), (3, 5), (4, 5)$ are inserted (panel (c)); each raises the count of its larger-labeled endpoint, so d_5 goes from 2 to 5. Only then is the pivot removed: deleting edges $(0, 1), (0, 3), (0, 4), (0, 5)$ decrements d_1, d_3, d_4, d_5 by one each, leaving $(d_1, d_2, d_3, d_4, d_5) = (0, 0, 0, 0, 4)$. Vertices 1, 3, and 4 transition to zero and are enqueued; vertex 5 does not.

This ordering is not incidental. A sampled edge raises the count of its larger-labeled endpoint, which is itself a neighbor of the pivot and therefore also due a decrement. Were the deletions applied first, such a vertex could transiently reach zero and be enqueued before its incoming sampled edge was recorded. Another thread could then begin eliminating a vertex that still had a pending dependency. Performing insertions first keeps the counter an upper bound on the outstanding dependencies at all times, so a zero crossing is always genuine.

Hence ParAC maintains the invariant that $d_t(i)$ equals the total multiplicity of lower-labeled incident edges of i in the current active multigraph. A vertex becomes ready exactly when this counter reaches zero, at which point it is inserted into the ready queue.

This dependency rule is independent of the hardware backend. The CPU and GPU implementations in the next section differ in how updates are stored and searched, but both use the same counter-based readiness criterion and ready-queue semantics.

a) *Distribution Preservation*: Distinct ready pivots are non-adjacent: if $i < j$ were both active and adjacent, then j would have the lower-labeled active neighbor i , giving $d_t(j) > 0$ and contradicting $j \in Q_t$. Consequently, eliminating one ready pivot leaves every other ready pivot’s incident star unchanged. Eliminating j deletes the edges incident to j and inserts sampled edges among $\mathcal{N}_j$; since $i \notin \mathcal{N}_j$, no edge incident to i is deleted and no sampled edge can land on i . Because SampleClique draws using only the weights on the pivot’s own star, each ready pivot therefore has the same conditional sampling distribution whether it is eliminated first, last, or concurrently. The pivots’ contributions need not be disjoint. Two ready pivots may share higher-labeled neighbors, and may even sample the same edge, which is why the active graph is a multigraph. But each contributes an additive rank-one term to the Schur complement, and addition commutes, so the accumulated result is independent of the order in which the contributions are applied. With independent random draws, eliminating all of Q_t simultaneously therefore produces the same joint distribution over the output factor G as any sequential order respecting the dependency partial order. In particular, the output distribution of ParAC matches that of the sequential rchol algorithm, which is provably breakdown-free on connected graph Laplacians [16, Thm. 2.7, Cor. 2.8].

IV. PARALLEL DESIGNS FOR CPUs AND GPUS

We now present our parallelization method, ParAC , based on the dependency tracking strategy discussed above. We dissect how the architectural differences between CPUs and GPUs shape the design details of ParAC and present two algorithms, respectively. Both store the factor in the scaled form $L \approx FDF^\top$, where F is unit lower triangular and D is diagonal, which avoids a square root per column; this is the factor G of Algorithm 1 under $G = FD^{1/2}$.

At a high level, ParAC first selects an elimination ordering and labels all vertices accordingly. The rest of the elimination steps can be divided into three main stages, following Algorithm 1 and Algorithm 2:

- 1) Search and organize fill-in updates for an eliminated vertex.
- 2) Sort the neighbors of the pivot and perform sampling.
- 3) Perform Schur-complement updates, update the dependencies, and schedule any vertex that is ready to be eliminated.

Details are discussed in the subsequent subsections, with the flow chart previewed in Figure 2.

A. Common Design Details

There are two considerations common to CPUs and GPUs. Our design starts from these common considerations.

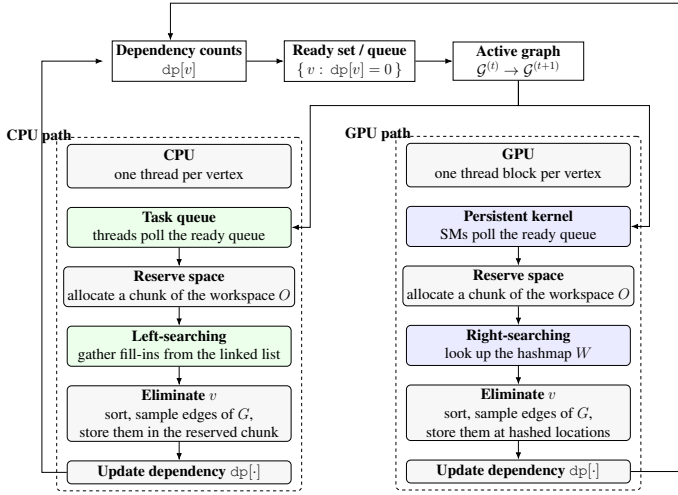


Fig. 2: Flow chart of the ParAC pipeline.

a) *Memory Layout*: A practical obstacle to an efficient parallelization is memory provisioning under randomization. Because AC randomly samples vertex pairs to form new edges, column-wise fill is unknown a priori. Rather than repeatedly resizing per-column containers, which is costly, we design data structures that avoid per-column reallocations and support concurrent producers.

Specifically, we pre-allocate a contiguous workspace O that will hold the final matrix and the temporary update vertices. The threads reserve a column space by using atomic `fetch_add` on the global variable shared by all threads (or thread blocks). This removes the need to know the exact per-column size, provided the reserved workspace U does not overflow. We set U to four times the size of the input by default, and to six times for `com-LiveJournal`, the densest matrix in our suite. For GPU implementation, there is an additional workspace W that functions like an open addressed hashmap.

b) *Work Scheduling*: A naive approach is to schedule the vertices once they are ready by using lock-related semantics, but this may cause severe thread contention that slows down the whole factorization. We take the following approach instead: whenever a vertex becomes ready, it is inserted into a queue. All the threads (or thread blocks) poll their respective search indices inside the queue, which are assigned in a round-robin fashion based on the thread/block id. For example, if there are four threads, thread 0 is assigned $\{0, 4, 8, \dots\}$ while thread 1 is assigned $\{1, 5, 9, \dots\}$.

B. CPU Algorithm

The CPU implementation is based on the “left-searching” mechanism: at each step, the Schur-complement updates are stored locally and chained together by a linked list. Vertices to be eliminated later will traverse the list (hence the term “left”) and aggregate these updates. This is analogous to the left-looking strategy in sparse LU factorization [32], where a

Algorithm 3 ParAC on CPUs

Input: Laplacian matrix $L \in \mathbb{R}^{N \times N}$, `num_threads` threads
Output: Output array O containing the triangular factor, diagonal matrix D

```

1:  $dp[i] \leftarrow |\{j \mid j < i, e_{ij} \neq 0\}|$  for all  $i$ 
2:  $q \leftarrow \{i \mid dp[i] = 0\}$ ;  $P \leftarrow$  linked-list heads
3: for  $id = \text{thread\_id}$  to  $N-1$  step num_threads do
4:    $k \leftarrow q[id]$ , spin-waiting if necessary
5:   Reserve a chunk of the workspace  $O$  for  $k$ 
6:    $\mathcal{N}_k \leftarrow$  traverse the linked list from  $P(k)$ 
7:    $D(k, k) \leftarrow \sum_{i \in \mathcal{N}_k} |\ell_{ik}|$ , skipping  $k$  if 0
8:   Merge duplicates in  $\mathcal{N}_k$ , then sort by ascending  $|\ell_{ik}|$ 
9:    $S \leftarrow$  suffix sums of  $|\ell_{ik}|$  over  $\mathcal{N}_k$ 
10:  for  $p = 1 : |\mathcal{N}_k| - 1$  do
11:     $u \leftarrow \text{vertex}(\mathcal{N}_k[p])$ 
12:    Sample  $v$  from  $\mathcal{N}_k[p+1:]$  w.p.  $|\ell_{kv}|/S[p+1]$ 
13:     $\text{AtomicAdd}(dp[\max(u, v)], 1)$ 
14:     $P(\min(u, v)) \leftarrow -\frac{S[p+1]\ell_{ku}}{\ell_{kk}} \mathbf{b}_{uv} \mathbf{b}_{uv}^\top$ 
15:  end for
16:  for  $p = 1 : |\mathcal{N}_k|$  do
17:     $u, m \leftarrow \text{vertex}(\mathcal{N}_k[p]), \text{multiplicity}(\mathcal{N}_k[p])$ 
18:     $r \leftarrow \text{AtomicSub}(dp[u], m)$   $\triangleright$  prior count
19:    if  $r = m$  then  $\text{AtomicAppend}(q, u)$ 
20:  end for
21: end for

```

column aggregates contributions from all previously factored columns before it is processed.

a) *Reserve Space and Left-Searching*: Consider a batch $B \triangleq \{v_1, \dots, v_m\}$ of pivots popped from the job queue. Let $\mathcal{N}_{v_k}$ denote the neighbors of v_k , $1 \leq k \leq m$. We first calculate an upper bound on the size of the local chunk in O required by v_k . The space needed by the fill-ins generated for v_k can be traced using a counter. Additionally, since each vertex in $\mathcal{N}_{v_k}$ only samples one new edge to form the spanning tree, the required space by the newly sampled edges is at most $|\mathcal{N}_{v_k}|$. Summing these two terms yields the needed space. We then add the sum to an atomic variable shared by all threads. The old atomic value indicates the starting index of the local chunk. After reserving space, we begin left-searching for the fill-ins of v_k , which are stored in a linked list. We write these elements into the local chunk that was just reserved.

b) *Eliminate and Schedule*: We then aggregate the fill-ins. Let $T \triangleq \{t_1, \dots, t_q\}$ be the set modified by the Schur-complement update of B and let $P \triangleq \{p_1 \rightarrow \text{fill-ins}, \dots, p_q \rightarrow \text{fill-ins}\}$ represent the pointers owned by T , where fill-ins refer to the existing fill-ins each vertex $t_h \in T$ must aggregate. We use fill-ins to indicate all entries, even if such an entry already exists (in which case, we merge them). Suppose $v_k \in B$ modifies t_h via the Schur-complement update. Then, v_k inserts the sample it generates into t_h 's linked list. That is, $p_h \rightarrow \text{sample}(v_k) \rightarrow \text{fill-ins}$, where $\text{sample}(v_k)$ is generated by some neighbor of v_k and is stored in the local chunk owned by v_k .

Since B is eliminated in parallel, a race condition may

Algorithm 4 ParAC on GPUs

Input: Laplacian matrix $L \in \mathbb{R}^{N \times N}$, num_blocks thread blocks

Output: Output array O containing the triangular factor, diagonal matrix D

```
1:  $\text{dp}[i] \leftarrow |\{j \mid j < i, e_{ij} \neq 0\}|$  for all  $i$ 
2:  $q \leftarrow \{i \mid \text{dp}[i] = 0\}$ ;  $W \leftarrow$  empty update workspace
3: for  $\text{id} = \text{block\_id}$  to  $N - 1$  step num_blocks do
4:    $k \leftarrow q[\text{id}]$ , spin-waiting if necessary
5:   Reserve a chunk of the workspace  $O$  for  $k$ 
6:    $\mathcal{N}_k \leftarrow$  probe  $W$  in parallel from  $\text{hash}(k)$ 
7:    $D(k, k) \leftarrow \sum_{i \in \mathcal{N}_k} |\ell_{ik}|$ , skipping  $k$  if 0
8:   Merge duplicates in  $\mathcal{N}_k$  by sort and prefix sum, then
   sort by ascending  $|\ell_{ik}|$ 
9:    $S \leftarrow$  parallel suffix sums of  $|\ell_{ik}|$  over  $\mathcal{N}_k$ 
10:  for  $p = 1 : |\mathcal{N}_k| - 1$  in parallel do
11:     $u \leftarrow \text{vertex}(\mathcal{N}_k[p])$ 
12:    Sample  $v$  from  $\mathcal{N}_k[p+1:]$  w.p.  $|\ell_{kv}|/S[p+1]$ 
13:    AtomicAdd( $\text{dp}[\max(u, v)], 1$ )
14:     $W(\text{hash}(\min(u, v))) \leftarrow -\frac{S[p+1]\ell_{ku}}{\ell_{kk}} \mathbf{b}_{uv} \mathbf{b}_{uv}^\top$ 
15:  end for
16:  for  $p = 1 : |\mathcal{N}_k|$  in parallel do
17:     $u, m \leftarrow \text{vertex}(\mathcal{N}_k[p])$ , multiplicity( $\mathcal{N}_k[p]$ )
18:     $r \leftarrow \text{AtomicSub}(\text{dp}[u], m)$   $\triangleright$  prior count
19:    if  $r = m$  then AtomicAppend( $q, u$ )
20:  end for
21: end for
```

happen if multiple elements in B update the same element in T . We use atomic exchanges to preserve the integrity of the linked list after updating dependencies. Dependency counts are then incremented and decremented, and ready vertices enqueued, exactly as specified in Section III.

The overall procedure is summarized in Algorithm 3.

C. GPU Algorithm

GPU design is considerably more involved. First, ParAC must run in units of warps to avoid warp divergence, whereas on the CPU the smallest unit is a single thread. Second, AC's sparse sampling yields low computational intensity per elimination step, so a naively launched kernel may not amortize its launch overhead. Third, inter-block synchronization is limited and error-prone, and kernels coordinating across blocks can deadlock. Fourth, ParAC must sort each neighborhood, and we are unaware of efficient warp-level sorts for a variable number of elements.

All these considerations prompt us to use a different mechanism to implement ParAC on GPUs. We use "right-searching," where the Schur-complement updates are written to the target location (rather than stored locally), such that vertices to be eliminated later do not need to aggregate previous updates.

a) Address Warp Divergence: We assign each job to a thread block consisting of warp(s). Then, we structure

the computation in a manner that takes advantage of multi-threading within each warp, by mapping the neighbors of each vertex to a thread. There may be idle threads, if the number of neighbors is fewer than 32, or we may need multiple rounds of coverage if there are more neighbors than threads.

b) Reduce Kernel Launches: Unlike previous approaches [41], we adopt a persistent kernel. All blocks remain active and continuously poll the queue at their assigned locations, fetching and performing a job whenever one is available. We repeat this in a loop until the factorization is complete. This strategy fits the entire factorization, minus some minor preprocessing, into one kernel, thereby reducing kernel launch costs.

c) Reserve Space and Right-Searching: Linked list-based design, as in the CPU case, is no longer practical, because pointer jumping cannot take advantage of intra-block parallelism. In order for a vertex $v_k \in B$ to efficiently search for its fill-ins, all fill-ins should be grouped together in a contiguous segment. This motivates a linear-probing, array-based hashmap design with a twist that elements are inserted in groups. We call this hashmap array W and refer to future searches conducted in this map as "right-searching," because Schur-complement updates are written to the target column's hash location rather than stored locally.

More specifically, the vertex v_k generates a hashcode $\text{hash}(v_k)$, which indicates the initial search location. The block of threads responsible for eliminating v_k will then search the array in parallel until it finds the expected number of fill-ins. The workspace W differs from O : W only temporarily stores fill-ins, and the region holding v_k 's fill-ins is freed for reuse once they are moved to O . Each entry of W is free, busy, or occupied; busy entries are being modified, so other threads spin-wait.

d) Sorting: We merge fill-ins by sorting and prefix-sum. Nvidia's CUB provides efficient warp-level sorting, but requires the elements per thread to be known at compile time, which the dynamically varying fill-in count violates. We therefore wrote block-level odd-even and bitonic sorts handling an arbitrary number of elements, choosing between CUB and ours by an element-count threshold. Our sorter pads the array with sentinel values to the next power of two, matching the network's fixed-radix structure.

e) Eliminate and Schedule: Consider the set of vertices updated by v_k 's Schur-complement update (i.e., $\mathcal{U} \triangleq \{t_h \mid t_h \in T, t_h \in \mathcal{N}(v_k)\}$). The thread block calculates a value $\text{hash}(a) + \text{fill_in_count}(a)$ for every $a \in \mathcal{U}$ and inserts them into the corresponding location in parallel, where $\text{fill_in_count}(a)$ refers to the number of existing fill-ins of a . Adding this value can potentially speed up insertion in most cases, since the spots before $\text{hash}(a) + \text{fill_in_count}(a)$ are likely taken. The dependency calculation and queue scheduling are similar to those of the CPU algorithm.

f) Efficient Hashing: Hashing quality has a significant impact on the performance of the GPU algorithm. When a batch B is eliminated, the vertices it updates are written to

their hashed locations in W , and probing conflicts arise when those locations fall close together. For the target set T of one batch, the quantity we want to be large is therefore the smallest circular gap between distinct hashed positions,

$$\min_{\substack{a,b \in T \\ a \neq b}} d(\sigma(a), \sigma(b)), \quad d(x, y) \triangleq \min(|x-y|, |W|-|x-y|),$$

where the circular distance d accounts for wraparound in the open-addressed table. No single σ can maximize this for every batch simultaneously, so we treat it as a heuristic: setting σ to a random permutation works well in practice, whereas the identity mapping is markedly worse. This permutation refers to the storage location to which the updates associated with a vertex are mapped; it is not the same as the elimination order.

With all the addressed details, the overall GPU implementation is summarized in Algorithm 4.

V. NUMERICAL RESULTS

a) Experimental Methodology: For each benchmark, we solve a linear system $Lx = b$. We draw a synthetic ground-truth solution x^* with entries sampled i.i.d. from the standard normal distribution $\mathcal{N}(0, 1)$, and set the right-hand side to $b = Lx^*$. `ParAC` computes an IC factorization $L \approx GG^\top$, where G is sparse and lower triangular. We then use this factor as a preconditioner for the preconditioned conjugate gradient (PCG) method, which returns an approximate solution $\tilde{x}$. We report three metrics:

- 1) **Factorization Time:** The wall-clock time required to construct the approximate Cholesky factor G .
- 2) **Solve Time:** The wall-clock time for the PCG solver to converge to the target tolerance.
- 3) **Iteration Count:** The number of PCG iterations required to reach a relative residual of 10^{-6} .

All timing results are arithmetic means over five independent trials. Because the algorithm is randomized, results vary slightly from run to run, but the variation is too small to affect the comparisons below. Previous studies of sequential AC [16], [17] report that the convergence of randomized preconditioning is stable across runs. Our parallel implementation behaves the same way in every experiment. Some matrices in our suite are not naturally SDD. On those, `ParAC` applies implicit diagonal compensation, raising the diagonal where needed to make it positive, and we precondition the original system with the resulting factors.

b) Ordering for ParAC: As in exact Cholesky factorization, the performance of `ParAC` depends on the initial ordering, that is, on the permutation applied to the input matrix. Minimizing fill-in is the primary objective in the sequential setting, but the ordering also determines how much parallelism is available. We evaluate three representative strategies:

- 1) **Approximate Minimum Degree (AMD)** [37]: A widely used fill-reducing heuristic, and an effective one for `rchol` [16].
- 2) **NNZ-Sorted:** Columns are ordered by ascending degree, where the degree of a column is its number of non-zero (NNZ) entries. Ties are broken randomly.

- 3) **Random:** A uniformly random ordering, adopted in [15] to make the theoretical analysis tractable.

Orderings designed specifically for randomized elimination have also been proposed [42]. We restrict our study to the three general-purpose heuristics above, none of which requires domain knowledge of the input.

c) Test Matrices: Table I lists the number of columns and the number of non-zeros (NNZ) for each matrix in our test suite, which falls into four groups:

- **General Scientific Computing:** Five matrices from the SuiteSparse Matrix Collection [43] representing CFD, ecology, structural mechanics, and circuit simulation.
- **Structured 3D Poisson:** Three matrices generated with the `Laplacians.jl` package [39]. Each is a finite-difference discretization of the Poisson equation on a regular 3D grid with the standard 7-point stencil. The three differ in their coefficients: uniform, anisotropic, and high-contrast, abbreviated `uniform`, `aniso`, and `contrast` Poisson in the results tables.
- **Porous Media Flow:** The `spe16m` matrix from the Society of Petroleum Engineers (SPE) benchmark [44], a large-scale industrial model of fluid flow in porous media.
- **Graph Laplacians:** Six network matrices from SuiteSparse [43]. These are adjacency matrices of weighted undirected graphs, which we convert to Laplacians by setting each diagonal entry to the negated sum of the off-diagonal entries in its row.

TABLE I: Properties of the test matrices.

Matrix Name	# Columns	NNZ	NNZ / Col
parabolic_fem	525,825	3,674,625	6.99
ecology1	1,000,000	4,996,000	5.00
ecology2	999,999	4,995,991	5.00
apache2	715,176	4,817,870	6.74
G3_circuit	1,585,478	7,660,826	4.83
uniform 3D Poisson	14,348,907	100,088,055	6.98
anisotropic 3D Poisson	14,348,907	100,088,055	6.98
high-contrast 3D Poisson	14,348,907	100,088,055	6.98
spe16m	16,003,008	111,640,032	6.98
GAP-road	23,947,347	57,708,624	2.41
com-LiveJournal	3,997,962	69,362,378	17.35
europe_osm	50,912,018	108,109,320	2.12
delaunay_n24	16,777,216	100,663,202	6.00
venturiLevel3	4,026,819	16,108,474	4.00
belgium_osm	1,441,295	3,099,940	2.15

d) Reference Baselines: We compare `ParAC` against deterministic incomplete factorizations and AMG solvers. For IC we use MATLAB on the CPU and NVIDIA `cuSPARSE` [45] on the GPU. For AMG we use BoomerAMG [35] through the HyPre library [46] on the CPU and NVIDIA `AmgX` [47] on the GPU. Applying the preconditioner $GG^\top$ requires triangular solves in the iterative phase, for both `ParAC` and IC. We perform these with Intel MKL on the CPU and `cuSPARSE` on the GPU, respectively.

e) Hardware Platform: We ran all experiments on the NERSC Perlmutter supercomputer, on compute nodes with

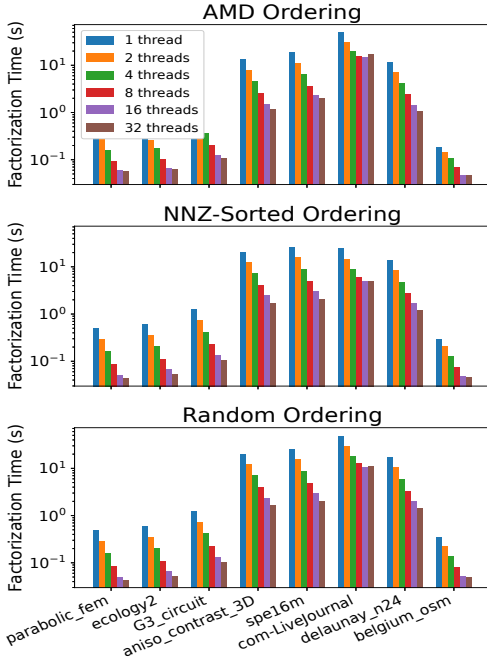


Fig. 3: CPU factorization time under three orderings.

AMD EPYC 7763 CPUs and NVIDIA A100 GPUs. Unless otherwise noted, CPU experiments use 32 threads. On the GPU, ParAC uses a default configuration of 512 blocks with one warp per block. The exception is `com-LiveJournal`, for which we use 96 blocks with 8 warps per block. Its high average degree makes the per-vertex workload much larger, and the wider configuration is needed to absorb it.

A. CPU Results

a) Impact of Ordering on CPU Performance: We evaluate the impact of the three ordering heuristics, AMD, NNZ-sorted, and random, on the factorization phase. Figure 3 shows the results for a representative subset of eight benchmarks. The same trends hold across the full 15-matrix suite.

Across most benchmarks, scaling to 32 threads yields approximately a $10\times$ **speedup**. **AMD ordering** typically gives the lowest wall-clock time, consistent with its fill-reducing objective and with what was observed for `rchol` [16]. The exception is `com-LiveJournal`, the densest matrix in our suite, where AMD is the slowest of the three orderings. Its factorization takes 17.3s at 32 threads, against 4.8s for NNZ-sorted. AMD also scales worst here, gaining only $2.9\times$ from 1 to 32 threads and losing 19% between 16 and 32. On a dense, structureless graph, aggressive fill reduction lengthens the critical path [48], and the parallelism lost outweighs the arithmetic saved. The same effect decides the ordering choice on GPUs (Section V-B).

AMD ordering brings a second benefit. The triangular factors it produces also cluster their non-zeros more tightly [16], which improves cache locality in the PCG solve phase and lowers the overall solution time. We therefore recommend

AMD as the default ordering for ParAC on CPUs, with NNZ-sorted preferable on very dense inputs.

b) Comparison with Baselines: We compare the wall-clock performance of ParAC against two baselines: threshold-based incomplete Cholesky (`ichol`) from MATLAB, and the AMG solver in the HyPre library. Table II reports the results.

Performance vs. `ichol`: Both ParAC and `ichol` are incomplete factorization methods, so the two are direct competitors. For an iso-memory comparison, we tuned the drop tolerance on each benchmark so that the `ichol` factor occupies as much memory as the ParAC factor, or slightly more. ParAC is faster than `ichol` on 14 of the 15 test cases. The one exception is `contrast Poisson`, where ParAC is about 20% slower.

ParAC also needs far fewer iterations than `ichol` on every benchmark. On several matrices, including `ecology1`¹ and `GAP-road`, the total execution time of ParAC is more than $10\times$ lower than that of `ichol`; on `com-LiveJournal` the margin is $5.9\times$. The gap comes from the iteration counts of `ichol`, not from MATLAB overhead. Where the gap is largest, factorization accounts for at most 1% of `ichol`'s total time, and the PCG solve accounts for the rest. The two methods also cost within $1.4\times$ of each other per iteration. What differs is the iteration count, which runs up to $50\times$ higher for `ichol` on the network problems (Table II).

Performance vs. BoomerAMG (HyPre)²: AMG is highly optimized for mesh-based PDE discretizations. The setup and factorization columns of Table II are methodologically comparable, since both measure end-to-end preconditioner construction. On this metric ParAC is uniformly cheaper: HyPre's setup exceeds ParAC's factorization on all 15 benchmarks, and by more than an order of magnitude on several network problems. In total time, however, HyPre is generally the faster solver. It wins on the structured PDE discretizations, and also on `GAP-road`, `europa_osm`, and `belgium_osm`, where its low iteration counts offset the higher construction cost. ParAC is substantially faster on `com-LiveJournal` ($7.2\times$), the densest matrix in the suite, and modestly faster on `aniso Poisson` (both highlighted in gray).

The advantage of ParAC over AMG therefore lies in robustness and construction cost, not in total time across the board. AMG's near-optimal convergence on structured discretizations is a property of the numerics rather than of our parallelization (Section II-B), so the structured-mesh comparison is settled in its favor by construction. Section V-C tests the robustness side of the trade directly, on SDD systems built to defeat coarsening heuristics.

The construction-cost margin also leaves headroom that we do not spend here. ParAC builds its preconditioner in a fraction of the time HyPre needs for setup, so an AC variant that invests more in construction could cut the iteration count and close the total-time gap. $AC(k)$ [17] is the natural candidate. It draws k samples per original entry, so $AC(2)$

¹`ecology1` is an ill-conditioned, near-singular discretization of an ecological PDE that causes classical IC to converge very slowly.

²We used the default parameter configuration.

TABLE II: CPU performance: ParAC vs. MATLAB `ichol` (threshold-based) and HyPre (BoomerAMG). **Pink** : iterations (randomized vs. deterministic IC). **Cyan** : factorization/setup cost. **Gray / light gray** : ParAC beats/matches HyPre.

Matrix Name	ParAC				MATLAB <code>ichol</code>				HyPre			
	Fact. Time (s)	Solve Time (s)	Total Time (s)	Iter. No.	Fact. Time (s)	Solve Time (s)	Total Time (s)	Iter. No.	Setup Time (s)	Solve Time (s)	Total Time (s)	Iter. No.
parabolic_fem	0.06	0.66	0.72	36	0.23	3.46	3.69	231	0.25	0.16	0.41	7
ecology1	0.06	1.09	1.15	42	0.13	16.12	16.25	637	0.39	0.23	0.62	7
ecology2	0.06	1.11	1.17	43	0.13	16.19	16.32	844	0.41	0.23	0.64	7
apache2	0.12	0.73	0.85	31	0.40	4.71	5.11	225	0.32	0.29	0.61	9
G3_circuit	0.11	2.19	2.30	48	0.43	9.43	9.86	222	0.68	0.48	1.16	8
uniform Poisson	2.61	16.74	19.35	30	8.96	61.62	70.58	102	9.37	5.47	14.84	8
aniso Poisson	1.18	5.59	6.77	11	5.17	4.53	9.70	7	4.22	4.66	8.88	6
contrast Poisson	1.57	75.67	77.24	142	6.70	56.35	63.05	91	8.23	5.50	13.73	8
spe16m	2.00	30.14	32.14	53	7.76	55.74	63.50	85	8.52	7.16	15.68	9
GAP-road	0.83	39.65	40.48	71	1.68	1727.31	1728.99	2285	13.28	13.39	26.67	13
com-LiveJournal	17.32	17.83	35.15	23	193.48	14.14	207.62	15	236.16	17.46	253.62	18
europa_osm	1.67	85.22	86.89	72	2.70	2752.78	2755.48	2008	31.71	33.85	65.56	15
delaunay_n24	1.10	18.69	19.79	33	5.81	1128.34	1134.15	1701	10.09	7.43	17.52	10
venturiLevel3	0.21	5.50	5.71	51	0.90	168.88	169.78	1458	1.95	1.64	3.59	9
belgium_osm	0.05	1.37	1.42	43	0.08	7.08	7.16	215	0.63	0.60	1.23	11

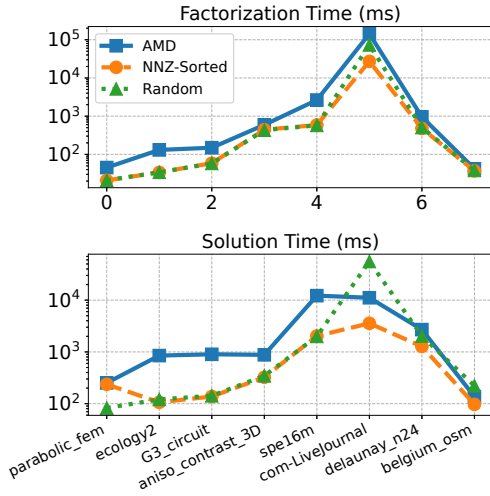


Fig. 4: GPU performance under three orderings.

yields a denser and more accurate factor than the single-sample rule that ParAC inherits from `rchol`. Whether the resulting drop in iterations repays the higher construction and application cost is untested, and we leave it to future work.

B. GPU Results

a) Impact of Ordering on GPU Performance: We repeat the ordering study on the GPU with AMD, NNZ-sorted, and random. AMD is the right choice on the CPU because it reduces fill-in most effectively, but Figure 4 shows that **NNZ-sorted ordering** consistently outperforms it on the GPU.

AMD minimizes the total number of non-zeros, but it typically yields a deep, narrow elimination tree that limits parallelism [34], [48]. Inside ParAC this appears as a longer dependency chain and smaller ready sets. NNZ-sorted and random orderings instead spread the high-degree columns, producing shallower trees with wider levels and exposing far more fine-grained parallelism. On a throughput-oriented architecture

such as the NVIDIA A100, the added concurrency is worth more than the extra arithmetic it costs. Our measurements are consistent with this account, but we do not have a formal explanation for the reversal.

The difference in memory bandwidth widens the gap further. According to the NERSC node specifications, the NVIDIA A100 (HBM2) provides roughly $8\times$ the peak bandwidth of the AMD EPYC 7763. Both the factorization and the triangular solves are bandwidth-bound, so the GPU can absorb the higher fill-in that NNZ-sorted ordering produces. We therefore recommend **NNZ-sorted** for GPU workloads: it exposes parallelism while preserving preconditioner quality. Random ordering, by contrast, degrades convergence and raises iteration counts.

b) Comparison with Baselines: We compare the wall-clock performance of ParAC against two baselines: incomplete Cholesky with zero fill-in (IC(0)) from `cuSPARSE`, and the AMG solver in the `AmgX` library. Table III reports the results.

Performance vs. `cuSPARSE` IC(0): To the best of our knowledge, no threshold-based incomplete Cholesky implementation exists for GPUs. The obstacle is that the fill-in locations are not known *a priori*, which makes the factorization hard to parallelize. This is the challenge ParAC addresses with dynamic dependency tracking. We therefore compare against `cuSPARSE` IC(0) instead.

ParAC improves total execution time on 13 of the 15 test cases. The two exceptions are `parabolic_fem`, where ParAC is about 10% slower, and `com-LiveJournal`, a relatively dense graph on which the better preconditioning quality of ParAC does not offset its higher factorization cost. ParAC reduces the iteration count on every benchmark. On `GAP-road` and `europa_osm`, IC(0) fails to reach the 10^{-6} tolerance within 10,000 iterations, whereas ParAC converges in about 100 iterations.

Performance vs. `AmgX` (through the `amgx_capi.c` interface): The multilevel framework of `AmgX` incurs a higher

TABLE III: GPU performance: ParAC vs. cuSPARSE IC(0) (zero fill-in) and AmgX. We report only a total for IC(0), since its factorization time is a negligible fraction of that total. ParAC totals include the mandatory cuSPARSE symbolic analysis. Red: $> 10^4$ iterations or OOM. Pink : iterations; cyan : factorization/setup; gray / light gray : ParAC beats/matches AmgX.

Matrix Name	ParAC				cuSPARSE IC(0)		AmgX			
	Fact. Time (ms)	Solve Time (ms)	Total Time (ms)	Iter. No.	Total Time (ms)	Iter. No.	Setup Time (ms)	Solve Time (ms)	Total Time (ms)	Iter. No.
parabolic_fem	21	237	527	40	467	923	52	16	68	10
ecology1	34	107	162	48	1146	1846	46	200	246	24
ecology2	34	106	163	49	1402	2181	74	22	96	11
apache2	49	94	177	25	723	1141	118	29	147	11
G3_circuit	59	138	227	37	1033	1019	98	33	131	11
uniform Poisson	819	1780	2937	28	5174	256	1106	162	1268	9
aniso Poisson	442	324	941	10	8551	431	361	159	520	11
contrast Poisson	545	4851	5625	127	12 544	638	516	194	710	12
spe16m	587	2028	2865	48	15 437	694	440	210	650	13
GAP-road	481	2985	3607	106	213 456	10000	438	3311	3749	32
com-LiveJournal	26 354	3698	35 224	27	3517	95	Out of Memory			
europe_osm	1040	6041	7546	104	444 754	10000	873	10 557	11 430	28
delaunay_n24	465	1420	2052	46	107 135	4555	336	503	839	13
venturiLevel3	131	373	552	54	14 756	4391	114	53	167	11
belgium_osm	39	86	144	50	4201	5432	53	807	860	28

setup cost and a larger memory footprint, and the latter becomes prohibitive within the limited memory of a GPU: AmgX ran out of memory (OOM) on com-LiveJournal. ParAC nonetheless outperforms AmgX on several instances (gray rows). On the ill-conditioned ecology1 matrix, for example, AmgX needs 24 iterations and 200 ms of solve time. That is nearly an order of magnitude more solve time than on the closely related ecology2 (11 iterations, 22 ms), and it is the reason ParAC wins on total time here. ParAC is also faster on network problems such as europe_osm and belgium_osm, because the classical and aggregation coarsening schemes in AmgX are tuned primarily for mesh-based PDE discretizations.

Effect of Matrix Density: The role of matrix density is clearest on com-LiveJournal, the densest matrix in our test suite (average degree ≈ 17 , see Table I). On the CPU, both HyPre and ichol are expensive here. HyPre spends 236 seconds in setup, because its multilevel coarsening finds little structure in a social graph, and ichol spends 194 seconds in factorization. ParAC completes the full solve in 35 seconds, roughly $6\times$ faster than ichol and $7\times$ faster than HyPre. On the GPU, the multilevel hierarchy of AmgX exhausts device memory outright (OOM). cuSPARSE IC(0) avoids this with zero fill-in and finishes in 3.5 seconds, ahead of the 35-second total for ParAC. It pays for this with $3.5\times$ more iterations (95 vs. 27), which reflects the trade-off between factorization cost and preconditioning quality on dense graphs. ParAC is therefore the only GPU solver here that both fits in device memory and keeps the iteration count low.

C. Robustness on Adversarial SDDM Graphs

To test robustness beyond the benchmark suite, we run all solvers on the 200 chimera matrices of the SDDM2023 benchmark [17]. A chimera graph is built by grafting together diverse graph families, including expanders, grids, trees, and random graphs, at shared vertices. The construction is de-

signed to stress solvers that rely on graph-structure heuristics such as AMG coarsening. The systems come from base graphs with $n = 100,000$ vertices; the boundary variants contain 97,872 unknowns after boundary removal. We run every solver to a target relative residual of 10^{-8} . The solvers do not all compute their reported residual the same way, and a recursively updated residual can drift from one recomputed directly from $\tilde{x}$. We therefore allow an order of magnitude of slack, counting a run as converged when its recorded relative residual is at most 10^{-7} and it neither crashes nor reaches the iteration cap. We test ParAC on the CPU against HyPre and on the GPU against AmgX. Table IV summarizes the results.

TABLE IV: Convergence on the 200 SDDM2023 chimera systems. “Failures” counts crashes, stagnation, and breakdowns.

Solver	Platform	Failures / 200
HyPre	CPU	15
ParAC	CPU	0
AmgX	GPU	19
ParAC	GPU	0

The HyPre failures include hard crashes (double free or corruption) on the s35 seed in all four chimera families. The rest are stagnation and factorization breakdown on further weighted and unweighted instances. AmgX fails on 19 instances, 18 of them in the weighted-Laplacian family, where its aggregation-based coarsening yields a poor hierarchy. ParAC does not build a multilevel coarsening hierarchy, and it converges on every instance in both the CPU and GPU studies. On these irregular SDD systems, ParAC converges where the AMG solvers do not.

VI. CONCLUSIONS

We introduced ParAC, a parallel execution framework for randomized incomplete Cholesky (IC) factorization. Replacing

static symbolic dependency estimation with **dynamic dependency tracking** lets `ParAC` resolve AC's stochastic sparsity pattern at runtime, which exposes more parallelism than level-based or coloring-based IC schemes.

Our evaluation on the NERSC Perlmutter supercomputer spans 15 matrices from scientific and network domains. `ParAC` scales to 32 threads on AMD EPYC 7763 CPUs and to the fine-grained parallelism of NVIDIA A100 GPUs. Against deterministic IC, end-to-end speedups reach **57.3** $\times$ on the CPU and **52.2** $\times$ on the GPU, measured on cases where the baseline itself converges. On irregular graph Laplacians where AMG's coarsening heuristics struggle, `ParAC` is up to **7.2** $\times$ faster on the CPU and **6.0** $\times$ faster on the GPU.

These results also delimit where `ParAC` belongs. AMG remains the better preconditioner for structured PDE discretizations on regular meshes, where its convergence is near-optimal. `ParAC` is the better choice on irregular graph Laplacians and general SDD systems, when device memory cannot hold a multilevel hierarchy, or when classical IC fails to converge on an irregular sparsity pattern. All three cases arise in our experiments: `AmgX` exhausted GPU memory on `com-LiveJournal`, `cuSPARSE IC(0)` reached the iteration cap on two road networks, and the AMG solvers failed on 15 and 19 of the 200 chimera systems.

Several questions remain open. A mathematical characterization of the trade-off between fill reduction and structural parallelism would put the ordering guidance of Section V-B on firmer ground, and the design of orderings that balance the two is a further question. Extending dynamic dependency tracking beyond graph Laplacians, to general SDD matrices, is a natural next step. The mechanism is not specific to Cholesky factorization either, and should carry over to other randomized sparse primitives whose structure is revealed only at runtime.

A further direction is to use `ParAC` as a parallel smoother *within* algebraic multigrid rather than as a standalone preconditioner. Recent work shows that incomplete Cholesky smoothing is effective inside a multigrid V-cycle and tolerates relatively low precision [49]. `ParAC` is breakdown-free by construction and parallelizes well, which makes it a natural candidate for that role, particularly on the irregular problems where AMG's own coarsening is least reliable. A smoother may also need far less accuracy than the spectral equivalence a preconditioner requires. How much sampling can be dropped under that weaker requirement is an open question.

ACKNOWLEDGMENT

This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research's Applied Mathematics Competitive Portfolios program under Contract No. AC02-05CH11231. We used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility, under NERSC award ASCR-ERCAP-33069. T.L. is supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 2146752. H.L. was supported in part by U.S. National Science

Foundation grant NSF-DMS 2412403. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The authors thank Shang Zhang, Steven Rennich, and Sergey Klevtsov of NVIDIA for their helpful insights.

USE OF AI ASSISTANCE

The authors used a generative AI assistant to improve the clarity, concision, and organization of the writing, and to help check the manuscript for internal consistency and correct reference metadata. The research contributions, algorithm design, implementation, experiments, and all reported results are entirely the authors' own. The authors reviewed and verified every suggestion, and take full responsibility for the content of this paper.

REFERENCES

- [1] S. Guattery and G. L. Miller, "On the performance of spectral graph partitioning methods," in *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '95. USA: Society for Industrial and Applied Mathematics, 1995, p. 233–242.
- [2] P. Humbert, B. L. Bars, L. Oudre, A. Kalogeratos, and N. Vayatis, "Learning Laplacian matrix from graph signals with sparse spectral representation," *Journal of Machine Learning Research*, vol. 22, no. 195, pp. 1–47, 2021. [Online]. Available: <http://jmlr.org/papers/v22/19-944.html>
- [3] R. K. Ando and T. Zhang, "Learning on graph with Laplacian regularization," in *Proceedings of the 20th International Conference on Neural Information Processing Systems*, ser. NIPS'06. Cambridge, MA, USA: MIT Press, 2006, p. 25–32. [Online]. Available: <https://doi.org/10.7551/mitpress/7503.003.0009>
- [4] J. Chen, Y. Saad, and Z. Zhang, "Graph coarsening: From scientific computing to machine learning," *SeMA Journal: Bulletin of the Spanish Society of Applied Mathematics*, vol. 79, no. 1, pp. 187–223, 2022. [Online]. Available: <https://doi.org/10.1007/s40324-021-00282-x>
- [5] E. G. Boman, B. Hendrickson, and S. Vavasis, "Solving elliptic finite element systems in near-linear time with support preconditioners," *SIAM Journal on Numerical Analysis*, vol. 46, no. 6, pp. 3264–3284, 2008. [Online]. Available: <https://doi.org/10.1137/040611781>
- [6] K. D. Gremban, "Combinatorial preconditioners for sparse, symmetric, diagonally dominant linear systems," Ph.D. dissertation, Carnegie Mellon University Pittsburgh, 1996.
- [7] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, "A survey of direct methods for sparse linear systems," *Acta Numerica*, vol. 25, pp. 383 – 566, 2016. [Online]. Available: <https://doi.org/10.1017/S0962492916000076>
- [8] A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall Professional Technical Reference, 1981.
- [9] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Society for Industrial and Applied Mathematics, 2003. [Online]. Available: <https://epubs.siam.org/doi/abs/10.1137/1.9780898718003>
- [10] D. S. Kershaw, "The incomplete Cholesky—conjugate gradient method for the iterative solution of systems of linear equations," *Journal of Computational Physics*, vol. 26, no. 1, pp. 43–65, 1978. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0021999178900980>
- [11] I. James W. Watts, "A conjugate gradient-truncated direct method for the iterative solution of the reservoir simulation pressure equation," *Society of Petroleum Engineers Journal*, vol. 21, no. 3, pp. 345–353, 1981. [Online]. Available: <https://doi.org/10.2118/8252-PA>
- [12] N. Munksgaard, "Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients," *ACM Transactions on Mathematical Software*, vol. 6, no. 2, pp. 206–219, 1980. [Online]. Available: <https://doi.org/10.1145/355887.355893>
- [13] M. T. Jones and P. E. Plassmann, "An improved incomplete Cholesky factorization," *ACM Transactions on Mathematical Software*, vol. 21, no. 1, pp. 5–17, 1995. [Online]. Available: <https://doi.org/10.1145/200979.200981>

- [14] T. A. Manteuffel, "An incomplete factorization technique for positive definite linear systems," *Mathematics of computation*, vol. 34, no. 150, pp. 473–497, 1980. [Online]. Available: <https://doi.org/10.1090/S0025-5718-1980-0559197-0>
- [15] R. Kyng and S. Sachdeva, "Approximate Gaussian elimination for Laplacians-fast, sparse, and simple," in *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2016, pp. 573–582. [Online]. Available: <https://doi.org/10.1109/FOCS.2016.68>
- [16] C. Chen, T. Liang, and G. Biros, "RCHOL: Randomized Cholesky factorization for solving SDD linear systems," *SIAM Journal on Scientific Computing*, vol. 43, no. 6, pp. C411–C438, 2021. [Online]. Available: <https://doi.org/10.1137/20M1380624>
- [17] Y. Gao, R. Kyng, and D. A. Spielman, "AC(k): Robust solution of Laplacian equations by randomized approximate Cholesky factorization," *SIAM Journal on Scientific Computing*, vol. 48, no. 3, pp. A1643–A1684, 2026. [Online]. Available: <https://doi.org/10.1137/24M1673577>
- [18] D. Hysom and A. Pothén, "Efficient parallel computation of ILU(k) preconditioners," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC'99)*. ACM, 1999. [Online]. Available: <https://doi.org/10.1145/331532.331561>
- [19] —, "A scalable parallel algorithm for incomplete factor preconditioning," *SIAM Journal on Scientific Computing*, vol. 22, no. 6, pp. 2194–2215, 2001. [Online]. Available: <https://doi.org/10.1137/S1064827500376193>
- [20] G. Haase, "Parallel incomplete Cholesky preconditioners based on the non-overlapping data distribution," *Parallel Computing*, vol. 24, no. 11, pp. 1685–1703, 1998. [Online]. Available: [https://doi.org/10.1016/S0167-8191\(98\)00046-5](https://doi.org/10.1016/S0167-8191(98)00046-5)
- [21] S. Doi and T. Washio, "Ordering strategies and related techniques to overcome the trade-off between parallelism and convergence in incomplete factorizations," *Parallel Computing*, vol. 25, no. 13–14, pp. 1995–2014, 1999. [Online]. Available: [https://doi.org/10.1016/S0167-8191\(99\)00064-2](https://doi.org/10.1016/S0167-8191(99)00064-2)
- [22] A. Venkat, M. S. Mohammadi, J. Park, H. Rong, R. Barik, M. M. Strout, and M. Hall, "Automating wavefront parallelization for sparse matrix computations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. IEEE Press, 2016. [Online]. Available: <https://doi.org/10.1109/SC.2016.40>
- [23] M. Naumov, "Parallel incomplete-LU and Cholesky factorization in the preconditioned iterative methods on the GPU," *Nvidia Technical Report NVR-2012-003*, 2012.
- [24] H. Anzt, T. Ribizel, G. Flegar, E. Chow, and J. Dongarra, "ParILUT - a parallel threshold ILU for GPUs," in *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 231–241. [Online]. Available: <https://doi.org/10.1109/IPDPS.2019.00033>
- [25] T. Xu, R. P. Li, and D. Osei-Kuffuor, "A two-level GPU-accelerated incomplete LU preconditioner for general sparse linear systems," *The International Journal of High Performance Computing Applications*, vol. 39, no. 3, 2025. [Online]. Available: <https://doi.org/10.1177/10943420251319334>
- [26] H. Luo and Q. Zhu, "StructILU: Dependency-preserving incomplete LU with hierarchical parallelism for structured grid pdes on GPUs," in *Proceedings of the 2025 International Conference on Supercomputing (ICS '25)*. ACM, 2025. [Online]. Available: <https://doi.org/10.1145/3721145.3725745>
- [27] M. M. Monga Made and H. A. van der Vorst, "A generalized domain decomposition paradigm for parallel incomplete LU factorization preconditionings," *Future Generation Computer Systems*, vol. 17, no. 8, pp. 925–932, 2001. [Online]. Available: [https://doi.org/10.1016/S0167-739X\(01\)00034-6](https://doi.org/10.1016/S0167-739X(01)00034-6)
- [28] —, "Parallel incomplete factorizations with pseudo-overlapped subdomains," *Parallel Computing*, vol. 27, no. 8, pp. 989–1008, 2001. [Online]. Available: [https://doi.org/10.1016/S0167-8191\(01\)00082-5](https://doi.org/10.1016/S0167-8191(01)00082-5)
- [29] E. Chow and A. Patel, "Fine-grained parallel incomplete LU factorization," *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. C169–C193, 2015. [Online]. Available: <https://doi.org/10.1137/140968896>
- [30] H. Anzt, E. Chow, and J. J. Dongarra, "ParILUT—a new parallel threshold ILU factorization," *SIAM Journal on Scientific Computing*, vol. 40, no. 4, pp. C503–C519, 2018. [Online]. Available: <https://doi.org/10.1137/16M1079506>
- [31] Y. Baumann and R. Kyng, "A framework for parallelizing approximate Gaussian elimination," in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 195–206. [Online]. Available: <https://doi.org/10.1145/3626183.3659987>
- [32] X. S. Li, "An overview of SuperLU: Algorithms, implementation, and user interface," *ACM Trans. Math. Softw.*, vol. 31, no. 3, p. 302–325, Sep. 2005. [Online]. Available: <https://doi.org/10.1145/1089014.1089017>
- [33] M. Benzi, "Preconditioning techniques for large linear systems: A survey," *Journal of Computational Physics*, vol. 182, no. 2, pp. 418–477, 2002. [Online]. Available: <https://doi.org/10.1006/jcph.2002.7176>
- [34] J. W. Liu, "The role of elimination trees in sparse factorization," *SIAM journal on matrix analysis and applications*, vol. 11, no. 1, pp. 134–172, 1990. [Online]. Available: <https://doi.org/10.1137/0611010>
- [35] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002. [Online]. Available: [https://doi.org/10.1016/S0168-9274\(01\)00115-5](https://doi.org/10.1016/S0168-9274(01)00115-5)
- [36] J. W. Ruge and K. Stüben, "Algebraic multigrid," in *Multigrid Methods*, ser. Frontiers in Applied Mathematics, S. F. McCormick, Ed. Philadelphia, PA: SIAM, 1987, vol. 3, pp. 73–130. [Online]. Available: <https://doi.org/10.1137/1.9781611971057.ch4>
- [37] P. R. Amestoy, T. A. Davis, and I. S. Duff, "Algorithm 837: AMD, an approximate minimum degree ordering algorithm," *ACM Trans. Math. Softw.*, vol. 30, no. 3, p. 381–388, Sep. 2004. [Online]. Available: <https://doi.org/10.1145/1024074.1024081>
- [38] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Philadelphia, PA: SIAM, 2006. [Online]. Available: <https://doi.org/10.1137/1.9780898718881>
- [39] D. A. Spielman, "Laplacians.jl: Algorithms inspired by graph Laplacians," <https://github.com/danspielman/Laplacians.jl>, 2020, Julia package.
- [40] D. J. Rose and R. E. Tarjan, "Algorithmic aspects of vertex elimination on directed graphs," *SIAM Journal on Applied Mathematics*, vol. 34, no. 1, pp. 176–197, 1978. [Online]. Available: <https://doi.org/10.1137/0134014>
- [41] S. C. Rennich, D. Stosic, and T. A. Davis, "Accelerating sparse Cholesky factorization on GPUs," *Parallel Computing*, vol. 59, pp. 140–150, 2016, theory and Practice of Irregular Applications. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S016781911630059X>
- [42] Z. Liu and W. Yu, "PowerRChol: Efficient power grid analysis based on fast randomized Cholesky factorization," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24, 2024, pp. 1–6. [Online]. Available: <https://doi.org/10.1145/3649329.3657308>
- [43] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: <https://doi.org/10.1145/2049662.2049663>
- [44] M. A. Christie and M. J. Blunt, "Tenth SPE comparative solution project: A comparison of upscaling techniques," *SPE Reservoir Evaluation & Engineering*, vol. 4, no. 04, pp. 308–317, 08 2001. [Online]. Available: <https://doi.org/10.2118/72469-PA>
- [45] NVIDIA Corporation, *cuSPARSE Library*, 2024, accessed: 04/01/2026. [Online]. Available: <https://docs.nvidia.com/cuda/cusparse/>
- [46] R. D. Falgout and U. M. Yang, "hypr: A library of high performance preconditioners," in *International Conference on computational science*. Springer, 2002, pp. 632–641. [Online]. Available: https://doi.org/10.1007/3-540-47789-6_66
- [47] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Regul, N. Sakharnykh et al., "AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods," *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. S602–S626, 2015. [Online]. Available: <https://doi.org/10.1137/140980260>
- [48] J. W. H. Liu, "Reordering sparse matrices for parallel elimination," *Parallel Computing*, vol. 11, no. 1, pp. 73–91, 1989. [Online]. Available: [https://doi.org/10.1016/0167-8191\(89\)90064-1](https://doi.org/10.1016/0167-8191(89)90064-1)
- [49] P. Vacek, H. Anzt, E. Carson, N. Kohl, U. Rüde, and Y.-H. Tsai, "Mixed precision multigrid with smoothing based on incomplete Cholesky factorization," *arXiv preprint arXiv:2511.04566*, 2025. [Online]. Available: <https://doi.org/10.48550/arXiv.2511.04566>