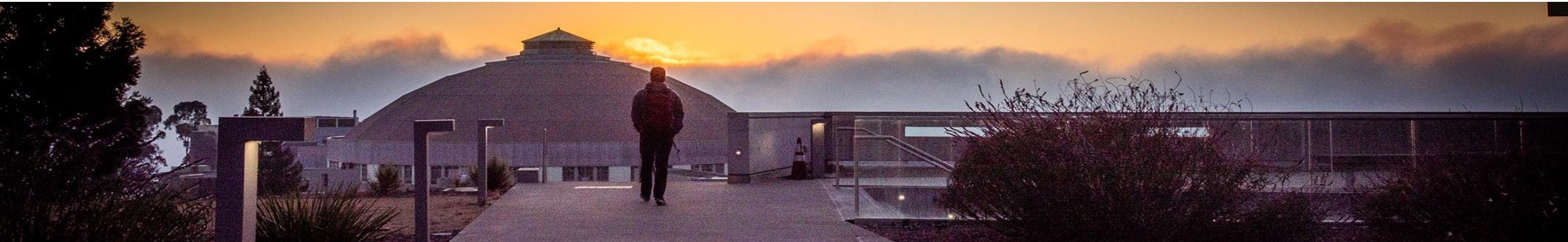
BERKELEY LAB
Bringing Science Solutions to the World

U.S. DEPARTMENT OF ENERGY
Office of Science

Parallelizing the Approximate Minimum Degree Ordering Algorithm: Strategies and Evaluation

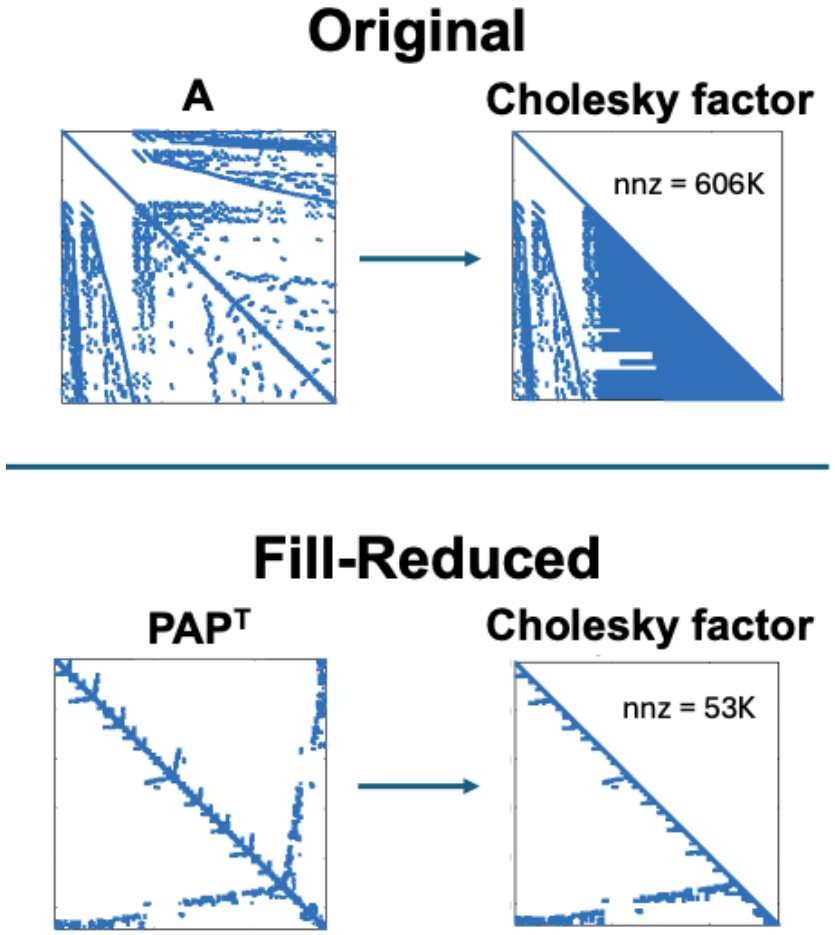
Yen-Hsiang Chang[1], Aydın Buluç[2,3], James Demmel[1]

[1]University of California, Berkeley
[2]NVIDIA Corporation
[3]work done while employed at Lawrence Berkeley National Laboratory and University of California, Berkeley



Reordering for sparse Cholesky is necessary

- **Reducing fill-in** for sparse Cholesky factorization is a cornerstone for solving sparse symmetric positive definite systems.
- Nested dissection and the **approximate minimum degree (AMD)** algorithm are the two heuristics being widely used nowadays.



Why do we even want to parallelize the AMD algorithm?

We want to make it more **future-proof** before Amdahl's law becomes prominent.

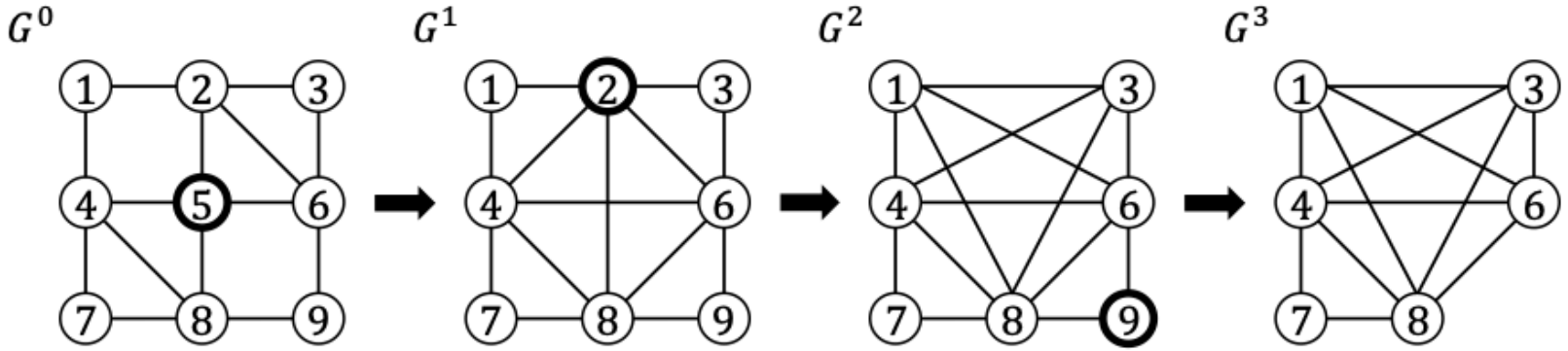
- **Closing the Gap:** While significant work has been done to parallelize nested dissection, no similar efforts have been made for the AMD algorithm.
- **Addressing the Bottleneck:** GPU Cholesky solvers have steadily improved over time, but the AMD algorithm has no improvement in the past decades.

Table 1: Time (in seconds) for SuiteSparse AMD (v7.12.1) on an AMD EPYC 7763 CPU, and for Cholesky solver cuSolverSp (v11.6.1) and its successor cuDSS (v0.7.1) on an A100 80GB GPU in double precision.

Matrix	SuiteSparse AMD	cuSolverSp	cuDSS
`nd24k`	0.82	117.17	1.97
`ldoor`	1.42	11.01	3.03
`Flan_1565`	4.85	out of memory	18.92
`Cube5317k`	13.94	out of memory	43.90

The AMD algorithm: elimination graphs

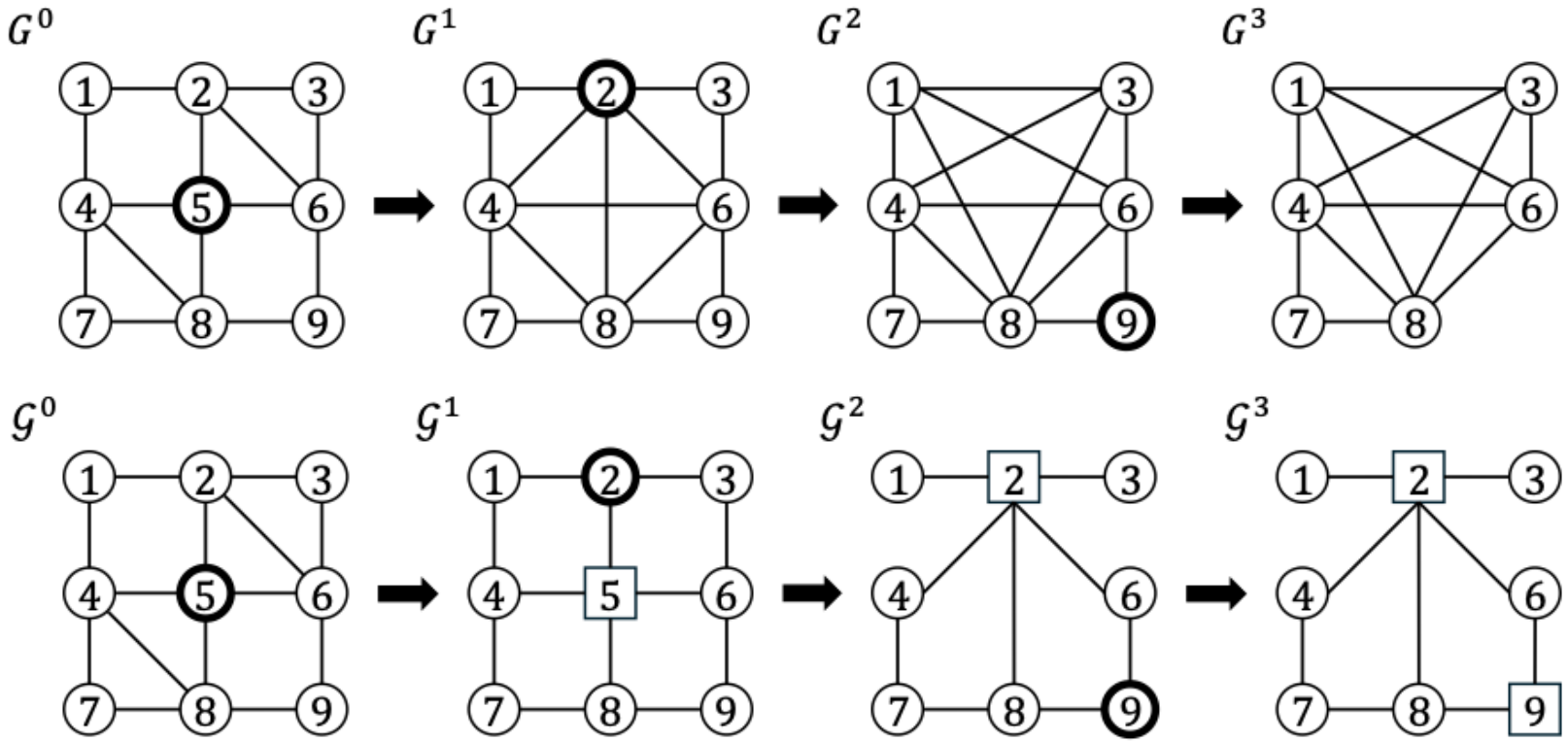
- Sparse Cholesky factorization can be represented using **elimination graphs**, where a rank-1 update is equivalent to removing a vertex and adding a clique in its neighborhood.
- To reduce fill-ins, the heuristic is to **select a minimum degree vertex as the pivot** to eliminate at each step.



The AMD algorithm: quotient graphs

Use the **invariant** $N_v = \left(\mathcal{A}_v \cup \bigcup_{e \in \mathcal{E}_v} \mathcal{L}_e\right) \setminus \{v\}$.

- To represent the sparsity pattern more efficiently, **quotient graphs** represent cliques more compactly—storing only the set of vertices in each clique.



The AMD algorithm: degree approximation

Maintaining degrees using the invariant $N_v = \left(\mathcal{A}_v \cup \bigcup_{e \in \mathcal{E}_v} \mathcal{L}_e\right) \setminus \{v\}$ is a bottleneck.

- To mitigate the **degree update bottleneck**, the AMD algorithm **estimates an upper bound** for the exact degree using three heuristics:
  - the size of the remaining submatrix
  - the worst-case fill-ins
  - union bound without double counting the neighborhood of the pivot

$$d_v^k = \min \left\{ \begin{array}{l} n - k - 1, \\ d_v^{k-1} + |\mathcal{L}_p \setminus \{v\}| - 1, \\ |\mathcal{A}_v \setminus \{v\}| + |\mathcal{L}_p \setminus \{v\}| + \sum_{e \in \mathcal{E}_v} |\mathcal{L}_e \setminus \mathcal{L}_p| \end{array} \right\}$$

Why is it hard to parallelize the AMD algorithm?

Intra-elimination parallelism is not enough!

- A straightforward approach to extract parallelism for the AMD algorithm is to parallelize using **atomic operations**. But there is **limited amount of work with high contention**.

$$d_v^k = \min \left\{ \begin{array}{l} n - k - 1, \\ d_v^{k-1} + |\mathcal{L}_p \setminus \{v\}| - 1, \\ |\mathcal{A}_v \setminus \{v\}| + |\mathcal{L}_p \setminus \{v\}| + \sum_{e \in \mathcal{E}_v} |\mathcal{L}_e \setminus \mathcal{L}_p| \end{array} \right\}$$

Algorithm 1 Computation of $|\mathcal{L}_e \setminus \mathcal{L}_p|$ for all $e \in \overline{V}$

```
1: Assume w(e) < 0 for all e ∈ V̄.
2: for each variable v ∈ L_p do
3:     for each element e ∈ E_v do
4:         if w(e) < 0 then
5:             w(e) ← |L_e|
6:         w(e) ← w(e) − 1
```

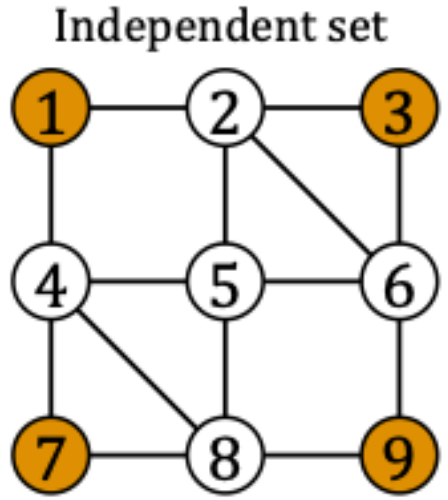
Table 2: Average sizes of the sets across all elimination steps, representing the amount of parallelism, the amount of work, and the number of unique elements accessed, respectively, by applying intra-elimination parallelism.

Matrix Name	$\lvert\mathcal{L}_p\rvert$	$\sum_{v \in \mathcal{L}_p} \lvert\mathcal{E}_v\rvert$	$\lvert\bigcup_{v \in \mathcal{L}_p} \mathcal{E}_v\rvert$
nd24k	329.7	587.5	14.0
Flan_1565	43.8	64.8	10.2
nlpkkt240	80.5	542.8	56.3

Why is it hard to parallelize the AMD algorithm?

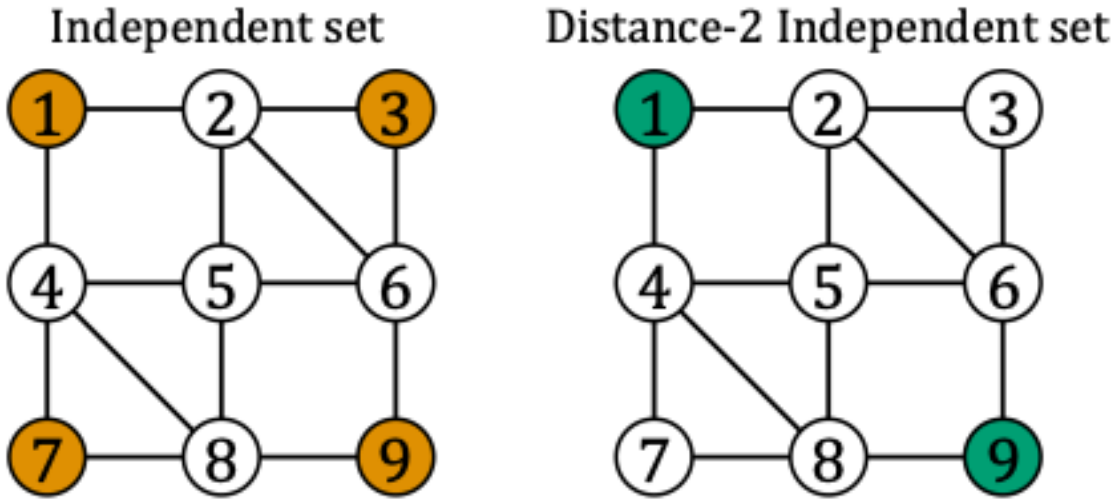
Inter-elimination parallelism is not enough either!

- The **multiple minimum degree** algorithm eliminates **a maximal independent set of pivots** with a consolidated degree update afterward.
- In the original design, performance gains from multiple elimination stemmed from the **overlap of pivots' neighborhoods**.
- However, such overlap is not well-suited to the AMD algorithm due to:
  - high contention
  - complex degree approximation



Our main contribution 1: multiple elimination via distance-2 independent sets

- To avoid high contention and complex degree approximation, we exploit **multiple elimination via distance-2 independent sets**.
- This allows each pivot to work independently in both quotient graph and degree updates.



Our main contribution 2: relaxation of the minimum degree criteria

- To get more parallelism, we **relax the minimum degree criteria by a multiplicative factor** $mult$.
- This indeed affects the reordering quality, but in a controllable manner as shown in our paper.

Table 3: Average sizes of maximal distance-2 independent sets across all elimination steps with various values of multiplicative relaxation factor, $mult$, where all variables in the set are eliminated in each step.

Matrix Name	$mult = 1.0$	$mult = 1.1$	$mult = 1.2$
nd24k	2.2	9.0	10.9
Flan_1565	42.0	448.5	678.1
nlpkkt240	57.5	4084.5	6695.8

Other contributions

- To make the full AMD algorithm parallelized, we also implemented the following techniques:
  - **Concurrent Quotient Graph Updates:** We redesigned the quotient graph data structure to concurrent updates
  - **Duplicated Degree Lists:** To mitigate memory contention during pivot selection, we duplicated the approximate degree lists.
  - **Parallel Distance-2 Independent Sets:** We used an analog of the Luby's randomized maximal independent set algorithm.
- Comprehensive implementation details are available in our paper.

Evaluation: environment and matrix suite

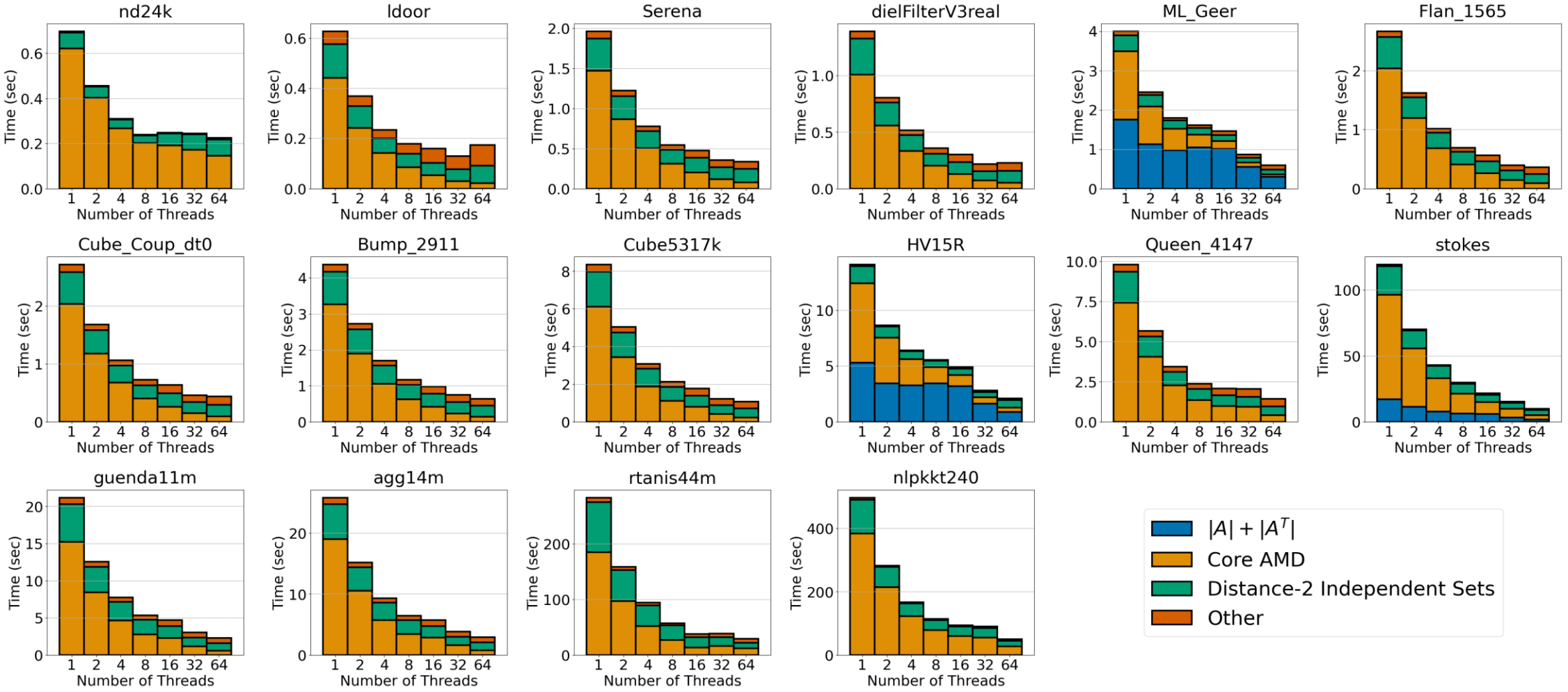
- We evaluated the performance on an AMD EPYC 7763 CPU with 64 cores.

Matrix	#rows	#nonzeros	Symmetric	Positive-definite	Description
nd24k	72.0K	28.7M	✓	✓	3D mesh problem
ldoor	952K	42.5M	✓	✓	Structural problem
Serena	1.39M	64.5M	✓	✓	Structural problem
dielFilterV3real	1.10M	89.3M	✓	×	Electromagnetic problem
ML_Geer	1.50M	111M	×	×	Poroelastic problem
Flan_1565	1.56M	114M	✓	✓	Structural problem
Cube_Coup_dt0	2.16M	124M	✓	×	Structural problem
Bump_2911	2.91M	128M	✓	✓	Geomechanical problem
Cube5317k	5.32M	223M	✓	✓	Elasticity problem
HV15R	2.02M	283M	×	×	Fluid dynamics problem
Queen_4147	4.15M	317M	✓	✓	Structural problem
stokes	11.4M	349M	×	×	Semiconductor process problem
guenda11m	11.5M	512M	✓	✓	Geomechanical problem
agg14m	14.1M	633M	✓	✓	Mesoscale problem
rtanis44m	44.8M	748M	✓	✓	Diffusion problem
nlpkkt240	28.0M	761M	✓	×	Optimization problem

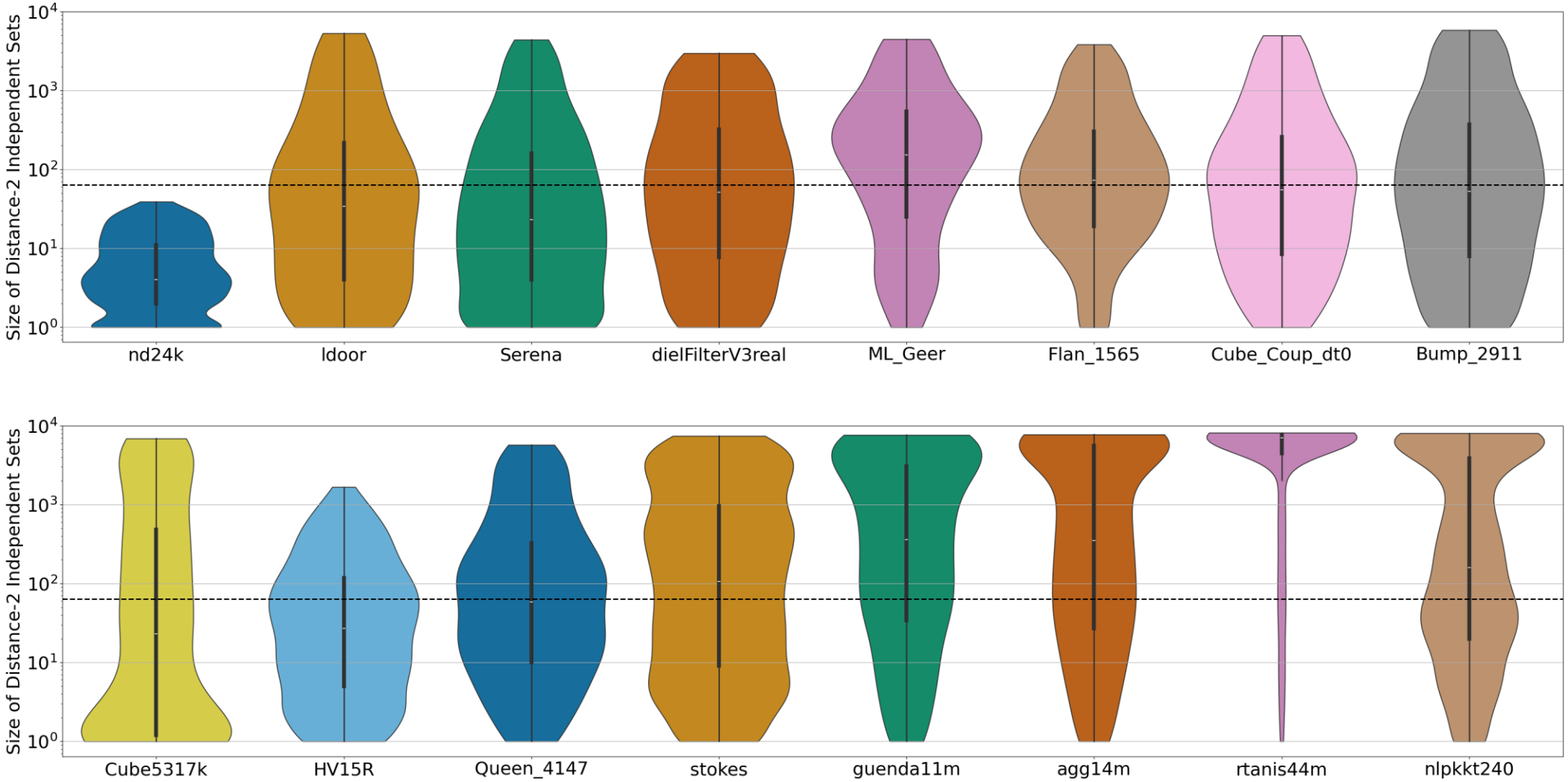
Evaluation: comparison of ordering time and quality

Matrix	Ordering Time (sec)		Our 64-thread Speedup over SuiteSparse	#Fill-ins		Our Fill-in Ratio
	SuiteSparse	Ours		SuiteSparse	Ours	
nd24k	0.82 ± 0.00	0.26 ± 0.00	$(3.18 \pm 0.04)\times$	5.03e+08	5.18e+08	1.02×
ldoor	1.42 ± 0.05	0.32 ± 0.03	$(4.39 \pm 0.24)\times$	1.52e+08	1.57e+08	1.04×
Serena	2.81 ± 0.17	0.56 ± 0.05	$(5.06 \pm 0.20)\times$	7.48e+09	8.79e+09	1.19×
dielFilterV3real	2.96 ± 0.33	0.54 ± 0.12	$(5.59 \pm 0.54)\times$	9.60e+08	1.11e+09	1.16×
ML_Geer	4.26 ± 0.48	0.77 ± 0.20	$(5.71 \pm 0.77)\times$	1.42e+09	1.46e+09	1.03×
Flan_1565	4.85 ± 0.56	0.95 ± 0.27	$(5.31 \pm 0.80)\times$	3.71e+09	3.94e+09	1.06×
Cube_Coup_dt0	5.21 ± 0.65	1.15 ± 0.30	$(4.65 \pm 0.58)\times$	2.24e+10	2.52e+10	1.15×
Bump_2911	6.78 ± 0.74	1.52 ± 0.50	$(4.76 \pm 0.91)\times$	4.27e+10	4.83e+10	1.15×
Cube5317k	13.94 ± 0.92	2.90 ± 0.67	$(4.99 \pm 0.89)\times$	6.29e+09	6.54e+09	1.04×
HV15R	14.43 ± 1.31	2.34 ± 0.33	$(6.23 \pm 0.66)\times$	5.57e+10	5.63e+10	1.01×
Queen_4147	15.45 ± 1.81	2.87 ± 0.79	$(5.61 \pm 0.82)\times$	7.70e+10	8.43e+10	1.10×
stokes	85.35 ± 6.51	14.31 ± 4.81	$(6.47 \pm 1.52)\times$	7.79e+10	8.17e+10	1.06×
guenda11m	36.59 ± 2.55	8.53 ± 2.47	$(4.64 \pm 1.27)\times$	3.10e+11	3.68e+11	1.17×
agg14m	56.10 ± 9.38	10.63 ± 2.99	$(5.62 \pm 1.38)\times$	3.75e+11	4.04e+11	1.09×
rtanis44m	195.56 ± 26.25	42.76 ± 12.06	$(4.80 \pm 0.92)\times$	1.12e+12	1.25e+12	1.13×
nlpkkt240	393.36 ± 57.62	56.31 ± 15.18	$(7.29 \pm 1.45)\times$	1.60e+12	1.82e+12	1.14×

Evaluation: time breakdown



Evaluation: sizes of distance-2 independent sets



End-to-end comparison for solving sparse SPD systems

- Solving SPD systems using cuDSS (v0.7.1) on an A100 80GB GPU in double precision.
  - **Ours** v.s. **SuiteSparse AMD**

Matrix	Ours				SuiteSparse AMD			
	Ordering Time (sec)	GPU Solver Time (sec)	Overall Time (sec)	#Fill-ins	Ordering Time (sec)	GPU Solver Time (sec)	Overall Time (sec)	#Fill-ins
`nd24k`	0.26	1.97	2.23	5.18e+08	0.82	1.97	2.79	5.03e+08
`ldoor`	0.32	3.07	3.39	1.57e+08	1.42	3.03	4.45	1.52e+08
`Flan_1565`	0.95	19.62	20.57	3.94e+09	4.85	18.92	23.77	3.71e+09
`Cube5317k`	2.90	44.33	47.23	6.54e+09	13.94	43.90	57.84	6.29e+09

End-to-end comparison for solving sparse SPD systems

- Solving SPD systems using cuDSS (v0.7.1) on an A100 80GB GPU in double precision.
  - **Ours** v.s. **cuDSS ND (multi-threaded)**

Matrix	Ours				cuDSS ND (multi-threaded)			
	Ordering Time (sec)	GPU Solver Time (sec)	Overall Time (sec)	#Fill-ins	Ordering Time (sec)	GPU Solver Time (sec)	Overall Time (sec)	#Fill-ins
`nd24k`	0.26	1.97	2.23	5.18e+08	2.38	1.27	3.65	3.21e+08
`ldoor`	0.32	3.07	3.39	1.57e+08	0.67	2.48	3.15	1.41e+08
`Flan_1565`	0.95	19.62	20.57	3.94e+09	2.41	9.21	11.62	1.46e+09
`Cube5317k`	2.90	44.33	47.23	6.54e+09	7.43	33.45	40.88	4.02e+09

Conclusion and future work

- **Summary of Contributions:** We developed a scalable AMD implementation by leveraging **distance-2 independent sets** and designing specialized **concurrent data structures** tailored to the AMD algorithm.
- **Future Directions:**
  - Exploring **hybrid approaches** that combine intra- and inter-elimination parallelism.
  - Investigating the **fusion of AMD and nested dissection** for enhanced ordering efficiency.
  - Evaluating **more matrices, ordering methods, and solvers**.

Q&A

https://epubs.siam.org/doi/epdf/10.1137/1.9781611979022.1