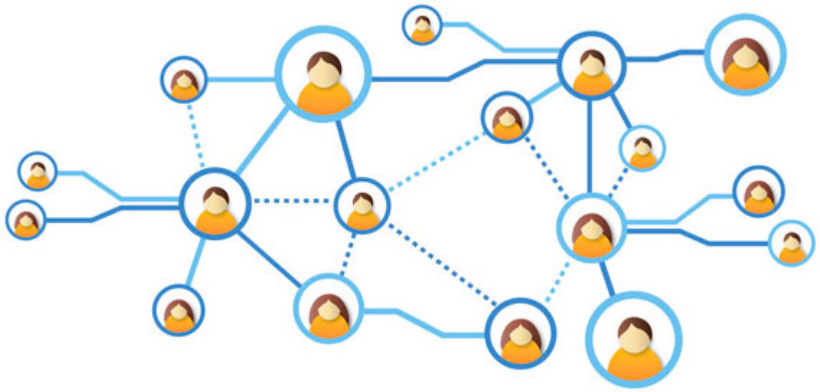
Parallel clique counting and peeling algorithms

Jessica Shi (MIT CSAIL)

Laxman Dhulipala (MIT CSAIL)

Julian Shun (MIT CSAIL)

Graph processing

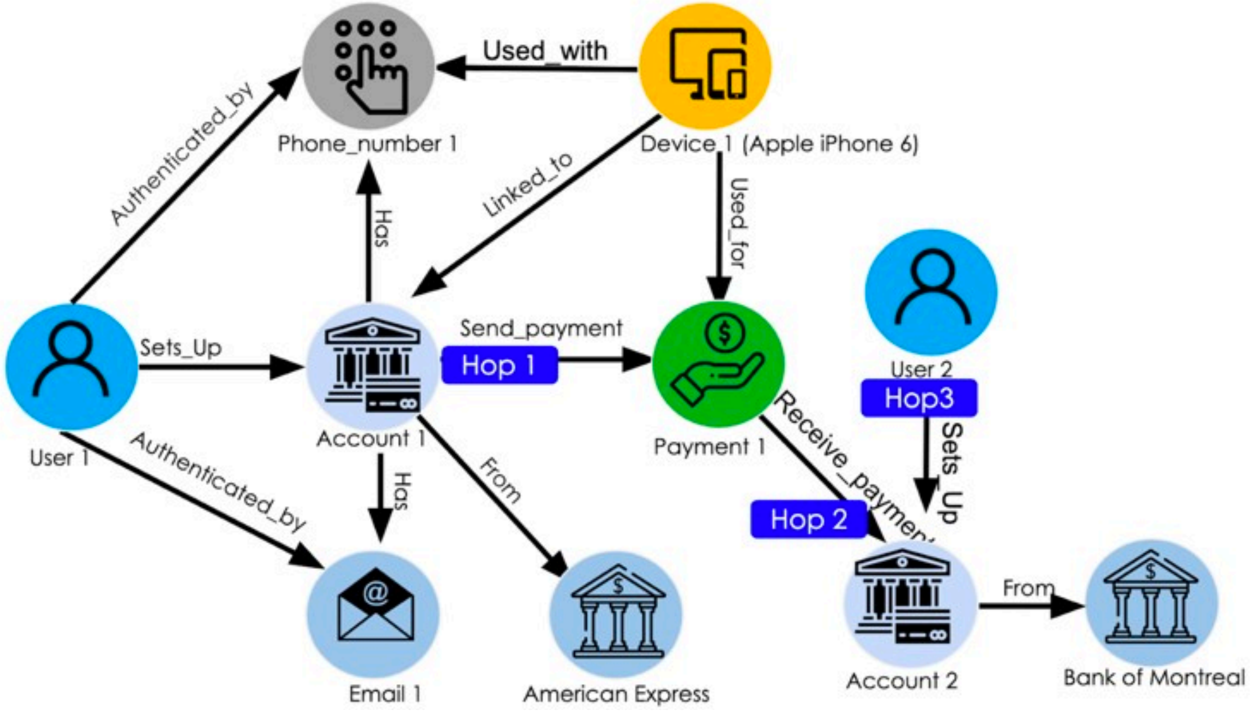


Social Network

https://blog.soton.ac.uk/skillted/2015/04/05/graph-theory-for-skillted/

Road Network

Data-driven Modeling of Transportation Systems and Traffic Data Analysis During a Major Power Outage in the Netherlands



Financial Transactions

https://www.rtinsights.com/how-the-worlds-largest-banks-use-advanced-graph-analytics-to-fight-fraud/

Parallelism

- Parallelism enables us to efficiently process large graphs

Apple, Microsoft, Intel, https://www.flickr.com/photos/66016217@N00/2556707493/, HP

Finding dense subgraphs

- Problem: Given a graph G, find the k-clique densest subgraph [1]
  - Subgraph that maximizes (# induced k-cliques) / (# vertices)
- Applications:
  - Community detection in social networks [2]
  - Link-spam detection in web graphs [3]
  - Motif detection in biological networks [4]

[1] Tsourakakis (15)
[2] Angel, Sarkas, Koudas, Srivastava (12)
[3] Gibson, Kumar, Tomkins (05)
[4] Bader, Hogue (03)

Main results

- Main goal: Develop efficient exact and approximate algorithms for k-clique counting and peeling

- New parallel algorithms for k-clique counting
- Comprehensive evaluation
  - Outperforms fastest parallel algorithms [1, 2] by up to 10x
  - Up to 39x self-relative speedups
  - Compute 4-clique counts on largest publicly-available graph with > 200 billion edges

[1] Danisch, Balalau, Sozio (18)
[2] Jain, Seshadhri (20)

Main results

- Main goal: Develop efficient exact and approximate algorithms for k-clique counting and peeling

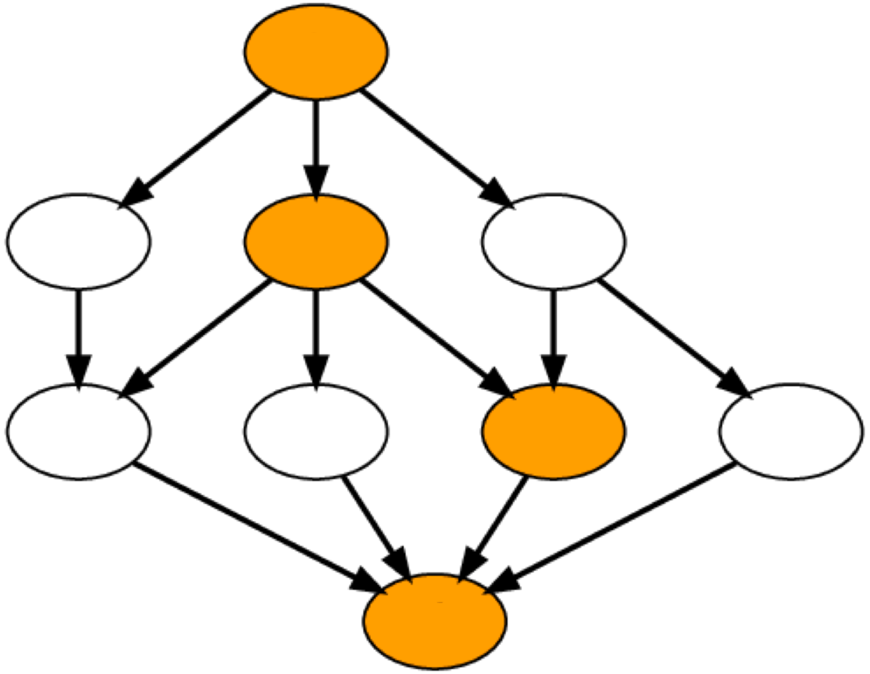
- New parallel algorithms for k-clique peeling
- Comprehensive evaluation
  - Outperforms fastest sequential algorithms [1] by up to 12x
  - Up to 14x self-relative speedups

[1] Danisch, Balalau, Sozio (18)

Important paradigms

- Strong theoretical bounds
  - Work = total # operations = # vertices in graph
  - Span = longest dependency path = longest directed path
  - Running time ≤ (work / # processors) + O(span)
  - Work-efficient = work matches sequential time complexity

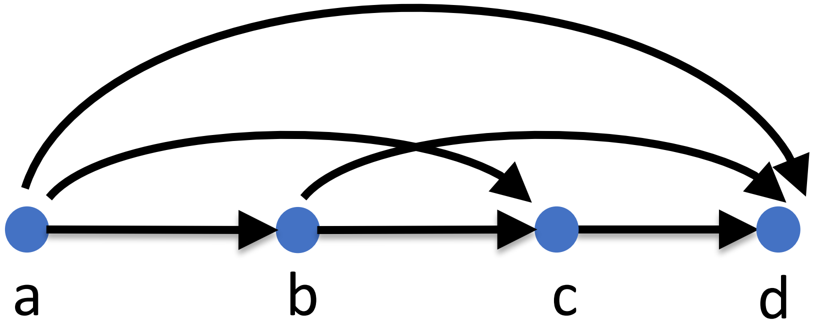
Parallel computation graph



https://www.researchgate.net/figure/Task-dependency-graph-each-node-contains-the-task-time-and-the-highlighted-tasks-form_fig1_320678407

k-clique counting components

- Obtain a total ordering of vertices
  - Non-increasing degree order [1]
  - Ordering given by k-core algorithm [2]
- Orient edges from vertices lower in the ordering to vertices higher in the ordering
- Count unique k-cliques starting from lowest vertex in ordering



[1] Chiba, Nishizeki (85)
[2] Danisch, Balalau, Sozio (18)

Graph orientation

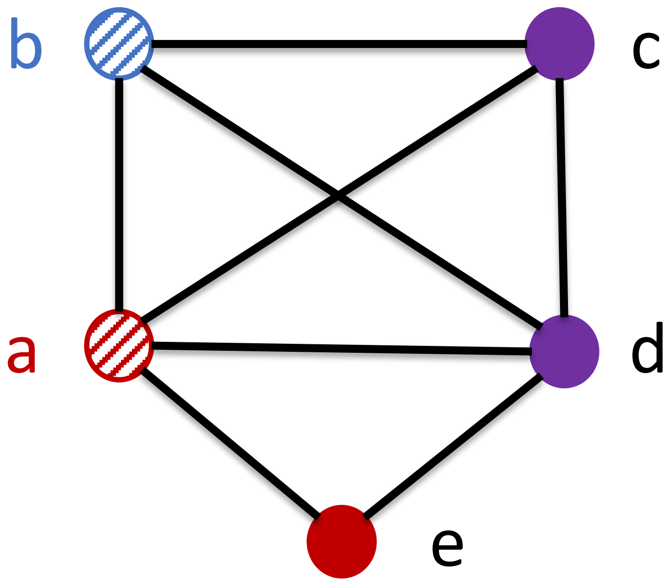
- c-orientation: Direct graph such that each vertex’s out-degree is upper bounded by c
- Arboricity orientation: O(α)-orientation
  - α = arboricity/degeneracy (O($\sqrt{m}$))
  - m = # edges

- Our work: Two arboricity orientation algorithms in O(m) work, O($\log^2$ n) span

Parallel k-clique counting algorithm

How do we find cliques?

- A clique is found by repeatedly intersecting the neighborhoods of vertices



▨ = initial vertices in 2-clique

N(a) = {b, c, d, e}

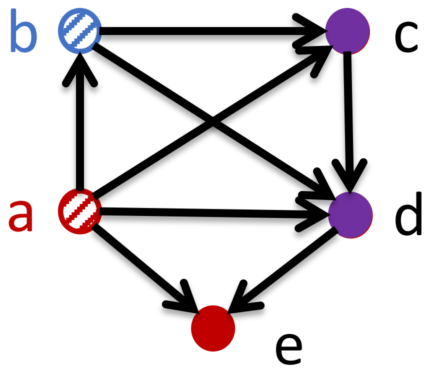
N(b) = {a, c, d}

Intersection = {c, d}

Two 3-cliques incident to {a, b}

Main idea

- Recursive subroutine:
  - S = set of vertices to consider in clique (initially V)
  - If it is the $(k - 1)^{th}$ recursive level, return |S| (number of k-cliques)
  - Parallel for each v in S: (v is added to the clique)
    - S' = intersection of S with arboricity-directed neighbors of v
    - Recurse on S'



▨ = clique

v = b

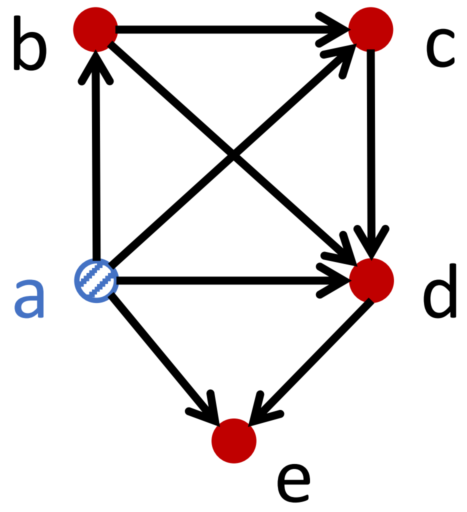
S = {c, d, e}

S' = {c, d}

Counting 4-cliques

Consider only vertices in the intersection of the neighborhood of the clique

Level 1



v = a

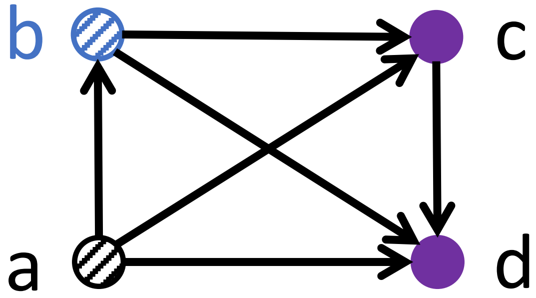
S = {b, c, d, e}

▨ = clique

Level 2

Clique = {a, b}

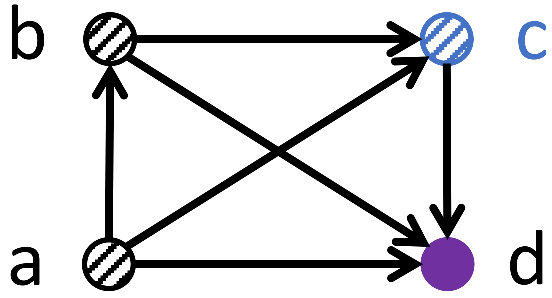
S’ = {c, d}



Level 3

Clique = {a, b, c}

S’ = {d}

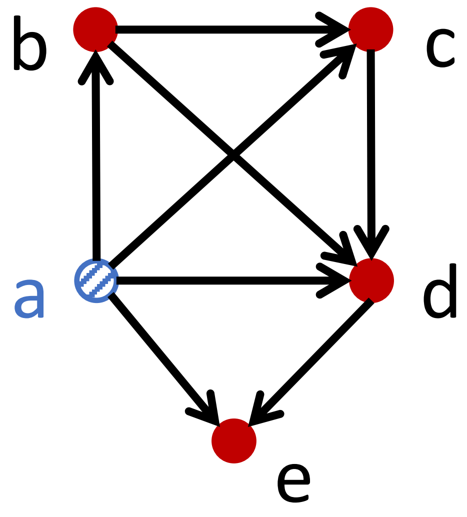


+ 1 four-clique

Counting 4-cliques

Consider only vertices in the intersection of the neighborhood of the clique

Level 1



v = a

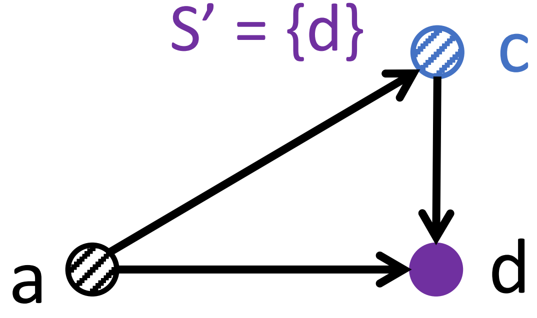
S = {b, c, d, e}

⊘ = clique

Level 2

Clique = {a, c}

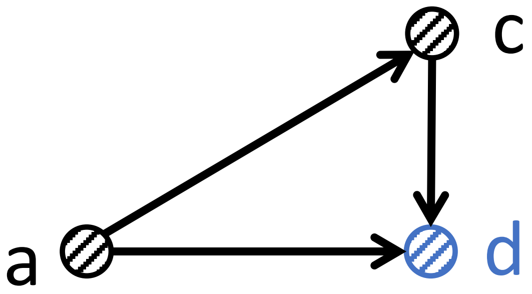
S’ = {d}



Level 3

Clique = {a, c, d}

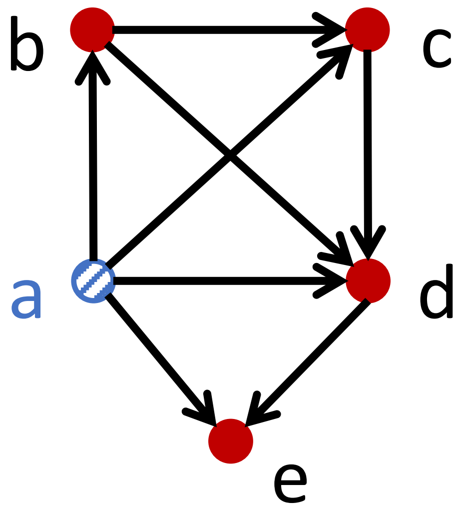
S’ = {}



+ 0 four-clique

Counting 4-cliques

- 1 four-clique on vertex a



- At each level, only store S = the set of vertices in the intersection of the neighborhood of the clique

Theoretical bounds

Recall S = Set of neighbors of clique under construction

- Arboricity orientation: $O(m)$ work, $O(\log^2 n)$ span
- Iterating through each v in S: $O(m)$ work over the first two recursive levels, multiply by α for subsequent recursive levels
- Intersecting S with arboricity-directed neighbors of v: $O(\alpha)$ work, $O(\log n)$ span whp

Total = $O(m\alpha^{k-2})$ work, $O(k \log n + \log^2 n)$ span whp

- Arboricity orientation: $O(m)$ space
- Storing S per recursive level: $O(P\alpha)$ space where P = # processors

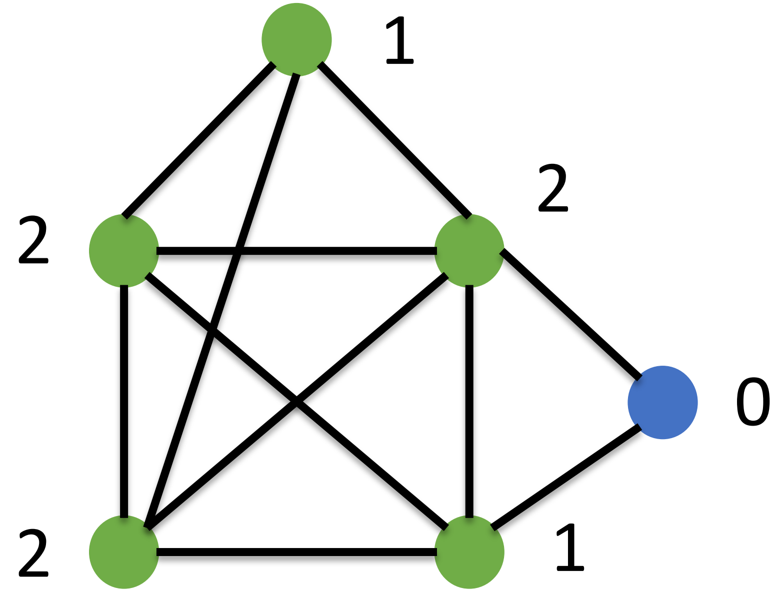
Total = $O(m + P\alpha)$ space

Parallel k-clique peeling algorithm

k-clique densest subgraph problem

- k-clique densest subgraph: Subgraph that maximizes (# induced k-cliques) / (# vertices)

- k-clique peeling: Gives a 1/k approximation to the k-clique densest subgraph problem[1]



2 four-cliques /
5 vertices

[1] Tsourakakis (15)

How do we peel k-cliques?

- Goal: Iteratively remove all vertices with min k-clique count

Subgoal 1: A way to keep track of vertices with min k-clique count
Subgoal 2: A way to update k-clique counts after peeling vertices

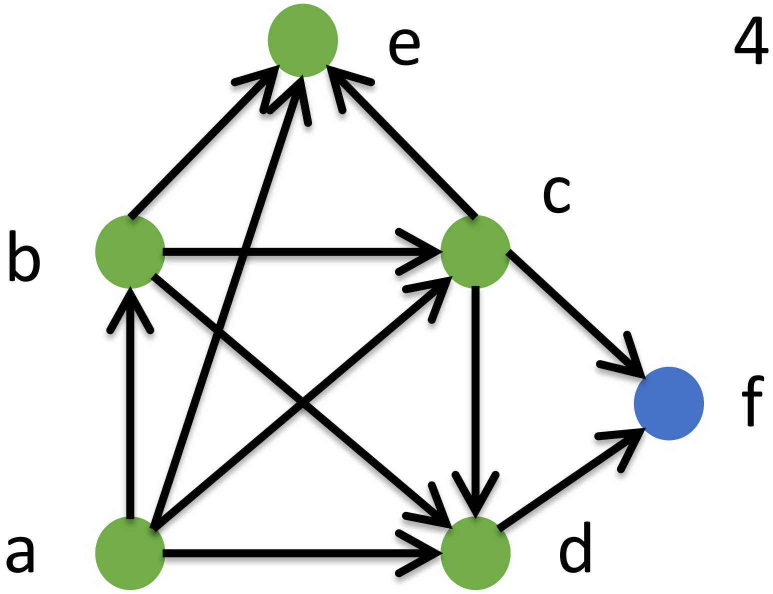
For subgoal 1: Use a work-efficient batch-parallel Fibonacci heap which supports batch insertions/decrease-keys (Shi and Shun, 2020)

For subgoal 2: Reuse counting algorithm

Main Idea

- Let B be our Fibonacci heap mapping vertices to # of k-cliques
- While not all vertices have been peeled:
  - Peel subset A of vertices with min k-clique count (using B)
  - Call recursive subroutine for each vertex v in A, with S = undirected neighbors of v
  - Update k-clique counts of incident vertices that have not been peeled

4-clique peeling example



Fibonacci Heap B:

4-clique count:	0	1		2		
Vertices:	f	d	e	a	b	c

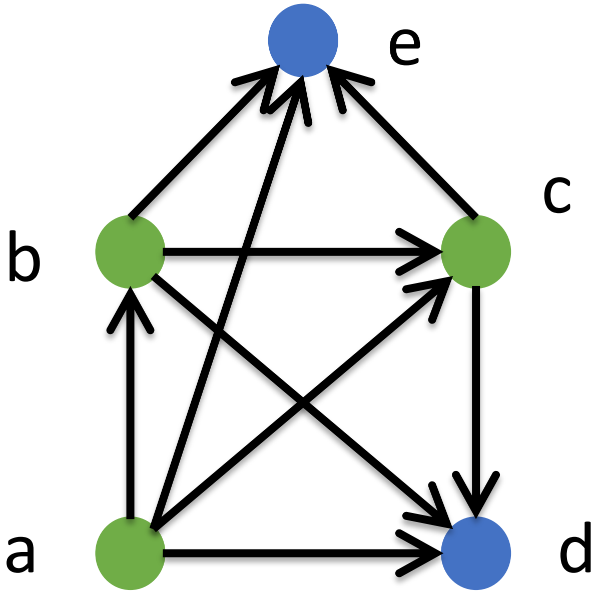
$S_f = \{c, d\}$

No 4-cliques on f

● = vertices to peel in this round

4-clique density: 0.25

4-clique peeling example



Fibonacci Heap B:

4-clique count:	1		2		
Vertices:	d	e	a	b	c

$S_d = \{a, b, c\}$

One 4-clique on d

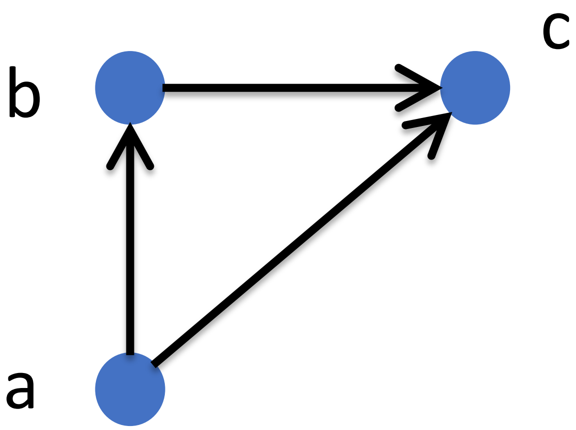
$S_e = \{a, b, c\}$

One 4-clique on e

● = vertices to peel in this round

4-clique density: 0.4

4-clique peeling example



Fibonacci Heap B:

4-clique count:	0		
Vertices:	a	b	c

No 4-cliques remaining

● = vertices to peel in this round

4-clique density: 0

Theoretical bounds

- Because S = **undirected** neighbors of v, we no longer have O(α)
- Nash-Williams Theorem: For every subgraph G’,

$$\alpha \geq {E(G')}/_{(V(G')-1)}$$

- The first level of recursion on S = N(v) is equivalent to
  - Constructing the induced subgraph of N(v) on G
  - Performing (k - 1)-clique counting

Theoretical bounds

- ρ_k: Number of peeling rounds necessary to completely peel G
- P-completeness result: k-clique peeling solves a P-complete problem

Total = $O(m\alpha^{k-2} + \rho_k \log n)$ amortized expected work,
$O(\rho_k\ k \log n + \log^2 n)$ span whp

We provide details in the paper

Evaluation

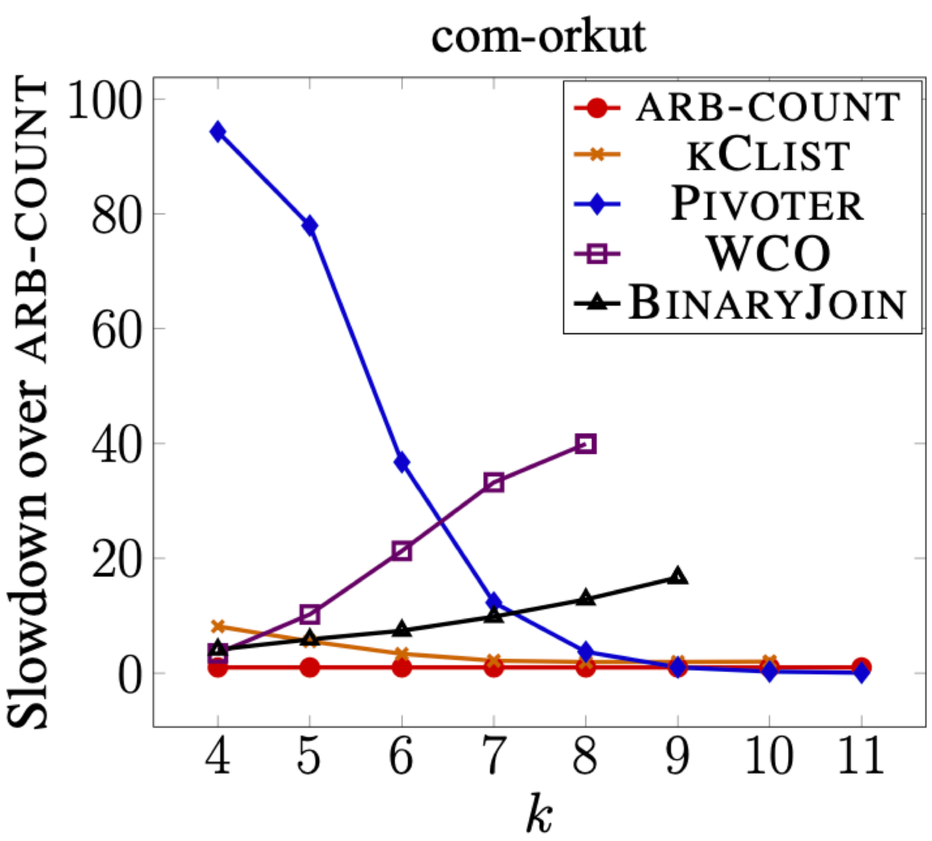
Environment

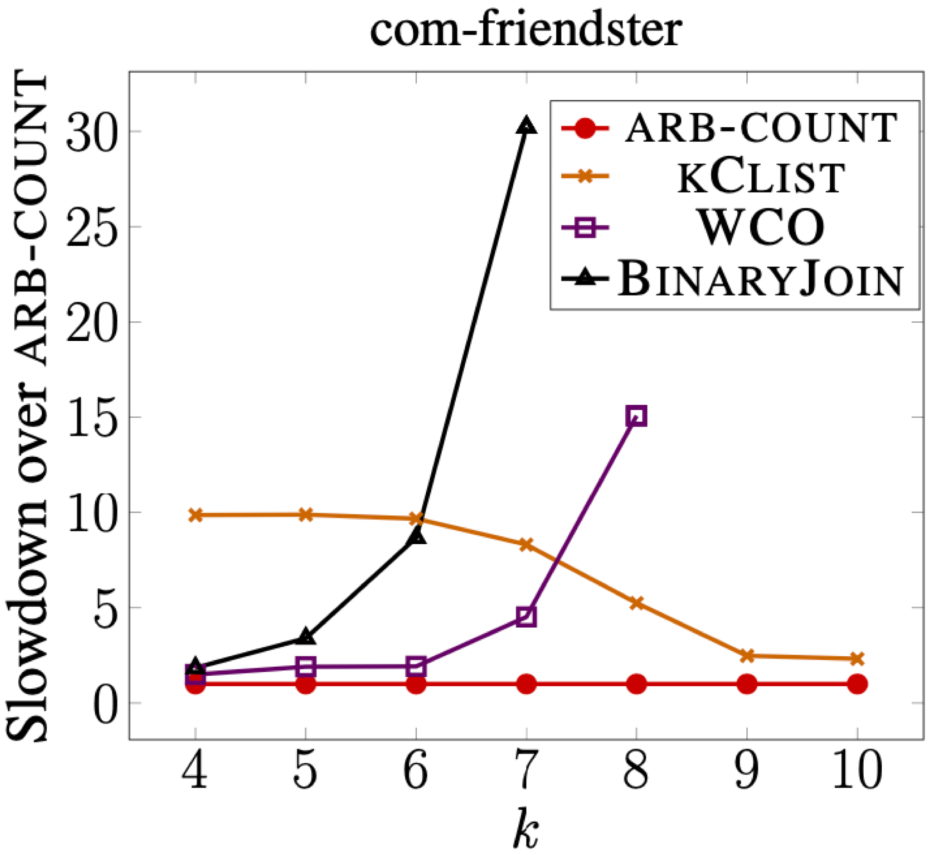
- 30-core GCP instance (2-way hyper-threading), 240 GiB main memory
- Real-world Stanford Network Analysis Platform (SNAP) graphs
- Use bucketing implementation from Julienne [1] instead of Fibonacci heap

Graph	# Vertices	# Edges
dblp	425957	2.10×10^6
skitter	1.70×10^6	1.11×10^7
lj	4.03×10^6	6.94×10^7
orkut	3.27×10^6	2.34×10^8
friendster	1.25×10^8	3.61×10^9

[1] Dhulipala, Blelloch, and Shun (17)

Comparison to other implementations (counting)





KClist: Danisch, Balalau, Sozio (18)
Pivoter: Jain, Seshadhri (20)
WCO: Mhedhbi, Salihoglu (19)
BinaryJoin: Lai et al. (19)

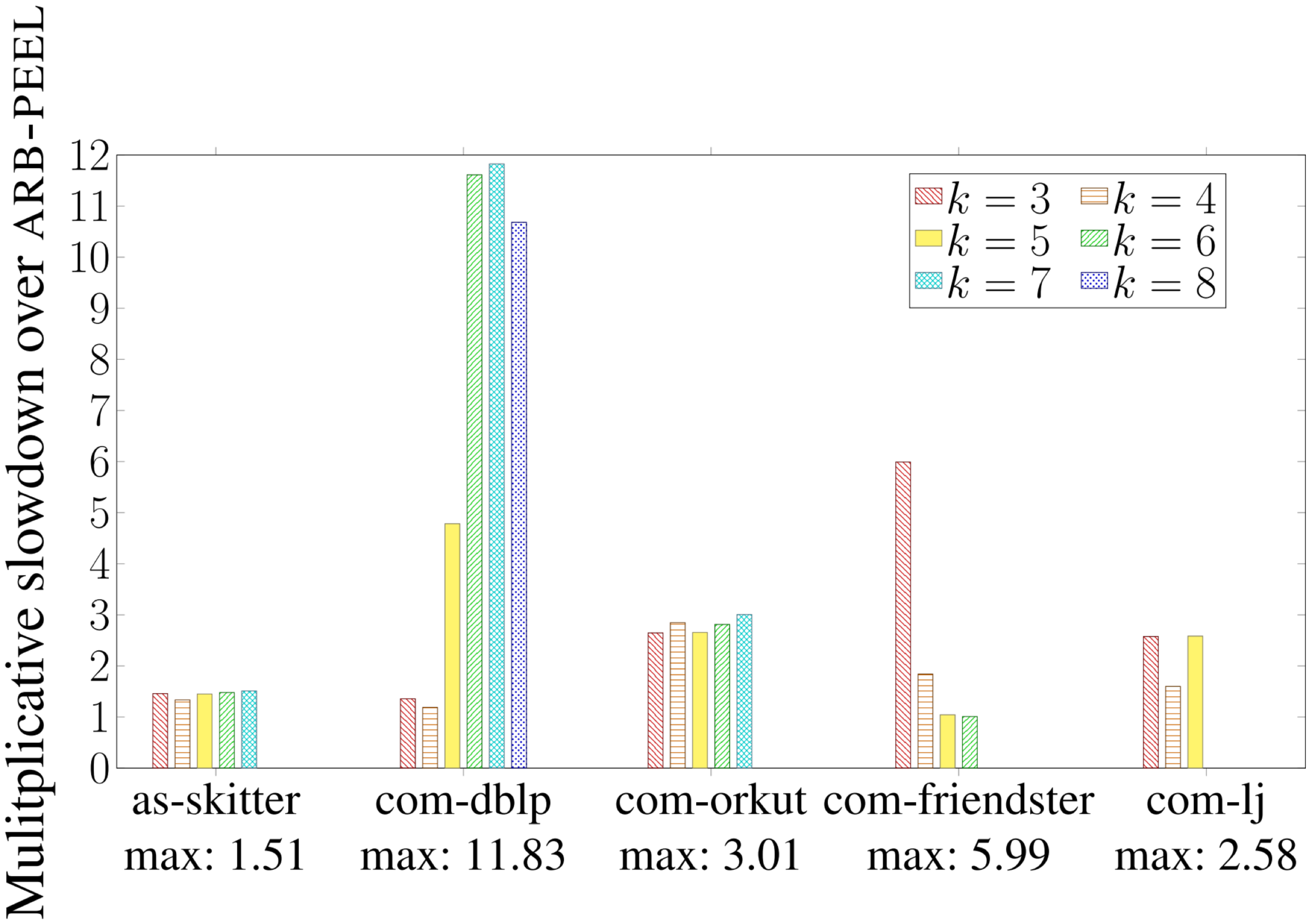
Evaluation (counting)

- Up to 9.88x speedups over parallel KClist
- Up to 79.20x speedups over serial KClist
- Up to 196.28x speedups over parallel Pivoter
  - Pivoter is faster: $k \geq 8$ on skitter, dblp, $k \geq 10$ on orkut

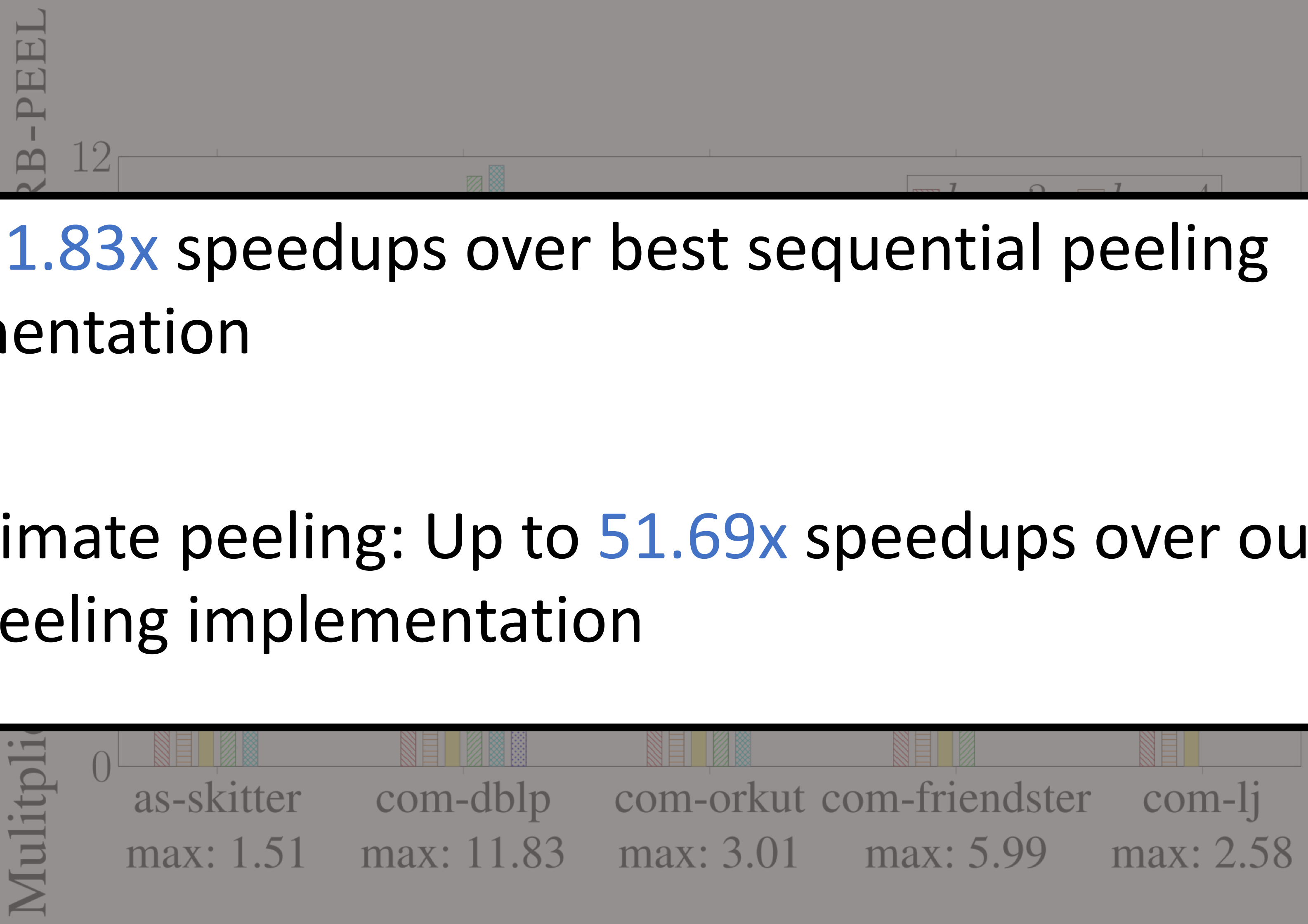
First to obtain 4-clique counts on:

ClueWeb (74 billion edges) in < 2 hours
Hyperlink2014 (~100 billion edges) in < 4 hours
Hyperlink2012 (~200 billion edges) in < 45 hours

Slowdown of serial KClist (peeling)



Slowdown of serial KClist (peeling)



- Up to 11.83x speedups over best sequential peeling implementation
- Approximate peeling: Up to 51.69x speedups over our parallel exact peeling implementation

Conclusion

Conclusion

- First work-efficient parallel algorithms for k-clique counting in polylogarithmic depth
- First work-efficient parallel algorithms for k-clique peeling
- k-clique counting scales to largest publicly available graphs
- Additional approximate k-clique counting and peeling results in paper

- Github: https://github.com/ParAlg/gbbs/tree/master/benchmarks/CliqueCounting

Thank you