Parallel algorithms for butterfly + clique computations

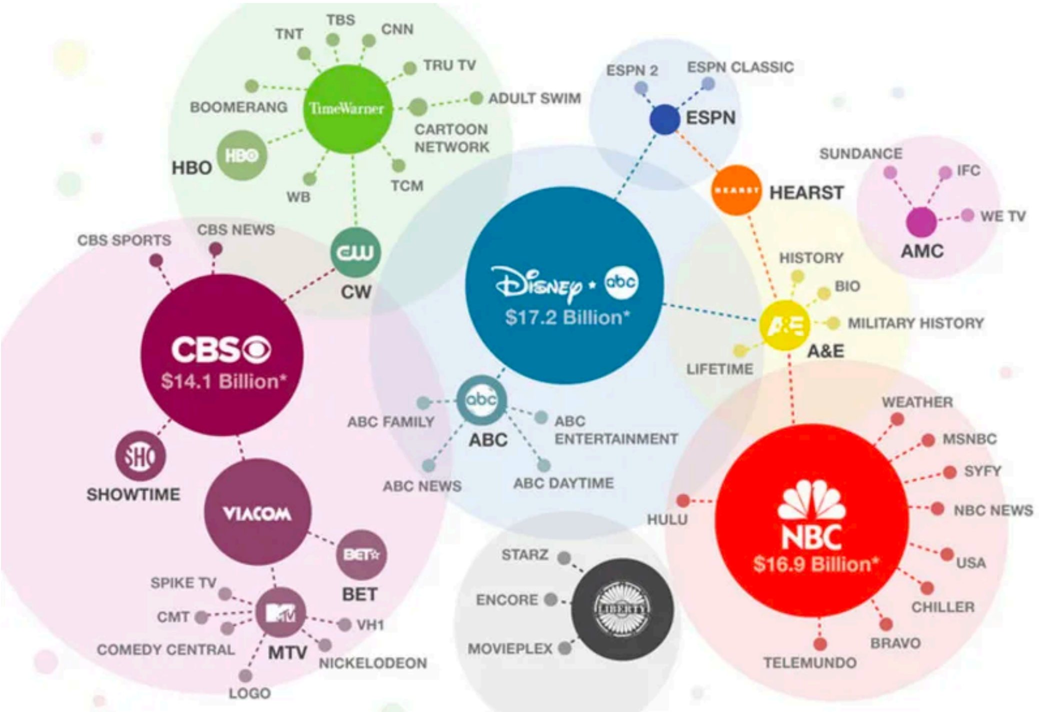
Jessica Shi (MIT CSAIL)
Laxman Dhulipala (MIT CSAIL)
Julian Shun (MIT CSAIL)

Outline

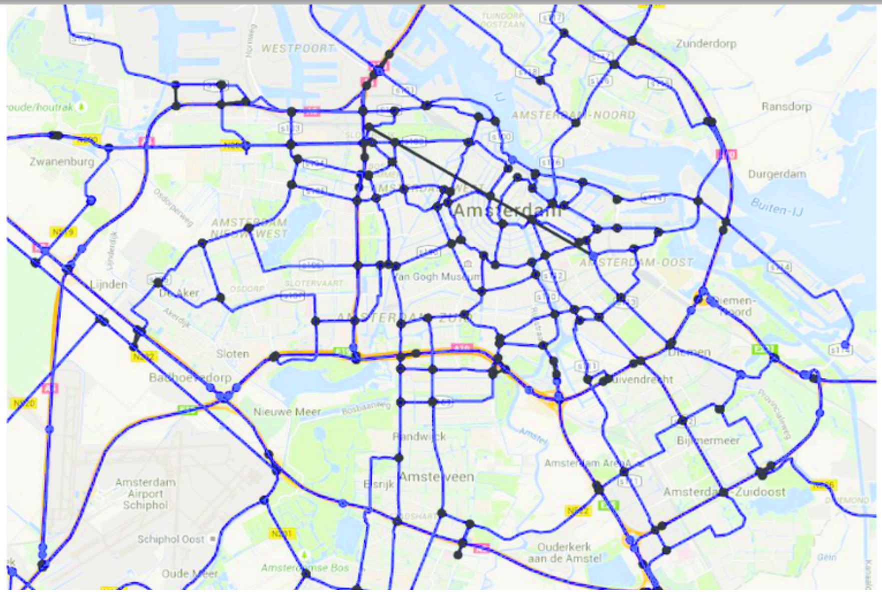
- Problem statement + Applications
- ParButterfly framework
  - Parallel butterfly counting
  - Parallel butterfly peeling
- Implementation + evaluation of ParButterfly
- Parallel clique counting framework
  - Parallel clique counting
  - Parallel clique peeling
- Implementation + evaluation of parallel clique counting + peeling
- Conclusion + Future work

Graph processing

- Graphs are ubiquitous



https://gizmodo.com/fascinating-graphic-shows-who-owns-all-the-major-brands-1599537576

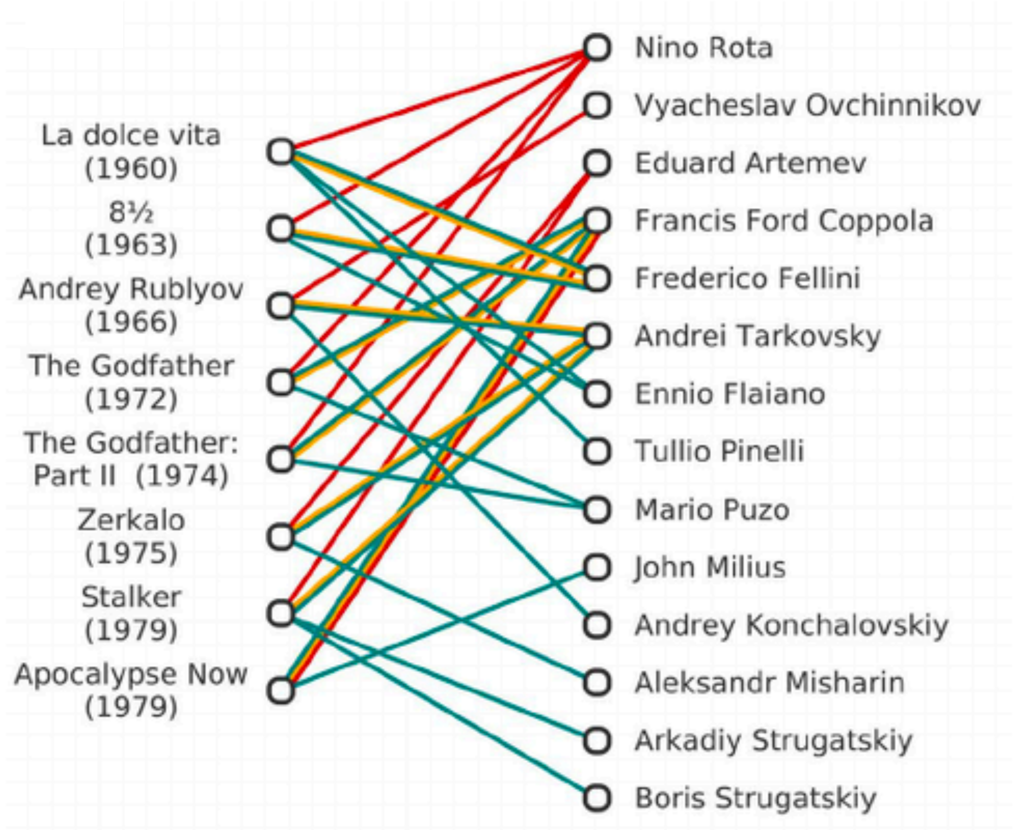


Data-driven Modeling of Transportation Systems and Traffic Data Analysis During a Major Power Outage in the Netherlands

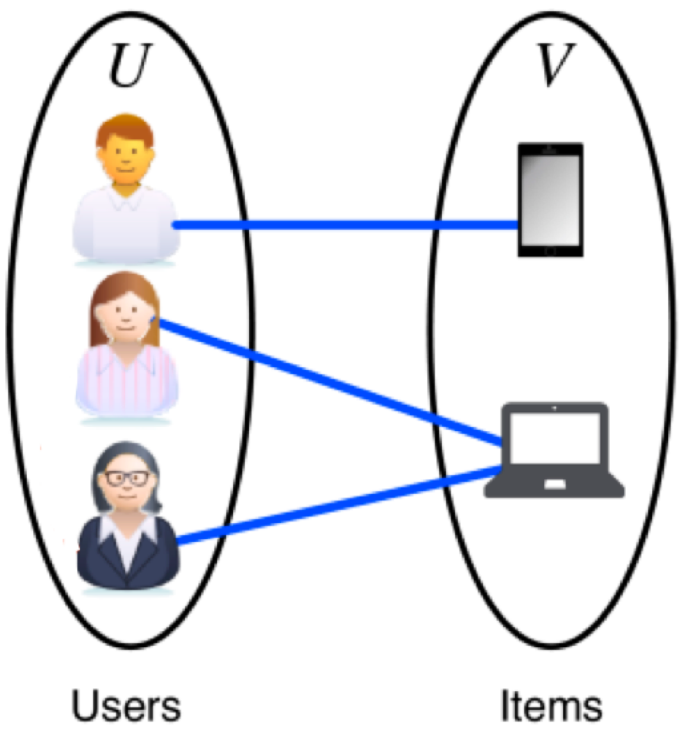
http://bitcoinwiki.co/wp-content/uploads/censorship-free-social-network-akasha-aims-to-tackle-internet-censorship-with-blockchain-technology.jpg

Bipartite graphs

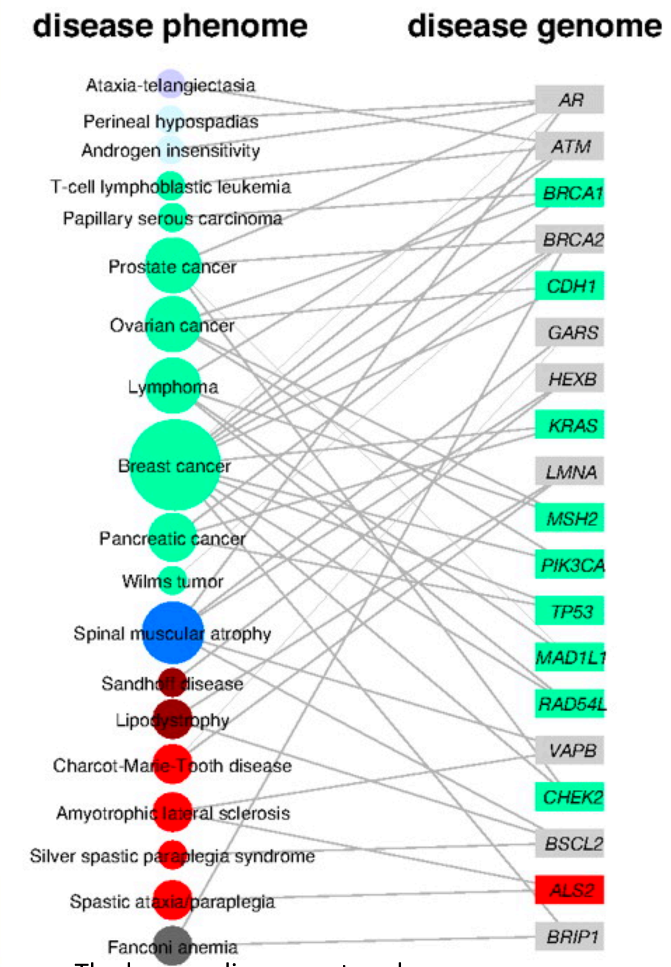
- Bipartite graphs: Represent relationships between two groups



Measuring Long-Term Impact Based on Network Centrality: Unraveling Cinematic Citations



Bipartite Graph Neural Networks for Efficient Node Representation Learning



The human disease network

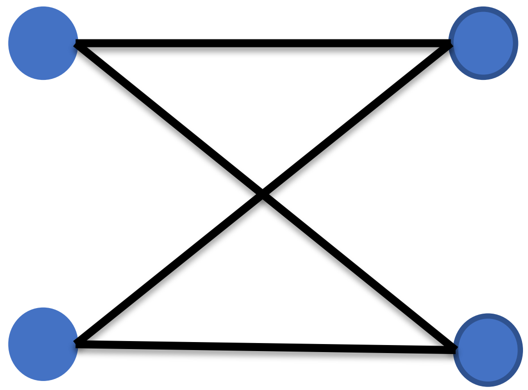
Parallelism

- Parallelism enables us to efficiently process large graphs

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

Bipartite graphs

- Butterflies = 4-cycles = $K_{2,2}$



Think of these as the bipartite analogue of triangles (K_3)

Note: Bipartite graphs contain no triangles

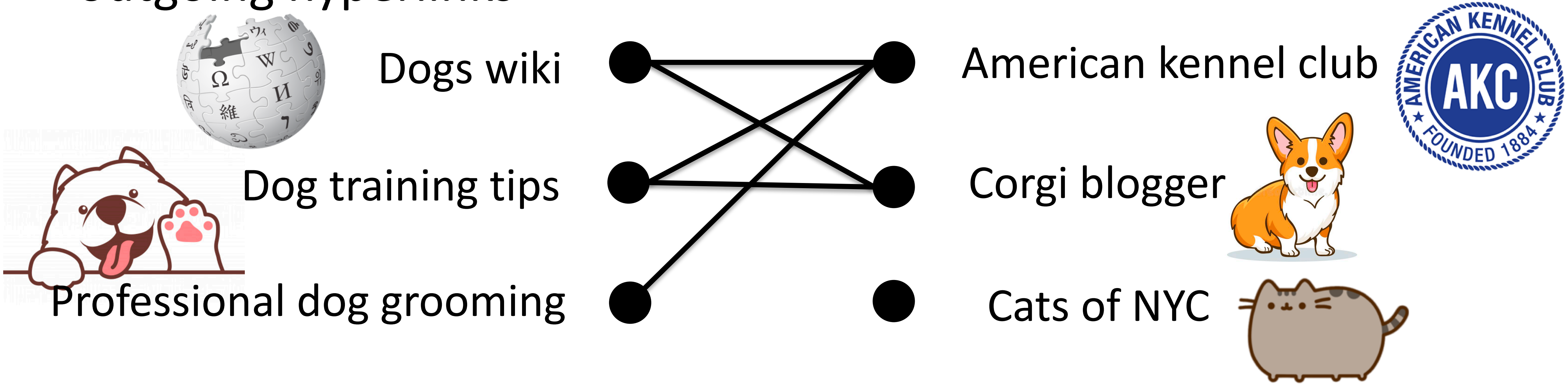
Finding dense subgraphs

- Problem: Given a graph G, find dense (bipartite) subgraphs

- Applications:
  - Find communities in social networks, websites, etc.
  - Discovering protein interactions in computational biology
  - Fraud detection in finance (tampered derivatives)

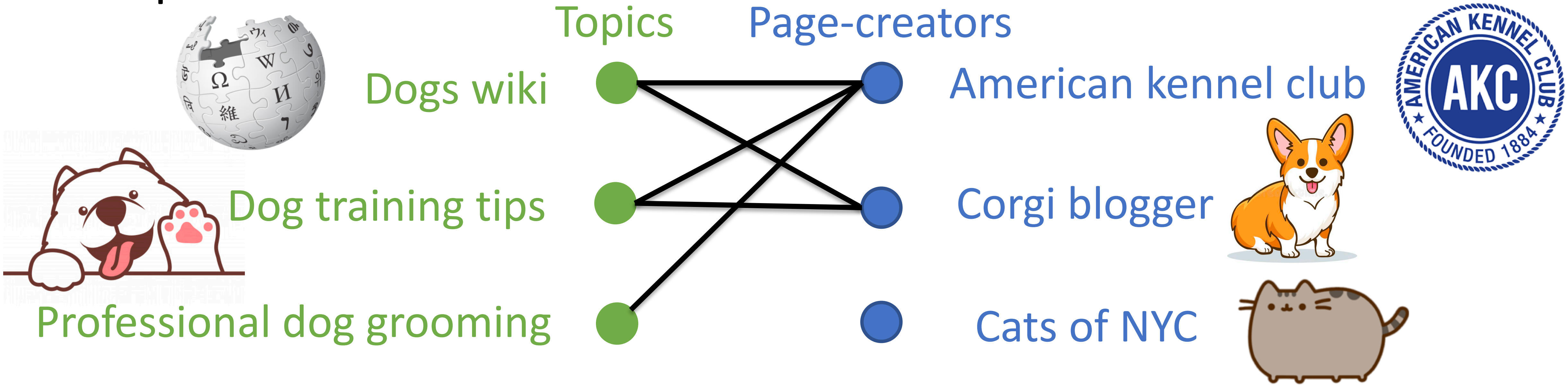
Link spam detection

- Link spam: Create many external links to a spam page, for web search ranking promotion
- Link graph: Webpages are nodes, connected by incoming / outgoing hyperlinks



Link spam detection

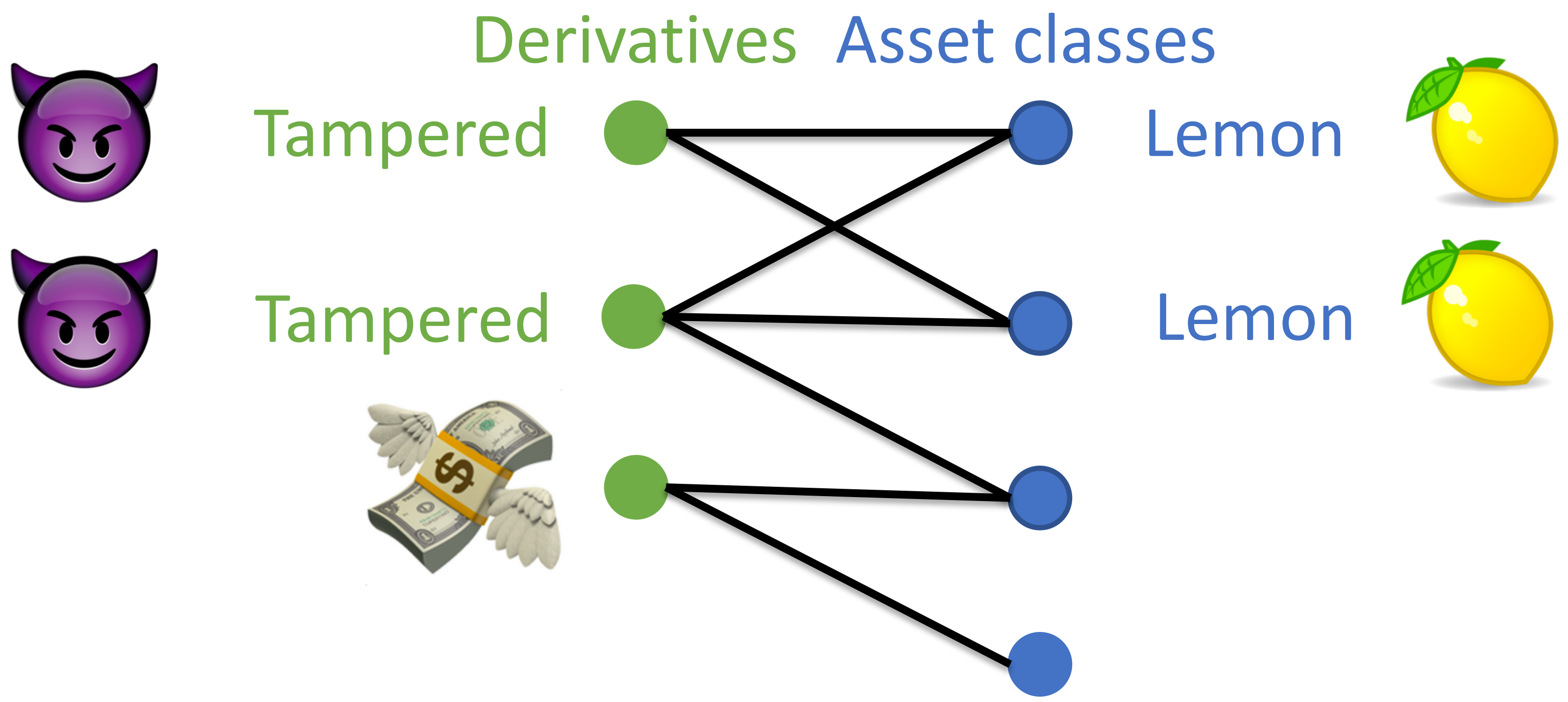
- Note: Web communities tend to be dense bipartite subgraphs[1]
- Web community bipartitions: topics, page creators interested in topics



[1] Kumar, Raghavan, Rajagopalan, Tomkins (99)

Tampered derivatives

- Tampered derivatives: Backed by set of assets/loans, tampered to contain many unprofitable (lemon) asset classes[2]



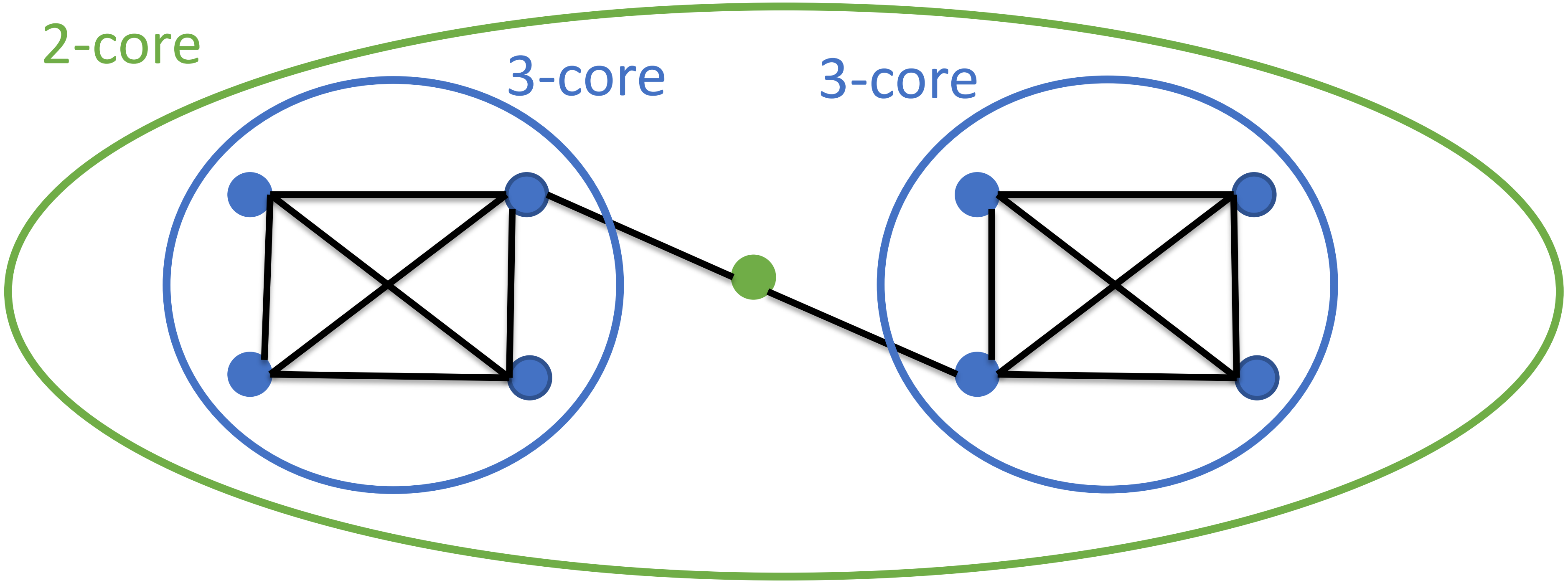
[2] Arora, Barak, Brunnermeier, Ge (09)

How do we find dense subgraphs?

- How do we find dense subgraphs (in general)?
- Algorithms:
  - K-core
  - Triangle peeling
- How do we find dense bipartite subgraphs?

How do we find dense subgraphs?

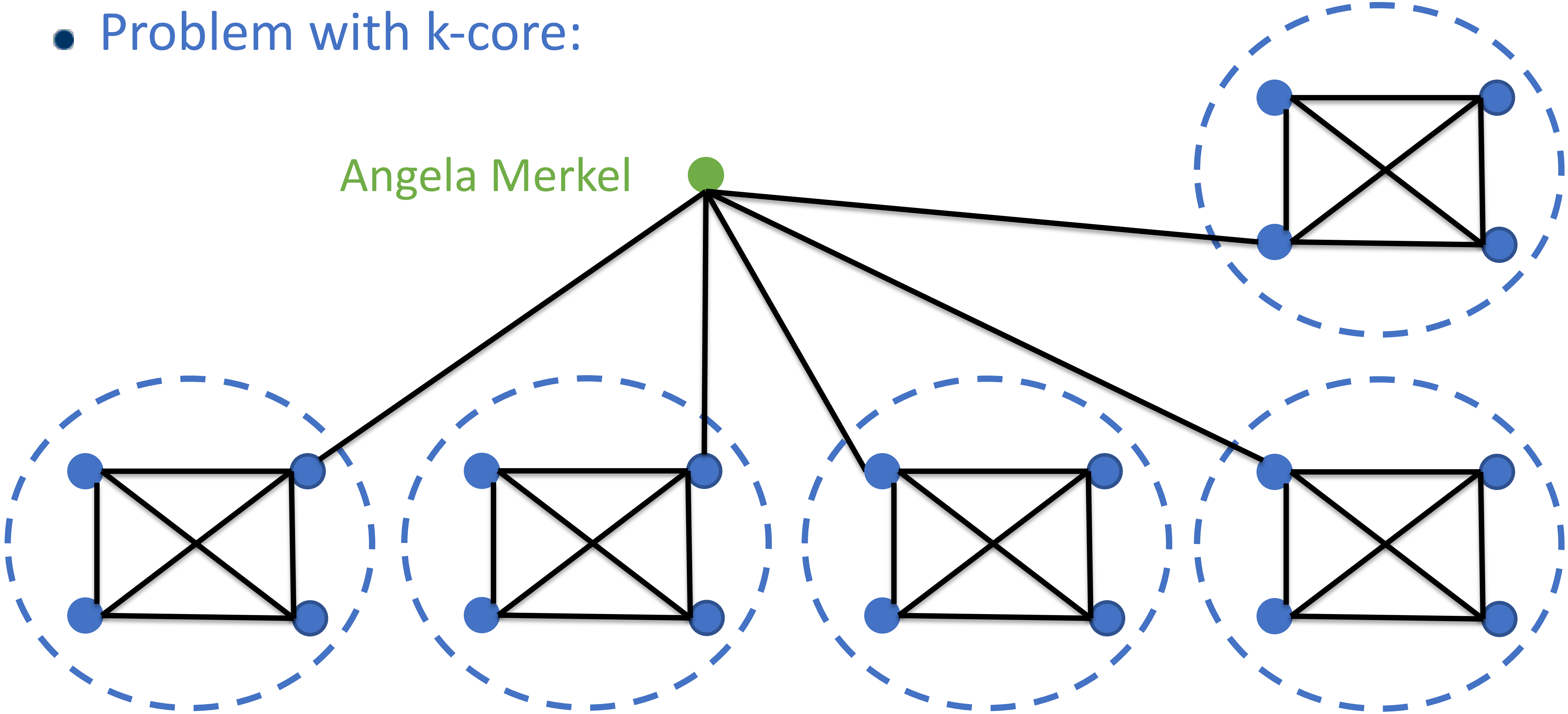
- K-core: Repeatedly find + delete min degree vertex



Formally: A k-core is an induced subgraph where every vertex has degree at least k

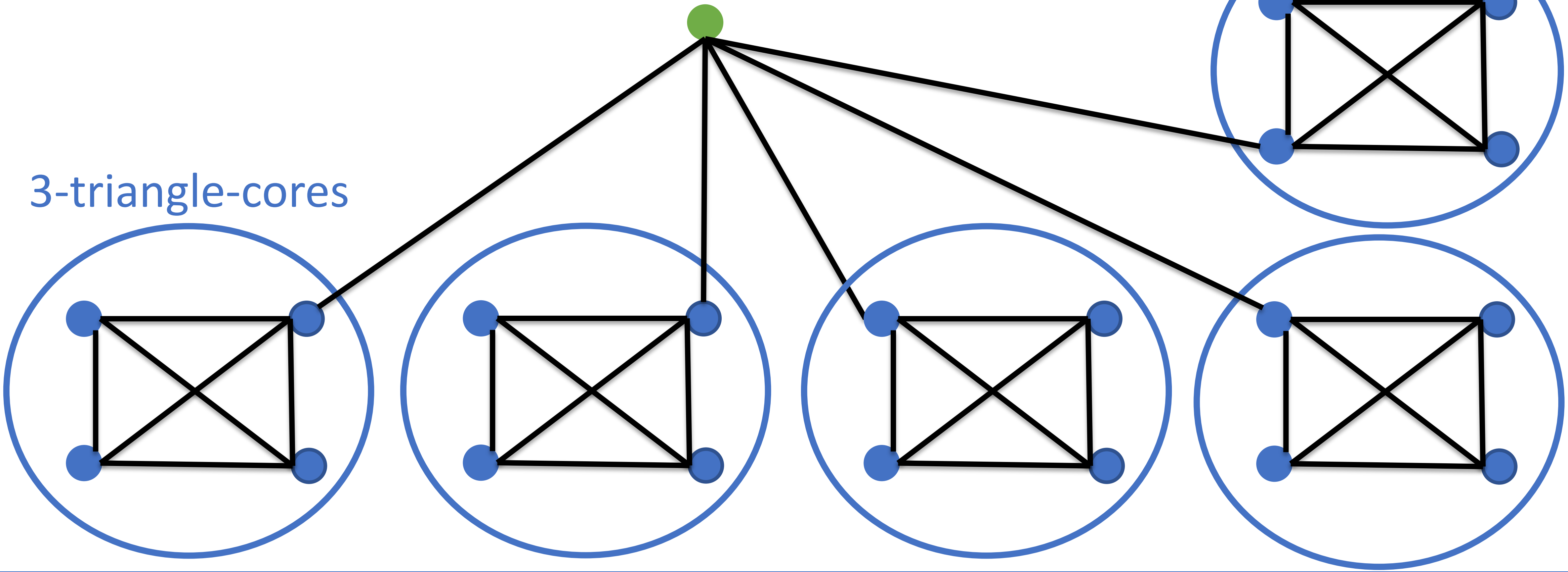
How do we find dense subgraphs?

- Problem with k-core:



How do we find dense subgraphs?

- Triangle peeling: Repeatedly find + delete vertex contained within the minimum # of triangles



How do we find dense subgraphs?

- Problem: Bipartite graphs do not contain any triangles
- Butterfly peeling: Repeatedly find + delete vertex containing min # of butterflies[3]

[3] Sariyuce and Pinar (18)

Outline (Butterflies)

- Main goal: Build a framework ParButterfly to count and peel butterflies

- New parallel algorithms for butterfly counting + peeling
- ParButterfly framework with modular settings
  - Tradeoff b/w theoretical bounds + practical speedups
- Comprehensive evaluation
  - Counting outperforms fastest seq algorithms by up to 13.6x
  - Peeling outperforms fastest seq algorithms by up to 10.7x

Outline (Cliques)

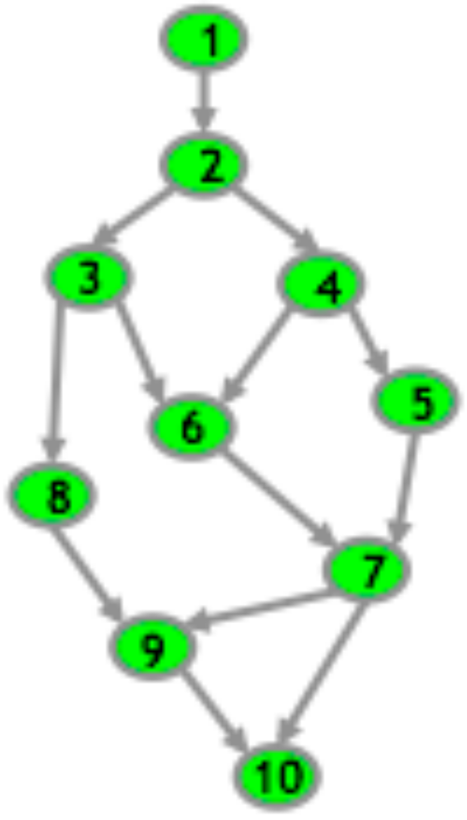
- Main goal: Develop efficient exact and approximate algorithms for k-clique counting and peeling

- New parallel algorithms for k-clique counting + peeling
- Comprehensive evaluation
  - Counting outperforms fastest parallel algorithms by up to 9.88x
  - Peeling outperforms fastest seq algorithms by up to 11.83x
  - Compute 4-clique counts on largest publicly-available graph with > 200 billion edges

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

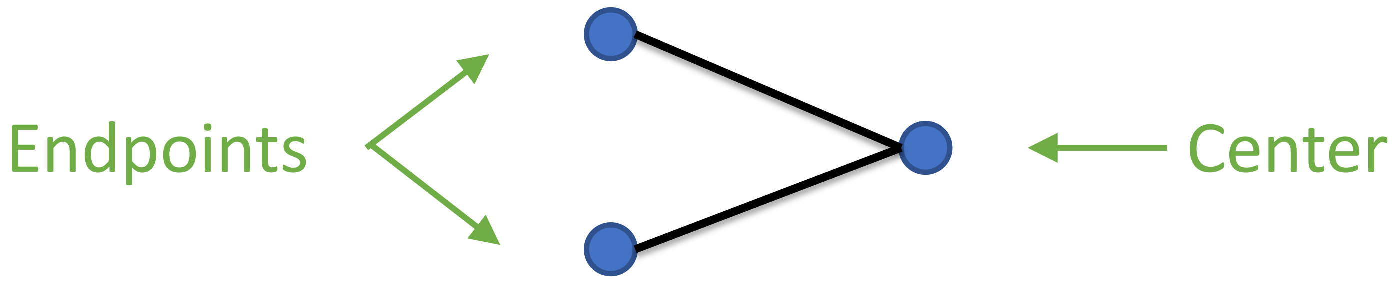


https://web.fe.up.pt/~jbarbosa/en/research_par.html

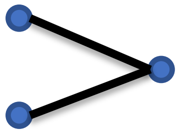
ParButterfly counting framework

How do we count butterflies? (per vertex)

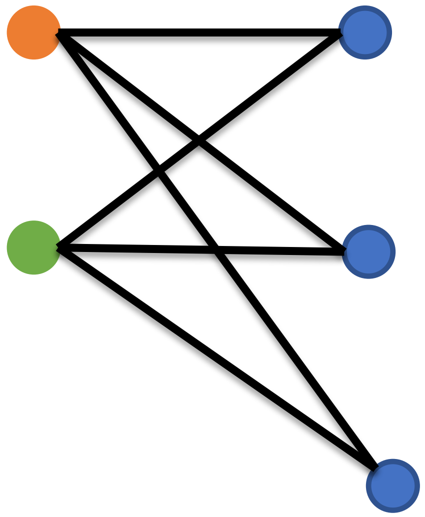
Wedge = P_2 =



How do we count butterflies? (per vertex)

Wedge = P_2 = 

Wedges with the same endpoints form butterflies:



wedges w/endpoints = w = 3

butterflies on endpoints = $\binom{w}{2} = \binom{3}{2}$ = 3

butterflies on each center = $w - 1$ = 3 – 1 = 2

Counting framework so far

1. Retrieve wedges

2. Aggregate wedges

For each pair of endpoints, count # wedges w

3. Compute butterfly counts

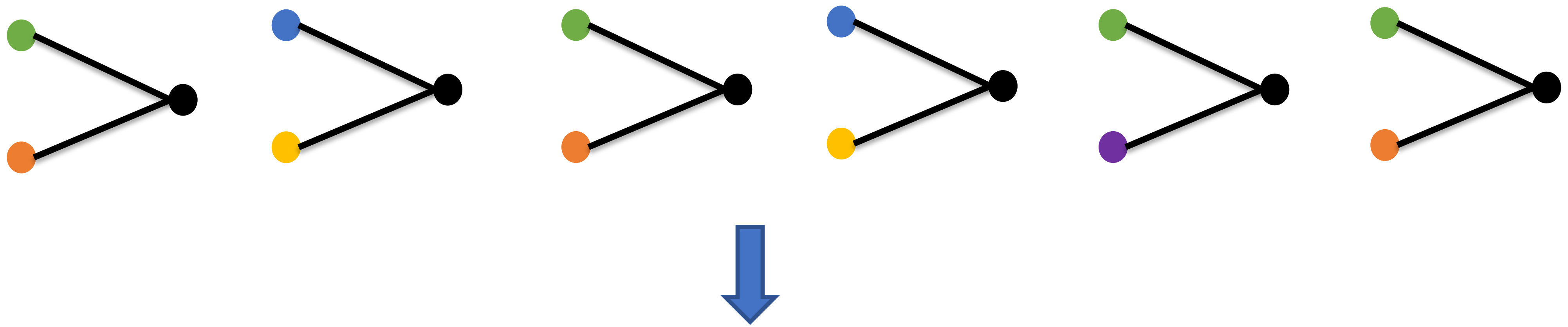
$+\binom{w}{2}$ for each endpoint	$+ w - 1$ for each center

One question: How do we aggregate wedges?

(will discuss wedge retrieval after)

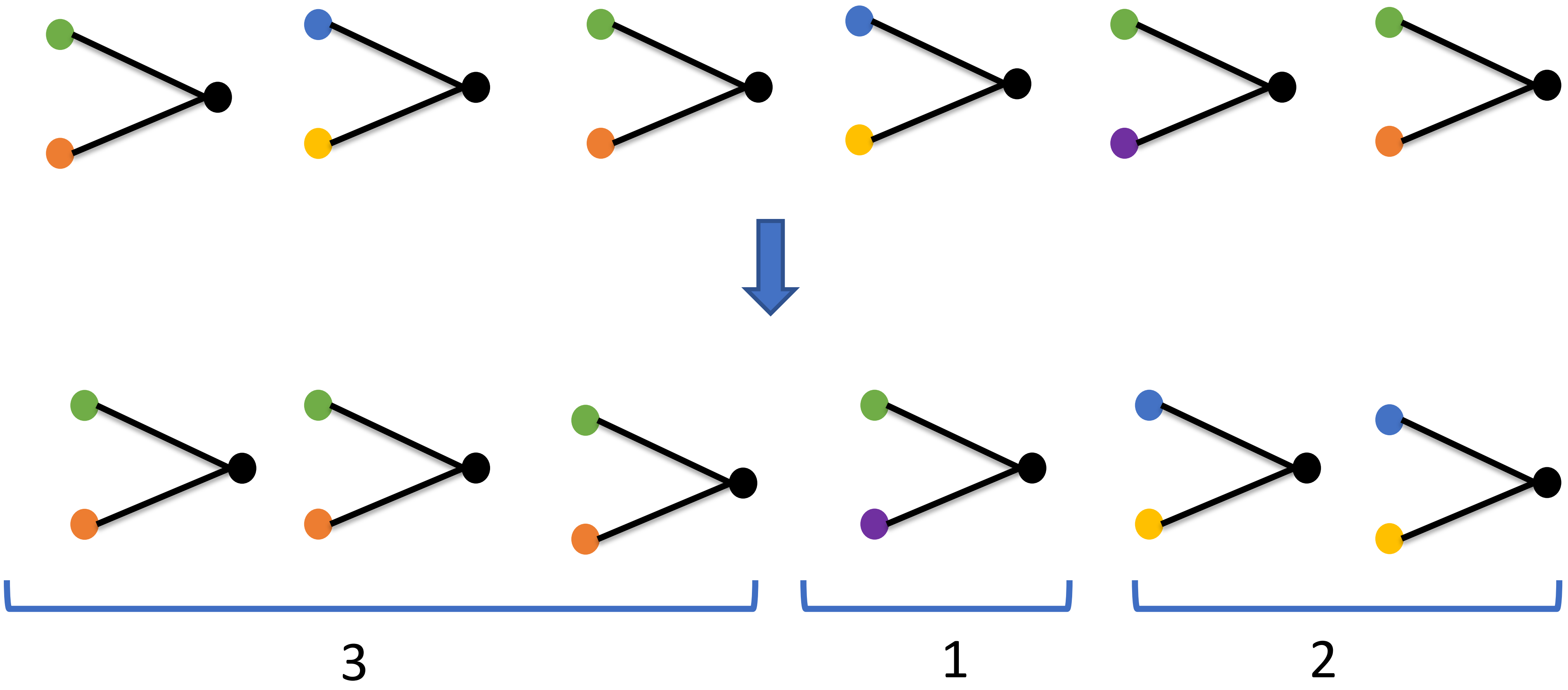
Wedge aggregating

- Method 1: Semisorting (on endpoints)



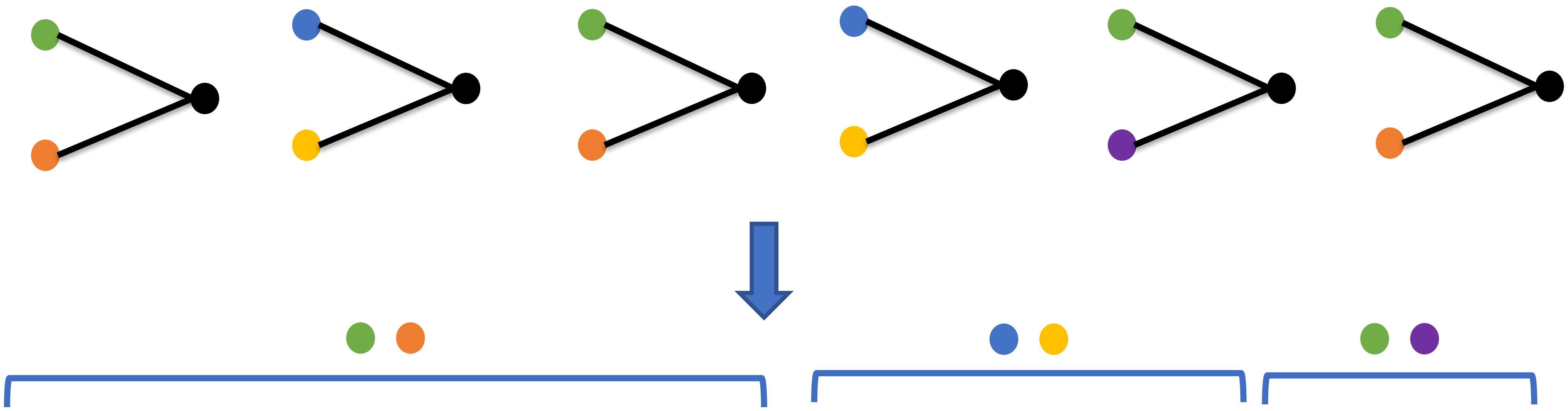
Wedge aggregating

- Method 1: Semisorting (on endpoints)



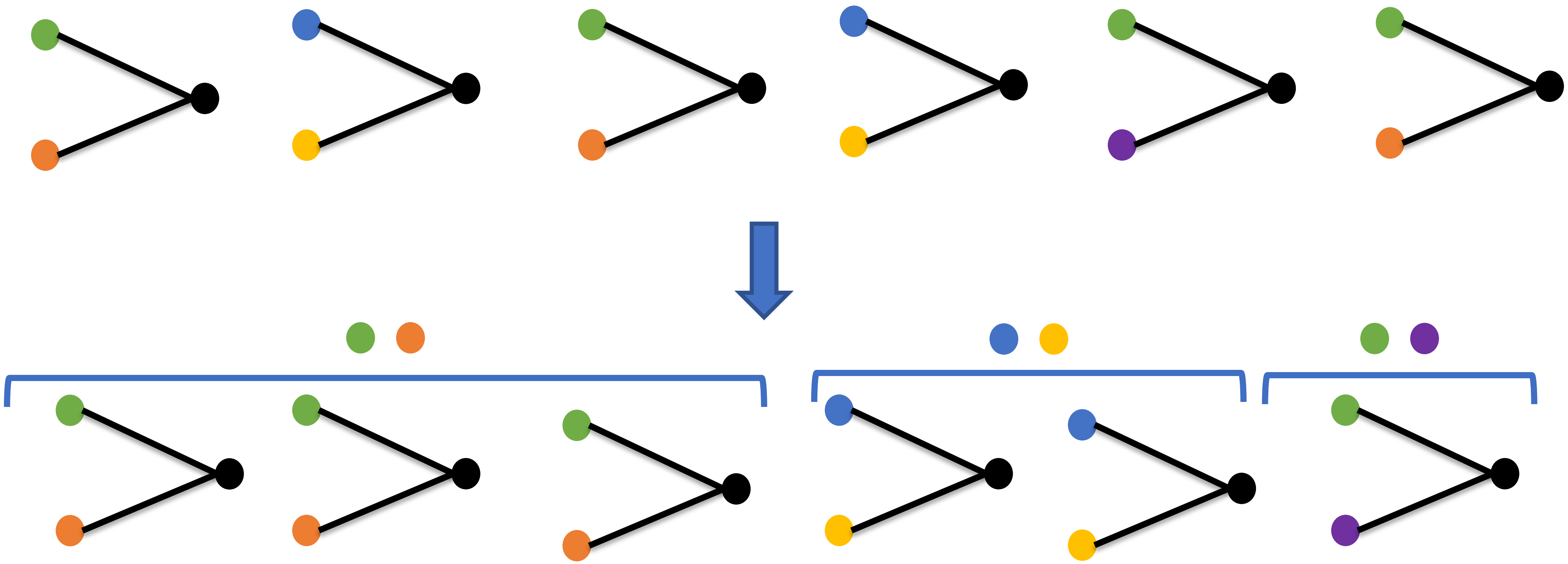
Wedge aggregating

- Method 2: Hashing (keys = endpoints)



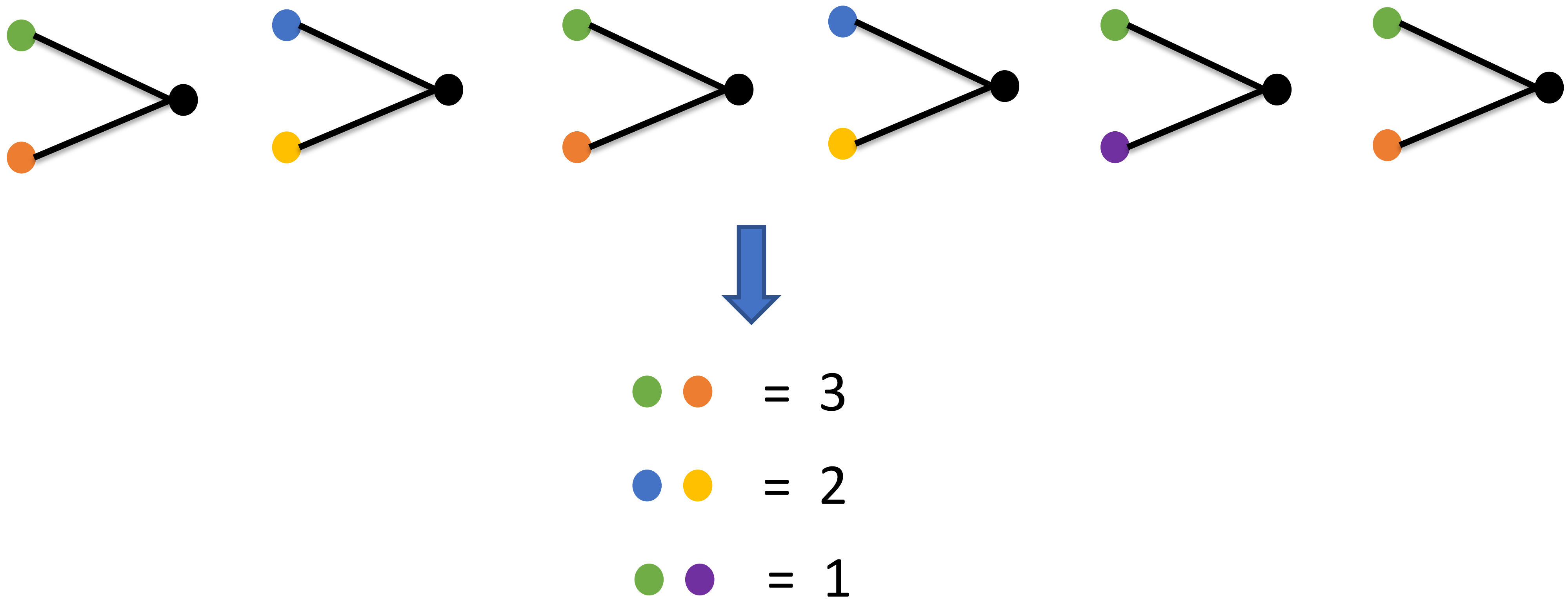
Wedge aggregating

- Method 2: Hashing (keys = endpoints)



Wedge aggregating

- Method 3: Histogramming (frequencies of endpoints)



Wedge aggregating bounds

Semisorting[1], hashing[2], and histogramming[3] are all work-efficient

w = # of wedges

O(w) expected work, O(log w) span whp

[1] Gu, Shun, Sun, and Blelloch (15)
[2] Shun and Blelloch (14)
[3] Dhulipala, Blelloch, and Shun (17)

Counting framework so far

1. Retrieve wedges

2. Aggregate wedges

Semisort, Hash Histogram

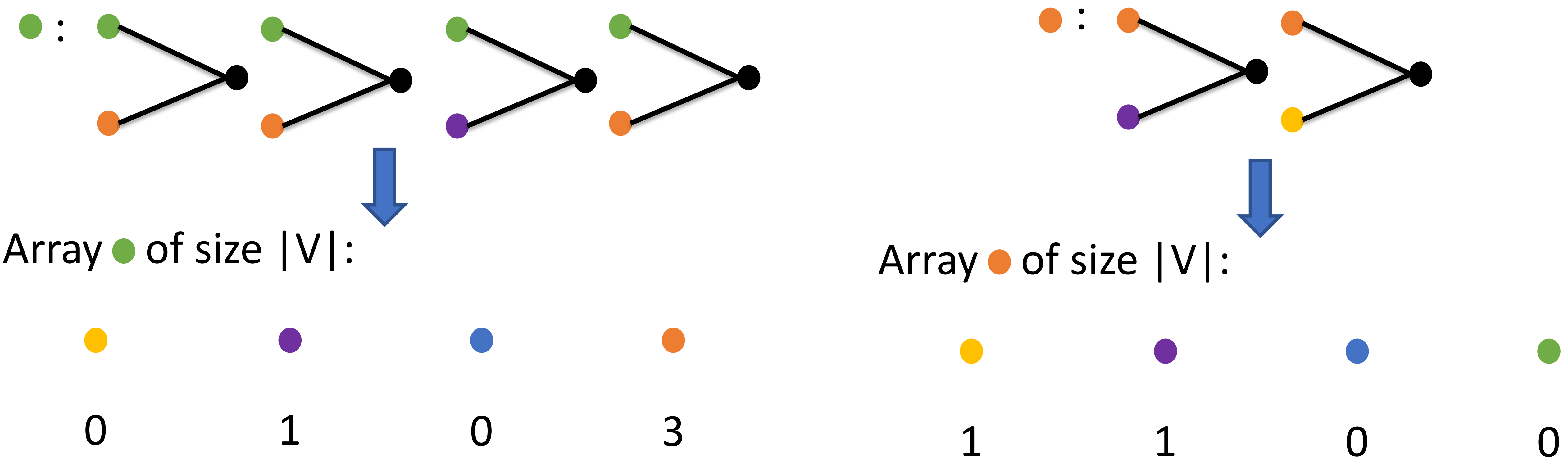
3. Compute butterfly counts

One more way to count wedges: Batching

(not with polylogarithmic span, but fast in practice)

Wedge aggregating (batching)

- Main idea: Process a subset of vertices in parallel, finding all wedges where those vertices are endpoints



Counting framework so far

1. Retrieve wedges

2. Aggregate wedges

Semisort, Hash Histogram, Batch

3. Compute butterfly counts

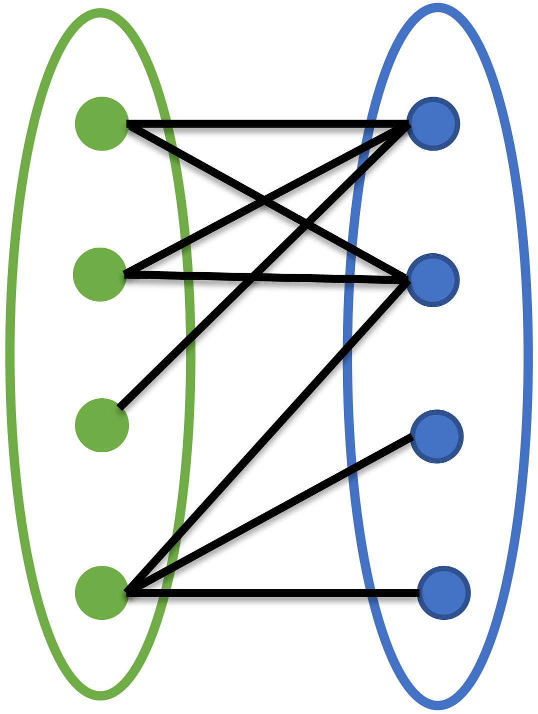
More questions:

How do we retrieve wedges?
How many wedges are there?

It depends!

- Method 1: Process wedges w/endpoints from one bipartition (Side) [1]

6 wedges



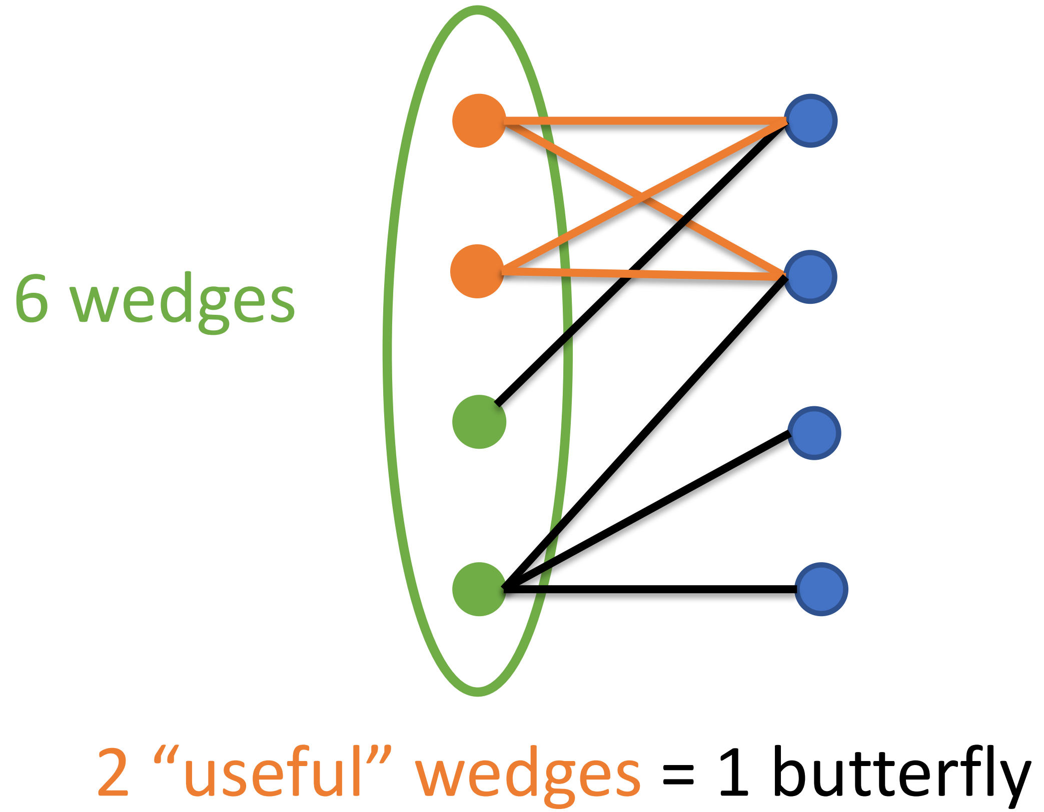
5 wedges

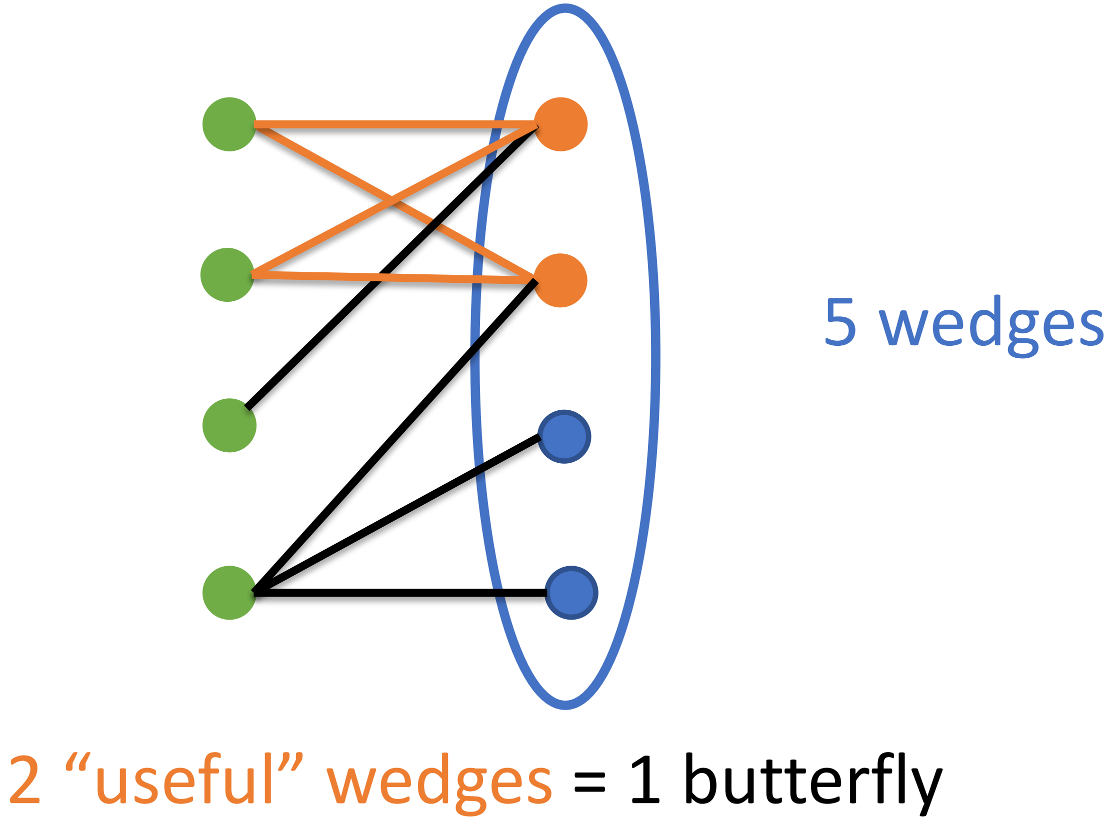
Is this optimal (min # wedges)? Not always.

[1] Sanei-Mehri, Sariyuce, Tirthapura (18)

(Note: Butterfly count remains the same)

- Regardless of which side we pick, butterfly count does not change – only some “useful” wedges create butterflies





Retrieve wedges

- Method 2: Degree ranking

Main idea:

Once we obtain all wedges with endpoint v, we do not have to consider wedges with endpoint v again.

[1] Chiba and Nishizeki (85)

Retrieve wedges

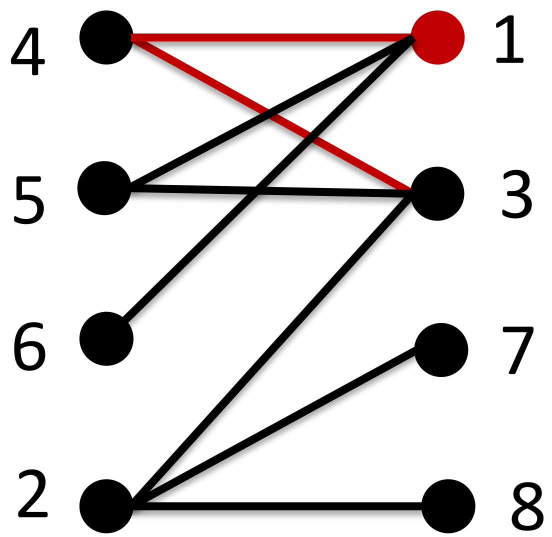
- Method 2: Degree ranking

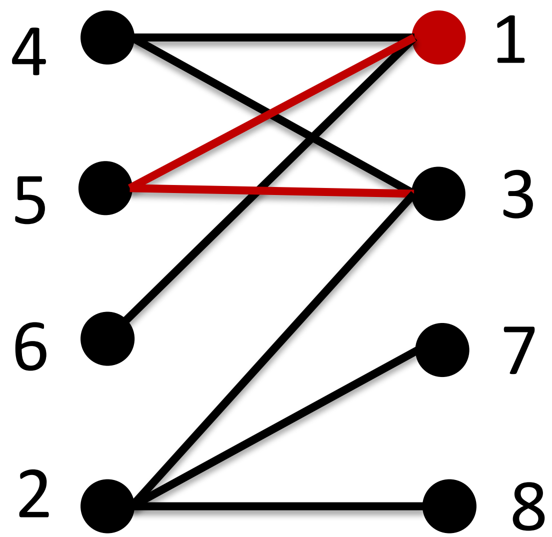
1. Order vertices by non-increasing degree
2. For each vertex v, only consider wedges with endpoint v that is formed by vertices later in the ordering than v

[1] Chiba and Nishizeki (85)

Retrieve wedges

- Method 2: Degree ranking

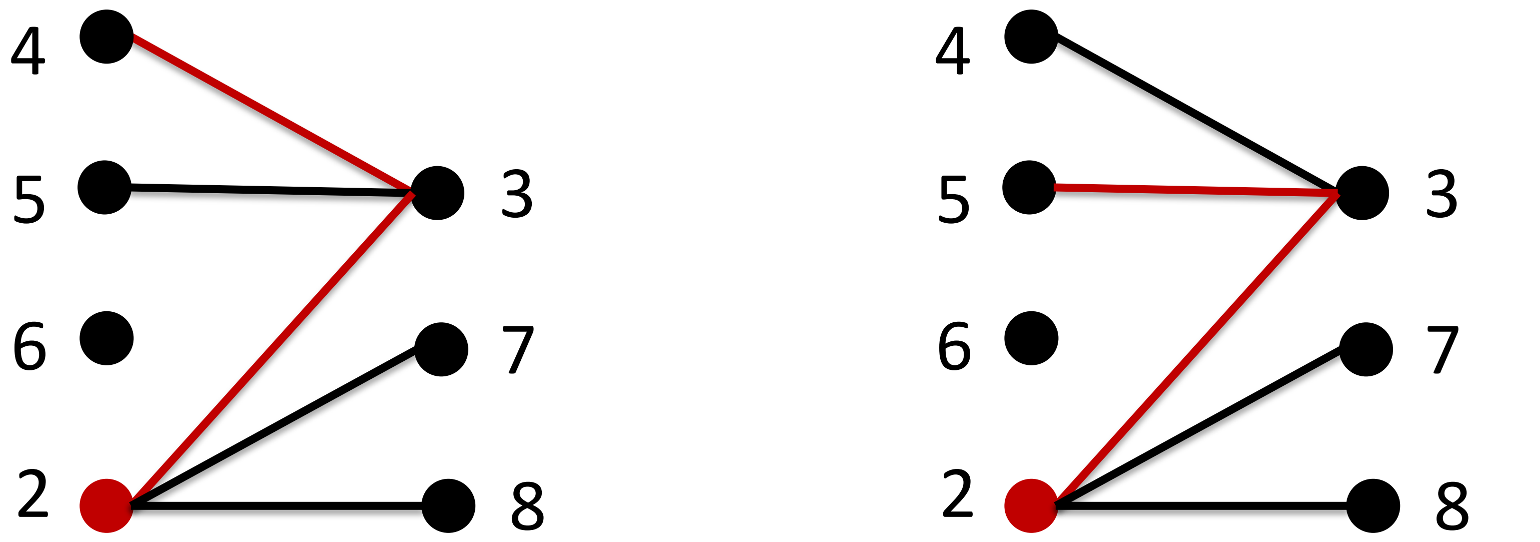




2 wedges

Retrieve wedges

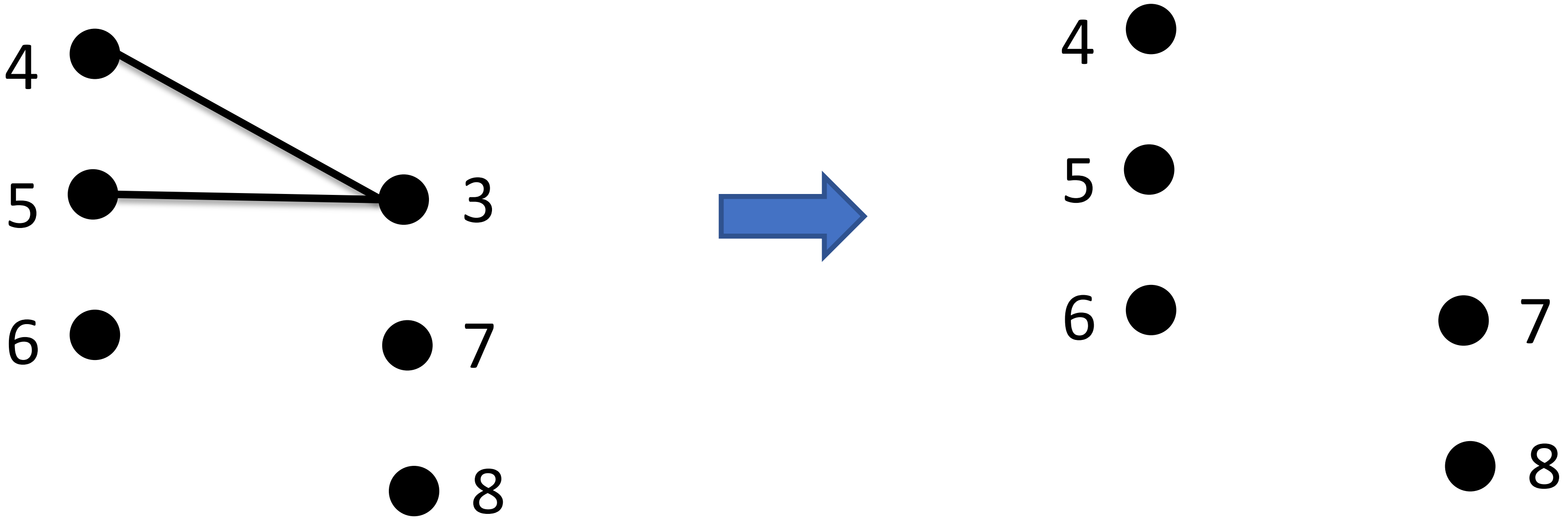
- Method 2: Degree ranking



2 wedges

Retrieve wedges

- Method 2: Degree ranking



We only processed 4 wedges!

Degree ranking

- # wedges processed using degree order = O(αm) [1]
  - α = arboricity/degeneracy (O($\sqrt{m}$))
  - m = # edges
- Therefore: (using work-efficient options)

Ranking vertices = O(m) expected work, O(log m) span whp
Retrieving wedges = O(αm) expected work, O(log m) span whp
Counting wedges = O(αm) expected work, O(log m) span whp
Computing butterfly counts = O(αm) expected work, O(log m) span whp

Total = O(αm) expected work, O(log m) span whp

[1] Chiba and Nishizeki (85)

Other rankings

- Approximate degree order
  - Log degree
- Complement degeneracy order
  - Ordering given by repeatedly finding + deleting greatest degree vertex
- Approximate complement degeneracy order
  - Complement degeneracy order, but using log degree

We show these are all work-efficient

Counting framework

1. Rank vertices

Side, Degree, Approx Degree, Co Degeneracy, Approx Co Degeneracy

2. Retrieve wedges

3. Aggregate wedges

Semisort, Hash Histogram, Batch

4. Compute butterfly counts

$O(\alpha m)$ expected work, $O(\log m)$ span whp

ParButterfly peeling framework

How do we peel butterflies?

- Goal: Iteratively remove all vertices with min butterfly count

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

Note: We've already done subgoal 2 in counting framework

For subgoal 1, we give a work-efficient batch-parallel Fibonacci heap which supports batch insertions/decrease-keys (see paper).

Peeling framework

1. Obtain butterfly counts

2. Iteratively remove vertices with min butterfly count

- Use batch-parallel Fibonacci heap to find vertex set S
- Count wedges with endpoints in S
  - Semisort, Hash, Histogram, Batch
- Compute updated butterfly counts

We show this algorithm is work-efficient

(with respect to peeling complexity)

ParButterfly evaluation

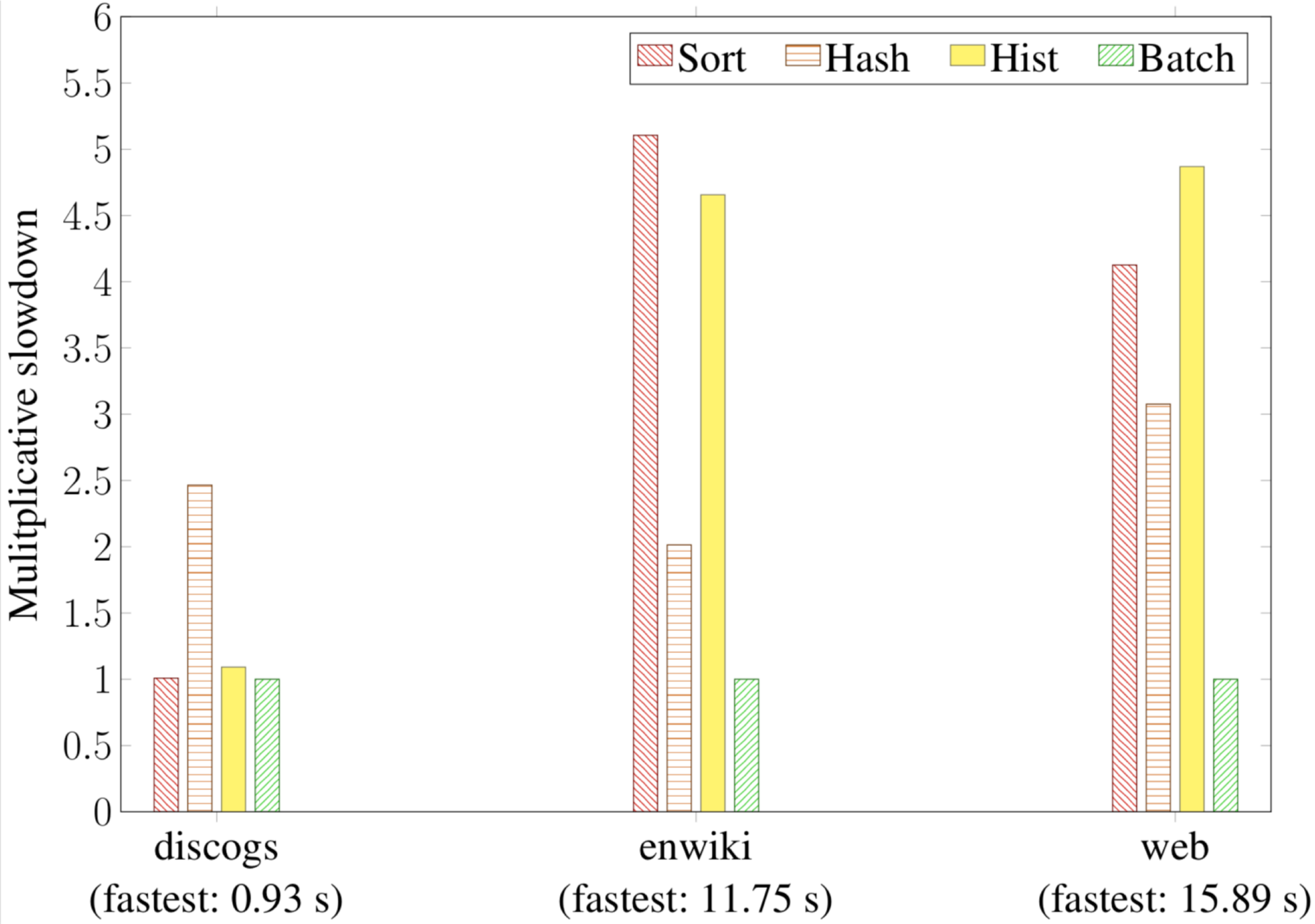
Environment

- m5d.24xlarge AWS EC2 instance: 48 cores (2-way hyper-threading), 384 GiB main memory
- Cilk Plus[1] work-stealing scheduler
- Koblenz Network Collection (KONECT) bipartite graphs
- Experiments for the different modular options in our framework
- Some modifications:
  - Julienne[2] instead of batch-parallel Fibonacci heap
  - Cannot hold all wedges in memory – batch wedge retrieval

[1] Leiserson (10)
[2] Dhulipala, Blelloch, and Shun (17)

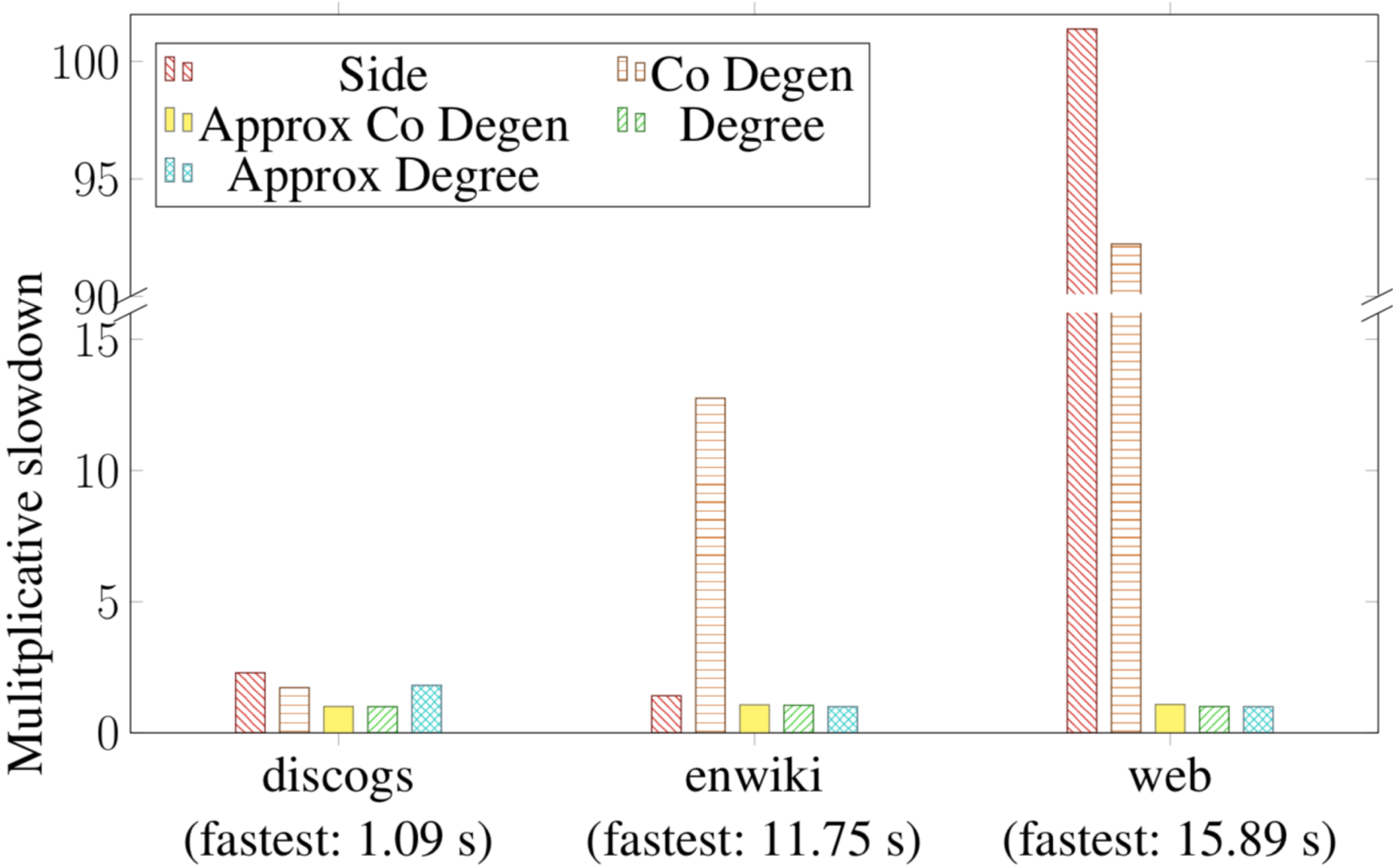
Counting:

Best aggregation method: **Batching**



Counting:

Best ranking method: **Approx Complement Degeneracy / Approx Degree**



Butterfly counting results

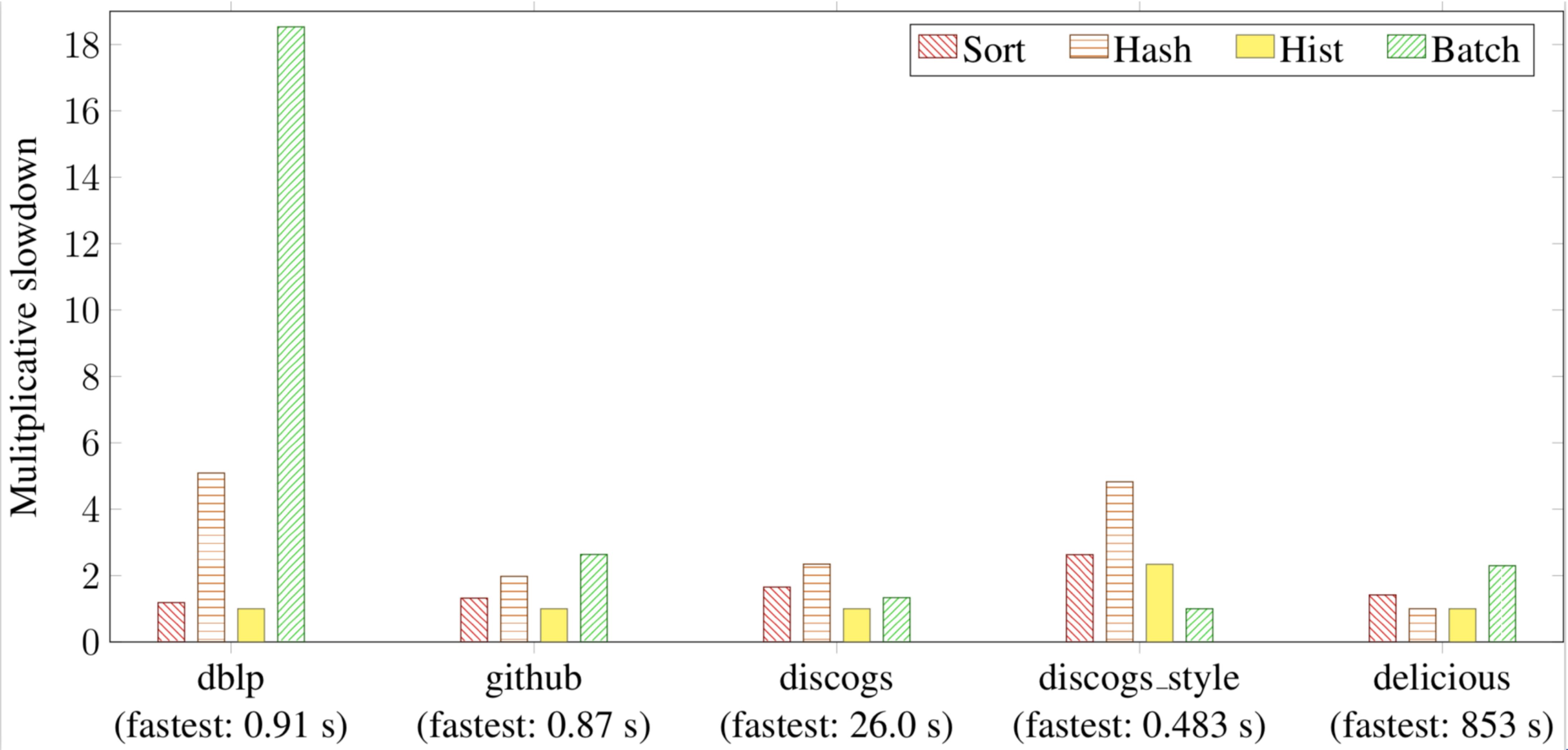
- 6.3 – 13.6x speedups over best seq implementations[1] [2]
- 349.6 – 5169x speedups over best parallel implementations[3]
  - Due to work-efficiency
- 7.1 – 38.5x self-relative speedups

- Up to 1.7x additional speedup using a cache-optimization[4]

[1] Sanei-Mehri, Sariyuce, Tirthapura (18)
[2] ESCAPE: Pinar, Seshadhri, Vishal (17)
[3] PGD: Ahmed, Neville, Rossi, Duffield, and Wilke (17)
[4] Wang, Lin, Qin, Zhang, and Zhang (19)

Peeling:

Best aggregation method: **Histogramming**



Butterfly peeling results

- 1.3 – 30696x speedups over best seq implementations[1]
  - Depends heavily on peeling complexity
  - Largest speedup due to better work-efficiency for some graphs
- Up to 10.7x self-relative speedups
  - No self-relative speedups if small # of vertices peeled

[1] Sariyuce and Pinar (18)

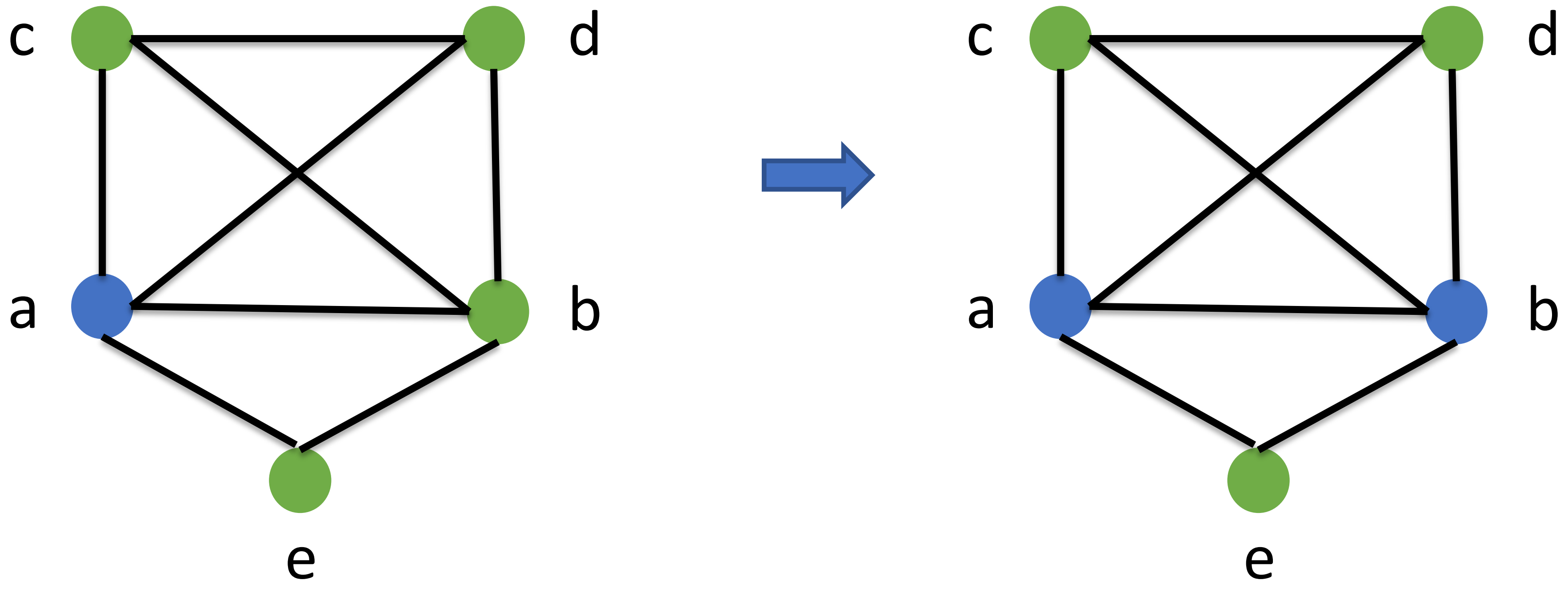
k-clique counting and peeling

How do we find k-cliques?

- Repeatedly intersect the neighborhoods of vertices

Neighbors of = {b, c, d, e}

Neighbors of = {c, d, e}



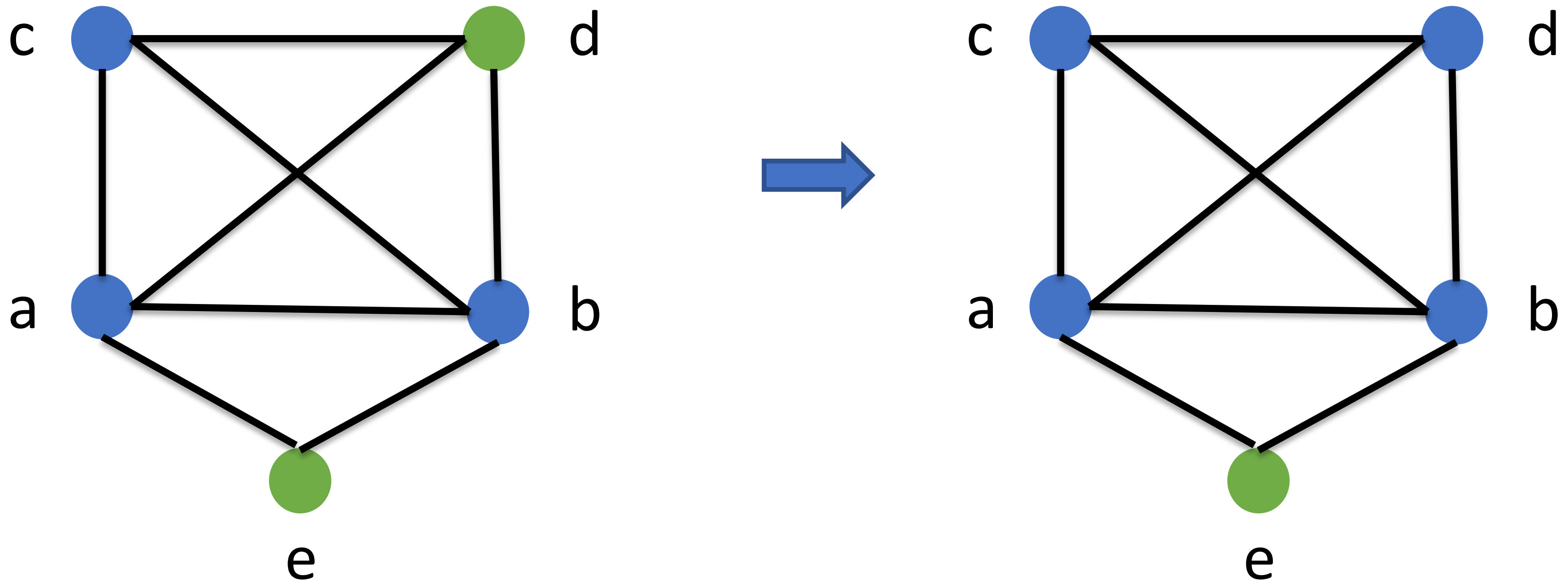
How do we find k-cliques?

- Repeatedly intersect the neighborhoods of vertices

Neighbors of ● = {d}

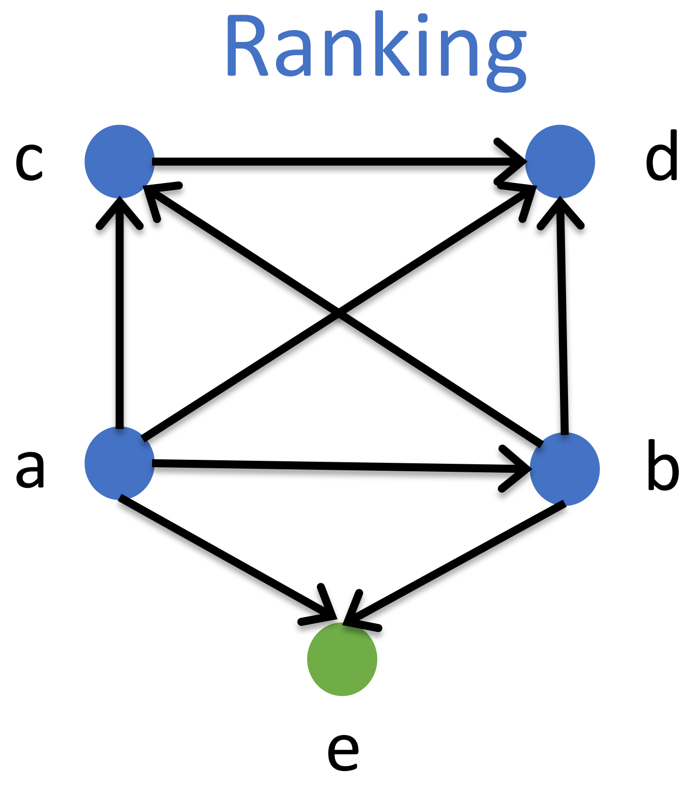
Neighbors of ● = {}

4-clique = {a, b, c, d}



Déjà vu: Ranking vertices

- How do we avoid double-counting k-cliques?



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

Ranking vertices

- Degree ranking
- K-core ranking

Used in previous work
Work-efficient serially

- Arboricity ranking
  - Goodrich-Pszona
  - Barenboim-Elkin

Our contribution (parallelizations of these algorithms)
Space-efficient in parallel
Polylogarithmic span in parallel

$O(\alpha)$-orientations in $O(m)$ expected work, $O(\log^2 n)$ span whp

k-clique counting in $O(m\alpha^{k-2})$ expected work, $O(k \log n + \log^2 n)$ span whp, $O(m + P\alpha)$ space

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

Note: We've already done subgoal 2 in the counting algorithm

And we've already done subgoal 1 in the butterfly peeling algorithm!

We show this algorithm is work-efficient
(with respect to peeling complexity)

An aside on k-clique peeling

- k-clique peeling uses essentially the same algorithm, but must consider the undirected neighborhood of the peeled vertex
- Nash-Williams Theorem gives work-efficient bounds
  - For every subgraph S, $\propto \geq {}^{E(S)}/_{V(S)-1}$

k-clique evaluation

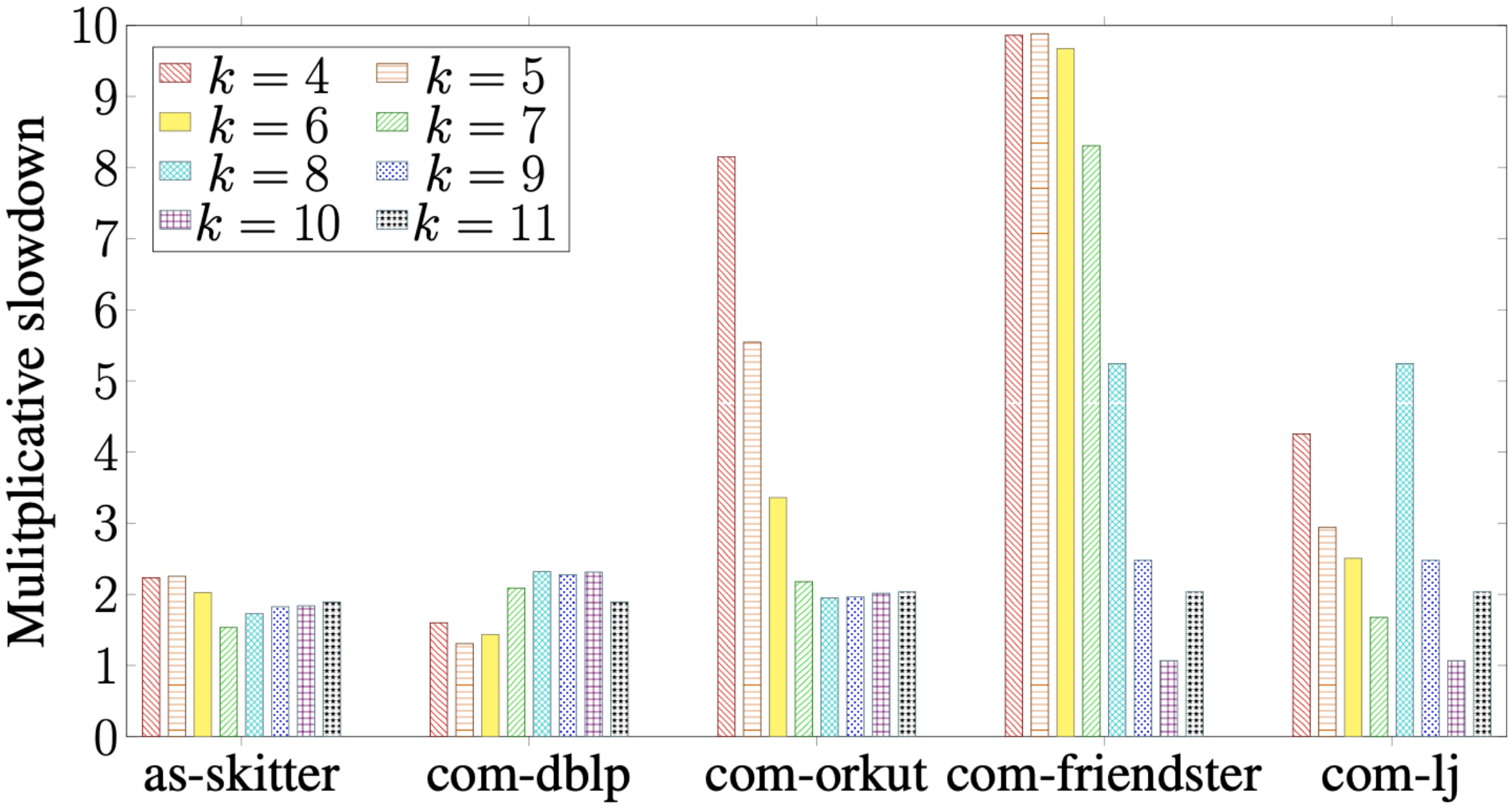
Evaluation (k-clique counting)

- 60-core GCP instance (two-way hyperthreading)
- 1.31 – 9.88x speedups over parallel KClist[1]
- 2.26 – 79.20x speedups over serial KClist
- Up to 196.28x speedups over parallel Pivoter
  - Pivoter[2] is faster: $k \geq 8$ on as-skitter, com-dblp; $k \geq 10$ on com-orkut
- Obtain 4-clique counts on
  - ClueWeb (74 billion edges) in < 2 hours
  - Hyperlink2014 (~100 billion edges) in < 4 hours
  - Hyperlink 2012 (~200 billion edges) in < 45 hours

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

Evaluation (k-clique counting)

- Comparison to KClist



Evaluation (k-clique peeling)

- 60-core GCP instance (two-way hyperthreading)
- 1.01 – 11.83x speedups over serial KClist
- Constrained by peeling complexity

Conclusion

Conclusion

- New parallel algorithms for butterfly counting/peeling
- Modular ParButterfly framework w/ranking + aggregation options
- Strong theoretical bounds + high parallel scalability
- Github: https://github.com/jeshi96/parbutterfly

- New parallel algorithms for k-clique counting/peeling
- Strong theoretical bounds + high parallel scalability
- Github: https://github.com/ParAlg/gbbs/tree/master/benchmarks/CliqueCounting

Future Work

- Cycle counting (for $k \geq 6$)[1, 2, 3]
- Dynamic/Streaming subgraph counting[4, 5]
- Nucleus decomposition[6]
- Objective function for butterfly peeling[7]
- GraphIt extensions
- Hypergraph algorithms

[1] Bera, Pashanasangi, Seshadhri (19)
[2] Kowalik (03)
[3] Pinar, Seshadhri, Vishal (16)
[4] Sanei-Mehri, Zhang, Sariyuce, Tirthapura (19)
[5] Eppstein, Spiro (09)
[6] Sariyuce, Seshadhri, Pinar, Catalyurek (15)
[7] Tsourakakis (15)

Thank you

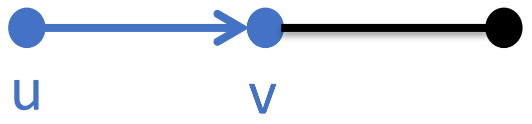
Limitations

- Butterfly peeling is P-complete (limited speedups)
- Work-efficient butterfly counting is not the fastest in practice
  - Reducing space usage in butterfly counting
- Not easily generalized to other subgraphs

Deriving αm

- # wedges = $\sum_{x \in V} \sum_{y \in N_x(x)} deg_x(y)$
  - Where $N_x(y)$ and $deg_x(y)$ refer to neighbors / degree of y considering vertices with rank > rank(x)

$$\leq \sum_{(u,v)\in E} \min(\deg(u), \deg(v))$$



(where u has higher degree (lower rank) than v)

$$\leq \sum_{\text{forest } F} \sum_{(u,v)\in F} \min(\deg(u), \deg(v))$$

$$\leq \sum_{\text{forest } F} \sum_{v\in V} \deg(v)$$

$$= O(\alpha m)$$

Priority queue for butterfly counts

Batch-parallel Fibonacci heap:

- *k* insertions: O(*k*) amortized expected work, O(log(*n*+*k*)) span whp
- *k* decrease-keys: O(*k*) amortized work, O($\log^2 n$) span whp
- delete-min: O(log *n*) amortized expected work, O(log *n*) span whp

Analysis follows directly from serial Fibonacci heap analysis, except marks are integers instead of booleans

Additionally, we use a parallel hash table to maintain buckets for butterfly peeling

Peeling framework bounds

- By vertex: (ρ_v = number of peeling rounds across all vertices)

$O(\min(\text{max-}b_v, \rho_v \log m) + \sum \text{degree}(v)^2)$ expected work, $O(\rho_v \log^2 m)$ span whp, $O(n^2 + \text{max-}b_v)$ space

- By edge: (ρ_e = number of peeling rounds across all edges)

$O(\min(\text{max-}b_e, \rho_e \log m) + \sum_{(u,v)} \sum_{u' \in N(u)} \min(\text{degree}(u), \text{degree}(u')))$ expected work, $O(\rho_e \log^2 m)$ span whp, $O(m + \text{max-}b_e)$ space

(Using batch-parallel Fibonacci heap and Julienne)

Peeling framework bounds

- By vertex: (ρ_v = number of peeling rounds across all vertices)

$O(\rho_v \log m + \sum \text{degree}(v)^2)$ expected work, $O(\rho_v \log^2 m)$ span whp, $O(n^2)$ space

- By edge: (ρ_e = number of peeling rounds across all edges)

$O(\rho_e \log m + \sum_{(u,v)} \sum_{u' \in N(u)} \min(\text{degree}(u), \text{degree}(u')))$ expected work, $O(\rho_e \log^2 m)$ span whp, $O(m)$ space

(Using batch-parallel Fibonacci heap)

Peeling framework bounds (Storing all wedges)

- By vertex: (ρ_v = number of peeling rounds across all vertices)

  $O(\rho_v \log m + b)$ expected work, $O(\rho_v \log^2 m)$ span whp, $O(\alpha m)$ space

- By edge: (ρ_e = number of peeling rounds across all edges)

  $O(\rho_e \log m + b)$ expected work, $O(\rho_e \log^2 m)$ span whp, $O(\alpha m)$ space

(Using batch-parallel Fibonacci heap)

Peeling framework bounds (Storing all wedges)

- By vertex: (ρ_v = number of peeling rounds across all vertices)

O(b) expected work, O($\rho_v \log m$) span whp, O(αm + max-b_v) space

- By edge: (ρ_e = number of peeling rounds across all edges)

O(b) expected work, O($\rho_e \log m$) span whp, O(αm + max-b_e) space

(Using Julienne)

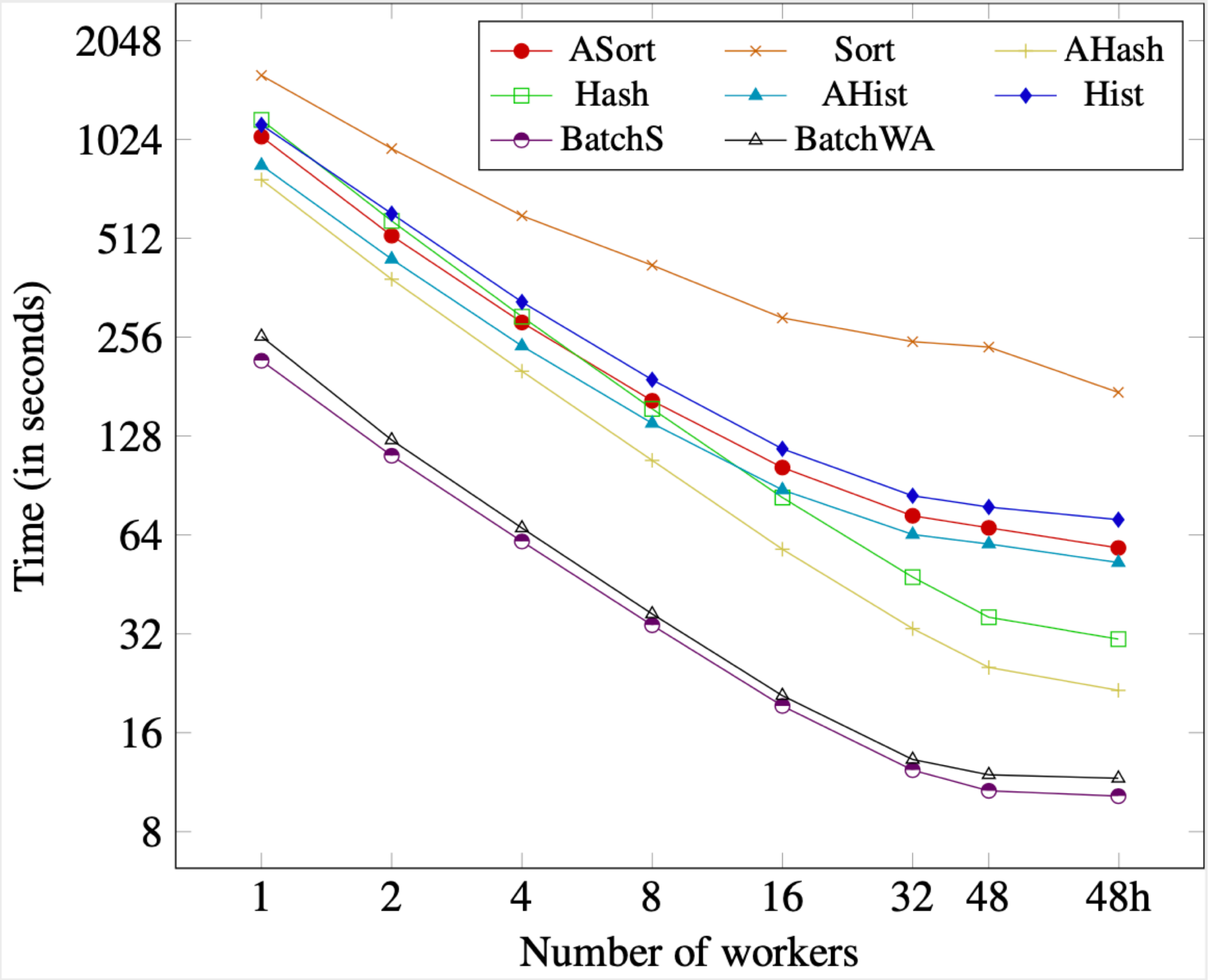
Sampling

- Edge sparsification: Keep each edge independently w/probability *p*

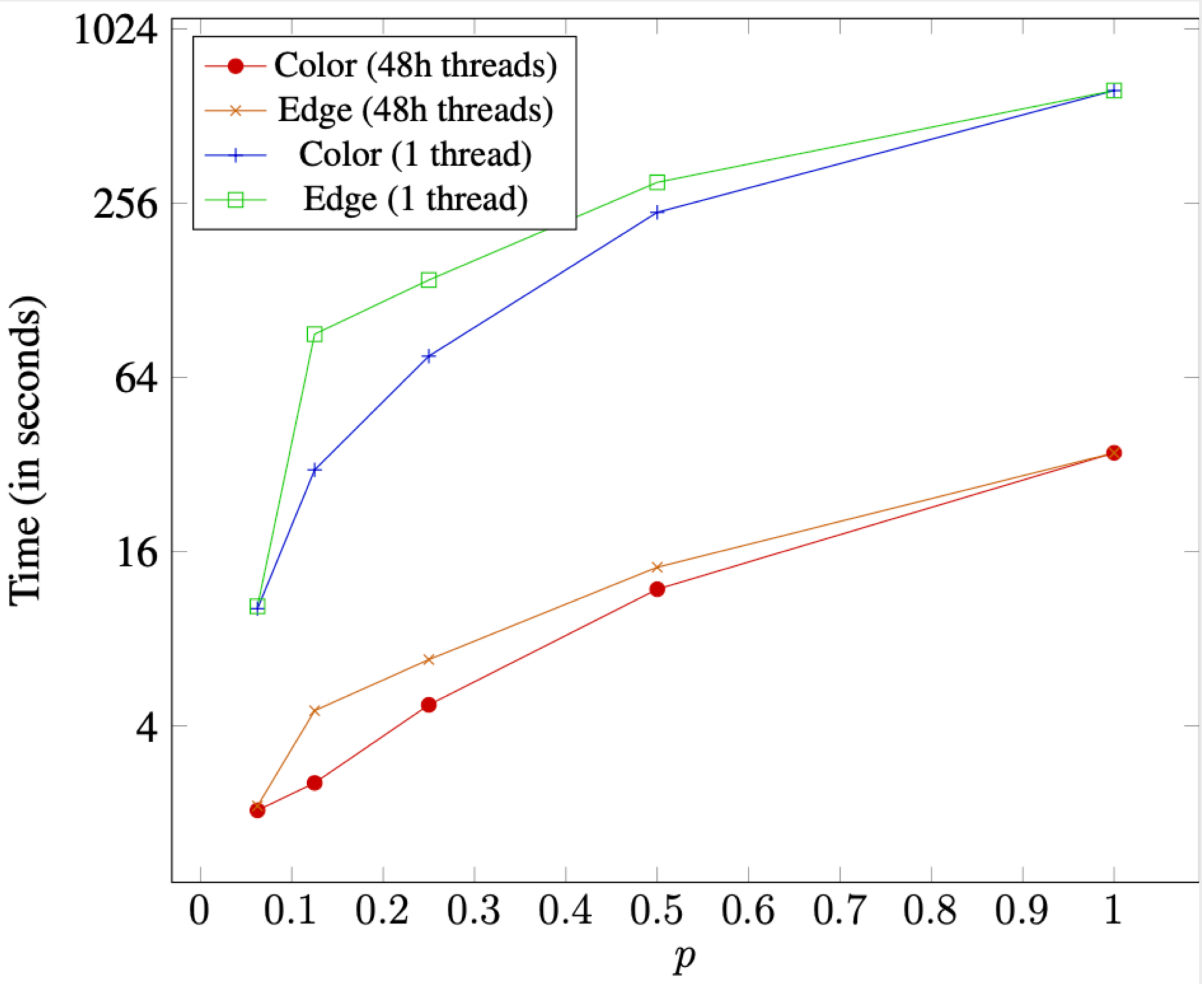
- Colorful sparsification: Assign a random color [1, …, 1/p] to each vertex + keep each edge if the endpoints match

[1] Sanei-Mehri, Sariyuce, Tirthapura (18)

Scalability (Per vertex counting)



Sampling



Color (48h threads)
Edge (48h threads)
Color (1 thread)
Edge (1 thread)
Time (in seconds)
1024
256
64
16
4
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
p

Wedge Aggregation (Per vertex counting with cache optimization)