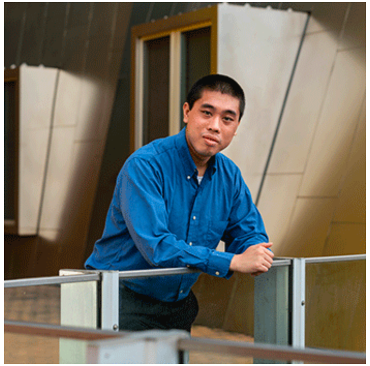
Theoretically and Practically Efficient Parallel Nucleus Decomposition


Jessica Shi
(MIT / Google)


Laxman Dhulipala
(University of Maryland)


Julian Shun
(MIT)

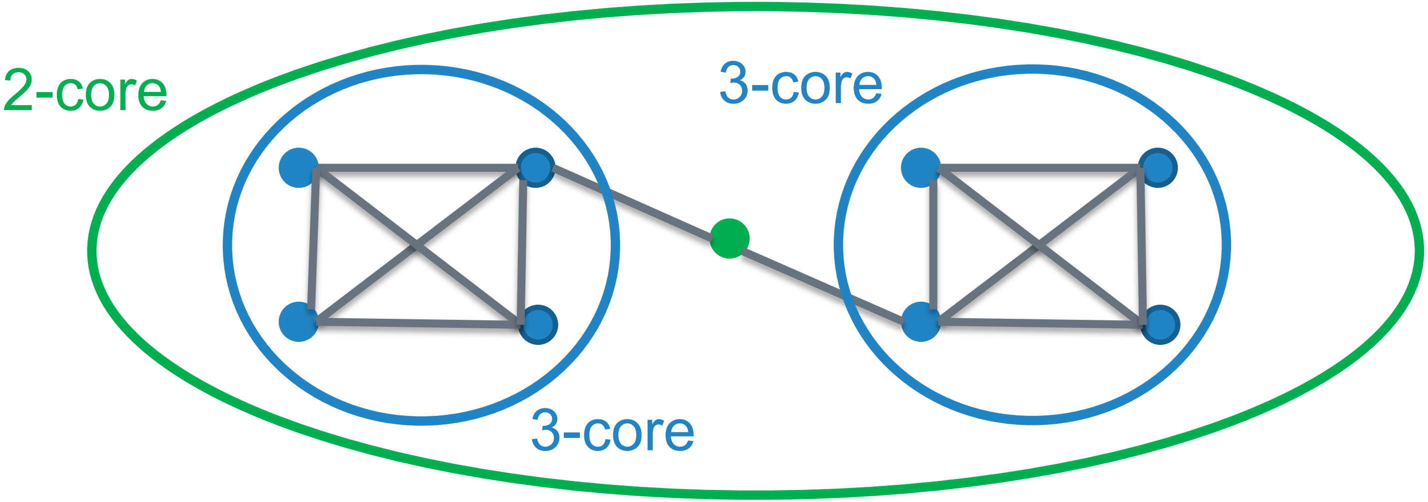
How do we cluster a graph?

- A fundamental idea:

How well-connected are certain nodes or subsets of nodes in a graph?

“Well-connected” nodes

- k-core: Repeatedly find + “delete” min degree vertex

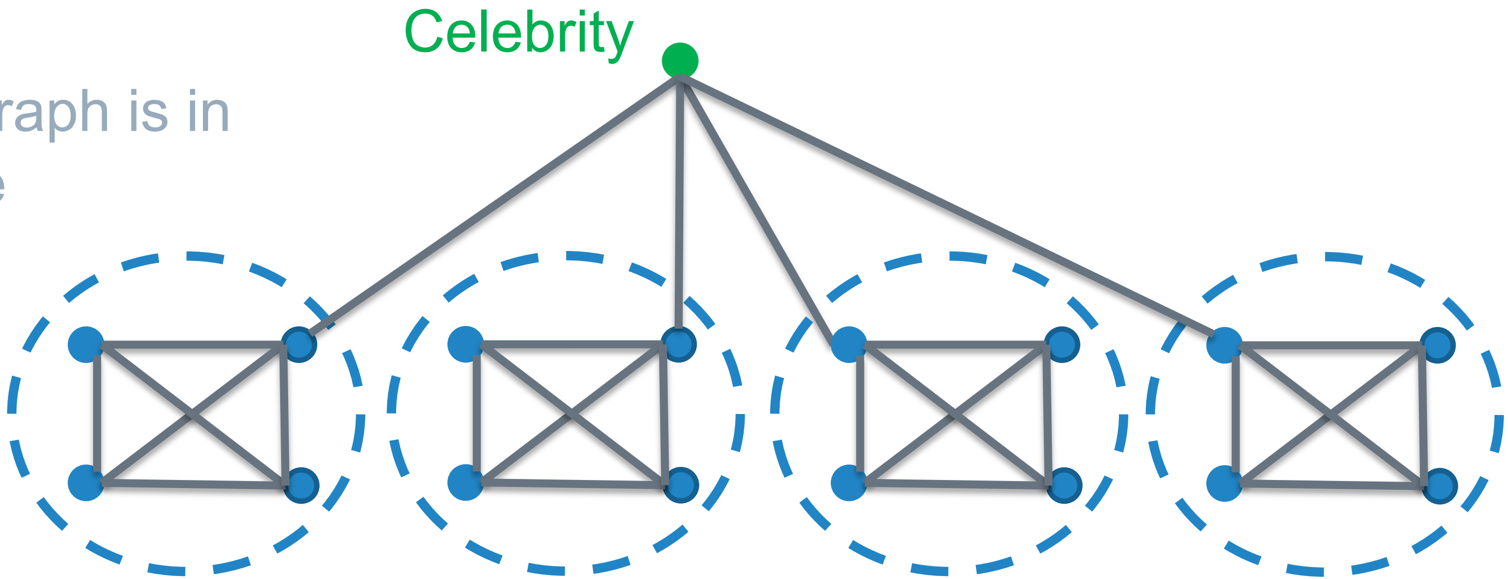


Formally: A k-core is an induced subgraph where every vertex has degree at least k

A problem with k-core

▷ k-core: Repeatedly find + “delete” min degree vertex

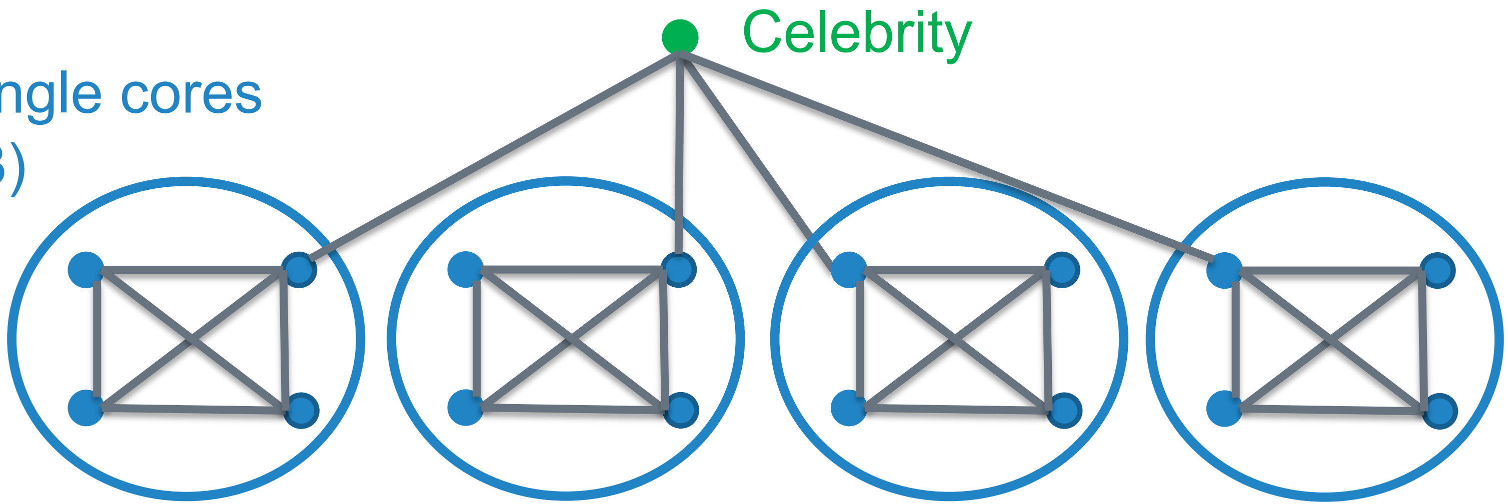
Entire graph is in a 3-core



s-clique peeling

- s-clique degree: Number of s-cliques each vertex participates in
- s-clique peeling: Repeatedly find + “delete” min s-clique degree vertex

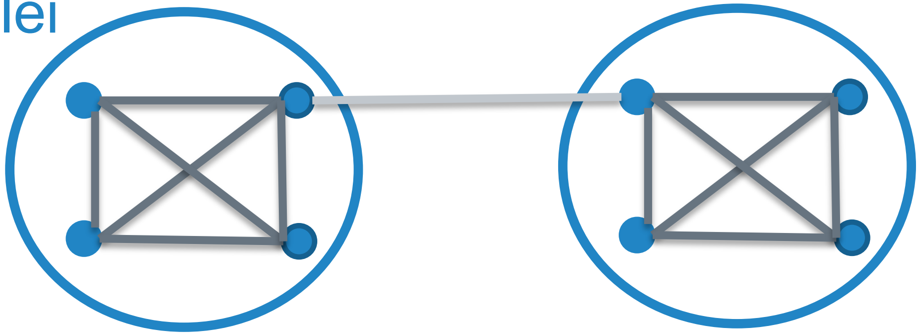
3-triangle cores
(s = 3)



(r, s)-nucleus decomposition

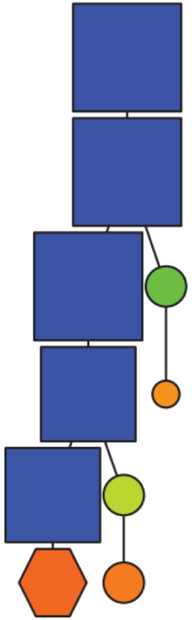
- s-clique degree of a r-clique: Number of s-cliques each r-clique participates in
- (r, s)-nucleus decomposition: Repeatedly find + “delete” r-clique with min s-clique degree

2-( , ) nuclei
(r = 2, s = 3)

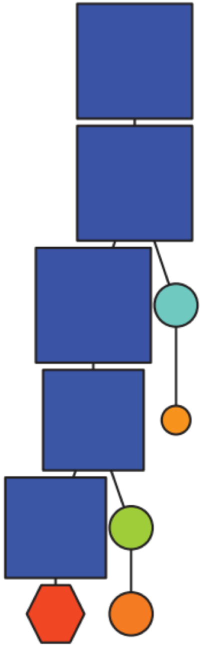


(r = 2, s = 3 is also known as k-truss)

(r, s)-nucleus decomposition

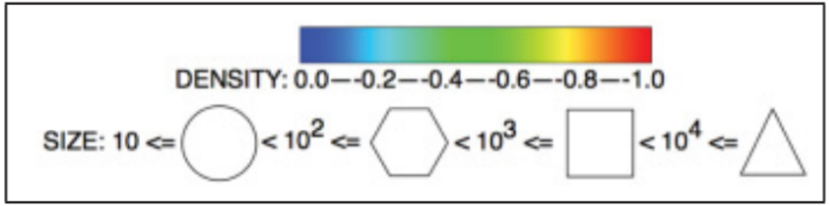


(1, 2)-nuclei = k-core



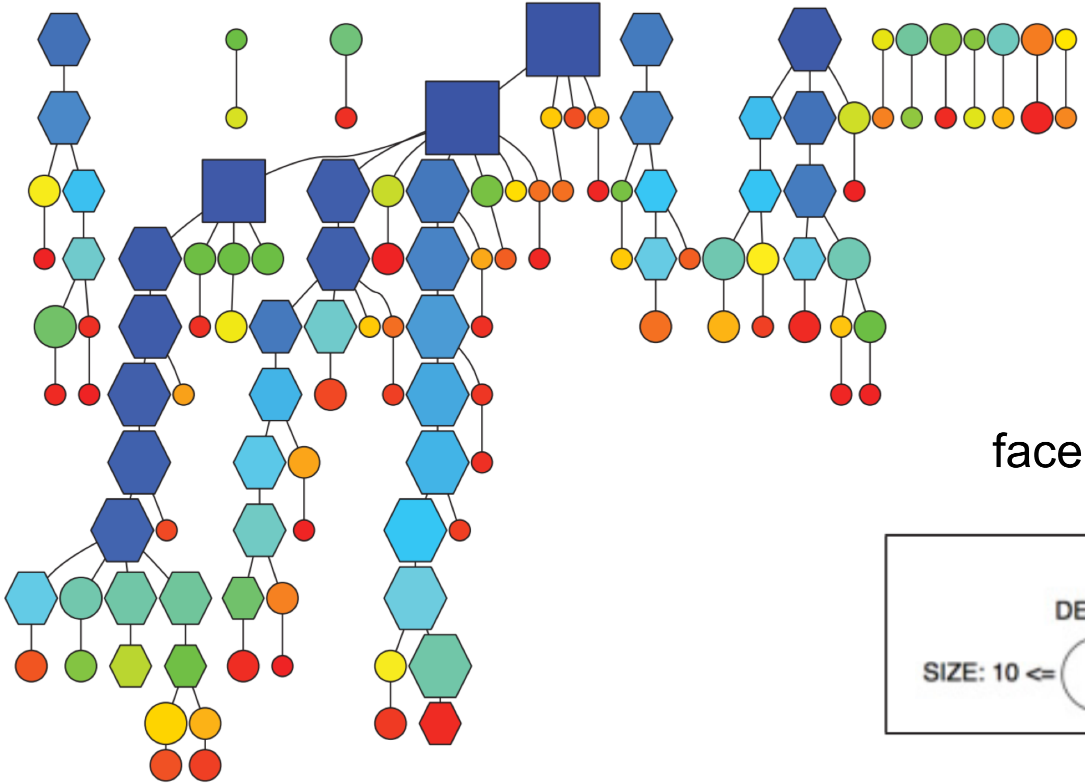
(1, 3)-nuclei =
triangle-peeling

facebook graph (88k edges)



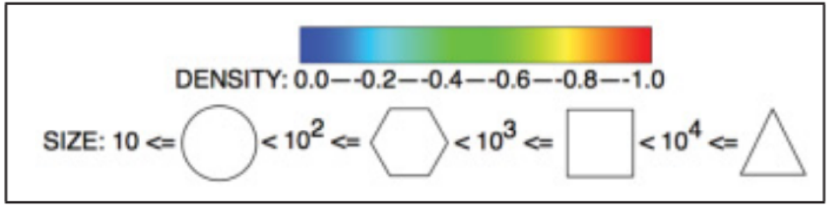
Sariyuce, Seshadhri, Pinar, Catalyurek (2017)

(r, s)-nucleus decomposition



(3, 4)-nuclei

facebook graph (88k edges)



Sariyuce, Seshadhri, Pinar, Catalyurek (2017)

Main results

- New shared-memory parallel algorithms for nucleus decomposition with strong theoretical guarantees

- Comprehensive evaluation, showing we outperform state-of-the-art parallel algorithms by a couple orders of magnitude

Computational barriers: Sequential subgraph decomposition can be slow

- Environment: 30-core GCP instance (2-way hyperthreading), 240 GiB main memory

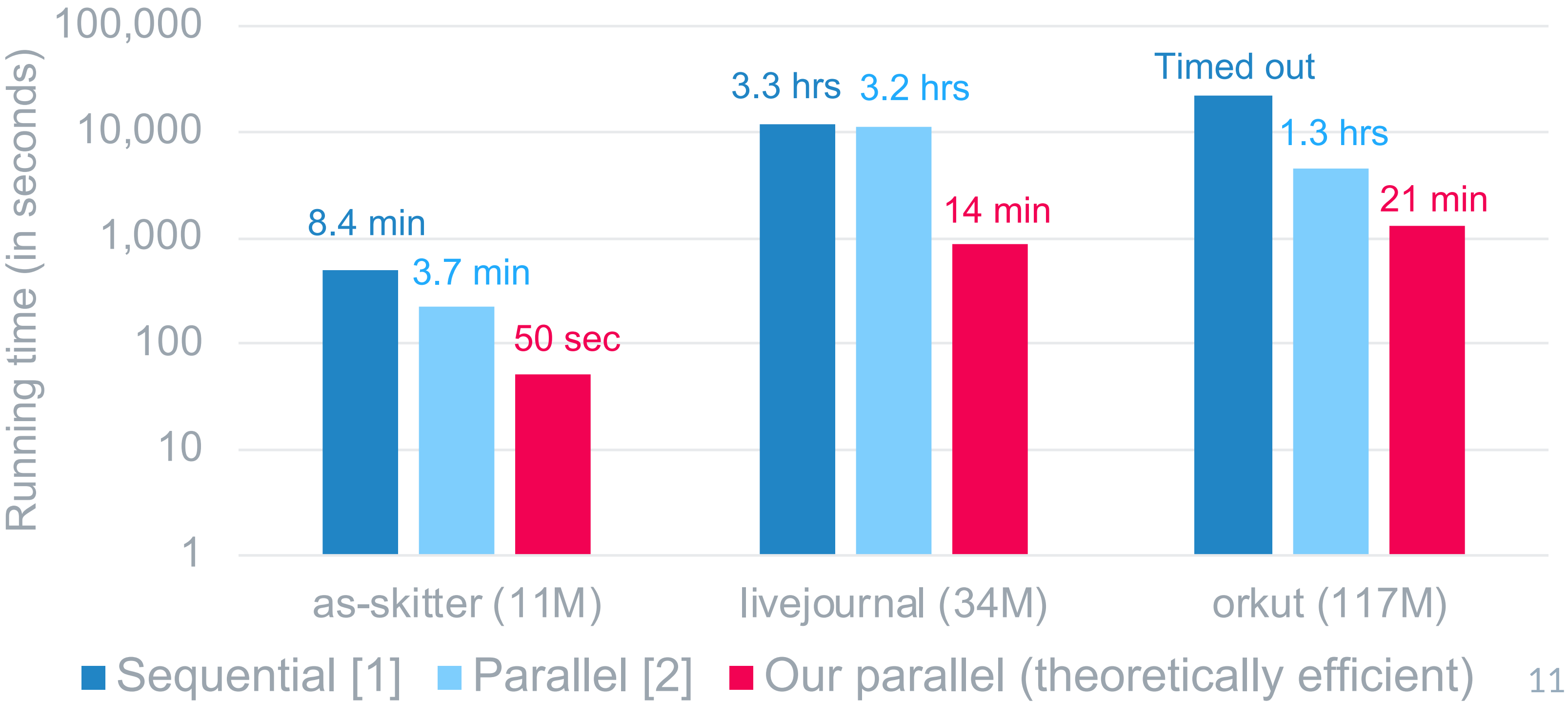
Graph	# Edges	Sequential (3, 4)-nucleus decomp [1]
as-skitter	11 million	8.5 minutes
livejournal	34 million	3.3 hours
orkut	117 million	> 6 hours

- Goal: < 15 min

[1] Sariyuce, Seshadhri, Pinar, Catalyurek (2017)

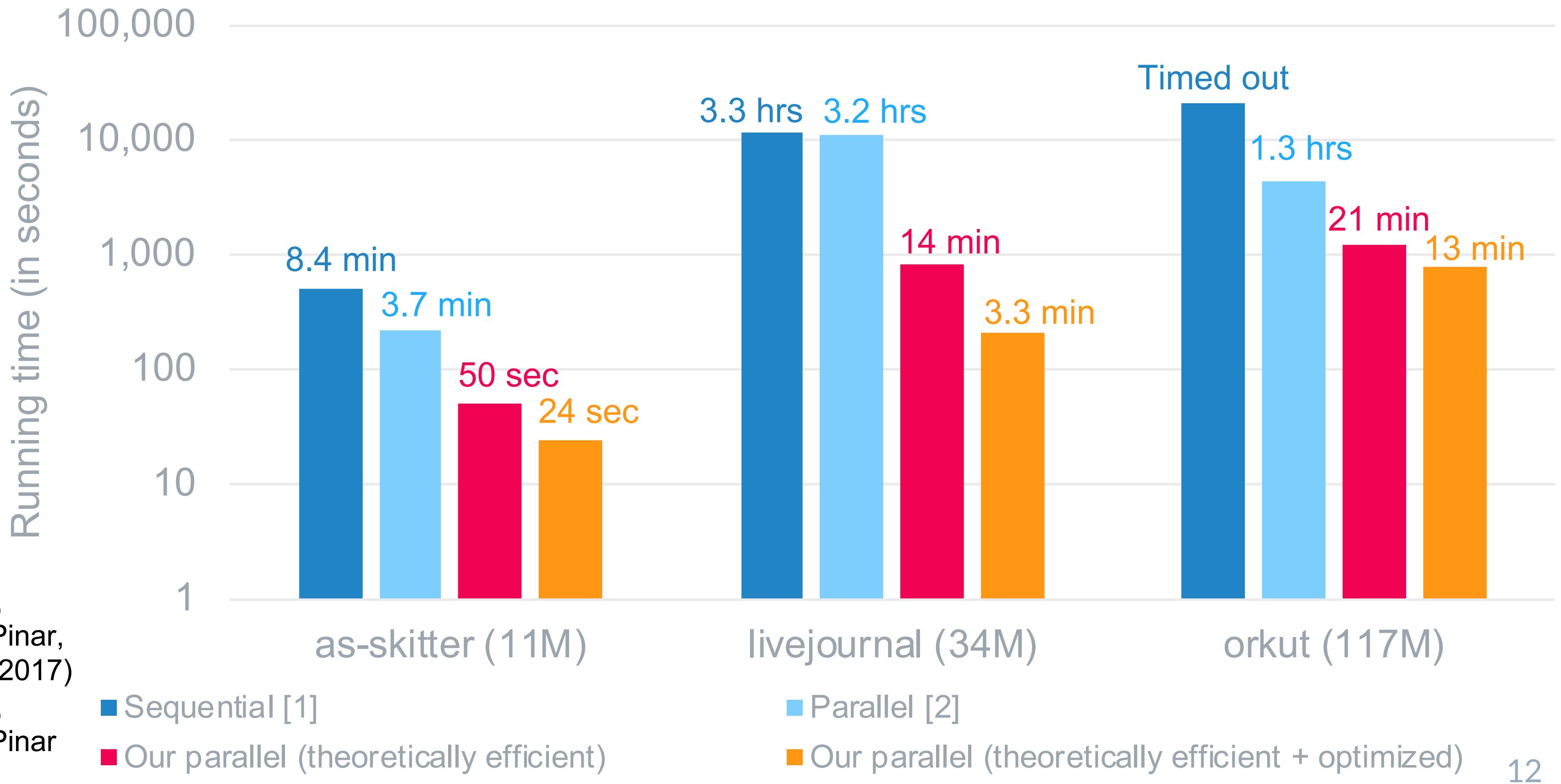
Theoretically efficient algorithms are fast

- Previous parallel nucleus decomposition [2]: Not theoretically efficient



[1] Sariyuce, Seshadhri, Pinar, Catalyurek (2017)
[2] Sariyuce, Seshadhri, Pinar (2018)

Practical optimizations



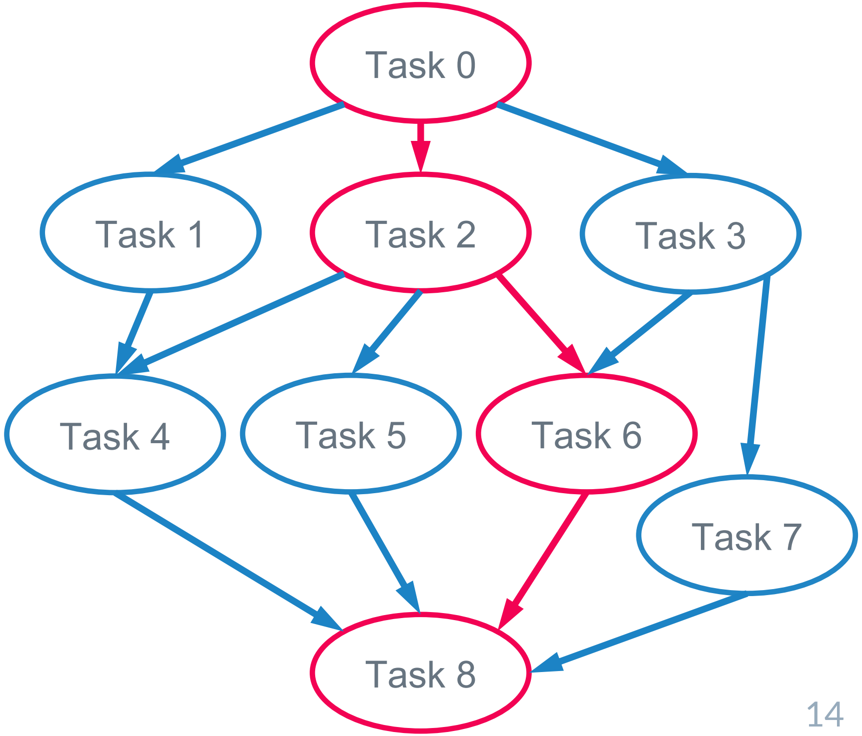
[1] Sariyuce, Seshadhri, Pinar, Catalyurek (2017)
[2] Sariyuce, Seshadhri, Pinar (2018)

Preliminaries

Preliminaries

- Work = total # operations
- Span = longest dependency path
- Running time ≤ (work / # processors) + O(span)
- Work-efficient = work matches best sequential time complexity

Parallel computation graph



Graph orientation

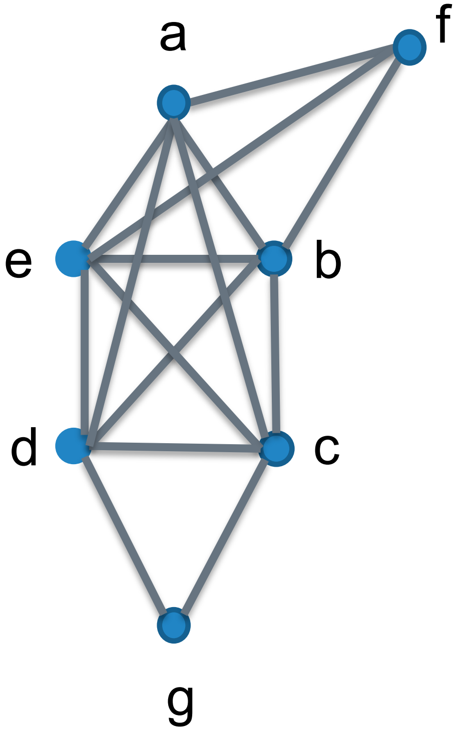
- α = arboricity = minimum # of spanning forests needed to cover all edges of the graph
  - Upper bounded by $O(\sqrt{m})$ where m = # edges

- c-orientation: Direct graph such that each vertex's out-degree is upper bounded by c
- Arboricity orientation: $O(\alpha)$-orientation
- Our prior work: Two theoretically efficient arboricity orientation algorithms [1]

[1] Shi, Dhulipala, Shun (2021)

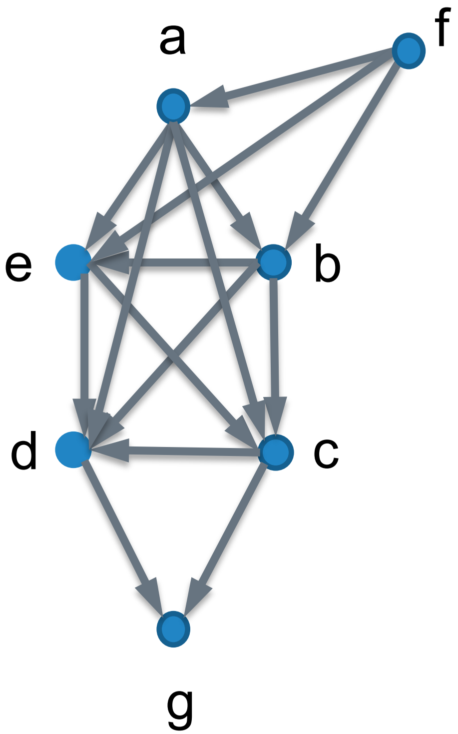
Parallel nucleus decomposition

(r, s)-nucleus decomposition (r=3, s=4)



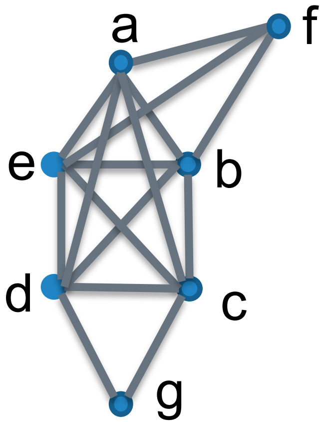
- Direct the graph (DG) using an arboricity orientation
- Count # s-cliques per r-clique using DG
- Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

(r, s)-nucleus decomposition (r=3, s=4)



- Direct the graph (DG) using an arboricity orientation
- Count # s-cliques per r-clique using DG
- Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

(r, s)-nucleus decomposition (r=3, s=4)



No 4-cliques: cdg

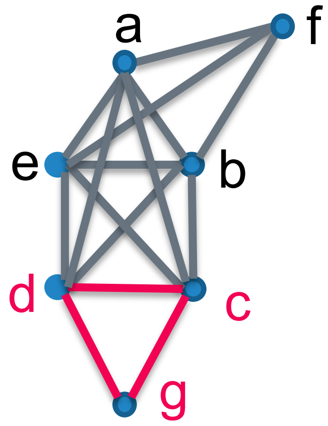
One 4-clique: All triples in {a,b,e,f} except abe

Two 4-cliques: All triples in {a,b,c,d,e} except abe

Three 4-cliques: abe

- Direct the graph (DG) using an arboricity orientation
- Count # s-cliques per r-clique using DG
- Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

(r, s)-nucleus decomposition (r=3, s=4)



No 4-cliques: cdg

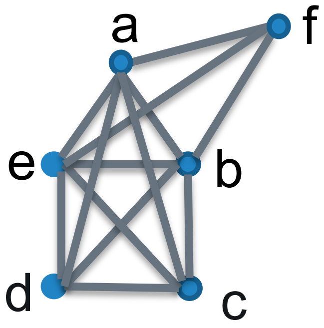
One 4-clique: All triples in {a,b,e,f} except abe

Two 4-cliques: All triples in {a,b,c,d,e} except abe

Three 4-cliques: abe

- Direct the graph (DG) using an arboricity orientation
- Count # s-cliques per r-clique using DG
- Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

(r, s)-nucleus decomposition (r=3, s=4)



No 4-cliques:

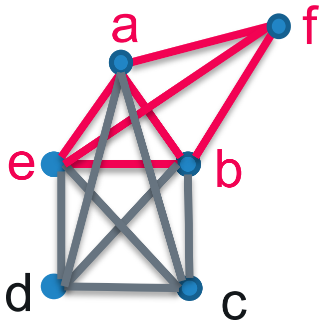
One 4-clique: All triples in {a,b,e,f} except abe

Two 4-cliques: All triples in {a,b,c,d,e} except abe

Three 4-cliques: abe

- Direct the graph (DG) using an arboricity orientation
- Count # s-cliques per r-clique using DG
- Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

(r, s)-nucleus decomposition (r=3, s=4)



No 4-cliques:

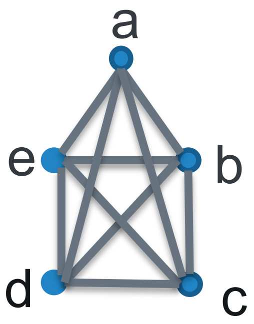
One 4-clique: All triples in {a,b,e,f} except abe

Two 4-cliques: All triples in {a,b,c,d,e} except abe

Three 4-cliques: abe

- Direct the graph (DG) using an arboricity orientation
- Count # s-cliques per r-clique using DG
- Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

(r, s)-nucleus decomposition (r=3, s=4)



No 4-cliques:

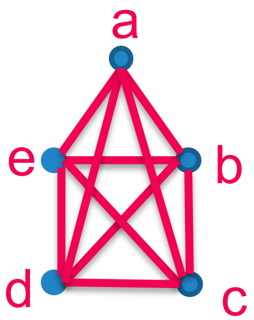
One 4-clique:

Two 4-cliques: All triples in {a,b,c,d,e}

Three 4-cliques:

- ▷ Direct the graph (DG) using an arboricity orientation
- ▷ Count # s-cliques per r-clique using DG
- ▷ Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- ▷ While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

(r, s)-nucleus decomposition (r=3, s=4)



No 4-cliques:

One 4-clique:

Two 4-cliques: All triples in {a,b,c,d,e}

Three 4-cliques:

- Direct the graph (DG) using an arboricity orientation
- Count # s-cliques per r-clique using DG
- Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

(r, s)-nucleus decomposition (r=3, s=4)

- Direct the graph (DG) using an arboricity orientation
- Count # s-cliques per r-clique using DG
- Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

No 4-cliques:

One 4-clique:

Two 4-cliques:

Three 4-cliques:

(r, s)-nucleus decomposition

$O(m)$ work, $O(\log^2 n)$ span

$O(m\alpha^{s-2})$ work,
$O(s \log n)$ span whp

- Direct the graph (DG) using an arboricity orientation
- Count # s-cliques per r-clique using DG
- Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

Shi, Dhulipala, Shun (2021)

(r, s)-nucleus decomposition

$O(m)$ work, $O(\log^2 n)$ span

$O(m\alpha^{s-2})$ work,
$O(s \log n)$ span whp

Subgoal 1

Subgoal 2

- Direct the graph (DG) using an arboricity orientation
- Count # s-cliques per r-clique using DG
- Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

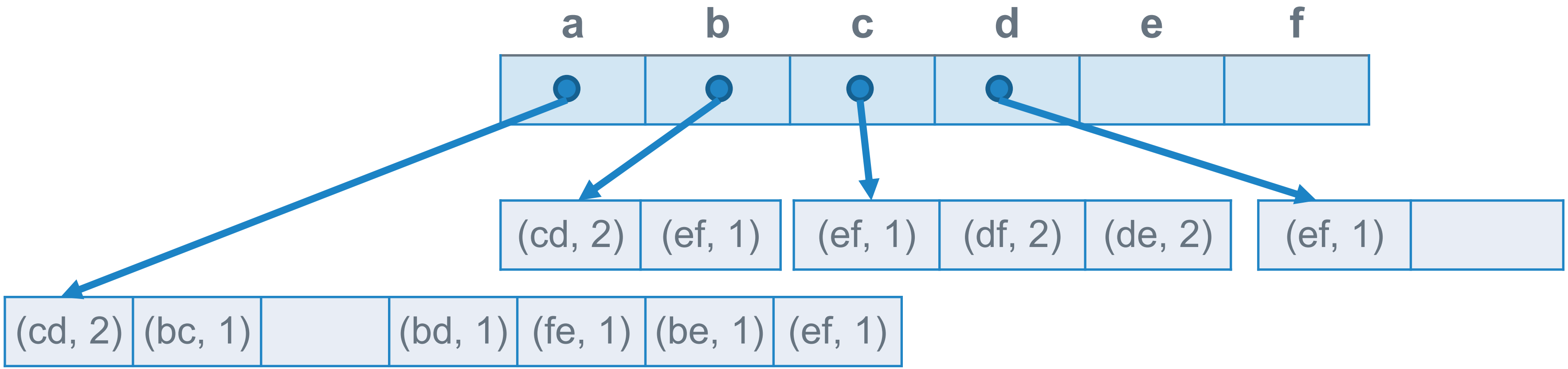
How do we peel r-cliques?

- Subgoal 1: A way to keep track of r-cliques with min s-clique count
- In theory: Use a batch-parallel Fibonacci heap [1]
  - k insertions: $O(k)$ amortized expected work, $O(\log n)$ span whp
  - Extract min: $O(\log n)$ amortized expected work, $O(\log n)$ span whp
- In practice: Fibonacci heaps are not efficient
  - Julienne: Efficient parallel bucketing structure [2]

[1] Shi, Shun (2020)
[2] Dhulipala, Blelloch, Shun (2017)

In practice: Store r-cliques

- To save space:
- Two-level array and hash table:



Refer to r-cliques by index in last-level tables

Additional optimization for cache behavior: Store last-level tables contiguously in memory

In practice: Store r-cliques

- To save space:

- Space savings compared to standard hash table:
- Up to 1.8x reduction in space usage on (2, 3)-nucleus and (2, 4)-nucleus
- Up to 2.2x reduction in space usage on (3, 4)-nucleus
- Up to 2.5x reduction in space usage on (4, 5)-nucleus

(cd, 2)	(bc, 1)		(bd, 1)	(fe, 1)	(be, 1)	(ef, 1)

Refer to r-cliques by index in last-level tables

Additional optimization for cache behavior: Store last-level tables contiguously in memory

(r, s)-nucleus decomposition

$O(m)$ work, $O(\log^2 n)$ span

- Direct the graph (DG) using an arboricity orientation

$O(m\alpha^{s-2})$ work,
$O(s \log n)$ span whp

- Count # s-cliques per r-clique using DG

$O(m\alpha^{r-2} + \rho \log n)$ amortized expected work,
$O(\rho \log n)$ span whp

where ρ = # rounds to peel entire graph

- Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

Subgoal 2 →

How do we update s-clique counts?

- Subgoal 2: A way to update s-clique counts after "deleting" r-cliques
  - In theory and practice: We use a key lemma that improves upon the previous best theoretical bounds for sequential nucleus decomposition
  - In practice: Also use software optimizations

Theoretically: Update s-clique counts

- Subgoal 2: A way to update s-clique counts after "deleting" r-cliques

- Modify parallel s-clique counting subroutine to efficiently obtain updated s-clique counts from "deleted" r-cliques

- Theorem: Over all c-cliques in a graph $C_c = \{v_1, \ldots, v_c\}$, $\sum_{C_c} \min_{1 \le i \le c} \deg(v_i) = O(m\alpha^{c-1})$. [1]

[1] Eden, Ron, Seshadhri 2020

(r, s)-nucleus decomposition

$O(m)$ work, $O(\log^2 n)$ span

$O(m\alpha^{s-2})$ work,
$O(s \log n)$ span whp

$O(m\alpha^{r-2} + \rho \log n)$ amortized expected work,
$O(\rho \log n)$ span whp

where ρ = # rounds to peel entire graph

$O(m\alpha^{s-2})$ amortized expected work, $O(\rho \log n)$ span whp

- Direct the graph (DG) using an arboricity orientation
- Count # s-cliques per r-clique using DG
- Construct a bucketing structure mapping r-cliques to a bucket based on # s-cliques
- While not all r-cliques have been peeled:
  - Peel set of r-cliques with minimum s-clique count
  - Update s-clique counts of remaining r-cliques

(r, s)-nucleus decomposition

- Practical optimizations:
- Up to a 5x speedup over our unoptimized parallel nucleus decomposition

$O(m)$ work, $O(\log^2 n)$ span ▷ Direct the graph (DG) using an arboricity orientation

$O(m$

$O(s$

g r-

$O(m$

amo

$O(\rho \log n)$ span whp

where ρ = # rounds to peel entire graph

- ○ Peel set of r-cliques with minimum s-clique count
- ○ Update s-clique counts of remaining r-cliques

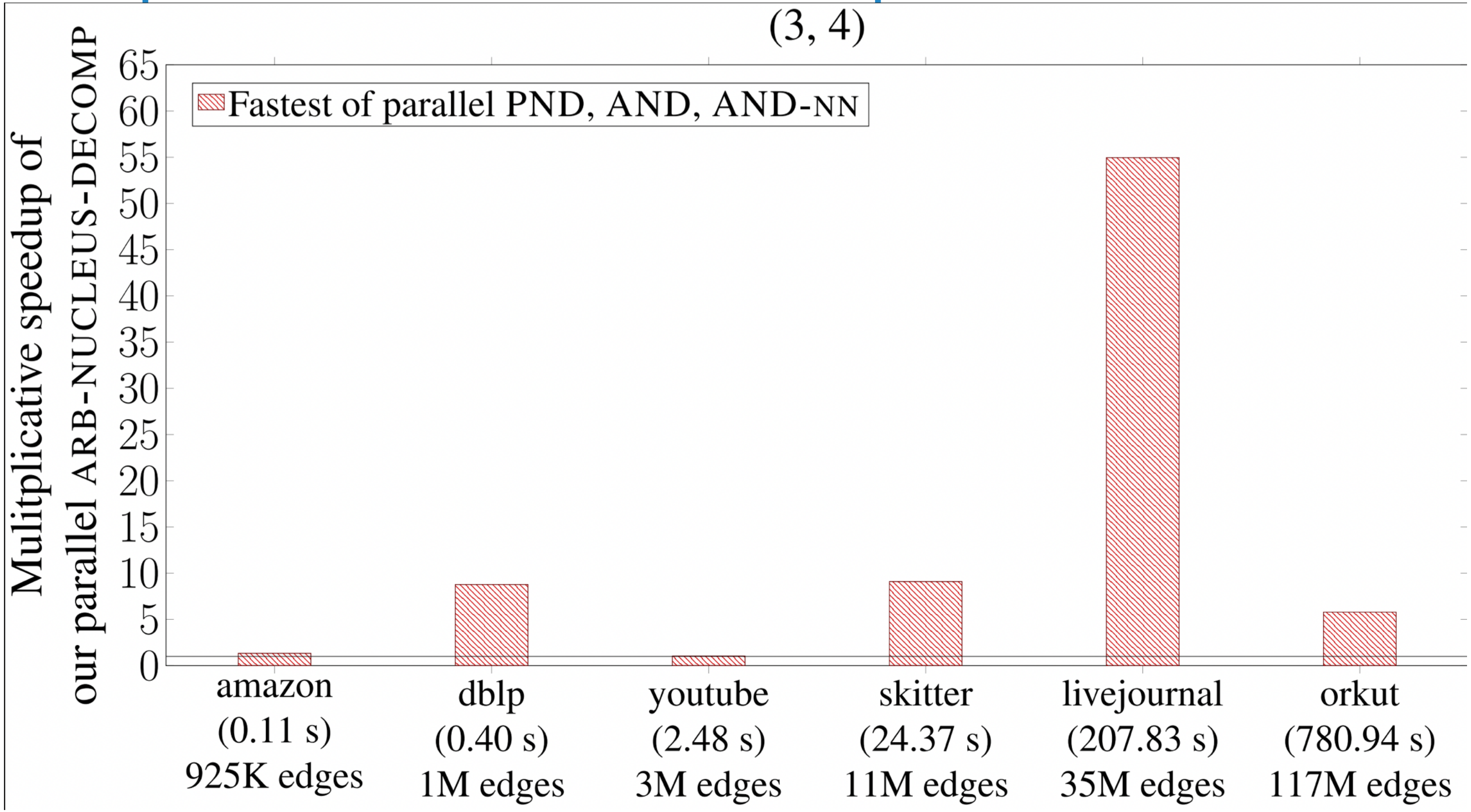
$O(m\alpha^{s-2})$ amortized expected work, $O(\rho \log n)$ span whp

Experiments

Environment

- 30-core GCP instance (2-way hyperthreading), 240 GiB main memory
- Used real-world Stanford Network Analysis Platform (SNAP) graphs

Comparison to other implementations



AND, AND-NN, PND: Sariyuce, Seshadhri, Pinar (2018)

Other implementations are not theoretically efficient

- Speedups up to 55x, median 9x over fastest of PND, AND, AND-NN ($r = 3, s = 4$)
- Up to 40x self-relative speedups ($r < s \leq 7$)

- PND, AND, AND-NN have large span, are not work-efficient, or are not space-efficient (runs OOM)

Conclusion

Conclusion

- Summary:
  - Shared-memory parallel clustering algorithms developed with strong theoretical guarantees + practical optimizations = highly efficient and scalable implementations
- Future directions:
  - Dynamic nucleus decomposition
  - Other subgraph decompositions for other classes of graphs (e.g., bipartite graphs)
    - Generalization of (α, β)-decomposition

Conclusion

- Nucleus Decomposition Github: https://github.com/jeshi96/arb-nucleus-decomp

- Contact me: jeshi@mit.edu

Thank you!

In practice: Keep track of r-cliques

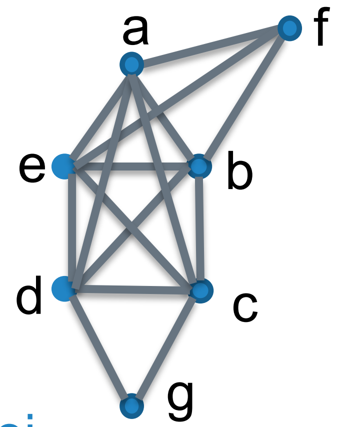
- Subgoal 1: A way to keep track of r-cliques with min s-clique count

- Julienne: Efficient parallel bucketing structure [1]
- Requirement 1: Map r-cliques to unique keys
- Requirement 2: Obtain constituent r-clique vertices from keys

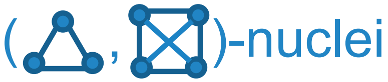
[1] Dhulipala, Blelloch, Shun (2017)

In practice: Keep track of r-cliques

- **Julienne**: Efficient parallel bucketing structure [1]



- Bucket # = # of four-cliques
- Each key in the buckets corresponds to a triangle
  - e.g., key 0 = cdg, key 1 = abe



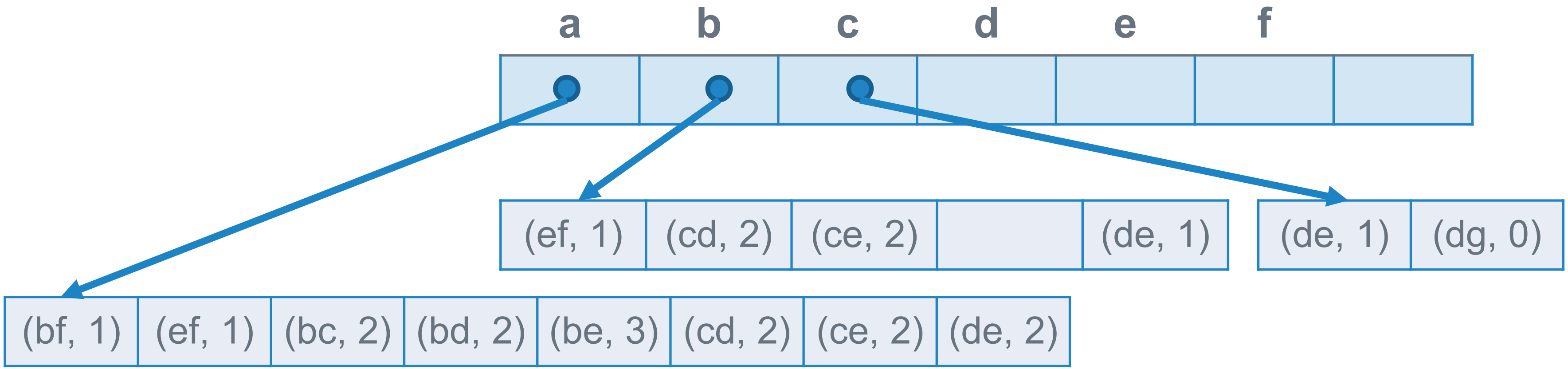
Julienne:

Bucket 0	Bucket 1	Bucket 2	Bucket 3
0	2, 6, 7	3, 4, 5, 8, 9, 10, 11, 12, 13	1

[1] Dhulipala, Blelloch, Shun (2017)

In practice: Map r-cliques to keys

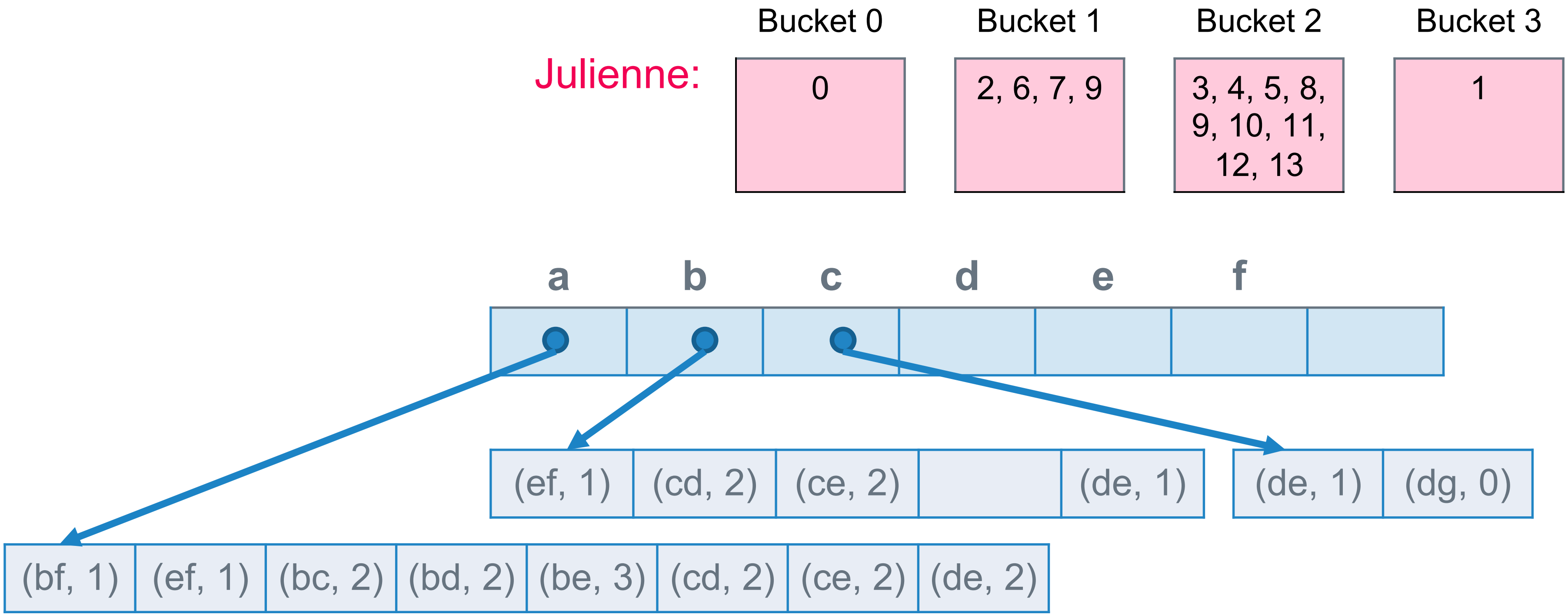
- An option for space savings:
- Two-level array and hash table:



Keys = index of r-clique in last-level tables, Values = # s-cliques

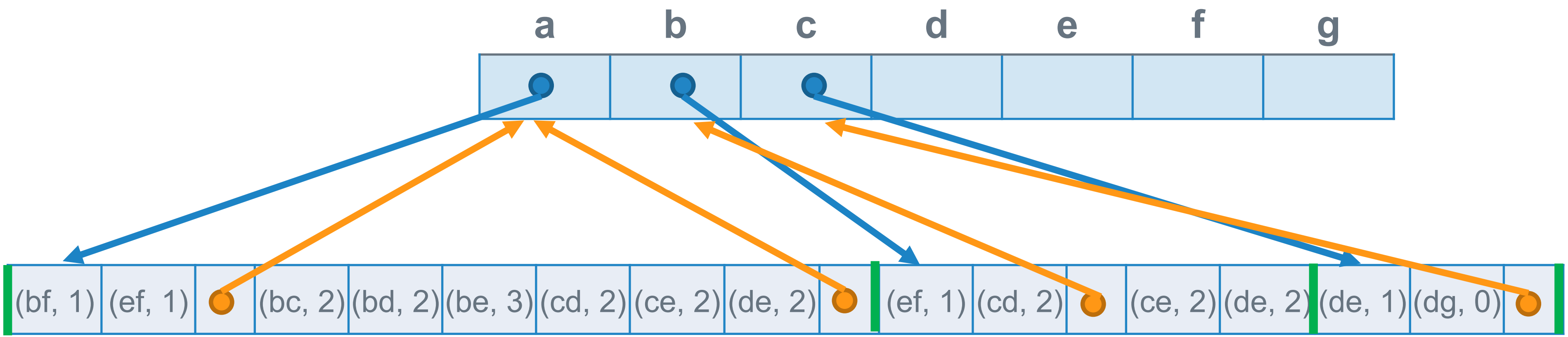
Additional optimization for cache behavior: Store last-level tables contiguously in memory

In practice: Obtain r-clique vertices from keys



In practice: Obtain r-clique vertices from keys

- Stored pointers:



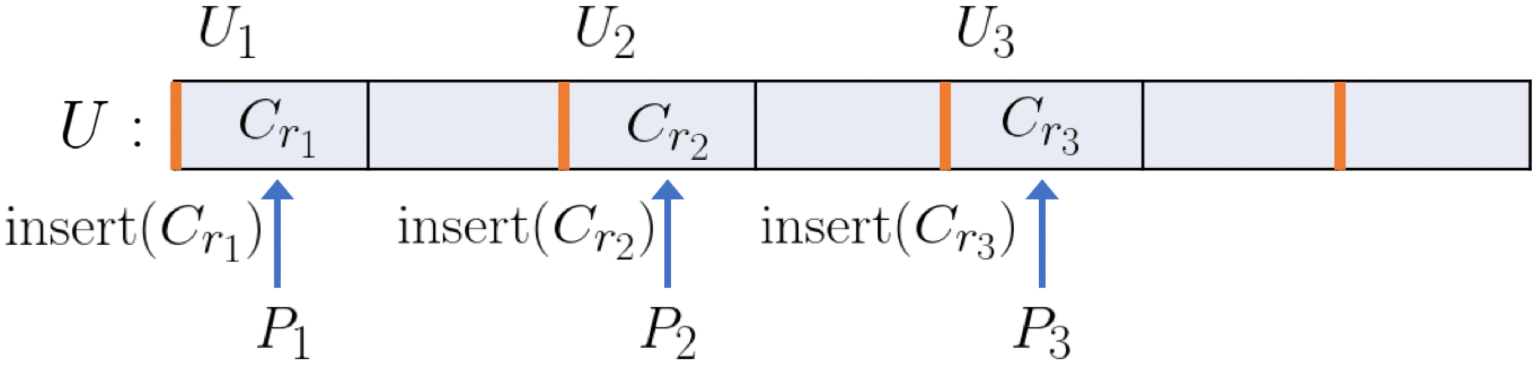
In practice: Update s-clique counts

- Subgoal 2: A way to update s-clique counts after "deleting" r-cliques

- How do we aggregate r-cliques with updated s-clique counts in parallel?

In practice: Obtain set of updated r-cliques

- List buffer:



- Contention only when getting a new block

Other implementations are not theoretically efficient

- PND: Large span (*> 80,000x sequential rounds compared to our alg*)
- AND: Not work-efficient (*up to 46x # of 4-cliques discovered compared to our alg*)
- AND-NN: Not work-efficient and not space-efficient (*up to 3.5x # of 4-cliques discovered compared to our alg, out of memory for skitter, livejournal, and orkut*)

Comparison to other implementations

- Up to 55x speedups over PND (average 23x)
- Up to 60x speedups over AND (average 14x)
- Up to 9x speedups over AND-NN (average 3x)
- AND-NN runs out of memory on graphs with > 11 million edges
- Up to 40x self-relative parallel speedups

(3, 4)

925K edges | 1.05M edges | 2.99M edges | 11.1M edges | 34.7M edges | 117M edges

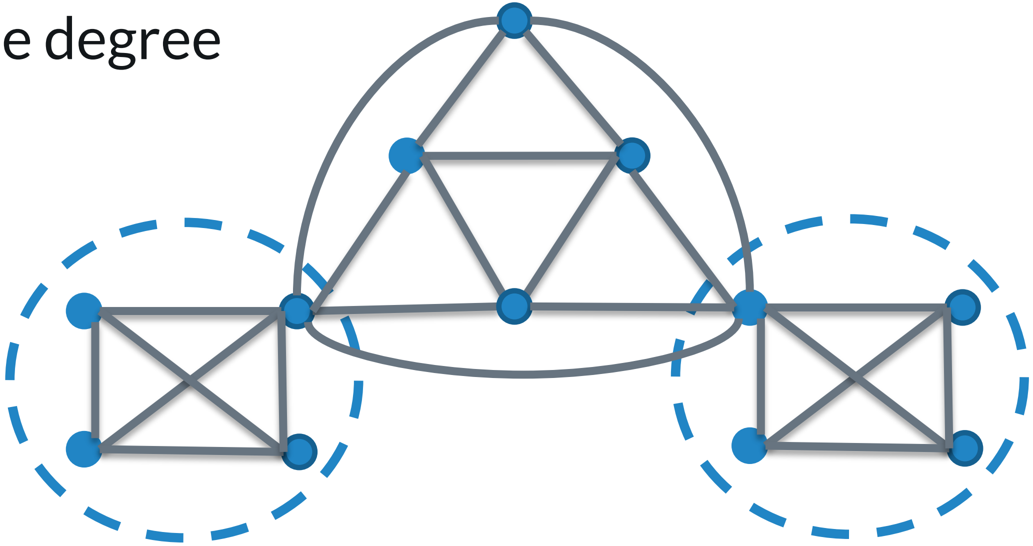
ND: Sariyuce, Seshadhri, Pinar, Catalyurek (17)
AND, AND-NN, PND: Sariyuce, Seshadhri, Pinar (18)

(r, s)-nucleus decomposition

- s-clique degree of a r-clique: Number of s-cliques each r-clique participates in
- (r, s)-nucleus decomposition: Repeatedly find + "delete" r-clique with min s-clique degree

Entire graph is in a 3-triangle-core

Entire graph is in a 2-(2, 3) nucleus



(r, s)-nucleus decomposition

- s-clique degree of a r-clique: Number of s-cliques each r-clique participates in
- (r, s)-nucleus decomposition: Repeatedly find + “delete” r-clique with min s-clique degree

1-(3, 4) nuclei
(r = 3, s = 4)