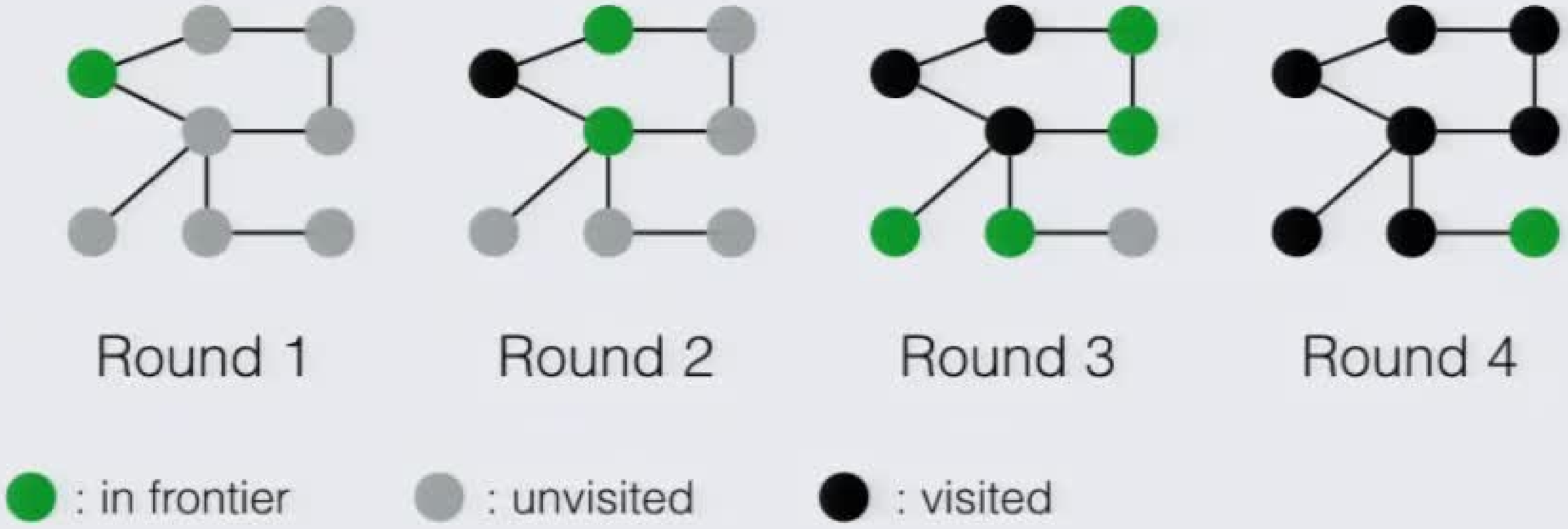
Ligra

Shared memory framework for *frontier-based algorithms*

Algorithm:

- Runs over a sequence of *rounds*
- Each round, the *frontier,* a subset of vertices is processed
- Terminates once the frontier becomes *empty*

Breadth-First Search:



Julienne

Julienne

Shared memory framework for *bucketing-based algorithms*

Algorithm:

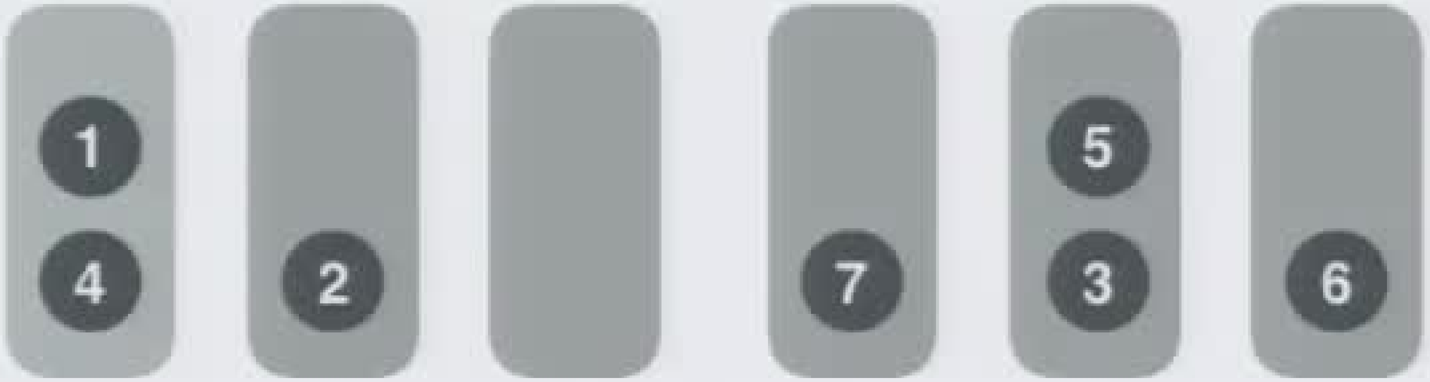
- Runs over a sequence of *rounds*
- Vertices are stored in a set of *ordered buckets*
- Each round, vertices in the *next bucket* are processed
- Terminates once the bucket structure is empty

Julienne

Shared memory framework for *bucketing-based algorithms*

Algorithm:

- Runs over a sequence of *rounds*
- Vertices are stored in a set of *ordered buckets*
- Each round, vertices in the *next bucket* are processed
- Terminates once the bucket structure is empty



Bucketing interface:

- Maintains dynamic mapping from *identifiers* to *buckets*
- Identifiers can represent vertices, edges, triangles, etc

Julienne

Shared memory framework for *bucketing-based algorithms*

Framework extends Ligra with:

- Interface for bucketing
- Work-efficient parallel implementation of the interface

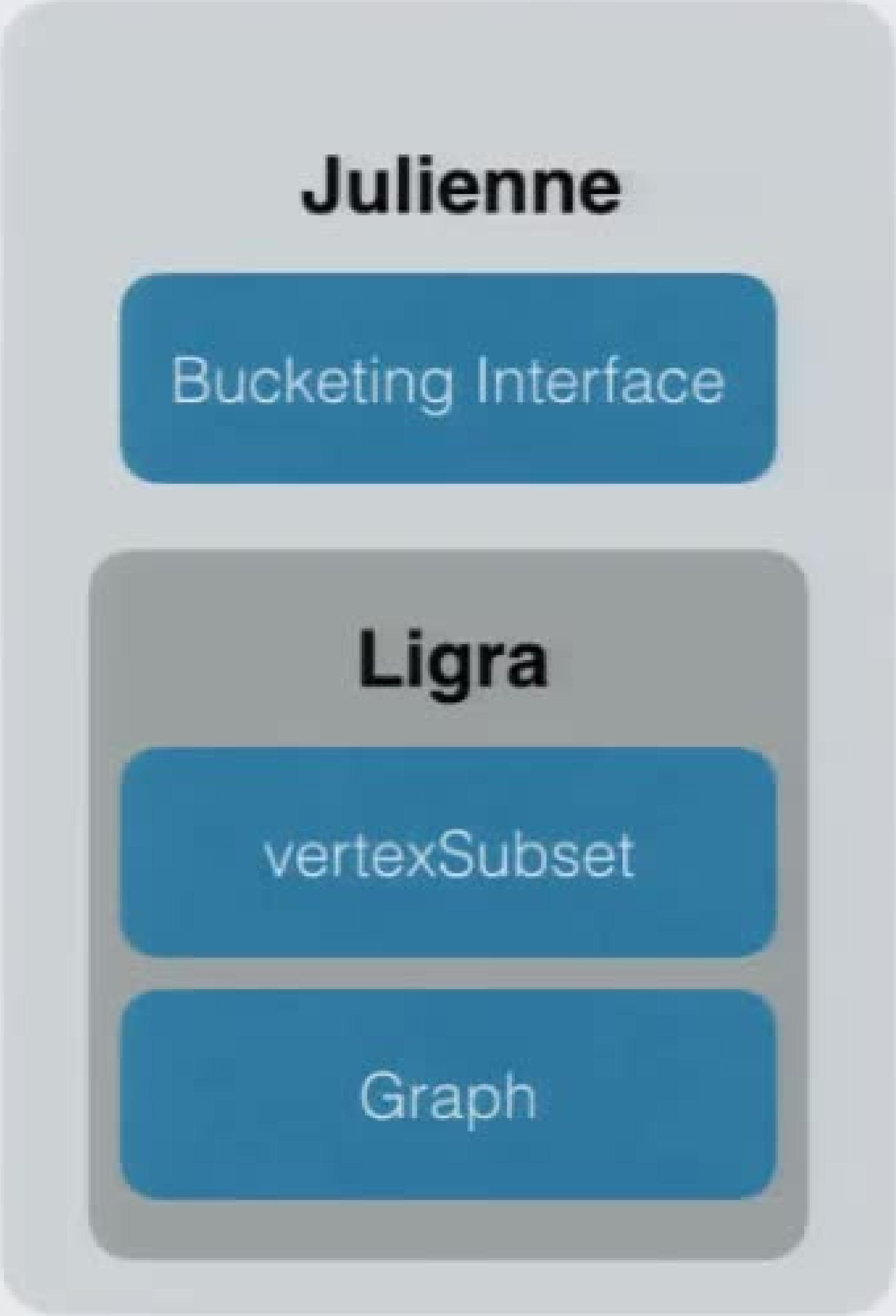
Implementations of:

- k-core
- Weighted Breadth-First Search
- Delta-Stepping
- Parallel Approximate Set Cover

This talk

Please see the paper for more details!

Julienne: Interface



Julienne

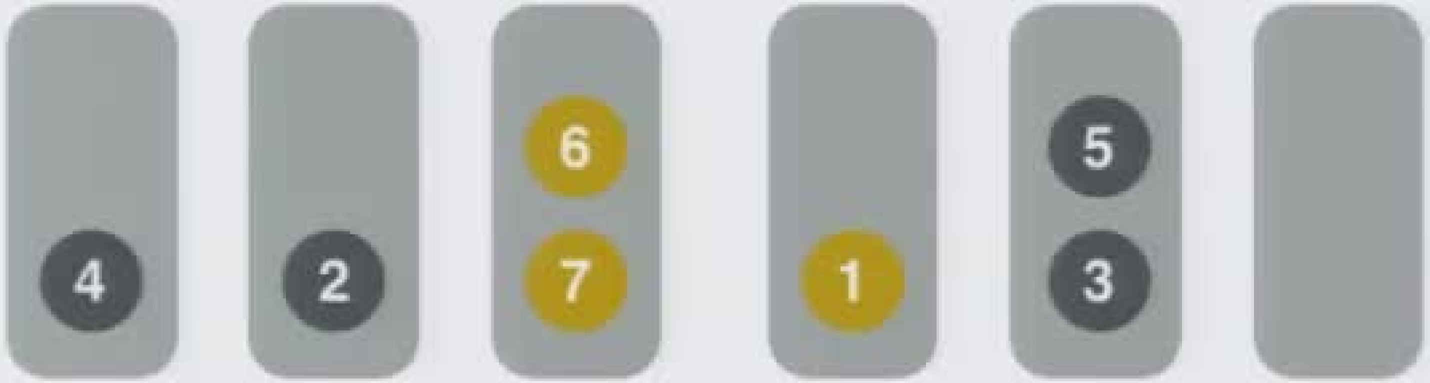
Bucketing Interface

Ligra

vertexSubset

Graph

Julienne: Interface



[(1,3), (7,2), (6,2)]

UpdateBuckets

k : int

F : int → (identifier, bucket_dest)

Update buckets for k identifiers

Parallel Bucketing

Can implement parallel bucketing with:

- n identifiers
- T total buckets
- K calls to UpdateBuckets, where each updates the ids in S_i
- L calls to NextBucket

in $O(n + T + \sum_{i=0}^{K} |S_i|)$ expected work and

$O((K + L) \log n)$ depth w.h.p.

Implementation:

- Use dynamic arrays
- MakeBuckets: call UpdateBuckets. NextBucket: parallel filter

Parallel Bucketing

UpdateBuckets:

- Use work-efficient semisort [Gu et al. 2015]
- Given k (key, value) pairs, semisorts in $O(k)$ expected work and $O(\log k)$ depth w.h.p.

$$[(3,9), (4,7), \ldots, (2,1), (1,1)]$$

$$\downarrow$$

$$[\underbrace{(2,1), (1,1), (7,1)}_{\text{All ids going to bucket 1}}, \ldots, (4,7), (6,7), \ldots, (3,9)]$$

- Prefix sum to compute #ids going to each bucket
- Resize buckets and inject all ids in parallel

k-core and Coreness

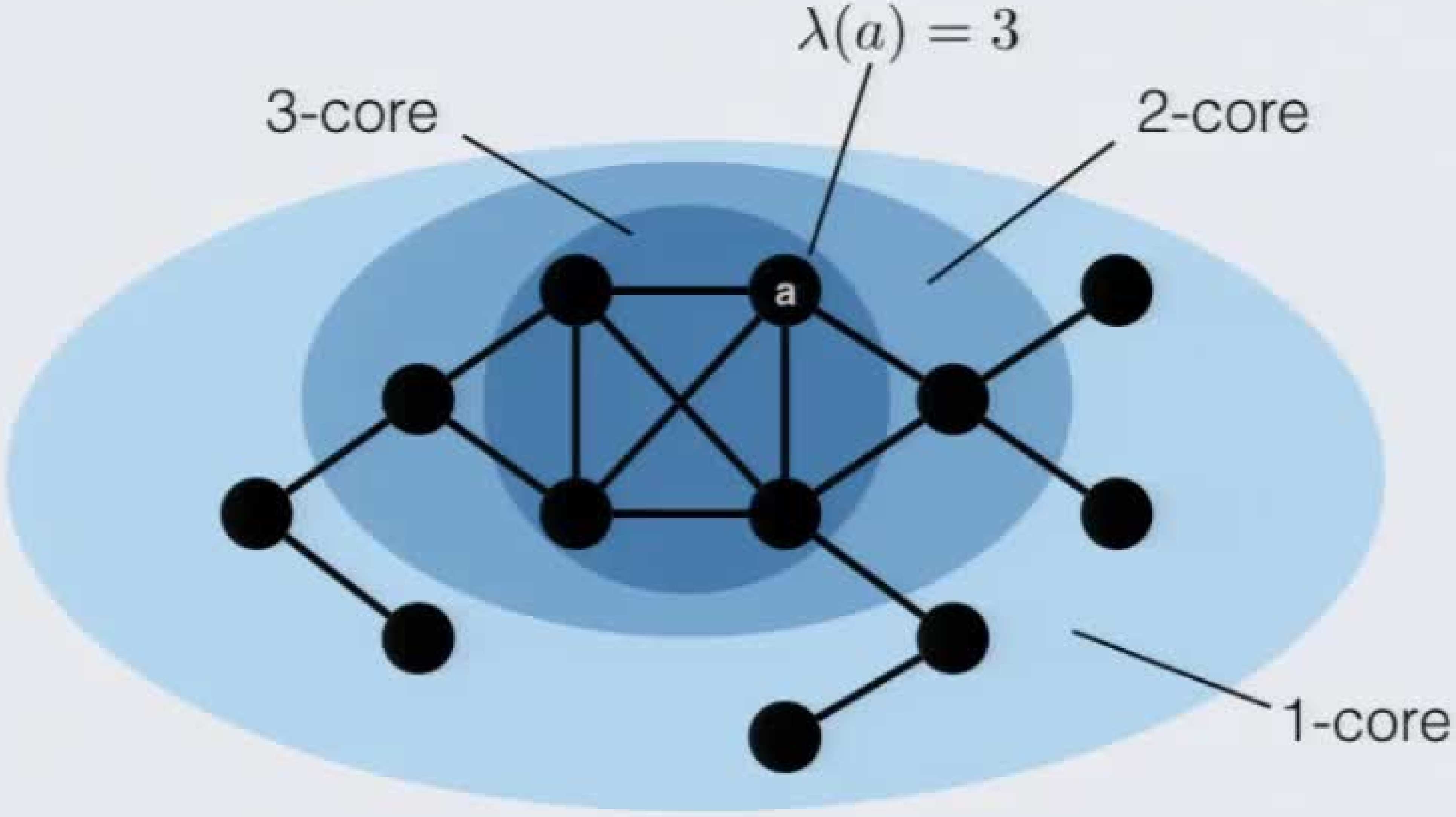
k-core : maximal connected subgraph of G s.t. all vertices have degree $\geq k$

$\lambda(v)$: largest k-core that v participates in

k-core and Coreness

k-core : maximal connected subgraph of G s.t. all vertices have degree $\geq k$

$\lambda(v)$: largest k-core that v participates in



k-core and Coreness

Sequential Peeling:

- Bucket sort vertices by degree
- Remove the minimum degree vertex, set its core number
    - Update the buckets of its neighbors

Each vertex and edge is processed exactly once:

$$W = O(|E| + |V|)$$

Existing parallel algorithms:

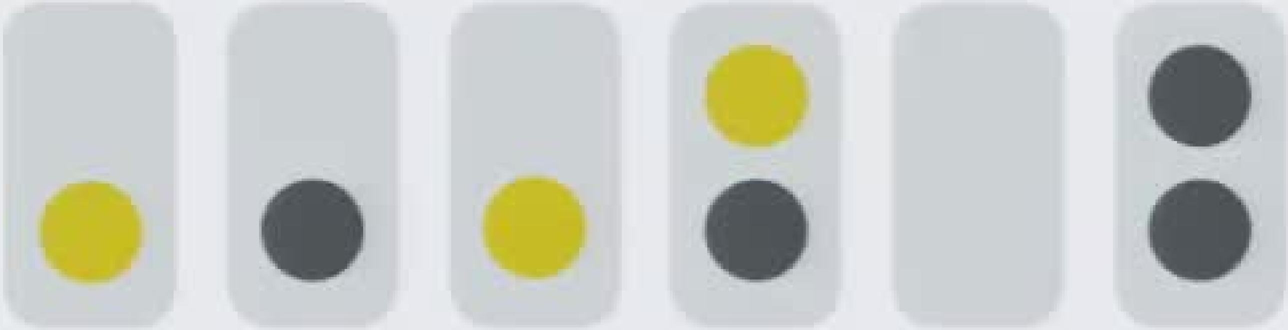
- Scan all remaining vertices when computing each core

$\rho =$ number of peeling steps done by the parallel algorithm

$$W = O(|E| + \rho|V|)$$

$$D = O(\rho \log |V|)$$

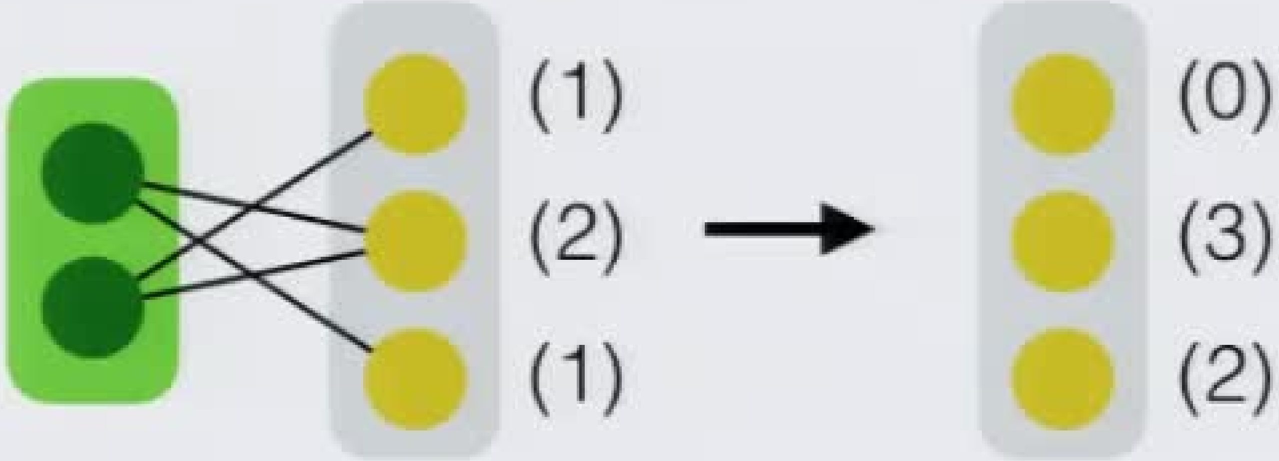
Work-efficient Peeling



Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier
3. Compute the new buckets for the neighbors
4. Update the bucket structure with the (neighbors, buckets)



Work-efficient Peeling

We process each edge at most once in each direction:

updates $= O(|E|)$

buckets $\leq |V|$

calls to NextBucket $= \rho$

calls to UpdateBuckets $= \rho$

Therefore the algorithm runs in:

$O(|E| + |V|)$ expected work

$O(\rho \log |V|)$ depth w.h.p.

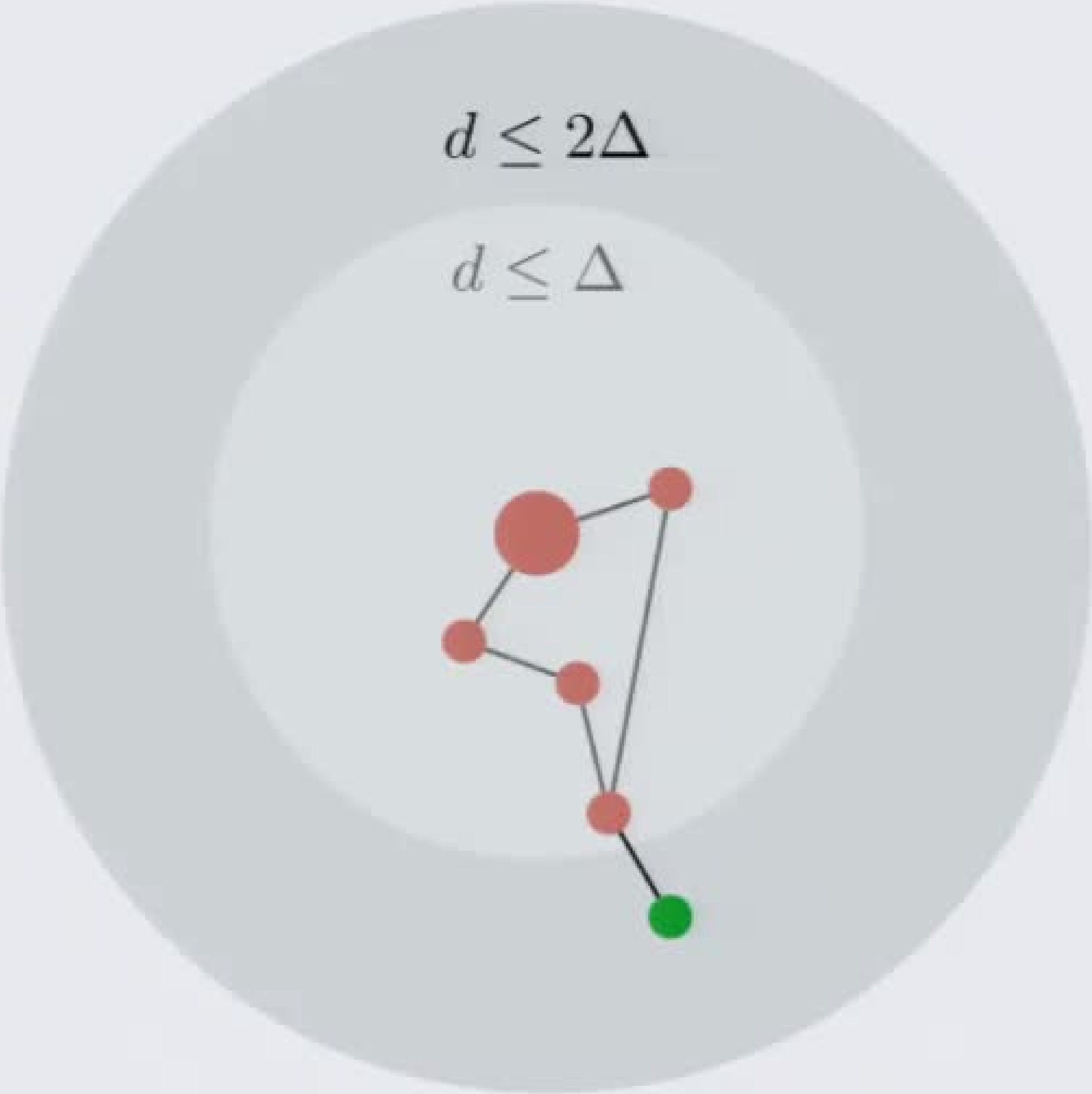
On the largest graph we test on, $\rho = 130,728$

On 72 cores, our code finishes in a few minutes, but the work-inefficient algorithm does not terminate within 3 hours

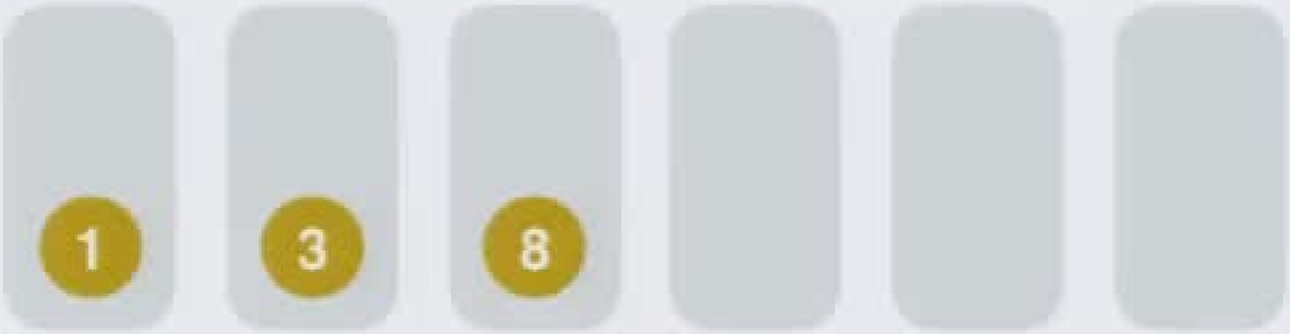
Delta-Stepping and wBFS

Idea: Only process vertices within the current annulus

$\Delta = 10$



Delta-Stepping and wBFS

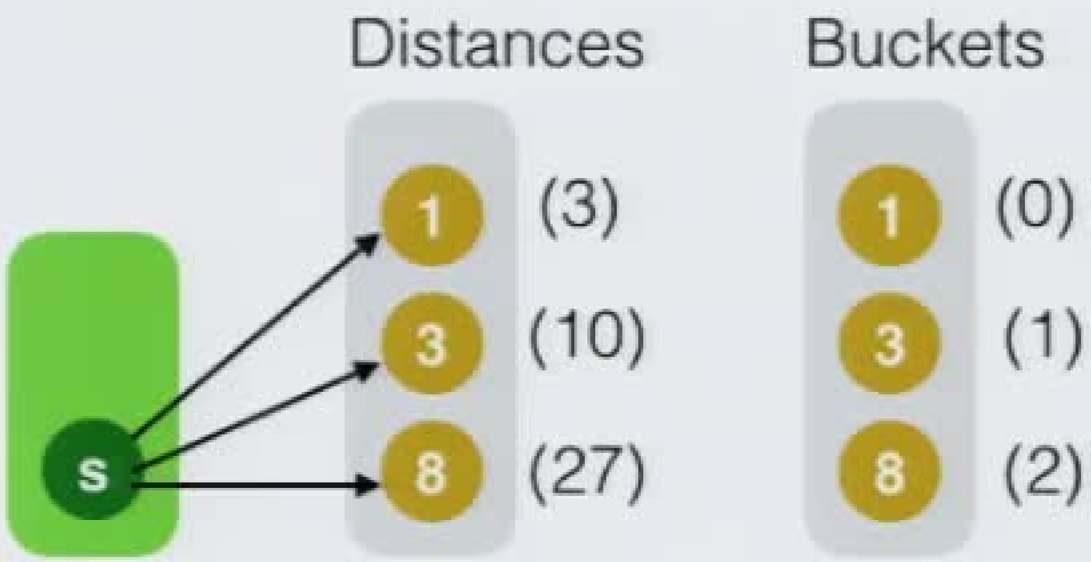


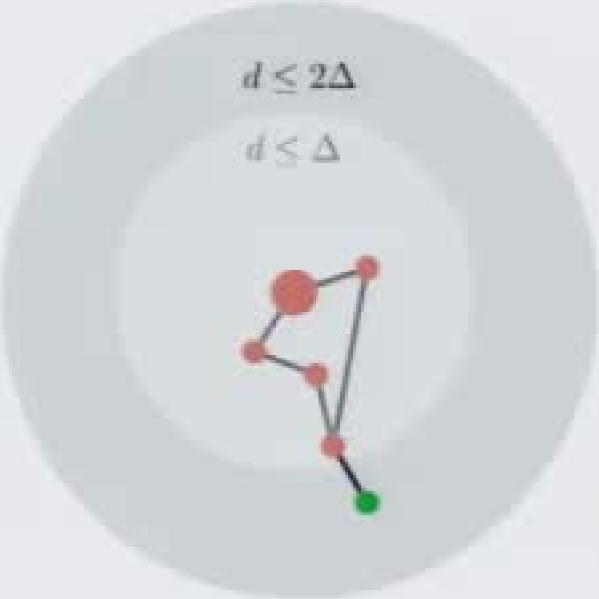
$\Delta = 10$

Insert **s** into the first bucket (annulus)

While the bucket structure is not empty:

1. Extract the next bucket
2. Relax neighbors of vertices in this bucket
3. Compute new bucket for each relaxed vertex
4. Update buckets with relaxed (vertex, bucket)





Delta-Stepping and wBFS

On a graph with constant integer edge weights, eccentricity r_{src} and $\Delta = 1$:

updates $= O(|E|)$

# of identifiers $\leq |E|$

# of buckets $\leq r_{src}$

calls to NextBucket = # calls to UpdateBuckets $\leq r_{src}$

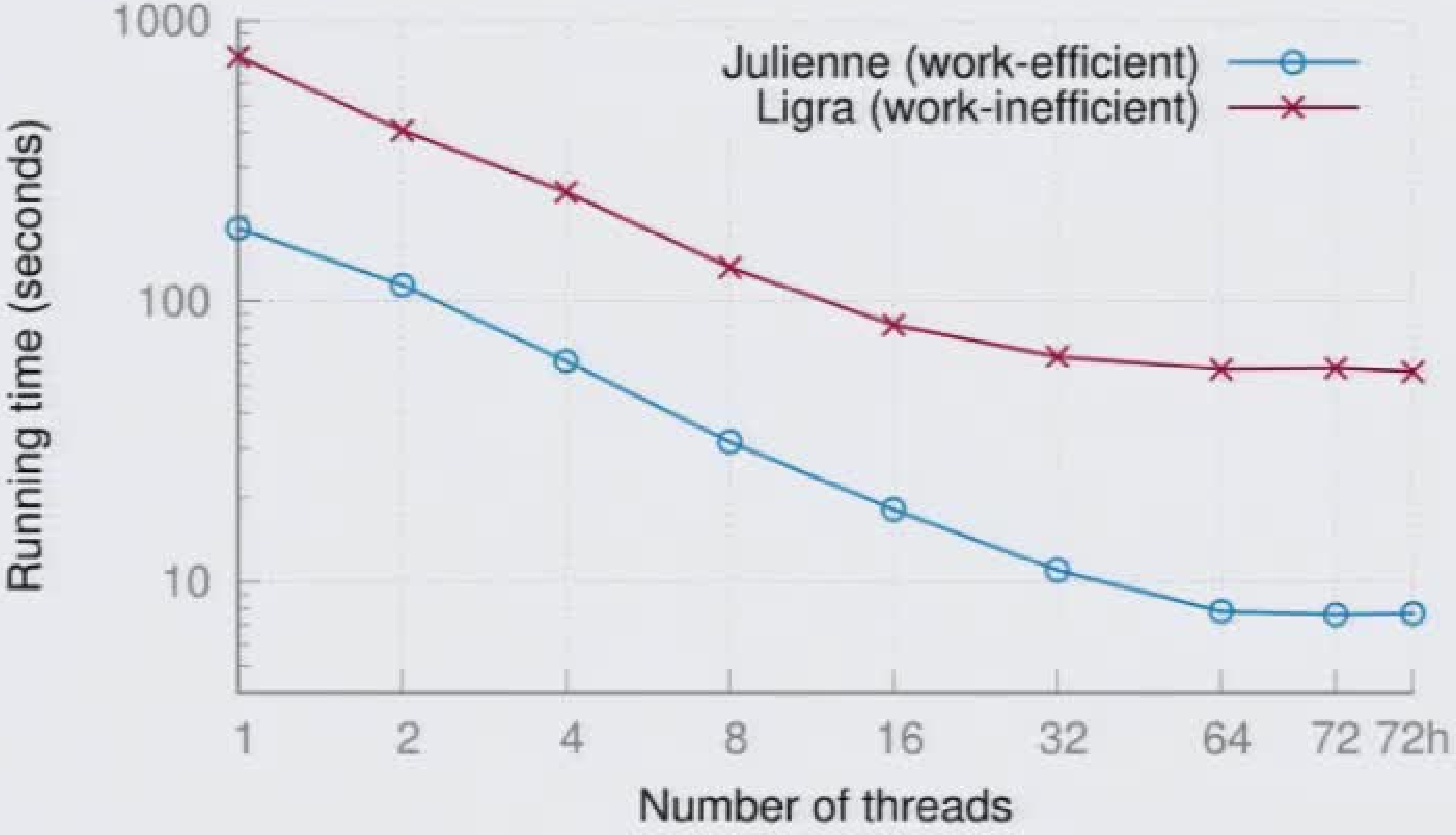
$O(r_{src} + |E|)$ expected work

$O(r_{src} \log |V|)$ depth w.h.p.

In general, our implementation is

- "Work-efficient" w.r.t. to the original delta-stepping algorithm
- Not work-efficient w.r.t. Dijkstra's algorithm with Fibonacci heaps

Experiments: k-core



Friendster

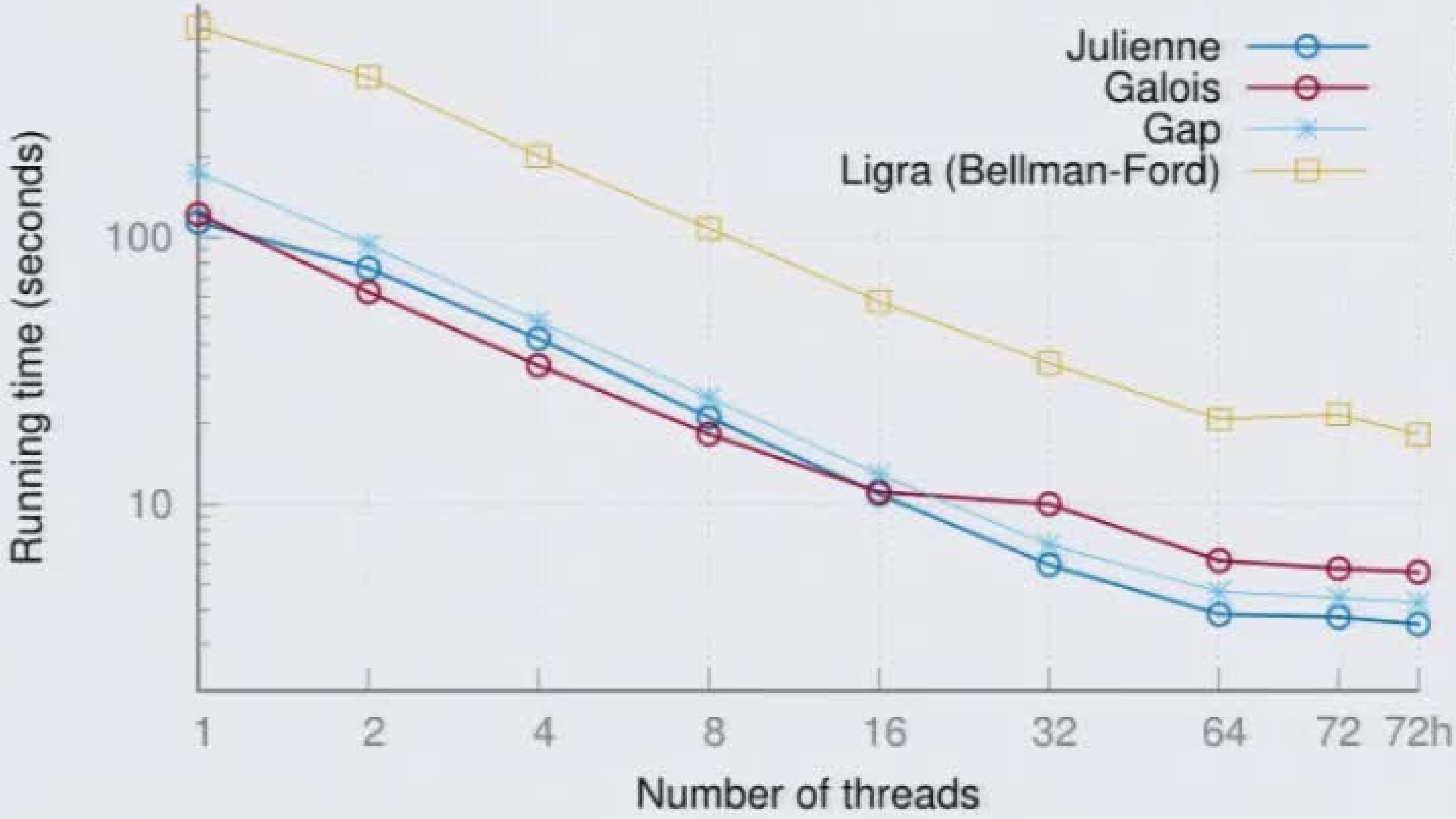
|V| = 121M

|E| = 3.6B

Across all inputs:

- Between 4-41x speedup over sequential peeling
- Speedups are smaller on small graphs with large ρ
- 2-9x faster than work-inefficient implementation

Experiments: Delta-Stepping



Friendster

|V| = 121M

|E| = 3.6B

Across all inputs:

- 18-32x self-relative speedup, 17-30x speedup over DIMACS solver
- 1.1-1.7x faster than best existing implementation of Delta-Stepping
- 1.8-5.2x faster than (work-inefficient) Bellman-Ford

Experiments: Hyperlink Graphs

Hyperlink graphs extracted from Common Crawl Corpus

Graph	|V|	|E|	|E|(symmetrized)
HL2014	1.7B	64B	124B
HL2012	3.5B	128B	225B

- Previous analyses use supercomputers [1] or external memory [2]
- Able to process in main-memory of 1TB machine by compressing

[1] Slota et al., 2015, Supercomputing for Web Graph Analytics

[2] Zheng et al., 2015, FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs

Experiments: Hyperlink Graphs

Graph	k-core	wBFS	Set Cover
HL2014	97.2	9.02	45.1
HL2012	206	—	104

Running time in seconds on 72 cores with hyperthreading

- 23-43x speedup across applications
- Compression is crucial
    - Julienne/Ligra codes run without any modifications
    - Can't run other codes on these graphs without significant effort