Practical Parallel Hypergraph Algorithms

Julian Shun

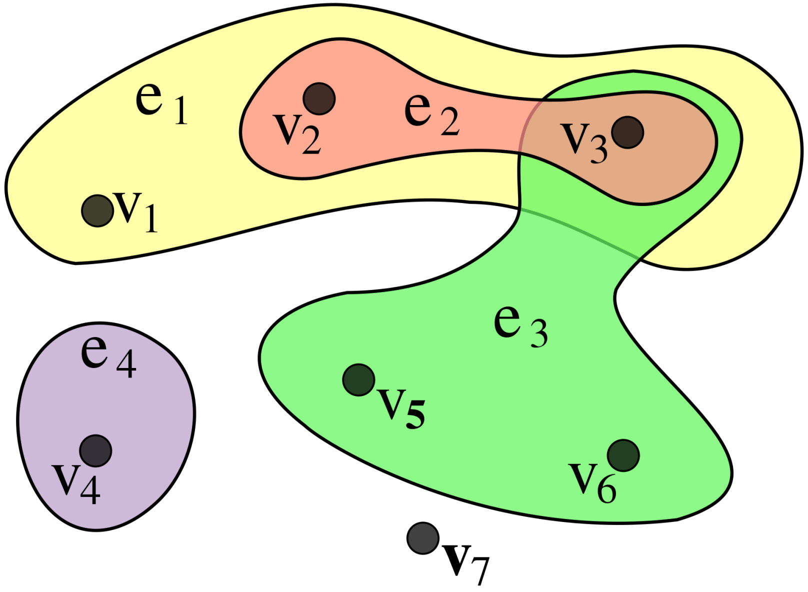
Graphs

- **Vertices** model objects
- **Edges** model relationships between objects
- Applications: social networks, biological networks, Web, scientific computing, etc.
- Lots of research on high-performance parallel graph algorithms, frameworks, and libraries

Some graph processing solutions

Pregel, Giraph, GPS, GraphLab, PowerGraph, PRISM, Pegasus, Knowledge Discovery Toolbox, CombBLAS, GraphChi, GraphX, Galois, X-Stream, Gunrock, GraphMat, Ringo, TurboGraph, TurboGraph++, FlashGraph, Grace, PathGraph, Polymer, GPSA, GoFFish, Blogel, LightGraph, MapGraph, PowerLyra,Graphine, PowerSwitch, Imitator, XDGP, Signal/Collect, PrefEdge, EmptyHeaded, Gemini, Wukong, Parallel BGL, KLA, Grappa, Chronos, Green-Marl, GraphHP, P++, LLAMA, Venus, Cyclops, Medusa, NScale, Neo4J, Trinity, GBase, RADAR, HyperGraphDB, Horton, GSPARQL, Titan, ZipG, Cagra, Milk, Ligra, Ligra+, Lux, Julienne, GraphPad, Mosaic, GraFBoost, Graphene, Mizan, Green-Marl, PGX, PGX.D, Wukong+S, Stinger, cuStinger, Distinger, Hornet, GraphIn, Tornado, Bagel, KickStarter, Naiad, Kineograph, GraphMap, Presto, Cube, Giraph++, HATS, Photon, TuX2, GRAPE, GraM, Congra, MTGL, GridGraph, NXgraph, Chaos, Mmap, Clip, Floe, GraphGrind, DualSim, ScaleMine, Arabesque, GraMi, SAHAD, TAO, Weaver, G-SQL, G-SPARQL, gStore, Horton+, S2RDF, Quegel, EAGRE, Shape, RDF-3X, CuSha, Garaph, Totem, GTS, Frog, GBTL-CUDA, Graphulo, Zorro, Coral, CellIQ, GraphTau, Wonderland, GraphP, SAGE, Laika, nvGRAPH, cuGraph, GraphIt, GraPu, GraphJet, ImmortalGraph, LA3, Kaskade, AsyncStripe, Cgraph, GraphD, GraphH, ASAP, RStream, Automine, GraphOne, Aspen, GBBS, Gluon, Gswitch, SEP-Graph, SIMD-X, PnP, GraphA, Phoenix, Pregelix, ShenTu, Nepal, GraphSSD, LCC-Graph, RealGraph, Sedge, GraphMP, Tigr, PartitionedVC, DiGraph, Abelian, faimGraph, Falcon, Puffin, GraphBolt, GPOP, Omega, Slim graph, Log(Graph), RADS, CECI, BENU, GraphM, LIGHT, Pragh, Helios, GraphRex, Graphflow, MAGiQ, GAPBS, Wukong+G, GraphFrames, G-CORE, gRouting, Groute, TripleBit, SQLGraph, Graphphi, TuFast, Kaskade, etc.

Hypergraphs



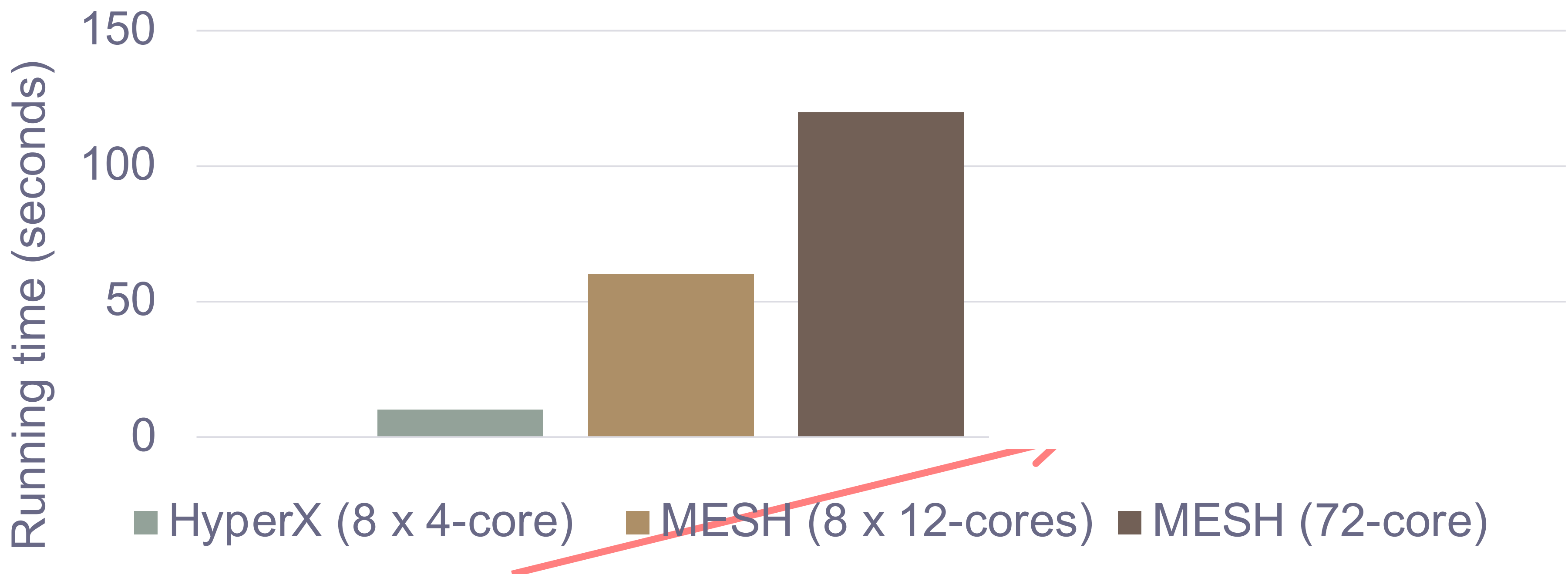
- **Hyperedges** can connect more than two vertices
- Captures more information than a graph representation
- Some applications:
  - Improved accuracy in image segmentation and spectral clustering [Zhou et al. 2006, Ducournau and Bretto 2014, Ding and Yilmaz 2008]
  - Better community detection [Bothorel and Bouklit 2011, Roy and Ravindran 2015]
  - Designing lookup tables, low-density parity-check codes [Jiang et al. 2017]
  - Satisfiability of Boolean formulas [Karp et al. 1988]
  - Protein network analysis [Ritz et al. 2017]

Parallel Hypergraph Processing

- Only two existing systems: HyperX [Jiang et al. 2019] and MESH [Heintz et al. 2019]
  - Both implemented on top of Apache Spark
- This paper:
  - A collection of theoretically-efficient parallel hypergraph algorithms for shared-memory multicores
  - Implemented using Hygra, a simple extension of the Ligra graph processing framework to support hypergraphs
    - Takes advantage of existing graph optimizations: direction-optimization, load-balancing, compression

Performance Comparison

1 iteration of PageRank on Orkut communities hypergraph
(2.3M vertices, 15.3M hyperedges, sum of hyperedge cardinalities = 107M)



- Performance difference due to higher communication costs of distributed-memory and overheads of Spark

Algorithms and Complexity Bounds

Work = # operations *Span = longest sequential dependence*

Algorithm	Work	Span
Betweenness centrality	O(H)	O(D log H)
Maximal independent set*	poly(H)	polylog(H)
K-core decomposition	O(H)	O(ρ log(H))
Hypertrees	O(H)	O(D log H)
Hyperpaths	O(H)	O(D log H)
Connected components	O(DH)	O(D log H)
PageRank (1 iteration)	O(H)	O(log H)
Single-source shortest paths	O(VH)	O(V log H)

H = size of hypergraph
D = diameter of hypergraph

V = number of vertices
ρ = peeling complexity

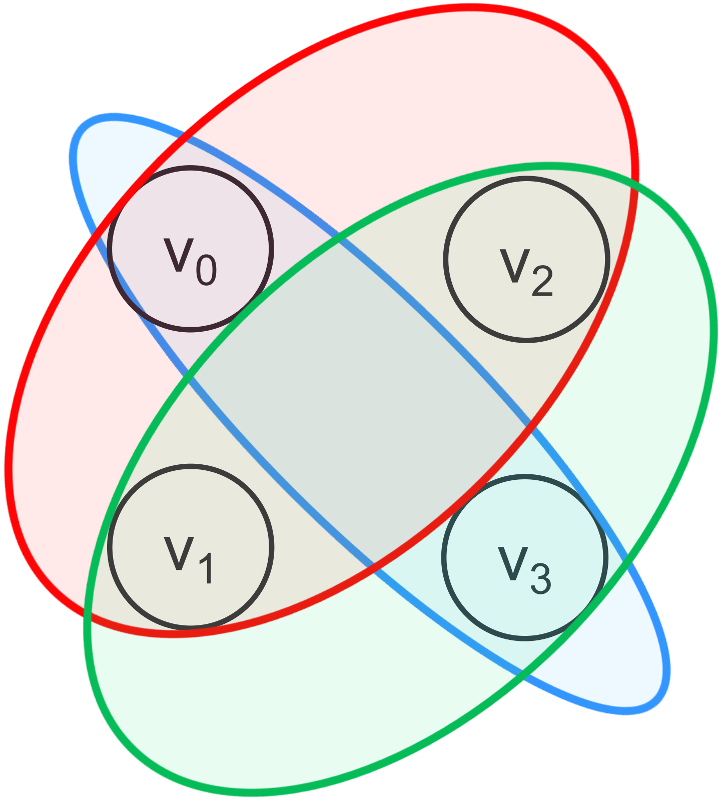
- Bounds at least as good as previous implementations (if any)

Remainder of the Talk

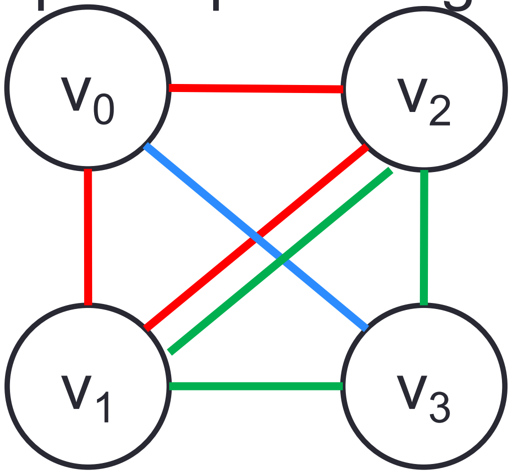
- Hypergraph representations
- Betweenness centrality algorithm
- K-core decomposition algorithm
- Experiments

Hypergraph Representations

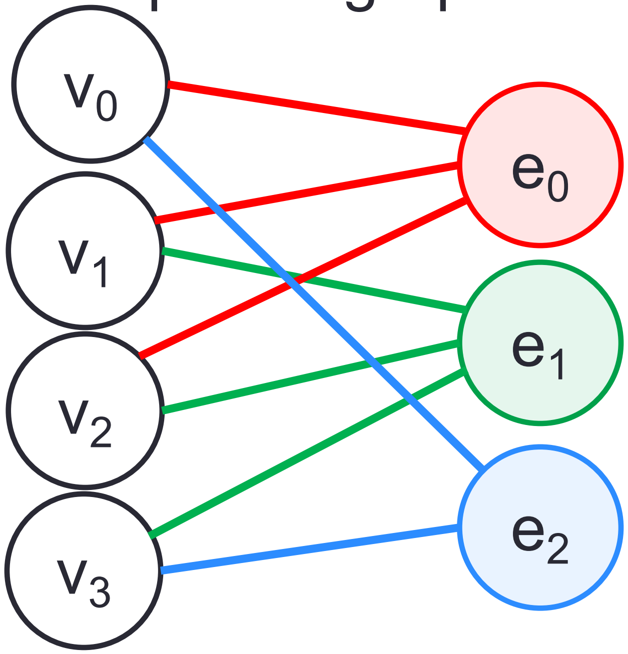
Original hypergraph



Clique-expanded graph



Bipartite graph



Hyperedge list

(v_0, v_1, v_2)
(v_1, v_2, v_3)
(v_0, v_3)

Betweenness Centrality

Betweenness Centrality

- Betweenness centrality of a vertex v is the fraction of shortest paths between all pairs of vertices that pass through v
- Brandes' algorithm works for graphs, does a two-phase BFS-like traversal from each vertex, taking linear work per vertex
- We present a parallel betweenness centrality algorithm for hypergraphs

Betweenness Centrality (per source s)

- Puzis et al. present a sequential algorithm for hypergraphs
- Forward and backward phase for each vertex (BFS-like traversals)
- Forward phase:
  - Compute $\sigma_{s,v}$, number of shortest paths between source vertex s and vertex v, for all vertices v in the graph
  - Puzis et al.'s algorithm takes $O(V + \Sigma_{e \in E} \text{cardinality}(e)^2)$ work overall, which is super-linear in size of hypergraph

Betweenness Centrality (per source s)

- Our algorithm stores intermediate values on hyperedges, so that the total work is O(V + $\Sigma_{e \in E}$ cardinality(e)) = O(H)

Vertex equation

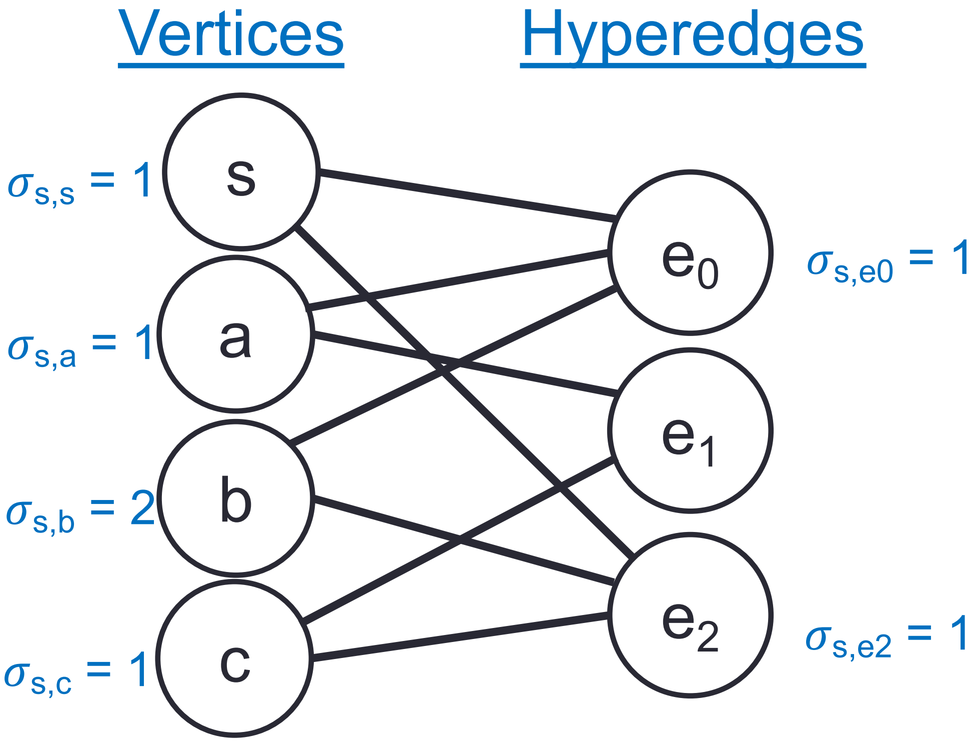
$$\sigma_{s,v} = \Sigma_{e \in P(v)} \sigma_{s,e}$$

P(v) are predecessor hyperedges of vertex v

Hyperedge equation

$$\sigma_{s,e} = \Sigma_{u \in P(e)} \sigma_{s,u}$$

P(e) are predecessor vertices of hyperedge e



Betweenness Centrality (per source s)

- Backward phase:
    - Compute dependency scores $\delta_{s\bullet}(v)$ for all v, which can be used to get betweenness centrality contribution from source s

$$\hat{\delta}_s(e) = \sum_{v\,:\,e \in P_V(v)} \frac{\delta_{s\bullet}(v)}{\sigma_{s,v}}$$

$$\delta_{s\bullet}(v) = 1 + \sum_{e\,:\,v \in P_E(e)} (\sigma_{s,v} \cdot \hat{\delta}_s(e))$$

- Vertex and hyperedge equations are different
- Total work is also O(V + $\Sigma_{e \in E}$ cardinality(e)) = O(H)

Aside: Hygra Interface

- Minor extension of Ligra to differentiate between processing vertices and hyperedges in bipartite representation

Hypergraph

VertexSubset

HyperedgeSubset

VertexMap

HyperedgeMap

VertexProp

HyperedgeProp

NextBucket

UpdateBuckets

- All operators take linear work and logarithmic span
- Can use direction-optimization and graph compression from Ligra

Betweenness Centrality (per source s)

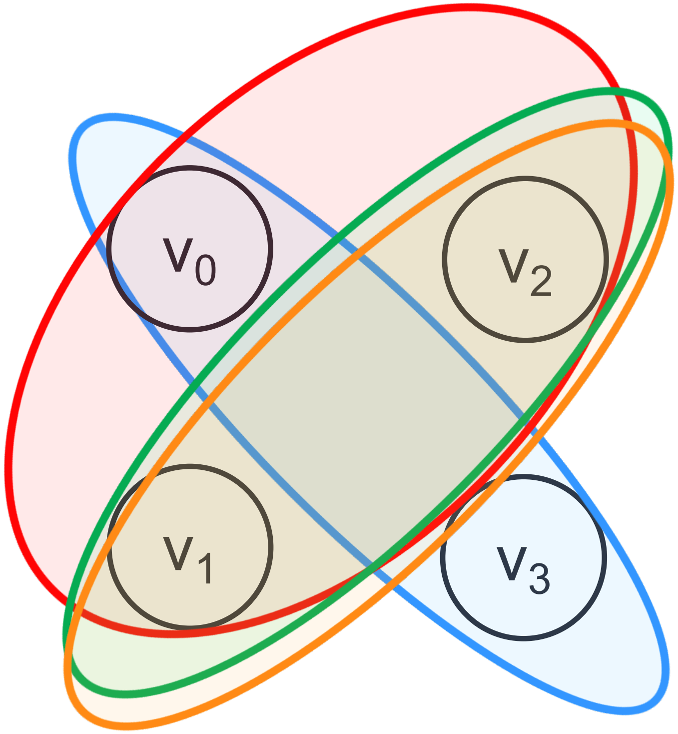
- Forward phase:
  - Each iteration keeps vertices and hyperedges on frontier as **VertexSet** and **HyperedgeSet**
  - Each iteration:
    - **VertexProp**: *Propagate path counts from* ***VertexSet*** *to incident hyperedges*
    - **HyperedgeMap**: *Mark hyperedges as visited*
    - **HyperedgeProp**: *Propagate path counts from* ***HyperedgeSet*** *to member vertices*
    - **VertexMap**: *Mark vertices as visited*
    - All functions are completely parallel
- Backward phase similar but with different functions
- Total work = O(H)   Total span = O(diam * log H)

K-core Decomposition

K-core Decomposition

- K-core is a maximal connected sub-hypergraph where every vertex has induced degree at least K
- Core number of a vertex is the maximum value of K for which it appears in that K-core
- Simple parallel algorithm:
  - K = 0
  - While hypergraph is not empty:
    - If any vertices have degree at most K:
      - Remove all vertices with degree at most K and their incident hyperedges, assigning them core value K
    - Else: K = K+1

K-core Decomposition



K=1

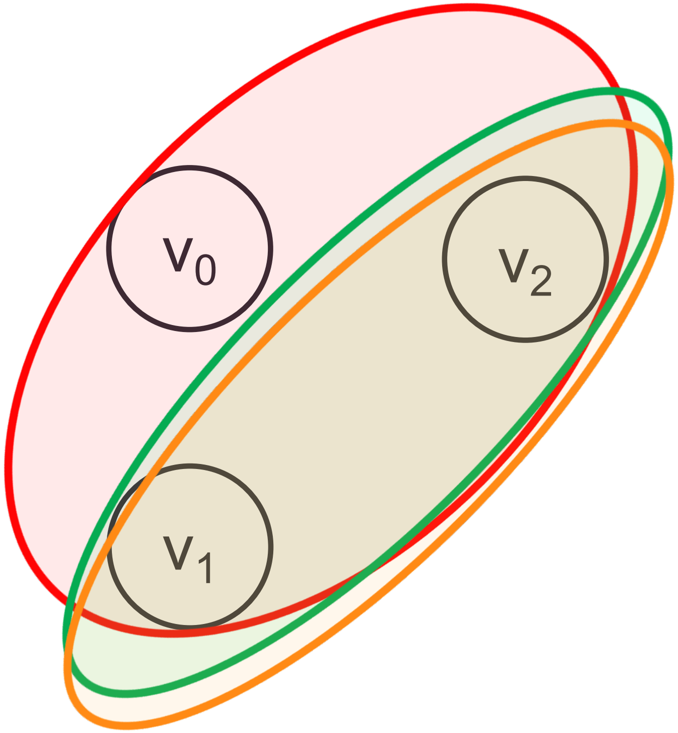
$\deg(v_0) = 2$

$\deg(v_1) = 3$

$\deg(v_2) = 3$

$\deg(v_3) = 1$

K-core Decomposition



K=1

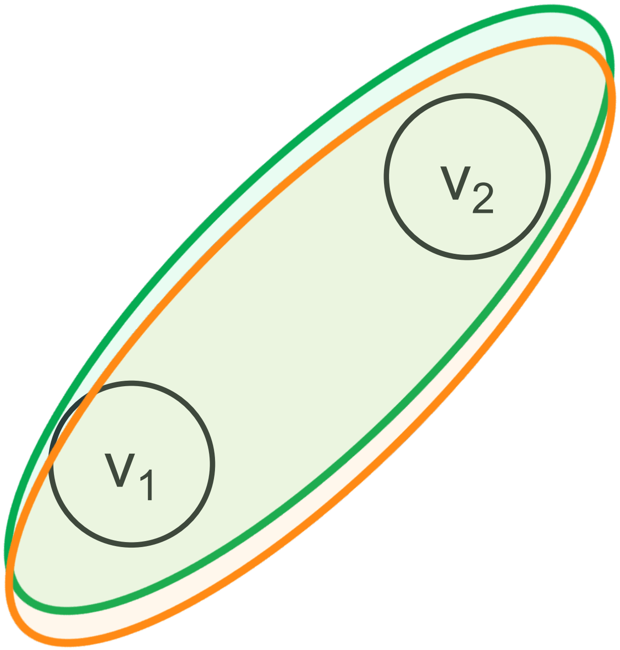
$\deg(v_0) = 1$

$\deg(v_1) = 3$

$\deg(v_2) = 3$

K-core Decomposition

K=2

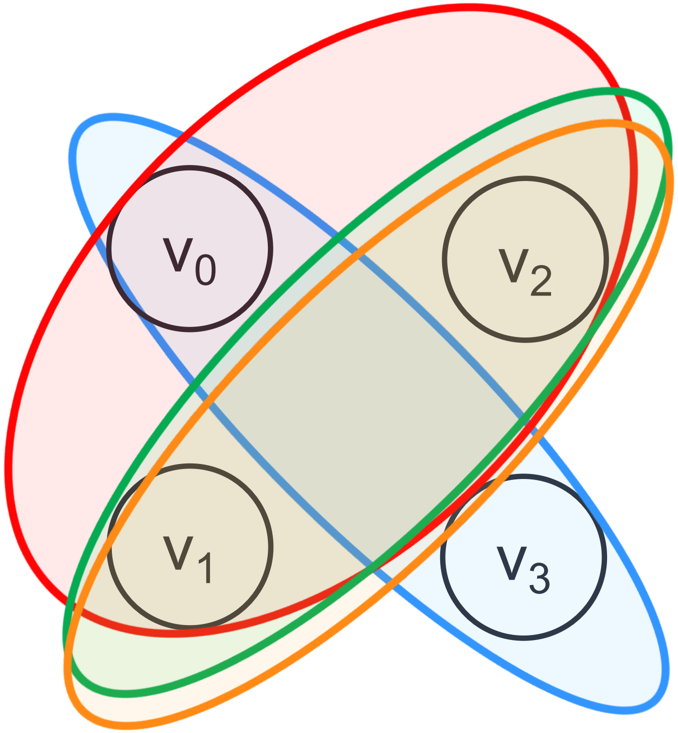


$\deg(v_1) = 2$

$\deg(v_2) = 2$

- No more vertices with degree at most 1, therefore increment K

K-core Decomposition



$\text{core}(v_0) = 1$

$\text{core}(v_1) = 2$

$\text{core}(v_2) = 2$

$\text{core}(v_3) = 1$

K-Core Decomposition

- Naïve implementation would take O(ρV+H) work, where ρ is the number of rounds needed (peeling complexity)

- Use **buckets** to group vertices based on their current degree [Dhulipala et al. 2017]
- Initialize bucketing structure with **MakeBuckets**
- While hypergraph is not empty:
  - **NextBucket**: Extract next smallest non-empty bucket
  - **VertexProp**: Remove vertices in extracted bucket and their incident hyperedges
  - **HyperedgeProp**: Decrement degrees of vertices in deleted hyperedges

K-Core Decomposition

- Each vertex extracted and deleted once
- Each hyperedge deleted once, and decrements degrees of all incident vertices when deleted

- Total work is $O(V + \Sigma_{e \in E} \text{cardinality}(e)) = O(H)$

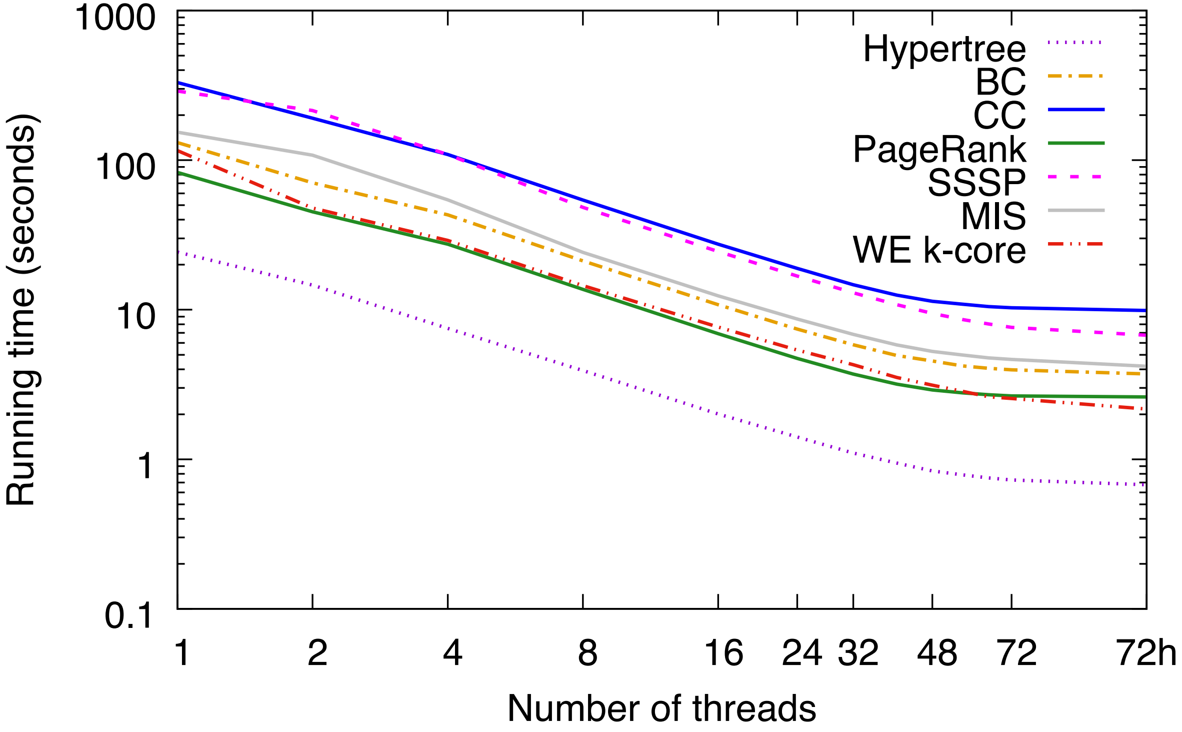
- Bucketing operations take O(log H) span, so total span is $O(\rho \log H)$

Experiments

Parallel Scalability

- Framework and algorithms implemented using Cilk Plus
- 72-core machine with hyper-threading

Random hypergraph



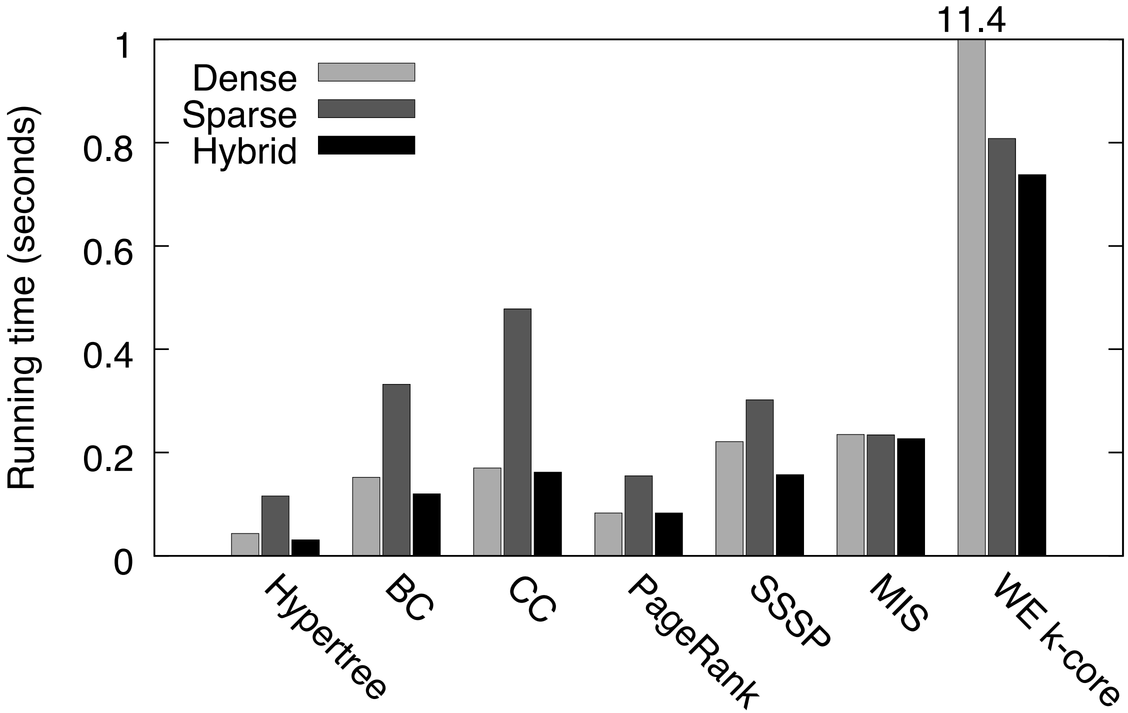
- Speedups ranging from 8.5x to 76.5x
- Average speedup of 38.7x
- Lower speedups on K-core due to many rounds

Direction Optimization

- **Dense**: Use a pull-based traversal applied to all vertices or hyperedges
- **Sparse**: Use a push-based traversal applied to just active vertices or hyperedges
- **Hybrid**: Use **Sparse** for small active sets and **Dense** for large active sets

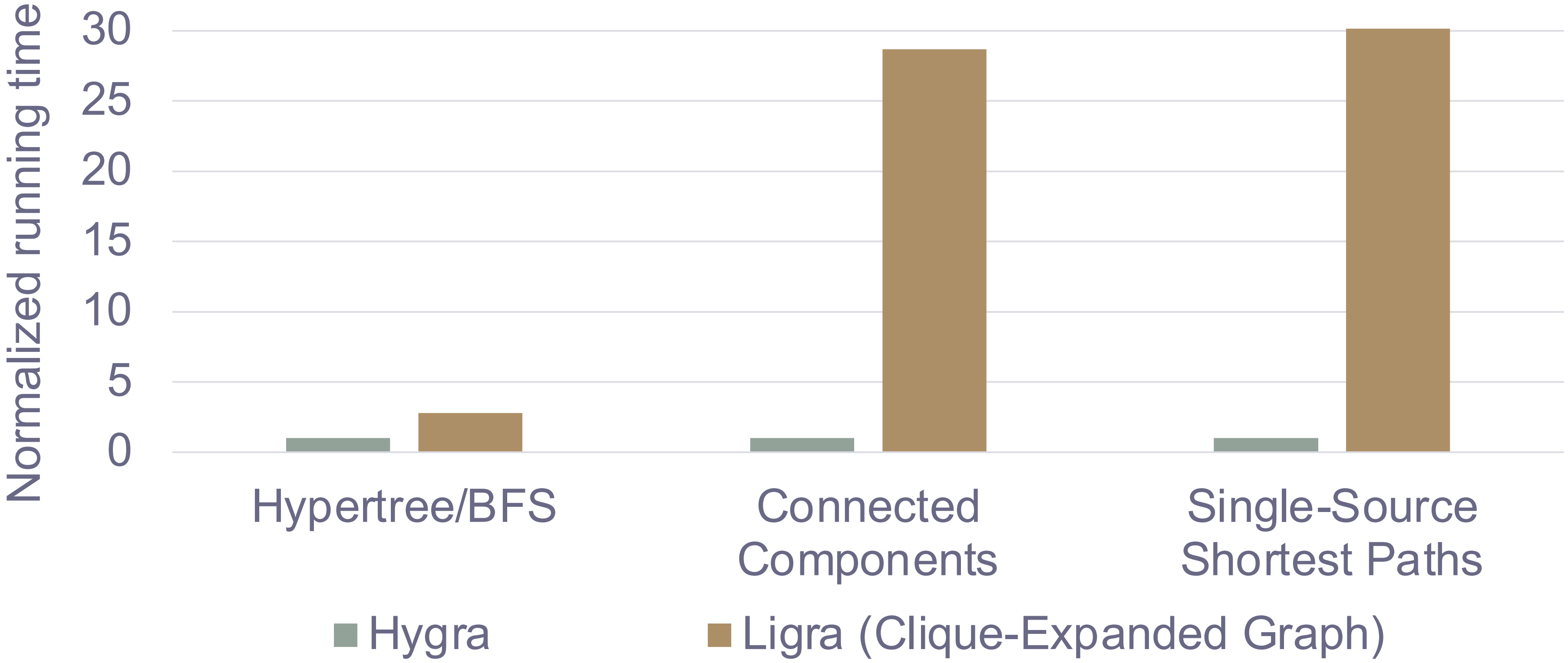
Orkut communities hypergraph

Threshold for switching: 1/20 of hypergraph size



Comparison with Clique-Expanded Graph

- Friendster hypergraph with 7.9M vertices, 1.6M hyperedges, and sum of hyperedge cardinalities was 23.5M
- Clique-expanded graph has 5.5B edges (235x larger)



- Hygra is 2.8-30.6x faster than using clique-expanded graph in Ligra

Conclusions

- New theoretically-efficient parallel hypergraph algorithms implemented using Hygra
- Lots of interesting topics for further research
  - Locality optimizations (e.g., reordering and cache/NUMA segmentation for bipartite graphs)
  - Implement and optimize for GPUs and other architectures
- Code and datasets are publicly-available at https://github.com/jshun/ppopp20-ae