Parallel Batch-Dynamic Subgraph Maintenance

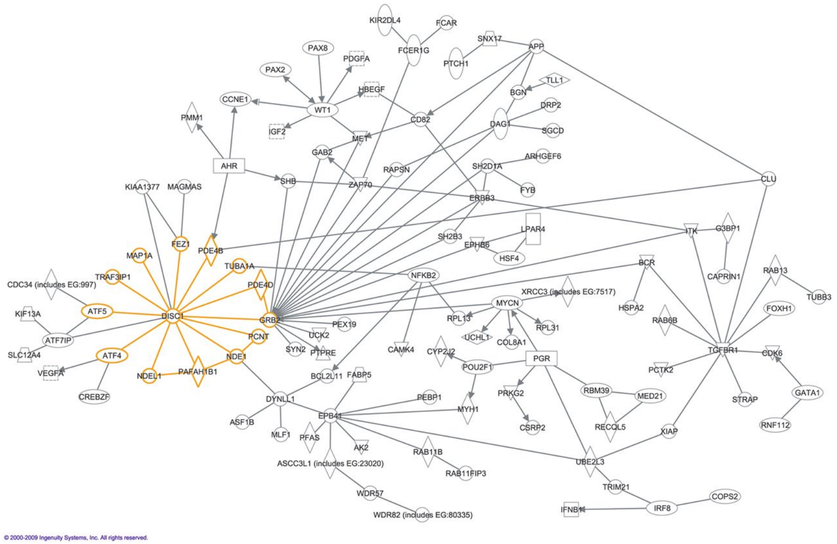
By Alex Fan and Alvin Lu
Mentored by Jessica Shi and Julian Shun

Outline

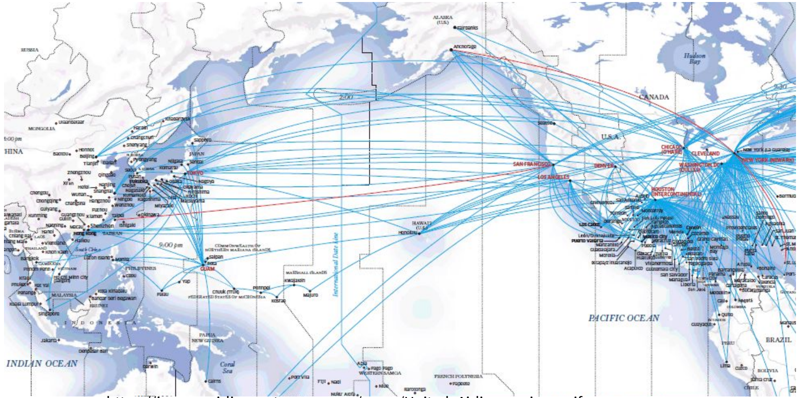
- Overview of the problem
- 3-vertex subgraph counting
    - Parallel algorithm
    - Implementation
- Evaluation
- Conclusion

Graph processing

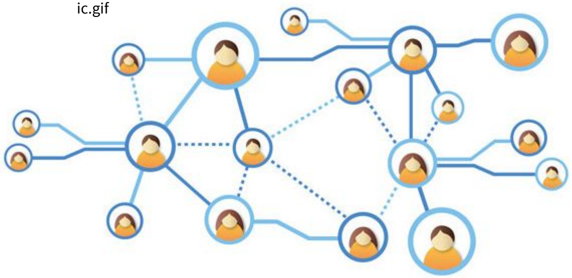
- Graphs represent a wide variety of complex networks, finding patterns within is very important



https://upload.wikimedia.org/wikipedia/commons/thumb/7/72/Network_of_how_100_of_the_528_genes_identified_with_significant_differential_expression_relate_to_DISC1_and_its_core_interactors.png/400px-Network_of_how_100_of_the_528_genes_identified_with_significant_differential_expression_relate_to_DISC1_and_its_core_interactors.png



https://images.airlineroutemaps.com/maps/United_Airlines_asia_pacific.gif



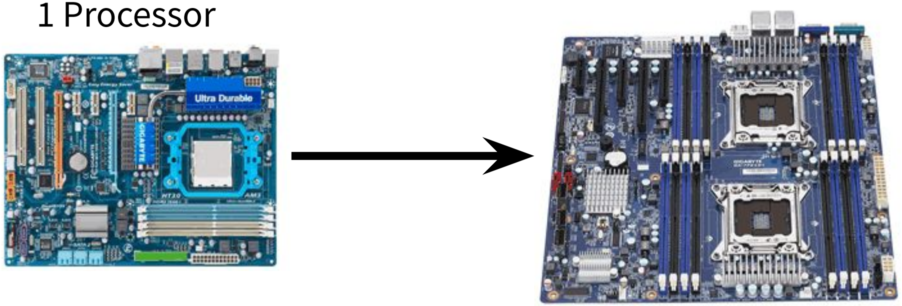
https://blog.soton.ac.uk/skillted/files/2015/04/social-network-grid.jpg

Parallelism

- Widely used: in phones, in large data centers, GPUs are parallelized
- All publicly available graphs fit in shared memory
- Process large datasets efficiently

1 Processor

2 Processors

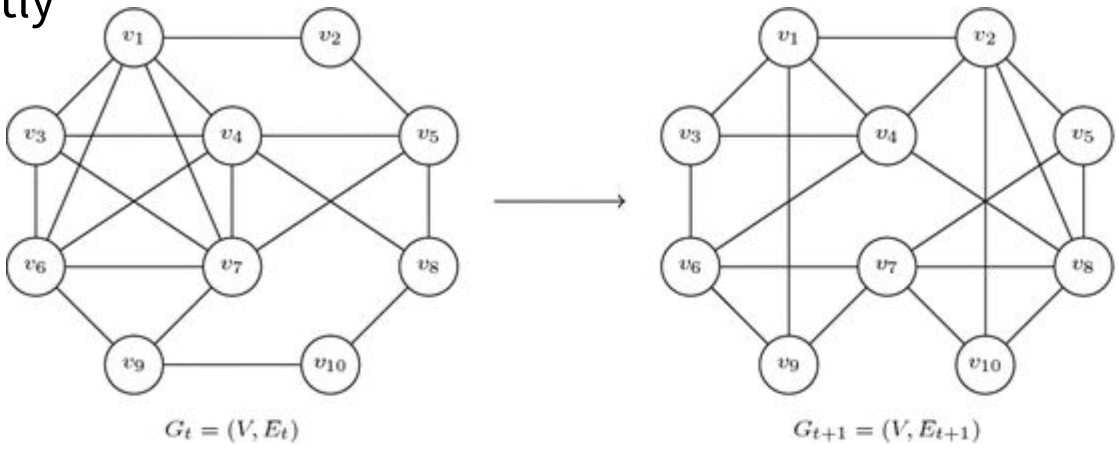


https://4.bp.blogspot.com/-DNSsmoZxJqI/VR0UXMOxy-I/AAAAAAAAAHs/20gKNUXfdgU/s1600/single%2Bvs%2Bmultiprocessor%2Bsys tems.gif

https://media.wired.com/photos/5b19a3fd985bbd041c32d0c3/125:94/w_2130,h_1602,c_limit/Summit-supercomputer---side-view-(w ide-shot)-TAFA.jpg

Dynamic Model

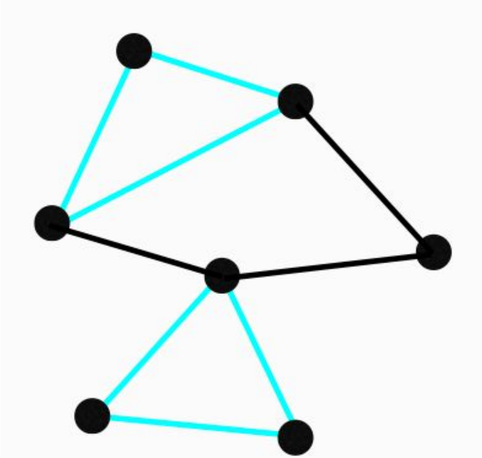
- Model which considers added and removed edges, real world graphs are often changing
- Perform real time updates, and update computation under model efficiently



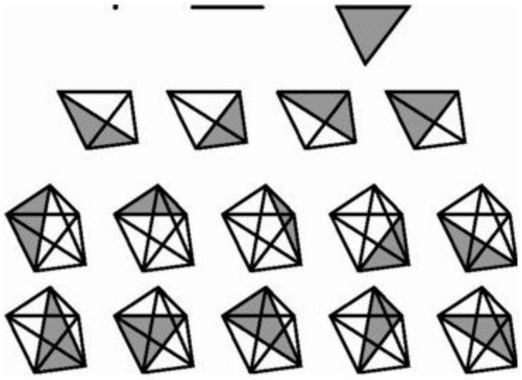
https://media.springernature.com/lw685/springer-static/image/art%3A10.1007%2Fs10732-017-9327-z/MediaObjects/10732_2017_9327_Fig1_HTML.gif

Dynamic Subgraph Counting

- **Problem:** Maintain subgraph counting in a batch parallel and dynamic setting
  - Given a graph G and a batch of updates, find the new number of specific 3-vertex subgraphs in parallel (e.g. triangles)



https://www.andrew.cmu.edu/user/dwise/15418/icon.png



https://lh3.googleusercontent.com/proxy/07GSoPdC6VF8YSqe8WxgM_5zDcTEqkN08c_CkCl9Yi__VonUIPvyYFyDSg9NHU4zQeFmE9plw-WMsy4TITpT6bucE1H5grb-0buoryjnW7s81Xz-v4UoHWSt7V6yNLdl

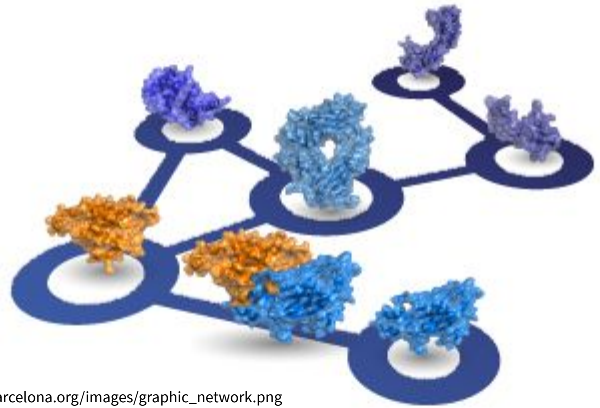
Other Works

Lots of works on counting but none are dynamic and parallel

- Serial, static 5-vertex counting: A. Pinar, C. Seshadhri, V. Vishal
- Parallel, static 4-vertex counting: N. Ahmed, J. Neville, R. Rossi
- Serial, dynamic 3-vertex counting: D. Eppstein, E. Spiro

Applications

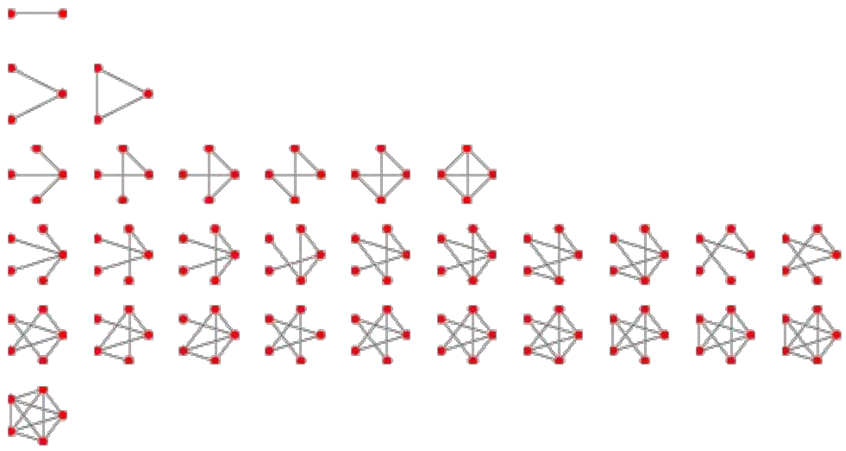
- Given an interactome, find patterns of interactions between different molecules in a cell
- Identify groups in social and communication networks to help people connect more easily (e.g. Facebook friend suggestions)
- Find subgraphs in air traffic to coordinate flights



https://interactome3d.irbbarcelona.org/images/graphic_network.png

Goal

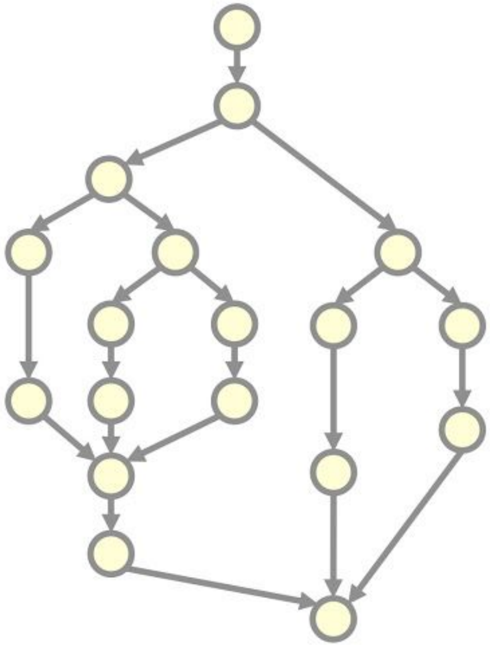
- New **parallel** algorithm for **dynamic** subgraph counting
- **Strong theoretical bounds** for runtime and memory
- Complete evaluation for counting **triangles**
- Foundation to extend to **four-vertex subgraphs** as well.



Important paradigms

- **Work and Span Model**
  - **Work** = Total operations = number of nodes in DAG
  - **Span** = The maximum number of nodes on a dependency chain = Longest path
  - **Work-Efficient** = The total work is the same as the best sequential version for the specific problem
  - **Running time** ≤ work/P + O(span) where P is the number of processors

Parallel Computing DAG

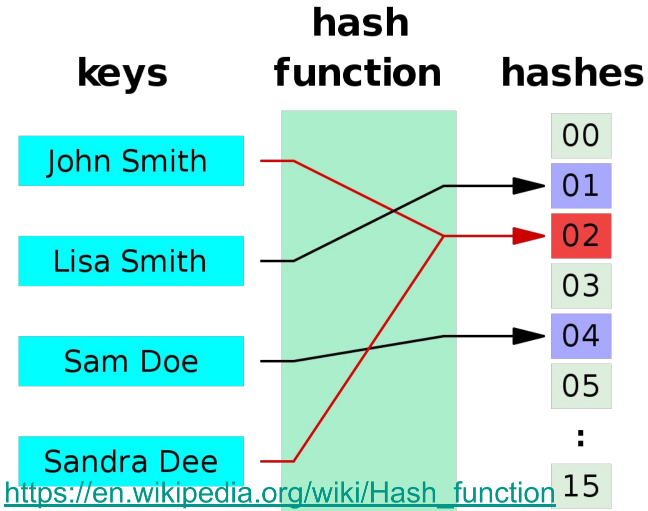


Parallel primitives

- **Parallel Filter**: Given an array of elements, filter out certain elements and concatenate the gaps afterward.
  - **Bounds**: O(N) work and O(logN) span
- **Parallel Reduce**: Given an array of elements, reduce them to a single "sum" under a commutative and associative operator.
  - **Bounds**: O(N) work and O(logN) span
- **Parallel Prefix Sum:** Given a list of numbers, generate a list of prefix sums. Formally, prefix[i] = Σ(j = 1 to i) arr[i]
  - **Bounds**: O(N) work and O(logN) span

Parallel primitives

- **Parallel Integer Sort**: Sort a given list of integers.
  - **Bounds**: O(N) work and O(logN) span
- **Parallel Hashing**: Hashes a list of elements to achieve fast random access.
  - **Bounds**: O(N) work and O(logN) span



https://en.wikipedia.org/wiki/Hash_function

Dynamic subgraph counting algorithm

Serial version from D. Eppstein and E. Spiro. The h-Index of a Graph and its Application to Dynamic Subgraph Statistics. *J. Graph Algorithms & Applications*, 16(2): 543-567, 2012

HSet: Dynamic h-index

HSet Overview

- HSet keeps track of all vertices
- Maintains set H
    - h-index = h = $|H|$
    - Largest h such that there are at least h vertices with degree greater than or equal to h
- Serial[1] maintains H in O(1) time for a single modified edge
- HSet will help reduce computation in triangle counting

[1]Serial version from D. Eppstein and E. Spiro. The h-Index of a Graph and its Application to Dynamic Subgraph Statistics. *J. Graph Algorithms & Applications*, 16(2): 543-567, 2012

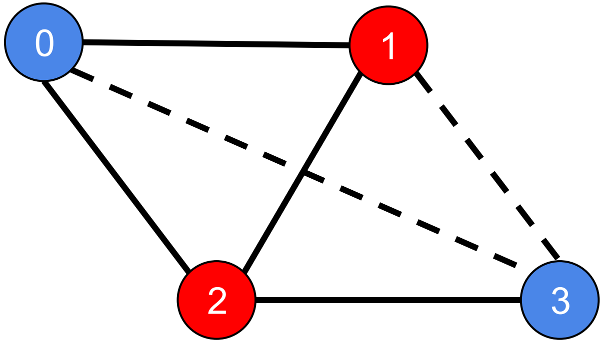
HSet - Outline

1. Remove endpoints of modified edge from HSet
2. Modify the edge
3. Re-add the endpoints back into HSet

HSet - Parallelizing

- Algorithm by Eppstein and Spiro inherently sequential
  - Multiple operations cause contention in HSet
- Our Parallelized version
  - Given a batch, h can change by at most |batch| = b
  - Prefix sum gets the number of vertices gained/lost, predicts new h
  - Expected work of O(b) and span O(log b) w.h.p.
    - Limited by taking the prefix sum and sorting the batch

Parallel HSet - Initial Graph



$b = |batch|$

$h = 2$

x = vertex $\notin$H

x = vertex $\in$H

x = untracked vertices (not in HSet)

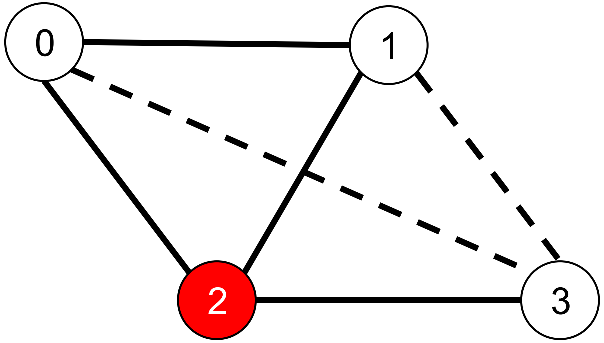
= Existing edge

= Edge not yet added

All vertices by degree

Degree	0	1	2	3
Vertices	Ø	{3}	{0, 1}	{2}

Parallel HSet - Removing Vertices



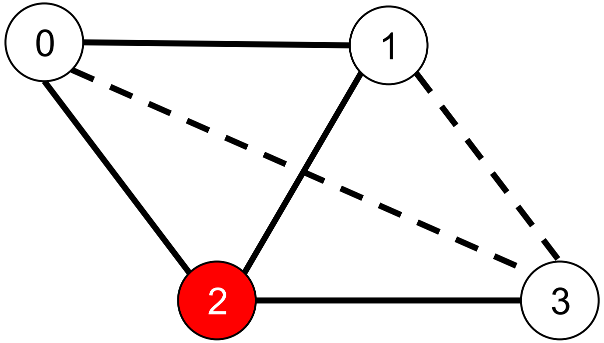
h = 2

Batch = all endpoints of all edges = {0, 1, 3}

- Remove in parallel

Degree	0	1	2	3
Vertices	Ø	{3} → Ø	{0, 1} → Ø	{2}

Parallel HSet - Removing Vertices



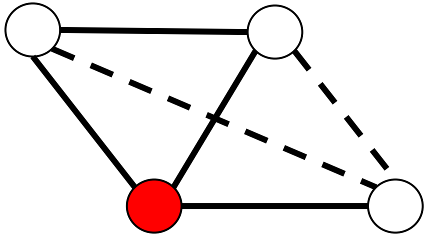
h = 2

# of Vertices w/ deg ≥ h

aboveH = 1 vertex

Degree	0	1	2	3
Vertices	Ø	Ø	Ø	{2}

Parallel HSet - Removing Vertices



Degree	0	1	2	3
Vertices	Ø	Ø	Ø	{2}

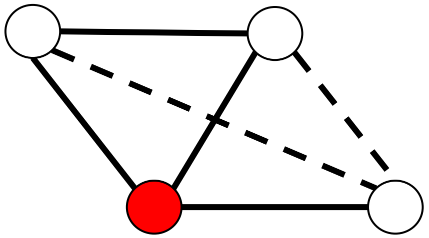
Prefix Sum Table: from h down to max(0, h - b)

Degree	h = 2	1	max(0, h - b) = 0
Size	---	0	0
Prefix Sum	0	0	0

Ignore size of table[h]

- Already included in aboveH

Parallel HSet - Removing Vertices



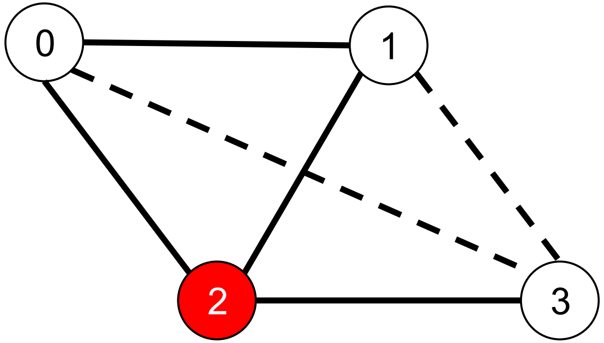
aboveH = 1

Largest degree such that
aboveH + prefixSum[deg] ≥ deg

Prefix Sum Table

Degree	2	1	0
Prefix Sum (vertices gained)	0	0	0
# of vertices above that degree	1 + 0 < 2	1 + 0 ≥ 1	1 + 0 ≥ 0

Parallel HSet - Removing Vertices

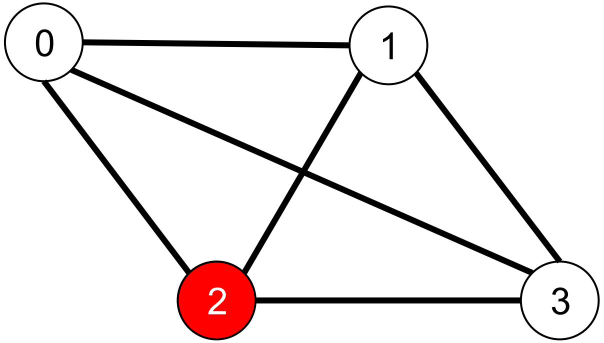


Set new h to be the largest degree where aboveH + prefixSum[deg] ≥ deg

h = 1

Degree	0	1	2	3
Vertices	Ø	Ø	Ø	{2}

Parallel HSet - Add (or Delete) Edges

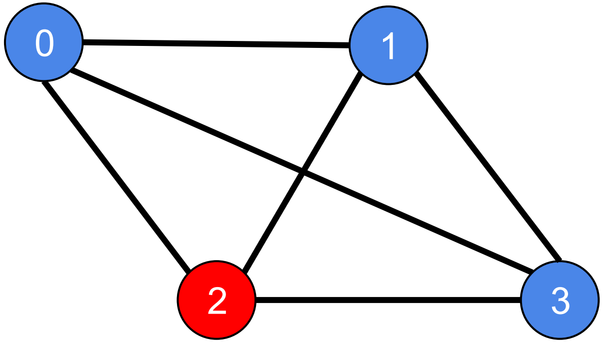


h = 1

Add or delete edges (which modifies the degrees)

Degree	0	1	2	3
Vertices	Ø	Ø	Ø	{2}

Parallel HSet - Re-adding Vertices



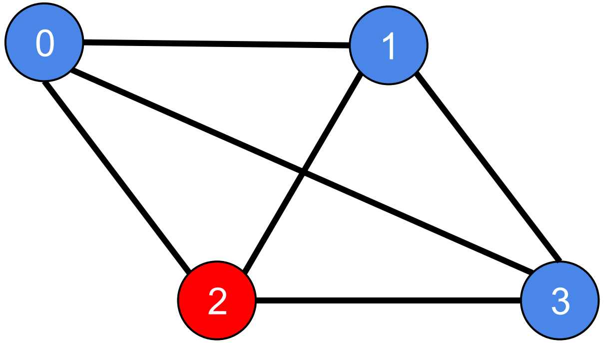
h = 1

Batch = all endpoints of all edges = {0, 1, 3}

- Add in parallel

Degree	0	1	2	3
Vertices	Ø	Ø	Ø	{2} → {0, 1, 2, 3}

Parallel HSet - Re-adding Vertices



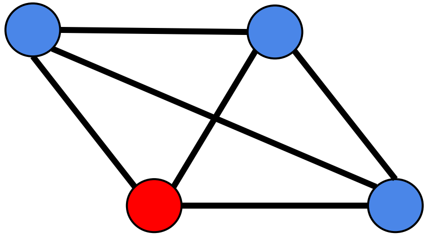
h = 1

# of Vertices w/ deg ≥ h

aboveH = 4 vertices

Degree	0	1	2	3
Vertices	Ø	Ø	Ø	{2} → {0, 1, 2, 3}

Parallel HSet - Re-adding Vertices

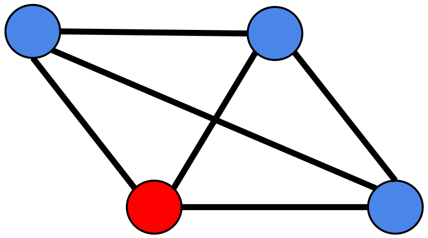


Degree	0	1	2	3
Vertices	Ø	Ø	Ø	{0, 1, 2, 3}

Prefix Sum Table: from h up to h + b

Degree	h = 1	2	3	h + b = 4
Size	0	0	4	0
Prefix Sum	0	0	4	4

Parallel HSet - Re-adding Vertices



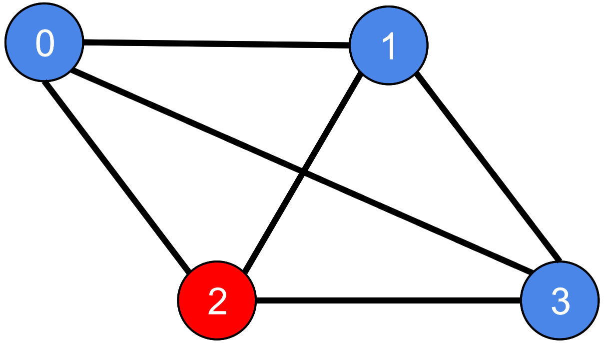
aboveH = 4

Smallest degree such that
aboveH - prefixSum[deg] < deg

Prefix Sum Table

Degree	1	2	3	4
Prefix Sum (vertices lost)	0	0	4	4
# of vertices above that degree	4 - 0 ≥ 1	4 - 0 ≥ 2	4 - 4 < 3	4 - 4 < 4

Parallel HSet - Re-adding Vertices

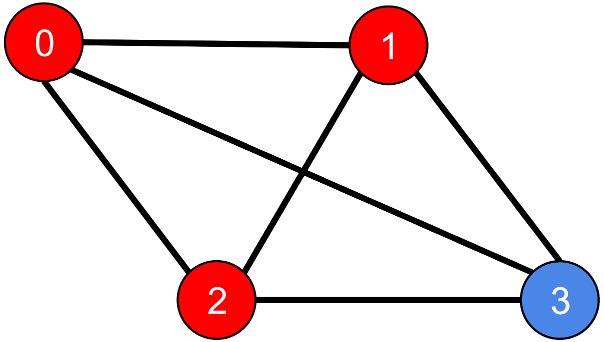


Set new h to be the smallest degree where aboveH - prefixSum[deg] < deg

h = 3

Degree	0	1	2	3
Vertices	Ø	Ø	Ø	{0, 1, 2, 3}

Parallel HSet - Result



h = 3

Can determine if a vertex is in H by comparing it’s degree to the h-index

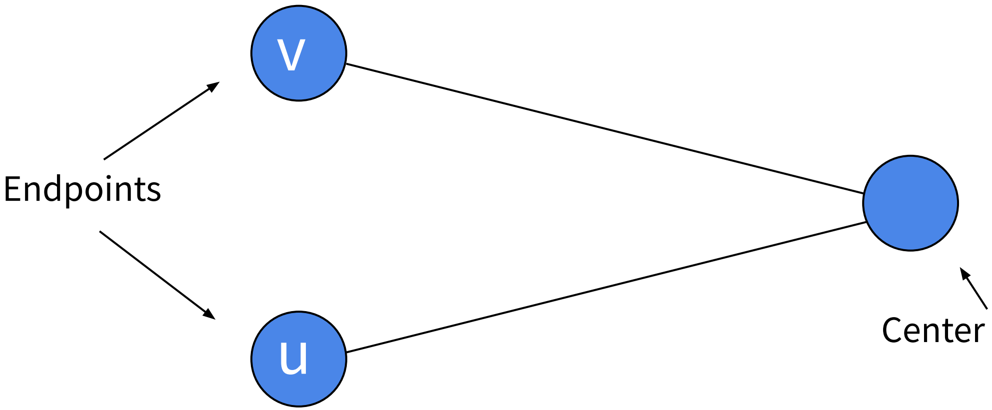
- Also accounts for vertices with degree equal to h-index but are not in H

Degree	0	1	2	3
Vertices	Ø	Ø	Ø	{0, 1, 2, 3}

Triangle Counting

Serial version from D. Eppstein and E. Spiro. The h-Index of a Graph and its Application to Dynamic Subgraph Statistics. *J. Graph Algorithms & Applications*, 16(2): 543-567, 2012

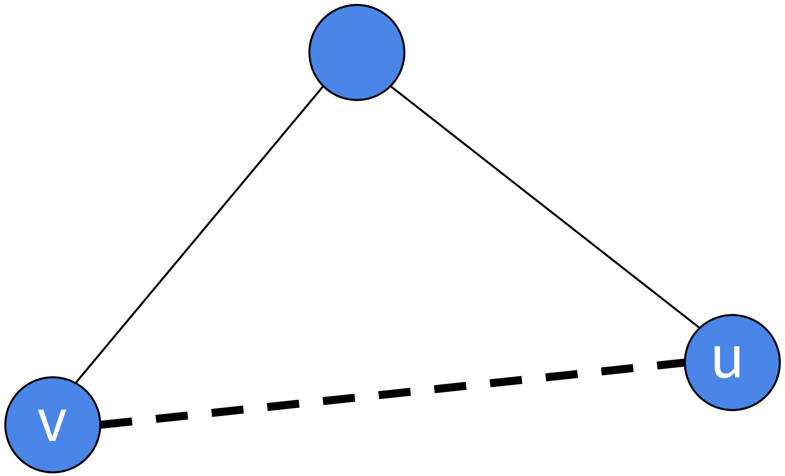
How do we find triangles?



Wedges! aka **2-Paths**

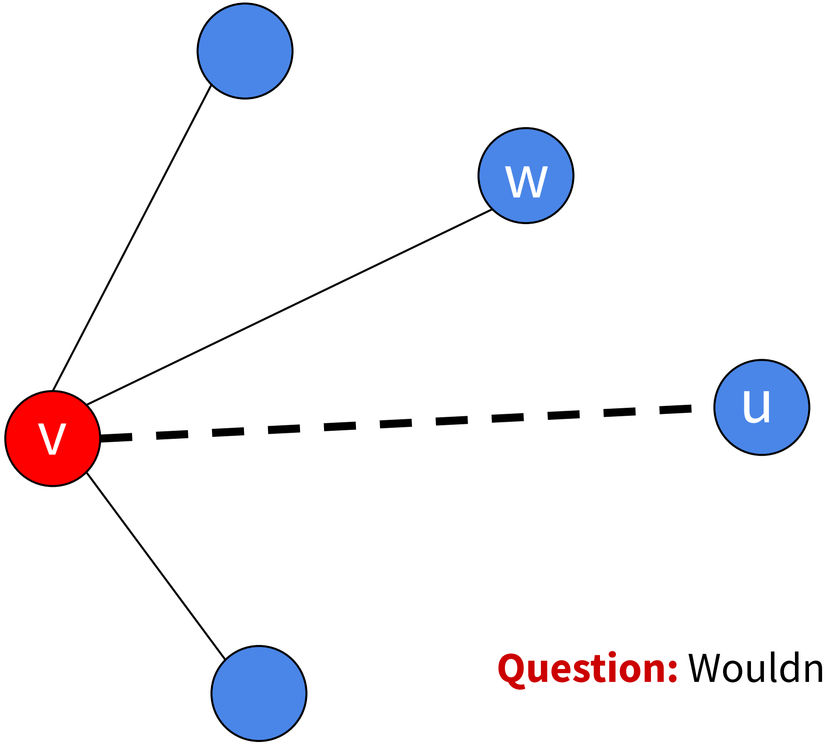
- Triangles and 4-vertex subgraphs are made of wedges
- **W(u,v)** = # of wedges endpoints **u** and **v**

Finding triangles from wedges



- For each added edge **W(u,v)**, triangles become complete

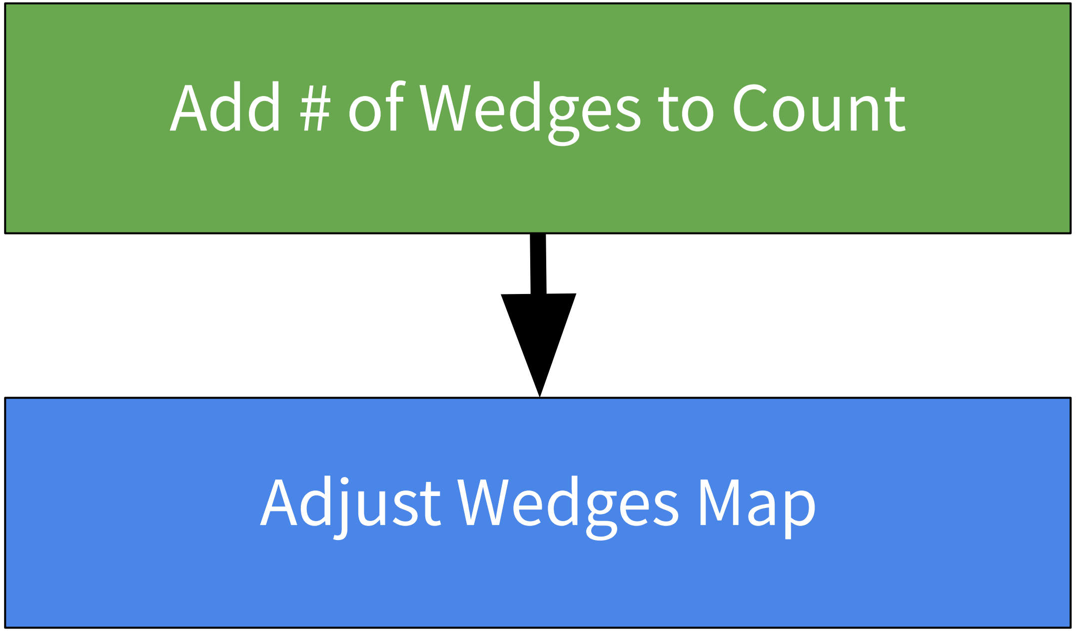
Maintaining wedges



- Brute force for all the neighbors
- For edge **(u,v)**, endpoint **v**, and each of its neighbors **w**, we add **1** to **W(w,u)**

Question: Wouldn't this be too slow?

Summary of Current Algorithm



TOO SLOW!
O(N) per edge

Note that edge deletion is symmetric

Optimization using HSet

We will use the previously introduced **HSet**

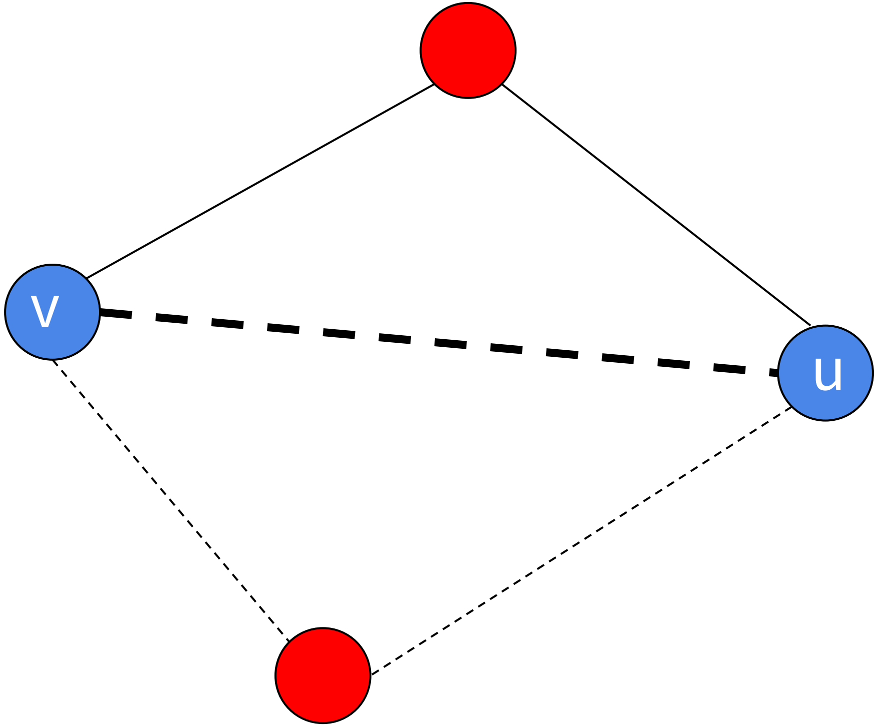
- For **W(u,v)**, keep track of wedges **with centers outside of the HSet**

x = vertex $\notin$H

x = vertex $\in$H

x = untracked vertices (not in HSet)

Optimization using HSet

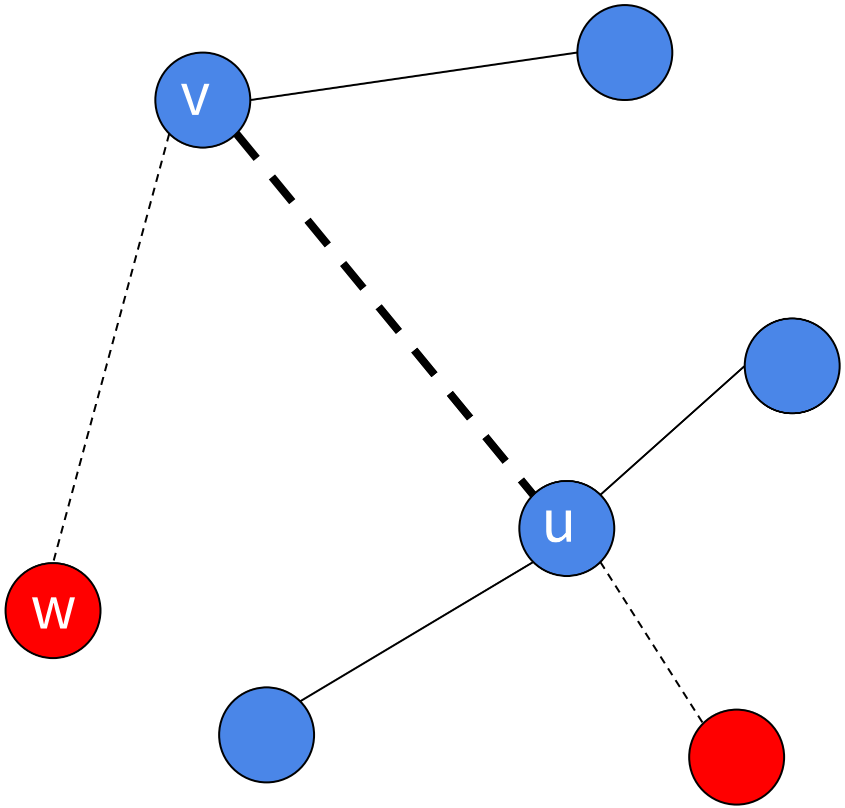


Problem: Missing triangles with centers in **HSet.**

Solution: Iterate through **HSet** and check if it can form another triangle

O(h) work per edge since we only iterated through the **HSet**

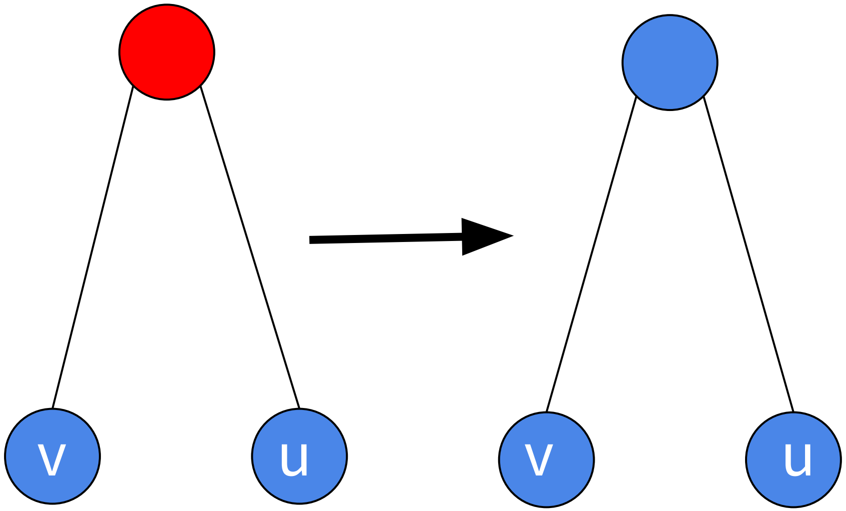
Optimization using HSet



1. Iterate through each of the endpoints that are **not in the HSet**
2. For edge **(u,v)**, and each of its neighbors **w**, we can add **1** to **W(w,u)**

O(h) work per edge since there are at most **h** neighbors

Optimization using HSet



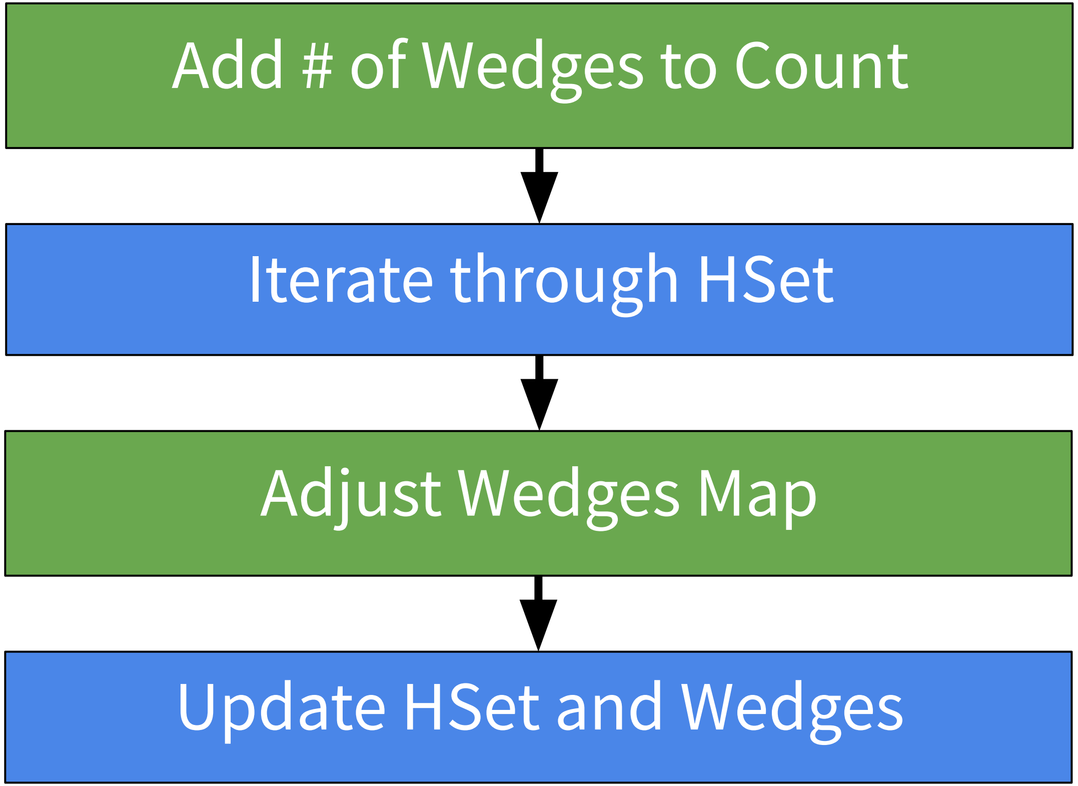
Problem: Nodes can cease to be in **HSet**.

Solution: For each pair of neighbors **u** and **v**, we add **1** to **W(u,v)**

HSet changes O(1/h) per edge amortized, so the actual complexity is still **O(h)**

Note that the converse when a node gets into **HSet** works the exact same way

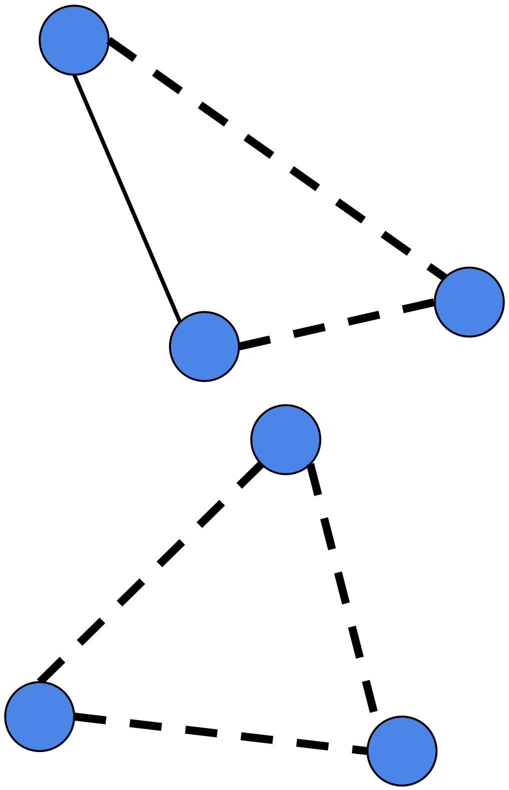
Summary of Optimized Algorithm



Time Complexity: **O(h)**

Deletion works symmetrically as well.

Problem arises when parallelized

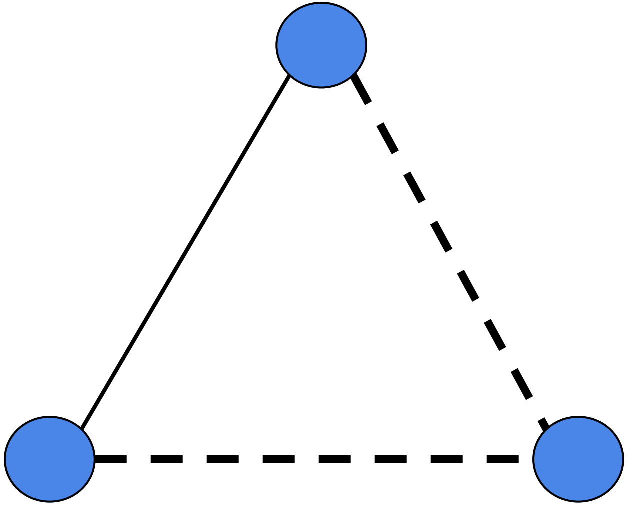


Problem: Won’t be able to update the wedges in time, therefore triangles on the left will not be counted

Solution: For each endpoint outside of **HSet**, iterate through all of its neighbors, and check if they form a triangle

Since there are at most **2h** neighbors, the work is at most **O(h)**

Another problem: Duplicate Triangles



Problem: Triangles like the on the left would be counted twice

Solution: We categorized all triangles into **11** types, each with their frequency. Instead of adding **1**, we add **1/frequency**

Evaluation

Implementation Detail: Storing HSet

- **Threshold**: Stores nodes with degree greater than a threshold in a hash table and the rest in a dynamic array
  - **Advantage**: Saves memory for sparse high-degree vertices
  - **Disadvantage**: Lots of overhead, difficult to adjust threshold
- **Dynamic Array**: Store nodes bucketed by their degree in a dynamic array
  - **Advantage**: Very little overhead, easy to manipulate.
  - **Disadvantage**: Takes memory proportional to the largest degree

Implementation Detail: Space Optimization

Storing Wedges Map **W(u,v)**

- **Hash Table**: We hash **W(u,v)** by the pair (u,v).
  - **Advantage**: Strong theoretical bounds $O(\min(N^2, \mathbf{Nh^2}))$ space
  - **Disadvantage**: Overhead in access/insertion due to cache misses
- **2D Ragged Array**: A 2D ragged array with the two side points as the indices
  - **Advantage**: Very little overhead
  - **Disadvantage**: It takes up $\mathbf{O(N^2)}$ space

Environment

- Google Cloud Computing VM (60 hyper threads, 240 GB memory)
- Single Machine
- Intel Xeon Scalable Processor (Cascade Lake)

https://www.freecodecamp.org/news/content/images/2020/10/gcp.png

https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.intel.com%2Fcontent%2
Fwww%2Fus%2Fen%2Fproducts%2Fprocessors%2Fxeon%2Fw-processors%2Fw-3175x.
html&psig=AOvVaw0-nJF6lvyw8oq-5TNfTcFf&ust=1603075410545000&source=images&c
d=vfe&ved=0CAIQjRxqFwoTCKiGjbWPvewCFQAAAAAdAAAAABAF

Experimental Data

- **DBLP:** Co-authorship network
  - 317080 Vertices
  - 1049866 Edges
  - 2224385 Triangles
  - 53.7s for static insertion
  - 2.65s per batch of 100000 edges
- **Youtube:** Video sharing social network where users can make friends
  - 1134890 Vertices
  - 2987624 Edges
  - 3056386 Triangles
  - 368.2s for static insertion
  - 7.31s for batch of 100000 edges

Conclusion

- **Current Work**
  - Strong theoretical bound: **O(bh) work and O(log b + log h) span**
  - Complete analysis for Triangle Counting
  - No significant difference between dynamic array and threshold implementation
- **Future Work**
  - 4-vertex subgraph counting
  - Extended experiments on code

Acknowledgements

- MIT PRIMES
- Jessica and Julian
- Family and friends who have supported us

Questions?