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
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://media.wired.com/photos/5b19a3fd985bbd041c32d0c3/125:94/w_2130,h_1602,c_limit/Summit-supercomputer---side-view-(wide-shot)---TFAA.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
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)
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
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** \( \leq \text{work}/P + O(\text{span}) \) where \( P \) is the number of processors
Parallel primitives

- **Parallel Filter**: Given an array of elements, filter out certain elements and concatenate the gaps afterward.
  - **Bounds**: $O(N)$ work and $O(\log N)$ 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 $O(\log N)$ span

- **Parallel Prefix Sum**: Given a list of numbers, generate a list of prefix sums. Formally, $\text{prefix}[i] = \sum(j = 1 \text{ to } i) \text{ arr}[i]$
  - **Bounds**: $O(N)$ work and $O(\log N)$ span
Parallel primitives

- **Parallel Integer Sort**: Sort a given list of integers.
  - **Bounds**: $O(N)$ work and $O(\log N)$ span

- **Parallel Hashing**: Hashes a list of elements to achieve fast random access.
  - **Bounds**: $O(N)$ work and $O(\log N)$ span

![Diagram showing hash function and keys]

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 $|\text{batch}| = b$
  - 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

**Vertices by Degree**

<table>
<thead>
<tr>
<th>Degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>∅</td>
<td>{3}</td>
<td>{0, 1}</td>
<td>{2}</td>
</tr>
</tbody>
</table>

- $b = |\text{batch}|$
- $h = 2$
- $x = \text{vertex} \notin H$
- $x = \text{vertex} \in H$
- $x = \text{untracked vertices (not in HSet)}$
- Existing edge
- Edge not yet added
Parallel HSet - Removing Vertices

Vertex Degree

<table>
<thead>
<tr>
<th>Degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>Ø</td>
<td>{3} → Ø</td>
<td>{0, 1} → Ø</td>
<td>{2}</td>
</tr>
</tbody>
</table>

h = 2

Batch = all endpoints of all edges = {0, 1, 3}
- Remove in parallel
Parallel HSet - Removing Vertices

h = 2

# of Vertices w/ deg ≥ h

aboveH = 1 vertex

<table>
<thead>
<tr>
<th>Degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>{2}</td>
</tr>
</tbody>
</table>
Parallel HSet - Removing Vertices

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

<table>
<thead>
<tr>
<th>Degree</th>
<th>$h = 2$</th>
<th>1</th>
<th>$\max(0, h - b) = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>---</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prefix Sum</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Ignore size of table[h]
- Already included in aboveH
Parallel HSet - Removing Vertices

Prefix Sum Table

aboveH = 1

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

<table>
<thead>
<tr>
<th>Degree</th>
<th>Prefix Sum (vertices gained)</th>
<th># of vertices above that degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>1 + 0 &lt; 2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1 + 0 ≥ 1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1 + 0 ≥ 0</td>
</tr>
</tbody>
</table>
Parallel HSet - Removing Vertices

Set new $h$ to be the largest degree where $\text{aboveH} + \text{prefixSum}[\text{deg}] \geq \text{deg}$

$h = 1$

<table>
<thead>
<tr>
<th>Degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>{2}</td>
</tr>
</tbody>
</table>
Parallel HSet - Add (or Delete) Edges

h = 1

Add or delete edges (which modifies the degrees)

<table>
<thead>
<tr>
<th>Degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>{2}</td>
</tr>
</tbody>
</table>
Parallel HSet - Re-adding Vertices

\[ h = 1 \]

Batch = all endpoints of all edges = \{0, 1, 3\}
- Add in parallel

<table>
<thead>
<tr>
<th>Degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>{2} → {0, 1, 2, 3}</td>
</tr>
</tbody>
</table>
Parallel HSet - Re-adding Vertices

h = 1

# of Vertices w/ deg ≥ h

aboveH = 4 vertices

<table>
<thead>
<tr>
<th>Degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>{2} → {0, 1, 2, 3}</td>
</tr>
</tbody>
</table>
Parallel HSet - Re-adding Vertices

Degree

<table>
<thead>
<tr>
<th>Degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>{0, 1, 2, 3}</td>
</tr>
</tbody>
</table>

Prefix Sum Table: from $h$ up to $h + b$

<table>
<thead>
<tr>
<th>Degree</th>
<th>$h = 1$</th>
<th>2</th>
<th>3</th>
<th>$h + b = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Prefix Sum</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Parallel HSet - Re-adding Vertices

aboveH = 4

Prefix Sum Table

<table>
<thead>
<tr>
<th>Degree</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix Sum (vertices lost)</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td># of vertices above that degree</td>
<td>4 - 0 ≥ 1</td>
<td>4 - 0 ≥ 2</td>
<td>4 - 4 &lt; 3</td>
<td>4 - 4 &lt; 4</td>
</tr>
</tbody>
</table>

Smallest degree such that aboveH - prefixSum[deg] < deg
Parallel HSet - Re-adding Vertices

Set new $h$ to be the smallest degree where $\text{aboveH} - \text{prefixSum}[\text{deg}] < \text{deg}$

$h = 3$

<table>
<thead>
<tr>
<th>Degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>{0, 1, 2, 3}</td>
</tr>
</tbody>
</table>
Parallel HSet - Result

**Degree**

<table>
<thead>
<tr>
<th>Degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>{0, 1, 2, 3}</td>
</tr>
</tbody>
</table>

$h = 3$

Can determine if a vertex is in $H$ by comparing its degree to the h-index.

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

Add # of Wedges to Count

TOO SLOW!
O(N) per edge

Adjust Wedges Map

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**

  $\times$ = vertex $\notin$ H

  $\circ$ = vertex $\in$ H

  $\circ$ = 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

1. Add # of Wedges to Count
2. Iterate through HSet
3. Adjust Wedges Map
4. Update HSet and Wedges

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 one 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 $O(\min(N^2, Nh^2))$ 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 $O(N^2)$ space
Environment

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