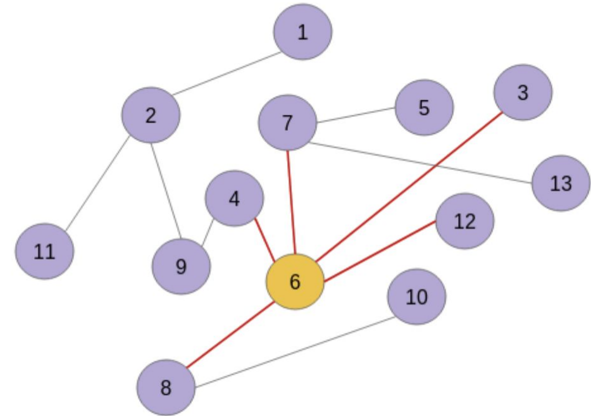
A New Parallel Algorithm for Connected Components in Dynamic Graphs

Robert McColl, Oded Green, David A. Bader

Presented by Temi Taylor

A Parallel Algorithm for Graphs

- **Graphs**: structure with a set of vertices (V) and a set of edges (E)
- Real-world datasets can be represented as graphs:
    - Social media networks
    - Links between websites
    - Research paper references



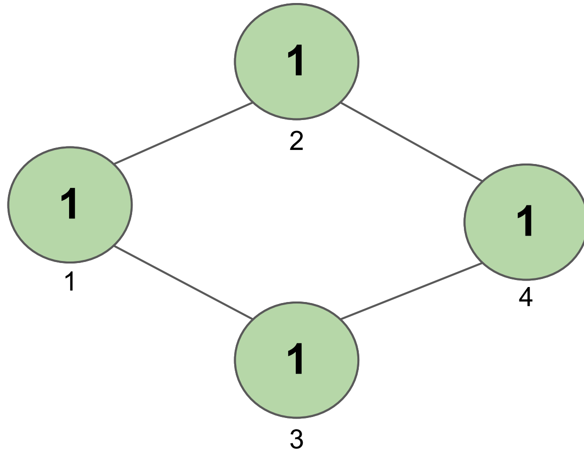
- People love graph computations
- People love parallelism

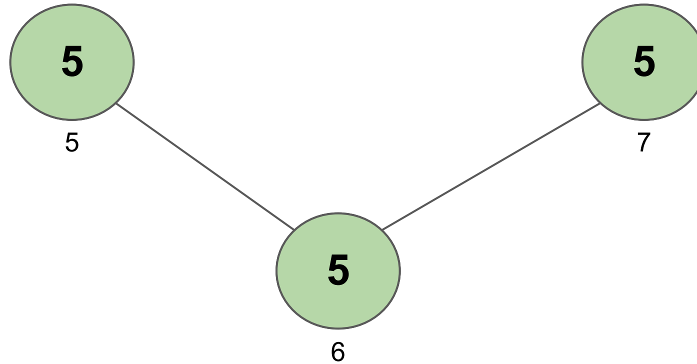
A Parallel Algorithm for Connected Components in Graphs

- **Connected Components**: a problem for undirected graphs
  - Goal: partition vertices into maximal components
  - Any pair of vertices in a component are connected by a path in the full graph
- Often represented by giving all vertices in a component a matching **Label**

- Can be a subroutine in other graph algorithms
  - Betweenness centrality
  - Community detection
  - Image processing

- Connected Components can be solved with BFS (or DFS)





- O(V + E) work
- Worst-case O(V + E) span

A Parallel Algorithm for Connected Components in Dynamic Graphs

- Real graphs can change quickly over time
- Real graphs can also be very large
- BFS is a **Static** approach - would take O(V + E) work for every update

- **Dynamic Algorithms**: focus on the effect of individual updates
  - Saves work if the update doesn't affect the solution
  - Can batch and apply multiple at once

A New Parallel Algorithm for Connected Components in Dynamic Graphs

- Insertions join two components…
  - …but only if the vertices were originally in different components
- Deletions break apart components…
  - …but only if there’s no other path between the vertices

- Goal: a lightweight way to determine whether vertices are still connected
- Up to 30.8x faster than a static recomputation on every update

A New Parallel Algorithm for Connected Components in Dynamic Graphs

- What Existing Algorithms Lacked
- Properties of Real World Graphs

- The Parents-Neighbors Algorithm
- Handling Insertions
- Checking Deletions

- How It Scales
- How It Compares

What Existing Algorithms Lacked

Efficient Static Algorithms

CONNECT

- Analogous to linked-list pointer-jumping
- O(log V * (V + log V)) work, O(log^2 V) span
    - “O(log^2 V) time using V^2 processors” - a remark on its parallelism?

1. **for all** i **do** $D(i) \leftarrow i$
   do steps 2 **through** 6 **for** lg n **iterations**
2. **for all** i **do** $C(i) \leftarrow \min_j \{D(j) \mid A(i,j) = 1 \text{ AND } D(j) \neq D(i)\}$
   if none then $D(i)$
3. **for all** i **do** $C(i) \leftarrow \min_j \{C(j) \mid D(j) = i \text{ AND } C(j) \neq i\}$
   if none then $D(i)$
4. **for all** i **do** $D(i) \leftarrow C(i)$
5. **for** lg n **iterations do**
   for all i **do** $C(i) \leftarrow C(C(i))$
6. **for all** i **do** $D(i) \leftarrow \min\{C(i), D(C(i))\}$

Shiloach-Vishkin

- Similar to CONNECT
- Does a constant number of updates per iteration (instead of lg n)
    - “O(log V) time using V + 2E processors”
- Commonly used at the time of publishing

…but static algorithms do the same amount of work for every update

Previous Dynamic Algorithms

Henzinger et. al.

- Color vertices based on degrees
- Update colors on deletion to detect new components
  - Still requires O(V + E) work for every deletion

Henzinger-King

- Maintain multiple spanning trees for each component
- Only consider component splits when deleting an edge in a tree
  - Reduced work for most updates
  - Can take up a huge amount of storage for large graphs

Previous Dynamic Algorithms

Shiloach-Even

- Components maintain a structure representing a BFS tree
    - **Level** of a vertex: distance from the root
- Same-level deletion: tree not disrupted, no update needed
- Different-level deletion: only split if the deeper vertex has no other neighbors above it

- Constantly updating the tree - amortized O(E + VlogV) time
- Still requires O(V + E) space
- Key idea: tree structure makes connectivity checks more local to vertices

Sparsification

- **Sparsification**: Dividing a graph into subgraphs
- Goal: O(V) edges in each
  - Degree of a subgraph vertex should be bound by a constant

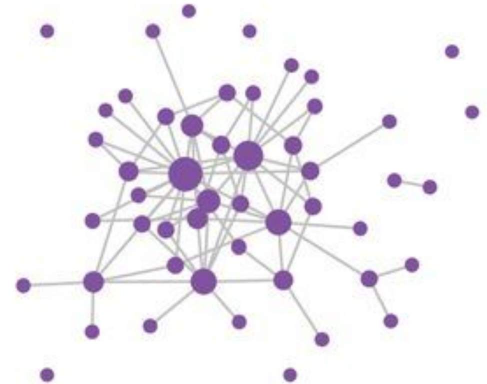
Ferragina

- Used sparsification in a parallel connected components algorithm
- Limits the amount of additional space used
- … but it was a static algorithm

Properties of Real-World Graphs

Real-World Graph Properties

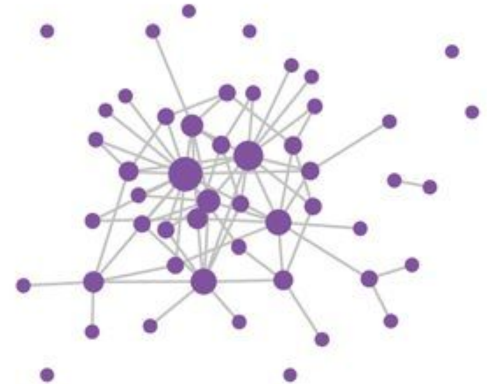
- Small-World Phenomenon
    - Real graphs often have low effective diameters
- Power Law Edge Distribution
    - A small number of vertices are incident to a large portion of edges
- Preferential Attachment
    - High-degree vertices are more likely to be incident to insertions
- Giant Components
    - A single component contains a majority of vertices



Implications

- A BFS tree will have a fairly low depth
    - The span of a parallel BFS will be reasonable
- A single arbitrary deletion is not likely to break apart a component

- Can generally get by with a small portion of the full graph's edges



The Parents-Neighbors Algorithm

(they didn’t give it an actual name)

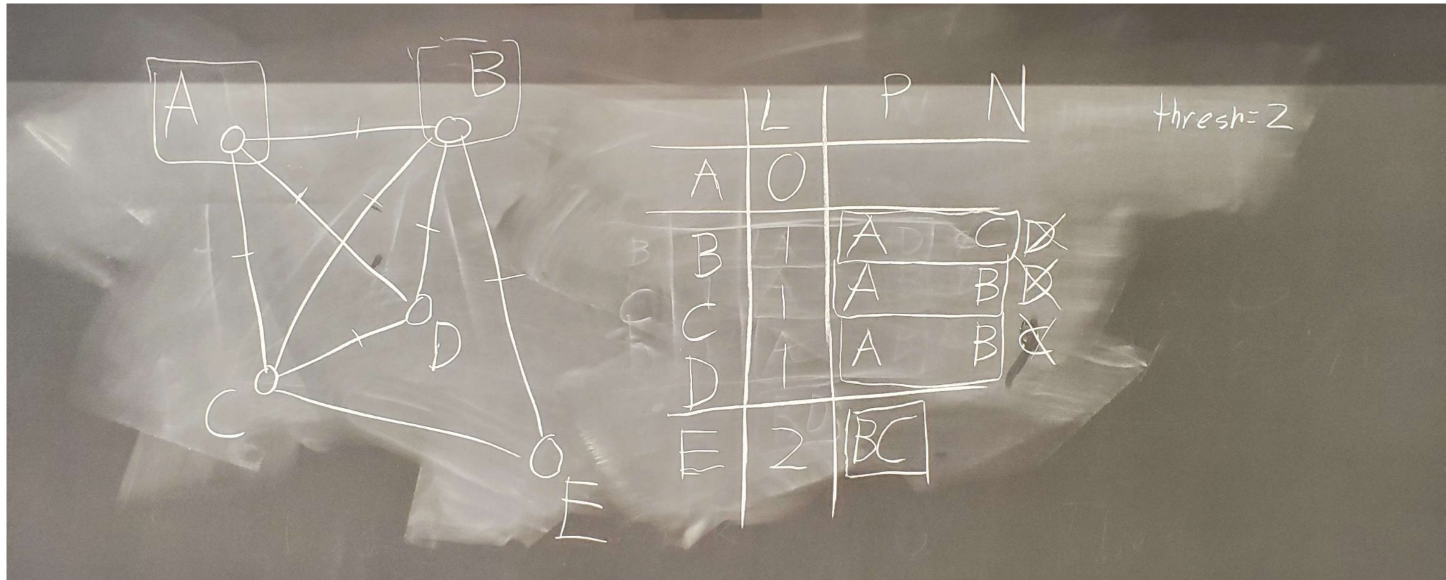
The Parents-Neighbors Subgraph

- Like Shiloach-Even, components maintain a structure representing a BFS tree
- **Parents** of a vertex: adjacent vertices in the previous level
- **Neighbors** of a vertex: adjacent vertices from the same level

- A vertex keeps a list of parents and neighbors in the order they visit it
    - Because this is a BFS, all parents will come before any neighbors
- Sparsification - a **Threshold** limits how many parents/neighbors are tracked

The Parents-Neighbors Subgraph

- Example built using a BFS from vertex A - try to trace it out



Space Taken

Table I

THE DATA STRUCTURES MAINTAINED WHILE TRACKING DYNAMIC CONNECTED COMPONENTS.

Name	Description	Type	Size (Elements)
C	Component labels	array	$O(V)$
$Size$	Component sizes	array	$O(V)$
$Level$	Approximate distance from the root	array	$O(V)$
PN	Parents and neighbors of each vertex	array of arrays	$O(V \cdot thresh_{PN}) = O(V)$
$Count$	Counts of parents and neighbors	array	$O(V)$
$thresh_{PN}$	Maximum count of parents and neighbors for a given vertex	value	$O(1)$
$\tilde{E}_I$	Batch of edges to be inserted into graph	array	$O(batch\ size)$
$\tilde{E}_R$	Batch of edges to be deleted from graph	array	$O(batch\ size)$

Input (Updates) Parents-Neighbors Subgraph Output (Components)

Everything aside from the input needs at most O(V) space, as desired

Handling Insertions

Insertions

- Compare labels to see if the vertices are in separate components
- If so:
  - The two components are being merged
  - Use a BFS to spread one component's label to the other
    - Can also update the subgraph
- If not:
  - No merge needed, but still need to manage the subgraph
  - Add the new neighbor/parent to the destination's PN list, possible
    - Parents can kick out neighbors, but not vice-versa

Checking Deletions

Safe Deletions

- Goal: efficiently determine if another path exists between the vertices
  - Explicitly searching for a path - O(V + E) Work

- **Safe deletion:** a deletion guaranteed not to cause a component split
  - Only unsafe deletions need to search for a path in some way
- Can do an approximate safety check in constant time
- Efficiency depends on minimizing incorrect unsafe deletion reports

Valid Vertices

- In a tree, if two vertices both have paths to the root, they are connected
- Vertices marked as **valid** can *definitely* be used in paths to the root

- All vertices are valid by default
- Deletions can invalidate vertices by removing their subgraph parents
  - If they also lack valid neighbors, the deletion is (probably) unsafe
- Insertions/component splits can re-validate vertices while updating the tree

Handling Unsafe Deletions

- Mark one vertex as the root of a new component
- Run a BFS to update labels and subgraph
- If a connection to the original component is found, use a second BFS to merge updated vertices back in
    - Handles the possibility of a false unsafety report
    - No vertices closer to the old root than the new root will be reached

- Unsafe deletions handled serially in case they were already resolved
- Edges won’t be traversed by multiple deletion handlings
    - Asymptotically comparable to static recomputation, but rarely happens if not necessary

Pseudocode

Creating Initial Components

Algorithm 1 A parallel breadth-first traversal that extracts the parent-neighbor subgraph.

Input: $G(V, E)$
Output: C_{id}, $Size$, $Level$, PN, $Count$

1: **for** $v \in V$ **do**
2: $Level[v] \leftarrow \infty$, $Count[v] \leftarrow 0$
3: **for** $v \in V$ **do**
4: **if** $Level[v] = \infty$ **then**
5: $Q[0] \leftarrow v$, $Q_{start} \leftarrow 0$, $Q_{end} \leftarrow 1$
6: $Level[v] \leftarrow 0$, $C_{id}[v] \leftarrow v$
7: **while** $Q_{start} \neq Q_{end}$ **do**
8: $Q_{stop} \leftarrow Q_{end}$
9: **for** $i \leftarrow Q_{start}$ **to** Q_{stop} **in parallel do**
10: **for each neighbor** d **of** $Q[i]$ **do**
11: **if** $Level[d] = \infty$ **then**
12: $Q[Q_{end}] \leftarrow d$
13: $Q_{end} \leftarrow Q_{end} + 1$
14: $Level[d] \leftarrow Level[Q[i]] + 1$
15: $C_{id}[d] \leftarrow C_{id}[Q[i]]$
16: **if** $Count[d] < thresh_{PN}$ **then**
17: **if** $Level[Q[i]] < Level[d]$ **then**
18: $PN_d[Count[d]] \leftarrow Q[i]$
19: $Count[d] \leftarrow Count[d] + 1$
20: **else if** $Level[Q[i]] = Level[d]$ **then**
21: $PN_d[Count[d]] \leftarrow -Q[i]$
22: $Count[d] \leftarrow Count[d] + 1$
23: $Q_{start} \leftarrow Q_{stop}$
24: $Size[v] \leftarrow Q_{end}$

BFS Queue

Assign discovered vertices a level and component label

Update the parents/neighbors list (neighbors are negative)

Insertions - Same Component

Algorithm 2 The algorithm for updating the parent-neighbor subgraph for inserted edges.

Input: $G(V, E)$, $\tilde{E}_I$, C_{id}, $Size$, $Level$, PN, $Count$
Output: C_{id}, $Size$, $Level$, PN, $Count$

```
1: for all⟨s, d⟩ ∈ Ẽ_I in parallel do E ← E ∪ ⟨s, d⟩
2:     insert(E, ⟨s, d⟩)
3:     if C_id[s] = C_id[d] then
4:         if Level[s] > 0 then
5:             if Level[d] < 0 then
6:                 // d is not "safe"
7:                 if Level[s] < −Level[d] then
8:                     if Count[d] < thresh_PN then
9:                         PN_d[Count[d]] ← s
10:                        Count[d] ← Count[d] + 1
11:                    else
12:                        PN_d[0] ← s
13:                    Level[d] ← −Level[d]
14:            else
15:                if Count[d] < thresh_PN then
16:                    if Level[s] < Level[d] then
17:                        PN_d[Count[d]] ← s
18:                        Count[d] ← Count[d] + 1
19:                    else if Level[s] = Level[d] then
20:                        PN_d[Count[d]] ← −s
21:                        Count[d] ← Count[d] + 1
22:                else if Level[s] < Level[d] then
23:                    for i ← 0 to thresh_PN do
24:                        if PN_d[i] < 0 then
25:                            PNV_d[i] ← s,
26:                            Break for-loop
27:        Ẽ_I ← Ẽ_I \ ⟨s, d⟩
```

Handle these in parallel

Make sure that the source is valid
If destination is invalid, can revalidate

If there's space in d's PN list, add s

Otherwise, if s is a parent, see if there's a neighbor to kick out

Insertions - Separate Components

```
28: for all⟨s, d⟩ ∈ Ẽ_I do
29:     if C_id[s] ≠ C_id[d] then
30:         if Size[s] = 1 then
31:             Size[s] ← 0
32:             Size[d] ← Size[d] + 1
33:             C_id[s] ← C_id[d], PN_s[0] ← d
34:             Level[s] ← abs(Level[d]) + 1, Count[s] ← 1
35:         else
36:             connectComponent(Input, s, d)
```

Handle these serially
(make sure it wasn't already handled)

Optimization for singleton components

Parallel BFS from the smaller component
Updates labels, recomputes parents/neighbors

Deletions - Safety Check

Algorithm 3 The algorithm for updating the parent-neighbor subgraph for deleted edges.

Input: $G(V, E)$, $\tilde{E}_R$, C_{id}, $Size$, $Level$, PN, $Count$
Output: C_{id}, $Size$, $Level$, PN, $Count$

1: **for all** $\langle s, d\rangle \in \tilde{E}_R$ **in parallel do**
2: $E \leftarrow E \backslash \langle s, d\rangle$
3: $hasParents \leftarrow$ false
4: **for** $p \leftarrow 0$ **to** $Count[d]$ **do**
5: **if** $PN_d[p] = s$ **or** $PN_d[p] = -s$ **then**
6: $Count[d] \leftarrow Count[d] - 1$
7: $PN_d[p] \leftarrow PN_d[Count[d]]$
8: **if** $PN_d[p] > 0$ **then**
9: $hasParents \leftarrow$ true
10: **if** (not $hasParents$) **and** $Level[d] > 0$ **then**
11: $Level[d] \leftarrow -Level[d]$
12: **for all** $\langle s, d\rangle \in \tilde{E}_R$ **in parallel do**
13: **for all** $p \in PN_d$ **do**
14: **if** $p \geq 0$ **or** $Level$[abs(p)] > 0 **then**
15: $\tilde{E}_R \leftarrow \tilde{E}_R \backslash \langle s, d\rangle$
16: $PREV \leftarrow C_{id}$
17: **for all** $\langle s, d\rangle \in \tilde{E}_R$ **do**
18: $unsafe \leftarrow (C_{id}[s] = C_{id}[d] = PREV_s)$
19: **for all** $p \in PN_d$ **do**
20: **if** $p \geq 0$ **or** $Level$[abs(p)] > 0 **then**
21: $unsafe \leftarrow$ false
22: **if** $unsafe$ **then**
23: **if** $\{\langle u, v\rangle \in G(E, V) : u = s\} = \emptyset$ **then**
24: $Level[s] \leftarrow 0, C_{id}[s] \leftarrow s$
25: $Size[s] \leftarrow 1, Count[s] \leftarrow 0$
26: **else**
27: **Algorithm 4**
28: **repairComponent(Input**, s, d)

Invalidate vertices with no parents
(set level to negative)

Safety check -
Look for parents or valid neighbors

Handle unsafe deletes serially
Repeat safety check in case already handled

Another singleton component optimization

Fixes PN lists, revalidates vertices

Deletions - Handling Unsafe Deletions

Algorithm 4 The algorithm for repairing the parent-neighbor subgraph when an unsafe deletion is reported.

Input: $G(V, E)$, $\tilde{E}_R$, C_{id}, $Size$, $Level$, PN, $Count$, s, d
Output: C_{id}, $Size$, $Level$, PN, $Count$

```
1:  Q[0] ← d, Q_start ← 0, Q_end ← 1
2:  SLQ ← ∅, SLQ_start ← 0, SLQ_end ← 0
3:  Level[d] ← 0, C_id[d] ← d
4:  disconnected ← true
5:  while Q_start ≠ Q_end do
6:      Q_stop ← Q_end
7:      for i ← Q_start to Q_stop in parallel do
8:          u ← Q[i]
9:          for each neighbor v of u do
10:             if C_id[v] = C_id[s] then
11:                 if Level[v] ≤ abs(Level[d]) then
12:                     C_id[v] ← C_id[d]
13:                     disconnected ← false
14:                     SLQ[SLQ_end] ← v
15:                     SLQ_end ← SLQ_end + 1
16:                 else
17:                     C_id[v] ← C_id[d]
18:                     Count[v] ← 0
19:                     Level[v] ← Level[u] + 1
20:                     Q[Q_end] ← v
21:                     Q_end ← Q_end + 1
22:             if Count[v] < thresh_PN then
23:                 if Level[u] < Level[v] then
24:                     PN_v[Count[v]] ← u
25:                     Count[v] ← Count[v] + 1
26:                 else if Level[v] = Level[v] then
27:                     PN_v[Count[v]] ← −u
28:                     Count[v] ← Count[v] + 1
29:     Q_start ← Q_stop
```

Track whether or not a link to the original root was found

Basically the same as initialization

Check if a vertex is closer to the old root than the new root

If so, add it to the queue and prepare for reverse BFS

Deletions - Handling Unsafe Deletions (Second BFS)

```
30: if disconnected then
31:     Size[d] ← Q_end
32: else
33:     for i ← SLQ_start to SLQ_end in parallel do
34:         C_id[i] ← C_id[s]
35:     while SLQ_start ≠ SLQ_end do
36:         SLQ_stop ← SLQ_end
37:         for i ← SLQ_start to SLQ_stop in parallel do
38:             u ← SLQ[i]
39:             for each neighbor v of u do
40:                 if C_id[v] = C_id[d] then
41:                     C_id[v] ← C_id[u]
42:                     Count[v] ← 0
43:                     Level[v] ← Level[u] + 1
44:                     SLQ[SLQ_end] ← v
45:                     SLQ_end ← SLQ_end + 1
46:                 if Count[v] < thresh_PN then
47:                     if Level[u] < Level[v] then
48:                         PN_v[Count[v]] ← u
49:                         Count[v] ← Count[v] + 1
50:                     else if Level[v] = Level[v] then
51:                         PN_v[Count[v]] ← −u
52:                         Count[v] ← Count[v] + 1
53:         Q_start ← Q_stop
```

If no connection to the root, the first BFS built the new component, so we're done

Otherwise, go back to the original label

Again, very similar to initialization

Finds vertices changed by first BFS,
Re-integrates them into the component

How It Scales

Implementation and Testing

- Implemented using STINGER
  - A data structure for dynamic graphs co-developed by the authors
  - Balances the update efficiency of Adjacency Matrices with the storage efficiency of CSR

- Generated input graphs using R-MAT
  - **R**ecursive weighted random choice of adjacency **mat**rix quadrant to add an edge to
  - Gives graphs with power-law edge distributions and giant components
  - Varied the graph size and number of initial edges

- Tested using streams of random updates
  - Each update has a fairly small chance (6.25%) to be a deletion of an already-inserted edge
  - Given in batches of 100K updates

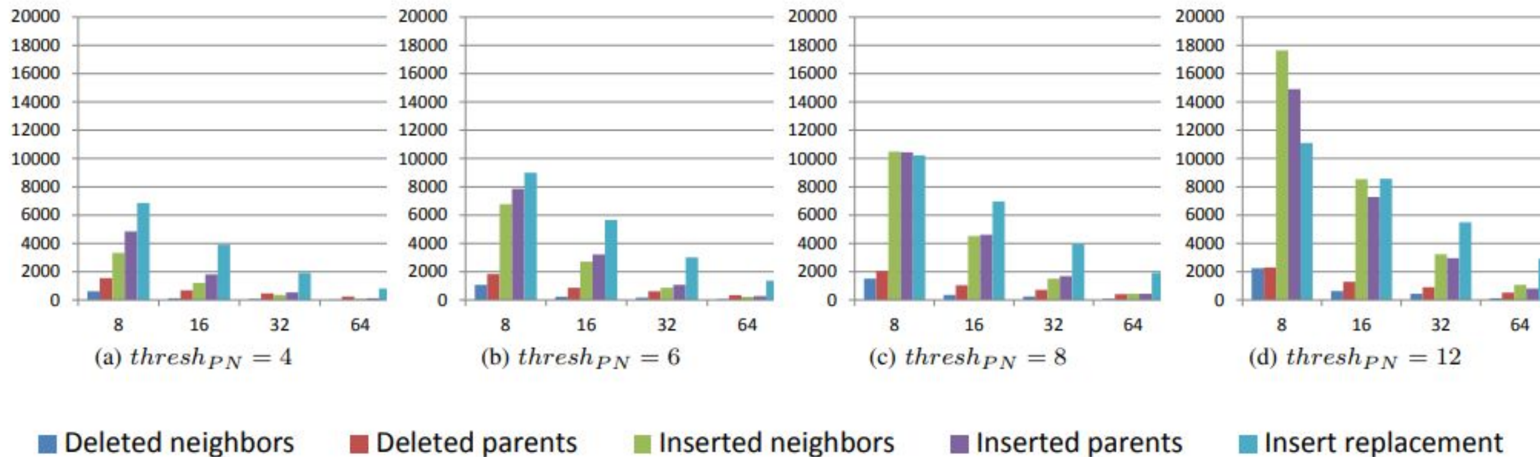


Figure 1. Average number of inserts and deletes in PN array for batches of $100K$ updates for RMAT-22 graphs. The subfigures are for different values of $thresh_{PN}$. Note that the ordinate is dependent on the specific bar chart. The charts for RMAT-21 graphs had very similar structure and have been removed for the sake of brevity.

Subgraph updates decrease with density

Generally less computation

Higher-density graphs initially have more parents than neighbors

Fewer replacements as edges added

Higher thresholds mean the subgraph contains more edges

More work checking deletions in denser graphs

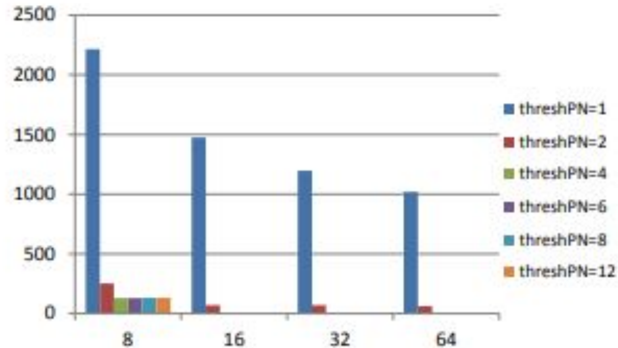


Figure 2. Average number of unsafe deletes in PN data structure for batches of $100K$ updates as a function of the average degree (x-axis) and $thresh_{PN}$ (bars).

Higher thresholds report less deletions as unsafe — Less work wasted on incorrect safety reports

Higher thresholds mean the subgraph contains more edges — More work checking deletions in denser graphs

What's a good setting for the threshold?

4 (because diminishing returns)

(i think the paper accidentally swapped fig 3 and fig 4)

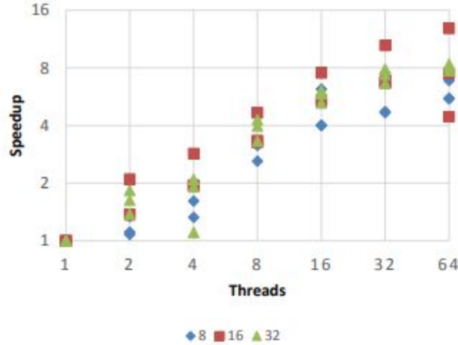


Figure 4. Speed up of the new algorithm over performing parallel static recomputation after each batch on three different RMAT-22 graphs with each average degree as a function of the number of threads.

Speedup compared to a single thread

- Nearly linear up to 32 threads, though not optimal
- Slightly improves as density increases

Portion of time spent updating subgraph

- Constant across thread count
- Decreases as density increases

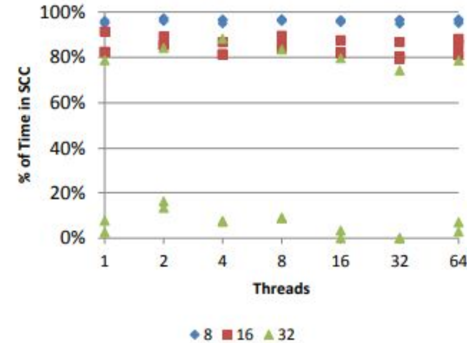


Figure 5. Fraction of the update time spent updating connected components over time spent updating the graph structure and connected components.

How It Compares

(i think the paper accidentally swapped fig 3 and fig 4)

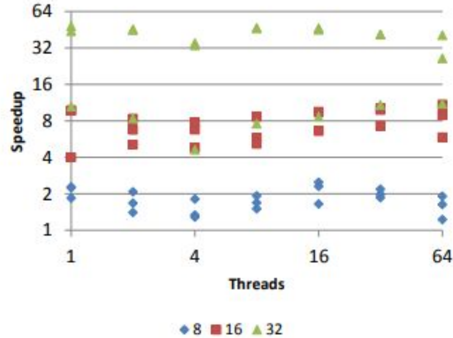


Figure 3. Strong scaling results on RMAT-22 graphs with different average degree as a function of the number of threads. Results include three graphs at each average degree.

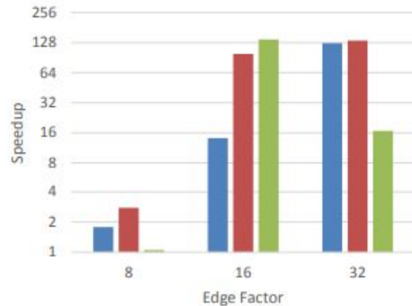


Figure 6. Speed up over performing static recomputation after each batch on scale 24 graphs for three graphs at each edge factor using 64 threads.

Speedup compared to running Shiloach-Vishkin after each batch

- Increases with graph size and density
- High variance between different graphs of the same size
  - Parents-neighbors algorithm is more sensitive to the graph structure and update stream
- Static algorithm’s cost is constant, but this algorithm can adapt to how often components actually change

Closing Thoughts

- Mentions using a more efficient BFS algorithm as a point for future work
    - Makes sense, since most of the heavy lifting is done by various forms of BFS
    - Could potentially benefit from vertex reordering
- Why was there no direct comparison to other dynamic algorithms?
    - Maybe because they implied that many of them weren't as space efficient
    - There is no theoretical comparison of the work done
- Pseudocode was thorough and pretty well-explained
    - You could probably figure out the work yourself if you really wanted to
        - Asymptotically, it would probably be pretty close to the work of a BFS

Image Credits

Generic Graph Example - https://blogs.cornell.edu/info2040/2020/10/02/using-graph-theory-to-identify-social-media-influencers-for-marketing/

Power Law Graph Example - https://slideplayer.com/slide/15433764/

Algorithms and related tables/graphs - from the paper

Deletions - Other Ideas

- Alternative ways to check deletions?
  - Intersections in the two vertices' adjacency lists
    - Only finds alternate paths of length 2; too many false unsafe reports
  - Spanning trees for each component (like some related work)
    - Safe if the deleted edge is not in the tree
    - Found to correctly identify only 90% of safe deletions
    - Needs additional recomputation if unsafe
  - Two spanning trees for each component
    - One avoids using the same edges as the other, safe if a parent exists in either tree
    - Correctly identifies 99.7% of safe deletions
    - Even more computationally intensive
  - Using a BFS to find a shortest path between the vertices
    - Due to real-world graph properties, can quickly end up including most of the component - lots of unnecessary work
- The presented approach was experimentally found to perform the best of the proposed ideas