A New Parallel Algorithm for Connected Components in Dynamic Graphs

By: Robert McColl, Oded Green, David A. Bader

College of Computing, Georgia Institute of Technology

Presentation by: Jessica Zhu

Motivations

- Real world graphs are billions of edges large and are constantly changing
- Statically computing analysis on these graphs isn’t sufficient
- Dynamic graph algorithms keep graphs up to date without re-computing for the entire graph

Background

- Connected component: there is a path between any two vertices in the same connected component
- If you know the connected components in a graph, determining if adding an edge joins two connected components is easy O(1), but determining if deleting an edge separates a connected component is harder

Related Work

- Many existing algorithms
  - Using DFS or BFS
  - Treating dense graphs as sparse graphs
  - Coloring vertices based on their degrees

- Too expensive in practice + too much storage
- Don’t take into account properties of real-world graphs
- Don’t consider multi-core systems

Contributions

- First dynamic algorithm for connected components that does not involve static recomputation
- Keeps an exact labeling of connected components by applying additions and deletions in batches
- STINGER: a high-performance graph data structure to solve dynamic graph problems
  - Faster insertions and better locality than adjacency lists
  - Designed for parallelism

The Algorithm: Challenges

- Correctness
- Parallelism
- Time Complexity
  - Must be better than a static implementation
- Storage Complexity
  - Better than a static implementation
  - Preferred: O(V), Max: O(V+E)

- Deletions: minimize false negatives (when you think a deletion creates two connected components but doesn’t actually)

The Algorithm: Parent—Neighbors Subgraph

- Extracted by running BFS on the original graph
  - One iteration of BFS per connected component
- Each vertex knows who its parents and neighbors (same level) are
  - Place a threshold on how much a vertex can store, memory becomes O(V*threshold)
  - Storing everything would cost O(V+E) space
- Memory: O(V)

The Algorithm: Data Structure + Details

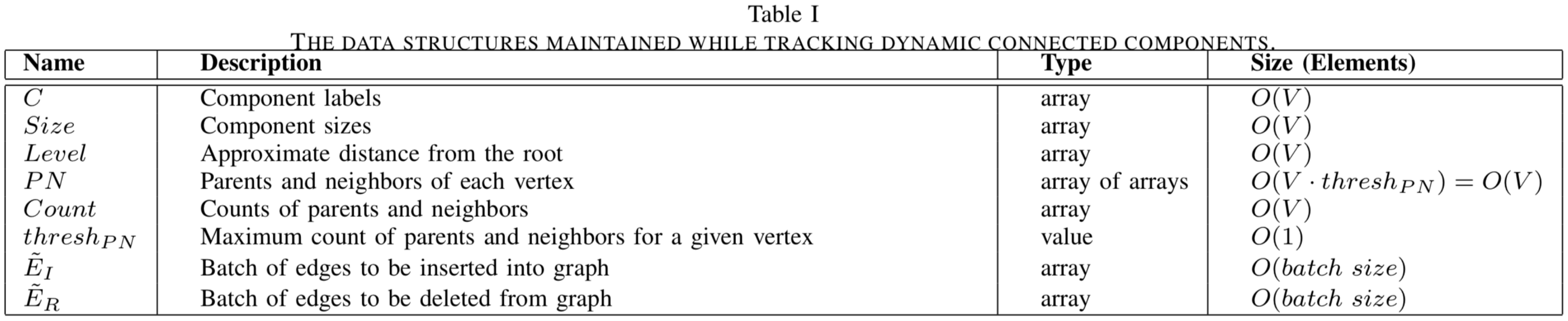


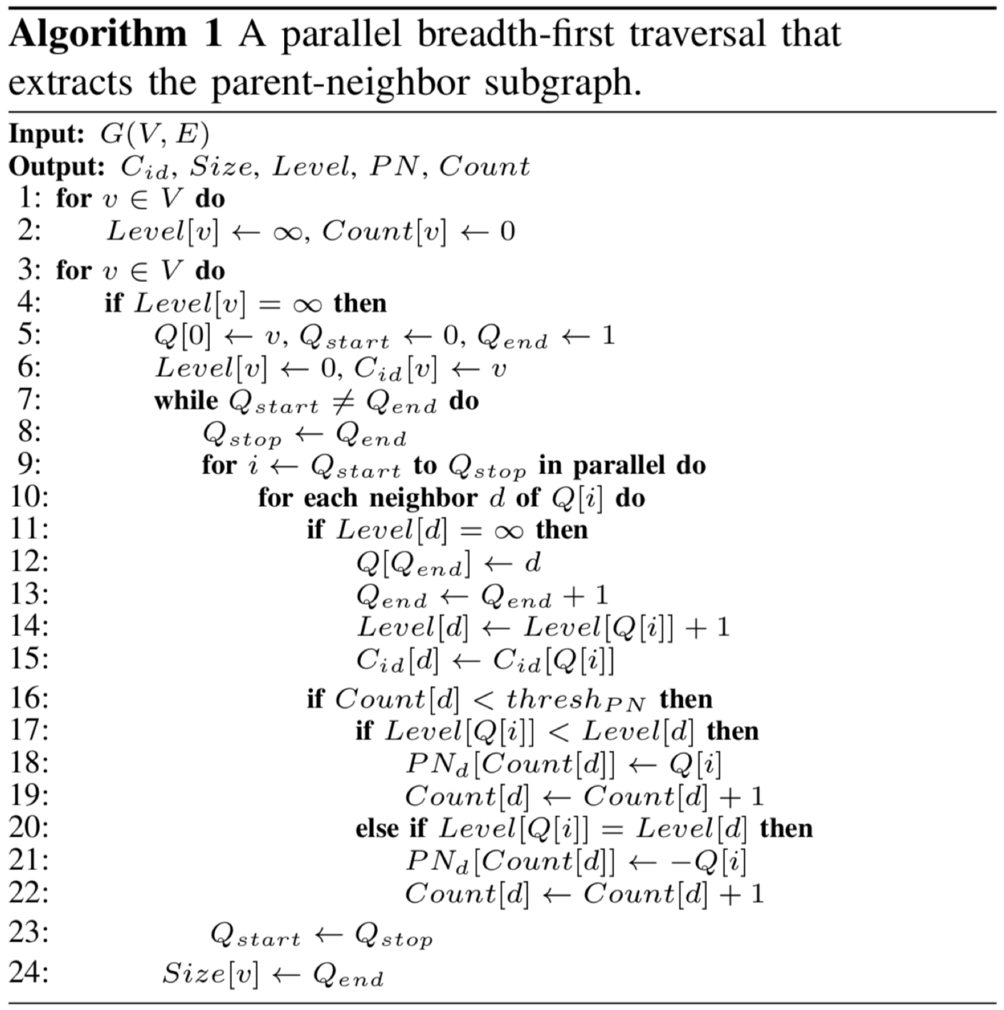
Table I
THE DATA STRUCTURES MAINTAINED WHILE TRACKING DYNAMIC CONNECTED COMPONENTS.

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	$O(batch\ size)$

The Algorithm: Initialization

- Every vertex’s level is not exact, but an approximation of how far it is from the root
- Every vertex’s level starts at infinity, counter = 0
- Start with arbitrary vertex, look at all of its neighbors concurrently
- Parents are always added before neighbors
- Synchronization handled through atomic compare-and-swap operations on level and fetch-and-add to PN and counter

The Algorithm



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:         $Size[v] \leftarrow Q_{end}$
```

Insertions

- The added edge (s, d) is within one connected component
  - Check the levels of s and d to see if there are new parent or neighbor connections
  - If level(s) <= level(d) and counter(d) < threshold, s is a parent of d
  - If counter(d) = threshold, look through parents and neighbors to replace something with s
- The added edge (s,d) joins two components
  - Parallel BFS starts at joint of smaller connected component and adds all vertices from that component into PN of larger component

Deletions

- Deleting an edge (s, d): check from both s and d’s perspective (undirected graph will be symmetric in this regard)
  - If level(s) <= level(d), if s is in PN(d), it is removed and we look for a remaining parent in PN(d)
    - If a parent still exists, deletion is safe in this direction
    - If a parent does not still exist, set a marker that is only removed once a new parent for d is found and update PN
    - Run BFS using d as the root node, and see if it encounters a vertex with a level less than it in the original component

- Worst case: must look at all edges in the graph

Experimental Setup

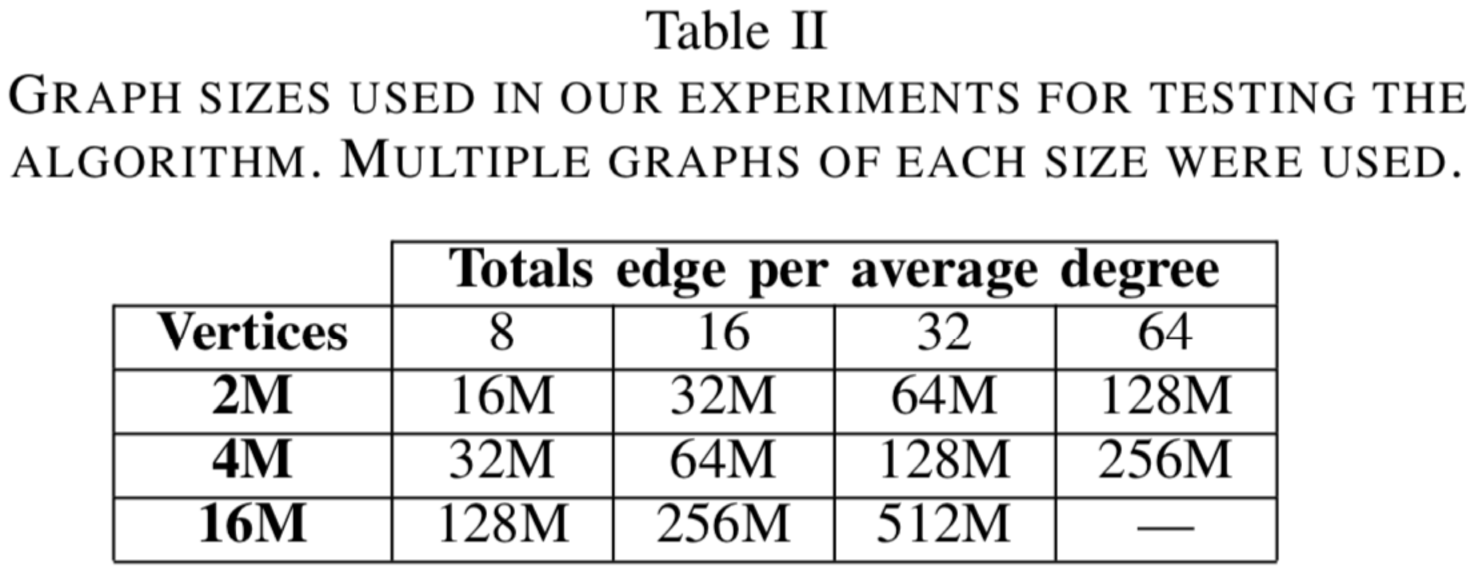


Table II
GRAPH SIZES USED IN OUR EXPERIMENTS FOR TESTING THE ALGORITHM. MULTIPLE GRAPHS OF EACH SIZE WERE USED.

	Totals edge per average degree			
Vertices	8	16	32	64
2M	16M	32M	64M	128M
4M	32M	64M	128M	256M
16M	128M	256M	512M	—

Experimental Results

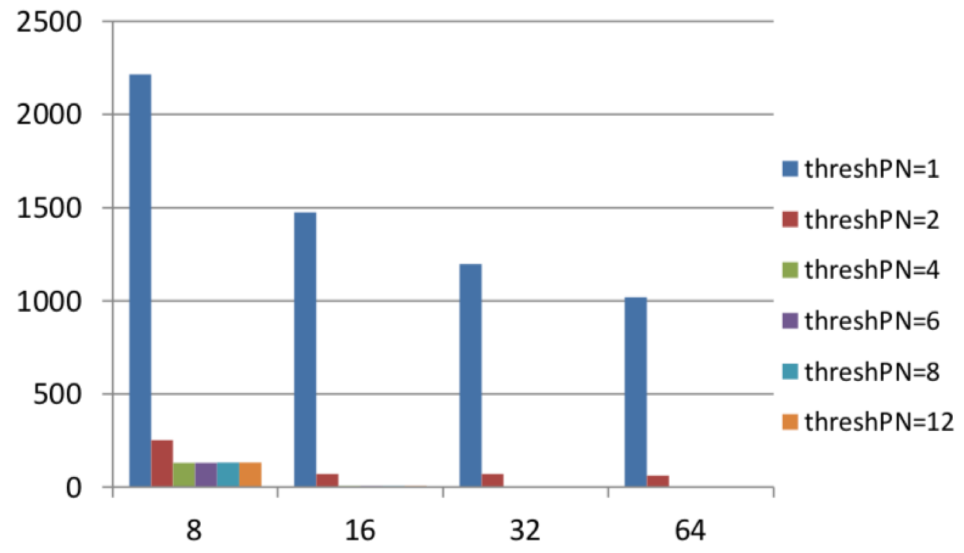


Figure 2. Average number of unsafe deletes in PN data structure for batches of $100K$ updates as a function of the average degree (x-axis) and $thresh_{PN}$ (bars).

Scalability

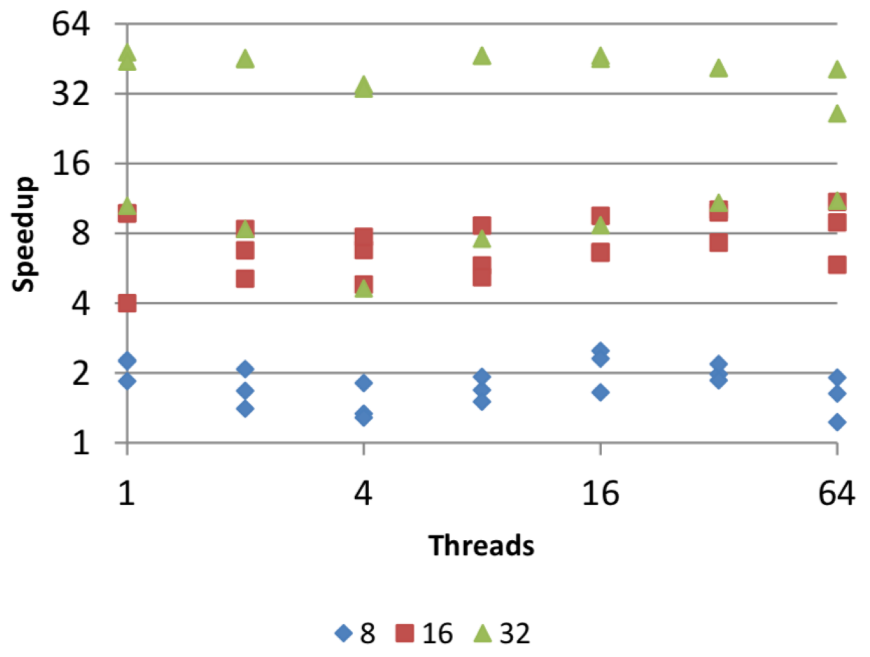
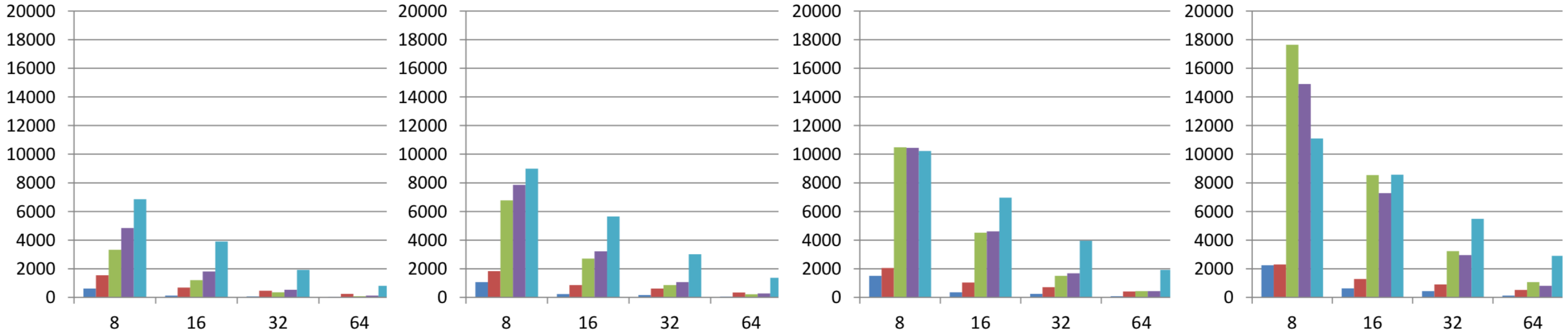


Figure 3. Strong scaling results on RMAT-22 graphs with different average degree as a function of the number of threads. Results include three graphs at each average degree.

Performance



(a) $thresh_{PN} = 4$ (b) $thresh_{PN} = 6$ (c) $thresh_{PN} = 8$ (d) $thresh_{PN} = 12$

🟦 Deleted neighbors 🟥 Deleted parents 🟩 Inserted neighbors 🟪 Inserted parents 🟦 Insert replacement

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.

Performance

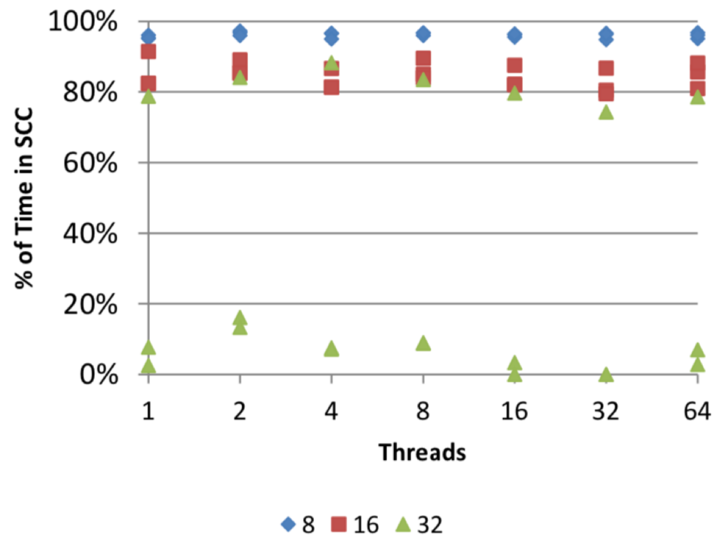


Figure 5. Fraction of the update time spent updating connected components over time spent updating the graph structure and connected components.

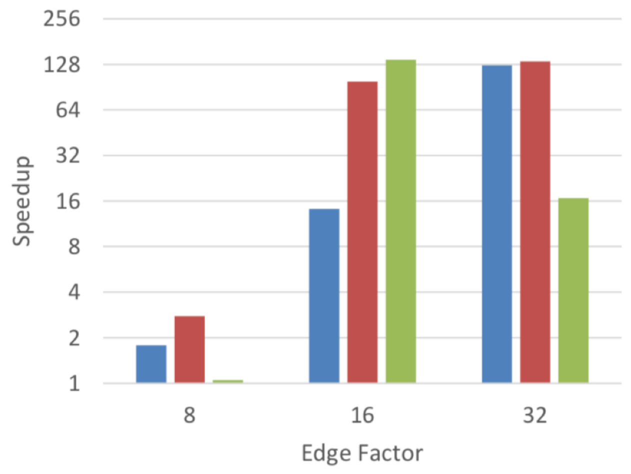


Figure 6. Speed up over performing static recomputation after each batch on scale 24 graphs for three graphs at each edge factor using 64 threads.

Strengths and Weaknesses

- Well organized paper, clear sections and explanations of algorithm
- Pseudocode is very detailed and helpful
- Lacking comparison to other algorithms (in terms of runtime, correctness, etc.)
- Not compared against real-world graphs

Discussion Questions

- What is the optimal tradeoff between balancing unsafe deletes and performance?
- Is there a theoretical way to determine the optimal value of the threshold?