A New Parallel Algorithm for Connected Components in Dynamic Graphs

Paper Review

Presented by Alex Mcneilly

Authors (all at Georgia Tech in 2013)

Robert McColl
Georgia Tech, MS
(2010-2014?)
CTO, Documi (2021—)
Parallel streaming graph applications, STINGER

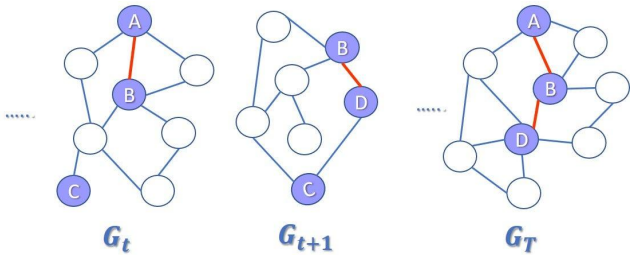
Oded Green
Georgia Tech, PhD
(2010-2014)
NVIDIA (2018—)
Large-scale graph algos, dynamic graphs, STINGER

David A. Bader
NJIT (2019 —)
Executive Director, HPC Group, Georgia Tech
(2010-2019)
STINGER

STINGER: dynamic graph data structure developed by Georgia Tech, basis for implementation

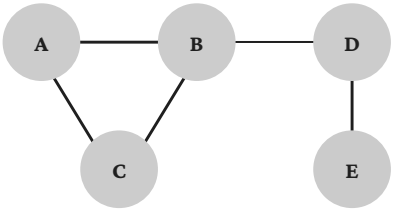
Motivation

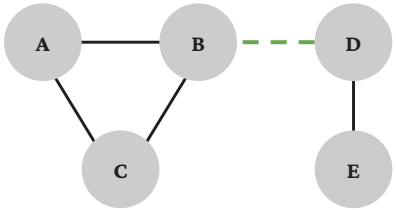
- Social networks, web graphs, and scientific datasets now have billions of relationships
- These relationships are **dynamic**, subject to change frequently
- Classic algorithms were designed for **static graphs** (snapshot)
- The time to re-run a static $O(V + E)$ algorithm is longer than time between graph changes
- We need to handle batches of updates **incrementally**



Challenge: Deletions are Expensive

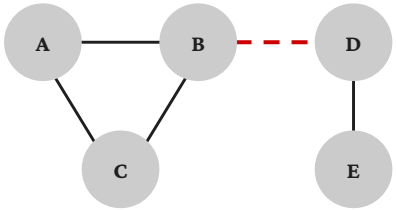
- We need to handle batches of updates **incrementally**
- **Insertions:** When adding *(u, v)*, just check if their component labels are different (*O(1)*)
- **Deletions:** When deleting *(u, v)*, did we just sever the *only* path holding a component together?
- Seeing if a deletion is safe requires a new search (BFS/SPSP), which is *O(V + E)*





O(1) check for diff. components

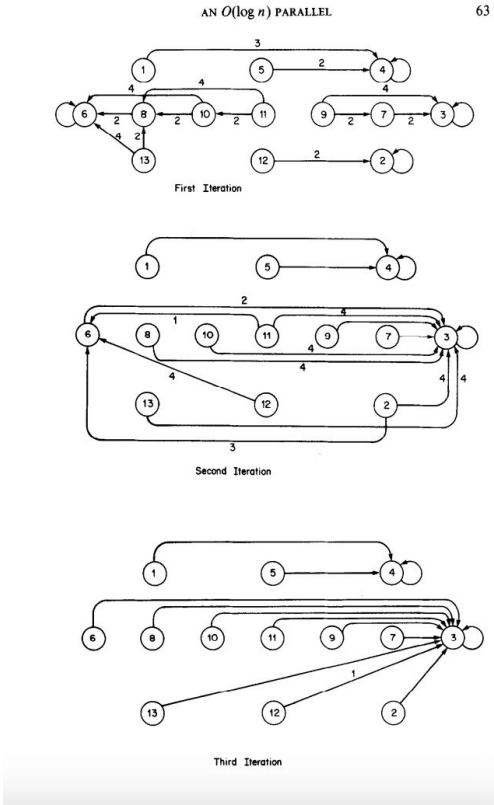
- Maintain an **exact** connected components labeling for a dynamic graph
- Process large batches of modifications **in parallel**
- Be faster than static recomputation, low memory



Is B still connected to D? Check.

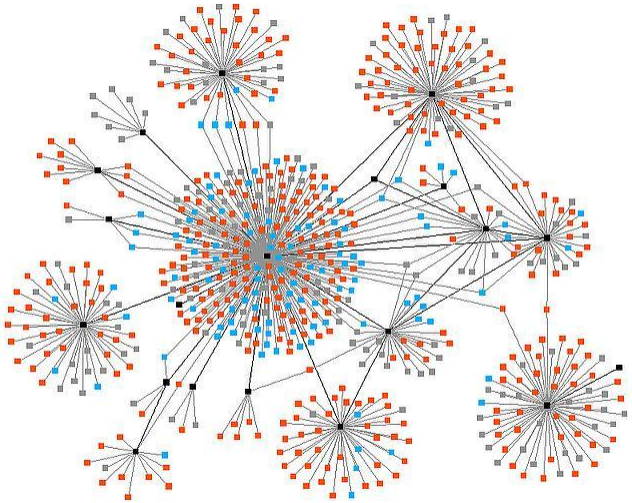
Related Work

- Static Parallel Algorithms (Shiloach-Vishkin)
  - A classic PRAM algorithm that finds components in *O(log V)* time. It's highly parallel and scalable.
  - It is a **static** algorithm. It recomputes *everything* from scratch for each batch.
  - Static recomputation baseline for paper
- Theoretical Dynamic Algorithms (Henzinger, King)
  - Algorithms that maintain complex auxiliary data structures, like multiple spanning trees, to answer queries
  - Too expensive in practice (*O(V + E)* memory, not parallel)



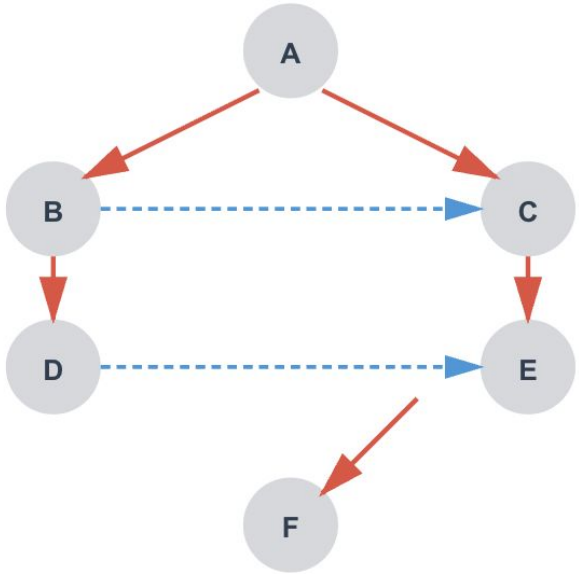
Initial Attempts

- Real-world graphs have power law distributions and are “small-world” (robust)
- A single edge deletion is **very unlikely** to disconnect a component
- **New Idea:** We shouldn’t need to track *O(E)* paths.
- Track a *small, fixed* number of “backup paths” for each vertex and get 100% accuracy with *O(V)* storage



Parent-Neighbor (PN) Sub-graph

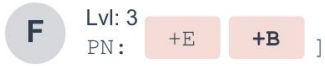
- We want to avoid an $O(V + E)$ search for alternate paths on every deletion
- What if we don't need *all* alternate paths? What if we just need *enough* to prove a vertex is still connected to the root?



- A directed sub-graph of the original graph extracted by a parallel BFS to prove connectivity.
- Each vertex v stores a small, fixed-size array $PN[v]$
- The size is set by a constant $thresh\_PN$ (4, 8, 12)
- **Total Storage:** $O(V * thresh\_PN) \rightarrow O(V)$

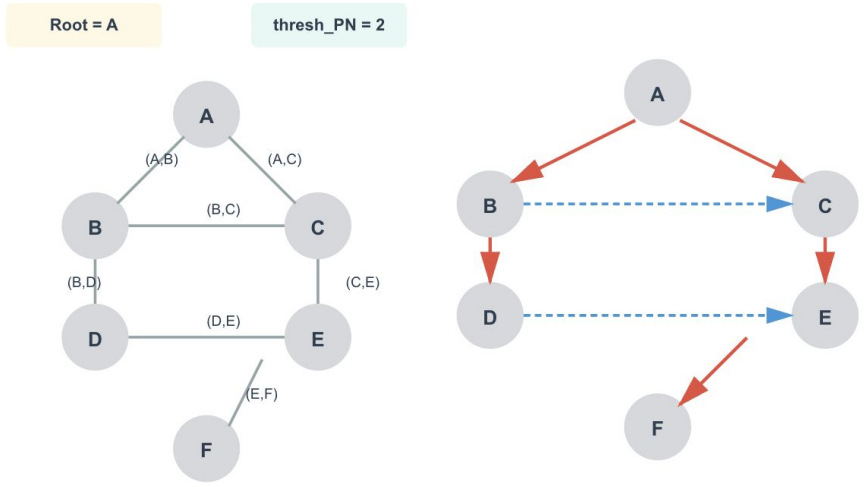
Parent-Neighbor (PN) Sub-graph

- What's in the *PN* array?
  - **Parents (+ID)**
    - A vertex *u* that is 1 level closer to the root
    - Stored as a positive integer
  - **Neighbors (-ID)**
    - A vertex *w* that is at the same level as *v*
    - Stored as a negative integer
  - **Level**
    - Approximate distance from the root
  - **Component ID (C_id)**



Algorithm 1: Building the PN Sub-graph

- Start at root (unlabeled vertex)
- BFS explores frontiers in parallel
- When exploring an edge *(u, v)*, where *u* in frontier:
  - If *v* is undiscovered (Level = ∞):
    - *v* is added to the next frontier, set level
    - add *u* as a **parent**
  - If *v* is at the same level as *u*:
    - Add *u* as a **neighbor** to *PN[v]*
  - If *v* is at a different level:
    - If *u* is closer to the root (*L(u) < L(v)*), then *u* is a **parent**
- The **thresh_PN** limit
  - A parent or neighbor is *only* added if **Count[v] < thresh_PN**
  - Parents are prioritized over neighbors



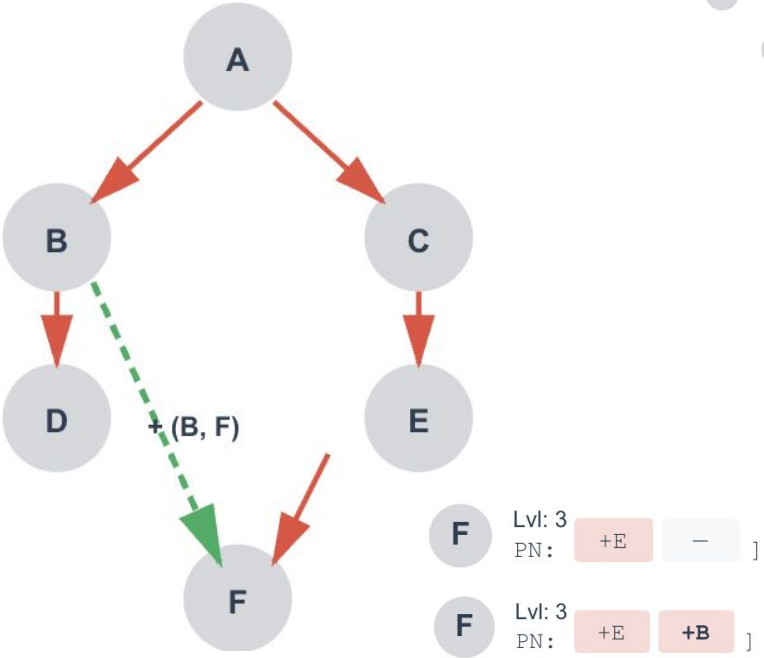
Algorithm 2: Edge Insertions

Case 1: Intra-Component Insertion

Insert (B, F)

1. Check ***C_id[B] = C_id[F]*** → intra-component update
2. Check from F's perspective: ***Count[F] < thresh_PN***
3. Check Levels: **Level[B] < Level[F]**
4. This means B is a *new parent* of F

PN[F] is updated from ***[+E,]*** to ***[+E, +B]***.



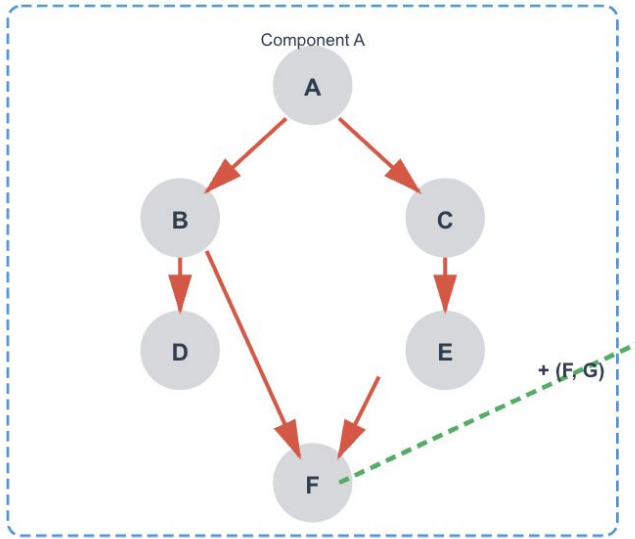
Algorithm 2: Edge Insertions

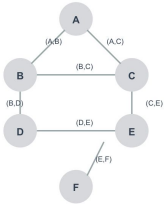
Case 2: Inter-Component Insertion

Insert (F, G)

1. Check ***C_id[F] (A) != C_id[G] (G)*** → inter-component merge
2. Check ***Size[G] == 1*** → singleton
3. Relabel G: ***C_id[G] = A***
4. Set G's level: ***Level[G] = Level[F] + 1***
5. Set G's parent: ***PN[G] = [+F,]***

If G was not a singleton, we would run a full parallel BFS on the smaller component to relabel it and rebuild its PN structure.





Node F:

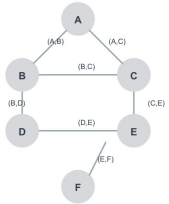
F
```
Level[F] = 3
C_id[F] = A
PN[F] = [+E, +B]
```

Node G (before):

G
```
Level[G] = 0
C_id[G] = G
PN[G] = [ , ]
Size[G] = 1
```

Node G (after):

G
```
Level[G] = 4   (Level[F] + 1)
C_id[G] = A    (relabeled)
PN[G] = [ +F  –  ]
```



Algorithm 3: Edge Deletions

Case 1: Safe Deletion (99.9% Case)

Delete (D, E)

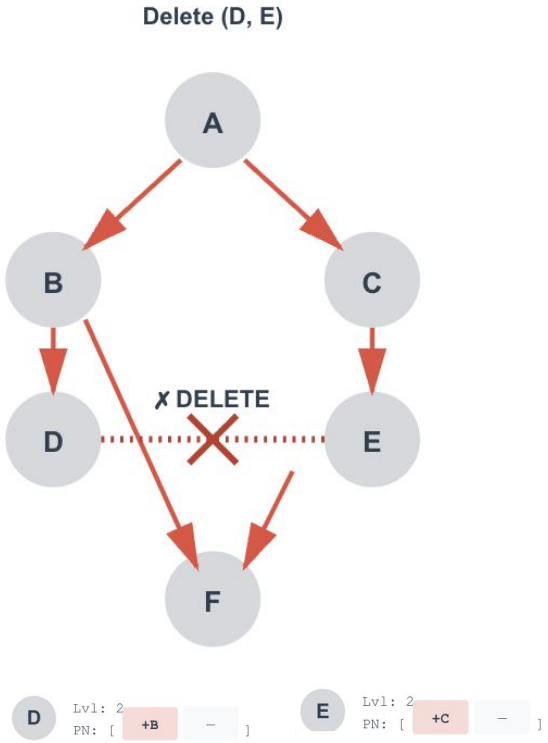
Logic (from D's side):

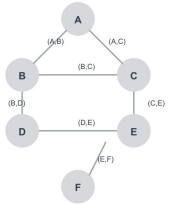
1. Check ***PN[D]***: Does it contain ***+E*** or ***-E***? Yes, -E.
2. Remove ***-E*** from ***PN[D]***
3. Scan ***PN[D]:*** Does ***D*** still have parents? ***hasParents*** flag.
4. Yes, ***+B*** is still in ***PN[D]***
5. ***hasParents*** is true. The deletion is **SAFE** from D's side.

Logic (from E's side):

1. Check ***PN[E]***: Remove ***-D***.
2. Scan ***PN[E]***: Does E still have parents?
3. Yes, ***+C*** is still in ***PN[E].***
4. ***hasParents*** is true. The deletion is **SAFE** from E's side.

Result: ***PN[D] = [+B,]***, ***PN[E] = [+C,]***. No repair needed.
This is an ***O(thresh_PN)*** check.





Algorithm 3: Edge Deletions

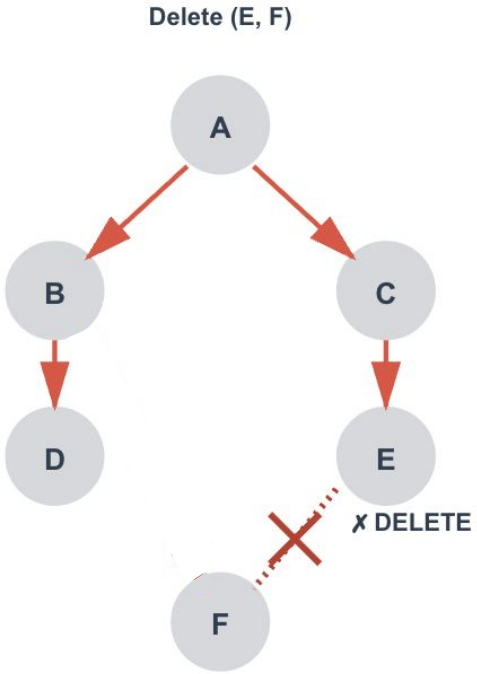
Case 2: Unsafe Deletion (Rare)

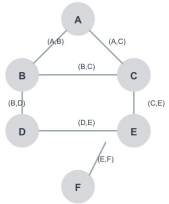
Delete (E, F)

Logic (from F's side):

1. Check ***PN[F]***: Remove ***+E***.
2. Scan ***PN[F]***: Does F still have parents? No. ***PN[F]*** is empty.
3. ***hasParents*** is false.
4. **CRITICAL STEP:** ***Level[F]*** is negated: ***Level[F] = -3***.

This tells F's neighbors "I am not a valid path to the root!"

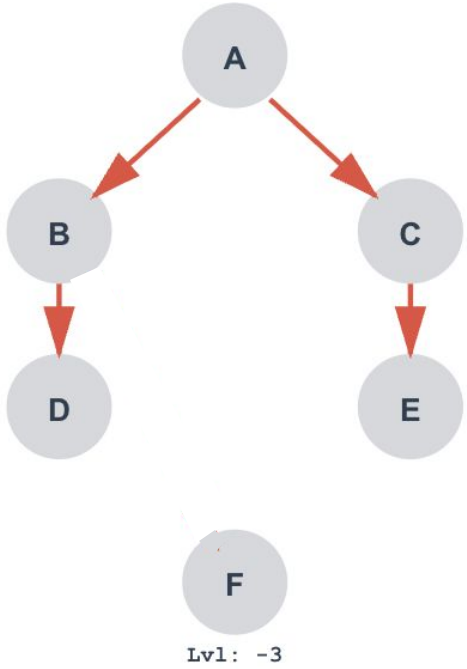




Algorithm 3: Edge Deletions

Unsafe Deletion (Safety Check)

1. The algorithm re-checks all deletions marked **unsafe** (like F's).
2. Does ***F*** have any parents (***p > 0***)? No.
3. Does ***F*** have any valid neighbors (***p < 0*** and ***Level[abs(p)] > 0***)? No.
4. The deletion ***(E,F)*** is confirmed UNSAFE



Result: We MUST call ***repairComponent(F)*** (Algorithm 4).

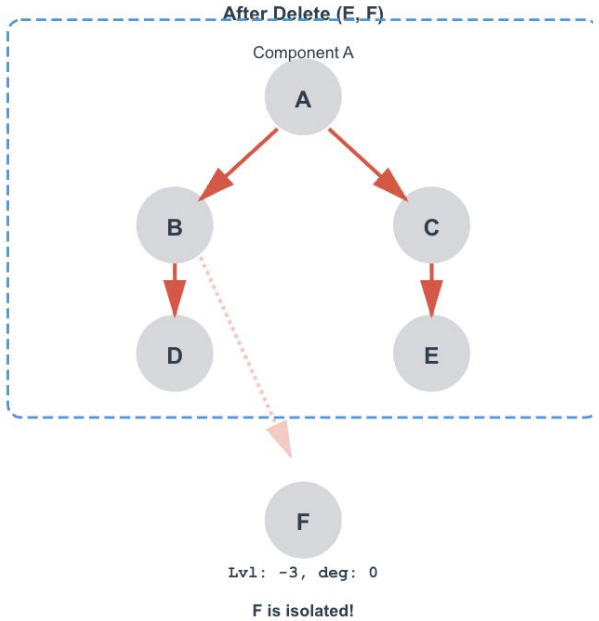
Algorithm 4: Repair

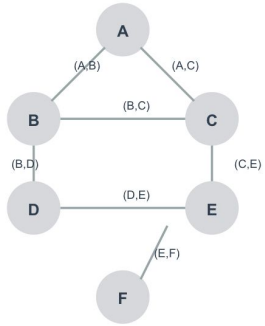
***repairComponent(F)* is called**

The algorithm must find out if ***F*** is truly disconnected or just needs a new path.

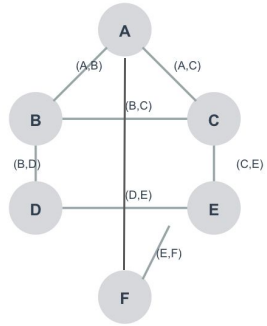
1. First, an optimization: Does ***F*** have any edges left in the *original graph*?
2. In our example, ***(E, F)*** was its only edge, so ***F*** has degree 0
3. The check ***{...} = ∅*** is ***TRUE***.

Result: ***F*** is a new singleton component. ***Level[F] = 0***; ***C_id[F] = F***; ***Size[F] = 1***

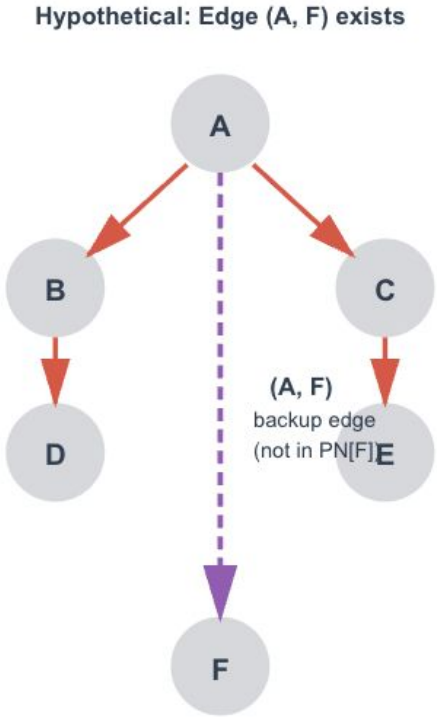




Algorithm 4: Repair



What if F had a backup edge?
Imagine we also had edge ***(A, F)*** (but it wasn't in ***PN[F]*** because ***thresh_PN*** was full).



1. The "degree 0" check is ***FALSE***.
2. ***repairComponent(F)*** runs.
3. It starts a new parallel BFS from F, setting ***C_id[F] = F*** and ***Level[F] = 0.***
4. This new BFS explores ***(F, A).***
5. It finds vertex ***A***, which is in the original component (***C_id[A] == C_id[s]***, where s is the original root).

Reconnection found. The disconnected flag is set to false. A second BFS is triggered to walk back from the connection point (***A***) and relabel ***F*** (and any other vertices that were explored) back to component ***A***, rebuilding their ***PN*** arrays along the way.

Result: The component was not split. The PN sub-graph is repaired, and ***F*** is correctly reconnected.

Experimental Results: Finding ***thresh_PN***

- Test setup
- What is the optimal ***thresh_PN***?
  - Balances memory ($O(V * k)$) vs. repair work (unsafe deletes)
  - ***thresh_PN*** = 1: Ineffective. Every deletion from the PN-subgraph is marked unsafe
  - ***thresh_PN*** = 2: Still very high (over 2,000 unsafe deletes)
  - ***thresh_PN*** = 4: Number of unsafe deletes is tiny
  - ***thresh_PN*** > 4: 8 or 12 gives significantly diminished returns
- ***thresh_PN*** = 4 is the sweet spot

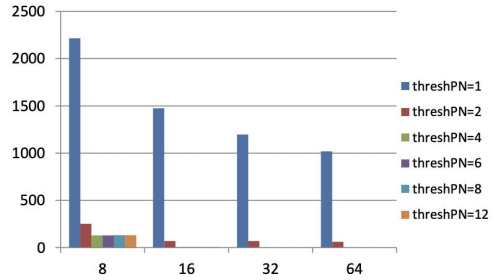


Figure 2. Average number of unsafe deletes in PN data structure for batches of $100K$ updates as a function of the average degree (x-axis) and $thresh_{PN}$ (bars).

Experimental Results: Performance vs. Static

- Parallel static recomputation using the **Shiloach-Vishkin** algorithm as a baseline
- The dynamic PN algorithm is massively faster
  - The speedup gets better as the graph gets denser.
  - **Avg. Degree 8:** 1.8x faster than S-V
  - **Avg. Degree 32:** 30.8x faster than S-V
  - **Maximum:** 48.3x speedup was observed in this test

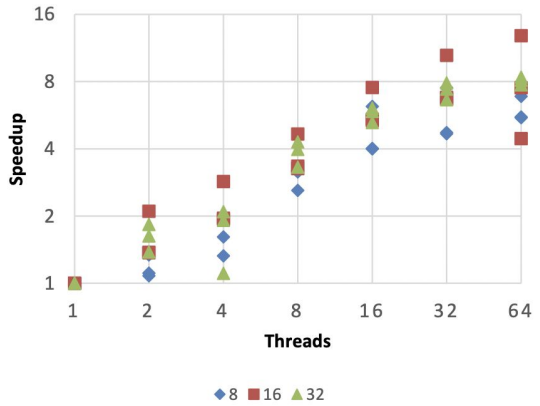


Figure 4. Speed up of the new algorithm over performing parallel static recomputation after each batch on three different RMAT-22 graphs with each average degree as a function of the number of threads.

Experimental Results: Performance vs. Static

- Parallel static recomputation using the **Shiloach-Vishkin** algorithm as a baseline
- Speedup on the largest graph (Scale 24, 16M vertices)
- Trend continues, denser graphs give better results
- On one Scale 24, edge factor 16 graph, the algorithm achieved a **1372x speedup** over S-V

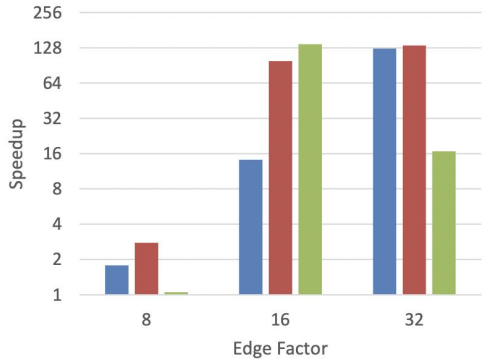


Figure 6. Speed up over performing static recomputation after each batch on scale 24 graphs for three graphs at each edge factor using 64 threads.

Experimental Results: Parallel Scaling

- If I hold the graph size constant and add more cores, does it get faster?
- Nearly linear but not optimal scaling up to 32 threads.
- **32 threads:** 10.5x speedup
- **64 threads:** 12.8x speedup
- Flattens out after ~32 cores
- Authors say this is because of only fine-grain parallelism like the repair step
- The speedup ratio between the PN algorithm and the static SV algorithm remains constant as threads increase.
- Figure 5 also shows the overhead of the PN algorithm remains a constant fraction of the total work
- The PN algorithm scales exactly as well as the underlying STINGER data structure. It doesn't introduce any new scaling bottlenecks.

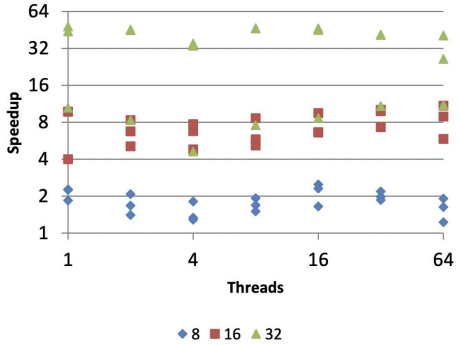


Figure 3. Strong scaling results on RMAT-22 graphs with different average degree as a function of the number of threads. Results include three graphs at each average degree.

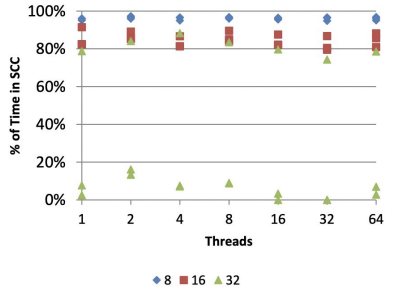


Figure 5. Fraction of the update time spent updating connected components over time spent updating the graph structure and connected components.

Results vs. Related Work

Approach	Storage	Correctness	Performance
Static (Shiloach-Vishkin)	$O(V + E)$	100%	Baseline (very slow)
Theoretical (Henzinger/King)	$O(V + E)$	100%	-
Heuristic (1 Spanning Tree)	$O(V)$	90%	Fast, but incorrect
Heuristic (2 Spanning Trees)	$O(V)$	99.7%	Better, but still incorrect and demanding
PN Sub-graph	$O(V)$	100%	Up to 1372x static

Strengths

- Low memory footprint is clever: *O(V)* extra storage is the main contribution of the paper. It makes it practical for massive graphs where *O(V + E)* isn't it
- Massive performance gain without big trade-offs (up to 1372x faster than static recomputation)
- 100% correctness is good. The "unsafe" check is a 100% true-positive test, with a low, manageable false-negative rate.
- Pragmatic foundations (Built on STINGER which is already high-performance and practically implemented)

& Weaknesses

- Only synthetic data was used. The evaluation is *only* on R-MAT synthetic graphs. This means, it could be *ideal* for this algorithm. Performance on real-world, messier data is unproven
- Scaling could be improved. It is not linear and flattens at 32 cores. This implies that some parts (likely the repair/A4) are serial
- The algorithm requires parameter tuning and the ***thresh_PN = 4*** was derived *from* the R-MAT data. It's not a universal constant. A different graph structure might require a different/unknown threshold

Future Directions

- The paper stated that it would investigate a more advanced BFS algorithm (like Beamer et al.) for the initialization and repair steps to improve traversal performance
- Why is the threshold global? If we assigned a ***thresh_PN*** per-vertex, based on its centrality or other properties, would that be more efficient?
- Why are we stalling for the repair? Is there any way to make it asynchronous and trade 100% correctness for *eventual* consistency and higher throughput?
- How could this PN-graph scaffolding be applied to other graph problems?

& Discussion Questions

- ***thresh_PN*** was found empirically on R-MAT graphs. How would we validate this parameter before analyzing a real-world social network?
- This paper is from 2013. Today, we have massive GNNs and hardware acceleration. If we were solving this problem today, would we still use this explicit scaffolding approach?
- Can the connectivity scaffold principle be applied to designing dynamic algorithms for SSSP?