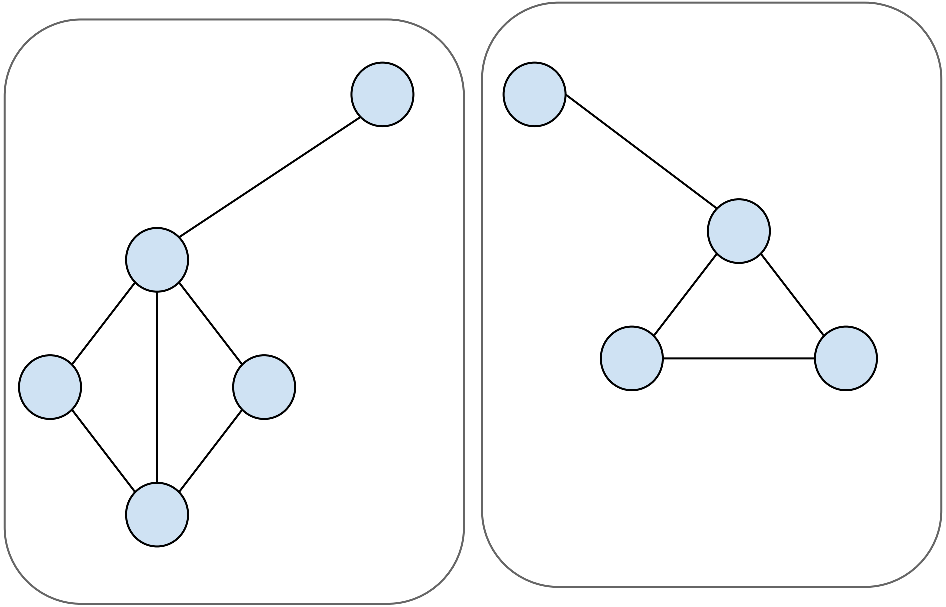
A New Parallel Algorithm for Connected Components in Dynamic Graphs

Authored by: Robert McColl, Oded Green,
David A. Bader
Presented by: Omar Obeya

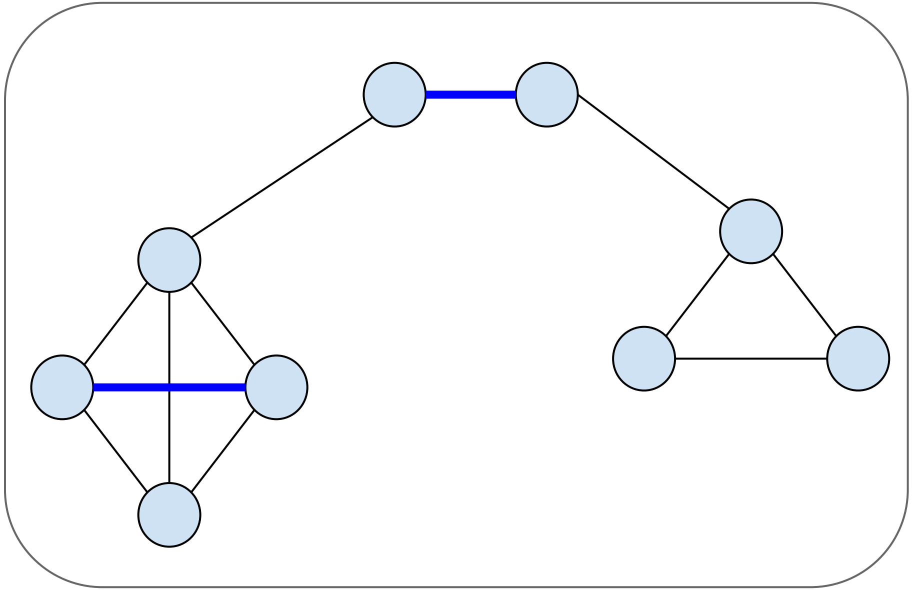
Connected Components Problem



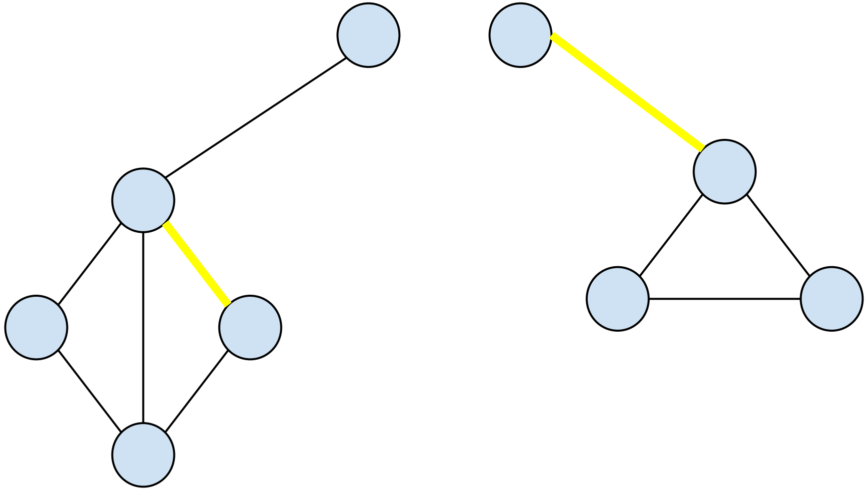
Dynamic Connected Components Problem

- Operations
  - Query node
  - Insert Edge
  - Delete Edge

Connected Components Problem



Connected Components Problem



Goals

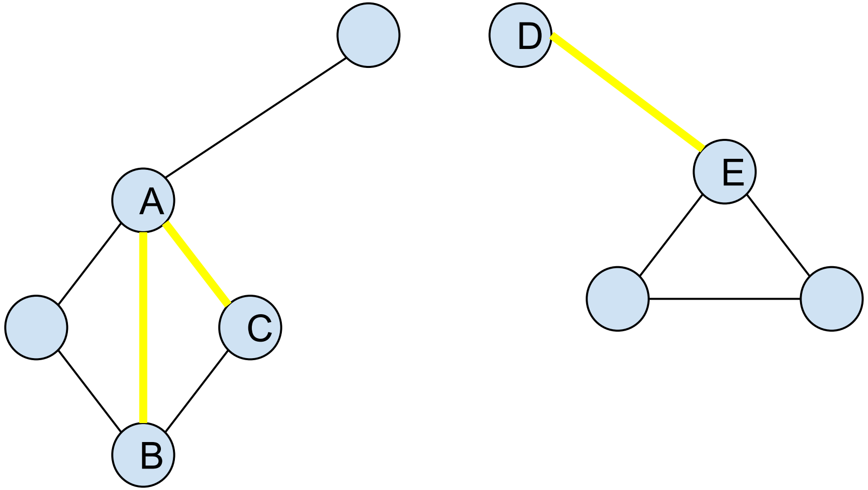
- Parallel
- Exact
- Dynamic

Reformatting the problem

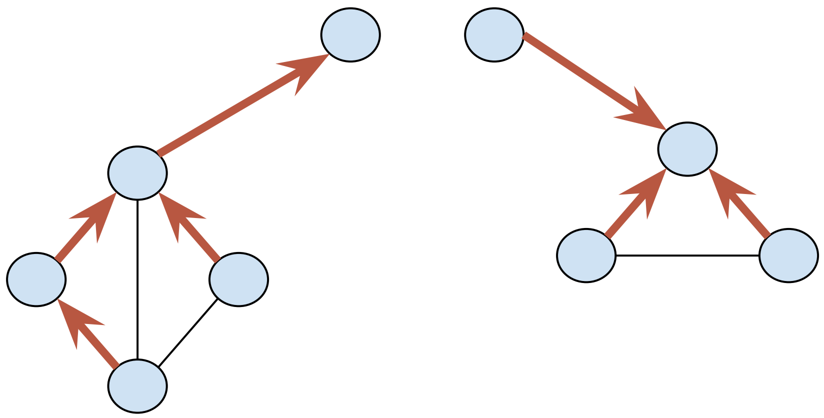
Given a deletion classify whether it is safe

- 100% True positive.
- Minimum false negative.

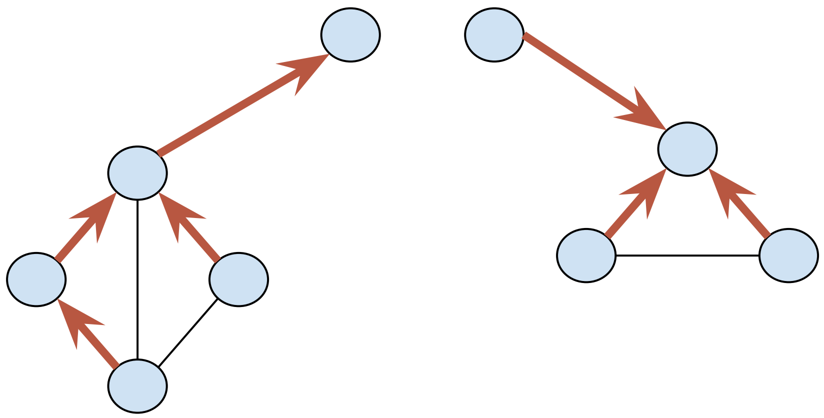
Approach I: Adjacency List Intersection



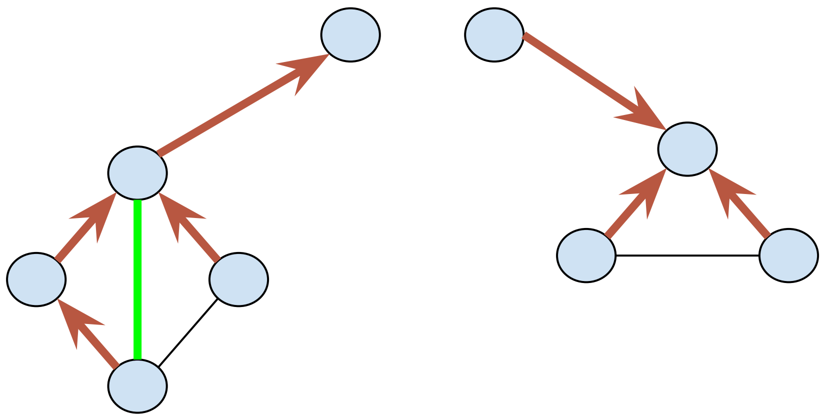
Approach II: Spanning Tree



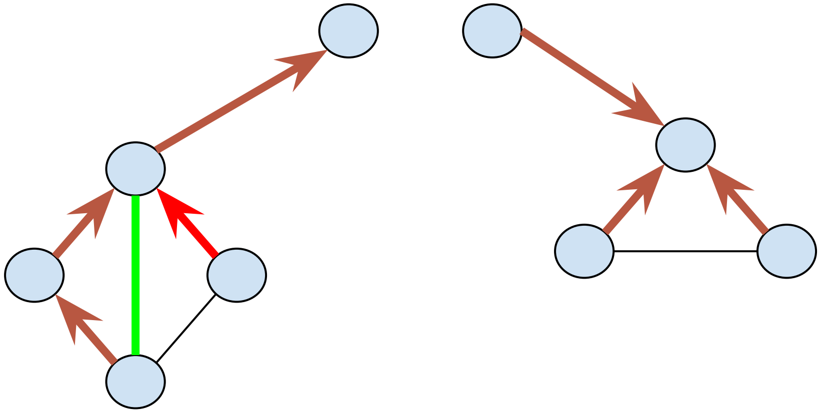
Approach II: Spanning Tree



Approach II: Spanning Tree



Approach II: Spanning Tree



Solution

- Maintain Neighbor-Parent list of ***ThreshPN*** “neighpars” only.
- At Deletion:
  - if it has a parent, or have a neighbour that has a parent.
  - Otherwise: recalculate.
- Use STINGER

Definitions

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	$O(1)$
$\bar{E}_I$	Batch of edges to be inserted into graph	array	$O(batch\ size)$
$\bar{E}_R$	Batch of edges to be deleted from graph	array	$O(batch\ size)$

Initialization

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

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

for $v \in V$ **do**
    $Level[v] \leftarrow \infty, Count[v] \leftarrow 0$
for $v \in V$ **do**
    if $Level[v] = \infty$ **then**
        $Q[0] \leftarrow v, Q_{start} \leftarrow 0, Q_{end} \leftarrow 1$
        $Level[v] \leftarrow 0, C_{id}[v] \leftarrow v$
        while $Q_{start} \neq Q_{end}$ **do**
            $Q_{stop} \leftarrow Q_{end}$
            for $i \leftarrow Q_{start}$ **to** Q_{stop} **in parallel do**
                for each neighbor d **of** $Q[i]$ **do**
                    if $Level[d] = \infty$ **then**
                        $Q[Q_{end}] \leftarrow d$
                        $Q_{end} \leftarrow Q_{end} + 1$
                        $Level[d] \leftarrow Level[Q[i]] + 1$
                        $C_{id}[d] \leftarrow C_{id}[Q[i]]$
                    if $Count[d] < thresh_{PN}$ **then**
                        if $Level[Q[i]] < Level[d]$ **then**
                            $PN_d[Count[d]] \leftarrow Q[i]$
                            $Count[d] \leftarrow Count[d] + 1$
                        else if $Level[Q[i]] = Level[d]$ **then**
                            $PN_d[Count[d]] \leftarrow -Q[i]$
                            $Count[d] \leftarrow Count[d] + 1$
            $Q_{start} \leftarrow Q_{stop}$
        $Size[v] \leftarrow Q_{end}$

- Parallel BFS
- Put neighpars in your list if there is room.
- Use -ve values for neighbours and +ve values for parents.

Insertion

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, $Count$

for all $\langle s, d\rangle \in \tilde{E}_I$ **in parallel do** $E \leftarrow E \cup \langle s, d\rangle$
    insert$(E, \langle s, d\rangle)$
    if $C_{id}[s] = C_{id}[d]$ **then**
        if $Level[s] > 0$ **then**
            if $Level[d] < 0$ **then**
                // d is not "safe"
                if $Level[s] < -Level[d]$ **then**
                    if $Count[d] < thresh_{PN}$ **then**
                        $PN_d[Count[d]] \leftarrow s$
                        $Count[d] \leftarrow Count[d] + 1$
                    else
                        $PN_d[0] \leftarrow s$
                    $Level[d] \leftarrow -Level[d]$
            else
                if $Count[d] < thresh_{PN}$ **then**
                    if $Level[s] < Level[d]$ **then**
                        $PN_d[Count[d]] \leftarrow s$
                        $Count[d] \leftarrow Count[d] + 1$
                    else if $Level[s] = Level[d]$ **then**
                        $PN_d[Count[d]] \leftarrow -s$
                        $Count[d] \leftarrow Count[d] + 1$
                else if $Level[s] < Level[d]$ **then**
                    for $i \leftarrow 0$ **to** $thresh_{PN}$ **do**
                        if $PN_d[i] < 0$ **then**
                            $PNV_d[i] \leftarrow s,$
                            Break for-loop
        $\tilde{E}_I \leftarrow \tilde{E}_I \backslash \langle s, d\rangle$
for all $\langle s, d\rangle \in \tilde{E}_I$ **do**
    if $C_{id}[s] \neq C_{id}[d]$ **then**
        if $Size[s] = 1$ **then**
            $Size[s] \leftarrow 0$
            $Size[d] \leftarrow Size[d] + 1$
            $C_{id}[s] \leftarrow C_{id}[d]$, $PN_s[0] \leftarrow d$
            $Level[s] \leftarrow$ **abs**$(Level[d]) + 1$, $Count[s] \leftarrow 1$
        else
            connectComponent(Input, s, d)

- Intra-connecting edges
  - In Parallel
  - If don't have a parent, add s if it was a parent.
  - If have a parent.
    - Try add s if it was neighbour
    - Must add s if it was parent (unless list is full of parents).

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, $Count$

```
for all⟨s, d⟩ ∈ Ẽ_I in parallel do E ← E ∪ ⟨s, d⟩
    insert(E, ⟨s, d⟩)
    if C_id[s] = C_id[d] then
        if Level[s] > 0 then
            if Level[d] < 0 then
                // d is not "safe"
                if Level[s] < −Level[d] then
                    if Count[d] < thresh_PN then
                        PN_d[Count[d]] ← s
                        Count[d] ← Count[d] + 1
                    else
                        PN_d[0] ← s
                    Level[d] ← −Level[d]
            else
                if Count[d] < thresh_PN then
                    if Level[s] < Level[d] then
                        PN_d[Count[d]] ← s
                        Count[d] ← Count[d] + 1
                    else if Level[s] = Level[d] then
                        PN_d[Count[d]] ← −s
                        Count[d] ← Count[d] + 1
                else if Level[s] < Level[d] then
                    for i ← 0 to thresh_PN do
                        if PN_d[i] < 0 then
                            PNV_d[i] ← s,
                            Break for-loop
        Ẽ_I ← Ẽ_I \ ⟨s, d⟩
for all⟨s, d⟩ ∈ Ẽ_I do
    if C_id[s] ≠ C_id[d] then
        if Size[s] = 1 then
            Size[s] ← 0
            Size[d] ← Size[d] + 1
            C_id[s] ← C_id[d], PN_s[0] ← d
            Level[s] ← abs(Level[d]) + 1, Count[s] ← 1
        else
            connectComponent(Input, s, d)
```

- Inter-connecting edges
    - In Series
    - Special handle singletons.
    - Merge the labeling of the smaller component with the bigger.

Deletion

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

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

for all $\langle s, d\rangle \in \hat{E}_R$ **in parallel do**
  $E \leftarrow E \backslash \langle s, d\rangle$
  $hasParents \leftarrow$ false
  for $p \leftarrow 0$ **to** $Count[d]$ **do**
    if $PN_d[p] = s$ **or** $PN_d[p] = -s$ **then**
      $Count[d] \leftarrow Count[d] - 1$
      $PN_d[p] \leftarrow PN_d[Count[d]]$
    if $PN_d[p] > 0$ **then**
      $hasParents \leftarrow$ true
  if (**not** $hasParents$) **and** $Level[d] > 0$ **then**
    $Level[d] \leftarrow -Level[d]$
for all $\langle s, d\rangle \in \hat{E}_R$ **in parallel do**
  for all $p \in PN_d$ **do**
    if $p \geq 0$ **or** $Level[\text{abs(p)}] > 0$ **then**
      $\hat{E}_R \leftarrow \hat{E}_R \backslash \langle s, d\rangle$
$PREV \leftarrow C_{id}$
for all $\langle s, d\rangle \in \hat{E}_R$ **do**
  $unsafe \leftarrow (C_{id}[s] = C_{id}[d] = PREV_s)$
  for all $p \in PN_d$ **do**
    if $p \geq 0$ **or** $Level[\text{abs(p)}] > 0$ **then**
      $unsafe \leftarrow$ false
  if $unsafe$ **then**
    if $\{\langle u, v\rangle \in G(E, V) : u = s\} = \emptyset$ **then**
      $Level[s] \leftarrow 0, C_{id}[s] \leftarrow s$
      $Size[s] \leftarrow 1, Count[s] \leftarrow 0$
    else
      Algorithm 4
      repairComponent(Input, s, d)

- Maintain Data structure
  - In Parallel
    - Remove S from list
    - Recalc Has Parent?
    - Mark safe if possible
  - In Series
    - Mark safe if possible
    - Otherwise repair

Repair

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

```
Input: G(V, E), Ẽ_R, C_id, Size, Level, PN, Count, s, d
Output: C_id, Size, Level, PN, Count
Q[0] ← d, Q_start ← 0, Q_end ← 1
SLQ ← ∅, SLQ_start ← 0, SLQ_end ← 0
Level[d] ← 0, C_id[d] ← d
disconnected ← true
while Q_start ≠ Q_end do
    Q_stop ← Q_end
    for i ← Q_start to Q_stop in parallel do
        u ← Q[i]
        for each neighbor v of u do
            if C_id[v] = C_id[s] then
                if Level[v] ≤ abs(Level[d]) then
                    C_id[v] ← C_id[d]
                    disconnected ← false
                    SLQ[SLQ_end] ← v
                    SLQ_end ← SLQ_end + 1
                else
                    C_id[v] ← C_id[d]
                    Count[v] ← 0
                    Level[v] ← Level[u] + 1
                    Q[Q_end] ← v
                    Q_end ← Q_end + 1
            if Count[v] < thresh_PN then
                if Level[u] < Level[v] then
                    PN_v[Count[v]] ← u
                    Count[v] ← Count[v] + 1
                else if Level[v] = Level[v] then
                    PN_v[Count[v]] ← −u
                    Count[v] ← Count[v] + 1
    Q_start ← Q_stop
if disconnected then
    Size[d] ← Q_end
else
    for i ← SLQ_start to SLQ_end in parallel do
        C_id[i] ← C_id[s]
    while SLQ_start ≠ SLQ_end do
        SLQ_stop ← SLQ_end
        for i ← SLQ_start to SLQ_stop in parallel do
            u ← SLQ[i]
            for each neighbor v of u do
                if C_id[v] = C_id[d] then
                    C_id[v] ← C_id[u]
                    Count[v] ← 0
                    Level[v] ← Level[u] + 1
                    SLQ[SLQ_end] ← v
                    SLQ_end ← SLQ_end + 1
                if Count[v] < thresh_PN then
                    if Level[u] < Level[v] then
                        PN_v[Count[v]] ← u
                        Count[v] ← Count[v] + 1
                    else if Level[v] = Level[v] then
                        PN_v[Count[v]] ← −u
                        Count[v] ← Count[v] + 1
        Q_start ← Q_stop
```

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

Output: C_{id}, $Size$, $Level$, PN, $Count$

$Q[0] \leftarrow d$, $Q_{start} \leftarrow 0$, $Q_{end} \leftarrow 1$
$SLQ \leftarrow \emptyset$, $SLQ_{start} \leftarrow 0$, $SLQ_{end} \leftarrow 0$
$Level[d] \leftarrow 0$, $C_{id}[d] \leftarrow d$
$disconnected \leftarrow$ true
while $Q_{start} \neq Q_{end}$ **do**
    $Q_{stop} \leftarrow Q_{end}$
    for $i \leftarrow Q_{start}$ **to** Q_{stop} **in parallel do**
        $u \leftarrow Q[i]$
        for each neighbor v **of** u **do**
            if $C_{id}[v] = C_{id}[s]$ **then**
                if $Level[v] \leq$ abs$(Level[d])$ **then**
                    $C_{id}[v] \leftarrow C_{id}[d]$
                    $disconnected \leftarrow$ false
                    $SLQ[SLQ_{end}] \leftarrow v$
                    $SLQ_{end} \leftarrow SLQ_{end} + 1$
                else
                    $C_{id}[v] \leftarrow C_{id}[d]$
                    $Count[v] \leftarrow 0$
                    $Level[v] \leftarrow Level[u] + 1$
                    $Q[Q_{end}] \leftarrow v$
                    $Q_{end} \leftarrow Q_{end} + 1$
            if $Count[v] < thresh_{PN}$ **then**
                if $Level[u] < Level[v]$ **then**
                    $PN_v[Count[v]] \leftarrow u$
                    $Count[v] \leftarrow Count[v] + 1$
                else if $Level[v] = Level[v]$ **then**
                    $PN_v[Count[v]] \leftarrow -u$
                    $Count[v] \leftarrow Count[v] + 1$
    $Q_{start} \leftarrow Q_{stop}$

- Parallel BFS
    - Search for a connection back to the s component

```
if disconnected then
    Size[d] ← Q_end
else
    for i ← SLQ_start to SLQ_end in parallel do
        C_id[i] ← C_id[s]
    while SLQ_start ≠ SLQ_end do
        SLQ_stop ← SLQ_end
        for i ← SLQ_start to SLQ_stop in parallel do
            u ← SLQ[i]
            for each neighbor v of u do
                if C_id[v] = C_id[d] then
                    C_id[v] ← C_id[u]
                    Count[v] ← 0
                    Level[v] ← Level[u] + 1
                    SLQ[SLQ_end] ← v
                    SLQ_end ← SLQ_end + 1
                if Count[v] < thresh_PN then
                    if Level[u] < Level[v] then
                        PN_v[Count[v]] ← u
                        Count[v] ← Count[v] + 1
                    else if Level[v] = Level[v] then
                        PN_v[Count[v]] ← −u
                        Count[v] ← Count[v] + 1
        Q_start ← Q_stop
```

- Disconnected -> done.
- Relabel all vertices discovered.
- Second Parallel BFS
    - Relabel more undiscovered vertices.

Quantitative Results

Unsafe Deletions

As we store more neighbors we get lower false negative

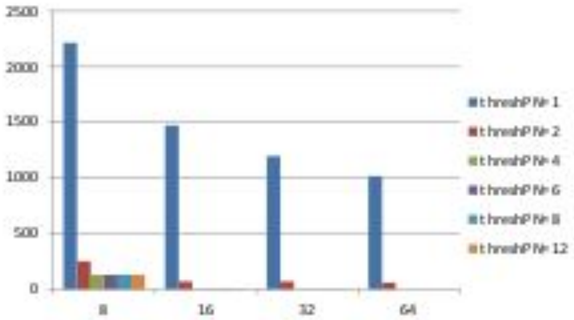


Figure 2. Average number of unsafe deletes in PN data structure for batches of $100K$ updates as a function of the average degree (x-axis) and $thresh_{PN}$ (bars).

Performance Results

Scaling

Scales sublinearly.

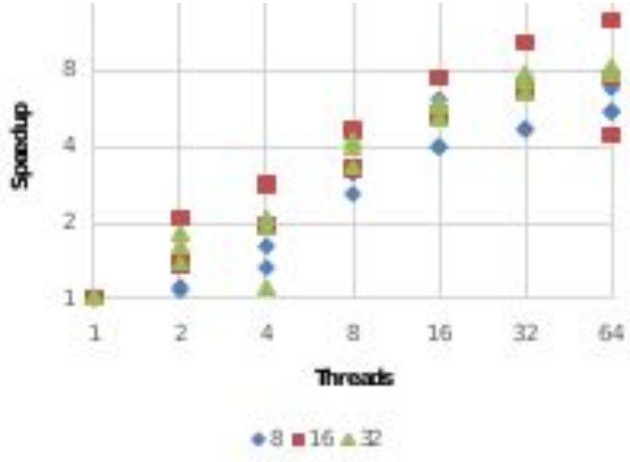


Figure 3. Strong scaling results on RMAT-22 graphs with different average degree as a function of the number of threads. Results include three graphs

Speed up

Speed up (over static) depends on graph density and not no. of threads.

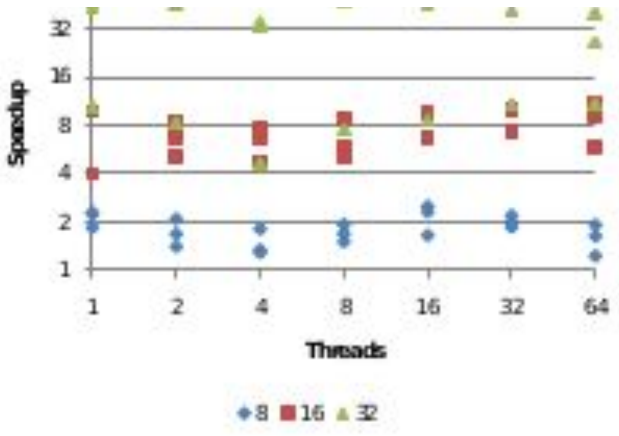


Figure 4. Speed up of the new algorithm over performing parallel static recomputation after each batch on three different RMAT-22 graphs with each average degree as a function of the number of threads.

Graph Dependency

The algorithm speedup is very graph dependent

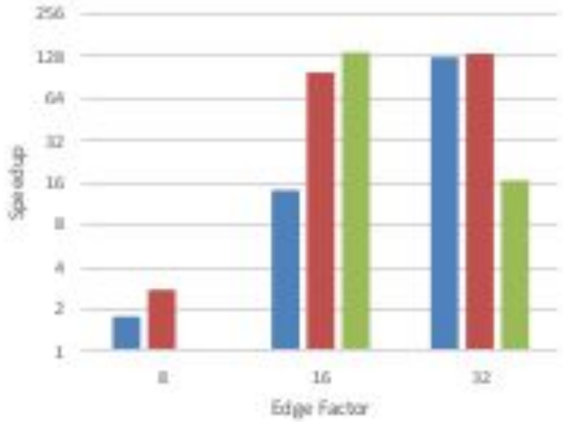


Figure 6. Speed up over performing static recomputation after each batch on scale 24 graphs for three graphs at each edge factor using 64 threads.

Weak Points

- Percentage of deletion is 6.25%
- Graph Dependent
- No probabilistic theoretical bounds

Future Work

- Threshold can be a function of no. of edges.
- What about adjacency matrix intersection using a popular node?

References

- McColl, Robert, Oded Green, and David A. Bader. "A new parallel algorithm for connected components in dynamic graphs." In High Performance Computing (HiPC), 2013 20th International Conference on, pp. 246-255. IEEE, 2013.