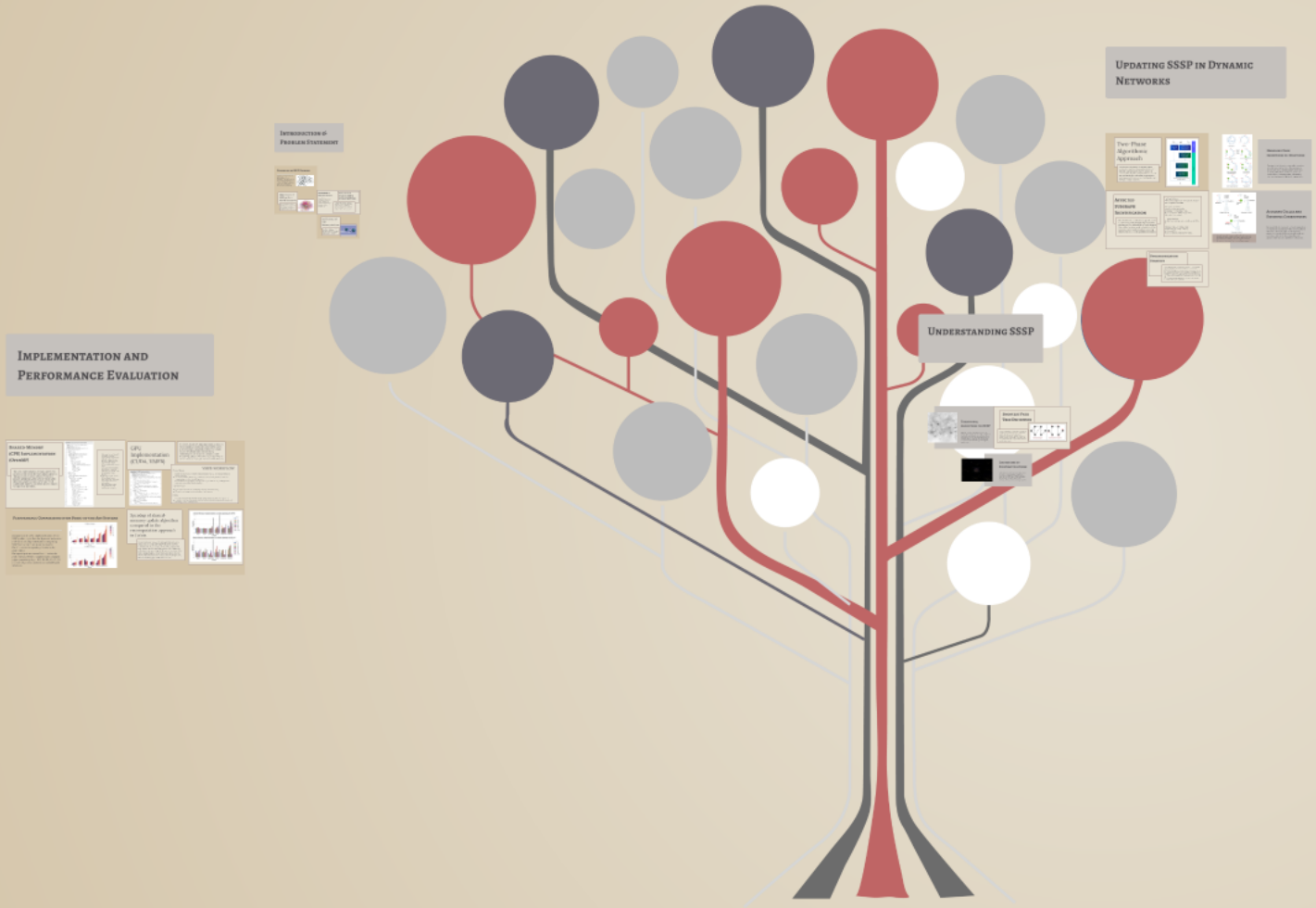


A Parallel Algorithm for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks

By Fakhir Ali 22I-0762 &
Ayna Sulaiman 22I-1054

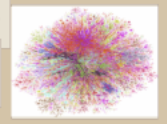
Introduction & Problem Statement

Definition of SSSP Problem

The Single Source Shortest Path (SSSP) problem seeks to determine the shortest paths from a single source vertex to all other vertices in a graph. This problem is categorized under graph theory and is crucial for navigating network structures efficiently.

Importance of SSSP in Real-World Networks

SSSP plays a pivotal role in various real-world applications such as transportation routes, communications, and social networks. By enabling efficient routing and accessibility, it enhances decision-making across sectors.



Definition of Dynamic Graphs

Dynamic graphs are characterized by the addition or removal of edges/nodes over time, reflecting real-world changes. Examples include social networks with fluctuating friendships and traffic networks where road conditions continuously vary.

Challenges in Updating SSSP in Dynamic Networks

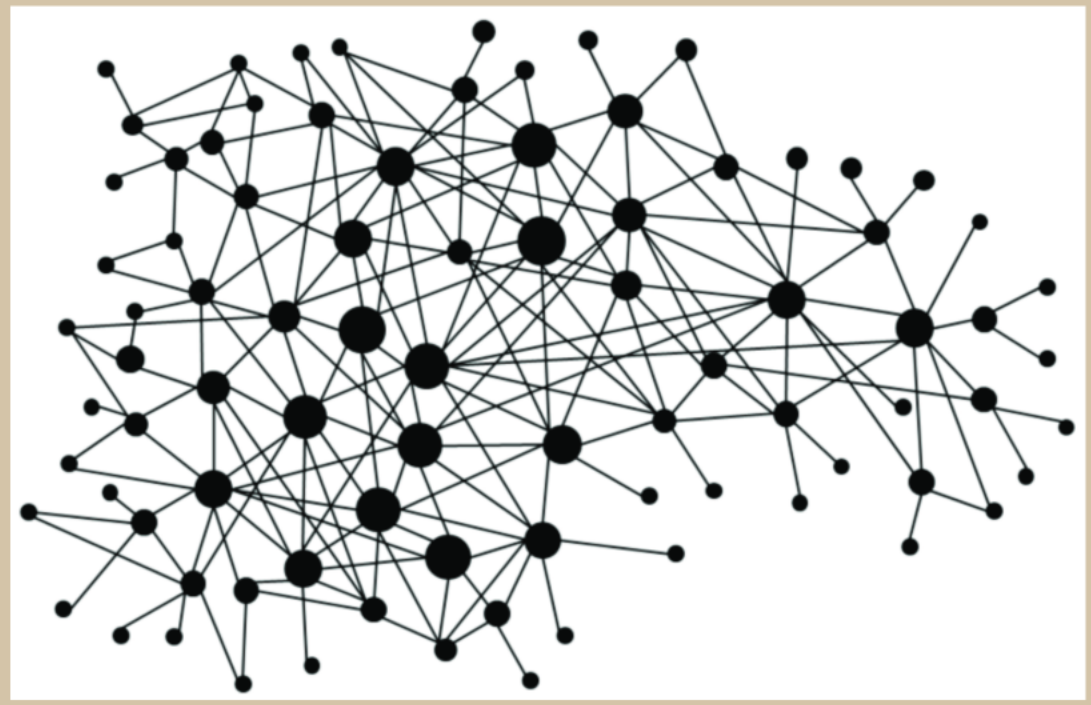
Dynamic networks often experience frequent changes in edges and nodes, complicating the SSSP updating process. These alterations necessitate efficient algorithms that can manage updates without full recalculation.

Inefficiency of Full Recomputations

Recomputing the SSSP from scratch after each change in the network is computationally expensive and inefficient. This inefficiency underscores the need for innovative updating algorithms that can minimize redundant calculations.

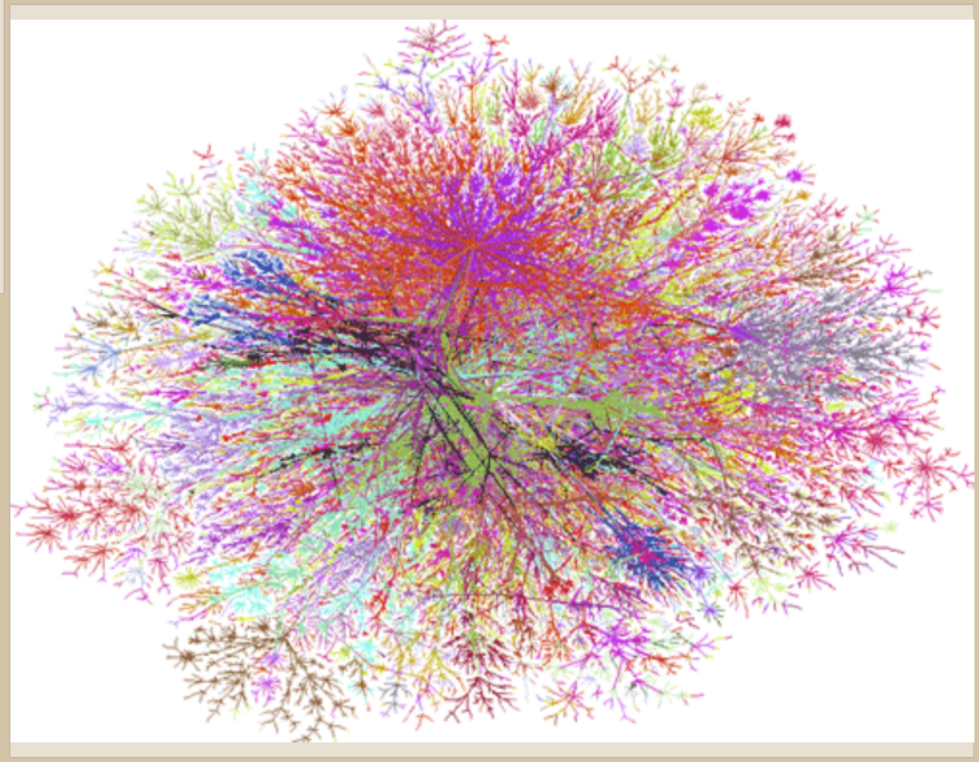
Definition of SSSP Problem

The Single Source Shortest Path (SSSP) problem seeks to determine the shortest paths from a single source vertex to all other vertices in a graph. This problem is categorized under graph theory and is crucial for navigating network structures efficiently.



Importance of SSSP in Real-World Networks

SSSP plays a pivotal role in various real-world applications such as transportation routes, communications, and social networks. By enabling efficient routing and accessibility, it enhances decision-making across sectors.



Definition of Dynamic Graphs

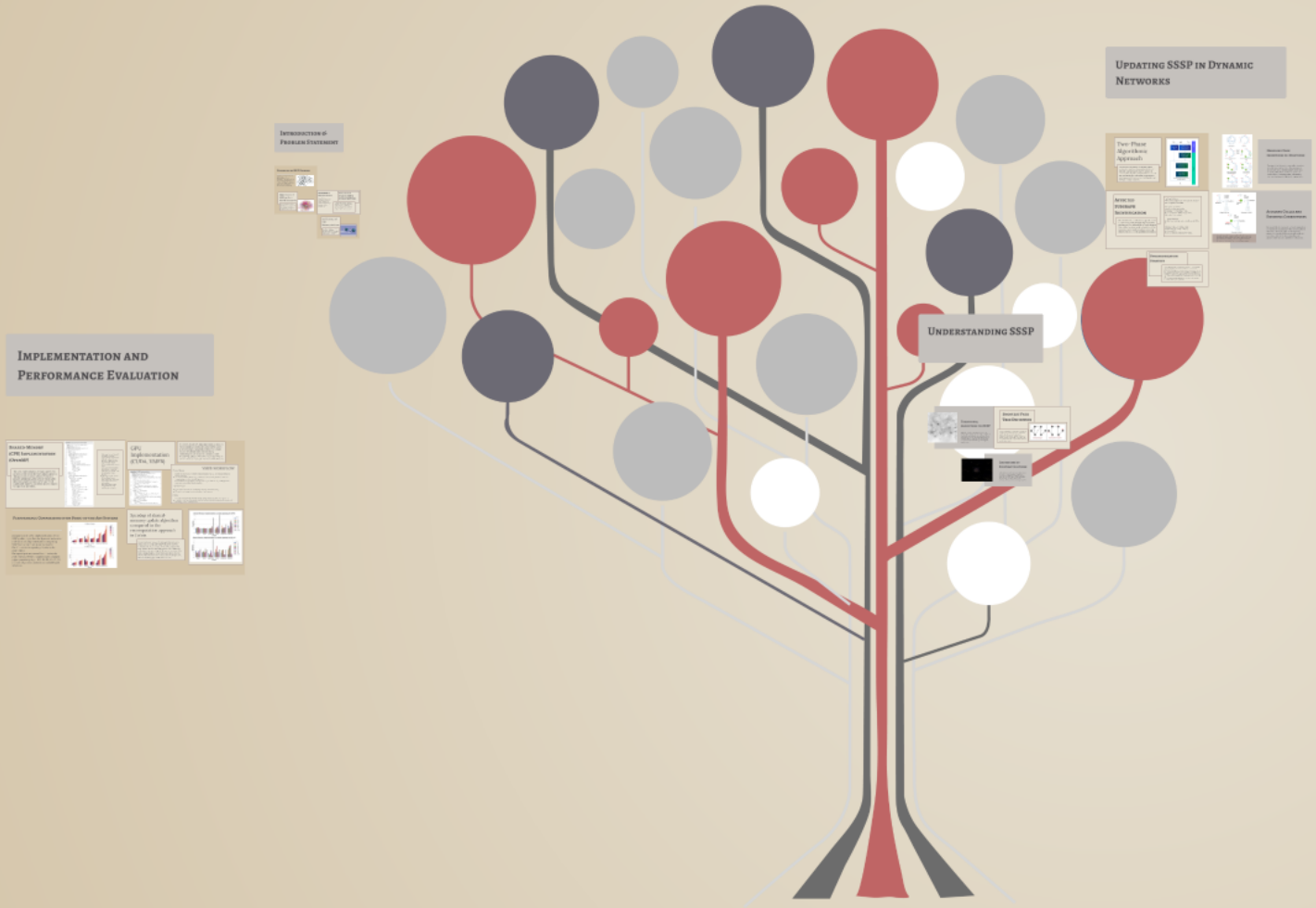
Dynamic graphs are characterized by the addition or removal of edges/nodes over time, reflecting real-world changes. Examples include social networks with fluctuating friendships and traffic networks where road conditions continuously vary.

Challenges in Updating SSSP in Dynamic Networks

Dynamic networks often experience frequent changes in edges and nodes, complicating the SSSP updating process. These alterations necessitate efficient algorithms that can manage updates without full recalculation.

Inefficiency of Full Recomputations

Recomputing the SSSP from scratch after each change in the network is computationally expensive and inefficient. This inefficiency underscores the need for innovative updating algorithms that can minimize redundant calculations.



A Parallel Algorithm for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks

By Fakhir Ali 22I-0762 &
Ayna Sulaiman 22I-1054

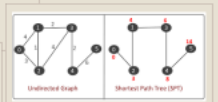
Understanding SSSP

Traditional Algorithms for SSSP

Dijkstra's algorithm is a well-known method for solving SSSP in graphs with non-negative weights. It operates by iteratively selecting the vertex with the minimum path cost and updating adjacent vertices, providing efficient results for static graphs but is less effective in dynamic settings where frequent updates occur.

Shortest Path Tree Definition

A shortest path tree is a spanning tree rooted at the source vertex that represents the shortest paths to all other vertices in the graph. Maintenance of this tree requires continuous updates as edges are added or removed, which can significantly strain computational efficiency in dynamic networks.



Limitations of Existing Solutions

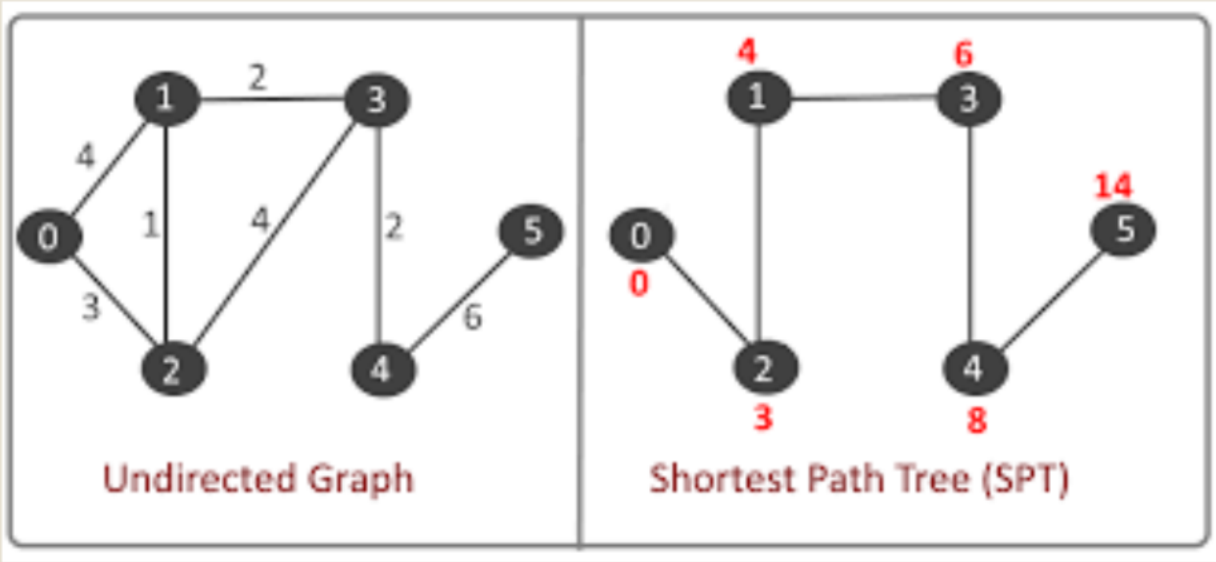
Existing solutions like Dijkstra's and Galois operate efficiently on static graphs but struggle with scalability during updates in dynamic networks. They often require complete recomputation after minor changes, leading to inefficiencies in scenarios where edges or nodes change frequently.

Traditional Algorithms for SSSP

Dijkstra's algorithm is a well-known method for solving SSSP in graphs with non-negative weights. It operates by iteratively selecting the vertex with the minimum path cost and updating adjacent vertices, providing efficient results for static graphs but is less effective in dynamic settings where frequent updates occur.

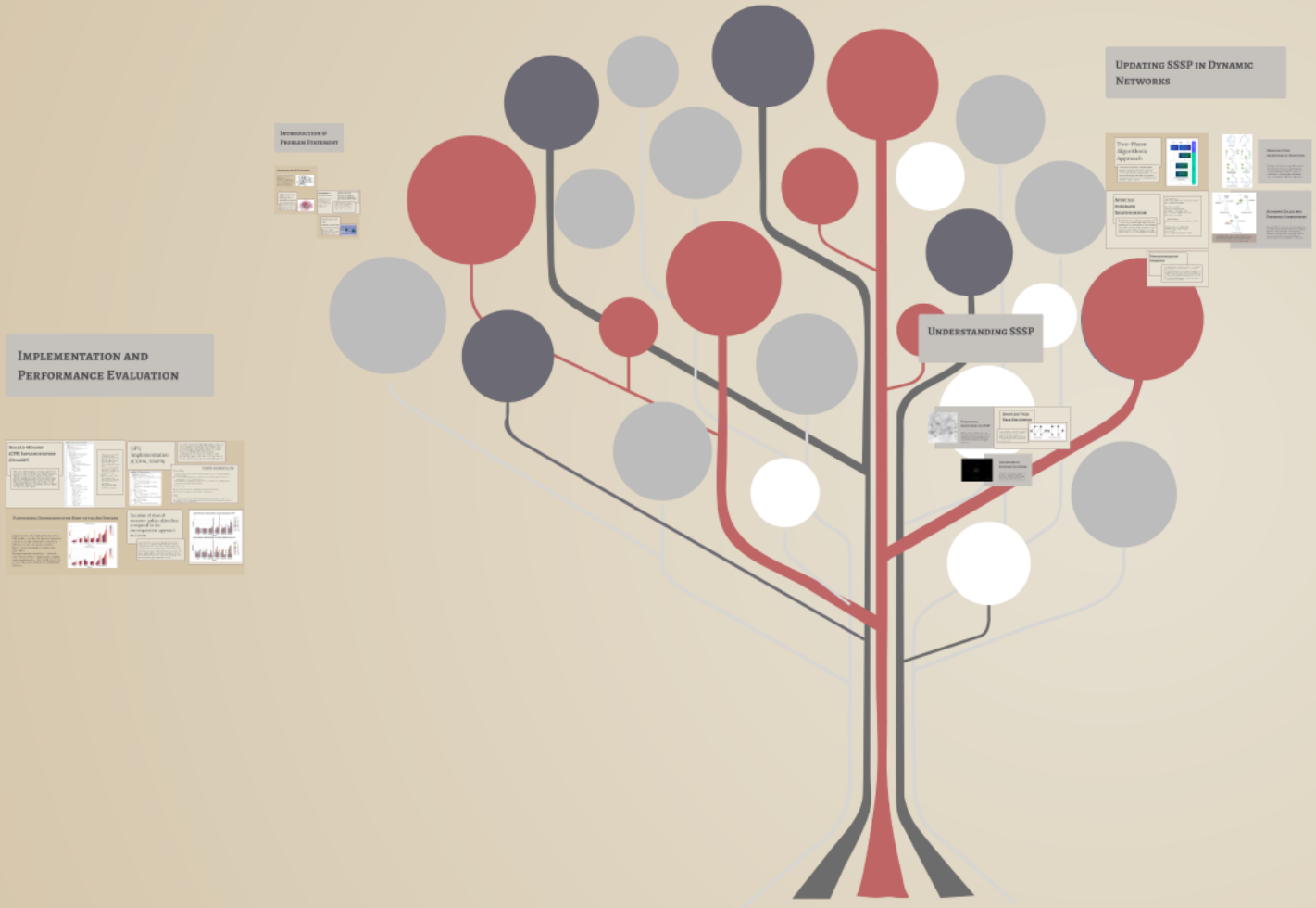
Shortest Path Tree Definition

A shortest path tree is a spanning tree rooted at the source vertex that represents the shortest paths to all other vertices in the graph. Maintenance of this tree requires continuous updates as edges are added or removed, which can significantly strain computational efficiency in dynamic networks.



Limitations of Existing Solutions

Existing solutions like Gunrock and Galois operate efficiently on static graphs but struggle with scalability during updates in dynamic networks. They often require complete recomputation after minor changes, leading to inefficiencies in scenarios where edges or nodes change frequently.



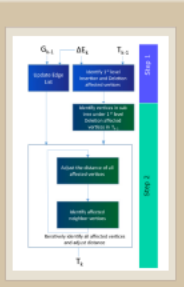
A Parallel Algorithm for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks

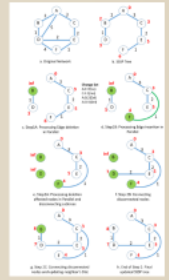
By Fakhir Ali 22I-0762 &
Ayna Sulaiman 22I-1054

Updating SSSP in Dynamic Networks

Two-Phase Algorithmic Approach

The proposed algorithm divides the update process into two key phases: identification of affected subgraphs and iterative updates of distances for impacted vertices. This structured approach minimizes redundant computations and improves efficiency when handling dynamic changes in large networks.





Handling Edge Insertions vs. Deletions

The algorithm distinctly treats edge insertions and deletions due to their differing impacts on the SSSP tree. Insertions may create new shortest paths while deletions require the recalibration of existing paths, maintaining the tree structure's integrity during updates.

Affected Subgraph Identification

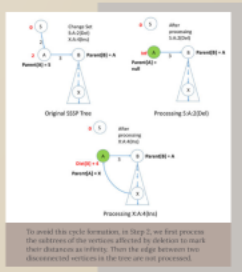
Identification of the affected subgraph occurs by processing each changed edge in parallel, marking vertices influenced by these changes. This method enables rapid recognition of the necessary areas needing updates, ensuring efficiency in the overall algorithm's execution.

- Edge Deletion:
If the deleted edge (u, v) is not part of the SSSP tree, no update is needed.

If it is part of the tree:
Assume u is the parent of v.
Set Dist[v] = ∞, Parent[v] = null.
Mark v as affected (Affected[v] = true, Affected_Del[v] = true).

- Edge Insertion:
Let the inserted edge be (u, v) with weight W(u, v).

If Dist[u] + W(u, v) < Dist[v], then:
Update Dist[v] = Dist[u] + W(u, v).
Set Parent[v] = u.
Mark v as affected (Affected[v] = true).



To avoid this cycle formation, in Step 2, we first process the subtrees of the vertices affected by deletion to mark their distances as infinity. Then the edge between two disconnected vertices in the tree are not processed.

Avoiding Cycles and Ensuring Correctness

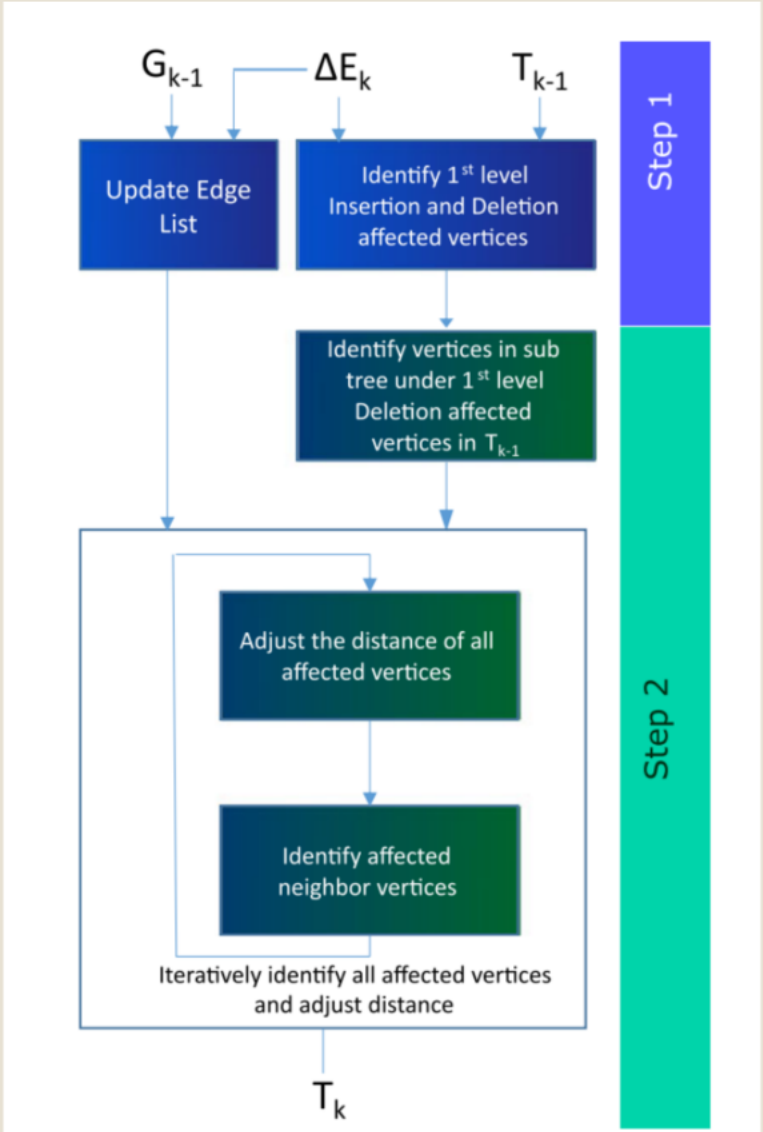
To prevent the formation of cycles during updates, the algorithm systematically disconnects subtrees affected by deletions. This careful approach guarantees that the SSSP tree remains valid and accurately reflects the shortest paths post-update, reinforcing the algorithm's robustness.

Synchronization Strategy

- Multiple processors may update the affected vertices, thus race conditions can occur where these vertices may not be updated to their minimum distance from the root.
- The traditional method to resolve this problem is by using locking constructs. However using these constructs reduces the scalability of the implementation. Instead of using locking constructs, the paper proposed iteratively updating the distances of the vertices to a lower value. Over the iterations the vertices do converge to their minimum distance from the source node or root of the tree.
- Although multiple iterations add to the computation time, the overhead is much lower than using locking constructs.

Two-Phase Algorithmic Approach

The proposed algorithm divides the update process into two key phases: identification of affected subgraphs and iterative updates of distances for impacted vertices. This structured approach minimizes redundant computations and improves efficiency when handling dynamic changes in large networks.



Affected Subgraph Identification

Identification of the affected subgraph occurs by processing each changed edge in parallel, marking vertices influenced by these changes. This method enables rapid recognition of the necessary areas needing updates, ensuring efficiency in the overall algorithm's execution.

- Edge Deletion:

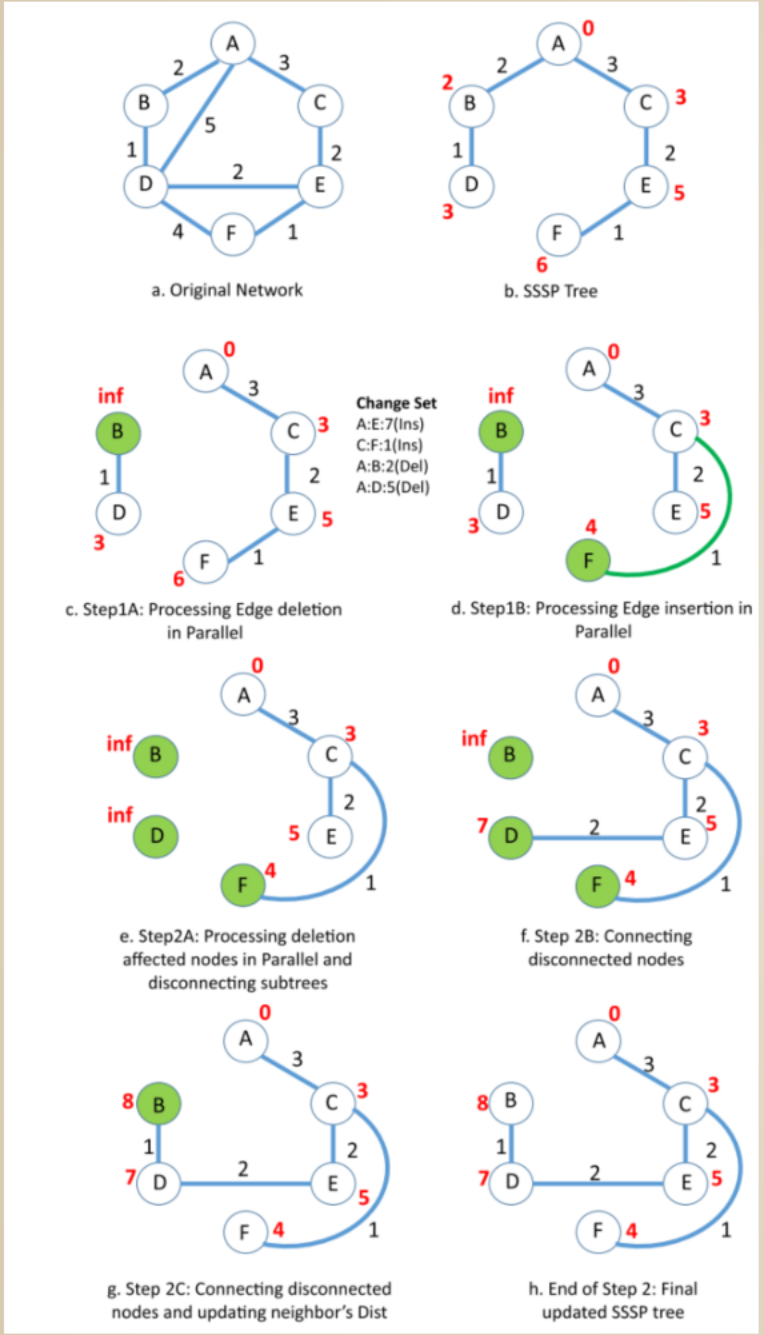
If the deleted edge (u, v) is not part of the SSSP tree, no update is needed.

If it is part of the tree:
Assume u is the parent of v.
Set Dist[v] = ∞, Parent[v] = null.
Mark v as affected (Affected[v] = true, Affected_Del[v] = true).

- Edge Insertion:

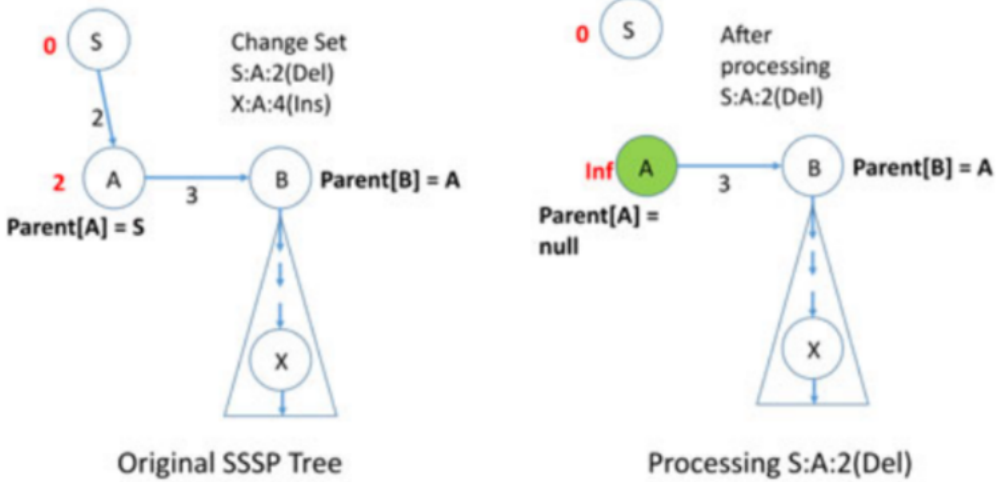
Let the inserted edge be (u, v) with weight W(u, v).

If Dist[u] + W(u, v) < Dist[v], then:
Update Dist[v] = Dist[u] + W(u, v).
Set Parent[v] = u.
Mark v as affected (Affected[v] = true).



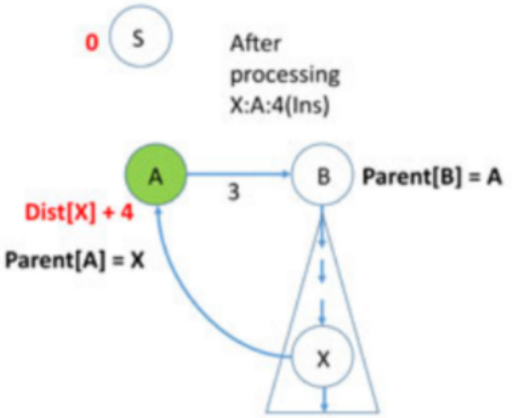
Handling Edge Insertions vs. Deletions

The algorithm distinctly treats edge insertions and deletions due to their differing impacts on the SSSP tree. Insertions may create new shortest paths while deletions require the recalibration of existing paths, maintaining the tree structure's integrity during updates.



Original SSSP Tree

Processing S:A:2(Del)



Processing X:A:4(Ins)

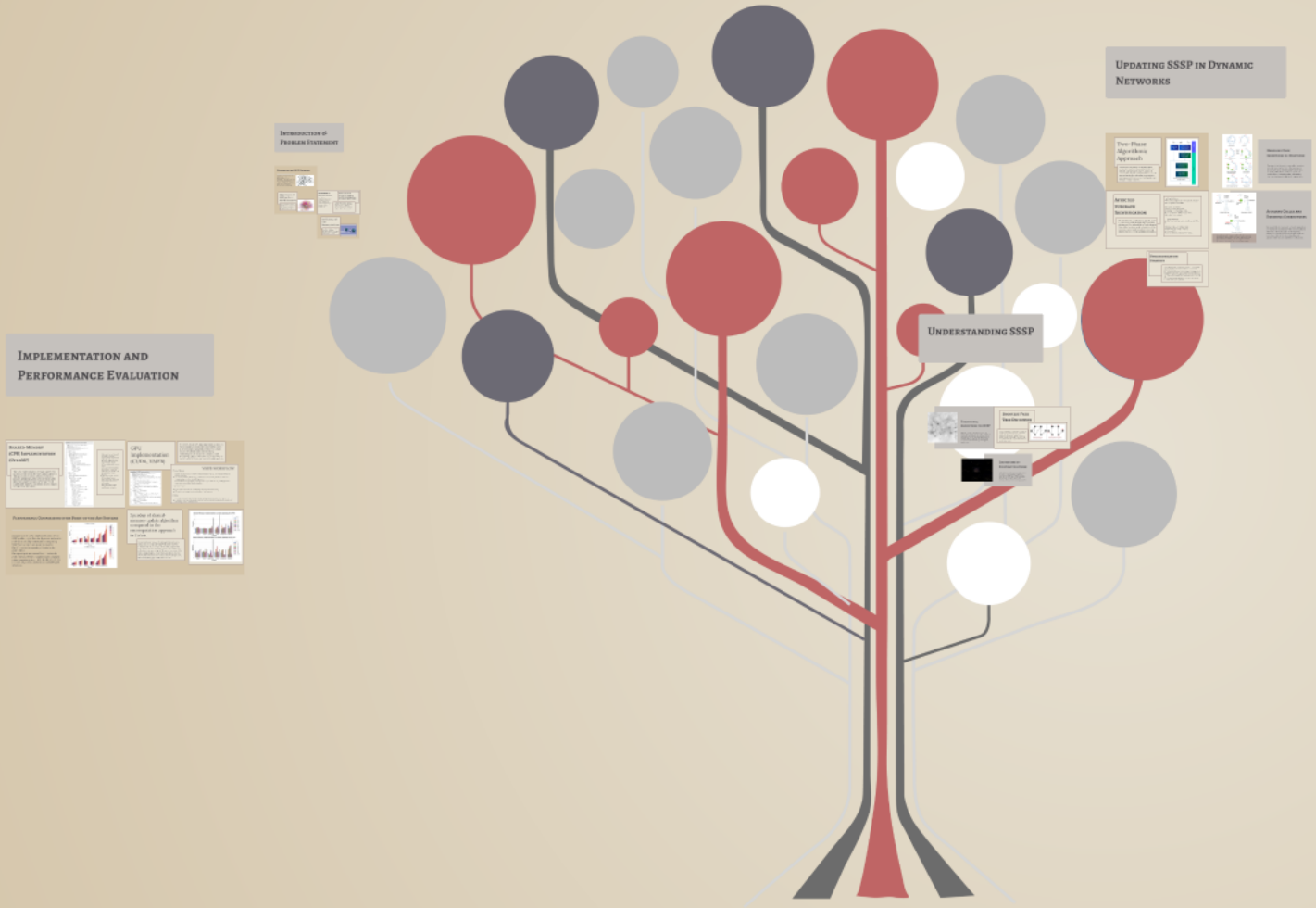
To avoid this cycle formation, in Step 2, we first process the subtrees of the vertices affected by deletion to mark their distances as infinity. Then the edge between two disconnected vertices in the tree are not processed.

Avoiding Cycles and Ensuring Correctness

To prevent the formation of cycles during updates, the algorithm systematically disconnects subtrees affected by deletions. This careful approach guarantees that the SSSP tree remains valid and accurately reflects the shortest paths post-update, reinforcing the algorithm's robustness.

Synchronization Strategy

- Multiple processors may update the affected vertices, thus race conditions can occur where these vertices may not be updated to their minimum distance from the root.
- The traditional method to resolve this problem is by using locking constructs. However using these constructs reduces the scalability of the implementation.
- Instead of using locking constructs, the paper proposed iteratively updating the distances of the vertices to a lower value. Over the iterations the vertices thus converge to their minimum distance from the source node or root of the tree.
- Although multiple iterations add to the computation time, the overhead is much lower than using locking constructs.



A Parallel Algorithm for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks

By Fakhir Ali 22I-0762 &
Ayna Sulaiman 22I-1054

Implementation and Performance Evaluation

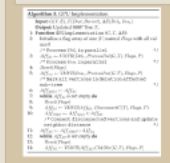
Shared-Memory (CPU) Implementation (OpenMP)

The CPU implementation leverages OpenMP to achieve parallelism in the SSSP updating process. This approach processes affected vertices in parallel, optimizing workload distribution while minimizing synchronization overhead, which significantly improves execution speed compared to sequential algorithms.

- Built using C++ and OpenMP for parallel execution.
- Edge deletions are processed first; affected vertices are disconnected and marked.
- Edge insertions update distances if they offer shorter paths.
- Uses asynchronous updates with dynamic scheduling.
- Each vertex has an affected flag to track necessary updates.
- Updates propagate until no further changes occur.
- Asynchrony level (k) controls sync frequency for better performance.
- Reach processing of edge changes improves load balancing and scalability.

GPU Implementation (CUDA, VMFB)

The CUDA-based GPU implementation utilizes the Vertex-Marking Functional Block (VMFB) system to process edge changes in parallel. This design reduces synchronization bottlenecks by minimizing the use of atomic operations and structuring updates efficiently, resulting in higher performance during edge insertions and deletions.



VMFB WORKFLOW

1) Vertex Marking:

- Each thread processes an element from an operand array (e.g., list of changed edges or affected vertices).
- If a certain condition (defined by a custom function Fn) is met, the thread marks the corresponding vertex in a shared Flags array.
- All threads only set the flag to a constant value (e.g., 1)—this one-way marking ensures correctness even without synchronization.

2) Synchronization:

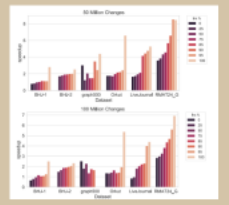
- A global barrier ensures all threads finish marking before continuing.
- This avoids any inconsistencies in the shared memory state.

3) Filter:

- This step collects only the marked vertices, using a predicate check (like "flag == 1").
- It creates a clean list of affected vertices for the next step, while removing duplicates and maintaining redundant computations.

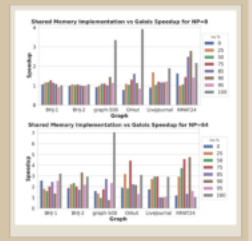
Performance Comparisons with State-of-the-Art Systems

Comparison of GPU implementation of the SSSP update algorithm for dynamic networks with Gunrock implementation computing SSSP from scratch on static networks.
The Y -axis is the speedup; X-axis is the graph input.
The speedup is measured for all networks with 50M and 100M changed edges changed edges consisting of p = 100, 95, 90, 85, 75, 50, 25, and 100 percent insertions and (100−p)% deletions.



Speedup of shared-memory update algorithm compared to the recomputation approach in Galois.

Experiments on real-world and synthetic networks show that the proposed framework performs well compared to the re-computation based approaches (e.g., Galois for CPU and Gunrock for GPU) when the edge insertion percentage is 50 percent or more of the total changed edges. Observation also shows that as the number of deletions increases beyond 50 percent, the recomputing approach performs well.



Shared-Memory (CPU) Implementation (OpenMP)

The CPU implementation leverages OpenMP to achieve parallelism in the SSSP updating process. This approach processes affected vertices in parallel, optimizing workload distribution while minimizing synchronization overhead, which significantly improves execution speed compared to sequential algorithms.

Algorithm 4. Asynchronous Update of SSSP

Input: $G(V, E)$, T, $Dist$, $Parent$, source vertex s, $\Delta E(Del_k, Ins_k)$
Output: Updated SSSP Tree T_k

1: **Function** `AsynchronousUpdating` ($\Delta E, G, T$):
2: Set Level of Asynchrony to A.
3: $Change \leftarrow True$
4: **while** $Change$ **do**
5: $Change \leftarrow False$
6: **pragma omp parallel schedule (dynamic)**
7: **for** $v \in V$ **do**
8: **if** $Affected\_Del[v] = True$ **then**
9: Initialize a queue Q
10: Push v to Q
11: $Level \leftarrow 0$
12: **while** Q *is not empty* **do**
13: Pop x from top of Q
14: **for** c *where* c *is child of* x *in* T **do**
15: Mark c as affected and change distance to infinite
16: $Change \leftarrow True$
17: $Level \leftarrow Level + 1$
18: **if** $Level \leq A$ **then**
19: Push c to Q
20: $Change \leftarrow True$
21: **while** $Change$ **do**
22: $Change \leftarrow False$
23: **pragma omp parallel schedule (dynamic)**
24: **for** $v \in V$ **do**
25: **if** $Affected[v] == True$ **then**
26: $Affected[v] \leftarrow false$
27: Initialize a queue Q
28: Push v to Q
29: $Level \leftarrow 0$
30: **while** Q *is not empty* **do**
31: Pop x from top of Q
32: **for** n *where* n *is neighbor of* x *in* G **do**
33: $Level \leftarrow Level + 1$
34: **if** $Dist[x] > Dist[n]+W(x, n)$ **then**
35: $Change \leftarrow True$
36: $Dist[x] = Dist[n]+W(x, n)$
37: $Parent[x] = n$
38: $Affected[x] \leftarrow True$
39: **if** $Level \leq A$ **then**
40: Push x to Q
41: **if** $Dist[n] > Dist[x]+W(n, x)$ **then**
42: $Change \leftarrow True$
43: $Dist[n]=Dist[n]+W(n, x)$
44: $Parent[n] = x$
45: $Affected[n] \leftarrow True$
46: **if** $Level \leq A$ **then**
47: Push n to Q

- Built using C++ and OpenMP for parallel execution.
- Edge deletions are processed first; affected vertices are disconnected and marked.
- Edge insertions update distances if they offer shorter paths.
- Uses asynchronous updates with dynamic scheduling.
- Each vertex has an Affected flag to track necessary updates.
- Updates propagate until no further changes occur.
- Asynchrony level (A) controls sync frequency for better performance.
- Batch processing of edge changes improves load balancing and scalability.

GPU Implementation (CUDA, VMFB)

The CUDA-based GPU implementation utilizes the Vertex-Marking Functional Block (VMFB) system to process edge changes in parallel. This design reduces synchronization bottlenecks by minimizing the use of atomic operations and structuring updates efficiently, resulting in higher performance during edge insertions and deletions.

Algorithm 5. GPU Implementation

Input: $G(V, E)$, $T(Dist, Parent)$, $\Delta E(Del_k, Ins_k)$
Output: Updated SSSP Tree T_u

1: **Function** `GPUimplementation` $(G, T, \Delta E)$:
2: Initialize a flag array of size $|V|$ named *Flags* with all values 0
`/* Process` Del_k `in parallel */`
3: $Aff_{del} \leftarrow VMFB(Del_k, ProcessDel(G, T), Flags, P)$
`/* Process` Ins_k `in parallel */`
4: $Reset(Flags)$
5: $Aff_{ins} \leftarrow VMFB(Ins_k, ProcessIns(G, T), Flags, P)$
`/* Mark all vertices in deletion affected sub-tree */`
6: $Aff_{Alldel} \leftarrow Aff_{del}$
7: **while** Aff_{del} *is not empty* **do**
8: $Reset(Flags)$
9: $Aff_{del} \leftarrow VMFB(Aff_{del}, DisconnectC(T), Flags, P)$
10: $Aff_{Alldel} \leftarrow Aff_{Alldel} \cup Aff_{del}$
`/* Connect disconnected vertices and update neighbor distance */`
11: $Aff_{All} \leftarrow Aff_{Alldel} \cup Aff_{ins}$
12: **while** Aff_{All} *is not empty* **do**
13: $Reset(Flags)$
14: $Aff_{All} \leftarrow VMFB(Aff_{All}, ChkNbr(G, T), Flags, P)$

VMFB WORKFLOW

1) Vertex Marking:

- Each thread processes an element from an operand array (e.g., list of changed edges or affected vertices).
- If a certain condition (defined by a custom function Fx) is met, the thread marks the corresponding vertex in a shared Flags array.
- All threads only set the flag to a constant value (e.g., 1)—this one-way marking ensures correctness even without synchronization.

2) Synchronization:

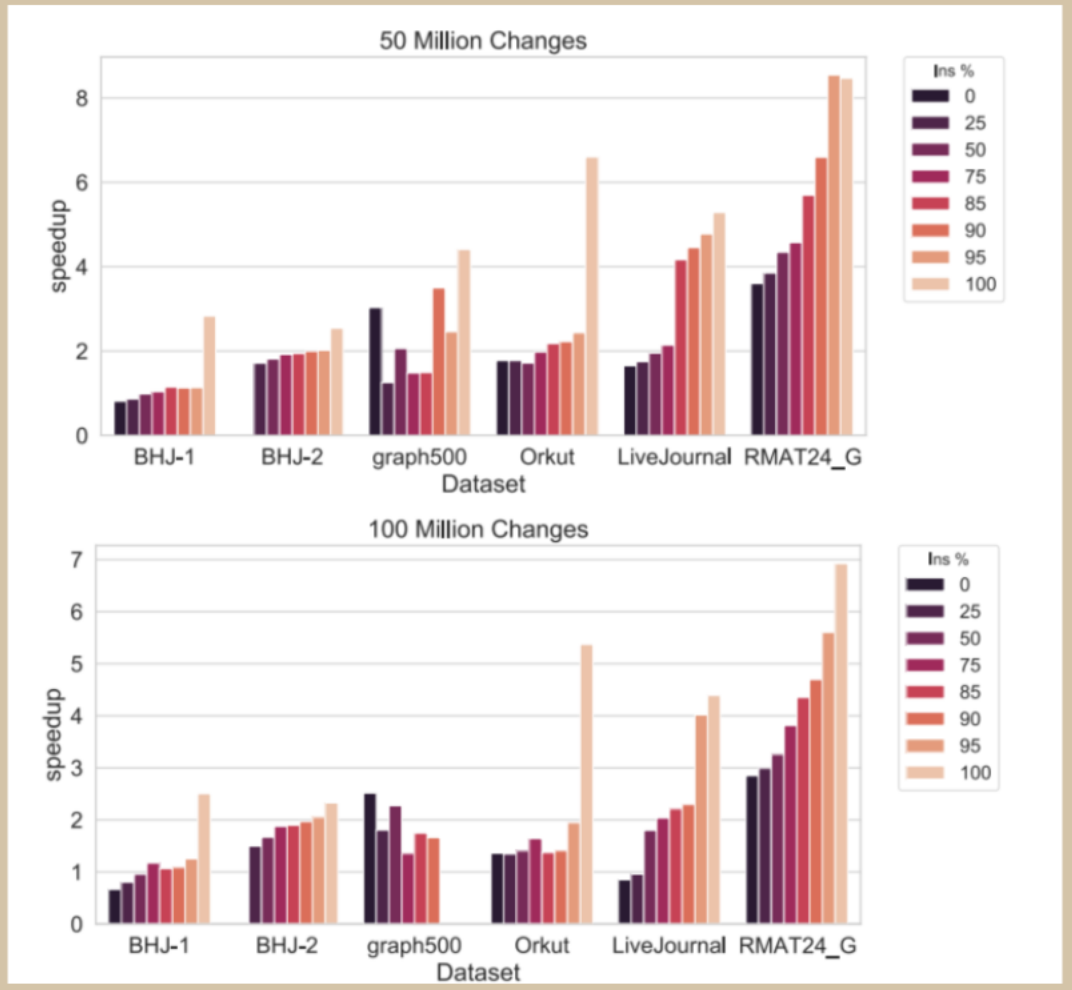
- A global barrier ensures all threads finish marking before continuing.
- This avoids any inconsistencies in the shared memory state.

3) Filter:

- This step collects only the marked vertices, using a predicate check (like "flag == 1").
- It creates a clean list of affected vertices for the next step, while removing duplicates and minimizing redundant computation.

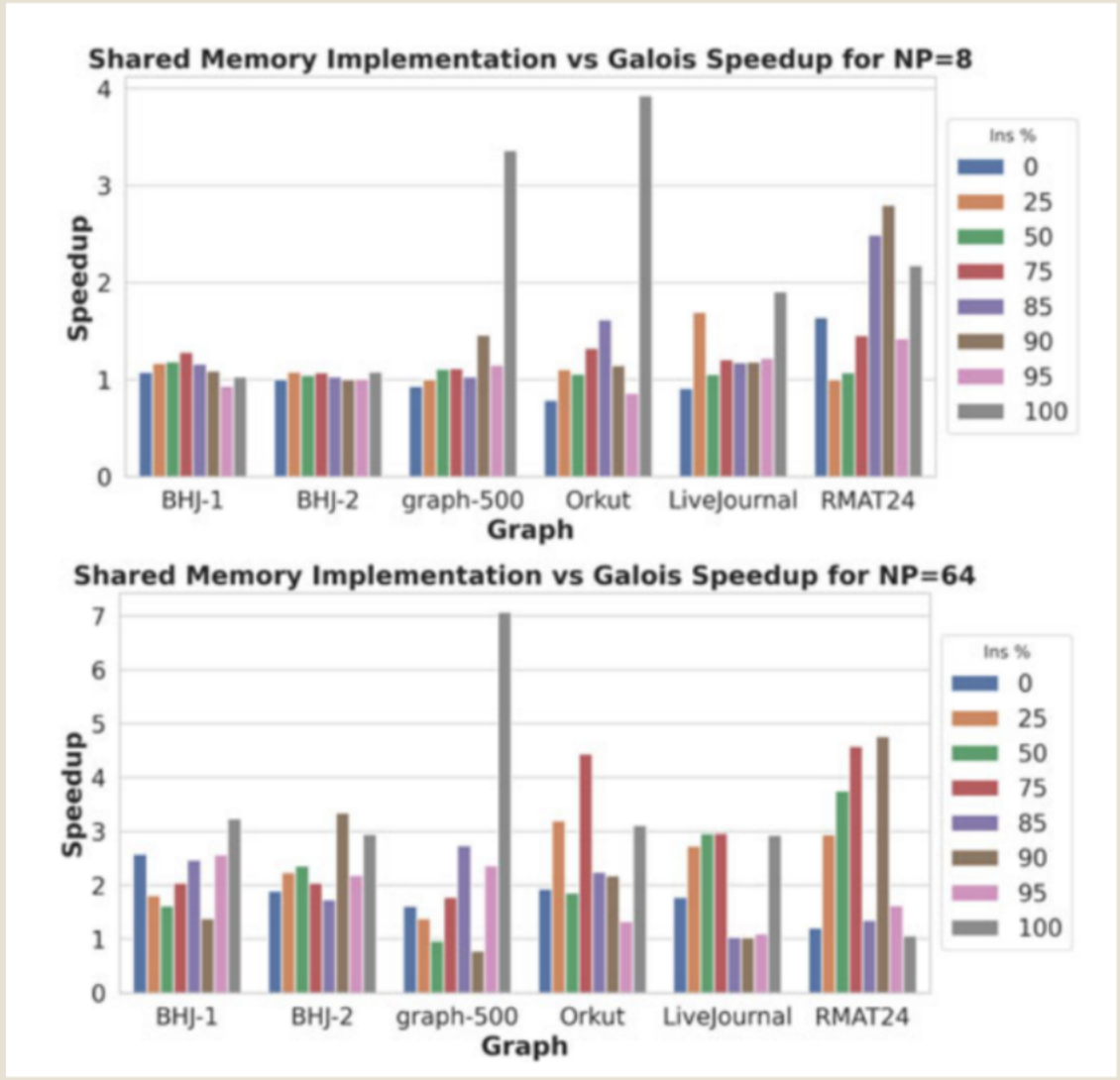
Performance Comparisons with State-of-the-Art Systems

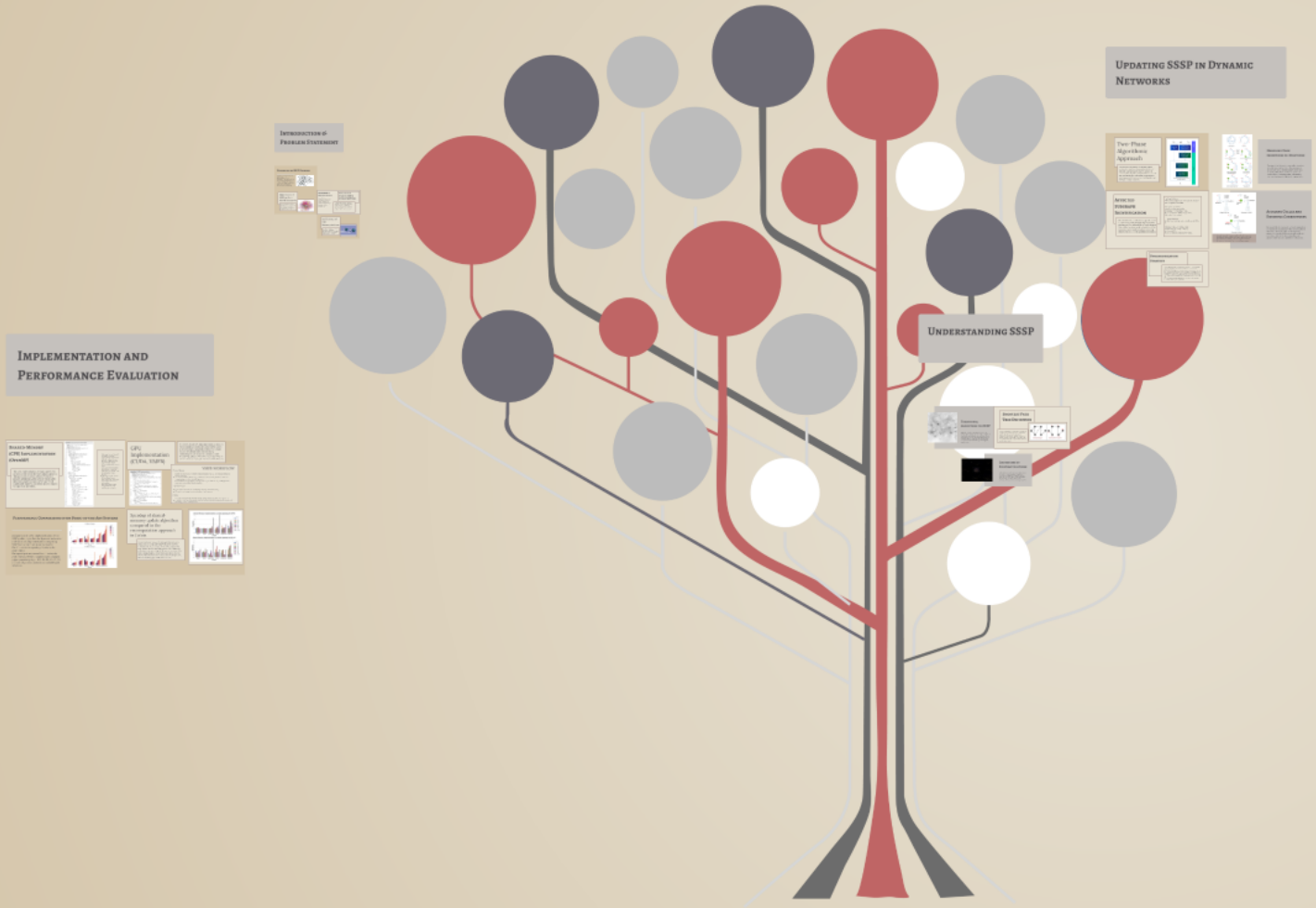
Comparison of GPU implementation of the SSSP update algorithm for dynamic networks with Gunrock implementation computing SSSP from scratch on static networks.
The Y -axis is the speedup; X-axis is the graph input.
The speedup is measured for all networks with 50M and 100M changed edges changed edges consisting of p = 100, 95, 90, 85, 75, 50, 25, and 100 percent insertions and (100-p)% deletions.



Speedup of shared-memory update algorithm compared to the recomputation approach in Galois.

Experiments on real-world and synthetic networks show that the proposed framework performs well compared to the re-computation based approaches (e.g., Galois for CPU and Gunrock for GPU) when the edge insertion percentage is 50 percent or more of the total changed edges. Observation also shows that as the number of deletions increases beyond 50 percent, the recomputing approach performs well.





A Parallel Algorithm for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks

By Fakhir Ali 22I-0762 &
Ayna Sulaiman 22I-1054