This research tackles the **Single Source Shortest Path (SSSP)** problem in **dynamic networks**, where the network structure changes over time—a scenario common in real-world systems. Traditional parallel algorithms assume **static graphs**, making them inefficient for dynamic settings.

To address this, the authors propose a **parallel algorithmic framework** designed specifically for **updating** the SSSP when the network changes, instead of recomputing everything from scratch. Their method identifies only the **affected parts of the network** and updates a **rooted tree data structure** that holds the most relevant edges.

They implemented the algorithm on both **shared-memory systems** and **GPUs**, and experimental results show that it's **scalable** and often **faster** than existing methods that reprocess the whole graph after each update.

1. Introduction Section

- **Networks** (or graphs) are key tools used to model complex systems across many domains—social networks, communication systems, biology, etc.
- These networks are often **large-scale** (millions of vertices, billions of edges) and **dynamic**, meaning their structure changes over time.
- To analyse such networks efficiently, we need:
    1. Algorithms that can **quickly update** when the network changes,
    2. **Parallel** versions of these algorithms to handle the scale.
- Existing parallel algorithms for Single Source Shortest Path (SSSP) mainly focus on **static graphs** and are often **tied to specific platforms**, limiting portability across CPU/GPU or exascale systems.
- The authors propose a **platform-independent framework** to update SSSP on **dynamic graphs**, usable on both **GPUs and shared-memory CPUs**.
- Their approach:
    - Detects **affected subgraphs** in parallel.
    - Updates only these subgraphs using an **iterative, rooted-tree-based method** that avoids heavy synchronization.
- **Results:**
    - On **GPU**, their method achieves up to **5.6x speedup** over Gunrock.
    - On **shared-memory CPU**, they outperform Galois with up to **5x speedup**.
    - Their framework can process up to **100 million graph changes**, 10x more than prior work.
- **Key Contributions:**
    - A unified framework for SSSP updates on both CPU and GPU.
    - A novel rooted-tree structure for better load balance and reduced synchronization.
    - A **functional-block-based GPU implementation** that minimizes redundant work.

2. Background Section

- The paper works with **weighted graphs** (non-negative edge weights), represented as **G(V, E)**, where:
  - V = vertices (nodes)
  - E = edges (connections)
- The **Single Source Shortest Path (SSSP)** problem involves finding the shortest paths from a **source vertex s** to **all other vertices** in the graph.
  - The result is a **spanning tree**, known as the **SSSP tree**.
  - It stores distances from the source and **parent-child links** to represent the tree.
- In the SSSP tree:
  - If vertex u has a shorter distance than v, and there's an edge between them, then u becomes the **parent** of v.

Computing SSSP – Dijkstra's Algorithm:

- Works for graphs with **non-negative weights**.
- Maintains two sets:
  - **Set1**: vertices whose shortest paths are finalized.
  - **Set2**: the remaining vertices.
- The algorithm:

1. Picks the vertex in Set2 with the **minimum path estimate**.
2. Adds it to Set1.
3. **Relaxes** all outgoing edges of the selected vertex.
   - Relaxing means checking if going through that vertex gives a **shorter path** to a neighbour.

- Using a **Fibonacci heap**, Dijkstra's algorithm runs in **O(|E| + |V| log |V|)** time.
- Other algorithms like **Bellman-Ford** also exist, especially for graphs with negative weights.

2.1 Dynamic Graph Change Example

Initial State:

➔ Graph G (V, E)
➔ SSSP Tree rooted at vertex S

Let's assume the following initial shortest distances from source S:

Dist[S] = 0

Dist[A] = 2

Dist[B] = 5

Dist[C] = 7

Edge Insertion: (A, C) with weight 2

1. **Determine affected vertex:**

   Dist[A] = 2, Dist[C] = 7

   → A is closer to S than C → x = C, y = A

2. **Update C's distance:**

   Dist[C] = Dist[A] + weight(A, C) = 2 + 2 = 4

   (improved from previous 7)

3. **Enqueue neighbours of C**

   PriorityQueue ← [neighbours of C]

4. **Update affected subgraph**
   - Relax each neighbour using Dijkstra-like logic
   - Update parent pointers if distance improves

Edge Deletion: (A, C)

1. **Determine affected vertex:**
   - Assume A was parent of C via (A, C)
   - On deletion, C's previous path is invalid
2. **Set Dist[C] = ∞**
   - Mark C as needing re-evaluation
3. **Enqueue C**
   - Start relaxing edges again
   - Try to find new shortest path to C
4. **Propagate changes**
   - If C was parent of others, their distances may also change

---

Repeat Until:

- Priority queue is empty
- All affected vertices have updated shortest paths

3.1 GPU-Based Implementation

GPU-Based SSSP Computation

Gunrock Framework

- **Library**: Gunrock
- **Key Idea**: Uses **data-centric abstraction**
- **Architecture**:
    1. **Advance**: Move across edges/vertices
    2. **Filter**: Discard unnecessary nodes
    3. **Compute**: Update values (e.g., distances)

GPU Bellman-Ford Implementation

- **Device**: Kepler GPU
- **Strategy**: Uses **two queues** for relaxed nodes
- **Tech**: **Dynamic parallelism**
- **Limitation**: Involves **atomic operations**, which serialize parts of the algorithm (reduces parallel efficiency)

Other GPU Studies

- Compares performance of SSSP on **temporal graphs** using:
    - Multicore CPUs
    - GPU accelerators

JavaScript GPU Update Algorithm

- First known dynamic SSSP implementation on GPU for updates
- Good performance **only if < 10% of edges change**

3.2 CPU-Based Implementation

OpenMP-Based Parallel SSSP

- **First shared-memory algorithm** for **updating** SSSP in **dynamic networks**
- Implemented using **OpenMP**
- Currently the **only known shared-memory parallel dynamic update** approach

---

Other Relevant Implementations (Not Dynamic + Shared-Memory):

Spark-Based Approach

- Updates SSSP on dynamic networks
- Not shared-memory; uses **Spark framework**

Bellman-Ford on Hypergraphs

- Parallel Bellman-Ford
- Focuses on **hypergraph representation**

D–Stepping on Static Graphs

- Two **shared-memory multicore** implementations
- But only for **static networks**, not dynamic updates

Graph Libraries & Models

Galois Framework

- Shared-memory
- Uses **amorphous data-parallelism**
- Supports **priority scheduling**
- Includes parallel **Dijkstra's algorithm**

Havoqgt

- Distributed computing platform
- Only supports **full re-computation** for SSSP
- No dynamic update support

Other Approaches

Streaming Approximations

- Focus on **streaming graphs**
- Provide **approximate SSSP** updates

---

4. Parallel Dynamic SSSP Algorithm

Goal

Efficiently update the **Single Source Shortest Path (SSSP)** tree in **parallel** as the graph **evolves over time**.

Key Concepts

- **Graph at time step k:**
  Gk (Vk,Ek)
  - Vk: vertices
  - Ek: edges
- **SSSP Tree at time k:**
  Tk – A valid SSSP rooted at the source vertex.
- **Changes in the graph (ΔE_k):**
  - Edge Insertions: Ins_k
  - Edge Deletions: Del_k
  - Update rule:
    Ek = (Ek−1 U Ins.k) \ Del.k

Tree-Based Representation

- The **SSSP result** is stored as a **rooted tree**:
  - Maintains **distance from the root**
  - Maintains **parent-child relationships**
- Tree Tk is derived from:
  - Previous tree Tk−1
  - The set of changed edges ΔEk

How It Works (At a High Level)

1. Take Tk−1 from the previous time step
2. Apply updates based on insertions and deletions of edges
3. Efficiently update affected parts of the SSSP tree **in parallel**

4.1 A Parallel Framework for Updating SSSP

This article outlines a parallel framework designed to efficiently update the Single-Source Shortest Path (SSSP) in dynamic networks, especially in response to edge insertions and deletions. The process is divided into two steps:

Step 1: Identifying Affected Subgraphs

- **Goal**: Identify vertices affected by changes in the network (edge deletions or insertions).
- **Data Structures**: The SSSP tree is stored as an adjacency list with additional arrays to track each vertex's parent, distance from the root, and whether it is affected by deletions or other changes.
- **Edge Deletion**: When an edge is deleted, the algorithm checks if it was part of the SSSP tree. If so, it marks the affected vertex and updates its distance and parent to indicate disconnection.
- **Edge Insertion**: When an edge is inserted, the algorithm checks if the new edge provides a shorter path to a vertex. If so, it updates the vertex's distance, parent, and marks it as affected.

Step 2: Updating the Affected Subgraph

- **Goal**: Update the affected subgraph to ensure the SSSP tree reflects the changes.
- **Handling Disconnected Vertices**: When a vertex becomes disconnected (due to edge deletion), its distance is set to infinity. All its descendants are similarly affected and marked.
- **Iterative Updates**: For each affected vertex, the algorithm checks if the distance to its neighbors can be reduced by passing through the vertex. If so, the distances and parent pointers are updated, and the process continues iteratively until no further updates are possible.
- **Dynamic Scheduling**: To balance the workload of updating potentially large subgraphs, a dynamic scheduler is employed for parallel processing.

Parallel Processing

- The identification of affected vertices and the updates are performed in parallel for efficient computation, minimizing synchronization overhead. The framework ensures that multiple changes can be processed simultaneously, improving the performance for large networks.

4.2 Challenges in Achieving Scalability

This section addresses the key challenges in achieving scalability for the parallel framework designed to update Single-Source Shortest Paths (SSSP) in dynamic networks. The article discusses three primary challenges and the solutions implemented to overcome them:

1. Load Balancing

- **Challenge**: The number of operations in Step 2 of the algorithm depends on the size of the subgraph affected by changes. Since these subgraphs can vary in size, there is a risk of imbalance in the work distributed across processing units.

- **Solution**: Rather than applying complex load balancing techniques based on the prior knowledge of subtree sizes, the algorithm uses dynamic scheduling. This approach offers good scalability with significantly lower overhead than more complicated methods.

2. Synchronization

- **Challenge**: When multiple processors update the affected vertices in parallel, race conditions can occur, where vertices may not be updated to their correct minimum distance.
- **Solution**: Instead of using locking constructs, which could reduce scalability, the algorithm iteratively updates the distances of affected vertices. Over successive iterations, vertices converge to their minimum distance from the root. While the added iterations increase computation time, the overhead remains much lower than locking mechanisms.

3. Avoiding Cycle Formation

- **Challenge**: The algorithm assumes that the data structure is a tree. However, when multiple edges are inserted in parallel, cycles can form, which violate the tree structure.
- **Solution**: To prevent cycle formation, Step 2 first processes the subtrees of vertices affected by edge deletion, setting their distances to infinity. This ensures that edges between disconnected vertices are not processed, avoiding potential cycles, even if an edge is inserted that could otherwise create one.

These solutions aim to maintain scalability while preventing common issues like load imbalance, synchronization overhead, and cycle formation. The approach enables efficient parallel processing of updates in dynamic networks.

4.3 Proofs and Scalability Analysis

Lemma 1: Parent-Child Relation Produces a Tree

- **Goal**: Prove that the parallel updating algorithm maintains a tree structure by showing that the parent-child relationships between vertices do not create cycles.
- **Proof Strategy**: The directed graph formed by parent-child relations must be a Directed Acyclic Graph (DAG), and since all connected components are DAGs, they form trees.
    - Consider any two vertices, a and b, in the graph.
    - If a is an ancestor of b, then the distance from a will always be less than the distance from b (since a is assigned as the parent of b only if Dist(a) < Dist(b)).
    - Because of this ordering property, there cannot be a cycle, as the distance of any vertex from its ancestor is always smaller than that of

its descendants. This prevents a reverse path from b to a (which would imply a cycle).
    - **Conclusion**: All connected components are DAGs, and the parent-child relations therefore form valid trees.

Lemma 2: Validity of the SSSP Tree

- **Goal**: Show that the tree produced by the parallel algorithm is a valid Single-Source Shortest Path (SSSP) tree.
- **Proof Strategy**: Suppose the tree produced by the algorithm, T algo, is not a valid SSSP tree, then there must be a vertex a for which the distance from the source in T algo is greater than in the known SSSP tree Tk.
    - Consider a subset of vertices S where the distances in both T algo and Tk are equal.
    - If a's distance in T algo is larger than in Tk, then a is part of an affected subgraph.
    - Given that an affected vertex a would have its distance updated during the parallel algorithm (either by comparison with its parent or via an edge weight), we show that the distances from a in both trees will be the same at the end of the algorithm.
    - **Conclusion**: The parallel algorithm produces a valid SSSP tree since it updates the distances correctly based on the original tree Tk.

Scalability Analysis:

- Parallelism in Step 1:
    - The time complexity for processing the m changed edges in parallel is O(m/p), where p is the number of processing units.
- Parallelism in Step 2:
    - In Step 2, the algorithm processes affected vertices and their neighbours. The number of affected vertices is x, and the degree of each vertex is d. The time complexity for each iteration in this step is O(xd/p), where d is the average degree of a vertex.
    - The maximum number of iterations needed depends on the diameter D of the graph.
    - Therefore, the upper bound time complexity for Step 2 is O(D.xd/p), where D is the diameter of the graph.
- Total Complexity:
    - Combining both steps, the overall complexity of the algorithm is:

$$O(m/p) + O(D \cdot xd/p)$$

The term D·xd represents the number of edges processed during Step 2. If the number of edges processed in the computation method is E+m, for the updating method to be efficient, the following condition must hold:

$$Ex<E+m$$

where Ex represents the number of edges processed in Step 2.

5.1 Shared-Memory Implementation

Platform:

- Implemented in **C++ with OpenMP** to exploit shared-memory parallelism.

Inputs:

- Graph G (V, E)
- Changed edges ΔE = (Del_k, Ins_k)
- Source vertex s
- Existing SSSP tree T

Key Features:

1. **Edge Processing (Parallel):**
    - Del_k: Deleted edges are only processed if they exist in the key edges of tree T.
    - Ins_k: Inserted edge e(u,v) is updated **only if** it decreases the distance of v.
2. **Affected Vertices:**
    - Vertices impacted by edge changes are flagged as Affected = True.
    - These are processed in **parallel** until no affected vertices remain.
3. **Asynchronous Update Mechanism (Algorithm 4):**
    - Affected vertices are explored using BFS-like logic.
    - Distance and parent information is updated if a shorter path is found.
    - A **Level of Asynchrony (A)** is used to reduce frequent thread synchronization.
    - Threads process neighbours up to a distance A before synchronizing.

    - Redundant computation is **allowed** to avoid locks and maintain scalability.
4. **Dynamic Scheduling:**
    - OpenMP's schedule(dynamic) is used for better load balancing.

Performance Optimizations:

- **Batch Processing:**
    - Changed edges are processed in **batches** rather than all at once.
    - Helps balance workload and avoid memory hotspots.
- **Asynchrony vs Synchronization Trade off:**
    - Higher asynchrony = fewer synchronizations but more iterations.
    - Optimal A level affects speed and efficiency (discussed in Section 6).

5.2 GPU Acceleration Model

- **SIMT Execution**: Each CUDA thread handles one element of the operand array, enabling massive parallelism.
- **Grid-Stride Loops**: Used when the operand array is larger than the number of threads, ensuring full coverage.
- **CSR Format**: Graph is stored in standard **Compressed Sparse Row** format, and the SSSP tree is represented by a structure T[i] = (Parent, Dist, Flag).

Vertex-Marking Functional Block (VMFB)

Purpose:

To minimize atomic operations while maintaining correctness in marking vertices affected by graph updates.

VMFB Components:

1. **Vertex Marking**:
    - Each thread performs a user-defined function (Fx) on an operand.
    - If a condition is satisfied, it marks the vertex in the shared Flags array.
    - No atomic ops here, so marking is unidirectional (0 → 1).
2. **Synchronization**:
    - A global barrier waits for all threads to complete.

3. **Filter**:
    - Collects indices where the predicate P(flag == 1) holds true using ballot sync + atomicAdd.
    - Produces a list of marked vertices (no duplication).

Algorithm 5: GPU-Based SSSP Tree Update

Inputs:

- Graph G(V, E)
- SSSP Tree T (with Dist, Parent)
- Edge Update Batch DE = (Del_k, Ins_k)

Step-by-Step Execution

Step 1: Process Edge Updates

- **Deletions**:
    - Del_k processed in parallel by ProcessDel via VMFB.
    - Affects and disconnects first-level deletion-affected vertices.
- **Insertions**:
    - Ins_k processed by ProcessIns via VMFB.
    - Marks first-level insertion-affected vertices.

Step 2: Disconnect Deletion-Affected Subtree

- Input: First-level deletion-affected vertices.
- Repeatedly run DisconnectC on output of previous VMFB until no new affected vertices.
- All such vertices are marked and disconnected from the SSSP tree.

Step 3: Reconnect and Update Distances

- Combine all affected vertices: AffAll = AffAlldel ∪ Affins.
- Repeatedly apply ChkNbr using VMFB to:
    - Reconnect nodes.
    - Update Dist for neighbors.
    - Mark further affected vertices if their distances change.
- Stop when no vertices are marked in an iteration.

Table 1: Functions Used

Function	Purpose
ProcessDel	Handles deletions, marks affected vertices
ProcessIns	Handles insertions, marks affected vertices
DisconnectC	Disconnects subtree from SSSP
ChkNbr	Reconnects and updates distances

6. Experimental Results

- **Datasets**:
  - Five real-world large graphs from the network-repository collection.
  - One synthetic R-MAT graph generated with parameters:
    a=0.45, b=0.15, c=0.15, d=0.
    This synthetic graph, referred to as **G**, is characterized by a **scale-free degree distribution** (common in real-world networks).
- **Hardware setup**:
  - **GPU experiments**:
    - **GPU**: NVIDIA Tesla V100 (32GB memory)
    - **Host CPU**: Dual 32-core AMD EPYC 7452
  - **Shared memory experiments** (CPU-based):
    - **CPU**: Intel Xeon Gold 6148
    - **Memory**: 384GB

6.1 GPU Implementation Results

- Execution time (not TEPS) is shown for CUDA-based SSSP update after:
  - 50M and 100M edge updates.
  - Edge updates include varying ratios of insertions and deletions (e.g., 75% insertions = 25% deletions).

- Why not use TEPS?
  TEPS (Traversed Edges per Second) is less meaningful for dynamic updates, where fewer edge traversals are preferred.
- Key Observations:
  - Execution time is mainly influenced by:
    - Number of affected vertices
    - Network structure
    - Location of changes
      (All are random in these experiments.)
  - When insertions = 100%:
    - No deletions ⇒ faster execution (less work).
    - Deletion-affected vertices do not need to be disconnected and reprocessed.
  - When deletions increase:
    - Execution time generally increases, since:
      - Deletion-affected vertices must be disconnected and then reconnected to the SSSP tree.
  - Overlap optimization:
    - When overlapping subgraphs occur due to deletions:
      - The Filter operation in VMFB merges them to reduce redundant work.
      - This leads to decreased execution time in some cases, especially when deletion percentage increases beyond 25%.
  - Exception:
    - If deletion-affected subgraphs do not overlap, execution time may spike unexpectedly due to lack of optimization benefits.

6.2 GPU Performance Comparison:

Our GPU SSSP Update vs. Gunrock (Static GPU SSSP)

- Why not compare with “Dynamic shortest paths using JavaScript on GPUS”
  - It handles only <10% edge updates, unsuitable for large updates like 50M or 100M.
- No known dynamic GPU SSSP alternatives, so:

    - Compared against Gunrock — a static GPU graph library known for high-performance SSSP.
- How the comparison is done:
    - Our algorithm updates SSSP dynamically.
    - Gunrock recomputes SSSP from scratch on the modified network (after applying edge changes).
    - Edge changes used: 50M and 100M updates.
    - Repeated 6 times → average time used for comparison.
- Key Results (Figure 6 highlights):
    - When insertions > 25%:
        - Our method is significantly faster (up to 8.5× speedup for 50M changes, 5.6× for 100M).
    - When insertions < 25% (i.e., deletions > 75%):
        - Gunrock may outperform our method — because:
            - The input graph becomes small (due to many deletions).
            - Gunrock's static recomputation becomes efficient on smaller graphs.
- Important Observation:
    - If changed edges > 50% of total graph, and deletions > 50%:
        - Recomputing (Gunrock) is often better than updating.
    - If insertions dominate:
        - Our approach is generally more efficient.

6.3 Shared-Memory Implementation Results

Comparison with Galois:

- Galois is a CPU-based SSSP recomputation system.
- Their experiments use 100M edge updates, mixing insertions and deletions.
- Changes are applied to the original graph before SSSP is recomputed from scratch.
- Finding: Their own shared-memory update approach is generally faster than recomputing via Galois.

- Exception: For some graphs (like Graph-500), speedup drops below 1 if more than ~85% of nodes are affected, suggesting recomputation may be better when change impact is too high.

Scalability with Threads

- Used 4 networks, 100M edge updates, tested different thread counts.
- Generally, more threads = faster updates, except in some cases:
    - LiveJournal (25% & 50% insertions) and Orkut (0% insertions): speedup stagnates.
    - Reason: Small percentage of nodes affected (10–15%), so thread gains are limited.

Effect of Asynchrony

- Asynchrony (defined in Section 5.1): Controls overlap in processing updates.
- Tested on Orkut and RMAT-24, for 50M and 100M edge updates.
- Result: Increasing asynchrony generally improves performance except for one case (Orkut with 75% insertions).

Batch Processing

- They test batch-wise edge processing (batch sizes: 15, 30, 50) on:
    - BHJ-1 and Graph-500 networks
    - Change types: 25%, 50%, and 100% insertions
    - Threads: 64 and 72
- Comparison: Against their original implementation Bhowmick, “A shared memory parallel algorithm for updating single-source shortest paths in large dynamic networks”, which doesn’t use batching.
- Result:
    - Batching improves performance, especially at high thread counts.
    - Low thread counts didn’t benefit much from batching.

Summary:

- Their shared-memory update algorithm generally outperforms full recomputation (Galois).

- It scales with threads, benefits from batching and asynchrony.
- But if too many nodes are affected (>80–85%), recomputation may be more efficient.

---

7. Conclusion

Key Contributions:

- Proposed a novel parallel framework to update SSSP (Single-Source Shortest Path) on dynamic graphs.
- Implemented in two HPC environments:
    - Shared-memory CPU
    - NVIDIA GPU

Findings from Experiments:

- The proposed update framework is faster than full recomputation (Galois on CPU, Gunrock on GPU) when:
    - Insertions ≥ 50% of the total changed edges.
- However, if deletions exceed 50%, recomputing becomes more efficient.

Future Work:

1. Hybrid Decision Framework:
    - Automatically decide whether to update or recompute based on the type of changes in a batch.
2. Non-Random Batches:
    - Explore how non-random patterns in edge changes affect performance.
3. Predictive Algorithms:
    - Use prior knowledge of changed edges to optimize updates in advance.