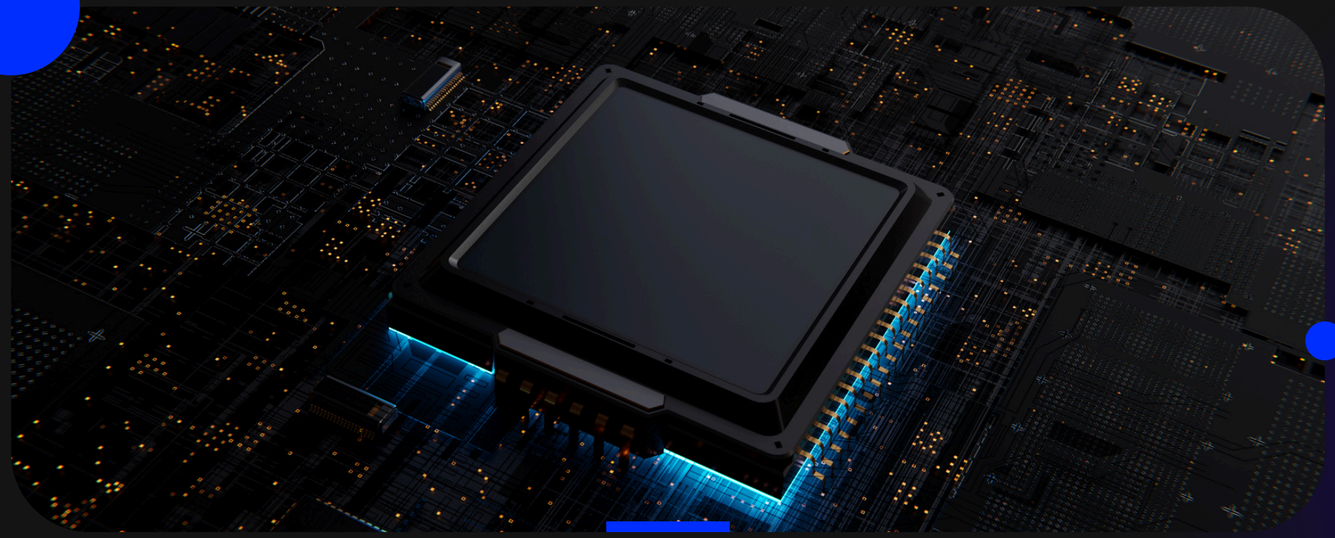
Advancements in Single Source Shortest Path Algorithms

Exploring Challenges, Implementations, and Future Directions in Dynamic Networks



Team :

Mahad Rehman

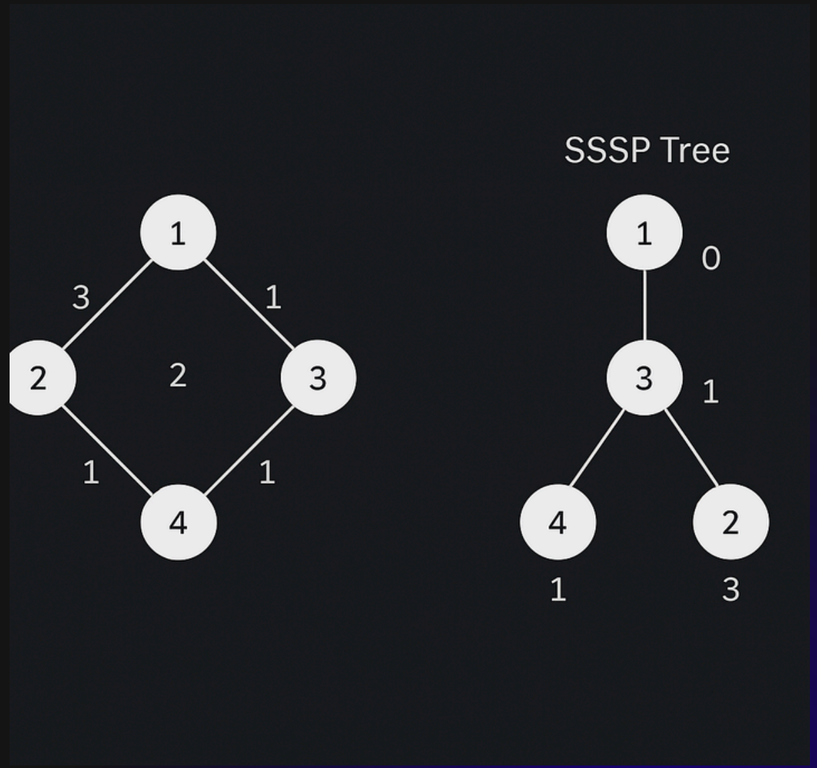
Aniq Noor

Asim Iqbal

Table of Contents

Introduction to Single Source Shortest Path (SSSP) Problem

- The Single-Source Shortest Path (SSSP) problem involves finding the shortest paths from a given source node to all other nodes in a weighted graph.
- The goal is to determine the minimum total cost or distance required to reach each node from the source, considering edge weights.



Introduction to Single Source Shortest Path (SSSP) Problem

Graph Theory Fundamentals

The Single Source Shortest Path (SSSP) problem is a key concept in graph theory.

Dynamic Network Challenges

Dynamic networks require efficient algorithms to manage structural changes.

Applications in Bioinformatics

SSSP algorithms are used to analyze biological networks.

Social Network Analysis

SSSP helps understand relationships and influence in social networks.

Transportation Network Optimization

SSSP is essential for optimizing routes in transportation systems.

Challenges in Dynamic Networks

Recomputing is Costly

- Dijkstra reprocesses the full graph after every change
- Wastes time when only a few nodes are affected

Blocking in MPI

- MPI processes wait at sync points
- Causes delays and underused processors

High Sync Overhead

- Frequent thread sync slows down shared-memory systems
- Leads to contention and stalls

Load Imbalance

- Some subtrees are large, others small
- Uneven work causes bottlenecks

Parallel Algorithm Framework for SSSP

Affected Subgraph Identification

The first step involves identifying the subgraphs affected by changes in the network.

Changed edges are processed in parallel to identify the affected subgraphs.

Iterative Update Method

The algorithm employs an iterative method to update the distances of affected vertices.

This method ensures that all affected vertices reach their minimum distances from the source.

Dynamic Network Adaptation

The framework updates only the relevant portions of the SSSP tree based on changes in the network.

This allows for efficient adjustments without recomputing the entire tree.

Scalability Challenges

Scalability is a key concern due to the varying sizes of affected subgraphs.

Effective load balancing techniques are essential for efficient processing across multiple units.

Synchronization Techniques

The algorithm minimizes the need for expensive synchronization constructs.

It allows concurrent updates without requiring locks, minimizing synchronization costs.

Graph Partitioning with METIS

Why METIS?

- Balances vertex load across MPI ranks
- Minimizes edges cut ⇒ reduces inter-rank communication

Workflow

- Partition:

  Run METIS on the full graph to assign each node to a partition (or MPI rank).

- Distribute:

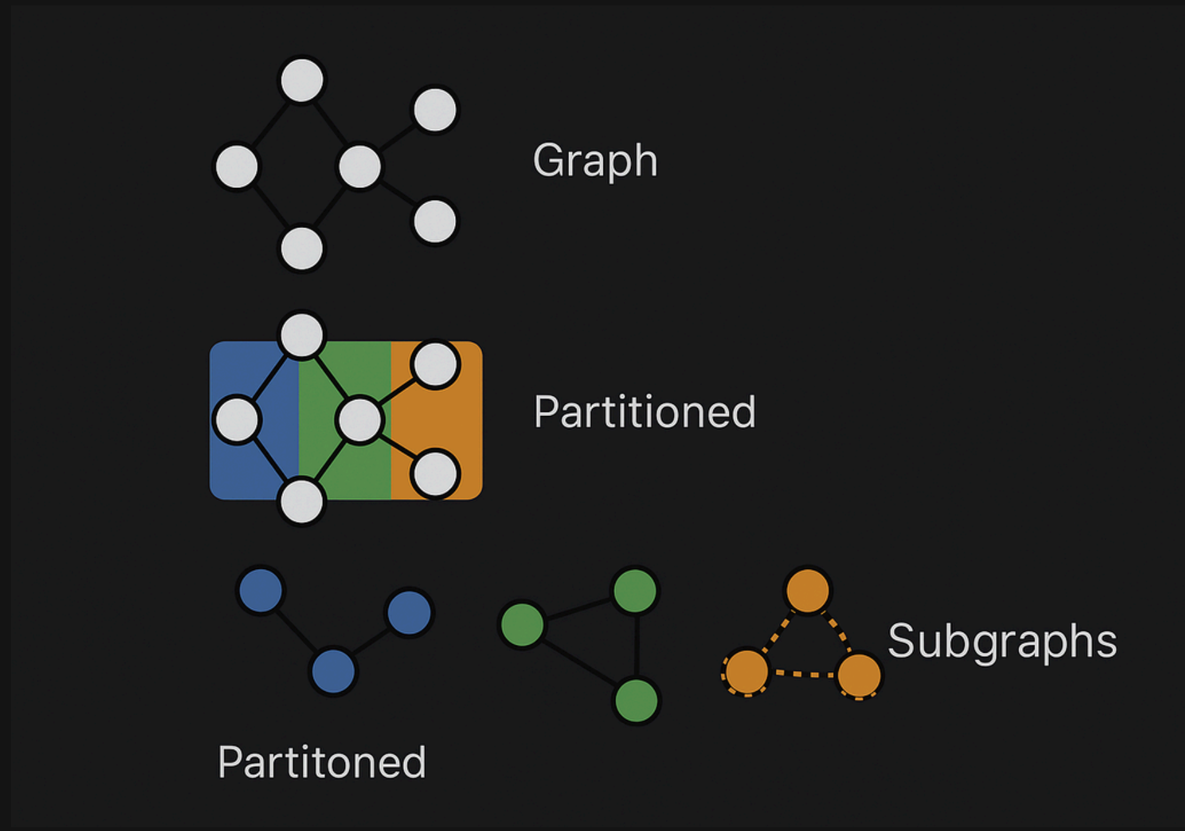
  Share the partition info (part[] array) with all MPI processes.

- Build Local Subgraphs:

  Each MPI rank keeps:

  → Its own assigned nodes

  → Neighboring nodes from other ranks (as halo nodes) for edge updates



Shared-Memory Implementation

- Implemented using #pragma omp parallel for
- Dynamic scheduling to balance load across threads
- Supports large update batches efficiently

OpenMP Snippet

```
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < changed_edges.size(); i++) {
 process_edge(changed_edges[i]);
}
```

Speedup Metrics

Achieved 8.5x speedup with 50 million edges and over 25% insertions.

GPU implementation

Speedup Metrics

1

Achieved a speedup of 5.6x compared to Gunrock with higher edge insertions.

Execution Time Analysis

2

Execution time decreases significantly with higher edge deletions.

Edge Insertion Impact

3

Performance is optimal when edge insertions exceed 50%.

Performance Comparison

4

The proposed algorithm consistently outperforms traditional recomputation methods.

Scalability Results

5

Scalability experiments show reduced update times with increased thread counts.

Comparison of CPU and GPU Implementations

Shared-Memory (CPU - OpenMP)

- Performs better with high edge deletions
- Scales well with more threads → faster updates
- Batch processing improves load balance and reduces contention
- Execution time is sensitive to how many nodes are affected
- Easier to debug and extend for larger graph sizes

GPU (CUDA + VMFB)

- Up to 5.6× faster than CPU on insertion-heavy updates
- Performance plateaus when insertions are sparse
- Asynchrony greatly boosts performance with frequent changes
- Network structure affects asynchrony's effectiveness
- Requires tuning for best memory and kernel performance

Experimental Results

Setup Overview

Evaluations were conducted on real networks and synthetic networks.

Performance Metrics

Execution times were recorded for various configurations.

Future Directions

Future work will explore hybrid approaches and predictive algorithms.

Execution Details

The framework was implemented in a shared-memory environment.

Outcome Analysis

Speedup was significant when edge insertions exceeded 25%.

Scalability Challenges and Solutions

Scalability in SSSP Updates

This section discusses the challenges faced in dynamic networks.

Load Balancing

Dynamic scheduling is used to optimize workload distribution in SSSP updates.

Synchronization Issues

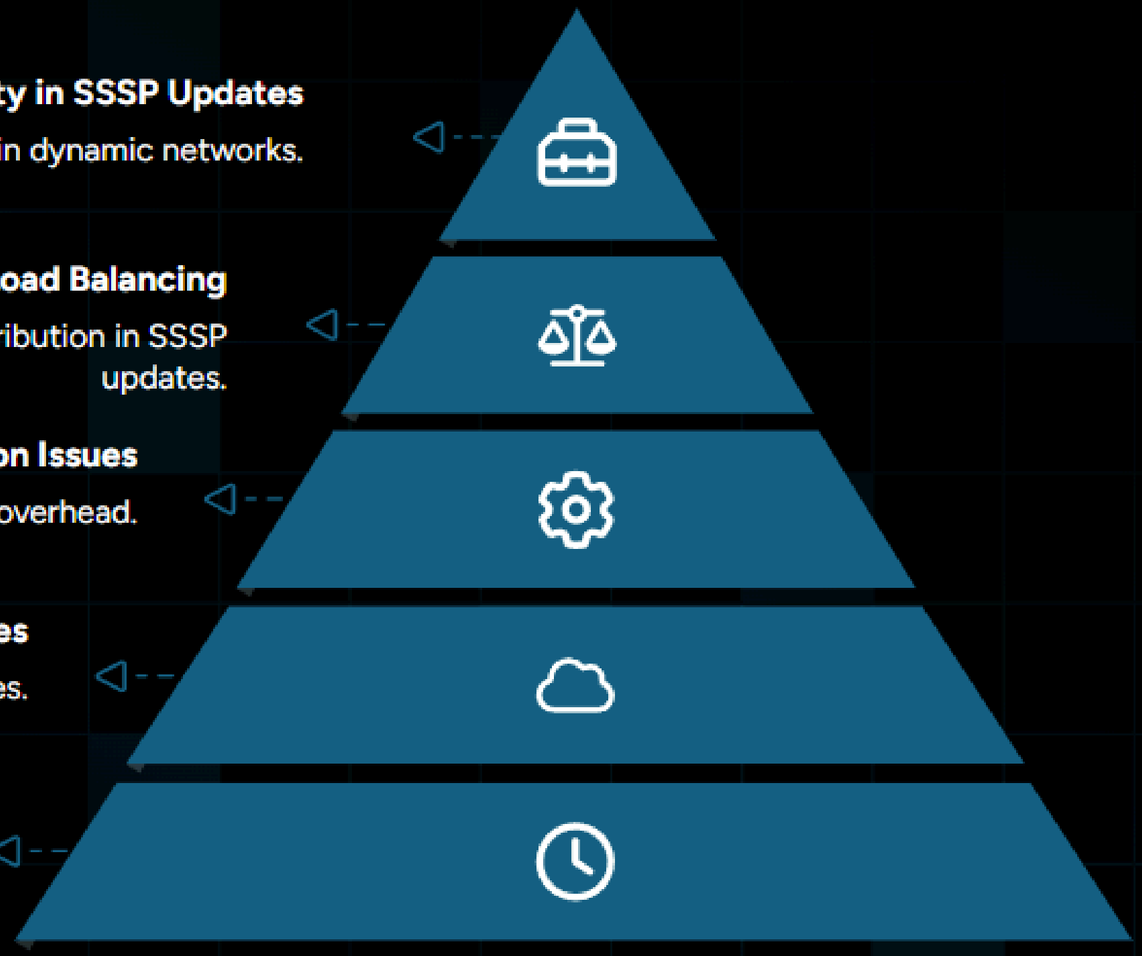
Iterative methods are employed to minimize synchronization overhead.

Asynchronous Updates

Asynchronous processing enhances the efficiency of updates.

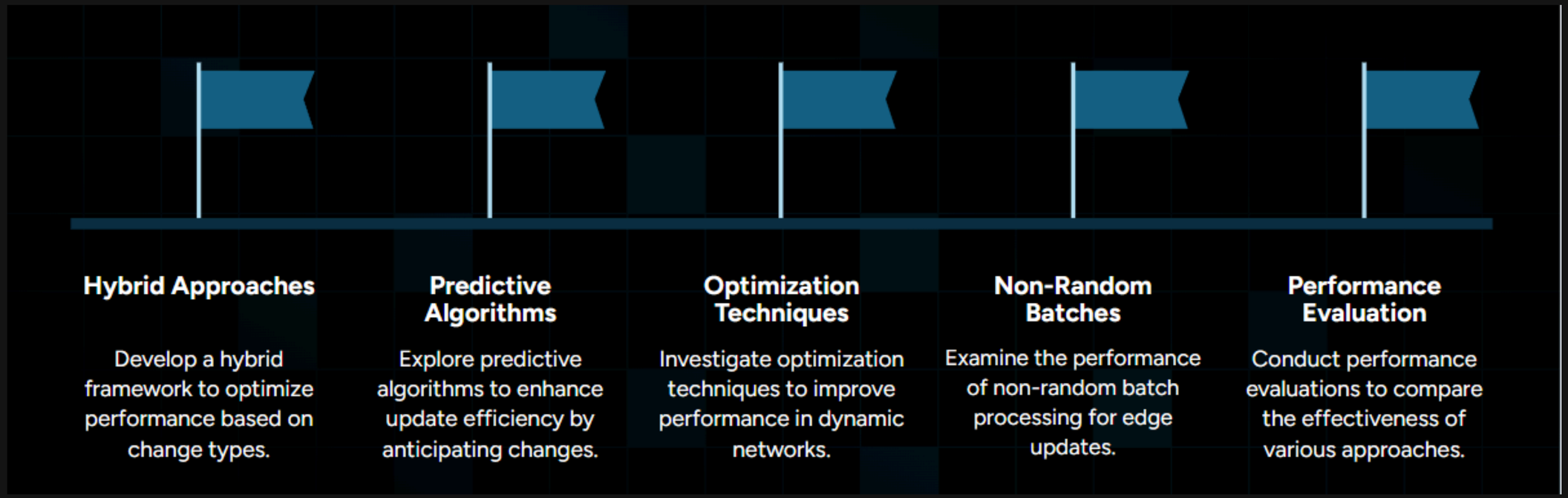
Dynamic Scheduling

Dynamic scheduling improves overall scalability with reduced overhead.



Future Work and Extensions

- Implement MPI + OpenMP hybrid model
- Use METIS for efficient graph partitioning
- Analyze scalability with real datasets
- Test performance on LiveJournal, Orkut, and synthetic graphs
- Use MPI analyzers for profiling and bottleneck detection



Conclusion

- Developed a novel parallel framework for efficiently updating SSSP in dynamic graphs
- Uses an iterative approach to update only affected nodes, avoiding full recomputation
- Minimizes synchronization using level-based batching and asynchrony
- Achieves strong scalability on both CPU (OpenMP) and GPU (CUDA) platforms
- Outperforms traditional methods, especially with large update batches and insertion-heavy changes