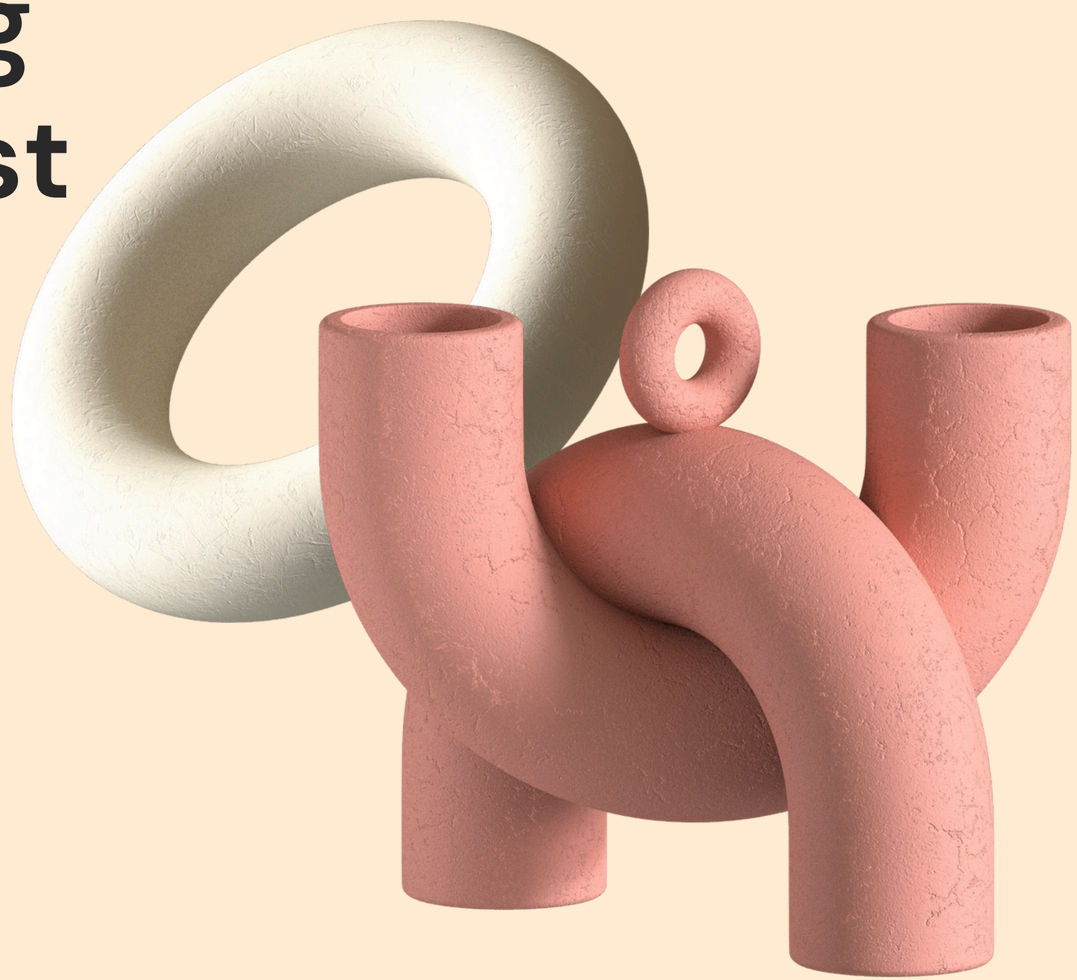
A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks



Brought to you by
Hashir Khan 22i-0754
Hania Aamer 22i-1154
Eman Hasan 22i-0745

Problem 101

What to Know

Background: SSSP and Dynamic Networks

Single-Source Shortest Path (SSSP)

- Computes shortest paths from a source vertex to all other vertices
- Output is a spanning tree (SSSP tree) rooted at the source
- Applications: transportation, communication, social networks
- Building block for other network properties (closeness, betweenness centrality)

Dynamic Networks

- Networks that change over time (edge/vertex insertions and deletions)
- Changes can significantly impact shortest paths
- Problem: Update SSSP efficiently without full recomputation

The Challenge of Dynamic Networks

Real-world networks are dynamic with structures changing over time

Traditional SSSP algorithms operate on static graphs

Recomputing SSSP from scratch after each change is inefficient

Traditional SSSP algorithms operate on static graphs

Need for efficient algorithms to update SSSP as network structure changes

Paper's Approach and Framework

STEP 1: IDENTIFY AFFECTED SUBGRAPHS

Process each changed edge in parallel
For edge deletion:

- Check if edge is part of SSSP tree
- If yes, disconnect child vertex (set distance to infinity)
- Mark as affected by deletion

For edge insertion:
- Check if new edge provides shorter path
- If yes, update parent and distance
- Mark as affected by insertion

STEP 2: UPDATE ONLY THESE SUBGRAPHS

Part 1: Update disconnected subtrees

- For vertices affected by deletion, traverse their subtrees
- Set distances to infinity
- Mark as affected

Part 2: Iteratively update distances

- For each affected vertex and its neighbors, check if distances can be reduced
- Update parent-child relationships
- Continue until no more updates are possible

Implementation Challenges and Solutions

Load Balancing

Challenge : Subgraphs of different sizes lead to imbalanced workloads
Solution : Dynamic scheduling

Synchronization

Challenge : Race conditions when multiple processors update vertices
Solution : Iterative updates instead of locking constructs

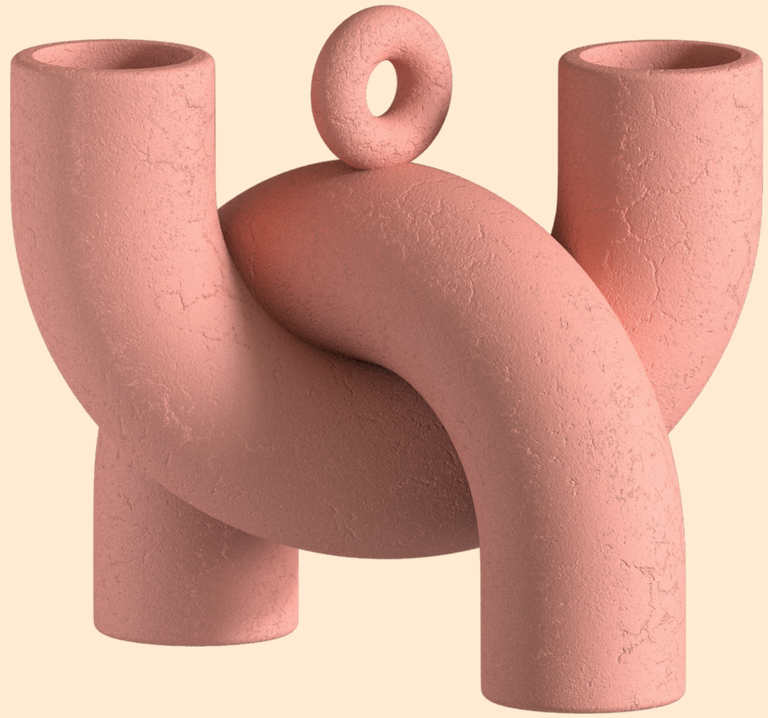
Avoiding Cycle Formation

Challenge : Parallel edge insertions can create cycles
Solution : Process deletion-affected vertices first

Implementation Details

Shared-Memory Implementation (OpenMP)

- Uses pragma omp parallel for directives
- Processes deletions before insertions
- Features asynchronous updates to reduce synchronization
- Processes changes in batches for better load balancing



GPU Implementation (CUDA)

- Leverages SIMT execution model
- Novel Vertex-Marking Functional Block (VMFB) approach
  - Minimizes atomic operations while maintaining correctness
  - Three steps: Vertex Marking, Synchronization, and Filter

Experimental Results

PERFORMANCE COMPARISON

GPU: Up to 8.5x speedup compared to Gunrock (state-of-the-art

CPU: Up to 5x speedup compared to Galois (state-of-the-art)

ADAPTIVE REPARTITIONING

Algorithm outperforms recomputation when:

- Edge insertion percentage is ≥50% of total changes
- Affected vertices are <80% of total vertices

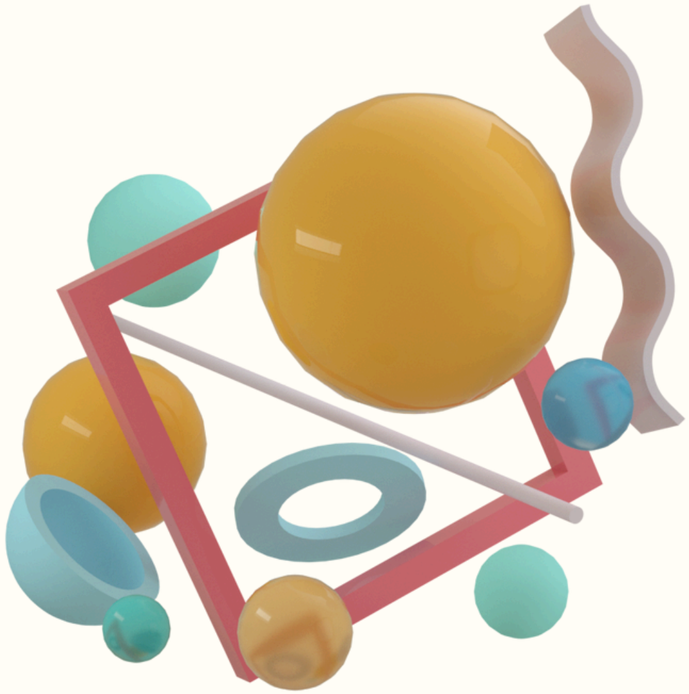
For primarily edge deletions (>50%), recomputing is more efficient

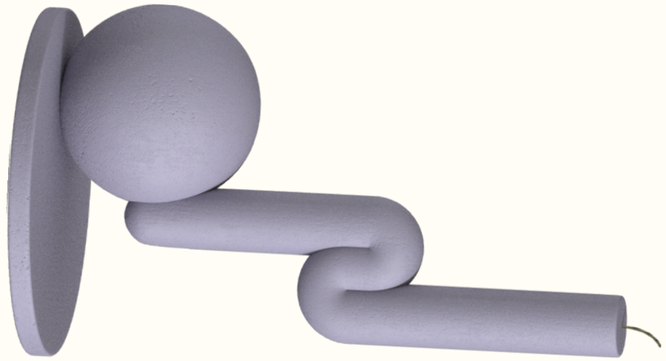
Processing in batches and asynchronous updates improve performance

Proposed Parallelization Strategy

A Hybrid Parallelization Strategy using:

- METIS for graph partitioning
- MPI for inter-node communication
- OpenMP for intra-node parallelism





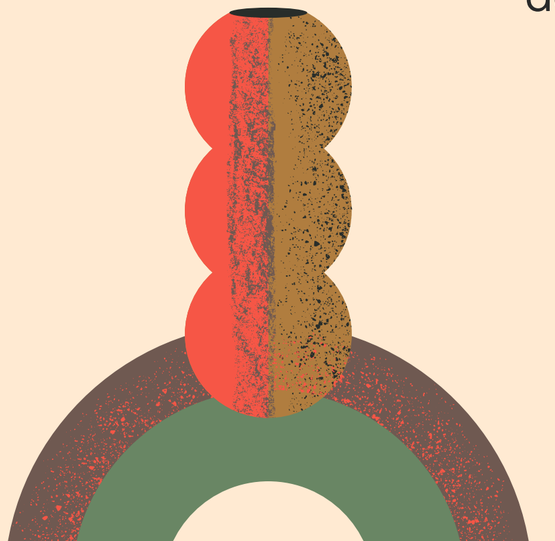
METIS for Graph Partitioning

PARTITION-BASED PROCESSING

Use METIS to partition the graph across compute nodes

Objectives:

- Balance computational load
- Minimize communication
- Preserve locality of affected subgraphs



ADAPTIVE REPARTITIONING

Periodically repartition if load becomes imbalanced
Consider affected subgraph distribution in partitioning decisions

MPI for Inter-Node Communication

PARTITION-BASED PROCESSING

Each MPI process handles a partition of the graph

Processes communicate at partition boundaries

COMMUNICATION PATTERN

Initial Distribution: Master node distributes graph partitions and changed edges
Step 1 Communication: Exchange information about affected vertices at partition boundaries
Step 2 Communication: Synchronize distance updates for boundary vertices
Final Collection: Gather updated SSSP tree

Intra-Node Parallelism

OPENMP FOR THREAD-LEVEL PARALLELISM

- WITHIN EACH MPI PROCESS, USE OPENMP FOR MULTI-THREADING
- PARALLELIZE BOTH STEPS OF THE ALGORITHM:

ADDITIONAL PARALLELISM (IF NEEDED)

- ADDITIONALLY OPENCL COULD BE USED TO PROCESS LARGE AFFECTED SUBGRAPHS
- LIKE HOW CUDA WAS IMPLEMENTED IN THE RESEARCH PAPER

Implementation Considerations

HYBRID APPROACH

Use MPI for node-level parallelism

Use OpenMP for thread-level parallelism within nodes

Use OpenCL for GPU acceleration where available

LOAD BALANCING

Dynamic work distribution based on affected subgraph sizes

Adaptive repartitioning for long-running applications

SYNCHRONIZATION STRATEGY

Minimize global synchronization points

Use asynchronous communication where possible

Conclusion

SUMMARY

- The paper presents a significant advancement in processing dynamic networks
- The parallel framework efficiently updates SSSP without full recomputation
- Implementations on both CPU and GPU show substantial performance improvements

OUR PROPOSED STRATEGY

- Extends the framework to distributed environments using MPI
- Leverages multi-level parallelism (MPI + OpenMP)
- Uses METIS for optimized graph partitioning
- Addresses key challenges in distributed dynamic graph processing