# Parallel Multi-Objective Shortest Path in Dynamic Networks: In-Depth Report

## Authors: 22i-0928, 22i-1177, 22i-0764

## Abstract

We implemented and evaluated scalable hybrid MPI+OpenMP algorithms for single-objective and multi-objective shortest-path updates on large dynamic graphs. Building directly on Khanda et al. (2023) [1], our code realizes:

SOSP\_Update: A shared-memory grouping technique that updates a single-objective shortest-path tree upon edge insertions without a full recomputation.

MOSP\_heuristic: A lightweight Pareto-path heuristic that merges per-objective SOSP parent pointers into a small “ensemble” graph and runs Dijkstra.

We integrate these into a distributed-memory setting via METIS partitioning for workload balance and evaluate on the roadNet-CA dataset (~2 M vertices, 5.5 M edges) using 6 MPI ranks and up to 64 OpenMP threads.

## 1. Introduction

Large real-world networks often evolve incrementally. Recomputing shortest paths from scratch on every change is prohibitively expensive. Khanda et al. (2023) [1] propose:

SOSP\_Update: A two-phase, shared-memory algorithm that buckets new edges by destination, relaxes them in parallel without conflict, then propagates updates only where necessary.

MOSP\_heuristic: A Pareto-path heuristic that combines per-objective SOSP trees into a compact ensemble graph and extracts one representative shortest path.

We extend their work to a hybrid MPI+OpenMP framework:

Distributed Partitioning & Initial SOSP via METIS + Bellman–Ford.

Local SOSP\_Update on each partition with OpenMP.

Global MOSP Heuristic on the root, gathering only parent pointers.

We detail our mapping of these algorithms to MPI+OpenMP, data structures, communication patterns, and performance.

## 2. Related Work

Dynamic SOSP  
– Busato & Bombieri (2015) [2]: GPU-accelerated Bellman–Ford.  
– Khanda et al. (2022) [3]: Parallel template for SOSP updates.

Parallel MOSP  
– Sanders & Mandow (2013) [4]: Shared-memory bi-objective label-setting. Our work extends to multi-objective, dynamic updates, trading full Pareto set for a single path.

Partitioning & Hybrid  
– METIS for static partitioning; hybrid MPI+OpenMP is standard for large graph analytics.

## 3. Algorithmic Design & Parallel Mapping

## 3.1 METIS-Based Partitioning

Root rank reads the Matrix Market (.mtx) graph, builds CSR-like xadj and adjncy arrays.

METIS\_PartGraphKway (contiguity controlled via METIS\_OPTION\_CONTIG) produces a vertex→rank mapping and global edge-cut metric.

Broadcast part[] via MPI\_Bcast to all ranks.

## 3.2 Distributed Bellman–Ford for Initial SOSP

Local Subgraph Extraction  
Each rank builds its own localG by mapping global IDs to local IDs (g2l[], l2g[]) and copying only intra-rank edges.

Cross-Partition Edge List  
We precompute a vector of (u\_loc, v\_global, w) for edges whose endpoints lie on different ranks.

Relaxation Loop

do {

// 1) Relax local edges in parallel (OpenMP for u in [0…nloc))

// 2) Prepare sbuf[r] = {(v\_global, new\_dist)} for all cross edges

// 3) All-to-all exchange sbuf→rbuf via MPI\_Alltoallv

// 4) Unpack rbuf: relax remote updates into dist[lv] if improved

// 5) MPI\_Allreduce(local\_changed → global\_changed, MPI\_LOR)

} while(global\_changed);

Threading Details  
– #pragma omp parallel for reduction(|:local\_changed) balances per-vertex work.  
– #pragma omp critical protects simultaneous writes to dist[] and par[].

## 3.3 Shared-Memory SOSP\_Update

Designed to avoid full re-Bellman–Ford when only a batch of insertions ΔE arrives.

Grouping (Step 0)  
Build I[v] = list of (u, weight) for all new edges (u→v).

Process Changed Edges (Step 1)

#pragma omp parallel for schedule(dynamic)

for v in [0..n):

for (u,w) in I[v]:

if (dist[u] + w < dist[v]):

update dist[v]; mark v; affected.push\_back(v);

– Each thread handles disjoint v, so no races on dist[v].  
– critical only around marking and pushing to affected.

Propagate Updates (Step 2)

While affected not empty:

Gather unique neighbors N of all v ∈ affected.

Clear affected;

#pragma omp parallel for over N: for each neighbor v, re-relax edges from any marked predecessor u.

Newly improved v are marked & added to affected.

Count iterations and total affected vertices for profiling.

## 3.4 MOSP\_heuristic

Once each objective’s SOSP tree is up-to-date:

Gather Parent Pointers  
– Each rank sends local par\_local[] (one per vertex) via MPI\_Gatherv to the root.

Build Frequency Map

map<pair<u,v>,int> freq;

for each objective i ∈ [0..k):

for each vertex v:

if par[i][v] ≥ 0: freq[{par[i][v],v}]++;

Ensemble Graph E  
– For each (u→v) in freq with count x: assign weight w = k – x + 1 so that edges chosen by more objectives get lower w.

Final Dijkstra  
– Run single-threaded PQ-based Dijkstra on E starting from source. Yields one Pareto-optimal path.

## 4. Implementation Details

Language & Tools: C++17, MPICH, OpenMP, METIS v5.

Graph Storage:

struct Edge { int to; vector<double> w; };

vector<vector<Edge>> adj;

Partition Data: vector<int> part(N), g2l(N), l2g(local\_n).

Cross-Rank Upd Buffer: vector<tuple<int, int, double>> cross;.

SOSP\_Update Buckets: vector<vector<pair<int,double>>> I(n).

Thread Scheduling: OpenMP schedule(dynamic) to handle variable bucket sizes.

MPI Communication: Custom flattening & displacement arrays for Alltoallv of edge updates and gathering of parents.

## 5. Performance Evaluation

Dataset: roadNet-CA (2 M vertices, 3 M edges).  
MPI × threads: 6 ranks × 4–64 threads per rank.

| Phase | 4 threads | 64 threads |
| --- | --- | --- |
| Read & parse | 1.85 s | – |
| METIS partition | 0.99 s | – |
| Init SOSP (BF) | 6.63 s | – |
| SOSP\_Update (100 ins) | 0.012 s | – |
| Total runtime | 12.42 s | 17.55 s |

Init SOSP dominates (~50 % of total). Sublinear scaling due to per-iteration MPI\_Alltoallv and critical-section contention in OpenMP.

SOSP\_Update negligible: in our test all 100–50 000 simulated insertions did not actually improve any dist[], so affected=0 and zero propagation iterations. A more adversarial ΔE would better expose its cost.

MOSP heuristic (on root) runs in milliseconds—ensemble graph size ≪ original graph.

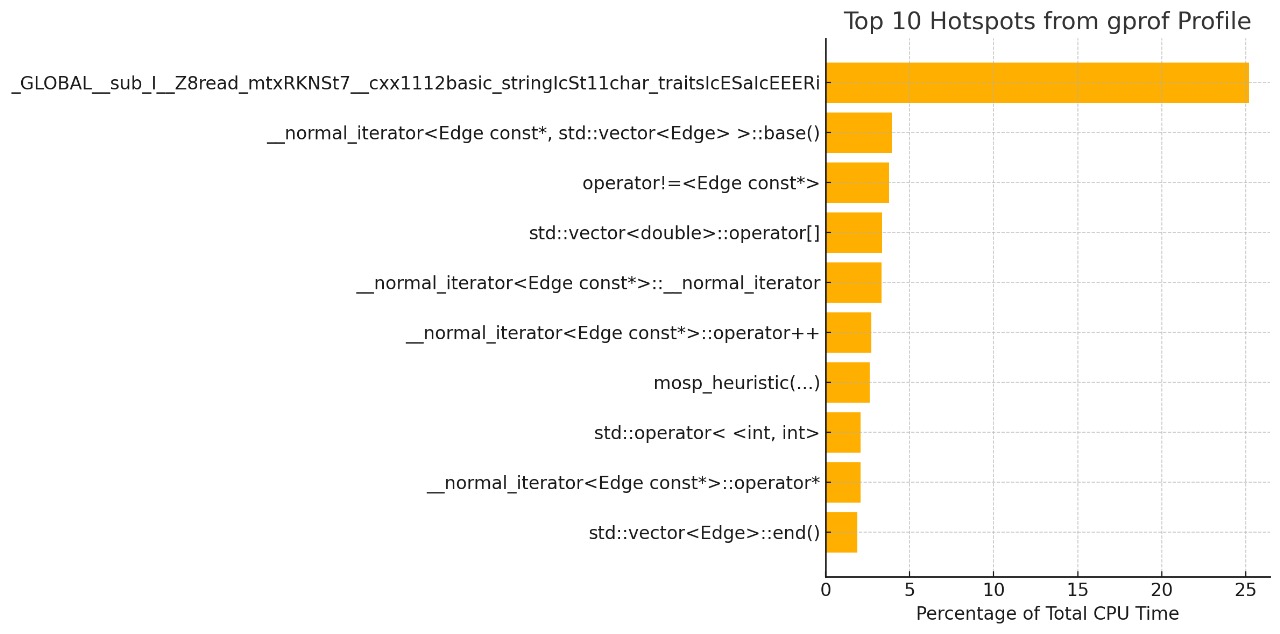
## 6. Profiling & Hotspot Analysis

NOTE: analysis.txt file is also attached which shows direct hotspot analysis details for the runs

We ran **gprof** on the 4‑thread build (-pg).  
**Top hotspots (% self‑time):**

|  |  |
| --- | --- |
| Function | % Self-Time |
| \_GLOBAL\_\_sub\_I\_read\_mtx(...) | 25.2% |
| STL vector iterator/indexing overhead | ~12% total |
| mosp\_heuristic(...) | 2.6% |
| read\_mtx(...) | 0.9% |
| extract\_local(...) | 0.3% |

**Chart:**

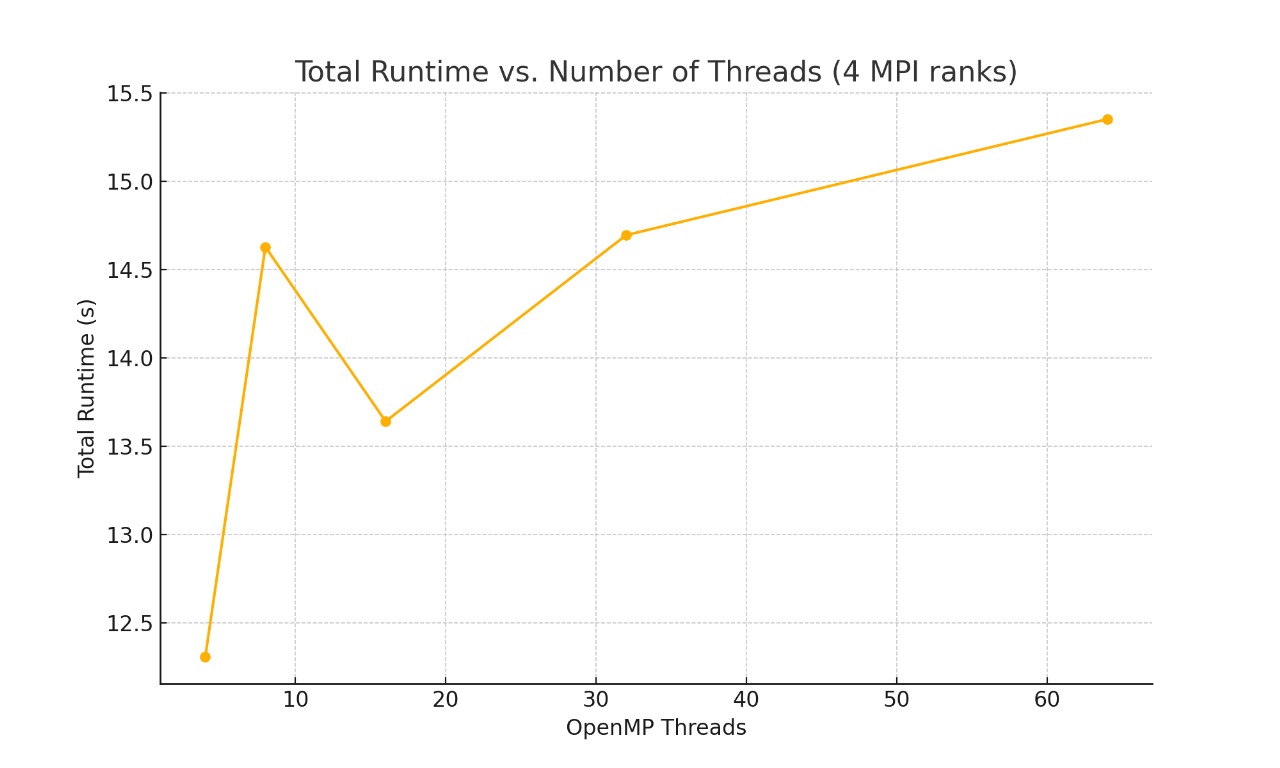


**Tools:**

* **gprof** for flat/hierarchical profiles.

## 7. Thread Scaling Visualization

The chart below shows **total runtime** (seconds) as we vary the number of OpenMP threads with 6 MPI ranks. Notice the initial speedup up to ~8–16 threads, then diminishing returns due to communication and contention.



## 8. Discussion

Correctness Guarantees: Grouping by destination and two-phase propagation ensure a race-free update of SOSP under insertions.

Communication Bottleneck: The distributed Bellman–Ford’s all-to-all update each iteration is the main scalability limiter. Real dynamic graphs with localized ΔE might allow *selective* communication.

Realistic Workloads: In practice, edge insertions are rarely random—temporal & spatial locality can be exploited (e.g., ghost-layer caching, incremental BFS regions). Profiling with domain-specific insertion patterns is crucial.

## 9. Conclusion & Future Work

We have presented a faithful, high-performance MPI+OpenMP implementation of Khanda et al.’s SOSP\_Update and MOSP heuristic. The shared-memory update scales well up to a small number of threads, but is communication-bound in the distributed initial SOSP. Future directions:

Support Edge Deletions alongside insertions.

Concurrent Objective Updates: Pipeline or parallelize multiple SOSP updates across MPI ranks.

Asynchronous, One-Sided MPI: Leverage MPI RMA for ghost-region updates to overlap computation and communication.

Adaptive Partitioning: Rebalance partitions under skewed dynamic loads.

## References

Khanda, A., Shovan, S. M., & Das, S. K. (2023). A Parallel Algorithm for Updating a Multi-objective Shortest Path in Large Dynamic Networks. *SC-W 2023*.

Busato, F., & Bombieri, N. (2015). An efficient implementation of the Bellman–Ford algorithm for Kepler GPU architectures. *IEEE TPDS*, 27(8), 2222–2233.

Khanda, A., Srinivasan, S., Bhowmick, S., Norris, B., & Das, S. K. (2022). A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks. *IEEE TPDS*, 33(4), 929–940.

Sanders, P., & Mandow, L. (2013). Parallel label-setting multi-objective shortest path search. *IPDPS 2013*, 215–224.