A Parallel Algorithm for Updating a Multi-objective Shortest Path in Large Dynamic Networks

Faris Ali _ 22I-0804

Faiz Ul Hasan _ 22I-0818

Jawad Ahmad Khan _ 22I-0791

Parallel and Distributed Computing Report

Github Repository

farisalicode/PDC_Project: Semester project for parallel and distributed computing course.

Overview

This paper introduces parallel algorithms designed to efficiently update shortest paths in large, dynamic networks—particularly focusing on multi-objective scenarios where multiple conflicting criteria must be optimized (e.g., distance, time, energy). The authors propose a two-step approach: first, a parallel method to update single-objective shortest paths (SOSP), and second, a heuristic leveraging SOSP updates to incrementally maintain multi-objective shortest paths (MOSPs). The algorithms are implemented in shared memory systems and tested on real and synthetic network datasets, demonstrating strong scalability and effectiveness.

Problem

The main problem addressed is the computational difficulty of updating shortest paths in large, dynamic networks where topology changes frequently (like adding new roads or network links). When optimizing for multiple objectives (MOSP), the complexity increases significantly, especially in ensuring Pareto optimality—where improving one objective shouldn't worsen another. Recomputing paths from scratch with every change is inefficient and computationally expensive, making existing methods inadequate for real-time or large-scale applications.

Solution

The paper solves the problem by introducing a parallel algorithm that incrementally updates single-objective shortest paths (SOSP) instead of recalculating them. This is then extended to update MOSPs using a heuristic approach built on the SOSP results. The algorithms are designed for parallel execution in shared memory environments, significantly speeding up the process and scaling well with larger networks. This method avoids full recomputation, adapts to changes efficiently, and maintains multi-objective optimization through approximate Pareto-optimal updates.

Serial Implementation Strategy

Initially a dataset is used to generate the starting state of the graph; each line of the dataset is an edge with k objective values. The edge list is used throughout the program to access these objective values.

Two major functions of the program are:

1. **make_static_mosp**
2. **make_dynamic_mosp**

make_static_mosp:

K loop iterations are executed to generate K Single Objective Shortest Paths(SOSPs) across the Graph based on k different objectives. Then using the weight assignment expression k-x+1, we make a combined graph that has weights(costs) on each edge and then we run dijkstra again to make an SOSP out of the combined graph. This is the final Multiple objective Shortest Path(MOSP).

The edge list is used to get cost vectors for k objectives in order to compute and display the cumulative costs on nodes coming from the source node.

The SOSPs are preserved because these will be reused when edge changes occur, this is the crux of this respective research paper.

make_dynamic_mosp:

After making edge changes(currently through commands in the main function) the make_dynamic_mosp function is called to fix the graph. The preserved SOSPs are called and the edge changes are used to identify the affected vertices.

Only the parents of affected vertices are recalculated(this happens K times; K objective trees), others are not recalculated. This is referred to as modification instead of recomputation from scratch. Then after SOSPs are updated the MOSP is calculated the same way.

MPI Implementation Strategy

Using the serial approach, we offload the computation of different SOSPs, as the paper suggests; to different nodes using MPI. The calculation of SOSPs is independent of each other, hence can be done without any communication.
Initially while creating the MOSP graph, when we calculate SOSPs from scratch, these are done by different ranks, once rank 0(MASTER) broadcasts the edge list to all other nodes. This may seem inefficient and cause overhead but without the entire graph, no SOSP can be calculated for any objective.

Similarly, in modifying the graph after changes, the SOSPs are again modified by different nodes.

The results of this approach are shown below in graphs in detail, in a nutshell it helped us remove this burden from a single node, the only tradeoff was the communication that takes place afterwards for the MASTER to get back the SOSPs from all other nodes.

MPI + OPENMP Implementation Strategy

According to the technique provided in the research paper about using different threads for each affected node, we have used openmp there to parallelize loops.

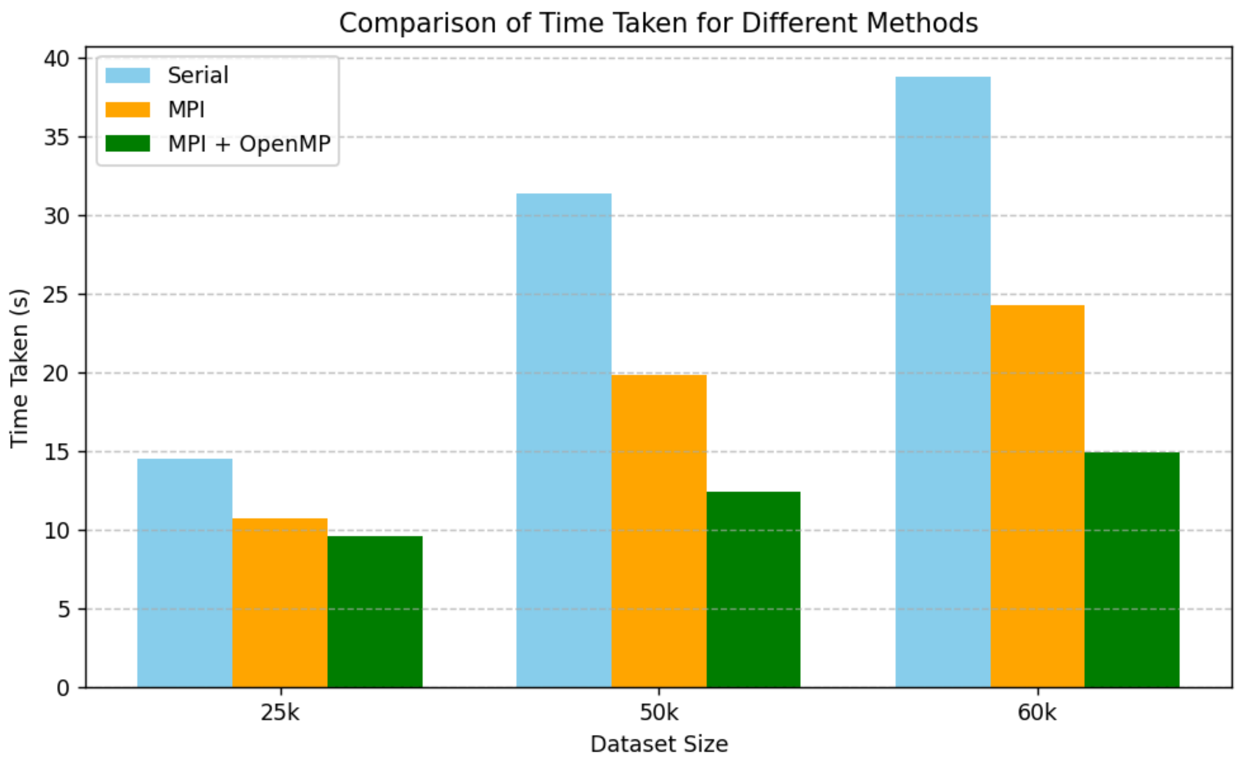
In make_dynamic_mosp, when we offload each SOSP to a different process, within that affected nodes are identified by selecting the destination of affected edges. All such unique vertices are handled independently, and then their neighbours and then so on, until no affected nodes are left. Here we used openmp to distribute these affected nodes to different threads so we can do these calculations in parallel.

The results of this approach proved slightly positive, but it highly depends on the dataset and the changes made afterwards. If the changes on a single node are too frequent, openmp will prove highly useful. Nevertheless, the results of our approach are shown as MPI+ in the following graphs.

Analysis

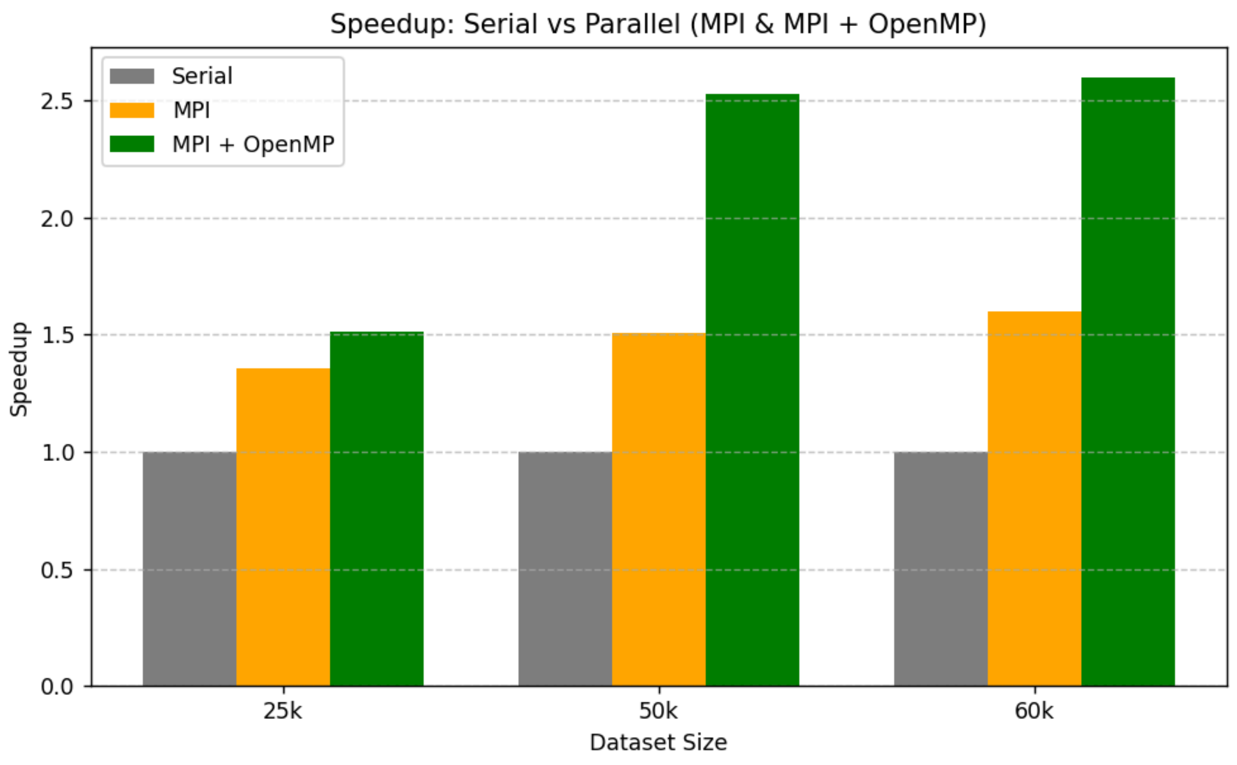
Time taken Serial vs Parallel

	Time Taken (s)		
Dataset	Serial	MPI	MPI + OpenMP
25k	14.5	10.7	9.59
50k	31.4	19.8	12.4
60k	38.8	24.25	14.9



Speed Up Serial vs Parallel

	Speedup		
Dataset	Serial	MPI	MPI + OpenMP
25k	1	1.355	1.511
50k	1	1.509	2.53
60k	1	1.6	2.6


Speedup: Serial vs Parallel (MPI & MPI + OpenMP)
Serial
MPI
MPI + OpenMP
Speedup
25k
50k
60k
Dataset Size