National University
Of Computer and Emerging Sciences

“Semester Project Report”

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

**In partial fulfillment
of the requirement for the course of**

***“Parallel and Distributed Computing”
CS-3006***

By

**Raima Tariq | 22I-1145_C
Rimsha Azam | 22I-1129_C
Jawairia Waseem | 22I-1274_G**

Table of Contents

Strategy for Updating Single-Source Shortest Paths in Dynamic Networks

This project implements an efficient and parallel strategy for updating Single Source Shortest Paths (SSSP) in large-scale dynamic graphs where edge insertions and deletions occur over time. The approach avoids recomputation from scratch and leverages OpenMP and MPI+METIS to enhance performance on multi-core and distributed systems.The implementation was developed and tested on a dual-boot Ubuntu setup for serial part and on cluster for MPI program and performance was profiled using Unix Time command.

Two-Phase Algorithmic Approach

Our implementation employs a two-step approach for efficiently updating Single-Source Shortest Paths (SSSP) after network changes:

Step 1: Changed Edge Processing

The process_changed_edges function identifies vertices directly affected by edge modifications:

- **For deletions**: When an edge in the SSSP tree is removed, the child vertex has its distance set to infinity, its parent relationship cleared, and is marked for further processing.
- **For insertions**: When a new edge creates a shorter path, the vertex with the higher distance is updated with the new shorter distance, its parent pointer is adjusted, and it's marked as affected.

Step 2: Affected Vertex Propagation

The update_affected_vertices function handles the cascading effects in two sequential phases:

- **Deletion Propagation**: Iteratively processes vertices affected by deletions, invalidating all downstream vertices in the SSSP tree that relied on these paths.
- **Distance Update**: Iteratively processes all affected vertices, examining their neighborhoods to find shorter paths until no further improvements are possible.

Implementation Variants

Serial Implementation (Baseline)

Key Features:

- Uses Dijkstra's algorithm for initial SSSP computation.
- Processes edge updates sequentially.
- Simple and easy to verify correctness.

Data Structures:

- Adjacency list for graph representation.
- Arrays for distance, parent, and affected vertex tracking.

OpenMP Parallel Implementation

Key Features:

- Parallelizes edge processing and vertex updates using OpenMP directives.
- Uses critical sections to protect shared data (distance/parent updates).
- Dynamic scheduling for load balancing.

Optimizations:

- Batched processing of affected vertices.
- Reduction operations for convergence detection.
- Thread-safe priority queue for initial SSSP computation.

MPI + METIS Distributed Implementation

Key Features:

- Uses METIS for graph partitioning to distribute workload across MPI processes.
- Each MPI process handles a subset of vertices.
- Collective communication (`MPI_Allreduce`, `MPI_Bcast`) for synchronization.

Optimizations:

- Local priority queues for initial SSSP computation.
- Custom reduction for parent-distance consistency.
- Efficient serialization of adjacency lists for MPI communication.

Performance Optimization Techniques

Data Structure Optimization

Graph represented as adjacency lists for efficient neighbor traversal.

Separate tracking arrays for deletion-affected vs. generally affected vertices.

Parallelization Techniques

OpenMP:

- Parallel initialization of distance and tracking arrays.
- Critical sections for safe updates.
- Reduction operations for convergence.

MPI + METIS:

- Graph partitioning minimizes edge-cut.
- Local computation with periodic global synchronization.

Load Balancing

- **OpenMP:** Dynamic scheduling via work collection vectors.
- **MPI:** METIS ensures balanced partitions.

Adaptive Execution

- Uses Dijkstra's algorithm for initial SSSP computation.
- Switches to custom propagation for incremental updates.

Performance Analysis

Serial vs. OpenMP vs. MPI Comparison

Serial Implementation:

- Simple, no parallel overhead.
- Suitable for small graphs.

OpenMP Implementation:

- Better utilization of multi-core CPUs.
- Overhead from critical sections and thread management.

MPI + METIS Implementation:

- Scales to large graphs by distributing computation.
- Higher communication overhead but better CPU utilization.

Time Profiling

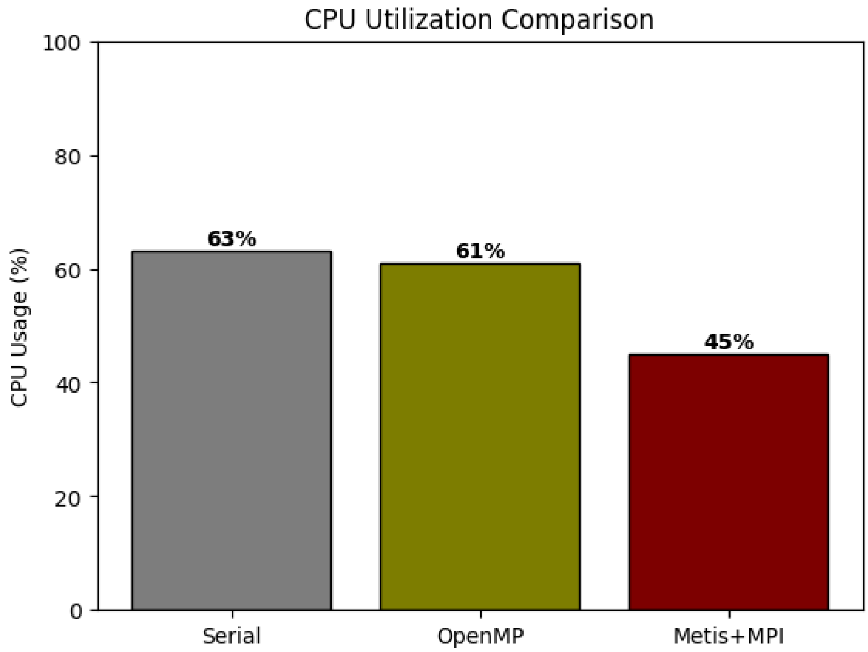
The table below provides a comparative performance analysis of different implementations of the same code: a purely sequential version, an OpenMP-parallelized version, a METIS-based partitioned version, and a combined METIS + MPI implementation. Performance metrics were collected using the Unix `time` command, capturing details such as user and system CPU time, elapsed time, CPU utilization, memory consumption, I/O operations, and page faults. This profiling enables a comprehensive evaluation of computational efficiency, resource utilization, and performance bottlenecks, highlighting the effects of parallelization and graph partitioning across implementations.

Features	Serial	OpenMP	Metis+MPI
Real	0.369	0.593	2.262
User	0.030	0.99	1.05
System	4.130	6.325	2.21
Elapsed	6.60	11.88	7.25
CPU that the Job Got	63%	61%	45%
Major Page Faults	0	1	0
Minor Page Faults	145	169	3532
Swaps	0	0	0

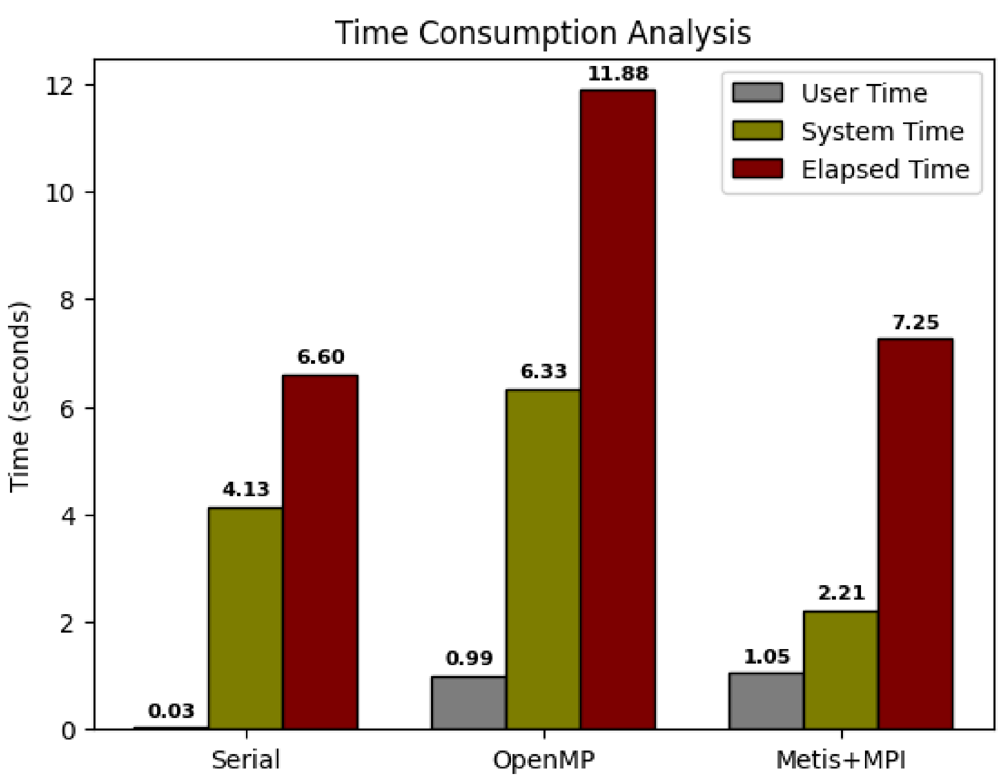
- /usr/bin/time -v ./exe
- time -v .exe

Visual Analysis

CPU Utilization



Time Consumption Analysis



Conclusion

The project successfully implements three variants of dynamic SSSP updates:

1. Serial version as a baseline.
2. OpenMP version for shared-memory parallelism.
3. MPI + METIS version for distributed computation.

Key Findings

- OpenMP improves performance on multi-core systems but requires careful synchronization.
- MPI + METIS scales better for large graphs but introduces communication overhead.
- The two-phase approach efficiently handles dynamic updates without full recomputation.