Project Report

for

CS-3006: PARALLEL & DISTRIBUTED COMPUTING

Parallel SSSP

Ali Hamza Azam, I222126
Zain Bin Asad, I221240
May 6, 2025

Abstract

This project implements a dynamic Single-Source Shortest Path solver that supports four execution modes—serial, OpenMP, MPI, and hybrid MPI+OpenMP—on weighted, undirected graphs that undergo batches of edge-weight updates. We describe the algorithmic design, data structures, parallelization strategies, and communication patterns for each mode, then present a comparative performance study on shared- and distributed-memory platforms.

Introduction

This report details the design and implementation of a multi-paradigm solution for the Single-Source Shortest Path (SSSP) problem in dynamic networks, which combines various parallel processing techniques to achieve optimal performance across different computational scenarios.

The project comprises four primary implementations:

Sequential Implementation:

The first implementation provides a baseline approach using Dijkstra's algorithm, which recomputes shortest paths from scratch after each network change. This serves as a reference point for evaluating the efficiency of parallel solutions and demonstrates the fundamental algorithmic challenges in dynamic SSSP computation.

OpenMP-Based Parallel Implementation:

The second implementation leverages shared-memory parallelism through OpenMP to accelerate SSSP computation. It employs asynchronous updating strategies that minimize synchronization overhead while maintaining correctness, enabling significant performance improvements for moderately sized networks through efficient thread-level parallelism.

MPI-Based Distributed Implementation:

The third module implements a distributed algorithm using MPI to compute shortest paths across process boundaries. This approach addresses scalability by partitioning the graph and distributing workload across multiple processes, though it faces challenges related to communication overhead and data dependencies inherent in graph algorithms.

Hybrid (MPI+OpenMP) Implementation:

The final implementation combines distributed and shared-memory parallelism to capitalize on multi-level hardware parallelism. This hybrid approach attempts to balance the strengths of both paradigms, though its effectiveness depends heavily on problem characteristics and hardware architecture.

Background and Problem Definition

Given a weighted, undirected graph $G=(V,E,w)$ and a source s, dynamic SSSP maintains $dist[v]$ for all $v \in V$ as batches of edge-weight changes $\Delta w(u,v)$ arrive. The goal is to update $dist[]$ in $o((|V|+|E|)\log|V|)$ per change, ideally proportional to the size of the "affected" subgraph E' and V'.

Applications include:

- Traffic routing under time-varying conditions
- Real-time network monitoring
- Dynamic resource allocation

Graph Representation and Utilities

All four implementations share:

- CSR format in graph.cpp & graph.hpp (arrays xadj[], adjncy[], adjwgt[])
- Distance array dist[] and parent array parent[]
- Affected flags per vertex in a std::vector<bool>
- Utilities for loading graphs and updates in utils.cpp & utils.hpp

The driver (src/main.cpp) parses arguments to select mode, reads the initial graph (e.g. data/*.edges), reads update streams (data/*_updates.edges), then invokes one of:

- serial::run_sssp()
- parallel::run_sssp_openmp()
- mpi::run_sssp_mpi()
- mpi::run_sssp_hybrid()

Sequential Implementation

Defines a baseline for correctness and performance.

• Utilizes the standard Dijkstra algorithm to compute all shortest paths from the source “from scratch” on the initial graph.

• Implements an incremental “SingleChange” routine that ensures localized repairs are triggered only for minor edge-weight updates, rather than a complete recompute.

• Demonstrates the fundamental work and associated costs of managing dynamic updates without any parallel processing capabilities.

Design and Implementation

Dijkstra: Uses a binary heap (std::priority_queue) in serial/dijkstra.cpp.

SingleChange: In serial/update.cpp, processes each Δw(u,v):

1. Update CSR weight in place.
2. If increase/deletion: enqueue u,v as invalidated, BFS to reset and re-relax downstream vertices.
3. If decrease/insert: check direct relaxation, push into min-heap, run Dijkstra on subgraph until no improvement.

Complexity: O(|E′| + |V′| log |V′|) per change in best case; O((|V|+|E|)log|V|) worst case.

OpenMP Shared-Memory Implementation

Accelerates the incremental update process on multicore, shared-memory machines.

• Batches change events and utilizes OpenMP loops or tasks to concurrently (and asynchronously) mark affected vertices, invalidate outdated paths, and relax edges.

• The objective is to overlap work across threads and minimize barrier/ synchronization overhead.

• Demonstrates how fine-grained parallelism (vertex/edge-level) can enhance the performance of dynamic SSSP when all data resides within a single address space.

Design and Implementation

Batching & Marking: In parallel/apply_changes.cpp, atomic writes mark endpoints of weight-increase changes in affected[].

Parallel Invalidation & Relaxation: In parallel/relax.cpp:

- Iterative sweep:

```
#pragma omp parallel for reduction(|:local_changed)
for (int i = 0; i < n; ++i) {
  if (affected[i]) { … relax neighbors … }
}
```

- Loop until no thread reports a change.

Asynchronous Tasks: In parallel/tasks.cpp, spawns #pragma omp task per affected vertex, with depend(in:...)/depend(out:...) to overlap work and reduce barriers.

Tuning:

Static vs. dynamic scheduling

First-touch NUMA placement in main.cpp

Thread pinning via OMP_PLACES and OMP_PROC_BIND

MPI Distributed Implementation

Scaling SSSP to Multi-Node Clusters via Graph Partitioning

The SSSP algorithm is scaled to multi-node clusters by partitioning the graph across processes. Each rank holds a subgraph (obtained through METIS partitioning), locally applies changes, and participates in global rounds of relaxation using MPI collectives or point-to-point messages.

This approach enables the algorithm to handle very large graphs that exceed the memory capacity of a single node. It also allows for the measurement of the trade-off between increased communication and distributed compute.

Furthermore, the algorithm explores both synchronous (blocking Allreduce) and asynchronous (nonblocking Iallreduce) variants to minimize idle time.

Design and Implementation

Partitioning: On rank 0, call METIS to partition CSR (mpi/partition.cpp). Scatter subgraphs with MPI_Scatterv.

Local Structures: Each rank builds local_to_global[], boundary edge lists, and dist_local[].

Dynamic Updates: In mpi/updates.cpp:

1. Apply local changes; forward cross-partition updates via MPI_Send.
2. Exchange invalidation flags for boundary vertices.
3. Iterative relax: local Bellman-Ford sweep, then MPI_Allreduce(MPI_MIN) on ghost distances, repeat until convergence.

Synchronous vs. Asynchronous:

- Synchronous uses blocking MPI_Allreduce.
- Asynchronous uses MPI_Iallreduce overlapped with local compute.

Gathering Results: MPI_Gatherv collects dist_local[] to rank 0, mapped back to global dist[].

Hybrid MPI+OpenMP Implementation

This approach seamlessly integrates inter-node parallelism through MPI with intra-node threading enabled by OpenMP, optimizing computational efficiency.

• Within each MPI rank, shared-memory threads concurrently apply updates and relax edges, facilitating nonblocking MPI communication exchanges for boundary data.

• The objective is to maximize hardware utilization within contemporary multi-core clusters by synchronizing communication with local computations.

• This demonstration illustrates how multi-level parallelism delivers substantial end-to-end speedups for large, dynamically evolving graphs.

Design and Implementation

- Within each MPI rank, relaxations/invalidation use the same OpenMP loops/tasks as OpenMP Implementation
- **Overlap Strategy:**
  1. MPI_Iallreduce on boundary data.
  2. While pending, threads process internal vertices.
  3. After completion, threads process boundary vertices.

Configuration:

Control number of MPI ranks vs. threads per rank via mpirun -np P and OMP_NUM_THREADS=T.

Experimental Setup and Results

Hardware

The Master–Worker Paradigm :

1. **Master Process (Rank 0):**
   The master reads the FASTA file using the read_fasta function, partitions the sequence records (header–sequence pairs) and distributes them to the worker processes using explicit send/receive calls.
2. **Worker Processes:**
   Each worker receives its portion of the records via point-to-point communication. They perform local computations, including converting sequences to numeric values and computing the FFT via seq_to_numeric and fft_transform (functions also used in the serial version).
3. **Global Maximum Sequence Length:**
   Instead of using an MPI collective call like MPI_Allreduce, the code gathers the local maximum sequence lengths from all processes using point-to-point messages. The master process computes the overall maximum and sends it back to the workers. This ensures that every process computes FFTs with a common padded length (determined by next_power_of_two), maintaining consistency with the serial implementation.

4. **FFT Results Collection:**
   Each process computes FFTs for its assigned sequences and then sends these results back to the master process using point-to-point

communication. The master aggregates all FFT results, computes a distance matrix using fft_correlation, and then performs progressive alignment via the progressive_alignment function. Finally, the alignment is printed using print_alignment.

Communication Strategy

All inter-process communication in this implementation is performed using direct send and receive operations:

1. **Distribution of Records:**
   The master uses MPI_Send to distribute the sequence records (headers and sequences) to workers. Workers use MPI_Recv to receive the data, ensuring that each process has the information needed for local processing.
2. **Global Maximum Determination:**
   The master process collects the local maximum sequence lengths from each worker via MPI_Recv, calculates the overall maximum, and then broadcasts this value back to each worker using MPI_Send.

Data Partitioning & Load Balancing

All inter-process communication in this implementation is performed using direct send and receive operations:

Dataset

Results and Analysis

Performance Comparison:

Observations:

Conclusion

We built a flexible dynamic SSSP framework in C++ that efficiently updates shortest paths under four paradigms. The hybrid MPI+OpenMP approach delivers the best performance on modern clusters. Future directions include adaptive batch sizing, GPU offloading of local relaxations, experiment with MPI neighborhood collectives, and dynamic load rebalance for skewed update patterns.