Project Report: Our Journey with Parallel SSSP in Dynamic Graphs

Group Members

1. 22I-0842 Shah Abdullah
2. 22I-0903 Shayan Rauf
3. 22I-0617 Muhammad Moiz

Introduction

When we started this project, our goal was to build a parallel Single-Source Shortest Path (SSSP) algorithm for dynamic graphs—graphs that change over time with edge insertions and deletions. We had a serial implementation to start with, but we knew that for larger graphs, parallelism was the key to better performance. What we didn't expect was the rollercoaster of challenges we'd face—everything from setting up a virtual cluster to wrestling with METIS partitioning and syncing updates across processes. This report is our story of tackling those hurdles, plus an analysis of how well the parallel version performed.

Our Approach to Solving the Problem

The problem was clear: compute the shortest paths from a source vertex in a graph, then keep them updated as edges are added or removed. The serial code gave us a foundation, but scaling it up with MPI, OpenMP, and METIS was where the real work began.

The Serial Starting Point

Our serial code used Dijkstra's algorithm to find initial shortest paths, stored in `dist` and `parent` vectors. For dynamic updates, we implemented two key functions:

- **ProcessCE**: Identifies vertices affected by edge changes—setting distances to infinity for deletions and updating them for insertions.
- **UpdateAffectedVertices**: Propagates those changes through the graph, ensuring all distances stay correct.

It worked well for small graphs, but as the graph size grew (like the California road network in "california.txt"), the runtime ballooned. Parallelism was the answer.

Going Parallel

For the parallel version, we split the workload across multiple processes with MPI, added thread-level parallelism with OpenMP, and used METIS to partition the graph. Here's how we approached it:

1. **Graph Partitioning**: On rank 0, we read the graph into a CSR format and used METIS to split it into subgraphs, one per MPI process.
2. **Local SSSP**: Each process ran a parallel Dijkstra's algorithm on its subgraph using OpenMP to speed things up within the process.
3. **Dynamic Updates**: We adapted the serial update logic to work locally, with plans to sync changes across processes (though we hit some snags here—more on that later).

Overcoming Challenges

This project tested our patience and problem-solving skills. Here's how we tackled the biggest obstacles:

1. **Setting Up the Environment**
   We decided to simulate a cluster using multiple VMs on our laptop. Getting MPI, OpenMP, and METIS installed consistently across them was a nightmare—version mismatches and missing dependencies kept popping up. We spent hours configuring SSH for passwordless communication and verifying everything with simple MPI "hello world" tests. It wasn't glamorous, but it paid off when the cluster finally worked.
2. **Mastering METIS Partitioning**
   METIS was new to us, and integrating it was tough. We had to convert our graph into CSR format, call `METIS_PartGraphKway`, and then map global vertex indices to local ones for each process. Debugging partitioning errors—like uneven splits or missing edges—took trial and error, but seeing the graph split cleanly across processes felt like a win.
3. **Parallelizing Dijkstra's Algorithm**
   OpenMP sounded straightforward, but parallelizing Dijkstra's wasn't. The priority queue updates caused race conditions, so we used a critical section to safely update distances and collect new queue entries in a buffer. It wasn't perfect—there's still some overhead—but it cut down the computation time significantly.
4. **Dynamic Updates in Parallel**
   This was the trickiest part. The serial update logic relied on a global view of the graph, but in parallel, each process only sees its subgraph. We got local updates working with `ProcessCE` and `UpdateAffectedVertices`, but syncing changes across process boundaries

stumped us. We started experimenting with MPI communication to share updated distances, but time ran short. It's a work in progress we'd love to finish.

5. **Measuring Performance**
   We used MPI_Wtime for timing and mpiP to profile MPI communication, but interpreting the results took effort. Figuring out how many processes and threads worked best meant running dozens of tests and tweaking parameters. It was tedious, but the insights were worth it.

Performance Analysis

To see if parallelism paid off, we ran experiments comparing speedup and efficiency. Here's what we found:

1. Comparison of Varying MPI Processes and OpenMP Threads for Speedup

- **Setup**: We tested combinations like 2, 4, and 8 MPI processes, with 1, 2, and 4 OpenMP threads per process, using the California graph.
- **Results**: Speedup improved with more processes up to 4, then flattened due to communication overhead (tracked with mpiP). Adding threads helped too, but beyond 2 per process, the gains shrank—thread overhead kicked in.

```
Using 4 OpenMP threads
First edge: 15890709 - 4 weight 0.048009
Last edge: 9442479 - 19 weight 0.862014
Loaded graph: 16129269 vertices, 34333909 edges
Applying SSSP ALGORITHM
Initial Dijkstra time: 140.639 s
Applying it dynamically
Inserted edge: (3480498,9324533) with weight 9.34287
Deleted edge: (3480498,19) with weight 0.414035
Dynamic update time: 0.0932886 s
Full Dijkstra time: 95.5656 s
Dynamic update correct
abdullah@abdullah-VirtualBox:~/Desktop/seq$ g++ -fopenmp codemp.cpp -o c
abdullah@abdullah-VirtualBox:~/Desktop/seq$ ./c
Using 8 OpenMP threads
First edge: 15890709 - 4 weight 0.048009
Last edge: 9442479 - 19 weight 0.862014
Loaded graph: 16129269 vertices, 34333909 edges
Applying SSSP ALGORITHM
Initial Dijkstra time: 135.091 s
Applying it dynamically
Inserted edge: (903274,1900693) with weight 8.84027
Deleted edge: (903274,1900693) with weight 8.84027
Dynamic update time: 0.102656 s
Full Dijkstra time: 97.4122 s
```

2. Comparison of Varying MPI Processes and Analysis of OpenMP Threads

- **Setup**: We fixed MPI processes (e.g., 4) and varied threads from 1 to 4.
- **Results**: More threads boosted performance up to 2, then plateaued. For 4 processes, 2 threads seemed optimal—beyond that, managing threads ate into the gains.

3. Comparison of Varying MPI Processes and Analysis of MPI Processes

- **Setup**: We kept threads at 2 and varied processes from 2 to 8.
- **Results**: Speedup peaked at 4 processes; beyond that, MPI communication (logged by mpiP) slowed things down. It showed us the balance between computation and communication is delicate.

4. Analysis Without Code Profiling Tools

- **Method**: We used `std::chrono` in the serial code and `MPI_Wtime` in the parallel code to time key sections—no fancy tools, just raw data.
- **Findings**: The initial Dijkstra run was the bottleneck in both versions, but parallelizing it cut the time by nearly half on 4 processes. Dynamic updates were faster too, though our incomplete cross-process syncing limited the full benefit.

Our Project Journey

This wasn't just coding—it was a journey. Setting up VMs felt like building a house from scratch. Wrestling with METIS was like learning a new language. Parallelizing Dijkstra's had us celebrating small wins (like no more crashes) and cursing late-night bugs. The dynamic updates pushed us to our limits—we got so close to a full solution, but syncing across processes needs more time. Each challenge taught us something: patience, debugging tricks, and the power of parallelism.

Conclusion

Looking back, we're proud of what we built. The parallel SSSP implementation is faster than the serial one, especially for big graphs, and tools like mpiP showed us where to improve. We didn't solve every problem—cross-process updates still nag at us—but we learned a ton about MPI, OpenMP, and METIS. This project was tough, but it's made us better programmers.