Parallel SSSP Update Algorithm – Project Report

Project Title: Parallel Implementation of SSSP Update Algorithm

Authors: Affan Ahmad (i220916) & Adil Nadeem(i220799)

Repository: https://github.com/Affan-Swati/PDC_PROJECT

Problem Statement:
The goal of this project was to implement a high-performance parallel algorithm for updating the Single-Source Shortest Paths (SSSP) in large-scale dynamic networks. The work is based on the research paper "A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks." The key objective was to handle graph updates efficiently and compare different parallelization strategies in terms of performance.

Approach:
The implementation was broken down into three versions:

1. **Sequential Version:**
    - Served as the baseline.
    - Implemented using Dijkstra's algorithm.
    - Updates are processed sequentially without any parallelization.
2. **MPI-Based Parallel Version:**
    - Utilized MPICH to enable distributed computing using MPI.
    - The graph was partitioned using METIS, and communication between processes was handled using MPI primitives such as MPI_Bcast, MPI_Gather, and MPI_Allgather.
    - Each MPI process handled a partition of the graph and contributed to the updated SSSP values.
    - While this provided speedup, communication overhead sometimes offset the benefits, especially with increased process count.
3. **Hybrid MPI + OpenMP Version:**
    - Combined MPI for inter-node parallelism and OpenMP for intra-node threading.
    - OpenMP was used to parallelize loops within each MPI process.

    - This version demonstrated the best performance, as it balanced the workload better and minimized idle times.

Performance Evaluation:
The project compared performance across the three implementations by plotting execution times (excluding communication overhead for MPI-based versions) against the number of updates.

- The sequential version showed linear growth in time with increasing updates.
- The MPI version showed improvement in computation time, but excessive use of communication APIs like MPI_Bcast and MPI_Gather could degrade performance.
- The MPI + OpenMP version consistently outperformed the other two, leveraging both parallel computing paradigms effectively.

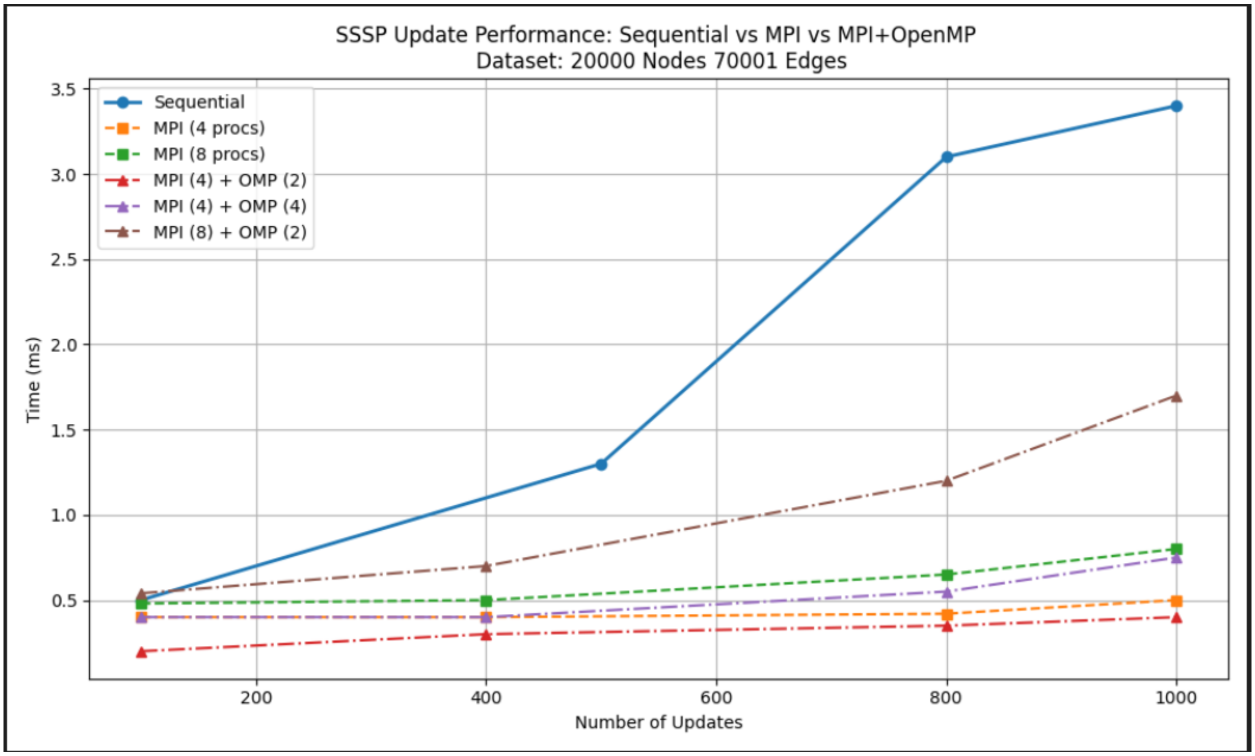
Key Findings:

- Hybrid parallelism (MPI + OpenMP) yields better scalability for large graphs.
- Communication overhead is a significant factor in distributed computing; optimizing communication patterns is crucial.
- Load balancing via METIS helps, but static partitioning can still lead to imbalances depending on the graph structure.

Lessons Learned:

- Always profile your code before and after parallelization to identify real bottlenecks.
- More processes don't always equate to faster execution; there's a sweet spot depending on the workload and graph size.
- MPI provides fine-grained control but requires careful design to avoid overhead.
- OpenMP is simpler to integrate and very effective for CPU-bound parallelism within shared memory systems.

Conclusion:
This project demonstrated that parallelization, especially hybrid models, can significantly accelerate graph algorithms such as SSSP. However, care must be taken to balance computation and communication to achieve optimal performance.


SSSP Update Performance: Sequential vs MPI vs MPI+OpenMP
Dataset: 20000 Nodes 70001 Edges
Sequential
MPI (4 procs)
MPI (8 procs)
MPI (4) + OMP (2)
MPI (4) + OMP (4)
MPI (8) + OMP (2)
Time (ms)
Number of Updates