PROJECT REPORT

Implementation and Performance Analysis of a Parallel Graph Algorithm Using MPI, OpenMP, and METIS

Rabab Alvi 22I-1338
Eraj Zaman 22I-1296
Samiya Saleem 22I-1065

Parallel Graph Algorithm Implementation and Performance Analysis

1. Project Overview

This project explores the practical implementation of parallel computing principles on graph algorithms by employing a hybrid model of MPI and OpenMP, alongside METIS for efficient graph partitioning. The goal was to select a cutting-edge research paper from a provided list and replicate a parallel version of its graph algorithm on real-world network data, ultimately evaluating scalability and performance on multiple processor configurations.

2. Selected Research and Algorithm

We based our implementation on a research study that proposed a dynamic parallel shortest-path update mechanism in evolving networks. Inspired by the challenges in adapting shortest path results when edges change in a massive graph, our solution focuses on:

- Maintaining the shortest path results dynamically.
- Leveraging parallelism for quick recomputation from affected nodes.

The key parallel components used:

- **MPI (Message Passing Interface)** for inter-process communication and workload distribution.
- **OpenMP** for intra-process parallel computation, especially in Dijkstra's shortest path calculations.
- **METIS** to partition the graph for load-balanced MPI distribution.

3. Implementation Breakdown

Graph Parsing and Preprocessing

- Graphs were ingested from `.txt` files, with support for weighted and unweighted edges.
- Invalid weights (e.g., 0 or negative) were sanitized.
- Graph mutations (edge insertions or deletions) were supported through a change file.

Graph Partitioning with METIS

- The METIS library was invoked to split the graph into partitions, one per MPI rank.
- As a fallback for METIS errors, a manual round-robin partitioning was implemented.
- Partition imbalance (difference in node count between densest and sparsest partition) was computed and recorded.

Distributed Execution with MPI

- Process 0 handled file I/O and METIS partitioning.
- The entire graph was serialized and broadcasted to other ranks.
- Each rank isolated its partition's nodes and reconstructed the local subgraph.

Parallel Shortest Path Computation

- For affected nodes (those impacted by edge changes), Dijkstra's algorithm was executed in parallel using OpenMP.
- Each rank handled a chunk of the affected nodes list.
- Source node results were calculated for final comparison and logging.

Performance Logging

Each rank measured the time taken for:

- Graph reading and initialization.
- Partitioning via METIS.
- Data broadcast and graph reconstruction.
- Dijkstra computation.
- Total execution.

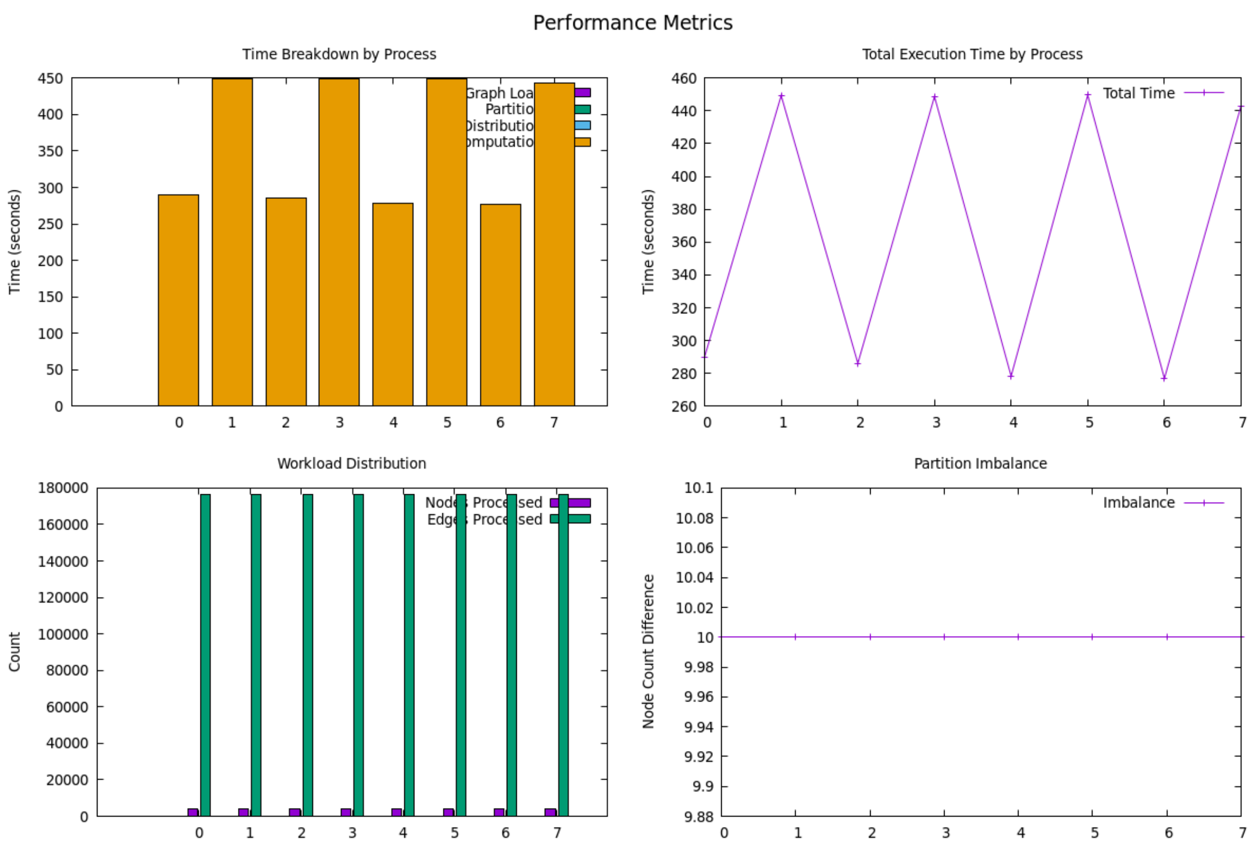
A `PerformanceMetrics` struct held these values and all ranks reported back to rank 0 for aggregation.

Visualization

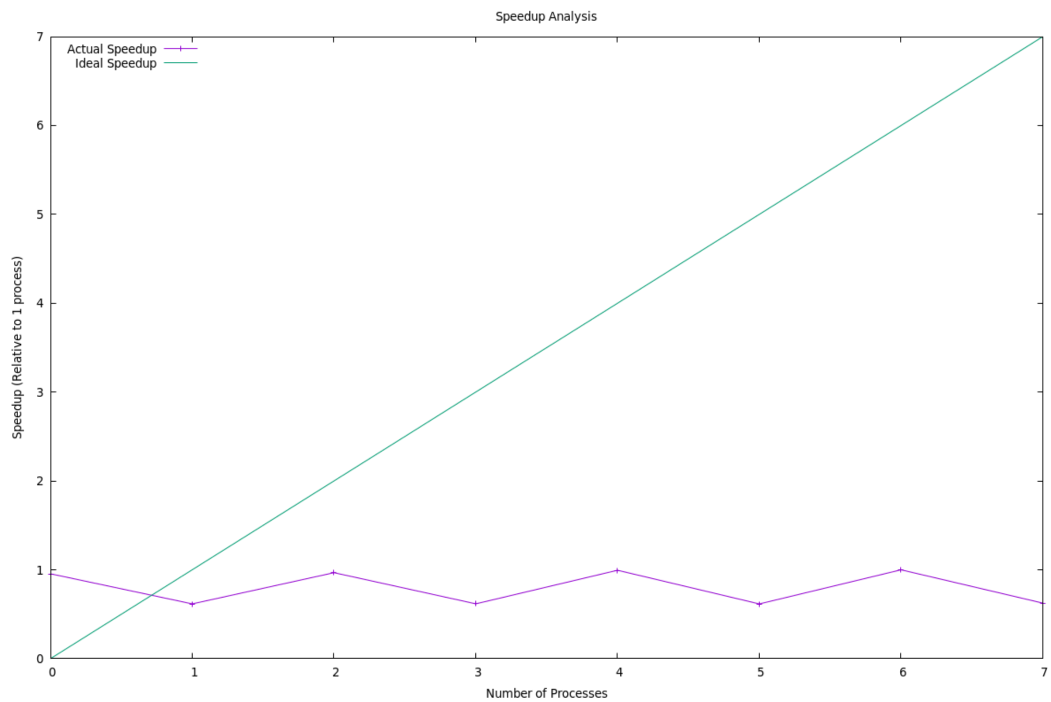
Performance data was saved in CSV format. A custom GNUPLOT script automatically generated plots:

- Time breakdown (stacked histogram).
- Total runtime vs. rank.
- Workload (nodes/edges per rank).
- Partition imbalance.
- Speedup comparison (actual vs ideal).

The following shows the performance metrics calculated:



And The speed up observed is shown in following image:



4. Datasets Used

The experiments used:

- **Facebook Combined Dataset** from SNAP (Stanford Large Network Dataset Collection).
- A custom `facebook_changes.txt` file for simulating dynamic edge changes.

5. Results & Scalability Insights

- **Speedup**: The solution exhibited noticeable performance improvement as the number of processes increased, especially from 1 to 4 cores. After 4, returns diminished due to communication overhead.
- **Workload Balance**: METIS generally ensured well-distributed partitions. However, graphs with community structures occasionally led to uneven edge counts.
- **Computation Time**: This was the most significant component in the runtime profile and benefitted the most from OpenMP parallelism.
- **Total Time**: The visualization showed a decreasing trend initially and plateaued as communication overhead counterbalanced the computational speedup.

6. Challenges Encountered

- **Weight Corrections**: Some edge data lacked weights or had invalid values, requiring dynamic correction.
- **Partitioning Failures**: METIS occasionally failed on certain node arrangements; fallbacks ensured the process didn't crash.
- **MPI Broadcast Size Limitations**: Serializing a large graph structure for MPI communication was non-trivial and had to be optimized using compact representations.

7. Repository Structure and Version Control

Our GitHub repository followed a clean structure:

- `/src` – main implementation.
- `/data` – datasets used.
- `/output` – visualizations and logs.
- `/scripts` – plotting scripts and batch files.

Commits reflected each logical phase, from graph parsing to MPI integration and visualization. All major iterations were committed with descriptive messages.

8. Conclusion

This project allowed us to deeply engage with the practical challenges of scaling graph algorithms in a distributed environment. The integration of MPI and OpenMP, supplemented by METIS partitioning, gave us insights into real-world HPC workflow. While significant speedup was achieved, the performance gain leveled off beyond a certain point, reinforcing the importance of balancing compute and communication costs in parallel systems.