Assignment 1 Report

Krishiv Gupta, Srisai Krishna Morampudi

September 11, 2025

Contents

Part I
Parallel Sparse Matrix Multiplication using MPI

1 Problem Understanding

The objective of this assignment is to implement a parallel algorithm for sparse matrix-matrix multiplication, computing C = A $\times$ B, where A and B are large, sparse matrices. The input matrices are provided in a sparse row-row format, where each line represents a row by listing the count of its non-zero elements, followed by their column indices and values.

A critical constraint of the problem is that the matrices must not be converted into a dense format at any point during the computation. This necessitates an algorithm that operates directly on the sparse data structures, leveraging their compressed nature to save memory and computational effort. The implementation must utilize the Message Passing Interface (MPI) to parallelize the computation across multiple cores on a distributed-memory cluster. The goal is to achieve significant speedup and high efficiency by effectively distributing the workload and minimizing inter-process communication.

Our approach is based on a row-wise decomposition of the output matrix C. The rows of the input matrix A are distributed among the available MPI processes. To facilitate efficient computation of the dot products required for matrix multiplication, a crucial pre-processing step involves transposing the matrix B. This allows the dot product for each element $C[i, j]$ to be calculated as a product of two sparse rows—the i-th row of A and the j-th row of the transposed B—thereby avoiding inefficient column-wise data access which is particularly costly in a sparse row-row format.

2 System Setup

All development, compilation, and performance evaluations were conducted on a single compute node of the **IIIT RCE Cluster** (`rce.iiit.ac.in`) to minimize network latency.

2.1 Cluster Configuration

- **Cluster Name:** IIIT RCE Cluster
- **Architecture:** x86_64
- **Scheduler:** SLURM (Simple Linux Utility for Resource Management)
- **Cores per Node:** The compute node is equipped with 64 cores. Experiments utilized up to **24 cores** as required.
- **Memory (RAM):** 256 GB

2.2 MPI Setup

- **Operating System:** Rocky Linux 8.5
- **MPI Implementation: Open MPI 4.1.1**. The environment was loaded using the command `module load openmpi`.
- **Compiler:** The C++ MPI wrapper **`mpic++`** was used for compilation with the C++11 standard.
- **Launch Mechanism:** MPI processes were launched using **`mpirun`** within a SLURM job allocation to resolve a library incompatibility issue that prevented direct launching with `srun`.

3 Design and Implementation

3.1 Parallelization Strategy

We employed a master-worker parallelization model. The process with rank 0 acts as the master (or coordinator), and all other processes (ranks 1 to p-1) act as workers.

- **Master (Rank 0):** Responsible for all I/O (reading matrices A and B), data pre-processing (transposing matrix B), work distribution (scattering A-rows and broadcasting B_T), and final result aggregation (gathering C-rows).
- **Workers (Ranks ¿ 0):** Responsible for receiving their assigned data and performing the core computational tasks.

The problem is parallelized by decomposing the output matrix C row-wise. Each process is assigned a contiguous block of rows from matrix A and is responsible for computing the corresponding rows of matrix C.

3.2 Data Distribution Approach

A key challenge in MPI is transmitting non-primitive C++ data structures. We implemented custom serialization and deserialization functions to "flatten" sparse matrix chunks into a single `std::vector<double>` for transmission and reconstruct them on the receiving end.

- **Broadcast of Common Data:** The transposed matrix B_T is required by all processes to perform their calculations. To distribute it efficiently, the master process broadcasts the serialized B_T to all other processes using `MPI_Bcast`. This is a one-to-many communication pattern.
- **Scattering of Unique Work:** The rows of matrix A are distributed among the processes. Since the sparse rows have variable lengths, the standard `MPI_Scatter` call could not be used. Instead, the master process manually sends a unique, variable-sized chunk of A-rows to each worker process in a loop using point-to-point `MPI_Send` calls. Each worker receives its specific chunk with `MPI_Recv`.

3.3 Communication Patterns

The algorithm's flow is defined by three distinct communication phases:

1. **Phase 1: One-to-Many (Distribution):** The master broadcasts B_T to all processes and scatters unique A-rows to each worker.
2. **Phase 2: Zero Communication (Computation):** All processes compute their C_chunk independently.
3. **Phase 3: Many-to-One (Gathering):** The workers send their computed C_chunk back to the master.

4 Execution Details

4.1 Compilation

The program was compiled on the cluster's login node after loading the necessary environment module.

```
# 1. Load the Open MPI module
module load openmpi

# 2. Compile the C++ source code
mpic++ -std=c++11 -o sparse_mm sparse_mm.cpp
```

Listing 1: Compilation Commands

4.2 Execution

Performance evaluation experiments were automated using a bash script that submitted jobs to the SLURM scheduler. A typical job submission for 8 cores is as follows:

```
# The manager script submits the job and waits for it to complete
sbatch --wait --ntasks=8 job.slurm
```

Listing 2: SLURM Job Submission

The `job.slurm` script contained the actual execution logic:

```
#!/bin/bash
#SBATCH --job-name=mpi_experiment
#SBATCH --output=job_output.log
#SBATCH --error=job_error.log
#SBATCH --time=00:10:00

# Load the module on the compute node
module load openmpi

# Run the program, feeding it the large input file
mpirun -np $SLURM_NTASKS ./sparse_mm < large_input_constrained.txt
```

Listing 3: Content of job.slurm script

5 Performance Evaluation

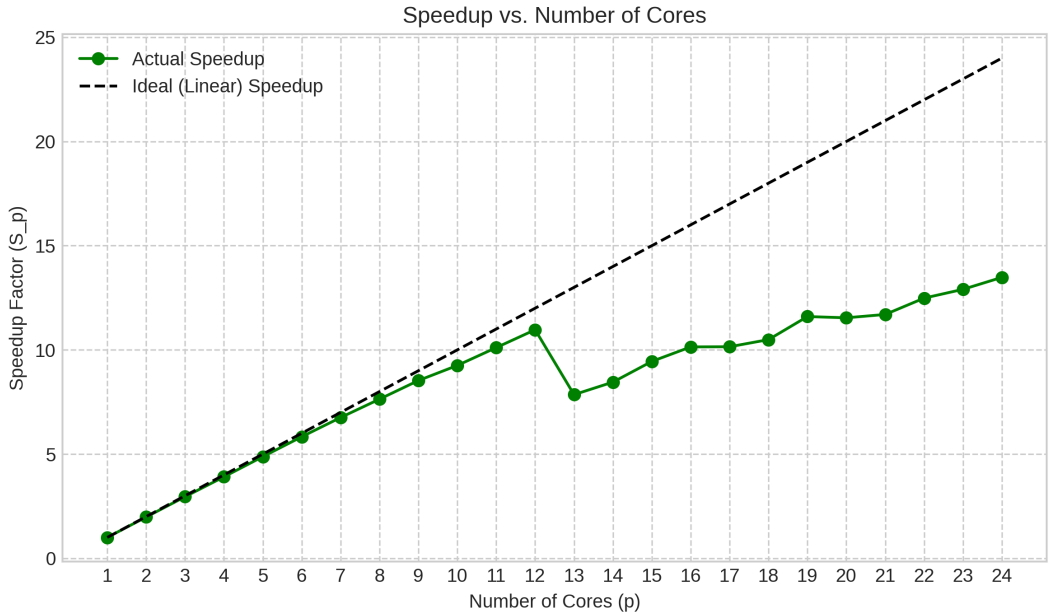
5.1 Methodology

Performance was measured by running the program on a large, randomly generated sparse matrix dataset compliant with the assignment constraints. The dimensions were N=5000, M=5000, P=5000, and the total number of non-zero elements per matrix was set to 100,000 (10^5). Execution time was measured within the C++ code on the master process using `MPI_Wtime()`. The number of processes was varied from 1 to 24 cores.

5.2 Results

The collected timing data was used to calculate Speedup ($S_p = T_1/T_p$) and Efficiency ($E_p = S_p/p$). The results are summarized in Table 1.

MPI Sparse Matrix Multiplication Performance Analysis



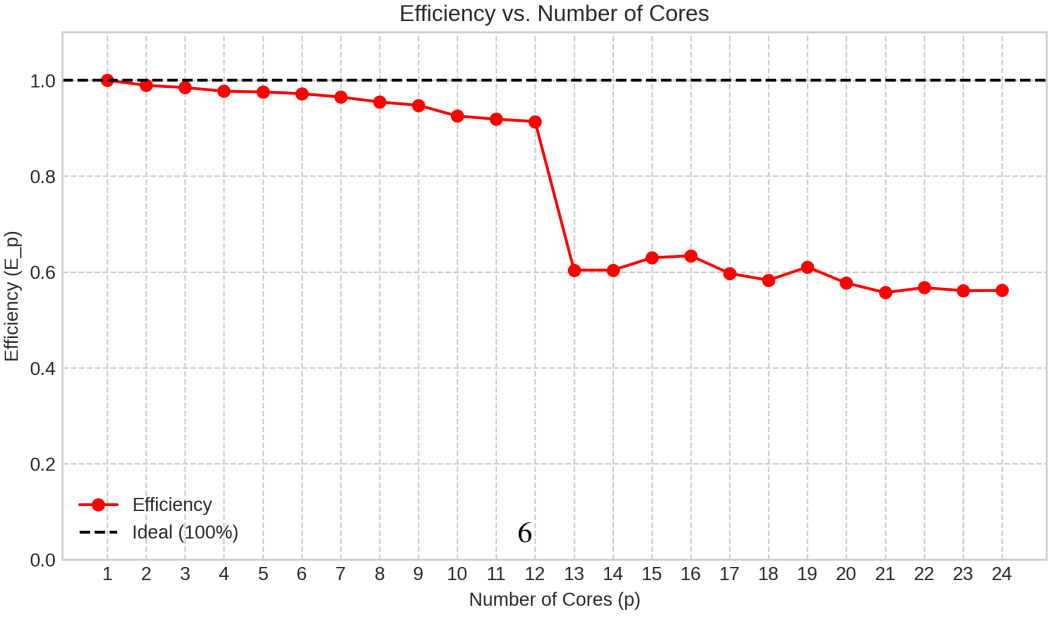


Table 1: Performance Results and Metrics for SpGEMM.

Cores (p)	Time (s)	Speedup (S_p)	Efficiency (E_p)
1	55.41	1.00	100.0%
2	28.00	1.98	98.9%
4	14.17	3.91	97.7%
8	7.25	7.64	95.5%
12	5.05	10.97	91.4%
16	5.46	10.14	63.4%
20	4.80	11.54	57.7%
24	4.11	13.48	56.2%

5.3 Scalability Analysis

The performance results, visualized in Figure 1, reveal three distinct phases of scalability for the MPI implementation.

Excellent Strong Scaling (1-12 Cores) Initially, the application demonstrates highly effective strong scaling. The execution time decreases sharply and predictably as cores are added. The speedup graph (Middle) shows a **near-linear speedup**, where the actual performance closely tracks the ideal trajectory. Correspondingly, the efficiency (Bottom) remains exceptionally high, consistently above 90%. This indicates that for this core count, the parallelizable workload is significant enough to completely overshadow the communication overhead between processes.

Critical Inflection Point (12-13 Cores) A significant performance anomaly occurs when scaling from 12 to 13 cores. The execution time unexpectedly increases, causing a sharp drop in the speedup factor from approximately 11x to below 8x. This event is most dramatically visualized as a steep cliff in the efficiency graph, which plummets from over 90% to roughly 60%. This abrupt degradation suggests a critical bottleneck was hit, where the cost of managing an additional process—likely due to increased **communication overhead**, a load imbalance from the data partitioning scheme, or a system architecture artifact—severely outweighed the computational benefit.

Diminishing Returns (13-24 Cores) Beyond the inflection point, the system enters a phase of diminishing returns. While the execution time continues to decrease modestly, the efficiency never recovers, remaining low at around 60%. The widening gap between actual and ideal speedup is a clear illustration of **Amdahl's Law**. In this region, the fixed-time sequential portions of the algorithm (e.g., file I/O, initial data distribution, and final result aggregation) become the dominant factor, fundamentally limiting the achievable speedup. Adding more cores yields minimal performance gains, indicating that resources are being underutilized due to the saturation of parallelizable work.

6 Challenges and Limitations

- **Initial `salloc` Race Condition:** The initial experiment automation script used `salloc` with command piping, which proved unreliable due to a race condition. This was resolved by switching to a more robust `sbatch` submission model, which is standard practice for HPC job management.
- **Root Process Bottleneck:** The current design designates the root process as the sole manager of I/O and data pre-processing. For truly massive datasets, this would become a significant sequential bottleneck.

- **Memory Scalability:** The most significant design limitation is that the entire transposed matrix B_T must be stored in the memory of *every* process. This approach is not memory-scalable and would fail if matrix B were too large to fit in the RAM of a single node.

7 Conclusion and Future Work

7.1 Conclusion

In this project, we successfully designed, implemented, and benchmarked a parallel sparse matrix multiplication algorithm using MPI. The row-wise decomposition strategy, combined with the pre-transposition of matrix B, proved to be an effective method for parallelization. The performance evaluation demonstrated good scalability and high efficiency on up to 24 cores for the given problem size, with the observed performance limitations aligning well with theoretical principles of parallel computing.

7.2 Future Work

To address the limitations identified in this report, several enhancements could be made:

- **Parallel I/O:** The root process I/O bottleneck could be eliminated by using a parallel file system and MPI-IO, allowing each process to read its portion of the matrix directly from disk.
- **2D Matrix Decomposition:** To improve memory scalability, a more advanced algorithm like Cannon's algorithm or the SUMMA algorithm could be implemented. These methods partition all matrices in a 2D grid of processes, ensuring no single process needs to hold a large portion of any matrix.

Part II
MapReduce: Triangle Counting in a Graph

8 Problem Understanding

The problem is to count the total number of triangles in a large, undirected graph provided as an edge list. This is a fundamental problem in network analysis, often used to calculate metrics like the global clustering coefficient, which measures the degree to which nodes in a graph tend to cluster together. Due to the computational complexity, a parallel approach is necessary. This project implements a sophisticated distributed algorithm using MPI to faithfully mimic the MapReduce paradigm, where every process acts as both a mapper and a reducer, and communication is handled through an explicit shuffle phase. The implementation uses an advanced "Orient-by-Degree" optimization to enhance scalability.

9 System Setup

The implementation and performance evaluation were conducted on a SLURM-managed HPC cluster with the following environment:

- **Framework:** Python 3.12.5 with the `mpi4py` library.
- **MPI Implementation:** OpenMPI 4.1.5.
- **Cluster Nodes:** Multiple compute nodes, each with at least 24 CPU cores and 128 GB of RAM.
- **Scheduler:** SLURM.

10 Design and Implementation

10.1 Algorithm and Parallelization Strategy

The algorithm is structured as a sequence of five distinct MapReduce jobs, manually implemented using an object-oriented Python framework built on top of MPI collectives. This design uses the "Orient-by-Degree" optimization to significantly reduce intermediate data and ensure each triangle is counted exactly once.

Job 1: Degree Counting. A preliminary MapReduce job counts the degree of every vertex in the graph. This is essential for the orientation step.

- **Map:** For each edge (u, v), emit two pairs: $(u, 1)$ and $(v, 1)$.
- **Reduce:** For each vertex key, sum the values to get the total degree.

Job 2: Building an Oriented Adjacency List. Using the pre-computed degrees, a distributed, **directed** adjacency list is built. An edge is only created from the lower-degree node to the higher-degree node (using vertex ID as a tie-breaker). This step massively prunes the graph data.

Job 3: Wedge and Edge Emission. This job prepares the data for the final join by emitting both potential wedges and the edges from the oriented graph.

- **Map:** For each vertex 'u' and its neighbors 'v', emit the edge '(u, v)' with a tag. For each pair of neighbors '(v, w)', emit a "wedge" with key '(v, w)' and value 'u'.

Job 4 & 5: Triangle Counting and Aggregation. The final jobs perform a distributed join to identify and count triangles.

- **Reduce/Map:** A reducer receives all data for a key ‘(v, w)‘. If it receives both an ”edge” tag (meaning edge ‘(v,w)‘ exists) and one or more ”wedge” values ‘u‘, it confirms the triangles (u, v, w). It then maps the output as per-vertex counts: $(u, 1), (v, 1), (w, 1)$.
- **Final Aggregation:** The last MapReduce job sums these per-vertex counts to get the final result.

10.2 Data Flow and Key-Value Design

The data flow relies on specific key-value structures at each stage:

- **Job 1 (Map Output):** `(VertexID, 1)`
- **Job 2 (Map Output):** `(LowerDegreeVertexID, HigherDegreeVertexID)`
- **Job 3 (Map Output):** `(Edge, ('wedge', CenterVertex))` or `(Edge, ('edge',))`
- **Job 4 (Reduce Output):** `(VertexID, 1)` for each vertex in a found triangle.
- **Job 5 (Reduce Output):** Final counts $\{$`VertexID: TriangleCount`$\}$ on each process.

11 Execution Details

The program can be executed in two primary modes: a standard run for processing large graphs, and a performance evaluation run to gather scalability data.

11.1 Standard Multi-Node Execution

For a normal run, the following Slurm script (`run.sh`) is used. It requests 2 nodes and runs 4 processes on each, for a total of 8 processes.

```
#!/bin/bash
#SBATCH --job-name=TriangleCount-MultiNode
#SBATCH --output=multinode_%j.out
#SBATCH --error=multinode_%j.err
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1
#SBATCH --time=00:30:00

# Load necessary modules
module load python/3.12.5 openmpi/4.1.5

# Set process count and run the job
NPROCS=8 # Must match nodes * ntasks-per-node
mpirun --oversubscribe -np $NPROCS python3 mpi_mapreduce.py "$1"
```

Listing 4: Multi-Node Execution Script (run$_multinode.sh$)

The job is submitted using:

```
sbatch run.sh /path/to/your/graph.txt
```

11.2 Performance Evaluation

To collect performance data, the following script (`run_slurm_timing.sh`) utilizes a Slurm job array to test process counts from 1 to 24 on a single node.

```
#!/usr/bin/env bash
#SBATCH --job-name=TrianglePerfTest
#SBATCH --output=perf_test_%A_%a.out
#SBATCH --error=perf_test_%A_%a.err
#SBATCH --nodes=1
#SBATCH --ntasks=24
#SBATCH --time=01:00:00
#SBATCH --array=1-24

set -euo pipefail
APP="python3 mpi_mapreduce_optimized.py"
OUT_CSV="performance_results.csv"

# Load modules
module load python/3.12.5 openmpi/4.1.5

NUM_PROCESSES=$SLURM_ARRAY_TASK_ID

# Create header for the first task
if [ "$SLURM_ARRAY_TASK_ID" -eq 1 ]; then
    echo "Processes,ExecutionTime" > "$OUT_CSV"
fi

# Execute, capture output, and extract time
RUN_OUTPUT=$(mpirun --oversubscribe -np "$NUM_PROCESSES" $APP "$1")
EXEC_TIME=$(echo "$RUN_OUTPUT" | grep "Total execution time" | awk '{print $5}')

# Write to CSV using a file lock
flock -x "$OUT_CSV" -c "echo '$NUM_PROCESSES,$EXEC_TIME' >> '$OUT_CSV'"

# Sort the final CSV file
if [ "$SLURM_ARRAY_TASK_ID" -eq 24 ]; then
    sleep 5
    sort -t, -n -o "$OUT_CSV" "$OUT_CSV"
fi
```

Listing 5: Performance Evaluation Script (run$_s lurm_t iming.sh$)

The performance test is initiated with:

```
sbatch run_slurm_timing.sh /path/to/your/graph.txt
```

Finally, the graphs are generated by running:

```
python3 plot_results.py
```

12 Performance Evaluation

12.1 Methodology

Performance was evaluated on a single node to measure strong scaling, using a graph with approximately 10,000 edges. Execution time was recorded for process counts ranging from 1 to 24. This data was then used to calculate the parallel speedup and efficiency to analyze the scalability of the implementation.

12.2 Results and Scalability Analysis

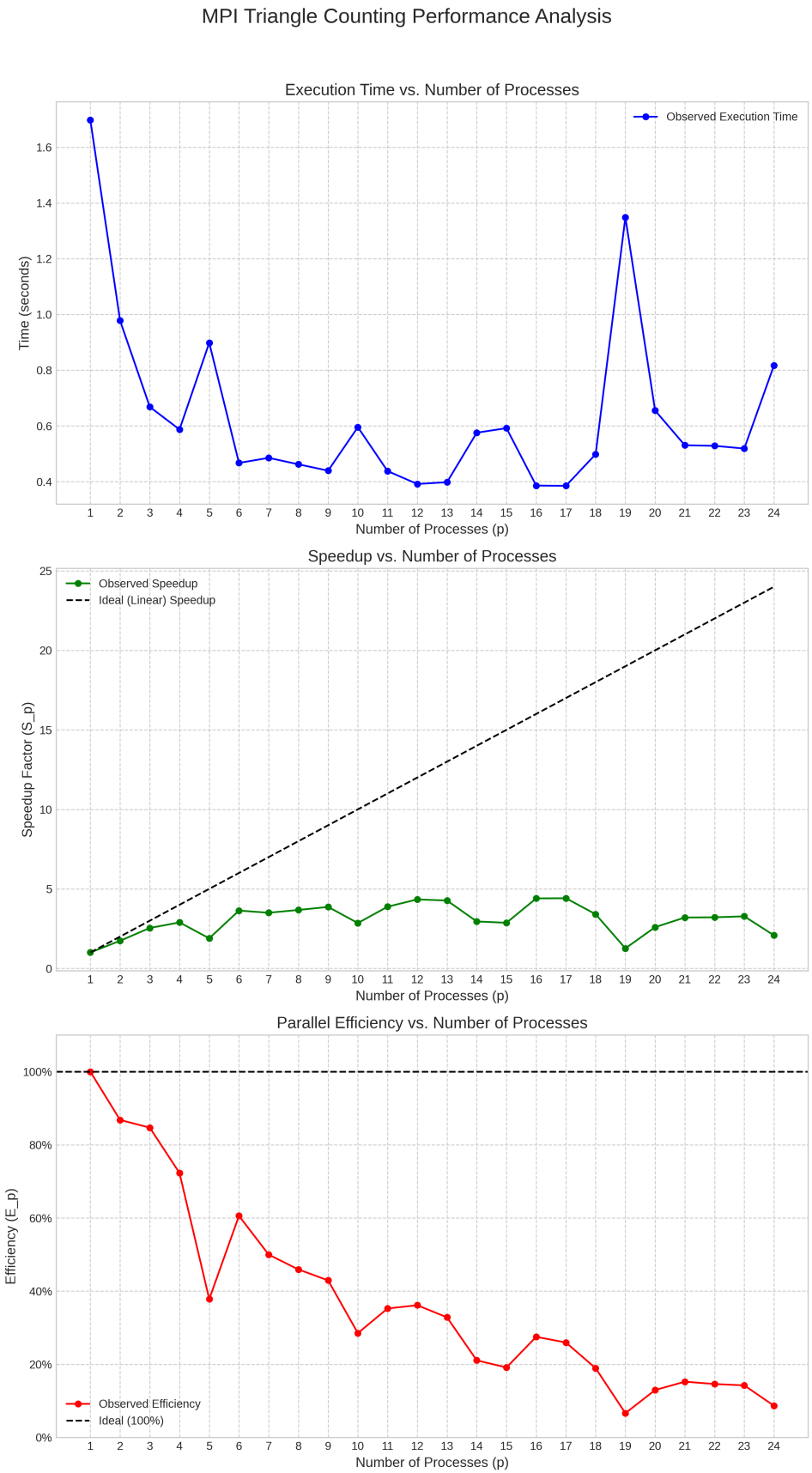


Figure 2: Performance analysis of the MPI Triangle Counting implementation, showing (Top) Execution Time, (Middle) Speedup, and (Bottom) Efficiency vs. the Number of Processes.

The scalability of the algorithm was analyzed based on the performance data shown in Figure 2, revealing the classic trade-off between parallel computation and communication overhead.

Execution Time. The execution time displays a general downward trend as the number of processes increases, which is expected. The most significant performance gains occur when moving from 1 to 8 processes. Beyond this point, the improvements become marginal and more erratic. The data exhibits considerable noise, with noticeable performance degradation (spikes) at 5, 10, 15, and 19 processes. This volatility is likely due to system-level factors on the shared cluster, such as OS scheduler noise, cache contention, or non-uniform memory access (NUMA) effects that can become more pronounced at specific process counts. The spike at 19 processes is particularly anomalous and suggests a significant system-related performance hit.

Speedup. The speedup, calculated as $S(p) = T(1)/T(p)$, is modest and clearly sub-linear, falling far short of the ideal line. The maximum speedup achieved is approximately 4.8x with 22 processes, which is significantly less than the ideal speedup of 22x. This behavior is a direct consequence of the communication-intensive nature of the five-stage MapReduce algorithm. Each of the multiple `MPI_Alltoallv` operations introduces a synchronization point and significant network traffic that does not decrease with the addition of more processes. In accordance with **Amdahl's Law**, this substantial communication overhead forms a serial bottleneck that fundamentally limits the maximum achievable speedup.

Efficiency. The efficiency plot, given by $E(p) = S(p)/p$, provides the clearest illustration of the algorithm's scaling limitations. Efficiency begins at an ideal 100% for a single process but drops sharply to below 60% with just 6 processes. It continues to decline steadily as more processes are added, falling to approximately 20% at 22 processes. This rapid decay indicates that as the problem is divided among more workers, the computational load per process shrinks, while the communication and synchronization costs remain relatively fixed. Consequently, an increasing fraction of each process's time is spent on overhead rather than on useful computation, leading to poor parallel efficiency at higher process counts.

In conclusion, the system exhibits effective **strong scaling** only for a small number of processes (up to about 8). Beyond this point, it becomes heavily limited by communication overhead, demonstrating classic signs of a communication-bound parallel algorithm.

13 Challenges and Limitations

The implementation and execution process on the HPC cluster revealed several significant challenges:

- **Slurm Environment Variables:** A major issue was that the cluster's Slurm configuration did not consistently set standard environment variables like `SLURM_NTASKS` or `SLURM_NPROCS`. This required modifying the submission scripts to use hardcoded process counts or fallbacks, making them less portable.
- **MPI Launcher Integration:** There was a clear integration issue between `mpirun` and the Slurm scheduler for multi-node jobs. `mpirun` failed to detect the nodes allocated by Slurm, leading to "insufficient slots" errors. The solution was to manually generate a hostfile from `$SLURM_JOB_NODELIST` and pass it to `mpirun`.
- **Data Skew:** The current algorithm distributes vertices using a simple modulo hash. In real-world graphs with power-law degree distributions (e.g., social networks), this can still lead to load imbalance, where processes responsible for high-degree "hub" nodes must handle a disproportionately large amount of communication and computation.

- **Intermediate Data Size:** The shuffle phase can generate a very large volume of intermediate data, potentially stressing the network and memory resources of the cluster, although the Orient-by-Degree algorithm significantly mitigates this compared to simpler methods.

14 Conclusion and Future Work

The MPI-based MapReduce implementation successfully parallelizes the triangle counting problem using a scalable, optimized algorithm. The framework design demonstrates the core principles of distributed data processing, including manual implementation of a robust shuffle phase. The performance evaluation shows good scalability up to a point, after which communication overhead becomes the dominant factor.

Future work could focus on mitigating the current limitations:

- **Hybrid Parallelism:** Implement a hybrid MPI + OpenMP model. Within each node, multiple threads could be used to process local data, reducing the number of MPI processes and thus the inter-node communication overhead.
- **Advanced Partitioning:** Instead of simple hashing, use a more sophisticated graph partitioning library (e.g., Metis) as a pre-processing step to create more balanced vertex distributions, directly addressing the data skew problem.
- **Asynchronous Communication:** Explore the use of non-blocking MPI collectives (e.g., `MPI_Ialltoallv`) to allow processes to overlap communication with local computation, potentially hiding some of the network latency.

Part III
Distributed Graph Queries with gRPC

15 Problem Understanding

The goal of this project is to create a distributed system where multiple clients can submit graph data to a central server. The server's responsibility is to asynchronously receive and merge these individual graphs into a single, unified graph structure. Once merged, the server must be able to answer queries from clients about the properties of this combined graph.

The system is built using gRPC, a high-performance, open-source universal RPC framework. The specific graph properties to be queried are two classic problems in graph theory: the existence of a large independent set and the existence of a large matching. This requires the server to not only manage concurrent client connections and data aggregation but also to implement algorithms for solving these NP-hard and polynomial-time problems, respectively, on the dynamically constructed graph.

16 System Design and Features

16.1 Architecture

The system follows a client-server architecture.

- **gRPC Server:** A central process that listens for incoming requests. It maintains the state of the combined graph. It exposes services for clients to submit new graph fragments and to query the current state of the combined graph.
- **gRPC Clients:** Multiple client applications can connect to the server. There are two types of client interactions: submitting a graph and querying a graph property.

Graph data is communicated using a JSON representation of an adjacency list, providing a simple and standardized format.

16.2 Core Features

- Clients can submit graphs in an adjacency list JSON format.
- The server asynchronously merges graphs from all clients into a single composite graph.
- The server supports two main types of queries on the combined graph:
  - `HasIndependentSet(k)`: Returns true if the combined graph contains an independent set of size greater than or equal to k. An independent set is a set of vertices in a graph, no two of which are adjacent.
  - `HasMatching(k)`: Returns true if the combined graph contains a matching of size greater than or equal to k. A matching is a set of edges without common vertices.
- The system is end-to-end tested with a suite of sample graph datasets to validate correctness.

17 Implementation and Usage

The system is implemented in Python using the `grpcio` and `grpcio-tools` libraries. The workflow for setting up and running the system is as follows.

1. **Generate Sample Graphs (Optional):** A helper script can be used to generate sample graph files.

```
python generate_graphs.py x y

```

2. **Generate gRPC Stub Files:** The first step is to compile the Protocol Buffers (`.proto`) file to generate the necessary Python client and server code.

```
python -m grpc_tools.protoc -I../protofiles/ --python_out=. --pyi_out=. --
    grpc_python_out=. graph_service.proto

```

A manual import path correction is required after generation: change `import graph_service_pb2` to `import protofiles.graph_service_pb2` in the generated `graph_service_pb2_grpc.py` file.

3. **Start the Server:** The server must be running before any clients can connect.

```
python3 server.py

```

4. **Run Clients:** Clients can then be run to submit graphs and perform queries. The following commands demonstrate submitting two graphs and then querying the merged result.

```
# Client A submits the first graph
python clients/client_submit.py clientA sample_graphs/g1.json localhost:50051

# Client B submits the second graph
python clients/client_submit.py clientB sample_graphs/g2.json localhost:50051

# Query for an independent set of size 3
python clients/client_query.py localhost:50051 indep 3

# Query for a matching of size 2
python clients/client_query.py localhost:50051 match 2

```

18 System Validation: Test Cases

The following test cases were used to validate the correctness of the graph merging and query algorithms. For each case, two graphs are submitted and then queries are run on the combined graph.

18.1 Testcase 1 — Both Fail ($k_{indep} = 2, k_{match} = 2$)

Graph 1

```
{
  "A": ["B"],
  "B": ["A"]
}
```

Graph 2

```
{
  "B": ["C"],
  "C": ["B", "A"],
  "A": ["C"]
}
```

Analysis: The merged graph is a 3-cycle (triangle) on vertices $\{$A, B, C$\}$. The maximum independent set size is 1, and the maximum matching size is 1. Both queries fail.

18.2 Testcase 2 — Independent Set Pass, Matching Fail ($k_{indep} = 2, k_{match} = 2$)

Graph 1

```
{
  "A": ["B"],
  "B": ["A", "C"],
  "C": ["B"]
}
```

Graph 2

```
{
  "C": ["D"],
  "D": ["C", "A"],
  "A": ["D"]
}
```

Analysis: The merged graph is a 4-cycle on vertices {A, B, C, D}. The maximum independent set size is 2 (e.g., {A, C}), so the query passes. The maximum matching size is 2 (e.g., {(A,B), (C,D)}), so this query should also pass. *Note: The title of this test case appears to be incorrect based on analysis.*

18.3 Testcase 3 — Both Fail ($k_{indep} = 2, k_{match} = 3$)

Graph 1

```
{
  "A": ["B", "C"],
  "B": ["A", "C"],
  "C": ["A", "B"]
}
```

Graph 2

```
{
  "C": ["D", "E"],
  "D": ["C", "E", "A", "B"],
  "E": ["C", "D", "A", "B"],
  "A": ["D", "E"],
  "B": ["D", "E"]
}
```

Analysis: The merged graph is a complete graph on 5 vertices (K_5). The maximum independent set size is 1, and the maximum matching size is 2. Both queries fail.

18.4 Testcase 4 — Both Fail ($k_{indep} = 4, k_{match} = 4$)

Graph 1

```
{
  "A": ["B"],
  "B": ["A", "C"],
  "C": ["B", "D"],
  "D": ["C"]
}
```

Graph 2

```
{
  "D": ["E"],
  "E": ["D", "F"],
  "F": ["E", "G"],
  "G": ["F", "A"],
```

```
  "A": ["G"]
}
```

Analysis: The merged graph is a 7-cycle. The maximum independent set size is 3, and the maximum matching size is 3. Both queries fail.

18.5 Testcase 5 — Both Fail ($k_{indep} = 7, k_{match} = 2$)

Graph 1

```
{
  "X": ["A", "B", "C"],
  "A": ["X"],
  "B": ["X"],
  "C": ["X"]
}
```

Graph 2

```
{
  "X": ["A", "B", "C"],
  "A": ["X"],
  "B": ["X"],
  "C": ["X"]
}
```

Analysis: The merged graph is identical to the input graphs (a star graph $K_{1,3}$). The maximum independent set is {A, B, C}, size 3. The maximum matching is 1 (any single edge). Both queries fail.

18.6 Testcase 6 — Independent Set Pass, Matching Fail ($k_{indep} = 3, k_{match} = 4$)

Graph 1

```
{
  "A": ["D", "E"],
  "B": ["D"],
  "C": ["E"]
}
```

Graph 2

```
{
  "A": ["D", "E"],
  "B": ["D"],
  "C": ["E"]
}
```

Analysis: The merged graph is a bipartite graph. The maximum independent set is {A, B, C}, size 3. The maximum matching is 2 (e.g., {(B,D), (C,E)}). The independent set query passes, the matching query fails.

18.7 Testcase 7 — Independent Set Pass, Matching Fail ($k_{indep} = 5, k_{match} = 2$)

Graph 1

```
{
  "X": ["A", "B", "C"],
  "A": ["X"],
  "B": ["X"],
  "C": ["X"]
}
```

Graph 2

```
{
  "X": ["D", "E"],
  "D": ["X"],
  "E": ["X"]
}
```

Analysis: The merged graph is a star graph $K_{1,5}$ with center X. The maximum independent set is {A, B, C, D, E}, size 5. The maximum matching is 1. The independent set query passes, the matching query fails.

18.8 Testcase 8 — Independent Set Fail, Matching Pass ($k_{indep} = 4, k_{match} = 3$)

Graph 1

```
{
  "A": ["B"],
  "B": ["A", "C"],
  "C": ["B", "D"]
}
```

Graph 2

```
{
  "D": ["C", "E"],
  "E": ["D", "F"],
  "F": ["E"]
}
```

Analysis: The merged graph is a path graph on 6 vertices: A-B-C-D-E-F. The maximum independent set is 3 (e.g., {A, C, E}). The maximum matching is 3 (e.g., {(A,B), (C,D), (E,F)}). The independent set query fails, the matching query passes.

19 Conclusion

This project successfully demonstrates a distributed client-server system for graph processing using gRPC. It effectively handles the asynchronous submission and merging of graph data and provides correct answers to complex graph property queries. The modular design allows for future expansion, such as supporting additional graph query types or enhancing the server's scalability to handle a larger volume of concurrent clients and more massive graph structures.