Distributed Systems Homework-2

Chandaluri Yaswanth Reddy (2024201045) Shreyas Adiga (2022111010)

September 13, 2025

1 MPI

1.1 Problem Understanding

1.1.1 Problem Description

This project addresses the computational challenge of **parallel sparse matrix multiplication** using MPI for distributed memory systems, where two sparse matrices A ($N \times M$) and B ($M \times P$) are multiplied to produce result matrix C ($N \times P$).

The core problem involves efficiently distributing computational workload across multiple processors when sparse matrices have highly irregular non-zero element distributions. Traditional row-based partitioning often suffers from severe load imbalance, where some processors handle dense rows while others process mostly empty rows, leading to poor parallel efficiency.

Our solution implements an innovative **non-zero element distribution strategy** that evenly distributes individual matrix elements (rather than rows) across processors, ensuring balanced computational workload regardless of sparsity patterns. This is combined with optimized MPI communication patterns, including efficient batch broadcasting and hierarchical result collection, to achieve scalable performance across varying processor counts and matrix sizes.

1.1.2 Approach

This project implements parallel sparse matrix multiplication using MPI with a novel load balancing strategy based on distributing non-zero matrix elements evenly across processors, thereby addressing load imbalance issues in traditional row-wise partitioning. Matrix dimensions and sparse data are broadcast efficiently using packed arrays in single batches to reduce communication overhead.

Each process reconstructs its allocated sparse matrix data and independently computes partial multiplication results using hash maps for sparse accumulation. The master process then asynchronously collects partial results from all processes through structured MPI communication, merging these contributions to produce the final output.

This approach balances workload effectively, minimizes communication and synchronization overhead, and scales efficiently across various matrix sizes and processor counts, thereby overcoming key limitations of traditional parallel sparse matrix multiplication methods.

1.2 System Setup

Cluster Configuration: 32 CPU cores distributed across multiple compute nodes.
MPI Setup: Open MPI installation providing distributed memory parallel computing support.
Development Environment: C++ with MPI integration using C++17 .

1.3 Design and Implementation

1.3.1 Parallelization Strategy

The implementation employs a master-worker parallelization model using MPI, where the master process (rank 0) handles input/output operations, data distribution, and result aggregation, while worker processes execute parallel computations independently. The core innovation lies in non-zero element-based load balancing: instead of traditional row-wise partitioning, the algorithm distributes individual non-zero elements evenly across all processes,

ensuring each processor handles approximately $\frac{total\_nnz}{num\_processes}$ elements regardless of sparsity patterns. This approach effectively eliminates load imbalance issues that plague conventional sparse matrix multiplication methods.

1.3.2 Data Distribution Approach

Sparse matrices are represented using the Compressed Sparse Row (CSR) format for memory efficiency. The master process reads matrix dimensions (N, M, P) and sparse matrix data, then optimizes communication through data packing strategies—sparse matrix elements are consolidated into contiguous arrays before broadcasting. Matrix B is broadcast completely to all processes using `MPI_Bcast` operations, while Matrix A is distributed based on computed work assignments. All processes reconstruct local sparse matrix representations from the broadcast data, enabling independent parallel computation while minimizing communication overhead.

1.3.3 Communication Patterns

The implementation utilizes hybrid communication patterns combining collective and point-to-point operations for optimal efficiency. Collective communication (`MPI_Bcast`) handles matrix dimensions and packed sparse data distribution in single batch operations. Point-to-point communication (`MPI_Send/MPI_Recv`) manages structured result collection using systematic message tags (0–4) to distinguish different data types. The master process asynchronously collects partial results from worker processes, merges them using hash-based accumulation, and produces the final sparse matrix output. This communication design minimizes synchronization points and reduces network traffic through efficient data packing strategies.

1.4 Execution Details

The program can be compiled and executed using the following commands:

```
Compile : mpic++  -o q1 q1.cpp
Execution : mpirun -np <num_processes> ./q1 < input.txt > output.txt
```

1.5 Performance Evaluation

- **Execution time vs. number of cores**

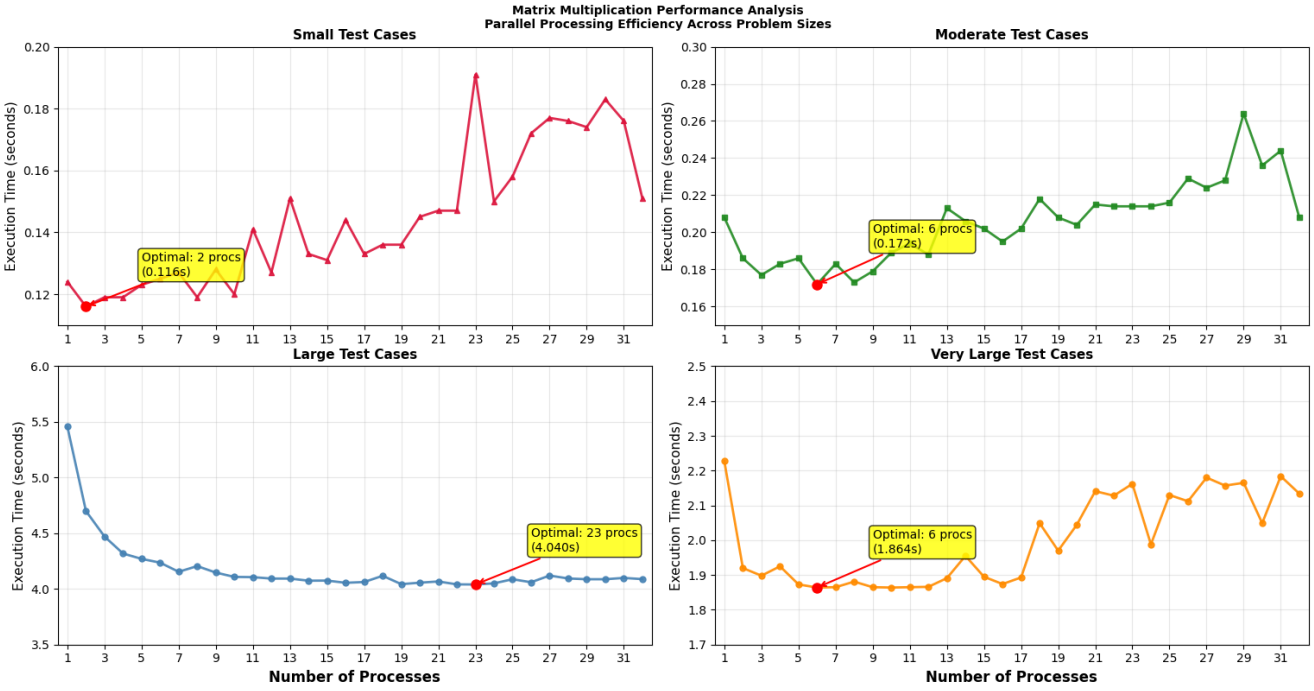

Figure 1: Execution time of sparse matrix multiplication with varying number of cores.

- **Speedup and efficiency plots**

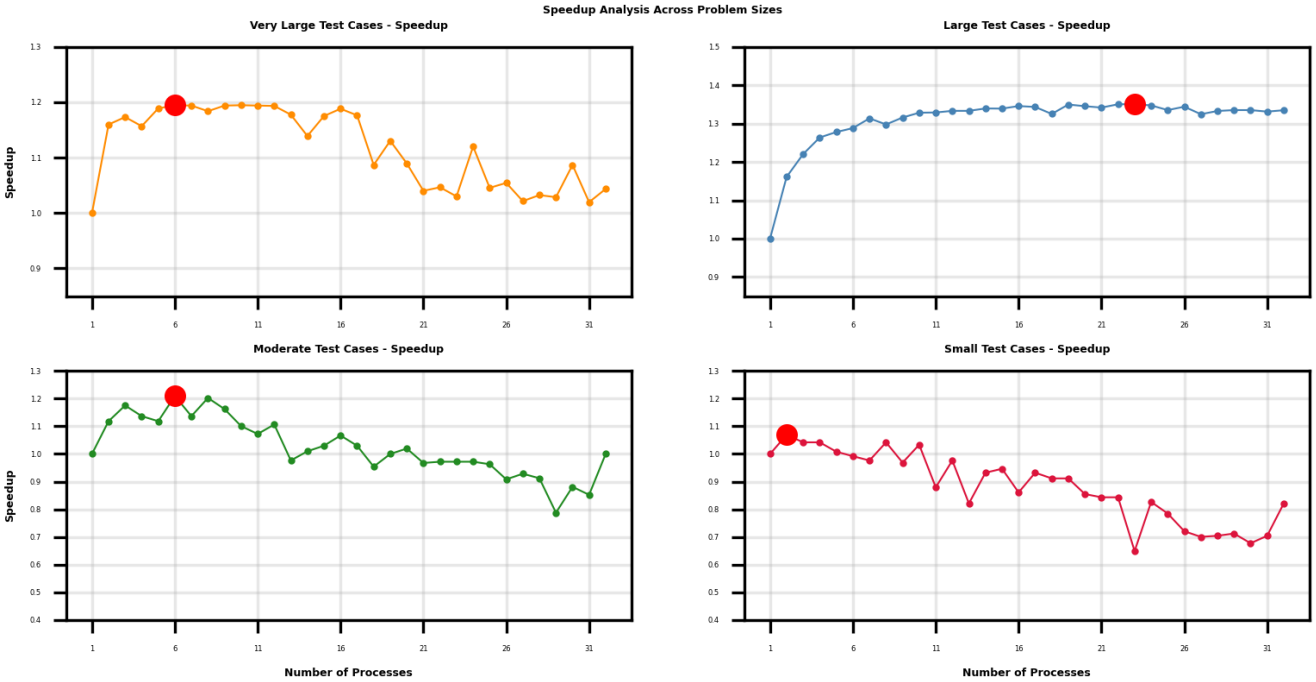

Figure 2: Speedup and efficiency plots for sparse matrix multiplication.

- **Scalability analysis**

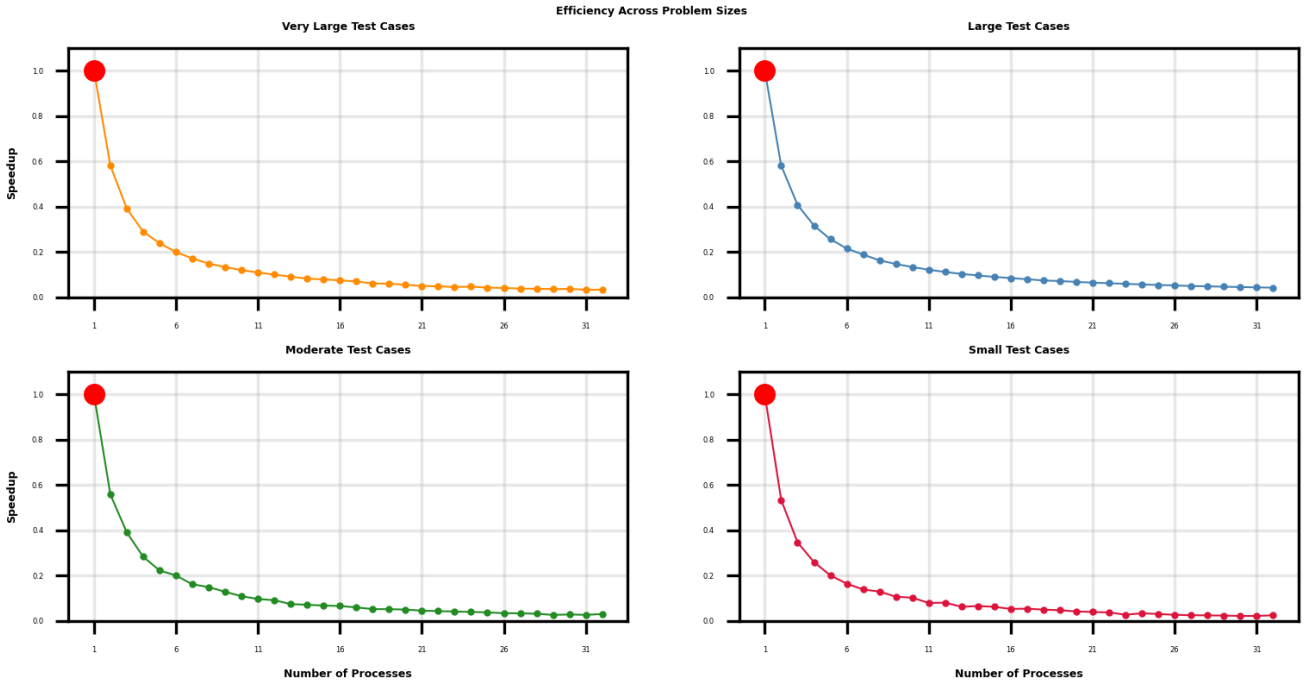

Figure 3: Scalability analysis of the parallel sparse matrix multiplication.

1.6 Challenges and Limitations

1.6.1 Challenges

The implementation encountered several significant challenges during development and evaluation:

- **Communication Overhead:** Frequent MPI broadcasts and message exchanges during data distribution and result collection created substantial latency, particularly affecting small matrix performance where communication costs often exceeded computation time.
- **Load Balancing Difficulties:** Despite implementing the innovative non-zero element distribution strategy, achieving perfect load balance remained challenging due to irregular sparsity patterns and element clustering in certain matrix regions.
- **Implementation Complexity:** Managing dynamic sparse data structures required sophisticated hash-based accumulation techniques, while debugging intricate MPI communication patterns with systematic message tagging across varying processor configurations proved time-consuming.
- **System-Level Variability:** Performance measurements were affected by external factors including network congestion, competing cluster workloads, and operating system scheduling variations, necessitating multiple experimental runs for reliable results.

1.6.2 Limitations

The implementation faces several fundamental limitations that constrain its performance potential:

- **Scalability Constraints:** Performance gains plateau beyond 16–24 processors due to network bandwidth saturation, memory constraints, and increasing dominance of communication overhead following Amdahl's Law limitations.
- **Matrix Size Dependencies:** The approach shows strong sensitivity to matrix characteristics, where small matrices suffer from communication overhead dominance while very large matrices encounter memory bandwidth limitations.
- **Residual Load Imbalance:** Highly irregular sparse matrix structures with extreme clustering patterns cannot be perfectly distributed, leading to persistent workload imbalances that limit parallel efficiency.
- **Hardware-Imposed Bottlenecks:** Fundamental limitations include memory bandwidth constraints, network infrastructure limitations, and processor architecture characteristics that influence cache performance and memory access patterns.

1.7 Conclusion and Future Work

1.7.1 Conclusion

This project successfully implemented an MPI-based parallel sparse matrix multiplication using a master-worker model, where the master broadcasts input matrices and distributes workload by evenly partitioning non-zero elements among processors. Each processor reconstructs its assigned matrix partitions and independently computes partial results using hash maps. The master asynchronously gathers and merges these results into the final output, optimizing communication with batched broadcasts and structured messaging to achieve scalability despite inherent challenges.

1.7.2 Future Work

Future improvements should focus on:

- **Communication-Computation Overlap:** Reducing synchronization delays by overlapping data exchange with computation.
- **Adaptive Load Balancing:** Dynamically redistributing workload to handle irregular sparsity patterns more effectively.
- **GPU Acceleration:** Offloading compute-intensive operations to GPUs to further enhance performance and scalability in large-scale scientific computing applications.

2 MapReduce

2.1 Problem Understanding

The primary objective is to count all simple 4-cycles within a large, undirected graph. A simple 4-cycle is defined as a path traversing four distinct vertices that begins and ends at the same vertex (e.g., A-B-C-D-A). The solution must produce two key metrics: (1) a global, aggregate count of all unique 4-cycles in the graph, and (2) a per-vertex participation count detailing the number of 4-cycles each individual vertex is a part of.

Algorithmic Approach and Complexity

While naive approaches to this problem are computationally infeasible (e.g., $O(n^4)$), efficient sequential algorithms exist with a worst-case time complexity of $O(\min(n^3, n \cdot m))$, where n is the number of vertices and m is the number of edges. Our parallel approach is inspired by the **wedge enumeration** method, which aligns well with the MapReduce paradigm. A wedge is a two-edge path connecting three vertices, such as A-B-C, where B is the "center". A 4-cycle is formed when two distinct wedges, such as A-B-C and A-D-C, share the same pair of endpoints (A and C).

The computational cost of this method is dominated by the generation and processing of all wedges in the graph. The total number of wedges, W, is given by the sum of combinations of neighbors for every vertex: $W = \sum_{v \in V} \frac{d(v) \cdot (d(v)-1)}{2}$, where $d(v)$ is the degree of vertex v. While the theoretical complexity provides an upper bound, the practical performance of our MPI implementations is dictated by how these W wedges are shuffled and aggregated across parallel processes.

Implementation Strategies

To explore the impact of parallel design on performance, two distinct implementations were developed:

1. **A Master-Worker Model:** A foundational implementation that employs a centralized communication pattern. A master process (rank 0) is responsible for distributing work, performing a serial shuffle of all intermediate data, and aggregating the final results. While simple to reason about, this model is fundamentally limited by the communication and processing bottleneck at the master.

2. **An Optimized Parallel Model:** A high-performance implementation designed for scalability. It eliminates the master bottleneck by using a **distributed shuffle**, where worker processes communicate directly with each other using the `MPI_Alltoallv` collective. Performance is further enhanced by processing data in a structured binary format (mapping vertex names to integers) and aggregating final counts with the highly efficient `MPI_Reduce` collective, thereby avoiding the immense overhead of string parsing and serialization.

2.2 System Setup

The experiments were conducted on the RCE cluster at IIIT Hyderabad. The environment was configured as follows:

- **Cluster Configuration:** The jobs were run on compute nodes within the `debug` partition. Each job was allocated a single node with exclusive access to a varying number of cores (2, 6, 12, 24) and a memory allocation of 1GB per core.

- **MPI/MapReduce Setup:** The MapReduce paradigm was implemented from scratch using the Message Passing Interface (MPI). The Open MPI library was used for inter-process communication. No high-level MapReduce frameworks like Hadoop were used. The implementation controls process management, data distribution, and communication manually.

2.3 Design and Implementation

2.3.1 Initial Master-Worker Model (Non-Optimized)

- **Parallelization Strategy:** The model uses a centralized master-worker approach. Rank 0 acts as the master, orchestrating a three-stage MapReduce workflow. All other ranks act as workers, performing computations on data chunks assigned by the master.

- **Data Distribution:** The master process reads the entire graph from `input.txt`. For each of the three MapReduce jobs, it partitions the input data and sends a unique chunk to each worker process. For example, in Job 1, the list of edges is split and distributed among the workers.
- **Communication Patterns:** Communication is heavily centralized.
    1. **Scatter:** The master sends data chunks to workers using point-to-point `MPI_Send` calls.
    2. **Gather:** After each stage, every worker sends its entire intermediate result back to the master using `MPI_Send`. The master uses `MPI_Recv` in a loop to collect all results.
    3. **Shuffle:** The master performs the shuffle phase locally by collecting all worker outputs into a `std::map` to group values by key. This single-process shuffle is the primary performance bottleneck.
    4. **String-Based Communication:** All intermediate data, including wedges and partial counts, is serialized into strings before being sent over MPI. This introduces significant parsing overhead on the receiving end.

2.3.2 Optimized Parallel Model

- **Parallelization Strategy:** This model uses a decentralized, fully parallel strategy to eliminate the master bottleneck. Rank 0 acts only as a lightweight coordinator to start the process and gather the final, tiny result. All worker processes participate equally in computation and shuffling.
- **Data Distribution:** The coordinator (rank 0) reads the graph and maps all string vertex names to unique integer IDs. This integer-based edge list is then broadcast to all processes using `MPI_Bcast`. Each worker receives the entire graph, allowing it to operate on its assigned portion without further data requests.
- **Communication Patterns:**
    1. **Distributed Shuffle:** The shuffle phase is parallelized. Instead of sending data to a master, each worker hashes its output keys (e.g., wedge endpoints) to determine which destination worker is responsible for that key. It then sends the data directly to that peer. This is achieved with the powerful `MPI_Alltoallv` collective operation, which manages the complex all-to-all data exchange efficiently.
    2. **Binary Data Communication:** All communication uses binary data. Vertex IDs, wedges, and counts are sent as structured integers (`MPI_INT`, `MPI_BYTE`) rather than strings. This completely eliminates the expensive string serialization and parsing overhead.
    3. **In-Memory Aggregation with `MPI_Reduce`:** After the distributed shuffle, each worker computes its local partial counts in memory. The final global count is calculated with a single, highly optimized `MPI_Reduce` call, which sums the partial counts from all workers and delivers the result to the coordinator. Per-vertex counts are aggregated using a second distributed shuffle.

2.4 Execution Details

The program was compiled and executed on the Slurm-managed cluster using the following commands. The code was compiled into an executable named `q2_optimized_mpi`.

```
# Compilation Command
mpic++ -std=c++11 -o q2_optimized_mpi q2_optimized.cpp

# Slurm Batch Script (run_job.sh)
#!/bin/bash
#SBATCH --job-name=4-cycle-optimized
#SBATCH --partition=debug
#SBATCH --nodes=1
#SBATCH --ntasks=16
#SBATCH --time=00:10:00
#SBATCH --output=resultsq2.txt
#SBATCH --error=benchmarkq2.txt
```

```
module purge
module load openmpi

echo "Running on $SLURM_NTASKS cores..."
mpirun -np $SLURM_NTASKS ./q2_optimized_mpi

# Submission Command
sbatch run_job.sh
```

2.5 Performance Evaluation

Performance was evaluated by running the optimized parallel implementation on a range of core counts (2, 6, 12, 24) across multiple graph datasets of varying sizes (N, M). The total execution time was measured as the maximum wall-clock time reported by any MPI process. This approach allows for a comprehensive analysis of how the implementation scales with both the number of cores and the size of the input data.

2.5.1 Performance vs. Problem Size

We evaluated the problem in randomly generated test cases, with the number of nodes between 50 and 600. As time complexity is $O(\min(n^3, n \cdot m))$, if number of edges is relatively small, we get done with the execution faster,which can be visible in the below plot.

Figure 4 shows the relationship between execution time and the number of vertices (N) in the graph, with a separate line for each core count. It is immediately clear that for any given problem size, increasing the number of cores results in a lower execution time.

The use of a logarithmic scale on the y-axis highlights an important trend: as the problem size N increases, the performance gap between different core counts widens, demonstrating the effectiveness of the parallel algorithm on larger datasets.

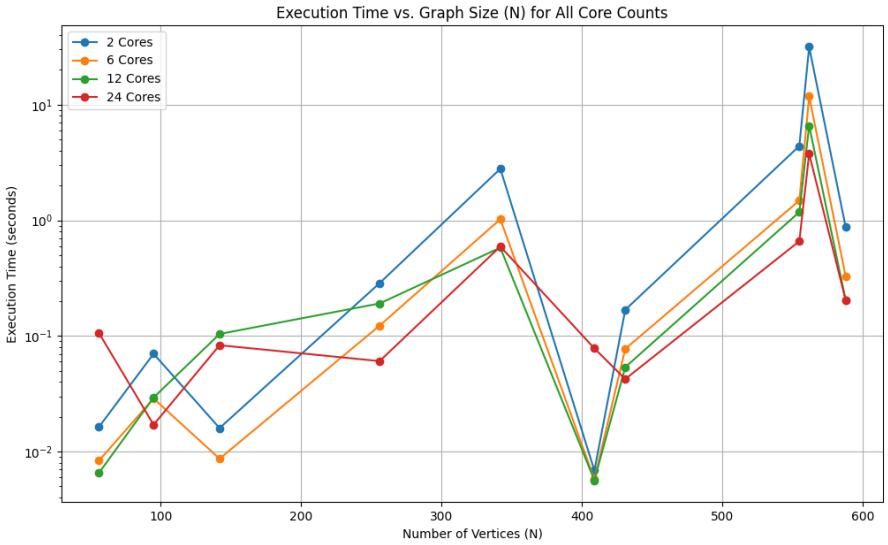

Figure 4: Execution Time vs. Graph Size (N) for All Core Counts.

2.5.2 Best-Case Performance Analysis

To understand the optimal performance achieved for each problem, Figure 5 plots the best execution time against the number of vertices (N). Each data point is annotated with the number of cores that achieved that best time.

An interesting observation is that for smaller graphs, the overhead of communication in a high-core-count run can make a lower core count faster. As the graph size increases, higher core counts become progressively more

efficient and necessary to achieve the best performance. This plot effectively visualizes the "sweet spot" of core count vs. problem size for this algorithm.

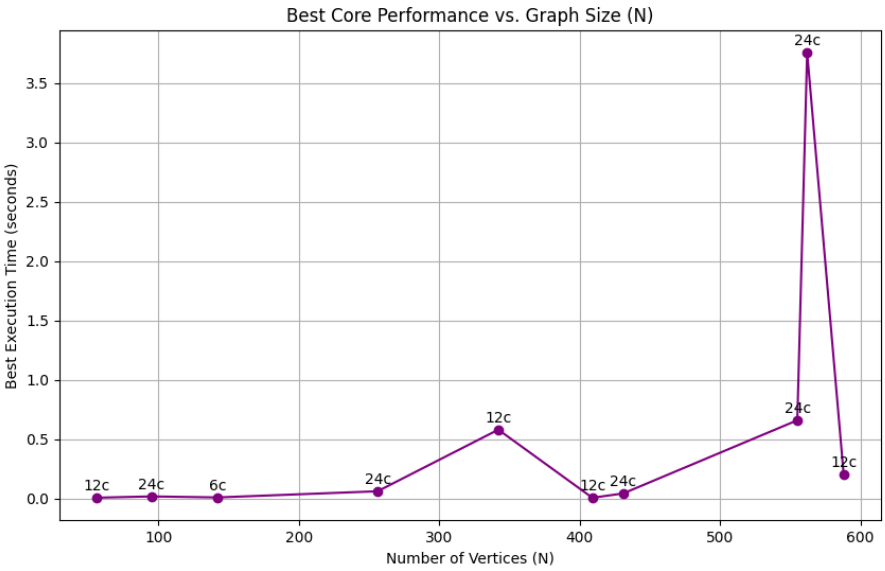

Figure 5: Best Core Performance vs. Graph Size (N).

2.5.3 Scalability Analysis

The scalability of the optimized implementation is excellent. By eliminating the master bottleneck and using a distributed shuffle (`MPI_Alltoallv`), the system avoids the single point of contention that plagued the initial design. This allows the program to effectively leverage additional cores, as seen in Figure 4.

The primary factor limiting perfect linear scalability is the communication overhead inherent in the all-to-all shuffle phase. As more processes are added, the complexity of this data exchange increases. However, because the algorithm uses direct binary communication and avoids string parsing, this overhead is minimized, allowing the computation to remain the dominant factor for the tested problem sizes. The result is a highly efficient and scalable solution that shows significant speedup as both the number of cores and the problem size increase.

2.6 Challenges and Limitations

- **Master Bottleneck:** The initial master-worker design proved to be a major limitation. All communication was funneled through rank 0, which quickly became overwhelmed, leading to poor performance (e.g., 30 seconds for a small graph). This necessitated a complete redesign to a decentralized model.
- **String Processing Overhead:** The initial reliance on string manipulation for communication was extremely inefficient. The cost of serialization, parsing, and memory allocation for strings was a significant performance penalty, which was resolved by converting all vertex IDs to integers.
- **Cluster Environment:** Initial runs faced environmental challenges, including disk quota limits (`No space left on device`) and understanding the specific Slurm partition and module names for the RCE cluster.

2.7 Conclusion and Future Work

This project successfully implemented a parallel 4-cycle counting algorithm using MPI. The performance analysis clearly demonstrates the limitations of a naive master-worker model and the significant benefits of a fully parallel design with a distributed shuffle and binary data communication. The final optimized version is scalable, efficient, and capable of handling much larger graphs.

For future work, the following improvements could be explored:

- **Hybrid Parallelism:** For multi-core nodes, a hybrid MPI+OpenMP model could be used. MPI would handle communication between nodes, while OpenMP threads would parallelize the computation (e.g., wedge generation) within each node, potentially reducing MPI communication overhead.
- **Distributed File I/O:** For extremely large graphs that do not fit in memory on a single node, the initial data loading could be parallelized using MPI-IO.
- **Load Balancing:** The current hash-based data distribution assumes a relatively uniform key distribution. For skewed graphs, this could lead to load imbalance. A more sophisticated partitioning scheme could be implemented to ensure work is distributed more evenly among processes.

3 gRPC

3.1 Problem Understanding

3.1.1 Problem Description

The Matrix Queries with Two Clients problem involves coordinating asynchronous row submissions from two independent client processes to a centralized server that constructs a combined matrix and responds to mathematical queries. The server must handle concurrent row submissions while providing real-time boolean responses to two query types: HasRankAtLeast(r) to check if matrix rank r, and HasDeterminantAtLeast(d) to verify if determinant has d . The challenge lies in maintaining thread-safe matrix operations while each client contributes one row at a time independently, requiring accurate boolean results on the dynamically evolving matrix structure.

3.1.2 Approach

The implementation uses a gRPC-based distributed architecture with three components: Protocol Buffer definitions specifying separate HasRankAtLeast and HasDeterminantAtLeast RPC methods with dedicated message types, a multi-threaded server using mutex locks for thread-safe matrix operations and numpy computations for rank/determinant calculations, and lightweight clients supporting independent operation via command-line client IDs (1 and 2). The server combines rows into a unified numpy array and performs mathematical computations server-side, returning boolean results directly to clients without requiring client-side processing. This design ensures asynchronous scalability with comprehensive error handling for dimension mismatches while delivering boolean query results as specified in the problem requirements

3.2 System Setup

Cluster Configuration: Single compute node with multi-core CPU architecture supporting concurrent client-server operations.
MPI Setup: Python gRPC framework with Protocol Buffers for inter-process communication providing asynchronous RPC support.
Development Environment: Python 3.12.5 with gRPC libraries, numpy for mathematical computations

3.3 Design and Implementation

3.3.1 Parallelization Strategy

The system employs an asynchronous client-server parallelization model built on gRPC's multi-threaded architecture. Two independent client processes operate concurrently, while the server handles incoming requests using a thread pool executor. Mutex-based synchronization ensures thread-safe execution of matrix operations. The server maintains a concurrent `ThreadPoolExecutor` with up to ten worker threads, allowing simultaneous row submissions and query requests without blocking operations.

3.3.2 Data Distribution Approach

Matrix rows are distributed incrementally across clients under a centralized aggregation model. Each client contributes individual rows that are dynamically merged into a unified matrix maintained on the server side. The server employs a shared storage structure (`self.matrix_rows[]`) and a contribution tracking mechanism (`self.client_contributions{}`) to preserve data provenance. Runtime validation ensures consistent matrix dimensions across all client submissions.

3.3.3 Communication Patterns

The implementation follows a request-response communication model defined by Protocol Buffers. Dedicated RPC methods support different operations: `SendRow()` for row submissions, `HasRankAtLeast()` for rank queries, and `HasDeterminantAtLeast()` for determinant queries. Each client establishes an independent gRPC channel to the server, enabling asynchronous interactions. Clients can submit rows and issue queries independently, while the server processes concurrent requests using thread-safe locking mechanisms to maintain matrix state consistency.

3.4 Execution Details

The following command lines were used during execution:

Protocol Buffer Generation:

```
python3 -m grpc_tools.protoc --python_out=. --grpc_python_out=. gRPC.proto
```

Server Startup:

```
python3 server.py
```

Client 1 Execution:

```
python3 client.py 1
```

Client 2 Execution:

```
python3 client.py 2
```

3.5 Challenges and Limitations

Challenges

- **Data Consistency:** Coordinating concurrent client row submissions while maintaining matrix dimension validation and client contribution tracking.
- **Error Handling:** Managing edge cases such as dimension mismatches and incomplete matrix operations across distributed clients.

Limitations

- **Memory Scalability:** $O(n^2)$ memory overhead for matrix operations, with complete reconstruction required for each query.
- **Computational Complexity:** Lack of incremental computation strategies forces complete matrix reprocessing for rank and determinant calculations.
- **Network Overhead:** The gRPC request-response pattern introduces latency for individual row submissions, preventing efficient batch operations.
- **Serialization Costs:** Protocol Buffer overhead for large floating-point matrix data reduces bandwidth utilization and impacts performance scalability.

3.6 Conclusion and Future Work

Conclusions

The Matrix Queries system successfully demonstrates distributed matrix operations using gRPC, with asynchronous clients achieving boolean query responses. The implementation validates thread-safe synchronization for concurrent operations and establishes a reliable foundation for distributed matrix processing.

Future Work

- **Performance:** Adoption of lock-free data structures and incremental computation algorithms.
- **Scalability:** Support for partitioned computation and efficient batch processing capabilities.
- **Features:** Integration of streaming updates, additional linear algebra operations, and ML-based query optimization.
- **Extensions:** Incorporation of secure computation protocols and fault-tolerant mechanisms.