Parallel and Distributed Computing

Project Report

Submitted to: Sir Fahad Shafique

Submitted By: Muhammad Ali Naveed
Hamza Omer
Ariyan Chaudhary

Parallel Implementation of the Bubble Sort Network using MPI and OpenMP

1. Introduction

This report presents the design, implementation, and analysis of a parallel algorithm based on the *bubble sort network* for generating parent permutations of the symmetric group Sn. The algorithm follows the model outlined in [Chitturi et al.], where permutations are organized using an adjacency network akin to sorting networks, and each permutation's parent is determined via swap operations governed by network rules.

We parallelized the algorithm using **MPI** for process-level distribution across nodes and **OpenMP** for multi-threaded computation within each MPI process. This hybrid model improves scalability and computation time for larger values of n, where the total number of permutations grows factorially.

2. Algorithm Description

The core algorithm determines, for each permutation in Sn, the parent permutations according to a layered, bubble sort-inspired swap network.

- For each permutation (vertex), n−1, parent permutations are computed by conditional swaps.
- The parent-finding logic implements complex rules as per the referenced paper, using position inversion arrays and ranking/unranking of permutations.
- The result for each permutation is stored as a CSV row containing the vertex ID, the permutation string, and the parent IDs for each transformation layer.

3. MPI + OpenMP Parallel Implementation

3.1 MPI Process Division

The factorial space Sn of permutations is evenly divided among the available MPI processes:

- Each process receives a distinct chunk of permutations to analyze.

- Load balancing is achieved using the division and remainder trick (`base = Nperm / size; rem = Nperm % size;`), ensuring fair distribution.

- Each process writes its results to a separate CSV file stored in `/tmp/results/`.

3.2 OpenMP Thread Division

Each MPI process internally uses OpenMP to further divide its chunk across threads:

- Threads independently compute parent permutations.

- Buffering is done locally per thread using a `vector<vector<string>> thread_lines`.

- A single write operation after the barrier ensures consistent file output with minimized I/O contention.

3.3 Timing and Output

- `MPI_Wtime()` is used to record per-process runtime.

Output files contain CSV data in the format:
`vertex_id,perm,T1,T2,...,T(n-1)` for all assigned permutations.

4. Correctness and Validation

- The implementation closely follows the specification and pseudocode of the original paper .

- We ensured correctness by comparing ranks and unranking outputs, verifying that the parent permutations correspond to expected transitions.

- The `is_identity()` function and custom `swap_symbol()` logic ensure accurate parent tracing.

5. Experimental Setup

5.1 Cluster Configuration

- **Total Nodes:** 3
- **Node Specs:**
    - Node A: 8 GB RAM
    - Node B: 16 GB RAM
    - Node C: 32 GB RAM
- **OS:** Ubuntu Server 22.04 LTS
- **MPI Implementation:** OpenMPI 4.1.5
- **Compiler:** GCC 11.3.0 with `mpicxx`
- **Networking:** Ethernet (some setup overhead noted)

5.2 Homelab Server

- **Single Node Specs:** 64 GB RAM, 12-core CPU
- **Results:** Better performance and lower latency due to unified memory access and no inter-node overhead.

6. Performance Analysis

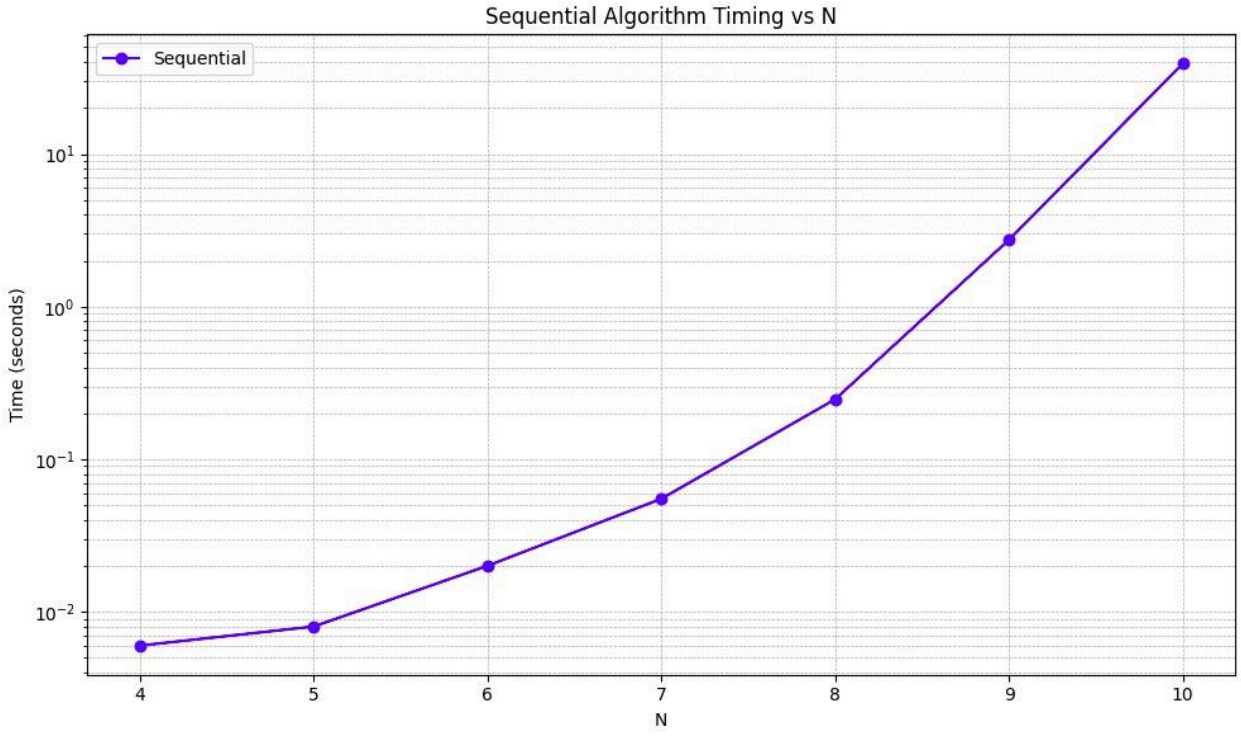
6.1 Tools Used

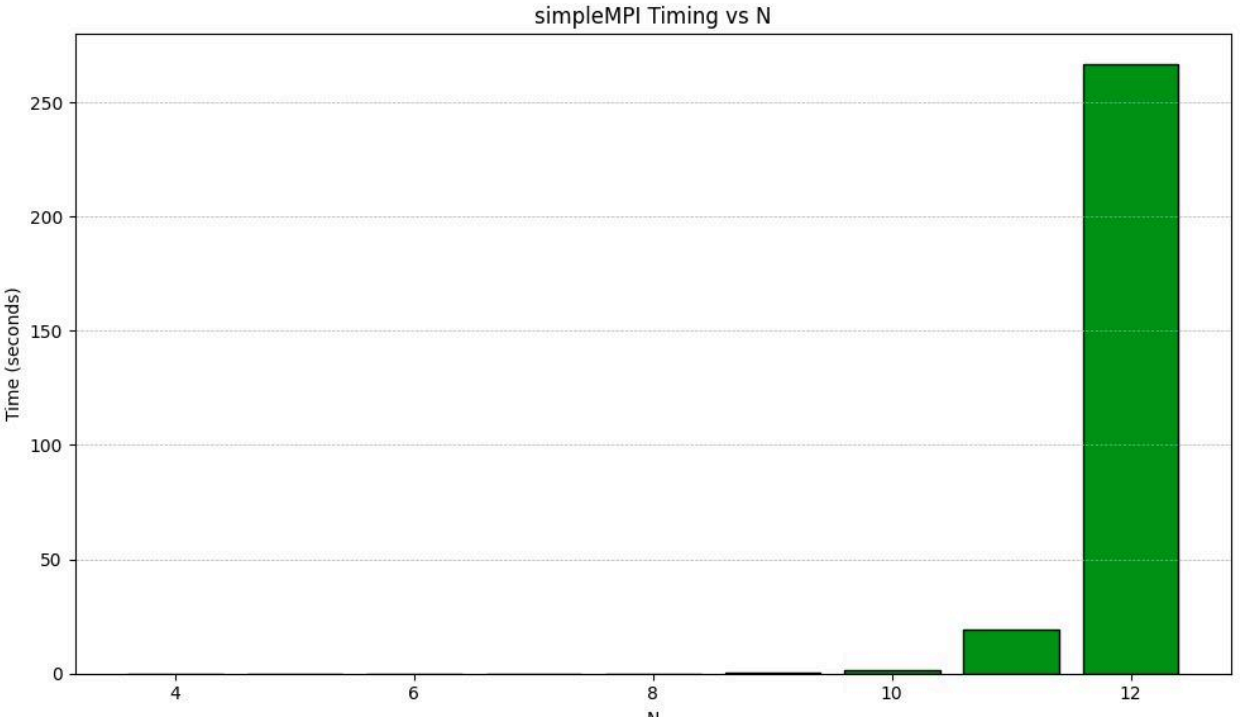
- `gprof` for profiling sequential versions.
- `MPI_Wtime()` and `std::chrono` for measuring time per process/thread.
- Plots and speedup charts will be inserted here.

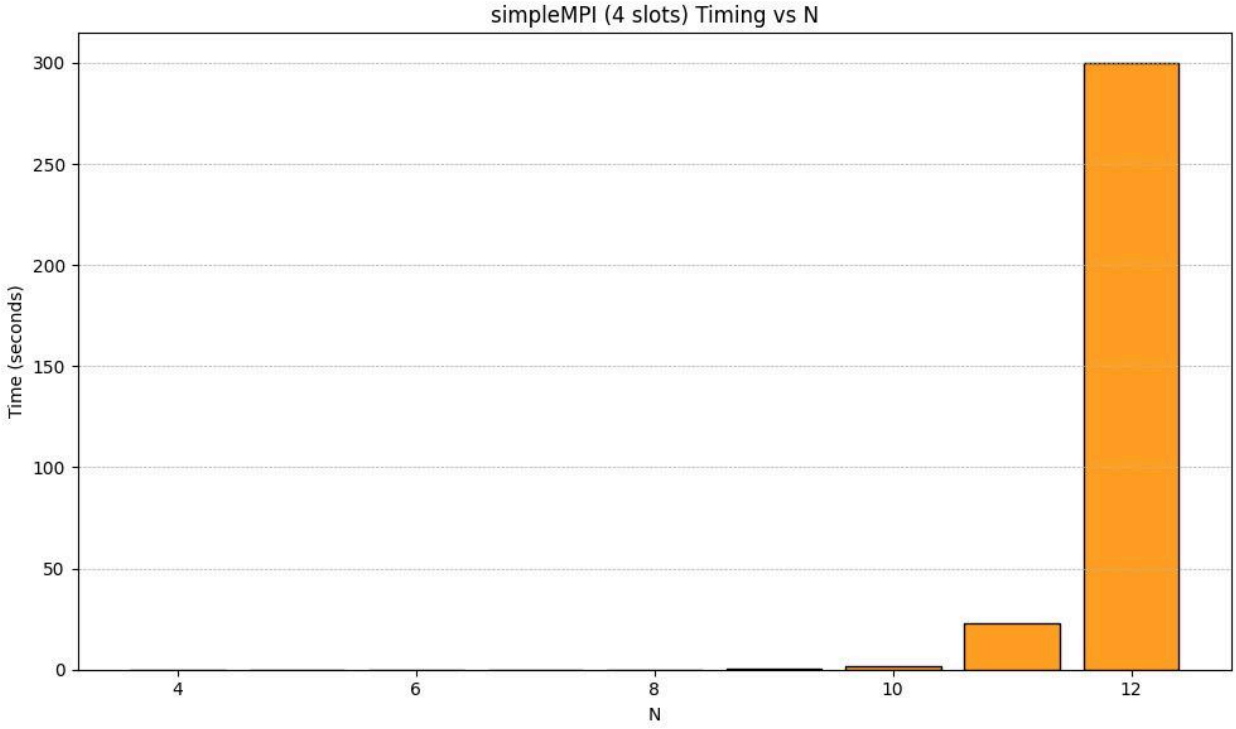
6.2 Scalability Observations

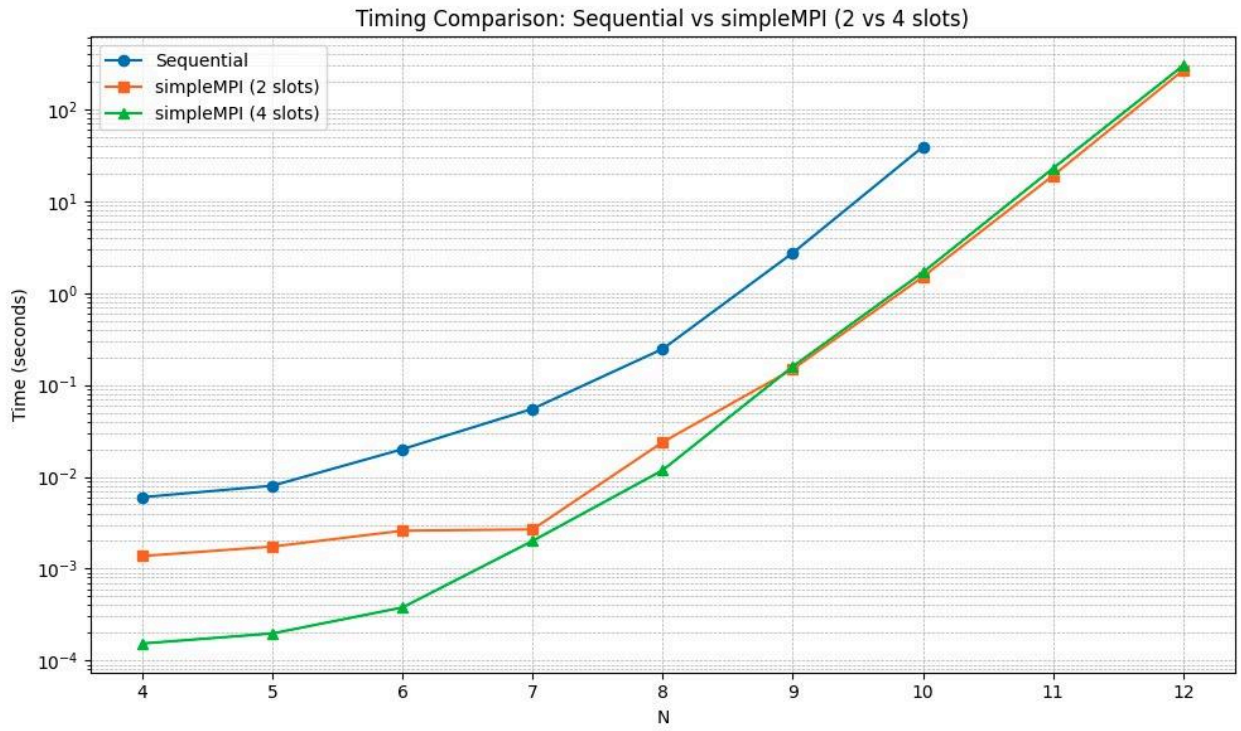
- For n=11:
    - **Sequential Code:** Crashes due to excessive memory.
    - **Parallel Code:** Completes successfully across all MPI processes.
- For n=12:
    - **Only 32 GB RAM Node** successfully completes execution.
    - **Other nodes** fail due to insufficient memory.

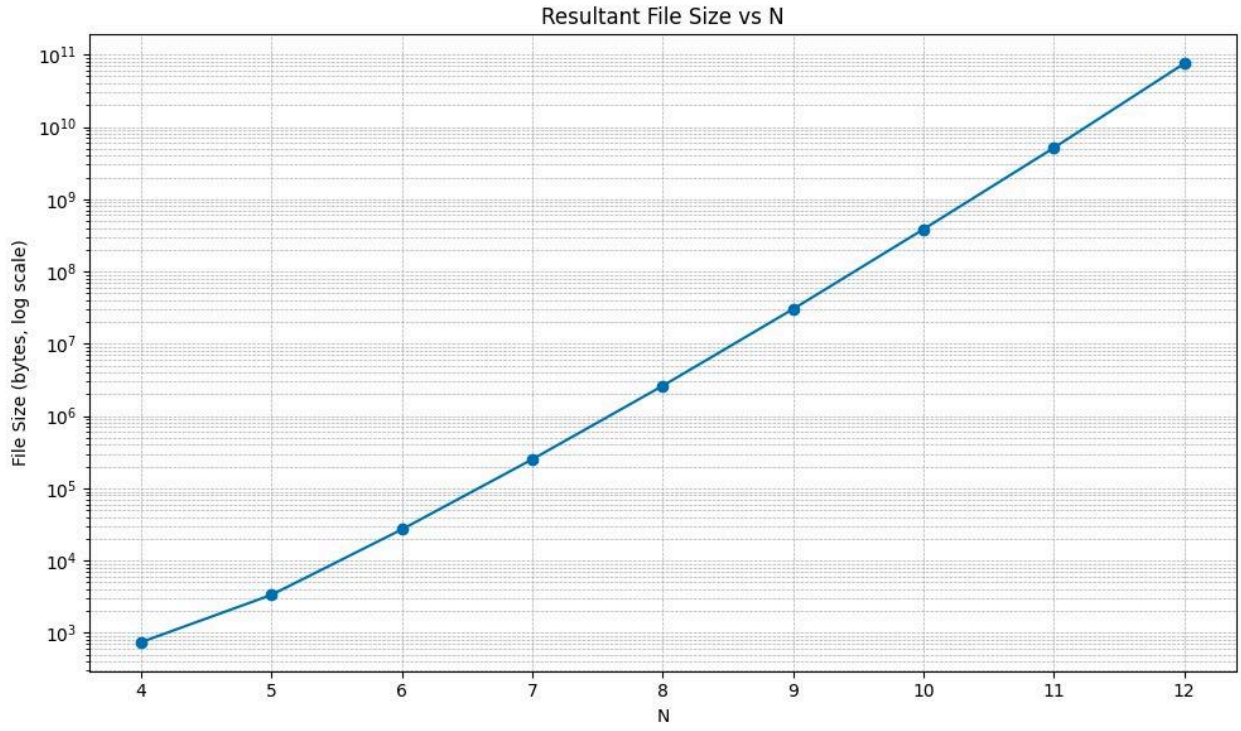
6.3 Chart: Execution Time vs N



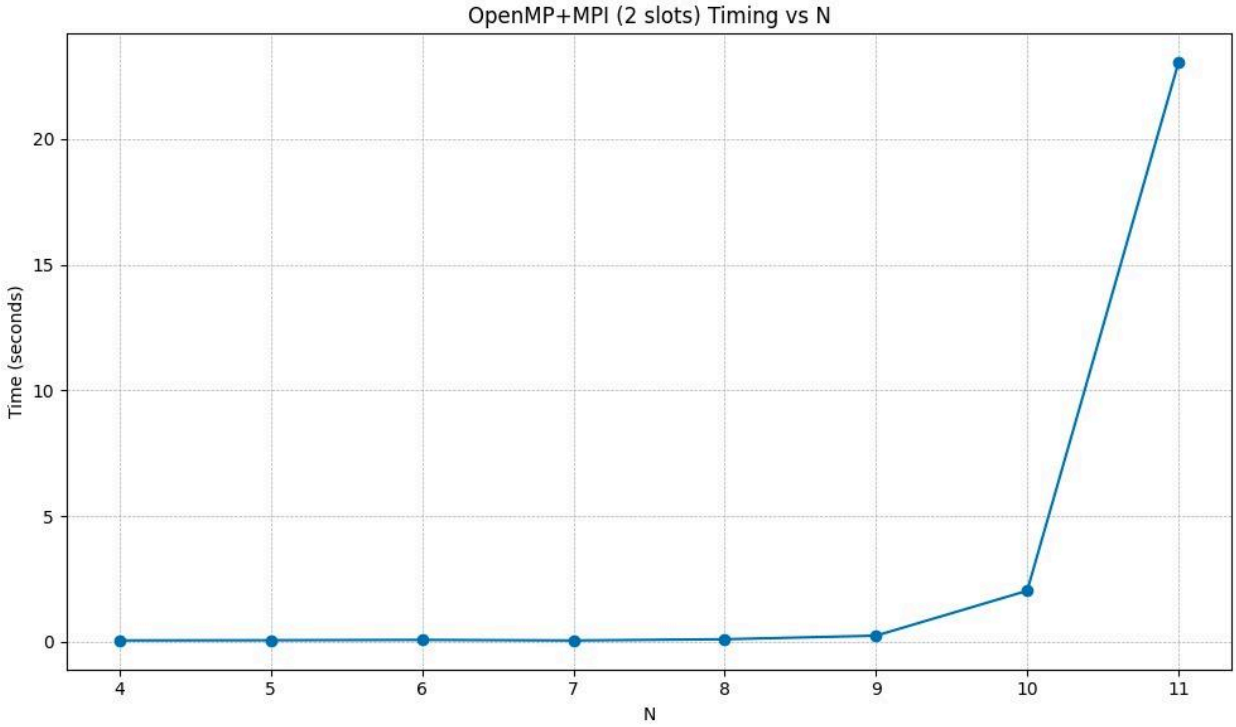

simpleMPI Timing vs N
Time (seconds)
0
50
100
150
200
250
4
6
8
10
12

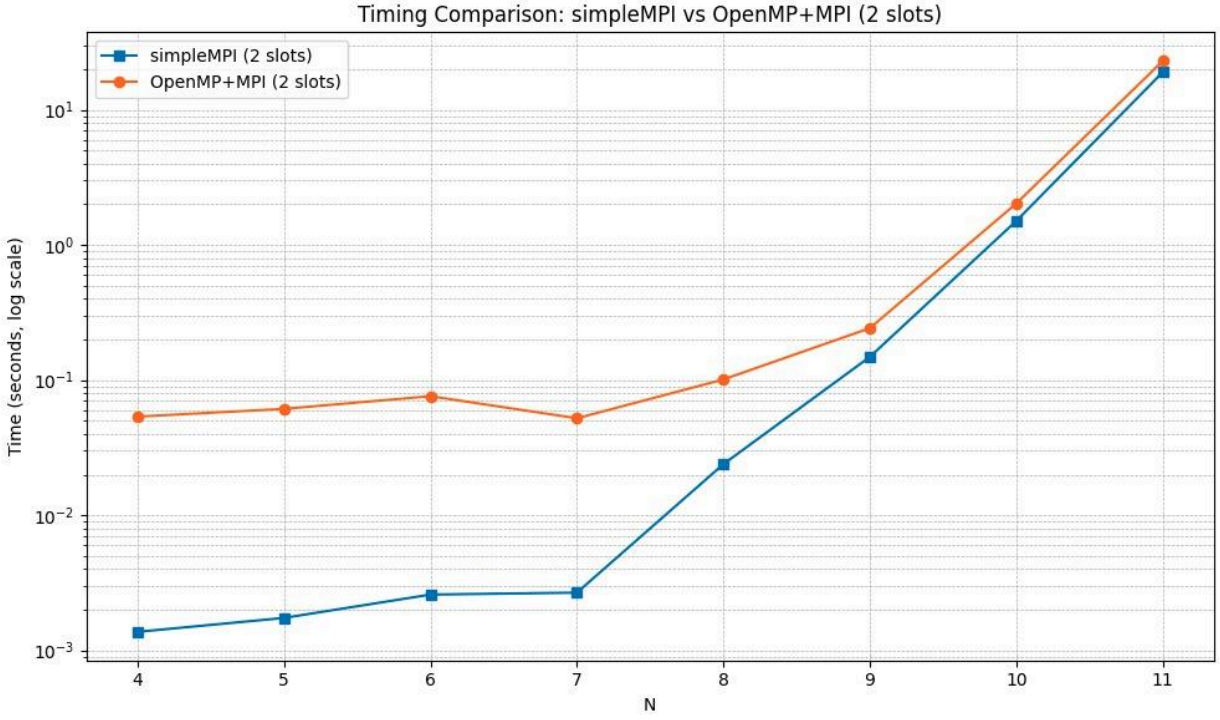

simpleMPI (4 slots) Timing vs N
Time (seconds)
0
50
100
150
200
250
300
4
6
8
10
12
N

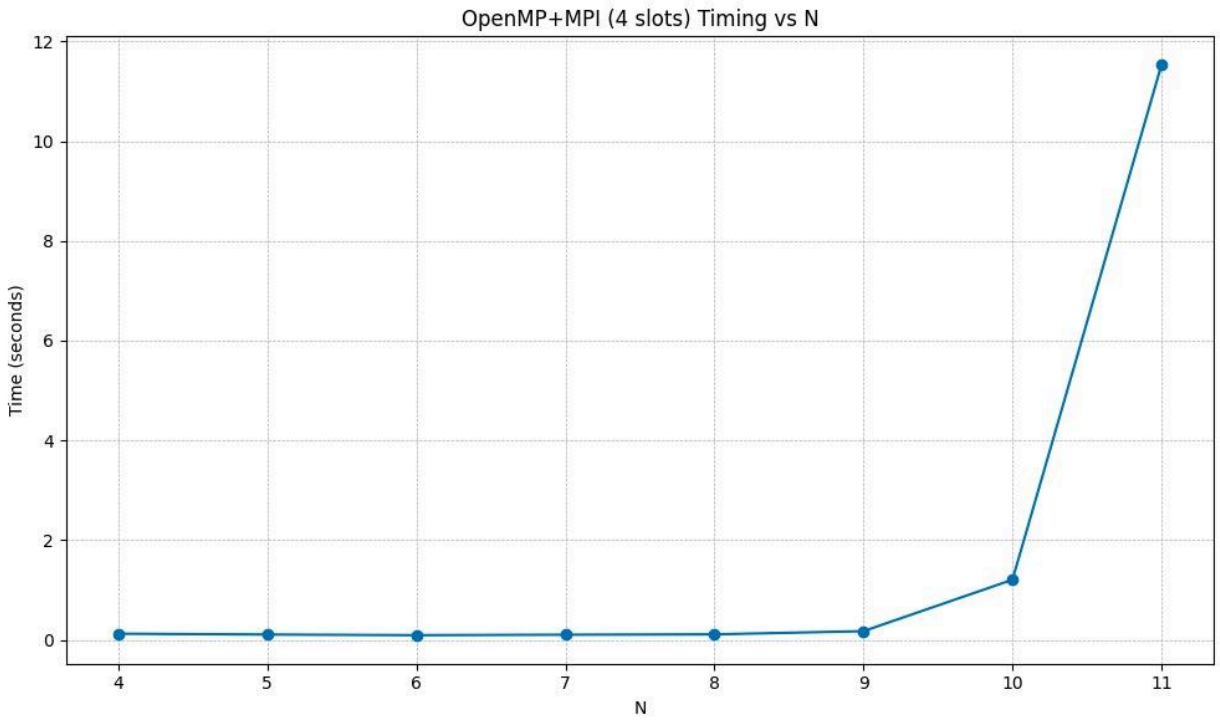


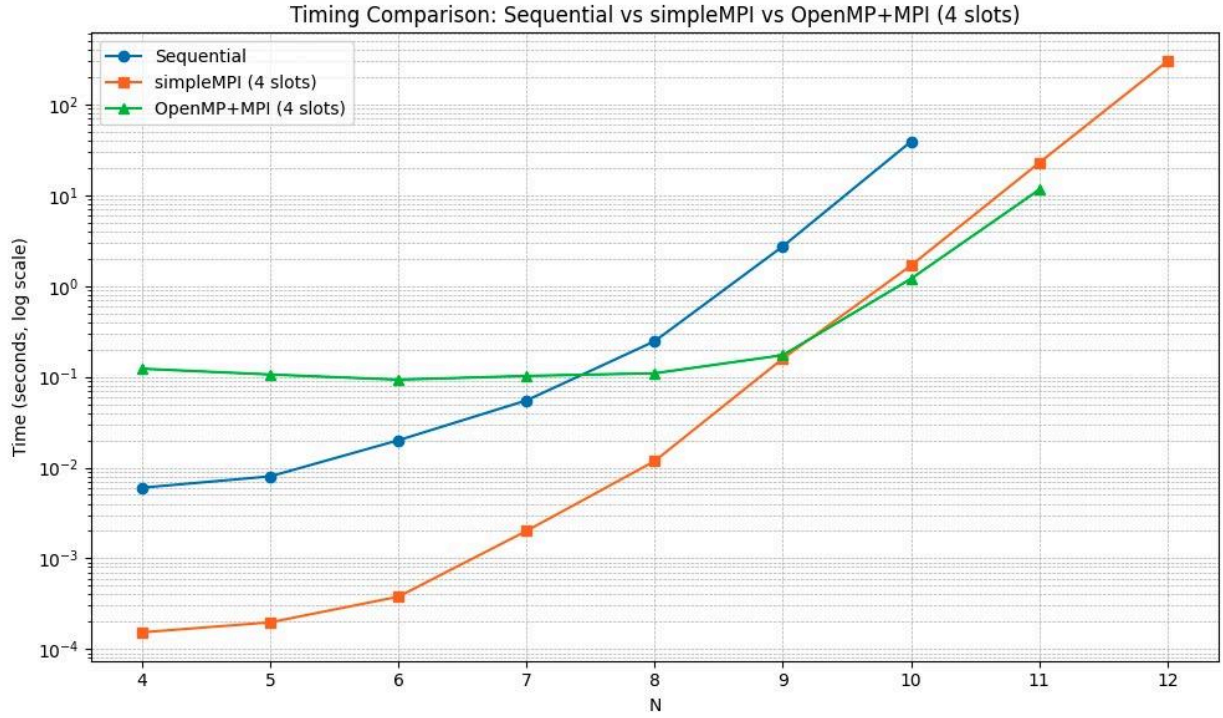


There were some storage issues Here.


OpenMP+MPI (2 slots) Timing vs N
Time (seconds)
N
0
5
10
15
20
4
5
6
7
8
9
10
11


Timing Comparison: simpleMPI vs OpenMP+MPI (2 slots)
simpleMPI (2 slots)
OpenMP+MPI (2 slots)
Time (seconds, log scale)
N
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
4
5
6
7
8
9
10
11





It can be seen openMP starts doing better from N = 10

6.4 Chart: Speedup Comparison

7. Observations and Challenges

- **High Setup Latency (20s)** observed on the MPI cluster due to networking overhead.
- **Homelab Performance** significantly better, attributed to single-node, high-bandwidth, and large shared memory.
- **I/O Bottlenecks** avoided using single-threaded final write after OpenMP region.

8. Conclusion

This work demonstrates the benefits of hybrid parallelism in computing complex permutation structures. We implemented a memory-optimized, MPI+OpenMP solution to compute parent networks for Sn permutations. The parallel code scales well and enables processing of problem sizes that are infeasible with a sequential approach.

Future work may focus on:

- Reducing I/O bottlenecks using MPI-IO or distributed file systems.
- Implementing a memory-efficient compressed output format.
- Further optimizing inter-process communication using shared memory within nodes.

Appendix

A. Compilation and Execution Instructions

```
mpicxx -O2 -fopenmp parallel_mpi.cpp -o parallel_mpi

mpirun --hostfile machines -np <num_processes> ./parallel_mpi <N>
```

B. Sample Output File Format

```
vertex_id,perm,T1,T2,...,T(n-1)

0,1-2-3-4,...,...
```