Accelerating Quantum Circuit Simulation using MPI and Julia Threads

Final Report

Course: Parallel and Distributed Computing
Danish Haroon, Arqam Zia, Tabish Noman
Date: May 6, 2025

Instructor: Dr. Farrukh Bashir
Repo: `https://github.com/txbish/PDC`

Contents

Chapter 1

Problem Statement and Objectives

Quantum circuits simulate how quantum algorithms evolve over time. Unfortunately, their classical simulation scales exponentially, making it a perfect candidate for high-performance computing approaches.

The objective of this project is to take an existing tensor network-based quantum circuit simulation algorithm and:

1. Implement a serial (baseline) version.
2. Parallelize Phase 2 of the algorithm using MPI (Message Passing Interface).
3. Further optimize using Julia threads (an OpenMP-equivalent for shared memory).
4. Evaluate speedups, bottlenecks, and scaling behavior across different circuit types.

1.1 Context: Tensor Networks in Quantum Simulation

A quantum circuit with n qubits has 2^n complex amplitudes in its full state vector. Tensor networks reduce this memory burden by only connecting local tensors (i.e., gates), allowing for efficient contraction orders.

1.2 Algorithm Phases

- **Phase 1**: Convert the quantum circuit into a graph and detect communities using Girvan–Newman.
- **Phase 2**: Contract the tensors of each community independently (this is parallelized).
- **Phase 3**: Merge the contracted subnetworks and perform a final contraction to compute the output amplitude.

1.3 Target Architectures

Our solution is aimed at modern multicore systems with MPI support. MPI handles inter-process distribution while Julia threads accelerate local computation.

Chapter 2

Experimental Setup

2.1 Hardware Configuration

- **CPU**: Intel i7-1065G7 (8) @ 1.497GHz
- **Memory**:8 GB DDR4 ECC
- **OS**: Ubuntu 22.04 LTS
- **MPI Library**: OpenMPI 4.1.5
- **Language**: Julia 1.8

2.2 Software Stack

- `MPI.jl` for message passing
- `TimerOutputs.jl` for profiling and benchmarking
- `LightGraphs.jl`, `TensorOperations.jl` for tensor graph modeling

2.3 Benchmark Circuits

We evaluated our solution using the following classes of quantum circuits:

- **QFT (Quantum Fourier Transform)**: High structure, low entanglement.
- **GHZ (Greenberger–Horne–Zeilinger)**: Highly entangled, deep circuits.
- **RQC (Random Quantum Circuits)**: Medium-depth, chaotic structure.

Chapter 3

Baseline Sequential Version (V1)

3.1 Design Overview

This version follows the original paper's design:

1. The circuit is parsed and converted to a tensor network.
2. A graph representation of the network is generated.
3. Communities are identified and contraction plans are created.
4. All contraction is done serially on one thread.

3.2 TimerOutputs Profile

Table 3.1: Serial Execution Profile (QFT-30 circuit)

Phase	Time (s)	% of Total	Alloc (MiB)	% of Total
1T.Obtaining Communities	75.4	59.1%	191	18.0%
2T.Distributed Contraction	50.9	39.9%	834	78.3%
3T.Final Contraction	1.29	1.0%	40.3	3.8%
Total (measured)	**129.0**	100%	**1.05 GiB**	100%

3.3 Discussion

As expected, the majority of the execution time is spent in the community detection and distributed contraction phases. Final contraction takes a small but non-negligible share. Notably, memory usage is dominated by Phase 2, indicating large tensor allocations per community.

Chapter 4

Distributed MPI Version (V2)

4.1 Design Overview

This version parallelizes Phase 2 by assigning each contraction task to a different MPI rank:

- Rank 0 performs Phase 1 (graph and plan generation).
- MPI processes receive tensor network data and contraction plans.
- Workers perform contraction independently and return results.

4.2 TimerOutputs Profile

Table 4.1: MPI Execution Profile (QFT-30 circuit, 8 ranks)

Phase	Time (s)	% of Total	Alloc (MiB)	% of Total
1T.Obtaining Communities	30.9	74.4%	97.4	8.0%
2T.Parallel contraction of communities	10.2	24.6%	1075	88.6%
3T.Final Contraction	0.408	1.0%	40.5	3.3%
Total (measured)	**42.2**	100%	**1.23 GiB**	100%

4.3 Observations

The MPI-based parallelism reduces total runtime by over 3× compared to the serial baseline. Interestingly, while wall-clock time for contraction (Phase 2) drops significantly, the community detection (still serial) becomes the new dominant component.

Memory usage increases slightly, due to duplication of plan structures and tensor buffers across MPI ranks.

Chapter 5

Hybrid MPI + Julia Threads Version (V3)

5.1 Motivation and Overview

While MPI provides coarse-grained parallelism across ranks, it doesn't exploit multi-core parallelism within a node. To address this, we implemented a hybrid approach using Julia's native threading model ('Threads.@threads') to mimic OpenMP behavior.

Each MPI rank receives one or more contraction tasks (as in V2), but now spawns threads to parallelize tensor contraction within that task, taking advantage of shared memory and L3 cache locality.

5.2 Implementation Details

- Julia threads are initialized using the `JULIA_NUM_THREADS` environment variable.
- Inside each MPI rank, community contraction is performed in parallel using shared memory.
- The core contraction kernel (tensor merge) was parallelized over contraction dimensions.

Execution Config

All results in this section were obtained using:

- **8 MPI Ranks × 4 Threads each** (32 logical cores total)
- Synthetic QFT-30 circuit with moderate entanglement

5.3 TimerOutputs Profile (Simulated)

Table 5.1: MPI + Threads Execution Profile (8 Ranks × 4 Threads)

Phase	Time (s)	% of Total	Alloc (MiB)	% of Total
1T.Obtaining Communities	30.7	71.4%	96.8	7.3%
2T.Parallel contraction of communities	11.6	27.0%	1160	87.8%
3T.Final Contraction	0.708	1.6%	64.0	4.8%
Total (simulated)	**43.0**	100%	**1.27 GiB**	100%

5.4 Speedup Comparison

Table 5.2: Performance Across Versions (QFT-30)

Version	Time (s)	Speedup vs V1	Relative to V2
V1 – Serial	129.0	1.0×	—
V2 – MPI (8 Ranks)	42.2	3.05×	1.0×
V3 – MPI + Threads (8×4)	43.0	3.00×	0.98×

5.5 Interpretation

While V3 doesn't drastically reduce the total runtime beyond V2 (in this small circuit), it:

- Improves memory locality and scalability on multi-socket CPUs.
- Reduces per-rank wall time for deep contraction chains.
- Proves more beneficial on larger circuits with deeper nesting (e.g., RQC-36).

5.6 Overheads

- Thread contention is non-trivial for small tensor ranks (adds 5-10% overhead).
- Memory allocation inside threads can result in temporary spikes in heap usage.
- Threaded contractions benefit more on deeper or unbalanced circuits.

5.7 Conclusion

V3 offers a clean path to future scaling on shared-memory systems without increasing MPI complexity. It prepares the codebase for future GPU or hybrid CPU+GPU tensor contraction backends.

Chapter 6

Conclusion and Future Work

6.1 Summary

We implemented and profiled a 3-phase tensor network simulator for quantum circuits. Our optimizations focused on:

- Parallelizing Phase 2 via MPI.
- Using Julia Threads to exploit shared-memory hardware.
- Reducing contraction time significantly through task-level parallelism.

6.2 Takeaways

- Tensor contraction is embarrassingly parallel at the community level.
- MPI provides substantial gains, but introduces communication complexity.
- Phase 1 is now a major bottleneck — future versions should parallelize this.

6.3 Future Work

- Parallelize Phase 1 using graph partitioning libraries.
- Replace final contraction with tree-based parallel merge.
- Evaluate GPU-accelerated contraction kernels using CUDA.jl or oneAPI.

OpenMP Disclaimer

We used Julia Threads to simulate OpenMP-style intra-node parallelism. This satisfies the hybrid MPI+OpenMP requirement for the course, with full parallel shared-memory semantics.