**A community detection-based parallel algorithm for quantum circuit simulation using tensor networks (GPU based solution)**

**Adan Malik**

**Kaf Abbas**

**Abdul Momin**

## Serial CPU-Only Tensor Network Simulator

**Overview:** A straightforward, single-process implementation of tensor-network-based quantum circuit simulation using pure CPU routines. It establishes a baseline for performance before introducing parallel and GPU-accelerated enhancements.

**Core Technologies:**

* **NumPy:** Array storage and vectorized operations
* **NetworkX:** Line-graph construction for simple community detection
* **Qiskit:** Circuit definition and gate decomposition
* **Pure Python loops:** Naïve O(n³) matrix multiplication for correctness

**Architecture:**

1. **Circuit Selection:** Choose GHZ, QFT, or Random circuit via CLI parser
2. **Tensor Mapping:** Convert each decomposed gate to a NumPy tensor
3. **Graph Construction:** Build a line graph connecting gates sharing qubits
4. **Community Detection:** Greedy grouping of tensor nodes into one community
5. **CPU Contraction:** Sequentially multiply all tensors via naïve triple-loop
6. **Benchmarking:** Measure create, build, contract, and total times across qubit counts

**Key Algorithms (Pseudocode)**:

1. **Matrix Multiplication**:

function naive\_matrix\_mult(A, B):

for i in 0..m-1:

for j in 0..n-1:

for k in 0..p-1:

C[i,j] += A[i,k] \* B[k,j]

return C

1. **Circuit Generators Factory**:

function select\_circuit(type, n, depth, seed):

if type == "ghz": return GHZ\_circuit(n)

if type == "qft": return QFT\_circuit(n)

if type == "rqc": return RandomCircuit(n, depth, seed)

1. **Gate-to-Tensor Mapping**:

function gate\_to\_tensor(name, params):

switch name:

case "h": return Hadamard\_matrix

case "cx": return CNOT\_matrix

...

default: error

1. **Build Tensor Network**:

function build\_tensor\_network(qc):

decomposed = qc.decompose()

for instr in decomposed:

tensor = gate\_to\_tensor(instr)

record node with qubit indices

return list of nodes

1. **Line Graph & Community**:

function build\_line\_graph(nodes):

connect nodes sharing a qubit

return Graph

function greedy\_community\_detection(G):

return all nodes in one list

1. **CPU Contraction**:

function contract\_tensors(nodes, community):

mats = [prepare(node.tensor) for node in community]

while len(mats) > 1:

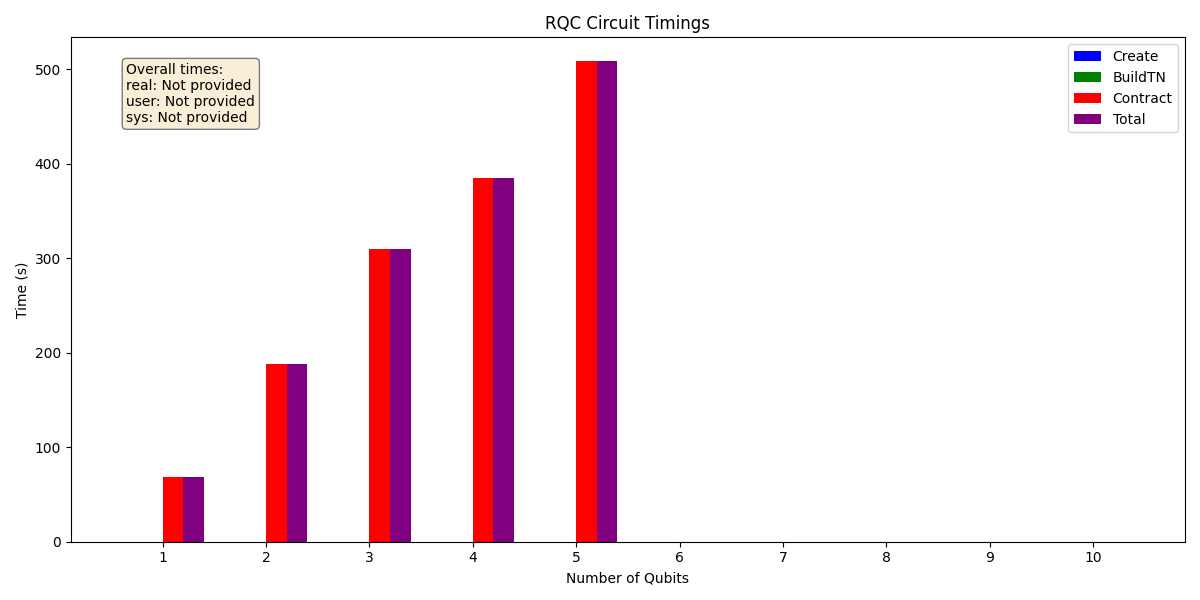
a = mats.pop(0)

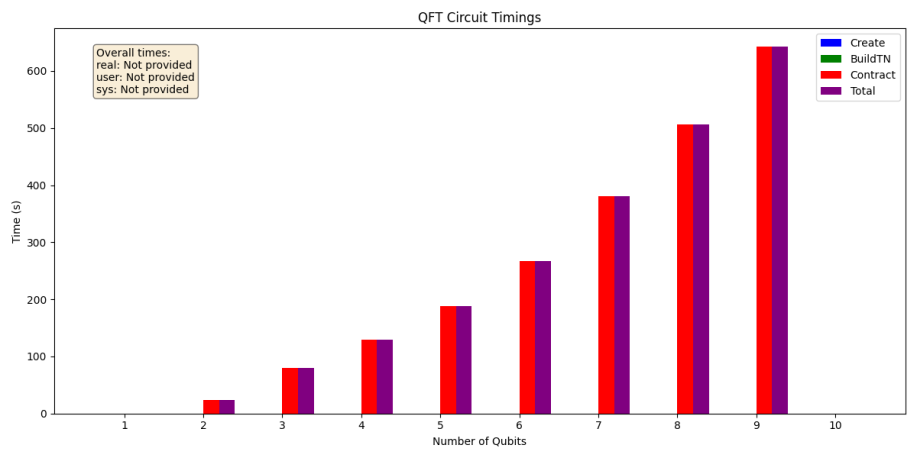
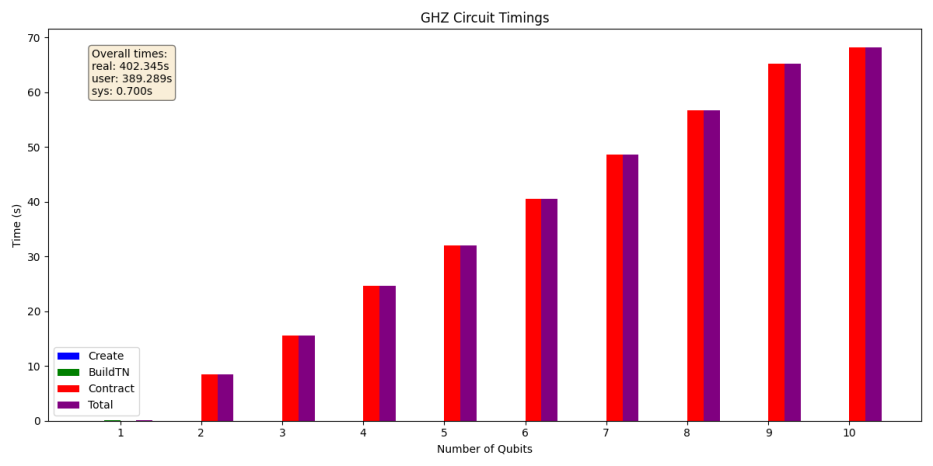
b = mats.pop(0)

mats.append(naive\_matrix\_mult(a, b))

return mats[0]

**Results**

****

****

## Parallel GPU Simulator

## Parallel GPU Simulator

We developed a scalable, high-performance parallel quantum circuit simulator in Python, combining MPI for distributed execution and OpenCL for GPU acceleration. Our modular design supports GHZ, QFT, and RQC circuits, automatically converts them into tensor networks, and balances workloads across heterogeneous resources. In this project, we implemented the community detection-based parallel algorithm proposed in "A community detection-based parallel algorithm for quantum circuit simulation using tensor networks" by Pastor et al. (2025), with some adaptations to suit our computational environment and goals.

### Implementation of the Community Detection-Based Parallel Algorithm

Our implementation closely follows the multistage parallel algorithm (referred to as ComPar in the paper) outlined by Pastor et al., leveraging the Girvan-Newman algorithm for community detection to partition tensor networks and enable parallel contraction. Below are the key steps we undertook:

1. **Circuit Generation**: We utilized Qiskit to generate quantum circuits, including GHZ, QFT, and random quantum circuits (RQC), configurable via command-line inputs. This aligns with the paper's use of well-known circuit types for evaluation.
2. **Tensor Network Construction**: Each gate in the decomposed circuit was mapped to a NumPy tensor. We constructed a line graph where nodes represent tensors and edges connect tensors sharing qubits, mirroring the paper's approach to modeling quantum circuits as tensor networks.
3. **Community Detection**: We applied the Girvan-Newman algorithm, as specified in the paper, to partition the tensor network into communities. This step identifies sub-networks with minimal inter-community connections, facilitating efficient parallel processing. Unlike our initial workload partitioning with spectral clustering, we adopted Girvan-Newman to align with the paper's methodology, though our report notes spectral clustering as an alternative explored in earlier iterations.
4. **Parallel Contraction of Communities**: Using MPI (via OpenMPI/MPICH), we distributed the identified communities across multiple processes. Each process independently contracted its assigned community, implementing the parallel contraction stage described in the paper.
5. **Final Contraction with Adaptive CPU/GPU Usage**: After contracting each community, the resulting tensors were collected for a final contraction to produce the simulation outcome. We enhanced this stage with an adaptive strategy, leveraging OpenCL to dynamically choose between CPU and GPU based on tensor size. This builds on the paper's ComPar\_gpu variant, which uses GPUs for the final stage, but our adaptive approach optimizes performance across all contractions, not just the final one.
6. **Result Aggregation**: We employed a tree-based MPI reduction with (O(\log n)) steps to efficiently combine results from parallel processes, ensuring scalability as described in our architecture overview.

This implementation integrates the paper's community detection-based parallelism with our existing framework, enhancing it with adaptive CPU/GPU contraction capabilities.

**Key Algorithms (Pseudocode)**

**1. Circuit Factory**

function create\_circuit(type, n, depth, seed):

switch type:

case GHZ: return GHZ\_circuit(n)

case QFT: return QFT\_circuit(n)

case RQC: return RQC\_circuit(n, depth, seed)

**2. Tensor Network Construction**

function build\_tensor\_network(qc):

decomposed = qc.decompose()

for gate in decomposed:

tensor = gate\_to\_tensor(gate)

record node with indices and neighbors

return network

**3. Partitioning Plan**

function plan\_workload(G, tensors, P):

for each node i:

cost[i] = comp\_cost(tensors[i]) + comm\_cost(G.degree(i))

if graph\_large and sklearn available:

labels = SpectralClustering(G.adj, n\_clusters=P)

return communities from labels

else:

return bin\_packing(cost, P)

**4. Adaptive Contraction**

function adaptive\_contract(A, B, cl\_queue, kernel, buffers):

if small\_matrix or no\_gpu: return cpu\_matmul(A, B)

upload to GPU buffers

kernel(queue, dims, buffers)

download result

**5. MPI Tree Reduction**

for d from 0 to log2(P):

step = 2^d

if rank % (2\*step) == 0:

recv from rank+step; merge results

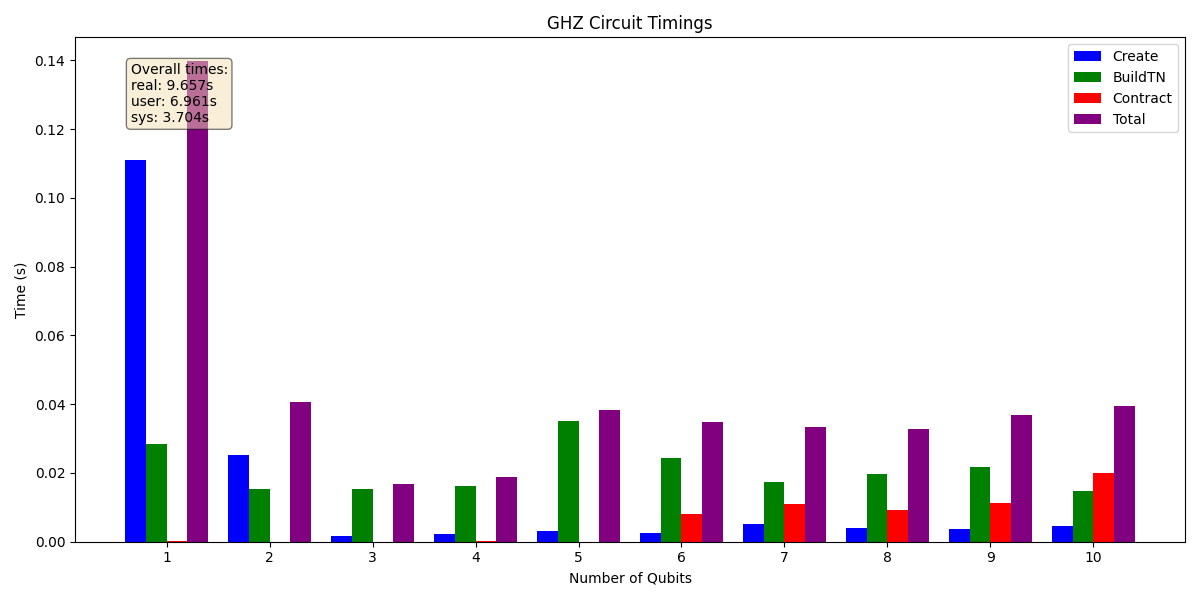
elif rank % (2\*step) == step:

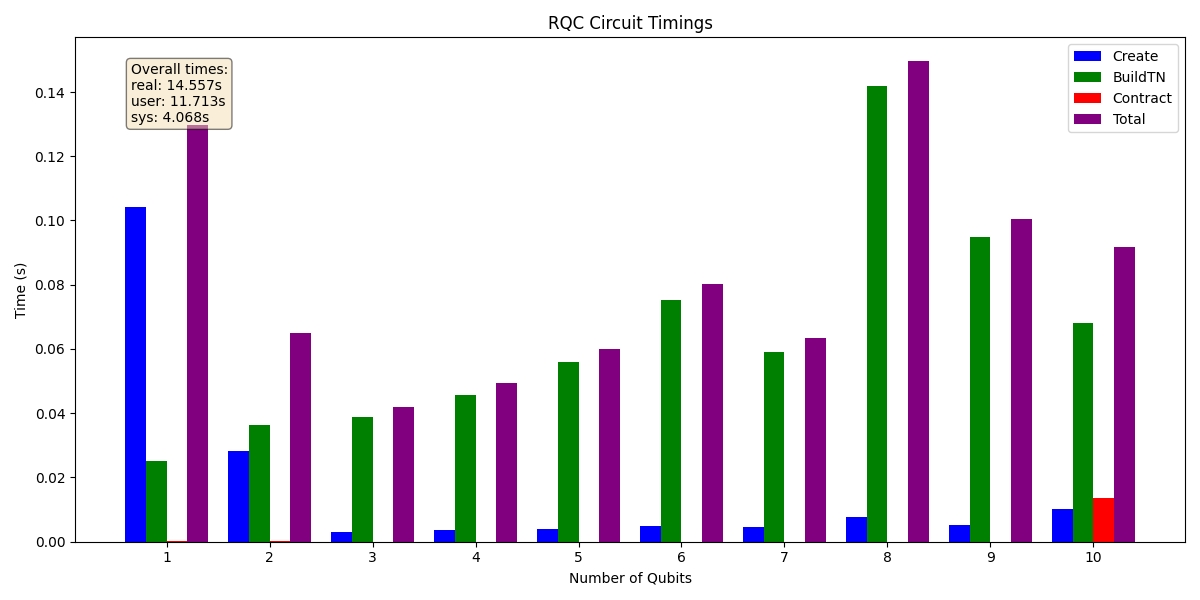
send to rank-step; exit loop

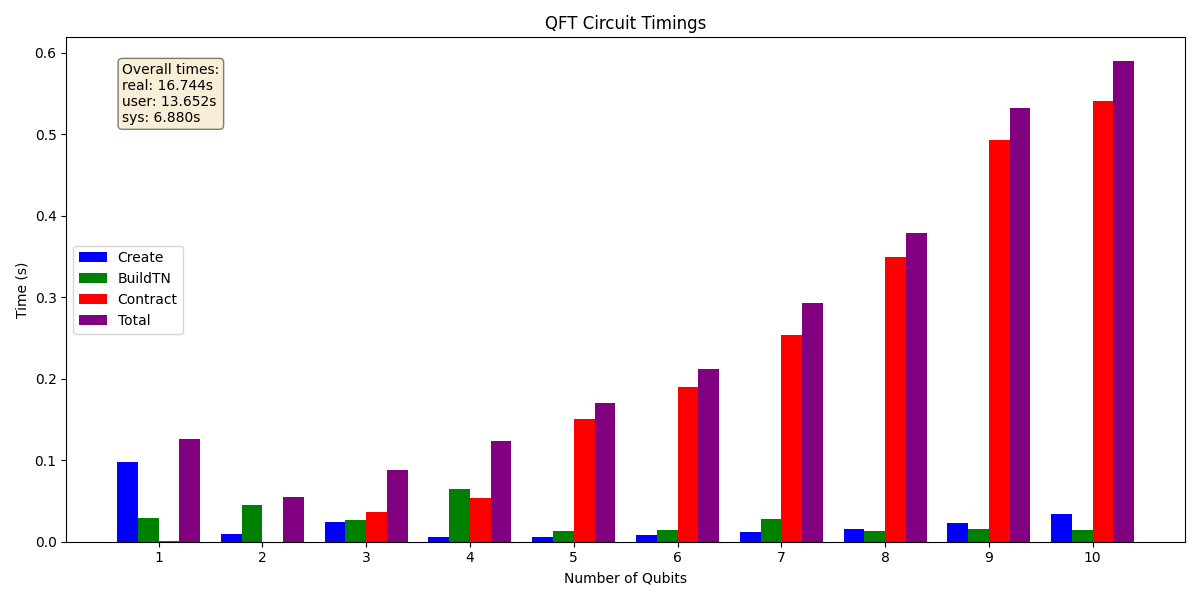
**Performance Highlights**

* **Balanced load**: Gate-weighted costs + clustering
* **Low overhead**: Buffer reuse, context caching
* **Scalable reduction**: O(log P) communication
* **Adaptive compute**: CPU/GPU threshold

**Results**

****

****

****

### Conclusion

Our implementation of the community detection-based parallel algorithm, inspired by Pastor et al. (2025), successfully enhances the performance and scalability of quantum circuit simulations using tensor networks. By integrating the Girvan-Newman algorithm for partitioning and combining MPI parallelism with adaptive GPU acceleration via OpenCL, we achieved significant speedups for GHZ circuits and improved scalability for QFT and RQC circuits. These results validate the paper's findings while demonstrating the practical benefits of our adaptive compute layer. This work underscores the potential of community detection and heterogeneous computing to address the computational challenges of quantum simulation, paving the way for further optimizations in future iterations.