Parallelization of Quantum State Vector Simulator

Atit Pokharel*, Nabin K. Singh*,
*Department of Electrical and Computer Engineering, UAH

1 Introduction

Advancements in quantum computing have accelerated in recent years at an unprecedented rate. Certain classes of problems are unsolvable on classical computers, like quantum chemistry, multi-variable optimization, and specific machine-learning tasks [1, 2]. Progress in theory and algorithms is rapid because of a large research community and industrial involvement. Despite this huge leap forward in research papers and algorithms, practical realization of large-scale and fault-tolerant quantum computing platforms has not reached [3]. As a result, researchers rely on quantum state vector simulators. They are mathematical tools that allow the study of quantum algorithms and quantum circuit behavior using classical computing resources [4].

A quantum state vector simulator represents the quantum state of an n-qubit system as a complex vector of size 2^n. This representation is supported by the Copenhagen interpretation [5] and enables tracing of the exact state evolution in a quantum circuit. Numerous simulators have been developed over the years, including Qiskit Aer [6] from IBM, PennyLane [7], TorchQuantum from MIT [8], Cirq by Google [9], etc. However, these simulators have a fundamental scalability challenge. For example, simulating a mere 30 qubits requires storing and processing over a billion complex numbers. Both the memory usage and computational cost grow exponentially with the number of qubits and circuit depth. For example, simulating a mere 30 qubits requires storing and processing over a billion complex numbers. Even shallow and moderate circuit sizes' simulation takes a tremendous time on a classical computer. With single-threaded execution, iterating through these massive arrays to calculate matrix products becomes extremely slow. This latency limits the ability of researchers to develop complex and useful circuit designs and validate them.

This project explores the use of OpenMP [10] for parallel computing to accelerate quantum state-vector simulation on multi-core CPUs. As mentioned, quantum states of an n-qubit system can be represented as complex state vectors. In these vectors, the actions of quantum gates mean simply some linear transformation derived from Pauli matrices [11]. Thus, we first define a state vector simulator in C++ that is able to simulate single-qubit and two-qubit gates through direct matrix operations. Then, as most of these tasks are matrix operations, the simulator is parallelized using OpenMP by exploiting the independent and data-parallel structure of gate applications. The independent subsets of state amplitudes can be calculated and updated in parallel.

1.1 Goals and Objectives

The main objective of this project is to bridge the gap between computationally intensive quantum simulations and efficient parallel computing resources. They are summarizes as follows

- **Runtime Reduction:** Accelerate the computation of quantum state vector simulations by parallelizing the resource-intensive gate operations using OpenMP.
- **Modular Framework Development:** Design a modular simulation tool that easily allows the addition of new quantum gates and different circuit structures.
- **Scalability Analysis:** Evaluate how the implementation scales with an increasing number of qubits by profiling Execution time (T_p), Speedup (S_p), Cost (C_p), and Parallel Efficiency (E_p).
- **Performance Benchmarking:** Demonstrate the performance gains by comparing the above metrics of the parallelized approach against serial methods on multi-core architecture.

2 Primers on Quantum Computing and its Simulation

To understand the computational load of a simulator, we define the mathematical representation of quantum states and operations in this section.

Qubits and State Vectors: Anyone in computer-system academia or industry must have heard multiple times that "a classical bit exists in a state of 0 or 1, whereas a qubit exists in a superposition of both." If we write this mathematically, the state of a single qubit is represented by a column vector of two complex numbers, $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where $|\alpha|^2 + |\beta|^2 = 1$. For a system of N qubits, such a state space grows exponentially. The combined state is represented by the tensor product of individual qubit states, resulting in a vector of size 2^N, as

$$|\Psi\rangle = \alpha_0|00\ldots0\rangle + \alpha_1|00\ldots1\rangle + \cdots + \alpha_{2^N-1}|11\ldots1\rangle$$

Pauli Matrices and Fundamental Gates: The building blocks of quantum operations are the Pauli matrices. These are 2×2 complex matrices that form a basis for the space of single-qubit operations. In this project, we explicitly implement the following gates, as

- **Identity (I):** Typical identity matrix, leaves the qubit unchanged.

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

- **Pauli-X (X):** Similar to the classical NOT gate (also called bit flip).

$$X = \sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

- **Pauli-Y (Y):** Combines bit flip and phase flip.

$$Y = \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

- **Pauli-Z (Z):** Flips the phase of the $|1\rangle$ component.

$$Z = \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

- **Hadamard (H):** Creates superposition from basis states.

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

In general, we represent them as U, since they are unitary. These gates acts on a state vector $|\psi\rangle$ and the new state $|\psi'\rangle$ is calculated via matrix-vector multiplication:

$$|\psi'\rangle = U|\psi\rangle$$

An example: The main challenge in simulation arises when applying operations to multi-qubit systems. If a gate acts on specific qubits while keeping others unchanged (like the Identity matrix I), the process is represented using the tensor product ($\otimes$)(Kronecker product). For example, applying a Pauli-X gate to the first qubit of a two-qubit system can be represented as:

$$U_{total} = X \otimes I = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{1}$$

Computing these products allows for parallelization because the individual elements of the resulting large matrix can be computed independently.

2.1 A benchmark for us: Variational Quantum Circuit

To evaluate the performance of our parallel simulator, we choose Variational Quantum Circuits (VQCs) as a benchmark. VQCs are a type of quantum algorithm used widely in Quantum Machine Learning (QML) [12] and optimization tasks (e.g., VQE, QAOA). They are also referred to as Parameterized Quantum Circuits (PQCs).

A typical VQC consists of a fixed gate structure (the ansatz). Here, it is essential to notice that the above-mentioned gate matrices can also be represented as the complex exponentiation of Pauli matrices ($R_P(\theta) = e^{-i\theta P/2}$).

$$R_x(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$$

$$R_z(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$$

Here, the gate parameters (i.e., the rotation angles θ) are tunable/trainable parameters and the goal of a VQC is to optimize these θ values to approximate some function (similar to a neural network in the classical domain). Such a circuit typically alternates between:

1. **Rotation Layers:** Single-qubit rotation gates (e.g., $R_x(\theta), R_y(\theta)$) applied to all qubits.
2. **Entanglement Layers (CNOT):** A fundamental two-qubit gate used to create entanglement. In Fig. 1, the two-qubit connection with an $\oplus$ connection is the CNOT gate operation. It flips the target qubit if and only if the control qubit is $|1\rangle$.

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Fig. 1 shows a typical structure of a VQC, where a fully entangled 5-qubit VQC with three arbitrary layers is shown.

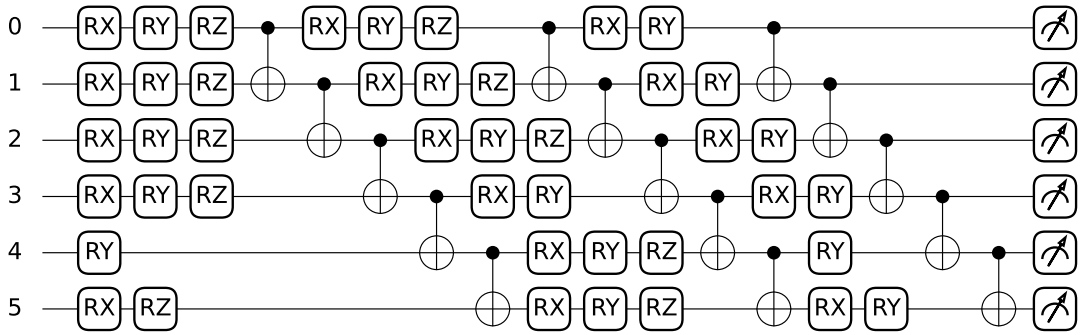

Figure 1: A typical 5-qubit PQC quantum circuit with three layers.

Why VQCs are an ideal benchmark: VQCs provide a solid test for simulation performance because they require the repeated application of dense matrix product operations across all qubits. Unlike simple algorithms that might only use a few gates, VQCs possess a deep and repetitive layered structure ("depth"). This stresses the simulator to compute complex state evolutions continuously. Such a structure allows us to easily scale the workload by simply increasing the number of qubits or the number of layers in the circuit. Thus, stress-testing the parallel efficiency of our OpenMP implementation becomes feasible and simple.

3 Methods

3.1 Theoretical Basis

The computational core of our simulation relies on linear algebra, particularly the Kronecker (tensor) product. Given an operator A of size $m \times n$ and an operator B of size $p \times q$, the product $C = A \otimes B$ results in a matrix of size $mp \times nq$. The element-wise expression is

$$C_{i,j} = A_{\lfloor i/p \rfloor, \lfloor j/q \rfloor} \times B_{i \pmod p, j \pmod q}$$

Conceptually, a quantum circuit is processed layer by layer. If we were to construct the full operator exactly, for a gate U applied to qubit k, the global unitary would be:

$$U_{global} = I^{\otimes k} \otimes U \otimes I^{\otimes (N-k-1)}$$

However, naively constructing this U_{global} (a $2^N \times 2^N$ matrix) is memory-intensive (usually results in segmentation faults). Therefore, we implement a State-Vector Update approach. It applies the mathematical equivalent of these tensor products directly to the state amplitudes without forming those large, sparse matrices.

3.2 Implementation Environment

The simulator was implemented in C++ (C++17) as it handles low-level memory management. Parallelization was achieved using the OpenMP API, and the complex number arithmetic was handled using the standard `std::complex<double>` library.

3.3 Simulation Flow

The simulation proceeds in the following steps:

1. **Initialization:** The state vector $|\Psi\rangle$ is initialized to the ground state $|0 \dots 0\rangle$ (a vector of size 2^N with the first element as 1 and all others 0).
2. **Layer Execution:** The circuit is processed sequentially. For each gate, we identify the target qubits and update the state vector in place.
3. **State Evolution:** The operations transform the complex amplitudes of the state vector (linear algebra based), effectively evolving the state of the system in Hilbert space.

3.4 Parallel Gate Application Algorithm

For a single-qubit gate acting on a target qubit t, we iterate through the state vector to identify pairs of indices (i_0, i_1) where the t-th bit is 0 and 1, respectively. We then update these amplitudes using the 2×2 gate matrix.

OpenMP Parallelization Strategy: We used the `#pragma omp parallel for collapse(2)` directive. This instructs the compiler to:

1. Spawn a team of threads.
2. Collapse the nested loops (iterating over the state vector blocks) into a single iteration space.
3. Distribute these iterations dynamically across the available CPU cores.

Listing 1: Parallel application of a single-qubit gate using OpenMP

```
void apply_single_qubit_gate(State &state, int n_qubits,
                             int target, const Mat2 &U) {
    const std::size_t dim = state.size();
    const std::size_t step = (1ULL << target);
    const std::size_t period = step * 2;

    // PARALLEL REGION: Automatically distributes loop iterations
    #pragma omp parallel for collapse(2)
    for (std::size_t base = 0; base < dim; base += period) {
        for (std::size_t i = 0; i < step; ++i) {
            std::size_t i0 = base + i;
            std::size_t i1 = base + i + step;

            Complex a0 = state[i0];
            Complex a1 = state[i1];

            // In-place update using the 2x2 Unitary U
            state[i0] = U.m[0][0] * a0 + U.m[0][1] * a1;
            state[i1] = U.m[1][0] * a0 + U.m[1][1] * a1;
        }
    }
}
```

3.5 Entangling Gates (CNOT)

The CNOT gate flips the target qubit if and only if the control qubit is $|1\rangle$. In our state-vector representation, this is done by swapping the amplitudes at indices where the control bit is 1 and the target bit differs. This operation is memory-bound rather than compute-bound. OpenMP is used here to parallelize the traversal of the state array.

Listing 2: Parallel implementation of the CNOT gate

```
void apply_cx(State &state, int n_qubits, int control, int target) {
    const std::size_t dim = state.size();
    std::size_t control_mask = (1ULL << control);
    std::size_t target_mask  = (1ULL << target);

    // PARALLEL REGION
    #pragma omp parallel for
    for (std::size_t idx = 0; idx < dim; ++idx) {
```

```
        // Only act if control bit is 1
        if ( (idx & control_mask) != 0 ) {
            std::size_t flipped_idx = idx ^ target_mask;

            // Swap only once per pair to avoid race conditions
            if (idx < flipped_idx) {
                std::swap(state[idx], state[flipped_idx]);
            }
        }
    }
}
```

4 Experiments and Results

For the evaluation of our parallel quantum circuit simulator, a series of experiments were conducted with the Variational Quantum Circuit (VQC) benchmark, described in Section II. The experiments were conducted with objective to measure the impact of OpenMP parallelization on the parameters: execution time (T_p), Speedup (S_p), Parallel Efficiency (E_p), and Computational Cost (C_p).

Simulation Settings: All experiments were conducted on a **Lambda Vector One** high-performance deep learning workstation, with the following specific hardware specifications:

- **Processor:** AMD Ryzen™ 9 7950X (Zen 4 architecture).
- **Core Count:** 16 physical cores, 32 logical threads (Simultaneous Multithreading).
- **Memory:** 64 GB DDR5 RAM.
- **Operating System:** Ubuntu Linux (via Lambda Stack).

This hardware configuration enabled us to test explicit scaling from 1 thread to the maximum available 32 logical threads.

4.1 Test Case 1: Scaling with Circuit Depth

For the first experiment, the system size was kept constant(20 qubits), and tested for the different number of layers in the VQC from 3 to 8. Thus, this tests the simulator's ability to handle deeper circuits where the state vector is continuously updated.

Figure 2 illustrates the execution time as a function of thread count. In the experiment we observed the significant decrease in runtime on the increase of number of threads. For a depth of 8 layers, the serial execution time was 12.11 seconds, but with 15 threads, this dropped to 1.72 seconds, showcasing a substantial performance gain. However, beyond 15 threads, the stagnation in performance was observed. This indicates that for 20 qubits, the workload per thread becomes low compared to the thread management overhead and memory bandwidth limitation, which restrict the further speedup.

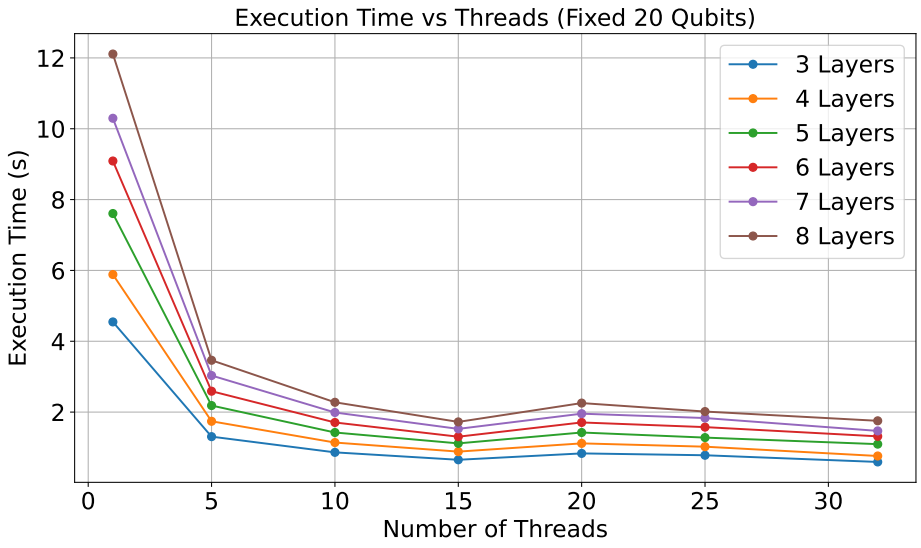

Figure 2: Execution Time vs. Number of Threads for fixed 20 qubits with varying circuit depths (3 to 8 layers).

4.2 Test Case 2: Scaling with System Size

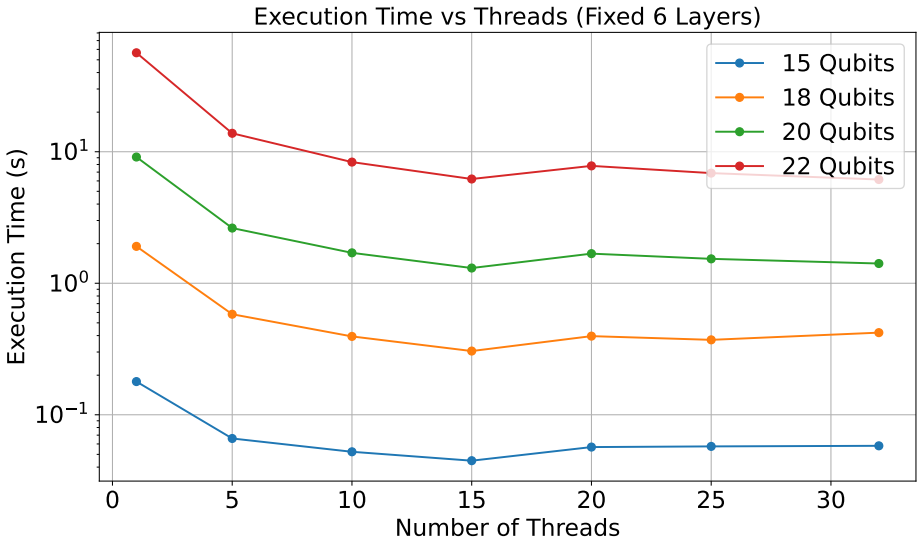

Figure 3: Execution Time vs. Threads for varying qubit counts (Log Scale). Note the exponential increase in serial time as qubits increase.

for the second experiment, we made the circuit depth constant (to 6 layers) and tested for the different number of qubits (from 5 to 22). This is the most critical test, as the state vector size grows exponentially (2^N).

Small System Overhead: For smaller systems (5 and 10 qubits), negligible or negative results (see data for 5 qubits) were observed with parallelization. The overhead of spawning

threads and managing parallel regions outweighed the computational cost of the trivial state vector updates.

Large System Acceleration: However for larger systems, the benefits of parallelization were evident. Figure 3 shows the execution time on a logarithmic scale.

- **22 Qubits:** The serial execution took **56.55 seconds**. Using 15 threads reduced this to **6.20 seconds**, a $\approx 9\times$ improvement.
- **20 Qubits:** Execution time dropped from 9.12 seconds (serial) to 1.30 seconds (15 threads).

4.3 Performance Metrics Analysis

Speedup (S_p):

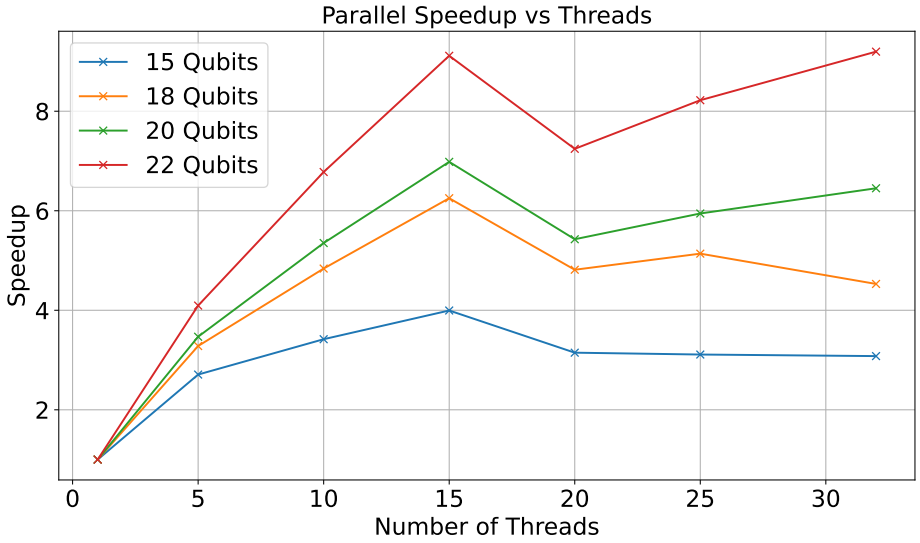

Figure 4: Speedup (S_p) vs. Threads. Larger qubit systems exhibit better speedup characteristics.

Speedup is defined as $S_p = T_1/T_p$. Figure 4 shows the speedup achieved for varying Qubit counts. The plot clearly demonstrates that larger workloads (22 Qubits) achieve better speedup because of the higher ratio of useful computation to parallel overhead. We observed a maximum speedup of approximately $9.1\times$ at 22 Qubits.

Parallel Efficiency (E_p):

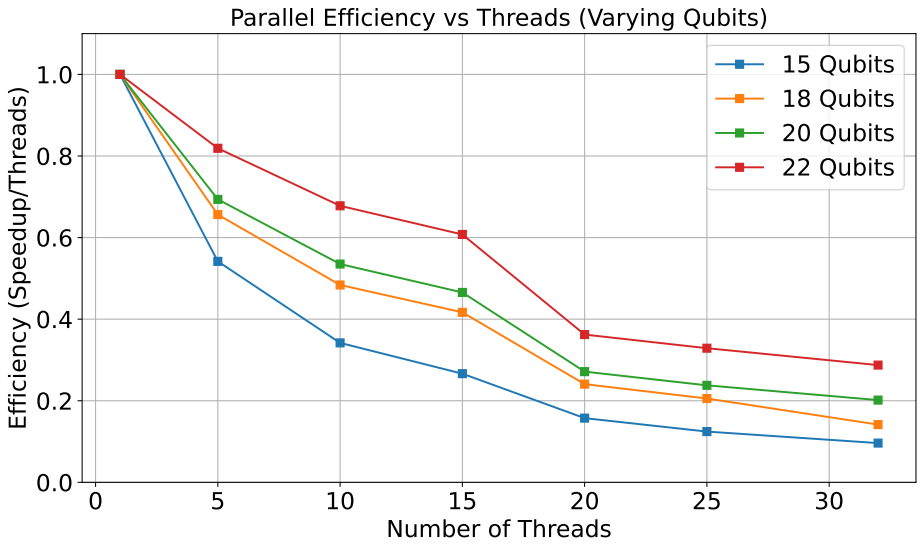

Figure 5: Parallel Efficiency vs. Threads. The drop in efficiency at higher thread counts indicates memory bandwidth saturation.

Efficiency ($E_p = S_p/P$), measures how effectively the utilized cores are being used. Figure 5 demonstrates, how efficiency drops as threads increases. The state-vector update method is highly memory-intensive (traversing large arrays). As we increase the number of threads, we eventually saturate the memory bandwidth of the hardware. This explains why the efficiency drops below 50% when the thread count exceeds 15, and why adding threads beyond this point (e.g., 32 threads) doesn't yield any significant results.

Computational Cost:

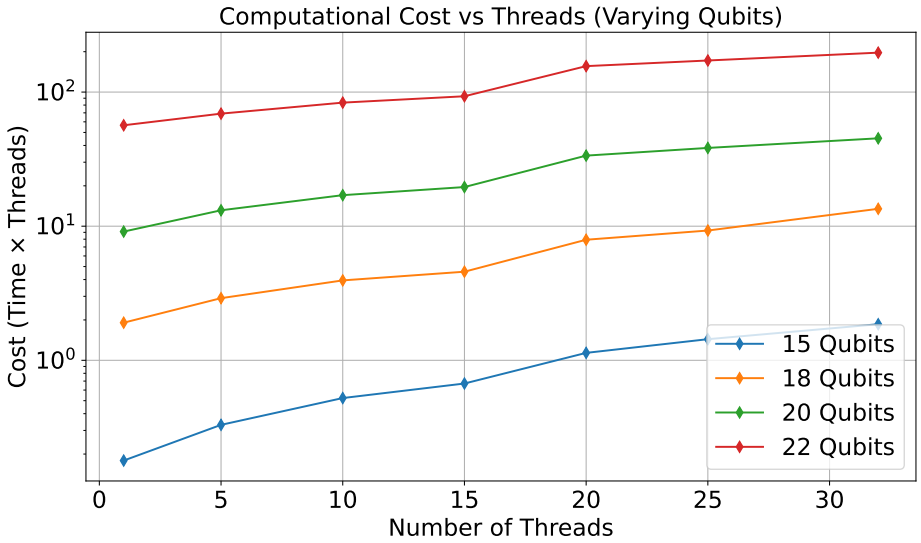

Figure 6: Computational Cost (Time × Threads) vs. Threads. The upward trend represents the overhead introduced by parallelization.

We also analyzed the Computational Cost (Time × Threads). Ideally, cost should remain constant. However, as shown in Figure 6, the cost increases with the number of threads. This increase reflects the "price" of parallelism, where the total CPU cycles grow due to parallel overhead, even though the wall-clock execution time decreases.

5 Conclusion

This project demonstrates the use of parallelization (OpenMP) as an effective way to accelerate quantum state vector simulation on a classical multi-core CPU. We simulated quantum computing in parallel on a classical computer because quantum hardware is not yet easily accessible. Furthermore, executing such simulations on a classical computer using a single core takes an extremely long time, since the state vector grows exponentially as the number of Qubits increases. Therefore, by running single-qubit and two-qubit gate operations in a C++ simulator across multiple CPU cores, the workload was distributed among the cores, which significantly reduced the execution time.

The experiments demonstrated an increase in speedup for larger quantum systems as the number of threads increased. However, the performance gains eventually stagnated after a certain threshold of threads due to parallel overhead and memory bandwidth limitations, which prevented further speedup for the same number of Qubits (state vectors). Overall, this work confirms that OpenMP-based parallel computing improves quantum simulations mainly for larger workloads and up to a certain number of threads, while overhead and limitation in memory bandwidth, limit the further gain in the performance.

References

[1] R. Courtland, "Google aims for quantum computing supremacy [news]," *IEEE Spectrum*, vol. 54, no. 6, pp. 9–10, 2017.

[2] C. P. Koch, U. Boscain, T. Calarco, G. Dirr, S. Filipp, S. J. Glaser, R. Kosloff, S. Montangero, T. Schulte-Herbrüggen, D. Sugny *et al.*, "Quantum optimal control in quantum technologies. strategic report on current status, visions and goals for research in europe," *EPJ Quantum Technology*, vol. 9, no. 1, p. 19, 2022.

[3] S. Resch and U. R. Karpuzcu, "Benchmarking quantum computers and the impact of quantum noise," *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–35, 2021.

[4] S. Endo, J. Sun, Y. Li, S. C. Benjamin, and X. Yuan, "Variational quantum simulation of general processes," *Physical Review Letters*, vol. 125, no. 1, p. 010501, 2020.

[5] H. P. Stapp, "The copenhagen interpretation," *American journal of physics*, vol. 40, no. 8, pp. 1098–1116, 1972.

[6] J. Faj, I. Peng, J. Wahlgren, and S. Markidis, "Quantum computer simulations at warp speed: Assessing the impact of gpu acceleration: A case study with ibm qiskit aer, nvidia thrust & cuquantum," in *2023 IEEE 19th International Conference on e-Science (e-Science)*. IEEE, 2023, pp. 1–10.

[7] V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, S. Ahmed, V. Ajith, M. S. Alam, G. Alonso-Linaje, B. AkashNarayanan, A. Asadi *et al.*, "Pennylane: Automatic differentiation of hybrid quantum-classical computations," *arXiv preprint arXiv:1811.04968*, 2018.

[8] H. Wang, Z. Liang, J. Gu, Z. Li, Y. Ding, W. Jiang, Y. Shi, D. Z. Pan, F. T. Chong, and S. Han, "Torchquantum case study for robust quantum circuits," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[9] S. V. Isakov, D. Kafri, O. Martin, C. V. Heidweiller, W. Mruczkiewicz, M. P. Harrigan, N. C. Rubin, R. Thomson, M. Broughton, K. Kissell *et al.*, "Simulations of quantum circuits with approximate noise using qsim and cirq," *arXiv preprint arXiv:2111.02396*, 2021.

[10] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[11] D. Gottesman, "The heisenberg representation of quantum computers," *arXiv preprint quant-ph/9807006*, 1998.

[12] A. Pokharel, R. Rahman, T. Morris, and D. C. Nguyen, "Quantum federated learning for multimodal data: A modality-agnostic approach," in *Proceedings of the Computer Vision and Pattern Recognition Conference*, 2025, pp. 545–554.

A Appendix

The source code is presented in this section. There are two C++ files: one defines the quantum circuit and associated utilities, and the other runs the benchmark. The .h header file is also provided.

The latter part consists of the bash script to run the whole project and its output file.

quantum_circuit.cpp

```cpp
#include "quantum_circuit.h"
#include <iostream>
#include <cmath>
#include <iomanip>
#include <omp.h>

//Gate Definitions
Mat2 make_I() {
    return Mat2{ { { Complex(1,0), Complex(0,0) },
                   { Complex(0,0), Complex(1,0) } } };
}

Mat2 make_X() {
```

```
    return Mat2{ { { Complex(0,0), Complex(1,0) },
                   { Complex(1,0), Complex(0,0) } } };
}

Mat2 make_Y() {
    return Mat2{ { { Complex(0,0), Complex(0,-1) },
                   { Complex(0,1), Complex(0, 0) } } };
}

Mat2 make_Z() {
    return Mat2{ { { Complex(1,0), Complex(0,0) },
                   { Complex(0,0), Complex(-1,0) } } };
}

Mat2 make_H() {
    const double inv_sqrt2 = 1.0 / std::sqrt(2.0);
    return Mat2{ { { Complex(inv_sqrt2,0),  Complex(inv_sqrt2,0) },
                   { Complex(inv_sqrt2,0),  Complex(-inv_sqrt2,0) } } };
}
//Rotational Gates
Mat2 make_RX(double theta) {
    double c = std::cos(theta / 2.0);
    double s = std::sin(theta / 2.0);
    return Mat2{ { { Complex(c, 0),   Complex(0, -s) },
                   { Complex(0, -s),  Complex(c, 0)  } } };
}

Mat2 make_RY(double theta) {
    double c = std::cos(theta / 2.0);
    double s = std::sin(theta / 2.0);
    return Mat2{ { { Complex(c, 0), Complex(-s, 0) },
                   { Complex(s, 0), Complex(c, 0)  } } };
}

Mat2 make_RZ(double theta) {
    double c = std::cos(theta / 2.0);
    double s = std::sin(theta / 2.0);
    // e^{-i theta/2} = cos - i sin
    // e^{+i theta/2} = cos + i sin
    return Mat2{ { { Complex(c, -s), Complex(0, 0) },
                   { Complex(0, 0),  Complex(c, s) } } };
}


//Operation Functions
State make_zero_state(int n_qubits) {
    std::size_t dim = (1ULL << n_qubits);
    State psi(dim);

```

```
    // Parallel initialization for large arrays
    #pragma omp parallel for
    for(size_t i=0; i<dim; ++i) {
        psi[i] = Complex(0,0);
    }

    psi[0] = Complex(1.0, 0.0);
    return psi;
}

void apply_single_qubit_gate(State &state, int n_qubits, int target, const Mat2 &U)
↪ {
    const std::size_t dim = state.size();
    // validation check
    if (dim != (1ULL << n_qubits)) return;

    const std::size_t step = (1ULL << target);
    const std::size_t period = step * 2;

    // PARALLEL REGION:
    #pragma omp parallel for collapse(2)
    for (std::size_t base = 0; base < dim; base += period) {
        for (std::size_t i = 0; i < step; ++i) {
            std::size_t i0 = base + i;
            std::size_t i1 = base + i + step;

            Complex a0 = state[i0];
            Complex a1 = state[i1];

            // Matrix-Vector multiplication for single qubit gate
            Complex b0 = U.m[0][0] * a0 + U.m[0][1] * a1;
            Complex b1 = U.m[1][0] * a0 + U.m[1][1] * a1;

            state[i0] = b0;
            state[i1] = b1;
        }
    }
}

void apply_cx(State &state, int n_qubits, int control, int target) {
    const std::size_t dim = state.size();

    if (control == target) return;

    std::size_t control_mask = (1ULL << control);
    std::size_t target_mask  = (1ULL << target);

    // PARALLEL REGION:
    #pragma omp parallel for
```

```
    for (std::size_t idx = 0; idx < dim; ++idx) {
        if ( (idx & control_mask) != 0 ) {
            std::size_t flipped_idx = idx ^ target_mask;
            // Ensure we swap only once per pair
            if (idx < flipped_idx) {
                std::swap(state[idx], state[flipped_idx]);
            }
        }
    }
}

void print_probabilities(const State &state) {
    const std::size_t dim = state.size();
    if (dim > 32) {
        std::cout << " (State vector too large to print: size=" << dim << ")\n";
        return;
    }

    std::cout << std::fixed << std::setprecision(4);
    for (std::size_t i = 0; i < dim; ++i) {
        double p = std::norm(state[i]);
        if (p > 1e-10) {
            std::cout << "|" << i << "> : " << p << "\n";
        }
    }
}
```

quantum_circuit.h

```
    #ifndef QUANTUM_CIRCUIT_H
#define QUANTUM_CIRCUIT_H

#include <complex>
#include <vector>

//type def
using Complex = std::complex<double>;
using State   = std::vector<Complex>;

struct Mat2 {
    Complex m[2][2];
};

//Gate Functions
Mat2 make_I();
Mat2 make_X();
Mat2 make_Y();
Mat2 make_Z();
Mat2 make_H();
```

```
Mat2 make_RX(double theta);
Mat2 make_RY(double theta);
Mat2 make_RZ(double theta);

//Operation Functions
// Initialize state |0...0> (zero state)
State make_zero_state(int n_qubits);

//single-qubit gate U to target qubit (single qubit gate)
void apply_single_qubit_gate(State &state, int n_qubits, int target, const Mat2 &U);

// CX (CNOT)
void apply_cx(State &state, int n_qubits, int control, int target);

//print probabilities
void print_probabilities(const State &state);

#endif


```

simple_quantum_circuit.cpp

```
        #include "quantum_circuit.h"
#include <iostream>
#include <omp.h>
#include <vector>
#include <random>

//VQC Benchmark
void run_benchmark(int n_qubits, int layers, int training_rounds) {
    std::cout << "\nVQC Benchmark (" << n_qubits << " qubits)\n";
    std::cout << "Threads: " << omp_get_max_threads() << "\n";
    std::cout << "Simulating " << training_rounds << " training rounds (forward
    ↪ passes only.)\n";

    double total_start_time = omp_get_wtime();
    std::mt19937 gen(42); //random number generator
    std::uniform_real_distribution<> dis(0.0, 2.0 * M_PI);//uniform distribution for
    ↪ VQC parameters

    for (int round = 0; round < training_rounds; ++round) {
        double round_start = omp_get_wtime();
        //reset state for each forward pass
        State psi = make_zero_state(n_qubits);
        for (int layer = 0; layer < layers; ++layer) { //for each layer
            // Parametrized gate to all: RX
            for(int i = 0; i < n_qubits; ++i) {
                double theta = dis(gen);
```

```
                apply_single_qubit_gate(psi, n_qubits, i, make_RX(theta));
            }
            //Parametrized gate RY
            for(int i = 0; i < n_qubits; ++i) {
                double theta = dis(gen);
                apply_single_qubit_gate(psi, n_qubits, i, make_RY(theta));
            }
            //RZ
            for(int i = 0; i < n_qubits; ++i) {
                double theta = dis(gen);
                apply_single_qubit_gate(psi, n_qubits, i, make_RZ(theta));
            }
            //All to all entanglement
            for(int i = 0; i < n_qubits; ++i) {
                for(int j = 0; j < n_qubits; ++j) {
                    if (i != j) {
                        apply_cx(psi, n_qubits, i, j);
                    }
                }
            }
        }
        double round_end = omp_get_wtime();
        std::cout << "Round " << (round + 1) << "/" << training_rounds
                  << " complete. Time: " << (round_end - round_start) << " s\n";
    }

    double total_end_time = omp_get_wtime();
    std::cout << "\nRun Complete.\n";
    std::cout << "Total Time: " << (total_end_time - total_start_time) << "
    ↪  seconds.\n";
    std::cout << "Average Time per Round: " << (total_end_time - total_start_time) /
    ↪  training_rounds << " seconds.\n";
}


//main
int main(int argc, char* argv[]) {
    //default cases
    int n_qubits = 20;
    int layers = 6;
    int rounds = 10;

    if (argc > 1) n_qubits = std::atoi(argv[1]);
    if (argc > 2) layers   = std::atoi(argv[2]);
    if (argc > 3) rounds   = std::atoi(argv[3]);

    run_benchmark(n_qubits, layers, rounds);

    return 0;
```

```
}

```

Bash Script:

```
    #!/bin/bash
#compile first
echo "Compiling..."
g++ -O3 -fopenmp simple_quantum_circuit.cpp quantum_circuit.cpp -o
↪  simple_quantum_circuit
OUTPUT_FILE="benchmark_results.txt" #output file
#clear ir first if file exists
echo "Results:" > $OUTPUT_FILE

THREADS_LIST=(1 5 10 15 20 25 32)
ROUNDS=5 # Reduced for speed in automated runs, increase if needed

#test case 1
QUBITS_FIXED=20
LAYERS_LIST=(3 4 5 6 7 8)

echo "Test Case 1: Varying Layers (Fixed 20 Qubits)"
echo "" >> $OUTPUT_FILE
echo "TEST CASE 1: Fixed Qubits = $QUBITS_FIXED, Varying Layers" >> $OUTPUT_FILE
printf "%-10s %-10s %-10s %-20s\n" "Threads" "Qubits" "Layers" "Total Time (s)" >>
↪  $OUTPUT_FILE #header

for LAYERS in "${LAYERS_LIST[@]}"; do #for each layer
    for THREADS in "${THREADS_LIST[@]}"; do
        echo "Running: Qubits=$QUBITS_FIXED, Layers=$LAYERS, Threads=$THREADS"
        ↪  #running

        RESULT=$(OMP_NUM_THREADS=$THREADS ./simple_quantum_circuit $QUBITS_FIXED
        ↪  $LAYERS $ROUNDS) #run benchmark
        #extract total time
        TIME=$(echo "$RESULT" | grep "Total Time:" | awk '{print $3}')

        printf "%-10s %-10s %-10s %-20s\n" "$THREADS" "$QUBITS_FIXED" "$LAYERS"
        ↪  "$TIME" >> $OUTPUT_FILE
    done
    echo "" >> $OUTPUT_FILE # Spacer between layer groups
done

#Test Case 2: Varying Qubits (Fixed 6 Layers)
LAYERS_FIXED=6
QUBITS_LIST=(5 10 15 18 20 22)

echo "Test Case 2: Varying Qubits (Fixed 6 Layers)"
echo "" >> $OUTPUT_FILE
```

```
echo "TEST CASE 2: Fixed Layers = $LAYERS_FIXED, Varying Qubits" >> $OUTPUT_FILE
printf "%-10s %-10s %-10s %-20s\n" "Threads" "Qubits" "Layers" "Total Time (s)" >>
↪  $OUTPUT_FILE #header

for QUBITS in "${QUBITS_LIST[@]}"; do #for each qubit
    for THREADS in "${THREADS_LIST[@]}"; do
        echo "Running: Qubits=$QUBITS, Layers=$LAYERS_FIXED, Threads=$THREADS"

        #running
        RESULT=$(OMP_NUM_THREADS=$THREADS ./simple_quantum_circuit $QUBITS
        ↪  $LAYERS_FIXED $ROUNDS)

        #extract total time
        TIME=$(echo "$RESULT" | grep "Total Time:" | awk '{print $3}')

        printf "%-10s %-10s %-10s %-20s\n" "$THREADS" "$QUBITS" "$LAYERS_FIXED"
        ↪  "$TIME" >> $OUTPUT_FILE
    done
    echo "" >> $OUTPUT_FILE
done

echo "Done. $OUTPUT_FILE"
```

Output (Run 1:)

```
        Results:

TEST CASE 1: Fixed Qubits = 20, Varying Layers
Threads    Qubits     Layers     Total Time (s)
1          20         3          4.54596
5          20         3          1.30675
10         20         3          0.863134
15         20         3          0.653564
20         20         3          0.833679
25         20         3          0.781296
32         20         3          0.592721

1          20         4          5.88434
5          20         4          1.74037
10         20         4          1.14022
15         20         4          0.884248
20         20         4          1.11676
25         20         4          1.02337
32         20         4          0.760781

1          20         5          7.60661
5          20         5          2.18475
10         20         5          1.42377
15         20         5          1.11657
```

```
20          20          5           1.4235
25          20          5           1.28007
32          20          5           1.09518

1           20          6           9.08939
5           20          6           2.58956
10          20          6           1.7066
15          20          6           1.30513
20          20          6           1.70739
25          20          6           1.57605
32          20          6           1.31584

1           20          7           10.2956
5           20          7           3.03262
10          20          7           1.98872
15          20          7           1.52616
20          20          7           1.95523
25          20          7           1.8321
32          20          7           1.46896

1           20          8           12.1098
5           20          8           3.46344
10          20          8           2.27673
15          20          8           1.72295
20          20          8           2.25464
25          20          8           2.01544
32          20          8           1.75398


TEST CASE 2: Fixed Layers = 6, Varying Qubits
Threads     Qubits      Layers      Total Time (s)
1           5           6           0.000264091
5           5           6           0.00153651
10          5           6           0.00201087
15          5           6           0.00245149
20          5           6           0.00328084
25          5           6           0.00374911
32          5           6           0.00427548

1           10          6           0.00388986
5           10          6           0.00698495
10          10          6           0.00770696
15          10          6           0.00843069
20          10          6           0.010934
25          10          6           0.0122837
32          10          6           0.014421

1           15          6           0.178828
5           15          6           0.0660308
```

```
10          15          6           0.052311
15          15          6           0.0447584
20          15          6           0.0568088
25          15          6           0.0574869
32          15          6           0.0580764

1           18          6           1.90769
5           18          6           0.581317
10          18          6           0.394283
15          18          6           0.305144
20          18          6           0.396245
25          18          6           0.371291
32          18          6           0.421195

1           20          6           9.1156
5           20          6           2.6281
10          20          6           1.7034
15          20          6           1.30543
20          20          6           1.67951
25          20          6           1.53332
32          20          6           1.41287

1           22          6           56.5532
5           22          6           13.8116
10          22          6           8.34245
15          22          6           6.20477
20          22          6           7.8064
25          22          6           6.8791
32          22          6           6.14886


```