Grand Valley State University

ScholarWorks@GVSU

Masters Theses | Graduate Research and Creative Practice

4-29-2021

Quantum Simulation Using High-Performance Computing

Collin Beaudoin
Grand Valley State University

Christian Trefftz
Grand Valley State University

Zachary Kurmas
Grand Valley State University

Follow this and additional works at: https://scholarworks.gvsu.edu/theses

Part of the Theory and Algorithms Commons

ScholarWorks Citation

Beaudoin, Collin; Trefftz, Christian; and Kurmas, Zachary, "Quantum Simulation Using High-Performance Computing" (2021). *Masters Theses*. 1011.
https://scholarworks.gvsu.edu/theses/1011

This Thesis is brought to you for free and open access by the Graduate Research and Creative Practice at ScholarWorks@GVSU. It has been accepted for inclusion in Masters Theses by an authorized administrator of ScholarWorks@GVSU. For more information, please contact scholarworks@gvsu.edu.

Quantum Simulation Using High-Performance Computing: Hermitian Observable Multiplication Algorithm

Collin Beaudoin[1], Zachary Kurmas[1] and Christian Trefftz[1]

[1] Grand Valley State University, Allendale MI 49401, USA
beaudoco@mail.gvsu.edu

Abstract. Hermitian matrix multiplication is one of the most common actions that is performed on quantum matrices, for example, it is used to apply observables onto a given state vector/density matrix.

$$\rho \rightarrow H\rho \tag{1}$$

Our goal is to create an algorithm to perform the matrix multiplication within the constraints of QuEST [1], a high-performance simulator for quantum circuits. QuEST provides a system-independent platform for implementing and simulating quantum algorithms without the need for access to quantum machines. The current implementation of QuEST supports CUDA, MPI, and OpenMP, which allows programs to run on a wide variety of systems.

Keywords: Quantum Computing, CUDA, MPI, OpenMP, HPC

1 Introduction

Quantum computing offers the ability to perform in seconds computations that would take classical computers thousands of years to complete [5]. As quantum computing becomes more available/larger in scale it will pose larger and larger threats to current security, allow scientists to answer questions that were previously unsolvable and will force a total paradigm shift in the world of computation. Quantum computer simulations offer users the opportunity to create algorithms, test theories and even obtain quantum algorithmic advantages. In the current Noisy Intermediate-Scale Quantum (NISQ) climate, it is important to have ways to verify results, prove theories in an inexpensive manner, and prepare for future security threats. One essential, fundamental component of quantum algorithms is being able to multiply observables onto quantum registers (an example of this would be the Pauli matrices). The knowledge of this necessity leads to the creation of an algorithm to simulate the quantum computation of a quantum register being multiplied by a dense Hermitian matrix. The result of this multiplication then replaces the current quantum register. We implement this basic quantum computation for the QuEST framework in OpenMP, MPI and CUDA. We then

compare the individual programs to demonstrate the benefit of each implementation, following with potential future improvements.

2 Related Work

Quantum simulation is the ideal candidate for HPC programming. It is easy to see the necessity of parallel programs to perform these simulations as even a relatively small 8 qubit array density matrix requires a matrix of 65,536 (2^8 x 2^8) double complex values to represent. Due to the exponentially growing size requirements of classical bits and the price of quantum computers, there are quite a few HPC quantum simulators that exist, including: Q#, QCGPU, and qHipster. Each of these simulators have limitations. QHipster from Intel [3] enables the use of arrays of machines, making it ideal for "larger" amounts of qubits; but with little to no performance tuning for smaller qubit problems. Q#, from Microsoft [2], enables the use of just a single machine, which means it cannot handle anything larger than a few qubits for computation. QCGPU is strictly for GPUs [4], which has the same smaller scale issue as Q# and requires specific hardware, further limiting its usage. The current state of quantum simulators forces the developer to potentially implement three different versions of their program depending on their intention/hardware availability. This severely limits scaling and forces the developer into a single ecosystem, that may or may not fit their needs. QuEST removes this issue by accounting for all three major HPC types within the simulator, allowing for developers to create a single program that only requires recompilation for scaling. This means any quantum algorithm creation will work machine agnostically.

3 Foundations

3.1 HPC:

HPC is an integral part of modern large-scale computations. It enables researchers to resolve large scale problems that are too computationally heavy to perform on a single threaded machine. It enables the use of multiple threads, the use of GPUs and the use of multiple machines to resolve singular problems. It also allows for developers/researchers to select different setups for their interests/problem sizes.

3.2 Quantum Computing:

Quantum computing is proven to be more performant in comparison to classical computers [5][6], both in the near term and long term of computation. It enables parallel computation, with a logarithmic scale of qubits to bits, by exploiting quantum mechanical properties such as superposition. Unfortunately, the state-of-the-art quantum computers are extremely expensive and do not offer many more usable qubits than what can reasonably be simulated using large arrays of machines.

3.3 Quantum Simulators:

Quantum simulators allow for both the simulation of state vectors and density matrices, enabling researchers to perform studies that would exemplify current NISQ computers. The biggest issue with current simulators is the lack of ability to replicate absolute randomness, because of this there can be errors in computation. However, simulators enable approximations and proof of concepts that would otherwise be unrealizable.

4 Approach

Our initial focus is to create an algorithm that will work given that ρ is a state vector. This is to remove any extra complexity that may occur with multi-dimensional matrix multiplication. Our first step is to design an algorithm that allows for the simulation of the quantum state vectors being multiplied by dense Hermitian matrices [10]. The result of this multiplication then replaces the current state vector. We approached the problem by obtaining the state vector, normalizing the values, and then obtaining the Hermitian matrix which we will use to perform our computations.

4.1 MPI:

MPI allows for the use of multiple machines to work on a single problem [7]. We begin the MPI implementation by creating a distributed solution to the linear algebra problem of matrix vector multiplication.

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_{11}b_1 + \cdots + a_{1n}b_n \\ \vdots \\ a_{n1}b_1 + \cdots + a_{nn}b_n \end{bmatrix}$$

Figure 1: Matrix Vector Multiplication Model

QuEST's memory management requires that both the Hermitian matrix and the state vector be distributed evenly over n-nodes (given the constraint that n is a power-of-2). Fortunately, the Hermitian matrix memory requirements have no definition beyond this, allowing for tuning to obtain the best algorithmic speed-up.

The bottleneck of MPI tends to be communication. This means the most time efficient approach is to use the least possible amount of communication between nodes.

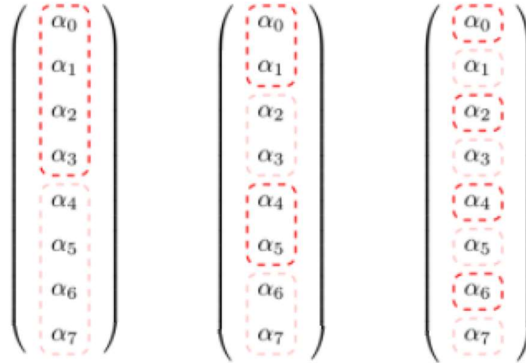

Figure 2: Three Qubit State Vector Memory Distribution Examples

Given that the state vector and Hermitian matrix are both to be evenly distributed, and that the size of the state vector would be Q times smaller than the Hermitian matrix (where Q is the size of the state vector), the obvious item to communicate is the state vector. To use the least amount of communication, the Hermitian matrix is split so the row of the state-vector's calculated entry will always be on the same machine. Ultimately, the memory distribution results in a column storage distributed in the same way as the state vector.

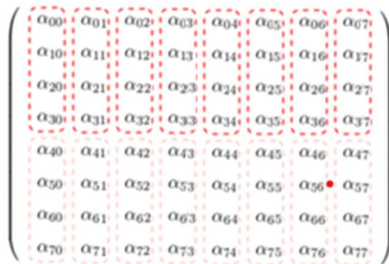

Figure 3: Three Qubit Two Node Dense Matrix Distribution

Following this memory distribution, it is possible to use MPI's send-receive to distribute the state vector in its entirety to each individual node. This allows for each entry of the state vector to be calculated with no calculation redundancy and no further reliance on other node communication.

4.2 OpenMP:

OpenMP relies on all memory existing on the machine that is performing the calculations [8]. This removes any need for communication. Out of simplicity, the distribution of the Hermitian is once again split into a column structure, as there is no need for special memory distribution. Following the implementation of the column memory structure, we implemented a simple loop that iterates through the Hermitian matrices for calculation of the state vector's individual entries. We implemented a critical section on the write to ensure that only a single write is occurring to the entry at a time.

4.3 CUDA:

CUDA relies on NVIDA GPUs for accelerated calculations [9]. To perform the Hermitian multiplication, there must be a communication between the CPU memory and the GPU memory. To stay consistent with OpenMP and MPI, the Hermitian is split into the aforementioned column structure. We concatenated these columns to feed them to the GPU. This is to remove any device dependent optimization or memory issues that CUDA's 2D operations would require addressing (such as pitch size). Following this, we implemented a kernel program that relies on a singular for loop. This allows the GPU to handle the matrix multiplication in O(N) time per process.

5 Performance

To better understand the overall performance there are a few items worth discussing. The general item to test is the average speed performance of the algorithms and comparing this to the sequential times of the program. However, with MPI there are three other important measurements that help explain how performance may scale: total throughput (TT), serial rounds of communication (SRC), and required buffer size (BS).

5.1 Sequential vs. HPC:

QuEST suggests that a 2GB GPU can run the equivalent of a 26-qubit simulation. To ensure there are no anomalies that occur from attempting maximum threshold computations, we used a 14-qubit simulation for the sequential, MPI, OpenMP and CUDA versions of the program for testing. This is to better understand the general use case of the program. For proper testing, each version of the program is run 5 times, it is also important to note that MPI is given 4 machines to run on while OpenMP is run on 2 threads. The resulting average runtime is shown in Figure 4.

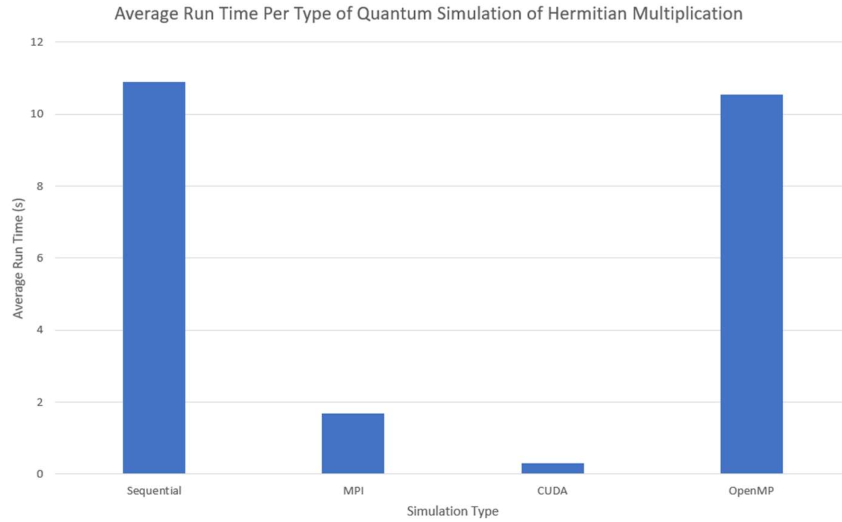

Figure 4: Average Run Time Per Quantum Simulation of 14 Qubit Hermitian Multiplication

Table 1. Average Run Time Per Quantum Simulation of 14 Qubit Hermitian Multiplication

Simulation Type	Sequential	MPI	CUDA	OpenMP
Average Runtime (s)	10.8914162	1.6871054	0.293759	10.5397898

To get a better idea of the potential realizable speedup from exploiting quantum's parallel properties, the speedup of each runtime is calculated.

$$Speedup = Time_{Sequential}/Time_{parallel} \quad (2)$$

Table 2. Average Run Time Per Quantum Simulation of 14 Qubit Hermitian Multiplication

Simulation Type	Sequential	MPI	CUDA	OpenMP
Average Speedup	1	6.455682141	37.07602559	1.033361804

5.2 MPI:

When working with MPI it is important to know how much communication is necessary for an algorithm's implementation, as this tends to be the bottleneck with larger scale problems. Starting with TT, the throughput for the algorithm is:

Total Throughput
Starting with TT, the throughput for the algorithm would be:

$$TT = (n^2 - n)A \tag{3}$$

Where:

$$A = \frac{Q}{n} \tag{4}$$

Which simplifies to:

$$TT = (n - 1)Q \tag{5}$$

Serial Rounds of Communication
Next is the SRC, this calculation is simple and accounts for the number of rounds of communication necessary for the entire process. This is a 1 round SRC, as the communication only runs once from a machine to another.

Buffer Size
Finally, the BS. To account for the calculation of the individual entries the buffer needs to be large enough to hold the partition's size for calculations. This additional buffer space must be added to each machine to accept:

$$BS = A \tag{6}$$

However, there is an additional buffer size that must be added to each machine, this is to accept the communication of the rest of the state vector to properly perform the calculation of the Hermitian multiplication. This additional memory is:

$$BS = A + (Q - A) \rightarrow BS = Q \tag{7}$$

6 Conclusions and Future Work

Quantum computation offers many benefits in terms of calculation speedups, but to exploit these benefits researchers need to be able to work through and prove algorithms. In the current status this is best performed with HPC quantum simulations. These

simulators need to enable researchers to focus on algorithms rather than hinder them with platform limitations and new syntax. This Hermitian multiplication algorithm allows researchers to use necessary calculations at quicker than sequential rates regardless of their platform limitations. To further improve MPI performance it would be plausible to add a secondary buffer to the QuEST storage structure. This would remove any need for cross communication, and it can be assumed that the machine can handle this memory addition as the machine is holding at least the same number of values for the Hermitian matrix.

References

1. Jones, T.: QuEST and High Performance Simulation of Quantum Computers. Scientific Reports (2019).
2. Q# About Page, https://docs.microsoft.com/en-us/azure/quantum/overview-what-is-qsharp-and-qdk, last accessed 2021/03/11.
3. Smelyanskiy, M.: The Quantum High Performance Software Testing Environment (2016).
4. Kelly, A.: Simulating Quantum Computers Using OpenCL (2018).
5. Arute, F.: Quantum supremacy using a programmable superconducting processor. Nature (2019).
6. Shor, P.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. (1995).
7. MPI Tutorial, https://computing.llnl.gov/tutorials/mpi/, last accessed 2021/03/11
8. OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/last accessed 2021/03/11
9. CUDA About Page, https://developer.nvidia.com/about-cuda, last accessed 2021/03/11
10. Beaudoin, C: Quantum Project, https://github.com/beaudoco/Quantum-Project, last accessed 2021/03/25