NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A PARALLEL QUANTUM COMPUTER SIMULATOR

by

James E. Fischer

September 2016

Thesis Advisor:	Ted Huffmire
Second Reader:	James Luscombe

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704–0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 2016	3. REPORT TYPE AND DATES COVERED Master's thesis

4. TITLE AND SUBTITLE A PARALLEL QUANTUM COMPUTER SIMULATOR	5. FUNDING NUMBERS
6. AUTHOR(S) James E. Fischer	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

The unique principles of quantum mechanics may one day enable computers to perform operations that would be impossible on a classical computer. Although no one knows whether it will be possible to build a large-scale, functional, and stable quantum computer, researchers can study quantum-mechanical systems and develop algorithms and circuits by simulating quantum systems in software. Performance and memory bottlenecks prevent most current quantum computer simulators from being able to simulate quantum systems that are large enough to be useful. In this thesis, we develop a matrix-free sequential quantum computer simulator to vastly improve both time and memory performance of sequential code on a single processor. Next, we distribute the matrix-free algorithm over multiple parallel processors using the Message Passing Interface in order to simulate quantum systems that are too large to reside wholly within the memory of a single processor. Finally, we simulate various quantum circuits using the Hamming high-performance computing cluster in order to conduct algorithmic analysis.

14. SUBJECT TERMS quantum computing, quantum computer simulation, parallel computing	15. NUMBER OF PAGES 129
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540–01-280-5500 Standard Form 298 (Rev. 2–89) Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

A PARALLEL QUANTUM COMPUTER SIMULATOR

James E. Fischer
Lieutenant, United States Navy
B.S., United States Naval Academy, 2008

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2016

Approved by: Ted Huffmire
Thesis Advisor

James Luscombe
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The unique principles of quantum mechanics may one day enable computers to perform operations that would be impossible on a classical computer. Although no one knows whether it will be possible to build a large-scale, functional, and stable quantum computer, researchers can study quantum-mechanical systems and develop algorithms and circuits by simulating quantum systems in software. Performance and memory bottlenecks prevent most current quantum computer simulators from being able to simulate quantum systems that are large enough to be useful. In this thesis, we develop a matrix-free sequential quantum computer simulator to vastly improve both time and memory performance of sequential code on a single processor. Next, we distribute the matrix-free algorithm over multiple parallel processors using the Message Passing Interface in order to simulate quantum systems that are too large to reside wholly within the memory of a single processor. Finally, we simulate various quantum circuits using the Hamming high-performance computing cluster in order to conduct algorithmic analysis.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION TO QUANTUM COMPUTING 1
- A. BACKGROUND AND QUANTUM COMPUTING FUNDAMENTALS 1
  - 1. Overview and Significance 1
  - 2. Superposition, Quantum Parallelism, and Bra-ket Notation 2
  - 3. Multiple Qubit Systems and Entanglement 4
- B. INTRODUCTION TO QUANTUM GATES, CIRCUITS, AND ALGORITHMS 7
  - 1. Quantum Gates and Circuits 8
  - 2. Quantum Algorithms 14
- C. DISSIPATION, DECOHERENCE, AND QUANTUM ERROR CORRECTION 24
  - 1. Classical Error Correction 26
  - 2. Quantum Error Correction 28
  - 3. Quantum Error Correction Codes 30
- D. QUANTUM COMPUTER SIMULATION 36
- E. LIMITATIONS OF SEQUENTIAL QUANTUM COMPUTER SIMULATORS 37

II. INTRODUCTION TO PARALLEL COMPUTING 39
- A. OVERVIEW OF PARALLELISM IN COMPUTING 39
  - 1. Bit Level Parallelism 40
  - 2. Instruction Level Parallelism 40
  - 3. Thread Level Parallelism across Multiple Processors 42
- B. PARALLEL PROGRAMMING, THE MESSAGE PASSING INTERFACE, AND THE JULIA LANGUAGE 43
  - 1. Parallel Programming with MPI 43
  - 2. Julia Programming Language 45
- C. PARALLEL ALGORITHM PERFORMANCE ANALYSIS: AMDAHL'S LAW AND GUSTAFSON'S LAW 46
- D. APPLICATION TO QUANTUM COMPUTER SIMULATION 49

III. DEVELOPMENT OF A PARALLEL QUANTUM COMPUTER SIMULATOR 51
- A. OPTIMIZING A SEQUENTIAL QCS 51
  - 1. Quantum Description Language 51
  - 2. Simulating Quantum Gates In-Place 53

3. Comparison of Full-Matrix and In-Place Algorithms 64
B. GOALS OF PARALLELIZING A QCS 68
C. IMPLEMENTING THE IN-PLACE ALGORITHM USING MPI 69

IV. PERFORMANCE ANALYSIS AND OPTIMIZATION OF THE PARALLEL QUANTUM COMPUTER SIMULATOR 75
A. ASCERTAINMENT OF CORRECTNESS OF THE PARALLEL QUANTUM COMPUTER SIMULATOR 75
B. PERFORMANCE ANALYSIS OF THE PARALLEL QUANTUM COMPUTER SIMULATOR 79
C. COMMUNICATION COSTS IN THE PARALLEL QUANTUM COMPUTER SIMULATOR 86
D. OPTIMIZING THE PARALLEL QUANTUM COMPUTER SIMULATOR 92
1. Optimizations Included in our Simulator 92
2. Potential Future Optimizations 93

V. CONCLUSION AND FUTURE WORK 97
A. CONCLUSION 97
B. FUTURE WORK 98

APPENDIX. SAMPLE QDL CIRCUITS. 99

SUPPLEMENTAL 105

LIST OF REFERENCES 107

INITIAL DISTRIBUTION LIST 111

LIST OF FIGURES

Figure 1 Bloch Sphere. Adapted from [1]. 11

Figure 2 Graphical Description of Quantum Gates. Adapted from [1]. 14

Figure 3 Black Box Function U_f. Adapted from [1]. 16

Figure 4 Deutsch's Algorithm Quantum Circuit. Adapted from [1]. 17

Figure 5 Quantum Circuit for the Deutsch-Jozsa Algorithm. Adapted from [1]. 20

Figure 6 Quantum Circuit for Grover's Search Algorithm. Adapted from [1]. 21

Figure 7 Shor's Algorithm Circuit Diagram. Adapted from [1]. 23

Figure 8 Quantum Circuit for Bit-Flip Error Correction. Adapted from [2]. 32

Figure 9 Quantum Circuit for Phase Shift Error Correction. Adapted from [2]. 34

Figure 10 The Shor Code. Adapted from [2]. 35

Figure 11 Performance of Various Languages Relative to C across Several Benchmarks. Adapted from [25]. 46

Figure 12 Master-Slave Paradigm Diagram. 70

Figure 13 Runtime Ratio Compared to Sequential QCS versus Number of MPI Processes for Various $2*\left(H^{\otimes N}\right)$ Circuits 84

Figure 14 Runtime Ratio to Baseline versus Number of MPI Processes For Various $2*\left(H^{\otimes N}\right)$ Circuits 85

Figure 15 Total Data Transmitted versus Number of MPI Processes for Various $2*\left(H^{\otimes N}\right)$ Circuits 90

Figure 16 Total Data Transmitted versus Number of MPI Processes for Various $2*\left(H^{\otimes N}\right)$ Circuits 91

Figure 17 Total MPI Communications and Complex Amplitudes Updated per Step of the $2*\left(H^{\otimes 8}\right)$ Circuit 95

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1 Computational Steps until Decohesion for Various Qubit Implementations. Adapted from [2]. 25

Table 2 Classical Three-Bit Repetition Code 27

Table 3 Error Syndrome for Each Classical State in the Bit-Flip Code 33

Table 4 Possible Mask Values for H-Gates on a Three-Qubit Register 57

Table 5 Values after the Left-Shift Operation in the *CNOT* Gate for *N*=3. 62

Table 6 Naïve versus In-Place Simulation (1.4 GHz Intel Dual Core i5, 4 GB Memory) 67

Table 7 Naïve versus In-Place Simulation Performance (AMD Opteron 6174 CPU) 67

Table 8 Naïve versus In-Place Simulation (Intel Xeon E-5 2698 CPU)..... 68

Table 9 Teleportation Circuit Final State Vectors 76

Table 10 Single-Qubit Gate Test Circuit State Vectors After Third Transform..... 77

Table 11 Single-Qubit Gate Test Circuit Final State Vectors..... 78

Table 12 QCS Time Performance Analysis..... 80

Table 13 $2*\left(H^{\otimes N}\right)$ Circuit Sequential and Parallel Simulation Times..... 82

Table 14 Strong Scaling of the QCS on a $2*\left(H^{\otimes 10}\right)$ Circuit..... 82

Table 15 Strong Scaling of the QCS on a $2*\left(H^{\otimes 12}\right)$ Circuit..... 83

Table 16 Strong Scaling of the QCS on a $2*\left(H^{\otimes 14}\right)$ Circuit 83

Table 17 Weak Scaling of the QCS 85

Table 18 Sequential QCS Memory Allocations..... 87

Table 19 Parallel QCS Communication Cost Analysis 88

Table 20 Communications Cost Strong Scaling on a $2*\left(H^{\otimes 10}\right)$ Circuit..... 89

Table 21 Communications Cost Strong Scaling on a $2*\left(H^{\otimes 12}\right)$ Circuit 89

Table 22 Communications Cost Strong Scaling on a $2*\left(H^{\otimes 14}\right)$ Circuit 89

Table 23 Communications Cost Weak Scaling of the QCS 91

LIST OF ACRONYMS AND ABBREVIATIONS

CNOT	controlled not
CPU	central processing unit
DLP	data-level parallelism
DSM	distributed shared memory
GCD	greatest common denominator
HPC	high performance computing
IC	integrated circuit
I/O	input/output
MPI	message passing interface
QDL	quantum description language
QFT	quantum Fourier transform
TLP	task-level parallelism

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would first like to thank my thesis advisor, Professor Ted Huffmire, for teaching me the basics—and the not-so-basics—of quantum computing. Without his guidance, technical knowledge, and patience, this thesis would not have been possible.

Second, I extend my fullest appreciation to Professor James Luscombe, my second reader, for taking his time to ensure the thoroughness and correctness of the details of quantum mechanics discussed in this thesis.

Next, I would like to thank Professor Jeremy Kozdon of the Mathematics Department for teaching me the fundamentals of parallel computing, including the Message Passing Interface. The technical side of this thesis would certainly not have been realized without his assistance.

Finally, I would like to thank the entire faculty of the Computer Science Department, whose tutelage instilled in me the critical thinking skills necessary to conduct this research.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION TO QUANTUM COMPUTING

A. BACKGROUND AND QUANTUM COMPUTING FUNDAMENTALS

At their most basic level, quantum computers utilize known quantum-mechanical principles in order to execute quantum algorithms on quantum bits. Although there are currently only a few well-studied quantum algorithms, some of these quantum algorithms are theoretically able to perform certain operations substantially faster than classical (i.e., non-quantum) algorithms [1], [2]. Despite the many technological advances in both classical hardware and software over the last half-century, there are still many problems that are believed to only be solvable in super-polynomial time using classical computers—such as prime integer factorization, which is critical to virtually every cryptosystem used around the globe [3]. Prime integer factorization can take even the world's most powerful supercomputer longer than the age of the known universe to perform, but can theoretically be computed in hours or minutes using known quantum algorithms [3]–[5]. Therefore, the implications of realizing a large-scale operational quantum computer would be enormous not just in the fields of computer science or physics, but to national security as a whole.

1. Overview and Significance

The field of quantum computing is still in its early infancy; as of early 2016, the realization of a large-scale, stable quantum computer is far from being implemented due to current technological restrictions. Despite these technical limitations, however, computer scientists are able to use classical computers in order to simulate the behavior of quantum computers using software. Programs known as quantum computer simulators allow researchers in the fields of physics, mathematics, and computer science to study and develop quantum algorithms in order to learn their benefits and limitations without having to build an actual quantum computer.

Much like classical computers, quantum computers execute quantum algorithms by manipulating bits using various logic gates. The main difference is that quantum algorithms are executed upon quantum bits, or "qubits," using various quantum logic

gates, both of which differ greatly from their classical analogues. Much of the speedup obtained by quantum algorithms over classical algorithms is a result of quantum parallelism, which is made possible by harnessing a unique property of a qubit called *superposition* [1], [2], [5]. Another unique aspect of some quantum systems is an interdependence between two physically separated qubits, known as *entanglement* [1], [6]. Both superposition and entanglement in quantum systems are cleverly exploited in various quantum algorithms to enable operations that are not physically possible using classical circuits, but can be studied through quantum computer simulation [1].

2. Superposition, Quantum Parallelism, and Bra-ket Notation

Unlike classical computers, which operate on bits whose values are restricted to either zero or one at any particular time, a quantum computer operates on qubits which, prior to measurement, can have a value that is probabilistically between zero and one [1].[1] The quantum-mechanical property of a qubit being able to be between two distinct states at a particular point in time is known as superposition. Whereas a classical register of N bits can only represent a single value at a particular time, a quantum register consisting of N qubits can be in an arbitrary superposition of up to 2^N states simultaneously [4], [5]. This phenomenon is known as quantum parallelism and can be harnessed to provide massive speedup to certain operations. The quantum state of an N-qubit register represents the probability that the quantum register will be measured at a particular value inclusively between 0 and 2^N-1. Once the state of the quantum register is measured, the system is said to collapse; the actual value to which the system collapses depends on that state's probability compared to the probability of measuring other possible states, and thus quantum computers are said to be probabilistic [1], [2].

A quantum computer simulator represents the state of qubits in a quantum system as a series of matrices. In the field of quantum mechanics, the *bra-ket* notation is the

[1] The precise details and interpretation of quantum measurement is beyond the scope of this thesis. The low-level physical implementations of qubits and quantum logic gates are abstracted away in the field of quantum computing.

standard notation for representing quantum states [1], [2]. In bra-ket notation, a column vector **A** is represented as $|A\rangle$ (pronounced "ket A"), as shown in Equation (1).[2]

$$|A\rangle = \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{n-1} \end{bmatrix} = (A_0)\cdot|e_0\rangle + (A_1)\cdot|e_1\rangle + \ldots + (A_{n-1})\cdot|e_{n-1}\rangle \tag{1}$$

In Equation (1), $|A\rangle$ is the linear combination consisting of A_0, A_1, ..., A_{n-1} ,which represent scalar quantities, and $|e_0\rangle$, $|e_1\rangle$, …, $|e_{n-1}\rangle$, which are the n-dimensional orthonormal basis vectors (i.e., orthogonal unit vectors) of the state $|A\rangle$, that is:

$$|e_0\rangle = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad |e_1\rangle = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \quad \ldots \quad |e_{n-1}\rangle = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \tag{2}$$

In practice, these basis vectors are often shorthanded to $|e_0\rangle = |0\rangle$, $|e_1\rangle = |1\rangle$, …, $|e_{n-1}\rangle = |n-1\rangle$ [1], [2]. A single qubit x can be represented in ket notation by the column vector shown in Equation (3).

$$x = \begin{bmatrix} c_0 & c_1 \end{bmatrix}^T = c_0 \cdot |0\rangle + c_1 \cdot |1\rangle \tag{3}$$

Here, c_0 and c_1 are complex numbers, referred to as complex amplitudes (or complex weights), which are normalized such that $|c_0|^2 + |c_1|^2 = 1$. The value $|c_0|^2$ equals the probability of measuring qubit x in state $|0\rangle$, and the value $|c_1|^2$ is the probability of measuring x in state $|1\rangle$ [1], [2]. Note that after measurement, qubit x is said to collapse

[2] The full bra-ket notation consists of two halves: the "bra" and the "ket. In the scalar action of a linear function on a complex vector space denoted by $\langle\Phi \mid \psi\rangle$, the "bra" refers to "$\langle\Phi|$," while the "ket" refers to "$|\psi\rangle$." The whole "bra-ket" in this case would refer to the inner (dot) product of the row vector represented by $\langle\Phi|$ and the column vector represented by $|\psi\rangle$ [1], [2]. In quantum mechanics, this translates to the complex amplitude (i.e., probability) for state ψ to collapse to state Φ upon measurement [1], [2]. For the purposes of quantum computing and quantum computer simulation, the ket notation is used frequently, but the bra notation is rarely used [1], [2].

out of superposition, and the qubit can *only* have a value of either zero or one at that point. Thus, the sum of the squares of a qubit's complex amplitudes must equal 1 according to the laws of probability [1].

3. Multiple Qubit Systems and Entanglement

Two qubits x and y, can both be represented by two separate column vectors, as shown in Equation (4). Here, x_0, x_1, y_0, and y_1 are the complex amplitudes.

$$\begin{aligned} x &= \begin{bmatrix} x_0 & x_1 \end{bmatrix}^T \\ y &= \begin{bmatrix} y_0 & y_1 \end{bmatrix}^T \end{aligned} \tag{4}$$

These two separate qubits can be assembled into a quantum register consisting of least-significant qubit, y, and most significant qubit, x. In ket notation, the four possible states of this register can be written, as shown in Equation (5).

$$\begin{aligned} &|x_0\rangle \otimes |y_0\rangle \ \rightarrow \text{both qubits measured zero} \\ &|x_0\rangle \otimes |y_1\rangle \ \rightarrow x \text{ measured zero}, y \text{ measured one} \\ &|x_1\rangle \otimes |y_0\rangle \ \rightarrow x \text{ measured one}, y \text{ measured zero} \\ &|x_1\rangle \otimes |y_1\rangle \ \rightarrow \text{both qubits measured one} \end{aligned} \tag{5}$$

Here, the symbol $\otimes$ represents the Kronecker product. The value of the entire register, canonically represented by the state vector $|\psi\rangle$, is given by the full Kronecker product of the two column vectors from Equation (4), as shown in ket notation in Equation (6) [1].

$$|\psi\rangle = x \otimes y = (x_0 \bullet y_0) \bullet |0\rangle + (x_0 \bullet y_1) \bullet |1\rangle + (x_1 \bullet y_0) \bullet |2\rangle + (x_1 \bullet y_1) \bullet |3\rangle \tag{6}$$

All Kronecker products of complex vector spaces are associative, and thus the same process can be generalized for an N-qubit register with least significant qubit x_0 and most significant qubit x_{N-1} as $|\psi\rangle = |x_0\rangle \otimes |x_1\rangle \otimes \ldots |x_{N-1}\rangle$ [1], [2]. The expanded state vector representing such an N-qubit register is shown in Equation (7), where $l = 2^N$.

$$|\psi\rangle = c_0 \bullet |0\rangle + c_1 \bullet |1\rangle + \ldots + c_{l-1} \bullet |l-1\rangle \tag{7}$$

In Equation (7), each element $|k\rangle$ represents one possible value of the quantum register. The register in state $|\psi\rangle$ is said to be in a probabilistic superposition of all of its constituent basis states $|0\rangle$, …, $|l\text{-}1\rangle$. The values of the complex amplitudes c_k indicate which particular variety of superposition $|\psi\rangle$ is currently in [1].

The exact probability, $P(k)$, that a quantum system in state $|\psi\rangle$ will end up in state $|k\rangle$ after being measured is equal to the square of the Euclidean norm (i.e., the 2-norm) of the complex amplitude of state $|k\rangle$ divided by the square of the Euclidean norm of the entire system, as shown in Equation (8). Note that regardless of the complex amplitudes in the state, $P(k) \in \mathbb{R}$ such that $0 \leq P(k) \leq 1$ [1], [2], [6].

$$P(k) = \frac{|c_k|^2}{\sum_{j=0}^{l-1} |c_j|^2} = \frac{|c_k|^2}{\big|\,|\psi\rangle\,\big|^2} \tag{8}$$

According to the laws of probability, since an arbitrary system must of course be located in one of its own basis states, the probabilities of the system being measured in each basis state must sum up to 1. That is, the sum of the squares of all complex amplitudes in a quantum system must always equal 1. For the system represented in Equation (7), this means that $|c_0|^2 + |c_1|^2 + \ldots + |c_{l-1}|^2 = 1$.

As a numerical example, consider a 2-qubit register's probability of being measured in position $|2\rangle$; this is equivalent to the most significant qubit being measured in state $|1\rangle$ and the least significant qubit being measured in state $|0\rangle$. The state vector for representing the register is shown in Equation (9), and the probability calculation is shown in Equation (10) [1].

$$|\psi\rangle = \left(\frac{1}{\sqrt{2}}\right)\cdot\begin{bmatrix} i \\ 0 \\ i \\ 0 \end{bmatrix}\begin{matrix} \rightarrow|0\rangle \\ \rightarrow|1\rangle \\ \rightarrow|2\rangle \\ \rightarrow|3\rangle \end{matrix} \tag{9}$$

$$P(2) = \frac{\left|\frac{i}{\sqrt{2}}\right|^2}{\left(\sqrt{\left|\frac{i}{\sqrt{2}}\right|^2 + 0 + \left|\frac{i}{\sqrt{2}}\right|^2 + 0}\right)^2} = \frac{2}{\left(\sqrt{2+0+2+0}\right)^2} = \frac{2}{4} = 0.5 \tag{10}$$

The Kronecker product representation of a quantum register from Equation (7), combined with the probabilistic nature of any quantum system, can be used to mathematically illustrate the curious effects of the interdependency of a multiple qubit system, known as quantum entanglement. Consider the two qubit system consisting of x and y shown in Equation (6), where $(x_0 \cdot y_0) = (x_1 \cdot y_1) = 1$ and $(x_0 \cdot y_1) = (x_1 \cdot y_0) = 0$. A simple evaluation of the Euclidean norms reveals that $P(x_0) = P(x_1) = P(y_0) = P(y_1) = 0.5$. In other words, prior to measurement, both particles have precisely a 50% chance of being measured at either of their two possible states [1], [2], [6].

Suppose now that particle x is measured in position $|0\rangle$, meaning that now the probability $P(0) = 1$. Because the complex amplitude of the term $0 \cdot |x_0\rangle \otimes |y_1\rangle$ is zero, the probability that qubit y is measured in a state of $|1\rangle$ must now also be zero. Therefore, it must be the case that qubit y is in position $|0\rangle$. Similarly, if qubit x were measured in position $|1\rangle$, then qubit y must be in position $|1\rangle$.

In the system described above, qubits x and y are said to be entangled. The phenomenon of entanglement is referred to as "symmetrical," meaning that the case would be identical if qubit y were measured first. Interestingly, the physical distance between these two qubits is irrelevant. If the two qubits are implemented as electrons where the state is determined by the electron's spin, the particles could be light years away from each other and a measurement of one particle's spin would instantaneously influence the state of the second particle [1], [2].

Not all multiple-qubit quantum systems are entangled, however. Consider another possible superposition of the quantum state of a register of qubits x and y, where $(x_0 \cdot y_0) = (x_0 \cdot y_1) = (x_1 \cdot y_0) = (x_1 \cdot y_1) = 1$, as shown in Equation (11).

$$|\psi\rangle \quad = \quad 1 \cdot |x_0\rangle \otimes |y_0\rangle + 1 \cdot |x_0\rangle \otimes |y_1\rangle + \ 1 \cdot |x_1\rangle \otimes |y0\rangle + \ 1 \cdot |x_1\rangle \otimes |y_1\rangle \tag{11}$$

An arithmetic analysis of the probability of each basis state in this quantum system reveals that a measurement of either particle reveals nothing whatsoever about the position of the other particle [1]. Such a quantum system is known as *separable*, as opposed to entangled [1]. Separable quantum systems are also important to quantum computing because they allow for circuits in which certain qubits are unaffected by the measurement of other qubits [1], [2], [6].

B. INTRODUCTION TO QUANTUM GATES, CIRCUITS, AND ALGORITHMS

At the lowest level, the principles behind implementing quantum computers are similar to those of classical computers in that they both operate by manipulating bits using logic gates. All operations executed by a classical computer are the result of bits being manipulated by a series of elementary logic gates (e.g., *AND*, *OR*, and *NOT* gates). Each one of these logic gates is made up of transistors, and many thousands of logic gates are wired together into byzantine circuits forming arithmetic logic units, registers, control units, and other elements that comprise modern Computer Processing Units (CPUs). As of 2016, there are commercially available CPUs with over 5 billion transistors on them, indicating the scope of the complexity of these circuits [7]. Regardless of complexity, however, any classical circuit can be created using only *NAND* gates or *NOR* gates. These two gates are therefore commonly referred to as "universal logic gates," since any possible Boolean function can be derived from combinations of these two gates [6].

Similarly, a quantum computer performs operations on qubits using a series of quantum logic gates. Quantum logic gates differ from their classical counterparts in many ways, but one of the most fundamental differences is the concept of reversibility. Due to

Landauer's principle and the second law of thermodynamics,[3] all quantum operations, other than measurement, performed on a qubit must be completely reversible [1]. Measurement, on the other hand is necessarily always irreversible [2]. In the context of logic gates, reversibility means that after an operation has been performed on a qubit, the exact input can be recovered from any output. It is clearly impossible to recover the original input of classical gates other than the *NOT* gate since other classical logic gates (such as the *AND*, *OR*, and *XOR* gate) all have two input bits and a single output bit. It is not possible to reverse these classical Boolean operations, and as a result, these irreversible gates are not permitted at the quantum level [1], [2].

1. Quantum Gates and Circuits

Just as any classical circuit can be built only using universal *NAND* or *NOR* gates, there exist sets of universal quantum logic gates from which any possible quantum circuit can be derived. It has been proven mathematically that any operation on a quantum state can be rewritten as a series of two elementary types of quantum logic gate: single qubit gates and Controlled Not (*CNOT*) gates [8].

Quantum logic gates are described mathematically using matrix multiplication upon the column vector $|\psi\rangle$ that represents the state of the quantum system. The state vector $|\psi\rangle$ is multiplied by a transformation matrix, resulting in a new state $|\psi\rangle'$. Since all operations performed on a quantum system must be reversible, this matrix multiplication must also be reversible, meaning that the transformation matrix must be unitary [1], [2], [6].

(1) The Hadamard Gate

One of the most commonly used gates in quantum computing is the Hadamard gate, abbreviated as *H*-gate, which is used to put a single qubit into superposition. The *H*-gate is applied to a single qubit and is represented mathematically by multiplying the qubit's state vector by the unitary 2 x 2 Hadamard matrix shown in Equation (12), from [1].

[3] A full discussion of Landauer's principle and/or thermodynamics is outside the scope of this thesis.

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{12}$$

Equation (13) depicts the application of the Hadamard transform to the state vector of a qubit as described in Equation (3). Applying the *H*-gate to a qubit results in mapping the basis state of $|0\rangle$ to $\left(|0\rangle + |1\rangle\right)/\sqrt{2}$ and the basis state of $|1\rangle$ to $\left(|0\rangle - |1\rangle\right)/\sqrt{2}$, as shown in Equation (13) [8].

$$\begin{aligned} |\psi\rangle' &= H \cdot |\psi\rangle = H \cdot \left(c_0|0\rangle + c_1|1\rangle\right) \\ &= \frac{1}{\sqrt{2}}\left[\left(c_0 + c_1\right)|0\rangle + \left(c_0 - c_1\right)|1\rangle\right] \end{aligned} \tag{13}$$

Applying Equation (8) to the resultant state $|\psi\rangle'$ from Equation (13) reveals that applying a Hadamard gate to a qubit results in that qubit having exactly a 50% chance of being measured in either state $|0\rangle$ or $|1\rangle$, regardless of its initial state. Furthermore, applying the Hadamard gate twice in sequence to the same qubit will always result in the qubit reverting to its initial value. The Hadamard gate is very important in quantum computing because it allows quantum algorithms to take advantage of the property of superposition and thus harness quantum parallelism. Since quantum parallelism enables many of the unique computations that are possible in a quantum computer but not a classical computer, the Hadamard gate is utilized in nearly every quantum algorithm [1], [2], [6].

(2) The Pauli and Phase Shift Gates

In addition to the Hadamard gate, there are four more commonly used single-qubit gates: the Pauli-X, -Y, and -Z gates, as well as the phase shift gate. Each of these gates can be represented mathematically as a rotation of a qubit's state vector within the Bloch sphere, which is a three-dimensional geometric representation of the state of a qubit. In the Bloch sphere, the "north pole" and "south pole" are arbitrarily chosen to represent a qubit's basis states $|0\rangle$ and $|1\rangle$. All other possible values of the state vector $|\psi\rangle$

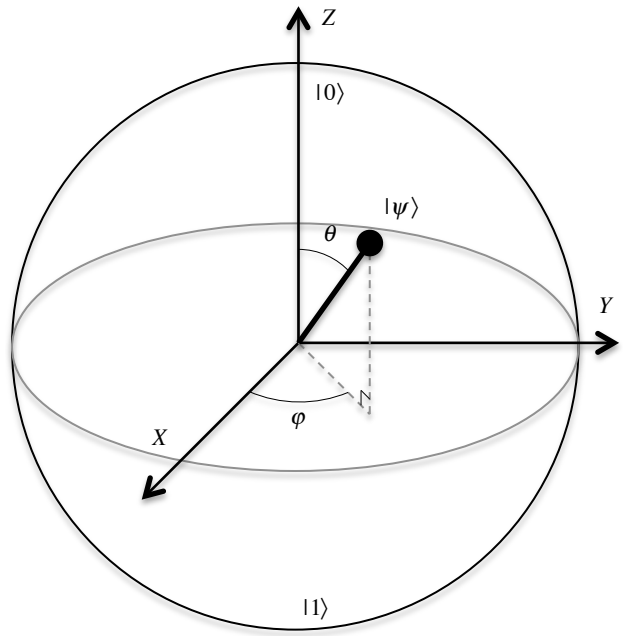
depicted in the Bloch sphere distinctly represent the infinite different possible superpositions of the qubit [1], [2], [6].

Figure 1 displays the Bloch sphere of a single qubit in an arbitrary superposition represented by the state vector $|\psi\rangle$. The angle θ depicted in Figure 1 represents the offset of $|\psi\rangle$ with respect to the z- axis, while φ represents the offset with respect to the x- axis. In ket notation, the entire state space of $|\psi\rangle$ can be represented by Equation (14), where $0 \le \theta \le \pi$ and $0 \le \varphi \le 2\pi$ [1], [2], [6].

$$\begin{aligned} |\psi\rangle &= \cos(\theta)\cdot|0\rangle + e^{i\varphi}\sin(\theta)\cdot|1\rangle \\ &= \cos(\theta)\cdot|0\rangle + \left(\cos(\varphi) + i\cdot\sin(\varphi)\right)\cdot\sin(\theta)\cdot|1\rangle \end{aligned} \tag{14}$$

The Pauli-X, -Y, and -Z gates (abbreviated as *X*-, *Y*-, and *Z*- gates, respectively) as well as the phase shift gate can be easily interpreted using the Bloch sphere representation. The Pauli-X gate rotates the state vector representing a single qubit around the x- axis by π radians. Similarly, the Pauli-Y and -Z gates represent rotations around the y- and z- axes by π radians, respectively. The *X*-gate maps a qubit in state $|0\rangle$ to a state of $|1\rangle$, and vice versa; this is equivalent to a classical *NOT* gate. The *Y*-gate has no classical equivalent, as it maps a qubit in a state of $|0\rangle$ to a state of $i\cdot|1\rangle$ and a qubit in state $|1\rangle$ to $-i\cdot|0\rangle$. Finally, the *Z*-gate does not change a qubit in basis state $|0\rangle$, but it maps a state of $|1\rangle$ to $-|1\rangle$ [1], [2], [6].

Figure 1 Bloch Sphere. Adapted from [1].



The Z gate is in fact a special case of a broader category of gates, known as the phase-shift gates. All phase shift gates leave the basis state of $|0\rangle$ unchanged but map a basis state of $|1\rangle$ to a state of $e^{i\varphi}\cdot|0\rangle$. The phase-shift gate, abbreviated $R(\phi)$, does not affect the probability of measuring a qubit in either basis state $|0\rangle$ or $|1\rangle$, but rather shifts the state vector's "latitude" on the Bloch sphere by ϕ radians. In the case of the Z-gate, the latitude shift is of $\phi = \pi$ radians, since $e^{i\pi} = -1$. The unitary transformation matrices for the X, Y, Z, and $R(\phi)$ gates are shown in Equation (15), and their ket notation action upon a qubit in state $|\psi\rangle$ is shown in Equation (16) [1], [2], [6].

$$
\begin{aligned}
X &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\
Y &= \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \\
Z &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \\
R(\phi) &= \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}
\end{aligned}
\tag{15}
$$

$$
\begin{aligned}
X \cdot |\psi\rangle &= \frac{1}{\sqrt{2}}(c_0 + i \cdot c_1) \cdot |0\rangle + \frac{1}{\sqrt{2}}(c_1 + i \cdot c_0) \cdot |1\rangle \\
Y \cdot |\psi\rangle &= \frac{1}{\sqrt{2}}(c_0 + c_1) \cdot |0\rangle + \frac{1}{\sqrt{2}}(c_1 + c_0) \cdot |1\rangle \\
Z \cdot |\psi\rangle &= c_0 |0\rangle - c_1 |1\rangle \\
R(\varphi) \cdot |\psi\rangle &= c_0 |0\rangle + c_1 e^{i\varphi} \cdot |1\rangle
\end{aligned}
\tag{16}
$$

(3) The CNOT and Toffoli Gates

The controlled-NOT gate operates on two qubits and forms a complete set of universal quantum gates together with the aforementioned single-qubit gates. The *CNOT* gate operates on a two qubit register consisting of a control qubit and target qubit; the target qubit is flipped if and only if the control qubit is in state $|1\rangle$. A two qubit register with the control qubit as qubit zero (i.e., the most significant bit) and the target qubit as qubit one is represented by the notation $CNOT_{1,0}$, and the associated transformation matrix is shown in Equation (17) [1].

$$
CNOT_{1,0} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}
\tag{17}
$$

The Toffoli gate, also known as a controlled-*CNOT* or *CCNOT* gate, is very similar to a *CNOT* gate, except that it operates on three qubits instead of two. The Toffoli

gate has two control bits, and flips the target qubit if and only if the two control bits are both in state $|1\rangle$. Equation (18) depicts the transformation of a Toffoli gate where qubits zero and one are the control qubits and qubit two is the target qubit, denoted as $T_{2,1,0}$ [1].

$$T_{2,1,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{18}$$

CNOT and Toffoli gates are both reversible (since their matrix representations are unitary), and are utilized in many quantum algorithms because they enable the entanglement and disentanglement of quantum systems. Ket notation operations of the $CNOT_{10}$ gate and the $T_{2,1,0}$ gate on a qubit in state $|\psi\rangle$ are shown in Equation (19).

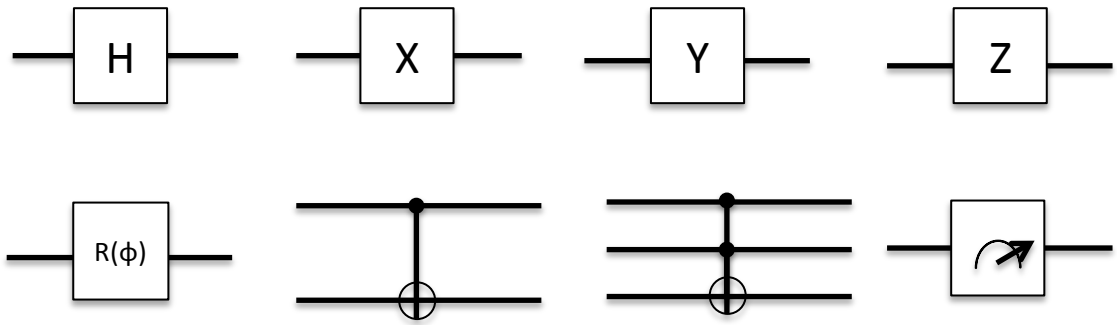
$$\begin{aligned} CNOT_{10} \cdot |\psi\rangle &= c_0|0\rangle + c_3|1\rangle + c_2|2\rangle + c_1|3\rangle \\ T_{210} \cdot |\psi\rangle &= c_0|0\rangle + c_1|1\rangle + c_2|2\rangle + c_7|3\rangle + c_4|4\rangle + c_5|5\rangle + c_6|6\rangle + c_3|7\rangle \end{aligned} \tag{19}$$

(4) From Quantum Gates to Quantum Circuits

The set of quantum gates described above, being universal gates, necessarily serve as the building blocks for any possible quantum circuit. Quantum circuits are, by convention, depicted graphically as a series of operations performed sequentially on each qubit [6]. Every qubit in a quantum register is depicted by a horizontal line separated at regular intervals. Each interval represents one "step" or operation performed in the quantum circuit. For modeling purposes, qubits are all depicted as travelling through the quantum circuit from left to right, one interval at a time, and all at the same rate. The symbols representing the previously described quantum gates are depicted in Figure 2. From left to right, top to bottom, they are Hadamard, Pauli-X, -Y, and -Z, phase-change, controlled NOT, and Toffoli gates. The bottom-right symbol indicates the measurement

of a qubit. In the CNOT and Toffoli gates, the closed circles represent the control bit and the open circle represents the target bit.

Figure 2 Graphical Description of Quantum Gates. Adapted from [1].



2. Quantum Algorithms

A quantum logic gate can be applied to the kth qubit in an N-qubit register by multiplying the state vector $|\psi\rangle$ by a series of Kronecker products of N 2x2 identity matrices (I), where the kth identity matrix is replaced by the desired gate to be applied. For example, an H-gate can be applied to the middle qubit of a 3-qubit system by multiplying the state vector $|\psi\rangle$ by $I \otimes H \otimes I$. Multiple gates can be applied in one operation using this method; Equation (20) depicts the operation required to apply H-gates to all qubits of an N-qubit register with initial state vector $|\psi\rangle$. For simplicity, the series of Kronecker products of H-gates from Equation (20) is abbreviated as $H^{\otimes N}$ [1].

$$|\psi\rangle' = |\psi\rangle \bullet \left[H_{(0)} \otimes H_{(1)} \otimes \ldots \otimes H_{(N-2)} \otimes H_{(N-1)} \right] = |\psi\rangle \bullet H^{\otimes N} \tag{20}$$

In this manner, the quantum gates and circuits described in the previous section are wired together in order to execute quantum algorithms on quantum registers. All quantum algorithms begin with their qubits in an arbitrary classical state (i.e., not in a superposition) [1]. The qubits are then placed into superposition and unitary (reversible) matrix operations are performed on the new quantum state. Finally, certain qubits are

measured, revealing the desired information [1], [2]. This section discusses some well-known quantum algorithms.

(1) Deutsch's Algorithm

Physicist David Deutsch described what is widely considered to be the first quantum algorithm, commonly referred to as Deutsch's algorithm [1], [9]. The algorithm itself is not particularly useful in terms of the computation it performs. Rather, it serves as a proof-of concept for exploiting the superposition principle to enable quantum parallelism and achieve speedup over classical algorithms [9].

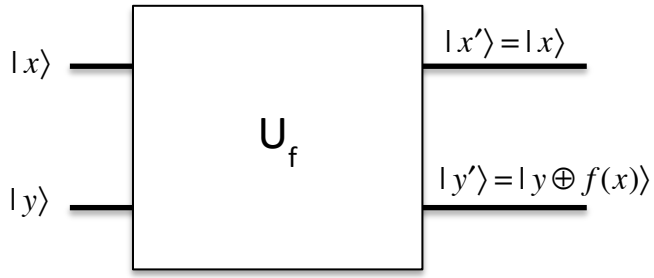
Deutsch's algorithm examines a binary function f which maps a domain of {0,1} onto a range of {0,1}. The function f is said to be *constant* if $f(0) = f(1)$ and *balanced* if $f(0) \neq f(1)$. Note that with a binary domain and range, both $f(0)$ and $f(1)$ must map to either 0 or 1 for a total of 4 possible functions $f(x)$. In order to determine if a particular unknown function is balanced or constant, a classical computer is required to evaluate $f(x)$ on both possible binary inputs and compare each output. A quantum computer, however, is able to determine whether f is balanced or constant after only a single evaluation of $f(x)$ using quantum parallelism [1], [6], [9].

Consider the balanced function $f(x)$ that maps a 0 to 1 and a 1 to 0. This is equivalent to multiplying the 2x2 matrix M in Equation (21) on the right by the initial state, clearly, $M \cdot |0\rangle = |1\rangle$ and $M \cdot |1\rangle = |0\rangle$ [1].

$$M = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{21}$$

In order to evaluate a function f, Deutsch's algorithm makes use of a reversible "black-box" function U_f, as shown in Figure 3. The black-box takes two qubits as inputs; in Figure 3, the top input $|x\rangle$ is the qubit that is being evaluated (as either balanced or constant) and the bottom input $|y\rangle$ is used to control the output. The top output, $|x'\rangle$, is equivalent to the top input, $|x\rangle$. The bottom output $|y'\rangle$, however, will be the value $|y \oplus f(x)\rangle$, where $\oplus$ indicates the exclusive-or (XOR) operation, which is equivalent to binary addition modulo two [1], [9].

Figure 3 Black Box Function U_f. Adapted from [1].

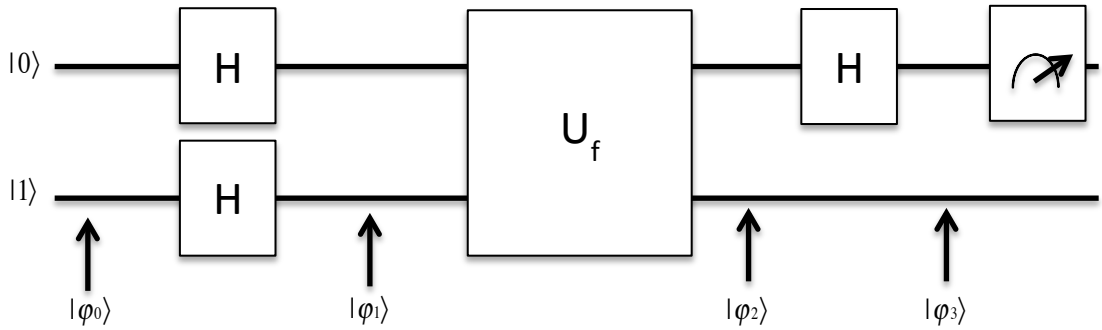


Using the function described in Equation (21), the black-box is given by the 4x4 unitary matrix U_f shown in Equation (22). Here, the index above the column indicates the input values of $|x, y\rangle$, and the index to the right of each column corresponds to the output $|x', y'\rangle$ [1].

$$U_f = \begin{array}{c} \begin{array}{cccc} \textbf{\textit{00}} & \textbf{\textit{01}} & \textbf{\textit{10}} & \textbf{\textit{11}} \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \begin{array}{c} \\ \textbf{\textit{00}} \\ \textbf{\textit{01}} \\ \textbf{\textit{10}} \\ \textbf{\textit{11}} \end{array} \tag{22}$$

Deutsch's algorithm uses this black-box function U_f in conjunction with H-gates in order to determine if f(x) is balanced or constant using only one evaluation. Figure 4 shows an example circuit for Deutsch's algorithm where the top qubit is initially in state $|0\rangle$ and the bottom qubit is initially in state $|1\rangle$. Thus, the initial value of the quantum register is $|\varphi_0\rangle = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}^T$ [1].

Figure 4 Deutsch's Algorithm Quantum Circuit. Adapted from [1].



After placing both qubits into superposition using *H*-gates, the value of the quantum register at $|\varphi_1\rangle$ is given by Equation (23) [1].

$$|\varphi_1\rangle = (H \otimes H) \cdot |\varphi_0\rangle = \left[\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right]\left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] = \begin{bmatrix} \left(\frac{1}{2}\right) & -\left(\frac{1}{2}\right) & \left(\frac{1}{2}\right) & -\left(\frac{1}{2}\right) \end{bmatrix}^T \quad (23)$$

With both quantum bits in superposition, the black-box function U_f is applied to the system. This is done by multiplying the matrix representing the black-box function by $|\varphi_1\rangle$. The black-box has no effect on the top qubit, but the bottom qubit in this example becomes $|1 \oplus f(x)\rangle$ [1], [9]. However, since at this point it is not known whether the function is balanced or constant, the value of f(x) cannot be determined. Thus, the two possible values for f(x) are left in the expression representing the quantum state $|\varphi_2\rangle$, as shown in Equation (24)[1].

$$|\varphi_2\rangle = U_f \cdot |\varphi_1\rangle = \left[\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right]\left[\frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}}\right] \quad (24)$$

Equation (24) shows the result of applying the black-box function to both qubits, which only affects the value of the bottom qubit. Note that $(-1)^{f(0)}$ and $(-1)^{f(1)}$ must have the same sign if f(x) is constant, while $(-1)^{f(0)}$ and $(-1)^{f(1)}$ must have different signs if f(x) is balanced. Therefore, the $(-1)^{f(x)}$ term can be extracted from Equation (24) and distributed through the first term as in Equation (25) [1].

$$|\varphi_2\rangle = (-1)^{f(x)}\left[\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right]\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right] = \left[\frac{(-1)^{f(0)}\cdot|0\rangle+(-1)^{f(1)}\cdot|1\rangle}{\sqrt{2}}\right]\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right] \quad (25)$$

In the aforementioned example where $f(0) = 1$ and $f(1) = 0$, the state $|\varphi_2\rangle$ becomes $\left(-1/\sqrt{2}\right)\cdot[|0\rangle-|1\rangle]\cdot[|0\rangle-|1\rangle]$. In general, the state of the quantum system at $|\varphi_2\rangle$ is shown for the four possible values of the function f are given in Equation (26) [1]. The sign on the leading (±1) term is indicates which way the function is constant or balanced (e.g., constantly 0 as opposed to constantly 1).

$$|\varphi_2\rangle = \begin{cases} (\pm1)\cdot\left[\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right]\cdot\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right], \text{ if f(x) is constant} \\ (\pm1)\cdot\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right]\cdot\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right], \text{ if f(x) is balanced} \end{cases} \quad (26)$$

The fourth state, $|\varphi_3\rangle$ is obtained from $|\varphi_2\rangle$ by applying an H-gate to the upper qubit, which is equivalent to multiplying the entire state vector by $H \otimes I$. From Equation (13), an H-gate simply maps $\left(1/\sqrt{2}\right)[|0\rangle+|1\rangle]$ to $|0\rangle$ and $\left(1/\sqrt{2}\right)[|0\rangle-|1\rangle]$ to $|1\rangle$, and therefore the final state $|\varphi_3\rangle$ is given in Equation (27), from [1].

$$|\varphi_3\rangle = \begin{cases} (\pm1)|0\rangle\cdot\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right], \text{ if f(x) is constant} \\ (\pm1)|1\rangle\cdot\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right], \text{ if f(x) is balanced} \end{cases} \quad (27)$$

In the balanced example from above where $f(0) = 1$ and $f(1) = 0$, the final state is given by $|\varphi_3\rangle = \left(-1/\sqrt{2}\right)\cdot|0\rangle[|0\rangle-|1\rangle]$. The final step in Deutsch's algorithm is to measure the top qubit; the function is constant if it is in state $|0\rangle$ and the function is balanced if it is in state $|1\rangle$. This is all performed in a single operation on a quantum computer, as opposed to two operations on a classical computer. While the problem that Deutsch's algorithm solves is contrived, it serves to demonstrate that a quantum

computer can produce a solution with one operation as opposed to the two operations required by a classical computer [1], [6], [9].

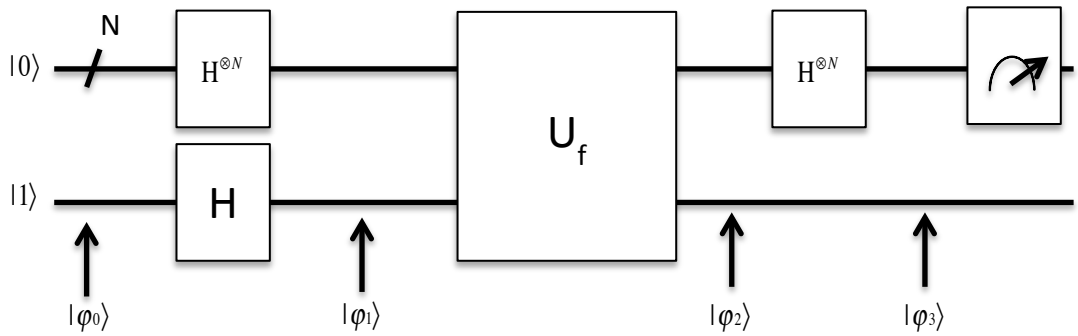
(2) The Deutsch-Jozsa Algorithm

David Deutsch collaborated with Richard Jozsa to develop the Deutsch-Jozsa algorithm, which is a generalized variation of Deutsch's algorithm [1], [10]. The Deutsch-Jozsa algorithm utilizes the same principles as Deutsch's algorithm in order to determine if a function f is constant or balanced, except the domain of the function is a register consisting of N qubits as opposed to merely a single qubit [1]. The function, now written as f: $\{0,1\}^N \rightarrow \{0,1\}$, is said to be balanced if exactly half of the inputs produce an output of 0 (the other half producing an output of 1). The function is said to be constant if all inputs go to 0 or all inputs go to 1. Furthermore, f is guaranteed to be either balanced or constant (no other possible functions are allowed in the context of the Deutsch-Jozsa algorithm). In order to determine if f is constant or balanced, a classical computer requires $2^{N-1}+1$ operations, as the computer must calculate f(x) for at least half of the possible inputs. In contrast, a quantum computer executing the Deutsch-Jozsa algorithm can determine if f is balanced or constant in a single operation, which provides significant speedup for large values of N [1], [10].

The algorithm begins with the N qubits being evaluated in a state of $|0\rangle$, and a control bottom qubit in a state of $|1\rangle$, as depicted at $|\varphi_0\rangle$ in Figure 5 [1]. Both the top N qubits and the bottom control qubit are then placed into superposition using H-gates to obtain $|\varphi_1\rangle$ [10]. All qubits are then fed through a black-box function (identical to the one from Figure 3 but with the top qubit replaced by N qubits) to reach state $|\varphi_2\rangle$. Again, the result of this black box is that the top N qubits are unchanged, and the bottom control qubit takes on the value $|y \oplus f(x)\rangle$ [1]. After the black-box function, H-gates are applied to all of the top N qubits to arrive at $|\varphi_3\rangle$. Finally, the top N qubits are measured [1], [10].

As indicated in the circuit diagram, the operations performed in the Deutsch-Jozsa algorithm are identical to those in Deutsch's algorithm except the top qubit from Figure 4 has been replaced by a register of N qubits, which is indicated whenever "$\otimes N$" appears in superscript [1].

Figure 5 Quantum Circuit for the Deutsch-Jozsa Algorithm.
Adapted from [1].



The entire matrix multiplication operation performed by the algorithm prior to measurement can be written as $|\psi\rangle' = \left(H^{\otimes N} \otimes I\right) \cdot U_f \cdot \left(H^{\otimes N} \otimes H\right) \cdot |0^{\otimes N}, 1\rangle$ [1]. The precise details of the matrix operations to arrive at each step $|\varphi_0\rangle$, …, $|\varphi_3\rangle$, however, are significantly more complex than those in Deutsch's algorithm due to the N qubits in the top register. As a result, a full analysis of each step is outside the scope of this thesis but can be found at [10]. Nevertheless, at this point, a single measurement of the top N qubits will reveal whether the function is balanced or constant [1], [10].

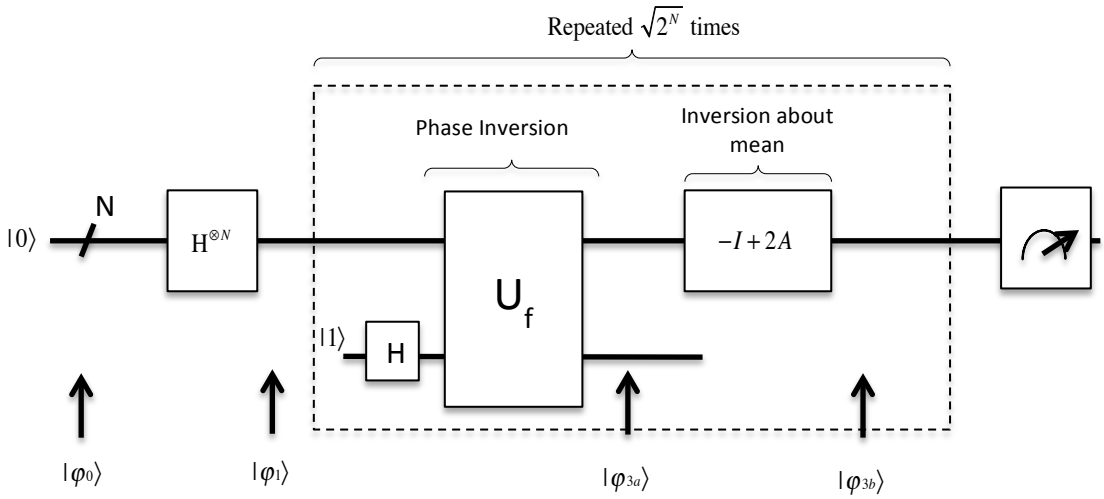
If f(x) is constant with an output of 1, the top qubits will be $(-1) \cdot |0^{\otimes N}\rangle$ upon measurement [1]. If f(x) is constantly 0, however, the top qubits will be measured in state $(+1) \cdot |0^{\otimes N}\rangle$ [1]. The top qubits will be measured at $0 \cdot |0^{\otimes N}\rangle$ if the function f is balanced. The Deutsch-Jozsa algorithm, like Deutsch's original algorithm, only solves a contrived problem that is not especially useful. Nonetheless, a quantum computer executing the Deutsch-Jozsa algorithm arrives at a solution after 1 operation, which is an exponential speedup over the $2^{N-1}+1$ operations required by a classical computer [1], [10].

(3) Grover's Search Algorithm

In 1996, Bell Labs computer scientist Lov Grover developed an algorithm that searches an unordered array of N elements in $\sqrt{N}$ queries. The fewest number of queries required on average to search such an array with a classical algorithm is $N/2$ queries [1], [11]. Grover's algorithm does not provide the exponential speedup of the Deutsch-Jozsa

algorithm, however the quadratic speedup from linear time to $O(\sqrt{N})$ is still significant improvement. Additionally, the problem of searching an unordered list is extremely useful in the field of computer science, as opposed to determining whether a function is balanced or constant. A diagram for the quantum circuit of Grover's algorithm is depicted in Figure 6.

Figure 6 Quantum Circuit for Grover's Search Algorithm. Adapted from [1].



In order to search an unordered list in sub-linear time, Grover's algorithm makes use of two unique mathematical processes known as phase inversion and inversion about the mean. Phase inversion involves changing the phase (i.e., rotation within the Bloch sphere) of the quantum state without affecting the probability of measuring any of the basis states [1], [11]. Changing a basis state's phase without changing the probability of measuring a particular value can be done by multiplying a state by *i,* -1, or both, as these operations do not affect any Euclidean norms of a quantum system. The individual elements of column vector v of length 2^N can be inverted about the mean of v by the operation shown in Equation(28) [1], [11].

$$v' = (-I + 2A)v \tag{28}$$

Here I is a 2^Nx2^N identity matrix, A is a 2^Nx2^N matrix where each element's value is 2^{-N} and v' is the column vector v with each element inverted about the mean of v.

Mathematical proof of each step of Grover's algorithm is beyond the scope of this thesis, but a full treatment of the algorithm can be found at [11].

(4) Shor's Factoring Algorithm

The best-known example of a quantum algorithm is Shor's factoring algorithm, which produces the prime factors of an integer in polynomial time—considerably faster than any known classical factorization algorithm [1], [3]. Currently, the most widely used public-key cryptography scheme is RSA, which is based on the assumption that it is difficult to factor large prime integers on a classical computer. More specifically, although no proof exists that there is not a faster classical prime integer factorizing algorithm, no known classical algorithm is able factor prime numbers more efficiently than the general number field sieve. The general number field sieve is estimated to run in $O(e^{c\cdot(\log(n))^{1/3}\cdot(\log(\log(n))^{2/3})})$ for some constant c, where n is the number of bits to be factored; this could take longer than the age of the known universe to solve for sufficiently large n [1], [3], [6]. A quantum computer executing Shor's algorithm, however, is theoretically capable of factoring integers in polynomial time. Shor's algorithm consists of both a quantum algorithm and classical post-processing. The quantum portion of the algorithm runs in $O((\log(n))^2 \cdot (\log(\log(n)) \cdot (\log(\log(\log(n))))$, and the classical post-processing runs in $O(\log(\mathrm{n}))$ [1], [3]. Therefore, if a stable, scalable quantum computer capable of executing Shor's algorithm were ever to be built, it would potentially be capable of breaking public-key cryptography schemes such as RSA in several hours or even minutes. As a result, this algorithm is perhaps the main driving force behind the push for the development of quantum computing, and has also resulted in a field of research known as post-quantum cryptography [1], [3].

Like Grover's search algorithm, the detailed mathematics behind Shor's algorithm is outside the scope of this thesis, and a full analysis of the algorithm can be found at [3]. At a high level, the algorithm executes the following steps to determine a factor, p (if p exists), of a positive integer N, where $n = \log_2(\mathrm{N})$:

1.) Classically determine if N is a power of a prime using a polynomial algorithm. If N is a power of a prime (or prime itself), then a factor has been found and no further work need be done [1].

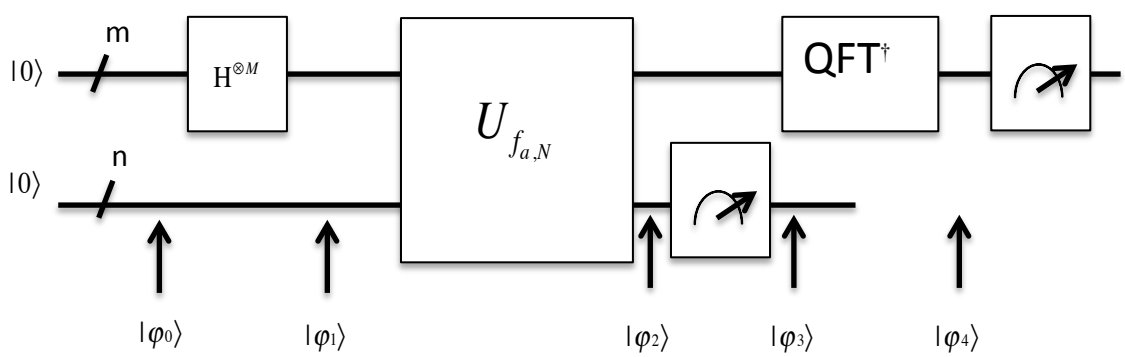
2.) Classically choose a random positive integer a (where $1 < a < N$) and determine the greatest common denominator (GCD) of a and N. This can be done in linear time using Euclid's algorithm. If the GCD is not equal to 1, this is the desired factor and the algorithm can stop here [1].

3.) Use the quantum circuit depicted in Figure 7 to determine the period r of the function given by $f(x) = a^x \mathrm{mod}(N)$. This is enabled by quantum parallelism [1].

4.) If r is odd, or if $a^r = -\mathrm{mod}(N)$, repeat to Step 2 with a different integer a [1].

5.) Classically determine the GCD of N with both $(a^{r/2}+1)$ and $(a^{r/2}+1)$, using Euclid's algorithm. The factor p is at least one of the nontrivial solutions [1].

Figure 7 Shor's Algorithm Circuit Diagram. Adapted from [1].



Shor's algorithm also makes use of an operation known as the Quantum Fourier Transform (QFT), which is the quantum analogue of the discrete Fourier transform [1], [12]. In the Shor circuit depicted in Figure 7, the Hermitian adjoint of the QFT (indicated by the superscripted †) is used to determine the period of the function $f(x) = a^x \mathrm{mod}(N)$. The precise details of the QFT are also outside the scope of this thesis, but more information can be found about it at [12].

Several groups have implemented small-scale working quantum computers that correctly execute Shor's algorithm. One group at IBM created a working Shor circuit on a quantum computer that factors 15 into 3x5 in 2001 [13]. In 2012, another group at the University of Bristol was able to develop a quantum computer capable of factorizing 21 using Shor's algorithm [14]. While these physical implementations of Shor's algorithm serve as a proof-of-concept for quantum computing, they are far from being able to crack RSA encryption schemes. One of the main challenges with Shor's algorithm is that a different circuit is required for factoring different integers, and there is no cookbook method for developing these so called "compiled circuits." In order to break a 2048-bit RSA key, it is estimated that a quantum computer with 4,000 qubits and over 100 million gates would be required [4]. There are many reasons preventing the implementation of a quantum computer of this size, but perhaps none is more important than the issue of quantum error correction.

C. DISSIPATION, DECOHERENCE, AND QUANTUM ERROR CORRECTION

The largest hurdle to the realization of a scalable quantum computer is a result of qubit errors that fall into two broad categories: errors caused by energy dissipation and those caused by quantum decoherence [1], [2]. All known physical implementations of individual qubits (such as the state of a single photon or the spin on a single electron) are inherently unstable because a single qubit is impossible to isolate from its surroundings [2], [12]. The circuits described in Section B.1 all assume ideal logical qubits free from error due to environmental interactions [12]. However, a quantum register cannot exist in isolation—there must of course be some supporting infrastructure to maintain control over a quantum register in order to perform useful operations upon it using quantum logic gates [2]. Invariably, all qubits will interact with this supporting infrastructure, which will necessarily alter their state over time [1].

The phenomenon of decoherence is described by [1] as the "loss of purity of state" of a quantum system that occurs when that system has an interaction with its surrounding environment that causes it to lose information. Essentially, this means that the states of individual qubits become unstable as a result of external environmental

factors such as ambient heat, cosmic ray interaction, or simply coupling with the matter comprising the quantum circuit itself [2].

In addition to qubit errors caused by decoherence, all qubits dissipate quantum energy, causing a change in state. Dissipation occurs when a qubit loses energy to its surroundings [2]. This can be due to a variety of reasons such as spontaneous photon emission or interaction with gas molecules within the quantum circuit [2]. All qubits are subject to both decoherence and dissipation in relatively short but unpredictable time intervals [2]. When a qubit experiences either phenomenon, the information encoded within it is necessarily, but not always irreversibly, lost or altered [12].

The expected time for a qubit to undergo decoherence varies widely depending on the physical implementation of the qubit [2]. Table 1 depicts the expected "cohesive lifespan," as well as the theoretical time required to perform a single quantum gate operation for several different physical qubit implementations. As shown, even the most stable qubit implementation—that based on the spin of a molecule's nucleus—has an expected lifespan of less than 3 hours, at which point the information encoded within the nucleus is lost [2]. As a result, errors due to quantum decoherence are very common in quantum computers [12]. Fortunately, there exist several intricate methods to detect and correct the errors caused by both decoherence and dissipation [1], [2], [6], [12].

Table 1 Computational Steps until Decohesion for Various Qubit Implementations. Adapted from [2].

Physical Qubit Implementation	Time per Gate Operation (s)	Expected Coherence Duration (s)	Steps Until Decohesion
Trapped Indium Ions	10^{-14}	10^{-1}	10^{13}
GaAs Electrons	10^{-13}	10^{-10}	10^{3}
Electron Spin	10^{-7}	10^{-3}	10^{4}
Electron Quantum Dot	10^{-6}	10^{-3}	10^{3}
Nuclear Spin	10^{-3}	10^{4}	10^{7}

1. Classical Error Correction

The concept of error detection and correction is not peculiar to the world of quantum computing; classical computers frequently undergo spontaneous bit-flip errors, and many robust methods exist to detect and correct these errors [2]. Correcting bit-flip errors in classical computers is not without cost, however, as additional bits are required to correct the error [1], [2], [15]. In general, the more robust the error detection/correction method, the higher the overhead required in terms of the number of physical bits that are required for each logical bit that is used to perform actual computation [2], [6], [15].

In coding theory, classical error correction has grown into a vast field of its own. The error correcting codes employed in modern computers and telecommunication systems have been thoroughly studied and are generally considered extremely reliable [1], [15]. Much of this work was developed by NASA based on the need to accurately send information to spacecraft over great distances through channels with very low signal-to-noise ratios [2]. In general, most classical error correcting codes map logical bits together with error correcting bits to create "codewords." These codewords have been specifically engineered to have the maximum possible Hamming distance[4] between them [2]. If a codeword becomes corrupted due to a bit flip error, the correct codeword can be recovered by replacing the incorrect codeword with the closet legal codeword (in terms of Hamming distance) [2], [12], [15].

One of the simplest possible implementations of classical error correction is the so-called three-bit repetition code described in [15]. Consider information being sent from Alice to Bob over a binary symmetric channel. For the sake of this example, assume that each logical bit, b, sent over this channel has an equal probability, p, of being flipped in transit. In other words, Bob receives the correct bit b with probability $(1-p)$, but receives a flipped bit with probability p. Here, p represents the inverse of the signal-to-noise ratio within the symmetric channel and must be less than 0.5 in order to enable

[4] The Hamming distance between two strings of equivalent length is defined as the number of places in which the two strings differ [2]. For example, binary strings *000* and *010* would have a Hamming distance of 1.

successful communication (clearly the channel is useless if $p = 0.5$ and with $p > 0.5$, Bob can simply flip each bit he receives) [15].

Using the three-bit repetition error correcting code from [15], Alice creates a codeword with a length of three physical bits by repeating each logical bit, *b,* two additional times. If Alice wishes to transmit a 0 to Bob, she instead sends 000. In the case of a 1, the codeword would be 111. Bob decodes each three-bit codeword he receives by a simple majority: he decodes *b* as 0 if two or more physical bits are 0, and he decodes a 1 if two or more physical bits are 1 [15]. Table 2 depicts all of $2^3 = 8$ possible codewords that Bob can receive as well as their associated decoding.

Table 2 Classical Three-Bit Repetition Code

Codeword Received by Bob	Bob's Decoded Logical Bit (*b*)
000	0 (error free transmission)
001	0
010	0
100	0
111	1 (error free transmission)
110	1
101	1
011	1

If fewer than two of the bits have been flipped during transmission, Bob can now recover the correct logical bit, *b*, with probability $3p^2-2p^3$ (note that this quantity is always smaller than p when $p < 0.5$) [15]. The three-bit repetition code will only correct a single bit flip for every three physical bits, and is not as robust as many of the methods currently in use to correct bit-flip errors [2], [15]. Nonetheless, it serves as a standard example for implementing bit-flip error correction on classical computers.

2. Quantum Error Correction

Bit-errors are much more common on quantum computers than on their classical counterparts due to the inherent instability of a qubit [12]. Error correcting codes must therefore be employed in all quantum circuits in order to ensure that decoherence and dissipation related errors do not cause quantum circuits to produce incorrect results during computation [1], [6]. Additionally, there are different types of errors possible when encoding information in a qubit that are not possible with a classical bit [2], [12]. The only type of bit-error possible with a classical bit is a bit-flip error, since a bit has only two possible states [2]. A qubit, however, can be in the infinitely many different superpositions represented within the Bloch sphere [1]. Therefore, instead of merely flipping from a state of 0 to 1 or vice versa, a qubit can shift to infinitely many different locations within the Bloch sphere, each resulting in a loss of the intended information encoded within that qubit's state [2].

All classical error correcting codes, such as the three bit repetition code, are made possible because of the fact that the basis state of a bit (0 or 1) can be inspected, and even copied for redundancy, at any point during computation or transmission [1], [2], [6], [12]. In a quantum system, however, it is impossible to measure the state of a qubit without irreversibly collapsing it out of its superposition; this is known as the no-cloning theorem [1], [2]. It is therefore not possible to inspect or copy a qubit without potentially altering that qubit's state. As a result of the no-cloning theorem, different mechanisms of error correction must be employed [2].

In order to avoid collapsing a logical qubit out of superposition, each logical qubit in a quantum circuit must be encoded into a codeword that is entangled with other physical qubits in such a way that any undesired shift within the Bloch sphere of the logical qubit can be inferred without directly inspecting that qubit [2]. The first quantum error correcting code was developed in 1995 by Peter Shor in [16], and since then, other codes for detecting and correcting qubit errors have been developed [2]. These codes entangle logical qubits with other physical qubits in such a way that a subset of the physical qubits can be measured without interfering with the logical qubits [16]. This allows the circuit to determine what [6] refers to as an "error syndrome." The error

syndrome indicates the type of error that has occurred to the logical qubit so that it can be reversed by the application of the appropriate single-qubit gates from Section B.1 [6]. Just as in classical error correction, each one the different methods for detecting and correcting logical qubit errors carries the cost of requiring additional physical qubits, as well as syndrome qubits, for each logical qubit [1], [2], [12].

In general, there are three main categories of bit-error for a qubit: bit-flip errors, phase-flip errors, as well as combined bit-flip and phase-flip errors [2], [6]. In a quantum computer, a bit-flip error is induced when a qubit dissipates energy, and a phase-flip is caused by decoherence [2], [16]. Since any 2 x 2 matrix can be rewritten as a sum of scalar multiples of the Pauli matrices from Equation (15), it is useful to model qubit errors using these matrices [6]. Moreover, since the Pauli matrices are all unitary, their action upon a quantum system is reversible, meaning that any qubit error can be reversed by simply applying the appropriate quantum gate from Section B.1 [2], [6], [16].

(1) Types of Qubit Errors

A bit-flip in a qubit is the same as in a classical bit; bit-flip errors map qubits in a basis state of $|0\rangle$ to a state of $|1\rangle$, and qubits in a basis state of $|1\rangle$ to a state of $|0\rangle$ [2]. For a qubit in an arbitrary superposition $|\psi\rangle$, as shown in Equation (29), a bit-flip error is equivalent to multiplying the qubit's state by the Pauli-X matrix from Equation (15) [6].

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{29}$$

Equation (30) shows the multiplication of the Pauli-X matrix carried through the state vector $|\psi\rangle$ from Equation (29) [2]. If a bit-flip error has been detected in a qubit, it can be reversed by simply applying an X-gate to that qubit [6].

$$X \cdot |\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \left(\alpha|0\rangle + \beta|1\rangle\right) = X \cdot \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \alpha|1\rangle + \beta|0\rangle \tag{30}$$

One type of error that is possible in a qubit but not a classical bit is a phase shift error [2], [12]. A phase shift error in a qubit can be modeled by multiplying the state

vector of the qubit by the Pauli-Z matrix, as shown in Equation (31), and can be reversed by applying a Z-gate to the affected qubit [6].

$$Z \cdot |\psi\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \cdot \left(\alpha |0\rangle + \beta |1\rangle\right) = Z \cdot \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \alpha |0\rangle - \beta |1\rangle \tag{31}$$

Qubits can also be subjected to bit-flip and phase shift errors concurrently [2], [6]. This type of error, referred to as a "bit-flip and phase shift error" can be modeled by multiplying the qubit's state by the Pauli-Y matrix, as depicted in Equation (32) [2]. Again, a combined bit-flip and phase shift error can be undone by applying a Y-gate to the error-afflicted qubit [6].

$$Y \cdot |\psi\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \cdot \left(\alpha |0\rangle + \beta |1\rangle\right) = Y \cdot \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = i \cdot \alpha |1\rangle - i \cdot \beta |0\rangle \tag{32}$$

3. Quantum Error Correction Codes

There are many quantum circuits designed to detect and correct the qubit errors that inherently develop due to decoherence and dissipation [1], [2], [6]. These quantum error correction circuits can be tested using a QCS by intentionally introducing an error into the quantum system to ensure that the circuit detects and reverses the error [2]. Circuits designed to test various quantum error correcting schemes typically exhibit the following common characteristics [2]:

1.) Begin with a quantum register of logical qubits and physical qubits in an entangled state, $|\psi\rangle$.

2.) Introduce an error into the entangled state.

3.) Further entangle the state with syndrome qubits in order to decode the entangled quantum state (including the introduced error) without disturbing the physical qubits.

4.) Determine the error syndrome based on this inspection.

5.) Apply the appropriate unitary matrix, or matrices, to correct the error and return the quantum state to its intended original value. If the error correction scheme has worked correctly, the quantum register will return to its initial value $|\psi\rangle$ [2].

Some of the most elementary error correcting codes are based on the classical three-bit repetition code and serve to identify either bit-flips or phase shifts in a qubit [2], [6]. After the type of qubit error has been correctly diagnosed, it can then be reversed using single-qubit gates [2]. The codes to identify and correct both bit-flips and phase shifts can be joined together in order to detect and correct a combined bit-flip and phase shift, as shown by Shor in [16].

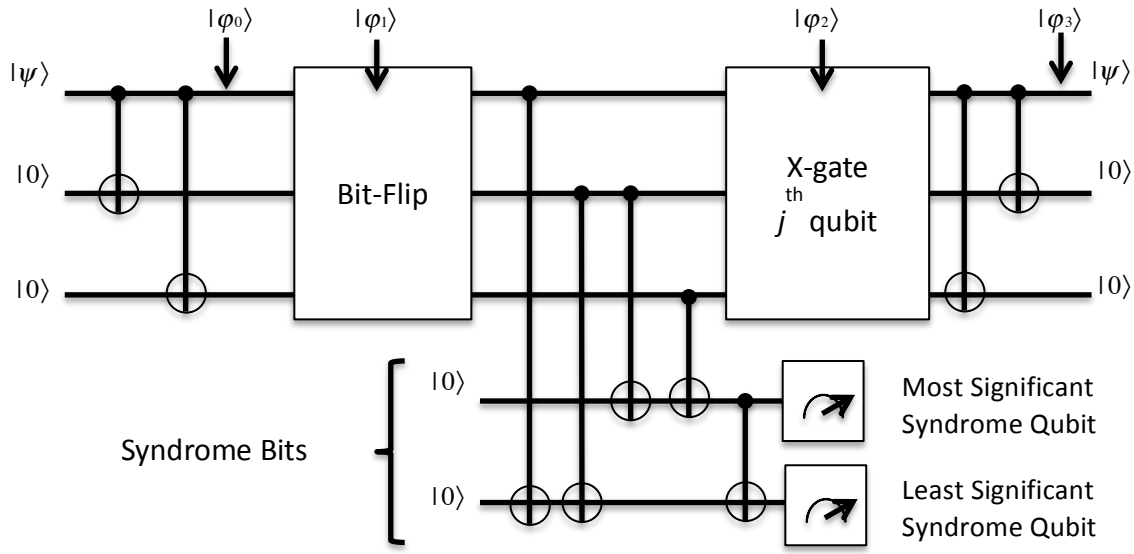
(1) Repetition Code for Bit-Flip Errors

Perhaps the most basic quantum error correcting code is the circuit designed to identify and fix a single bit-flip error [2]. A single logical qubit is encoded into a three-qubit codeword, as shown in Figure 8. The three physical qubits are entangled using *CNOT* gates, after which a single bit-flip of at most one of the three physical qubits can be detected and corrected [2], [6]. Equation (33) shows how the top qubit can be encoded with the two other physical qubits at step $|\varphi_0\rangle$; the top qubit is not being copied or inspected, so this operation is permissible on the quantum level [6].

$$|\psi\rangle = (\alpha|0\rangle + \beta|1\rangle) \Rightarrow \alpha|000\rangle + \beta|111\rangle \tag{33}$$

Next, at step $|\varphi_1\rangle$ of the simulation, a bit-flip error is introduced intentionally to any one of the three physical qubits [2]. If the top qubit becomes flipped, the state becomes $\alpha|100\rangle + \beta|011\rangle$. A bit-flip in the middle or bottom qubit will result in a state of $\alpha|010\rangle + \beta|101\rangle$ or $\alpha|001\rangle + \beta|110\rangle$, respectively [6].

Figure 8 Quantum Circuit for Bit-Flip Error Correction. Adapted from [2].



After introduction of the bit-flip error, syndrome bits are introduced to the circuit as target bits in a series of *CNOT* gates in order to help diagnose the error based on the parity of the top and middle bit, as well as the parity of the middle and bottom bit [2]. The next five *CNOT* gates begin the decoding process in order to determine the error syndrome. The first two *CNOT* gates determine the parity of the top and middle qubits, while the second two *CNOT* gates determine the parity of the middle and bottom qubits [2]. The syndrome qubits can then be measured in order to tell, with certainty, which physical qubit has undergone a bit-flip error. Table 3 depicts the state of the syndrome bits that result from each possible classical state of the top three physical qubits.

Once the error syndrome has been identified, an X-gate can be applied to the appropriate qubit at step $|\varphi_2\rangle$ in order to reverse the error [2]. If the syndrome bits are in state $|00\rangle$, there is no bit-flip error and nothing need be done [2]. For any other possible state of the syndrome bits, however, an X-gate must be applied to the jth physical qubit, where $j \in \{1, 2, 3\}$ is the decimal representation of the two syndrome bits [2], [6]. After applying an X-gate to the appropriate qubit at $|\varphi_2\rangle$, the reverse of the encoding procedure from Equation (33) is applied using two more *CNOT* gates. The result at $|\varphi_3\rangle$ is the

recovery of the top three physical qubits in their original state [6]. Important limitations of this circuit are that it only detects bit-flips, and only in one of the top three qubits.

Table 3 Error Syndrome for Each Classical State in the Bit-Flip Code

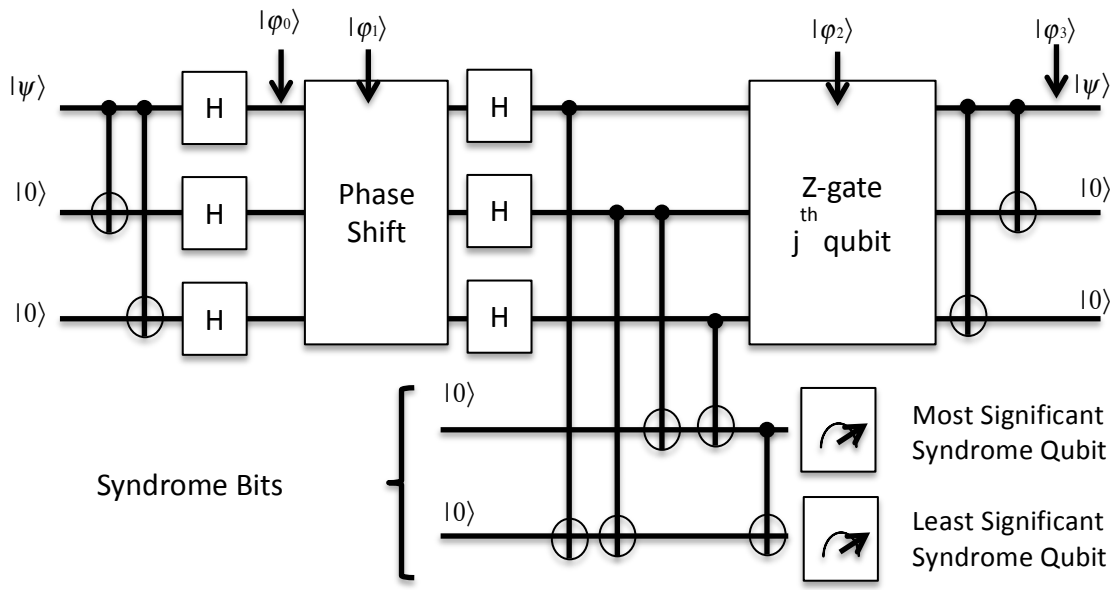
Classical State of Physical Qubits	State of Syndrome Qubits
$\|000\rangle$	$\|00\rangle$
$\|001\rangle$	$\|11\rangle$
$\|010\rangle$	$\|10\rangle$
$\|011\rangle$	$\|01\rangle$
$\|100\rangle$	$\|01\rangle$
$\|001\rangle$	$\|10\rangle$
$\|110\rangle$	$\|11\rangle$
$\|111\rangle$	$\|00\rangle$

(2) Repetition Code for Phase Shift Errors

Many of the same ideas used in the bit-flip quantum error correcting circuit are also used to detect and correct phase shift errors. After entangling the logical qubit with two physical qubits using *CNOT* gates, as in the bit-flip correcting code, the phase shift circuit applies an H-gate to each physical qubit [2], [6]. Note that since $(X+Z)/\sqrt{2} = H$ (from Equations (12) and (15)), a bit-flip in the X basis is analogous to a sign flip in the Z basis [6]. In other words, a bit-flip error is described by $X\cdot|0\rangle = |1\rangle$ and $X\cdot|1\rangle = |0\rangle$, while a phase shift error can be described by $Z\cdot|+\rangle = |-\rangle$ and $Z\cdot|-\rangle = |+\rangle$, where $|+\rangle$ and $|-\rangle$ are shorthand for the expressions given in Equation (34) [2].

$$|+\rangle = \frac{(|0\rangle + |1\rangle)}{\sqrt{2}}$$
$$|-\rangle = \frac{(|0\rangle - |1\rangle)}{\sqrt{2}} \tag{34}$$

Figure 9 Quantum Circuit for Phase Shift Error Correction.
Adapted from [2].



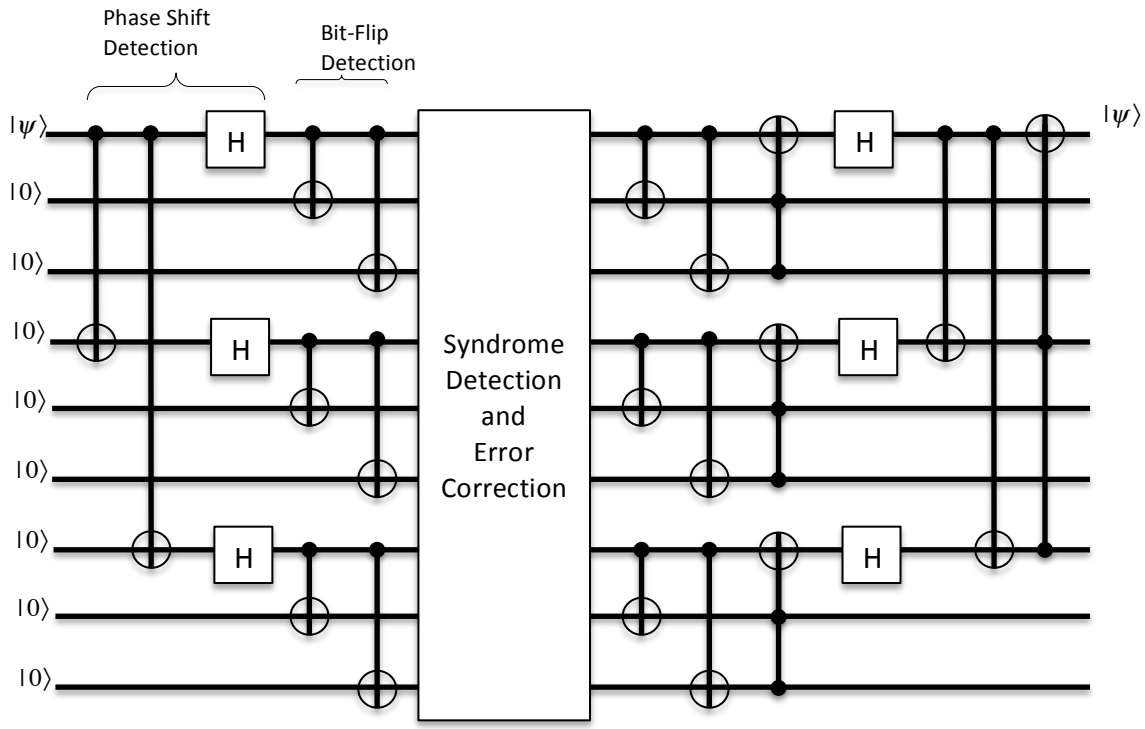
After a phase shift is introduced to one of the top three qubits at $|\varphi_1\rangle$, an H-gate is applied to all three physical qubits and the rest of the circuit is nearly identical to the bit-flip error correcting circuit [2]. The only minor difference is that a Z-gate instead of X-gate is applied to the jth qubit, where j is once again determined by the syndrome bits [2], [6]. Just as the bit-flip code can only correct a single bit-flip among the three physical qubits, the phase shift code can only correct a single-phase shift among the three logical qubits [6].

(3) Shor's Nine-Qubit Error Correcting Code

In order to protect against both bit-flip and phase shift errors, the three-qubit bit-flip code and the three-qubit phase shift code can be concatenated [2], [16]. This is the

essence of Shor's nine-qubit error correcting code (commonly referred to as simply the "Shor code"), which encodes a single logical qubit into nine physical qubits such that any type of qubit error can be detected and reversed [16]. Figure 10 depicts a circuit diagram for the Shor code.

Figure 10 The Shor Code. Adapted from [2].



Each of three logical qubit first encoded according to Equation (35), where a subscripted "L" indicates a logical qubit.

$$
\begin{aligned}
|0\rangle_L &= \left(\frac{1}{2\sqrt{2}}\right)\left(|000\rangle + |111\rangle\right)\otimes\left(|000\rangle + |111\rangle\right)\otimes\left(|000\rangle + |111\rangle\right) \\
|1\rangle_L &= \left(\frac{1}{2\sqrt{2}}\right)\left(|000\rangle - |111\rangle\right)\otimes\left(|000\rangle - |111\rangle\right)\otimes\left(|000\rangle - |111\rangle\right)
\end{aligned}
\tag{35}
$$

The code that protects against phase shifts is implemented first, as indicated by the *CNOT* and H-gates on qubits one, four, and seven [16]. Next, these three qubits are

entangled using the remaining *CNOT* gates in order to detect bit-flips [16]. Each triple of qubits (i.e., qubits {1,2,3}, qubits {4,5,6}, and qubits {7,8,9})is responsible for detecting bit-flip errors [2]. The error syndrome is then determined and the appropriate unitary matrix applied to the qubit with the error in a manner similar to the previous two error correcting codes but redacted for brevity [16]. Finally, the encoding of the nine physical qubits is reversed, and at this point, their final state is identical to their initial state [16]. The Shor circuit protects against a single qubit error of any kind in the nine physical qubits [2]. A mathematically detailed, step-by-step description of the Shor code is available in [16].

D. QUANTUM COMPUTER SIMULATION

A quantum computer simulator is nothing more than a software program run on a classical computer in order to simulate the effects of quantum gates on a register of qubits. The process of simulating the operations of a quantum computer using a classical computer is conceptually straightforward. The program begins with an initial state vector *a,* which represents the initial state of a quantum register. This state vector must begin the simulation in a classical state, devoid of superposition. Next, the vector *a* is multiplied by the matrices corresponding to the appropriate quantum gates and/or black-box functions, using the matrix multiplication described in Section B.1. All of the quantum algorithms outlined in Section B.2 can be simulated on a classical computer by simply carrying out the complex vector space matrix multiplication operations representing each gate, or series of gates.

Measurement of a qubit, or register of multiple qubits, can be simulated through the use of a pseudo-random number generator. A pseudo-random number, d, is generated (such that $0 \leq d \leq 1$) and compared to the probability of measuring the qubit in each one of its basis states, from Equation (8). If d is smaller than the probability of measuring the qubit in state $|0\rangle$, the simulation has "measured" the qubit in a state of $|0\rangle$. Otherwise, the simulation has measured the qubit in a state of $|1\rangle$. Using the pseudo-random number generator built-in to most programming languages is an easy way to simulate the probabilistic effects of measuring a qubit in superposition. This method of simulating

qubit measurement combined with the matrix operations for both single-qubit and *CNOT* gates allows the simulation of a universal quantum computer capable of simulating any and all quantum operations.

E. LIMITATIONS OF SEQUENTIAL QUANTUM COMPUTER SIMULATORS

It is easy to simulate the effects of a quantum circuit on a quantum register consisting of a small number of qubits using naïve matrix multiplication as previously described. The simulation of quantum systems using this method, however, is limited by the fact that the sizes of the matrices involved grow exponentially with the size of the quantum system. In order to simulate an N-qubit quantum system, a complex vector, $|\psi\rangle$, of length 2^N must be stored [1], [6]. This means that nearly 1 TB of memory is required to store a single arbitrary state for a quantum register with 36 qubits [8]. Additionally, performing operations on these quantum states requires exponentially more resources. For example, application of a single H-gate to the kth quantum bit of an N-qubit quantum register requires multiplication of the register's state vector $|\psi\rangle$ by the matrix $M = I_{(0)} \otimes \ldots \otimes H_{(k)} \otimes \ldots \otimes I_{(N-1)}$. M is a dense matrix with dimensions 2^Nx2^N; clearly, as the size of a quantum system grows, it rapidly becomes impossible to simulate that system using a classical computer [8].

The exponential growth of the matrices involved with simulating a quantum system puts both temporal and spatial limits on the size of system that can be simulated on a classical computer [8]. It is certainly possible to study very small (on the order of 10–20 qubits on the average personal computer) quantum circuits using the matrix multiplication method of simulating quantum states. However, many optimizations are required in order to study much larger circuits. The first of these optimizations is to perform the matrix multiplication operations using an in-place algorithm. This greatly decreases both the time and memory required to simulate the effects of quantum gates upon a system's state vector. Next, the operations can be parallelized across multiple processors in order to increase the amount of memory available to simulate the quantum systems. Both of these optimizations will be discussed in detail throughout the remainder of this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

II. INTRODUCTION TO PARALLEL COMPUTING

A. OVERVIEW OF PARALLELISM IN COMPUTING

Over the last 70 years, computer hardware has made constant and rapid progress in terms of processing power [17]. One of the major driving forces behind this increase in processing power is the doubling of the transistor count on integrated circuits (ICs) roughly every 18 to 24 months—an empirically observed trend commonly referred to as Moore's law [17]. A large reason for the transistor count increasing at such a fast pace is that transistors are consistently becoming smaller; ICs with transistors roughly 14 nm in length became commercially available in 2015 [18]. As transistors approach the size of atoms however, there must of course be some fundamental lower limit to the size of a transistor and thus to Moore's law. The trend in pure transistor density increase has also slowed down in the past decade due to power-density and heat dissipation concerns [19]. Additionally, a trend known as Rock's law, which states that the cost of fabricating a foundry for these ICs roughly doubles every four years, also operates in competition with Moore's law [17]. As a result of Rock's law, building a state-of-the-art foundry as of 2015 costs around $14 billion, which is prohibitively expensive for most organizations [20]. Physical transistor size limits, heat dissipation, and Rock's law are all factors are inhibiting the growth of chips in terms of pure transistor count. However, other computer hardware developments have enabled processors to perform computation with significantly more efficiency, regardless of transistor count [17], [21].

Most advancements in computer architecture that do not involve increasing transistor count on an IC are a result of harnessing various levels of parallelism in order to perform tasks more efficiently [17], [21], [22]. In computer science, parallelism is an overloaded term that refers to performing multiple computations simultaneously, either on the same processor or distributed among various processors. Hennessy and Patterson [17] have classically grouped all types of parallelism into two broad categories: Data-Level Parallelism (DLP), where many separate elements of data are operated on simultaneously, and Task-Level Parallelism (TLP), in which different tasks are created that can be executed independently.

1. Bit Level Parallelism

Bit level parallelism was the first and most primitive type of parallelism exploited by computer architects in order to improve processor performance [21]. This early form of DLP involved increasing the size of a computer "word," which is the amount of data that a computer can manipulate in a register in one clock cycle. From about 1970 to 1986, the word size used by most general purpose processors doubled incrementally from 4-bit words to 32-bit words [21]. In the mid-1990s, this trend increased to 64-bit words with the advent of the backwards-compatible Intel x86-64 architecture [21].

Increasing the word size of a computer's registers clearly enables the computer to process more information at once. For example, a computer with an 8-bit register can easily add two 8-bit numbers in a single operation. On the other hand, a computer with registers consisting of only 4 bits that attempts to add two 8-bit numbers must first add the least significant 4 bits, and then, in a separate instruction, add the most significant 4 bits together with a carry value. As of the early 2000s, this type of parallelism has plateaued among general-purpose CPUs at 64-bit words, which leaves sufficient accuracy for floating point number representation in most situations [22].

2. Instruction Level Parallelism

At a higher level of abstraction than bit-level parallelism lies another type of DLP, known as instruction level parallelism. Using instruction level parallelism allows a single CPU to execute multiple instructions concurrently [21]. Instruction level parallelism takes advantage of how a CPU executes individual instructions, and utilizes concepts such as instruction pipelining and speculative execution to execute and schedule instructions more efficiently [17], [21].

Instruction pipelining works by overlapping the execution of multiple instructions at different stages of their instruction cycles [17]. Pipelining takes advantage of the fact that CPU instructions are typically executed in discrete steps, each of which are performed by separate, dedicated hardware entities [21]. Instruction cycles vary widely from chip to chip. The classic example of a CPU instruction cycle consists of the following steps: fetch, decode, execute, and write [21]. The CPU first retrieves the next

instruction from memory during the fetch step, then decodes the fetched instruction. The decoded instruction is then executed, and if required, the results are written back to memory or another register [21]. The instruction cycle is then repeated for the next instruction to be executed. Older CPUs performed these steps sequentially for every instruction in a computer program. However, executing instructions sequentially is extremely inefficient since reading from and writing to memory are temporally expensive operations compared to most computation [17]. Therefore, many CPU clock cycles are wasted if the CPU is idle while waiting for the next instruction to be fetched or a value to be written to physical memory. Modern CPUs implement instruction-level parallelism through utilization of an instruction pipeline that prevents wasted clock cycles [17], [21].

In an instruction pipeline, a CPU executes multiple instructions concurrently, each at different steps of their instruction cycle. In the example instruction cycle given above, a CPU can theoretically be in the process of executing four different instructions during any particular clock cycle. Each of these instructions is at a different stage of the fetch, decode, execute, and write cycle [17]. It is worth noting that this can be done only if there are no data or control dependencies between the instructions being concurrently executed. All modern CPUs utilize an instruction pipeline with various levels of complexity [17]. The Intel Pentium 4, for example, has a 31 stage instruction cycle, enabling a very high-throughput but very complicated instruction pipeline [23].

In conjunction with pipelining, modern processors utilize the techniques of speculative execution and branch prediction in order to speed up computation [17]. If a processor has clock cycles that are not being used for whatever reason, it can perform computation during these idle clock cycles before it knows whether or not the results of this computation will be used at all [17]. A classic example of this is a conditional branch of code (i.e., an “if” statement). A CPU with available resources can compute the results of a conditional branch before it actually evaluates the predicate of the conditional and determines which branch will be taken. Moreover, a CPU can attempt to predict which conditional branch will be taken before the conditional predicate is evaluated; this is known as branch prediction and is used together with speculative execution in nearly all modern processors. There are many different implementations of branch prediction and

speculative execution, but their overall effect is to reduce response time in many situations [17], [21], [22].

Instruction level parallelism is almost always implemented implicitly (either in hardware, software, or both), with little to no involvement of the programmer or user [17], [21]. In hardware, instruction level parallelism is exploited dynamically at run time, whereas in software, it is discovered by the compiler and exploited statically at compile time [17]. In either case, the end user need not be aware that instruction level parallelism is occurring, and the result is indistinguishable from a faster, more efficient processor. One main advantage of this is that programs written sequentially are implicitly exploited for parallel execution by both the compiler and the processor. In order to exploit higher levels of parallelism between multiple threads or processors, however, more advanced parallel programming techniques are required [17], [21], [22].

3. Thread Level Parallelism across Multiple Processors

Parallelism at the thread level can be both DLP and TLP [17]. Most modern operating systems coordinate the execution of multiple threads on a single processor; this technique is known as multithreading [21]. Multiple threads can also be distributed to perform computation simultaneously across multiple separate processors according to either a tightly-coupled or loosely-coupled memory model. In the tightly-coupled memory model, referred to in [17] as "centralized shared-memory multiprocessing," a group of processors are controlled by a single operating system and perform computation using the same memory and address space. This model of shared-memory processors is commonly used in multi-core processors on today's personal computers, usually with eight or fewer total processors [17]. The advantage of having tightly-coupled processors is that communication between the processors is easy since the processors all have access to the same memory. This also means that all processors have the same memory latency, and as a result, this type of multiprocessor is also referred to as a "uniform memory access multiprocessor" [17].

As the number of processors increases, however, so does the latency associated with memory access due to the increased memory bandwidth [22]. In order to support a

large number of processors (e.g., hundreds), memory must be distributed among the processors in order to decrease the latency of each processor's memory access [17], [22]. The result is the loosely-coupled distributed shared memory (DSM) model, where each processor has both direct access to local memory attached to its own core, as well as indirect access to non-local memory attached to another processor's core [17]. This type of memory model is common among high-performance scientific computer clusters [21]. Distributing memory among processors reduces the latency of local memory access, since each processor's local memory is located near its own core [17]. However, accessing remote memory associated with another processor's core involves communication between the two processors, which is much more difficult to coordinate [17], [21]. Programmers must expend significant effort in order to implement efficient communication between processors and thus take full advantage of the increased memory bandwidth [21]. An efficient inter-processor communication mechanism enables the use of hundreds of processors to perform computation in parallel. Ideally, processors will only exchange communication when absolutely necessary, thus resulting in significant speedup for certain problems [22].

B. PARALLEL PROGRAMMING, THE MESSAGE PASSING INTERFACE, AND THE JULIA LANGUAGE

There are many different communication protocols and programming languages available to implement parallel programming across multiple processors [17], [21], [22]. This thesis will implement parallel quantum computer simulation in the Julia language using the Message Passing Interface (MPI).

1. Parallel Programming with MPI

One of the most commonly used methods of implementing parallel program execution is a communication protocol known as MPI [21]. In the MPI paradigm, programs are split among multiple processors. Each processor executing an MPI program typically only executes one thread or process. The idea is to partition a large computational task into many smaller sub-tasks, which are each assigned to different processors for independent execution [21].

The processors execute the program with a distributed memory model, meaning each processor has exclusive access to its own local memory [21]. However, these processors can share data in their local memory with other processors by engaging in message-passing [21]. Two processors, *proc0* and *proc1* engage in message-passing as follows: *proc0* "packs" a message in a buffer and sends that message to *proc1*, which receives the message in a buffer in its own local memory. Messages can be sent either from one individual processor to another (known as point-to-point communication), or broadcast to all other processors [21]. MPI also operates under the assumption that every processor can exchange messages with every other processor [21].

The format for sending these messages is given in various MPI communication libraries [21], [24]. This thesis will use the OpenMP implementation of MPI, which has language bindings available for C, C++, as well as Fortran [24]. Official documentation of the OpenMP MPI library is available in [24]. This thesis will focus on the specification known as MPI-1, which uses a static process model. The static process model means that the number of processors running a program is fixed when the program starts and cannot be changed [21]. In another specification, MPI-2, processes can be dynamically started and stopped during a program's execution [21].

Theoretically, each processor in an MPI program is capable of executing a different program, but in practice most MPI programs involve each processor running the same program on their own locally stored data (or at least different parts of the same program) [21]. An MPI program running on P processors is said to have a *size* of P, and the processors are indexed arbitrarily from 0 to $P-1$ [24]. This index is known as the processor's *rank*, which serves to uniquely identify it from the other processors [24]. It is important to designate only a single processor (e.g., processor 0) to conduct input/output (I/O), otherwise, all processors will be executing I/O separately and there is no way to control their order.

There are two main categories of communication between processors in MPI. The first is *blocking* communication, where communication completes before control is returned to the calling process and the next line of code is executed [21], [24]. At the point control is returned, all the data in a processor's send and/or receive buffer is valid

and can be used in computation. In contrast to blocking communication is *non-blocking* communication [21]. In non-blocking communication, control is returned immediately to the calling process and the next line of code is executed; the programmer must separately ensure that the results in the processor's send/receive buffers is valid and can be written over [21], [24].

It is imperative to coordinate blocking and non-blocking communication when programming in MPI to avoid both deadlock and race conditions. Deadlock results when two or more processors are in a mutual waiting condition that cannot be resolved and most commonly occurs with blocking communication [21]. If non-blocking communication is incorrectly implemented, a race condition can develop where a shared value is operated on by multiple processors simultaneously with unpredictable results [21]. Improperly coordinated communication among processors will thus result in a program that does not terminate, or produces incorrect results; either case can be very difficult to debug, since the results may vary each time the code is executed [21], [24].

2. Julia Programming Language

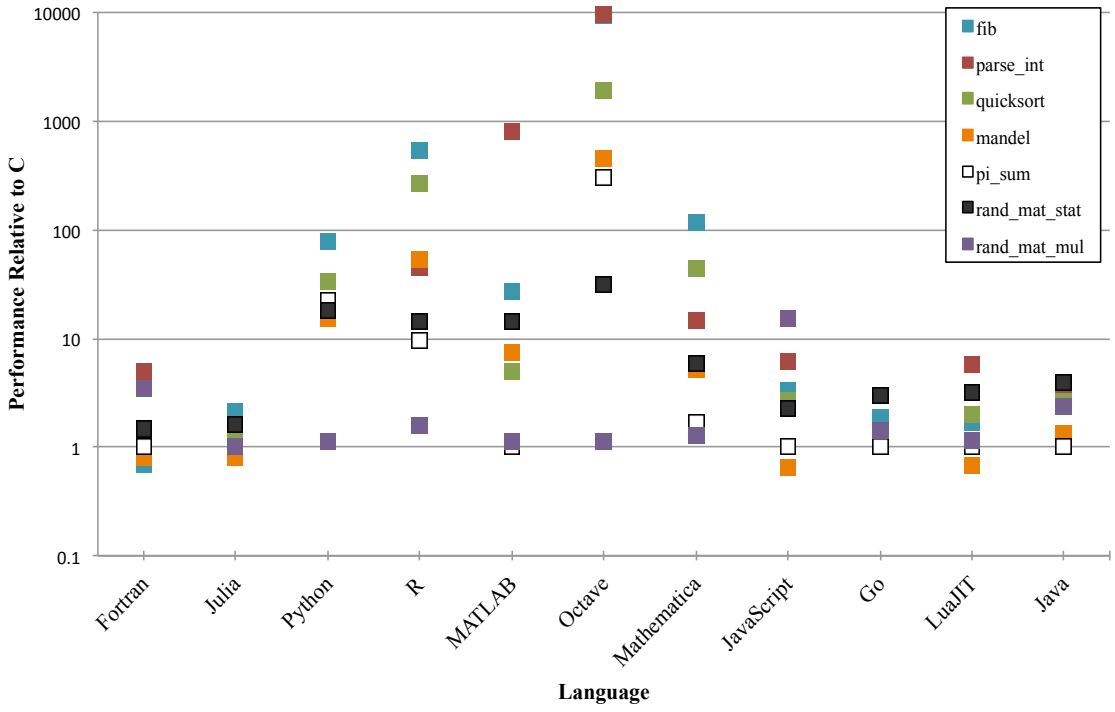
All programs in this thesis are written in the Julia programming language, which is a high-level language designed by researchers at MIT for high-performance technical computing [25]. Syntactically, Julia is similar to other technical computing languages such as MATLAB or Wolfram Mathematica. The C++ MPI library has also been wrapped for native use in the Julia language. Julia code is compiled using a low-level virtual machine just-in-time compiler, meaning that compilation of code is performed during the execution of the program, as opposed to prior to execution. The just-in-time compilation method allows the performance of Julia based code to approach the performance of C code in many situations. The performance of Julia code is compared with that of other languages relative to C across various benchmarks in Figure 11 from [25].

In Figure 11, C performance is equal to 1.0 with a smaller number indicating speedup over C performance. Julia code is only able to outperform C on several benchmarks, but overall Julia can be seen to have the best performance of any of the

other high-level languages. Full details of this experiment including source code for each benchmark are available at [25]. Ultimately, Julia is the language of choice for this thesis because of its high level of abstraction, its easy compatibility with MPI, as well as high performance across various benchmarks.

One final note about Julia is that it uses 1-based indexing (as opposed to the 0-based indexing in languages such as C or Python). Therefore, converting arrays back and forth from 0- to 1- based indexing will be required in a lot of the code for this thesis.

Figure 11 Performance of Various Languages Relative to C across Several Benchmarks. Adapted from [25].



C. PARALLEL ALGORITHM PERFORMANCE ANALYSIS: AMDAHL'S LAW AND GUSTAFSON'S LAW

In 1967, computer scientist Gene Amdahl famously developed a formula to quantify the theoretical speedup obtainable through improving some portion of a computer program [17], [21], [22], [26]. In the general case of Amdahl's law, speedup is

defined as the ratio of the execution times of the computer program with and without the performance improvement [17]. Speedup reveals how much faster (or slower, in the case of a performance degradation) a computer program will run after the performance enhancement [17]. From [22], Amdahl's law to describe the speedup of an entire task can be written as a function of *s*, which is the speedup in latency of the part of the computer program that benefits from the performance enhancement, as shown in Equation (36).

$$Speedup_{overall}(s) = \frac{Execution\ time_{old}}{Execution\ time_{new}} = \frac{1}{(1-f)+\frac{f}{s}} \tag{36}$$

In Equation (36), *f* is the fraction of the execution time of the whole task (before the performance gain) that is spent running the part of the task that is receiving the performance improvement [22]. One of the main implications of Amdahl's law in sequential code is to focus on improving routines that comprise a large portion of a program's execution. This effect can be seen in Equation (37); speedup in latency is clearly minimal for small values of *f* [26].

Amdahl's law was originally envisioned for sequential processing on a single core, but has also been adapted to analyze the theoretical speedup obtained in parallel code [26]. The theoretical speedup in latency for parallelizable code, S_p, is given as a function of the number of processors, *p*, and a fixed size of the problem to be solved, *n*, in Equation (37).

$$S_p(n,p) = \frac{T^*(n)}{T(n,p)} = \frac{T^*(n)}{f \cdot T(n) + \left(\frac{1-f}{p}\right) \cdot T(n)} = \frac{1}{f + \left(\frac{1-f}{p}\right)} \leq \frac{1}{f} \tag{37}$$

In Equation (37), $T^*(n)$ is the latency of the sequentially executed program on problem size *n*, $T(n,p)$ is the latency of the parallelized algorithm for a problem of size *n* running on *p* processors, and *f* is the fraction of inherently sequential computation [21]. Note that this version of Amdahl's law assumes that the parallel portion of the algorithm

used has been "perfectly parallelized" into precisely equal sub-tasks for each processor [21], [22], [27].

The most important implication of the parallel core version of Amdahl's law is that the fraction of code executed sequentially, *f*, serves as a theoretical upper bound for the obtainable speedup, regardless of the number of processors used [21]. For example, consider a program that must be executed 50% sequentially. Even with an infinite amount of processors, the maximum achievable speedup is $S_p(n) = 1/0.5 = 2$.

As previously mentioned, Amdahl's law for parallel cores as given in Equation (37) only determines speedup for problems of a fixed size. Therefore, it is only possible to study how increasing the number of processors contributes to the speedup of a fixed-size problem; this is known as the *strong scalability* of the problem [17], [27]. Clearly, from Equation (37), even so-called "embarrassingly parallel" problems that are easily adapted for parallel execution will begin to lose appreciable speedup as the number of processors *p* increases beyond a certain point [21].

Amdahl's law does not allow for analysis of a problem's *weak scalability,* which assumes a fixed problem size per processor [17], [27]. For many problems it is more important to study the weak scaling by varying the size of the problem while simultaneously varying the number of processors available [27]. In 1988, John Gustafson adapted Amdahl's law to analyze execution times as the problem size is allowed to change [28]. What has become known as Gustafson's law to compute the speedup of a parallel algorithm, S_p, running on a problem of size *n* across *p* processors is shown in Equation (38). Here, $s(n,p)$ is the fraction of time spent performing sequential operations [28].

$$S_p(n,p) = p + (1-p)\cdot s(n,p) \tag{38}$$

For any problem that can be adequately parallelized across multiple processors, Amdahl's law can be used to study the strong scaling by increasing the number of processors for a fixed problem size [21], [26]. At the same time, Gustafson's law can be used to see how the problem scales as the size of the problem increases linearly with the number of processes available to perform computation [27], [28]. Both of these laws are

used together to determine the most efficient way to utilize computational resources in the multicore environment [17], [27].

D. APPLICATION TO QUANTUM COMPUTER SIMULATION

Modeling real-life phenomenon using computers can often require a very large amount of computational resources, including both CPU time and memory—this is certainly true in the case of simulating quantum systems. As described in Chapter 1, in order to simulate a quantum register consisting of N qubits, a QCS must maintain a state vector of length 2^N [8]. Furthermore, applying a single quantum gate to one qubit in the quantum register requires multiplying the state vector by a dense $2^N x 2^N$ matrix. The simulation of a quantum system will thus involve both memory and time bottlenecks as the size of the system increases. Even with a state-of-the-art processor, it will be impossible to simulate a quantum circuit with a sufficient amount of qubits due to the limits on the processor's available memory. The exponential nature of the growth of the matrices involved with simulating the system simply result in matrices that are too large.

In order to more efficiently simulate a large quantum system, the work can be spread among numerous processors. The next chapter details the implementation of a QCS using MPI in order to simulate quantum systems that are too large to simulate on a single processor.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DEVELOPMENT OF A PARALLEL QUANTUM COMPUTER SIMULATOR

A. OPTIMIZING A SEQUENTIAL QCS

Before implementing a QCS that is executed in parallel across multiple nodes, we optimized the existing simulator to incorporate a matrix-free, in-place algorithm that vastly improved both time and memory performance of the sequential code. The optimized sequential QCS takes as input text files which carry descriptions of quantum circuits in what we call Quantum Description Language, adapted from the sequential, full-matrix QCS implemented in Java from [29].

1. Quantum Description Language

Quantum Description Language (QDL) is an intuitive text-based method of describing quantum circuits that is easily read by humans as well as easily parsed by computers using any standard high-level programming language. The idea behind QDL is to create a simple method of encoding quantum circuits in any standard text editor that can be parsed and adapted for use on any QCS without having to hard-code matrices and state vectors by hand in Julia or other languages. For comparison purposes, QDL was adapted with backwards compatibility in mind so that most circuits that can be simulated on the QCS from this thesis can also be simulated on the Java-based QCS from [29]. The only exceptions are circuits with a sufficiently large number of qubits, whose simulation is possible on this Julia QCS but not the Java QCS from [29] because of lack of memory (due to the full-matrix simulation implementation). An example QDL file is depicted below, which represents the quantum circuit for Deutsch's algorithm, as shown in Figure 4.

```
Define N 2
Define U 4
0 1 0 0
1 0 0 0
0 0 1 0
0 0 0 1
Define Phi0
```

```
0 1
Define Transform1
H I
Define Transform2
I H
Define Transform3
U
Define Transform4
H I
Define Transform5
M I
```

The QDL file begins by defining an *N* value, which is the number of qubits in the simulated quantum system (in this case, two). An *N* value is required in every QDL file. Next, the unitary transformation matrix *U* is defined; in this case the matrix *U* is the unitary transform matrix given in Equation (22), which corresponds to the black-box function depicted in Figure 3 and Figure 4. A transformation matrix is not required for circuits that do not utilize a transformation matrix. If no transformation matrix is required for a circuit, these lines are omitted, and *Phi0* is defined directly after *N*. If the transformation matrix is required, the *Phi0* value is defined immediately after the encoding of the unitary matrix *U*. A *Phi0* definition is required for every quantum circuit, and the number of bits defined must correspond to the number *N*. Additionally, only the classical binary values 0 and 1 are allowed in this field. *Phi0* corresponds to the initial classical state of the quantum register, $|\phi_0\rangle$, which in this case is $|0\rangle$ for the upper qubit and $|1\rangle$ for the lower qubit. After declaring the initial state of the quantum register, *Transform1* - *Transform5* are defined. These correspond to the unitary transform matrices for each step of the quantum circuit. In this case, *Transform1* corresponds to the matrix $H \otimes I$, *Transform2* to $I \otimes H$, *Transform3* to the unitary black-box matrix *U*, *Transform4* again to $H \otimes I$, and *Transform5* to a probabilistic measurement of the upper qubit.

Transforms using all gates described in Section I.B.1 are possible and should be fairly intuitive for the other single-qubit gates. For instance `Z I` in QDL corresponds to the transformation matrix $Z \otimes I$, `I Y I I` corresponds to $I \otimes Y \otimes I \otimes I$, etc.

Multiple-qubit gates can also be encoded into the QCS using QDL. For a *CNOT* gate, the words "Control" and "Target" are specified in the index of their respective qubits. An example QDL encoding of a transform involving a *CNOT* gate is shown below:

```
Define Transform1
I I I Control I Target I I I
```

The above QDL transform corresponds to a nine-qubit register with a $CNOT_{5,3}$ gate, meaning the third qubit is the control qubit and the fifth qubit is the target qubit (recall qubits are ordered with 0-based indexing). Perhaps not as intuitive is the implementation of the Toffoli gate in QDL, as depicted below:

```
Define Transform1
UT1 I I UT2 I I UT3 I I
```

This QDL transform describes another nine-qubit register with a $T_{0,3,6}$ gate. In QDL, *UT1* corresponds to the target qubit, *UT2* corresponds to the first control qubit, and *UT3* corresponds to the second control qubit. "UT" in this context stands for "Upside-down Toffoli" gate; this is a carryover from the simulator's implementation from [29] in order to ensure backwards compatibility. The Julia code to parse an entire QDL file for simulation is somewhat lengthy but trivial to implement. As such, it has been redacted from this text for brevity but can be found in the code for the full sequential simulator in the supplementary code repository for this thesis. Moreover, the parser is only intended for academic use and thus is not very robust in that it assumes a well-formed QDL file. Any QDL files with syntactical errors or impossible quantum circuits will either produce unpredictable results or cause the simulator program to crash.

2. Simulating Quantum Gates In-Place

After parsing the QDL file, the first step of simulation is to generate a state vector representing the initial classical value of the quantum system. In the Julia code for the

simulator (located in this thesis' supplemental code repository), this state vector is the buffer *a*. For a quantum system of *N* qubits, the vector *a* is clearly always of length 2^N. The basis states and corresponding complex amplitudes of the state vector *a* representing an arbitrary quantum register can be represented in binary, as shown in Equation (39).

$$\begin{aligned} |\psi\rangle = a(0_{(0)}0_{(1)}\ldots 0_{(N)})\cdot|0_{(0)}0_{(1)}\ldots 0_{(N)}\rangle + \ldots \\ \ldots + a(0_{(0)}0_{(1)}\ldots 1_{(N)})\cdot|0_{(0)}0_{(1)}\ldots 1_{(N)}\rangle + \ldots + a(1_{(0)}1_{(1)}\ldots 1_{(N)})\cdot|1_{(0)}1_{(1)}\ldots 1_{(N)}\rangle \end{aligned} \tag{39}$$

Moreover, since the quantum register begins each quantum circuit in a classical state, it of course must represent a single value with 100% probability. Therefore, the state vector can be generated easily by allocating a vector of zeros of length 2^N with a single 1 value at index *k*+1. Adding 1 to the value *k* is required since Julia uses 1-based indexing instead of 0-based indexing. The value of the index *k*, in binary format, is simply equal to the initial value in the quantum register, *Phi0*. The following Julia function generates the initial state vector *a*.

```
function generateA(phi, N)
    A = zeros(2^N)
    A[phi+1] = 1
    return A
end
```

After generating the initial classical state vector *a*, quantum gates are applied to the quantum register for every transform indicated in the QDL file. Generating a 2^Nx2^N unitary matrix for each transform in a quantum circuit is incredibly expensive in terms of both CPU time and memory. For example, over 17 GB of memory is required just to store the unitary transform matrix for a 17 qubit simulation. Furthermore, the size of this matrix doubles for every qubit added to the simulated register as discussed in Section I.E. In order to greatly reduce the memory and time required to perform these unitary transform operations, they can be done in-place using a matrix-free algorithm.

Two buffers, *a* and *a_prime*, are created in order to execute the in-place algorithm. Each buffer stores a column vector of length 2^N where each element represents a complex amplitude of each basis state of the quantum system. The vectors *a* and

a_prime represent the state of the quantum system before and after (respectively) any particular unitary transform. After the transform operation has been performed, the pointers for the two buffers are swapped. As an example, Equation (40) depicts the full-matrix relationship between *a* and *a_prime* for an *N*-qubit quantum system in which an *H*-gate is applied to the kth qubit.

$$a_prime = \left[I_{(0)} \otimes I_{(1)} \otimes \ldots \otimes H_{(k)} \otimes \ldots \otimes I_{(N-2)} \otimes I_{(N-1)} \right] \bullet a \tag{40}$$

As previously mentioned, it is possible to perform this operation without generating the series of Kronecker products that results in the full 2^Nx2^N matrix. As noted in [8], applying the Hadamard transform from Equation (13) to the kth qubit of the quantum register represented by the vector *a* is equivalent to transforming the amplitudes of *a* to those of *a_prime*, as shown in Equation (41). Here, the asterisks indicate that the values in the corresponding indices are identical.

$$\begin{aligned} a_prime(*_{\ldots}*0_{(k)}*_{\ldots}*) &= \frac{1}{\sqrt{2}}\left[a(*_{\ldots}*0_{(k)}*_{\ldots}*) + a(*_{\ldots}*1_{k)}*_{\ldots}*) \right] \\ a_prime(*_{\ldots}*1_{(k)}*_{\ldots}*) &= \frac{1}{\sqrt{2}}\left[a(*_{\ldots}*0_{(k)}*_{\ldots}*) - a(*_{\ldots}*1_{(k)}*_{\ldots}*) \right] \end{aligned} \tag{41}$$

As a concrete numerical example, consider a three-qubit system where all three qubits have initial values of 0 and an *H*-gate is applied to the most significant qubit. In QDL, this system is encoded as

```
Define N 3
Define Phi0
0 0 0
Define Transform1
H I I
```

The initial state vector, *a*, for this quantum register is given in Equation (42), which corresponds to a classical value of 0.

$$a = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T \tag{42}$$

To apply an *H*-gate to qubit zero naively, the state vector is simply multiplied by the 8x8 matrix $H \otimes I \otimes I$. The result of performing this matrix multiplication is shown in Equation (43).

$$a_prime = \left[H \otimes I \otimes I\right] \cdot a = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 \end{bmatrix}^T \tag{43}$$

Alternatively, the in-place algorithm from Equation (41) can be used to determine the new state, *a_prime* without forming the full $H \otimes I \otimes I$ matrix. This is done by iterating through each of the 2^N indices of the vector *a_prime* and determining the value at each index based on Equation (13). All values of *a* are known prior to applying the transformation matrix, so they can be simply substituted in algebraically. This is done in Equation (44), where the bold values indicate the qubit to which the *H*-gate is being applied (this value is referred to as *boldbit* in the Julia code).

$$\begin{aligned}
a_prime(0_{10}) &= a_prime(\mathbf{0}00_2) = \frac{1}{\sqrt{2}} \cdot \left[a(000_2) + a(100_2)\right] = \frac{1}{\sqrt{2}} \cdot [1+0] = \frac{1}{\sqrt{2}} \\
a_prime(1_{10}) &= a_prime(\mathbf{0}01_2) = \frac{1}{\sqrt{2}} \cdot \left[a(001_2) + a(101_2)\right] = \frac{1}{\sqrt{2}} \cdot [0+0] = 0 \\
a_prime(2_{10}) &= a_prime(\mathbf{0}10_2) = \frac{1}{\sqrt{2}} \cdot \left[a(010_2) + a(110_2)\right] = \frac{1}{\sqrt{2}} \cdot [0+0] = 0 \\
a_prime(3_{10}) &= a_prime(\mathbf{0}11_2) = \frac{1}{\sqrt{2}} \cdot \left[a(011_2) + a(111_2)\right] = \frac{1}{\sqrt{2}} \cdot [0+0] = 0 \\
a_prime(4_{10}) &= a_prime(\mathbf{1}00_2) = \frac{1}{\sqrt{2}} \cdot \left[a(000_2) - a(100_2)\right] = \frac{1}{\sqrt{2}} \cdot [1-0] = \frac{1}{\sqrt{2}} \\
a_prime(5_{10}) &= a_prime(\mathbf{1}01_2) = \frac{1}{\sqrt{2}} \cdot \left[a(001_2) - a(101_2)\right] = \frac{1}{\sqrt{2}} \cdot [0-0] = 0 \\
a_prime(6_{10}) &= a_prime(\mathbf{1}10_2) = \frac{1}{\sqrt{2}} \cdot \left[a(010_2) - a(110_2)\right] = \frac{1}{\sqrt{2}} \cdot [0-0] = 0 \\
a_prime(7_{10}) &= a_prime(\mathbf{1}11_2) = \frac{1}{\sqrt{2}} \cdot \left[a(011_2) - a(111_2)\right] = \frac{1}{\sqrt{2}} \cdot [0-0] = 0
\end{aligned} \tag{44}$$

Julia code to perform this iteration is shown below. First, a bitmask value, *mask,* is created in order to identify the index of the qubit that is undergoing the Hadamard

transform. The value *mask* is determined by taking the bitwise *XOR* of the value 2^N-1 ($1_{(0)}1_{(1)}...1_{(N-2)}1_{(N-1)}$ in binary) with the temporary value *tempIndex.* The value of *tempIndex* is equal to (1<<(*N*-1-*h*)), where *N* is the number of qubits, *h* is the index of the qubit to which the *H*-gate is applied, and "<<" indicates the logical left bitshift operation . Table 4 depicts the mask value in both binary and decimal format for all three possible *H*-gate locations in the three-qubit example system.

```
scalar = 1/sqrt(2)
tempIndex = (1<<(N-1-h))
mask = (2^N - 1) $ tempIndex
for i = 0:(2^N)-1
    boldbit = (i >> (N - 1 - h)) & 1
    if boldbit == 0
        a0 = (i & mask)
        a1 = (i & mask | tempIndex)
        a_prime[i+1] = (a[a0+1] + a[a1+1])*scalar
    elseif boldbit == 1
        a0 = (i & mask)
        a1 = (i & mask | tempIndex)
        a_prime[i+1] = (a[a0+1] - a[a1+1])*scalar
    end
end
```

Table 4 Possible Mask Values for H-Gates on a Three-Qubit Register

h (Index of H-Gate)	*tempIndex* Value	Resultant Mask
0	4_{10} (100_2)	3_{10} (011_2)
1	2_{10} (010_2)	5_{10} (101_2)
2	1_{10} (001_2)	6_{10} (110_2)

The algorithm next iterates through each index, *i,* of *a_prime* and populates each associated value *a_prime*(*i*). For each iteration, the *boldbit* value is determined based on

the binary representation of each index from Equation (44); this value is always either zero or one. Two intermediate values, *a0* and *a1*, are then determined.

The first intermediate value, *a0*, is computed by taking the bitwise *AND* of the *mask* value and the binary representation of the index *i*. This is made possible because the value of the mask in a register of length *N* will be a string of *N* 1's where the hth 1 has been replaced with a 0. For any binary value *X*, performing the operation $X \wedge 1$ will always return the original value *X*, while the operation $X \wedge 0$ will always return the value 0 (here "$\wedge$" indicates the logical *AND* operation).

The value *a1* uses this same result of the previous bitwise *AND* operation from *a0.* The bitwise *OR* operation is performed on the value *a0* and the previously computed *tempIndex* value. A simple analysis by hand will show that the results of these bitwise operations will indeed produce the values shown in Equation (44)

If the *boldbit* value is 0, the value of *a_prime* at index *i* is equal to the scalar multiplied by the sum of the values of *a* at indices *a0* and *a1*. In other words, $a_prime(i+1) = \frac{1}{\sqrt{2}}\left(a(a0) + a(a1)\right)$; again, 1 must be added to the index *i* because Julia uses 1-based indexing. If the *boldbit* value is 1, however, the value of *a_prime* at index *i* is equal to the scalar multiplied by the difference between *a*(*a0*) and *a*(*a1*), as given in Equation (41).

After the iteration through all indices of *a_prime* is complete, the pointers for buffers *a* and *a_prime* are swapped. Thus, the state vector *a_prime* becomes the initial state vector *a* for the next unitary transformation.

As another numerical example, consider the same three-qubit system as above but with the most significant qubit initialized to 1. A QDL encoding of this circuit is as follows:

```
Define N 3
Define Phi0
1 0 0
Define Transform1
```

```
H I I
```

The initial state vector of the quantum register is now in a classical state of 4, as indicated in Equation (45).

$$a = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}^T \tag{45}$$

The same bitwise operations are shown in Equation (46) in order to determine each value of *a_prime*.

$$\begin{aligned}
a_prime(0_{10}) &= a_prime(\mathbf{0}00_2) = \frac{1}{\sqrt{2}} \cdot \left[a(000_2) + a(100_2)\right] = \frac{1}{\sqrt{2}} \cdot [0+1] = \frac{1}{\sqrt{2}} \\
a_prime(1_{10}) &= a_prime(\mathbf{0}01_2) = \frac{1}{\sqrt{2}} \cdot \left[a(001_2) + a(101_2)\right] = \frac{1}{\sqrt{2}} \cdot [0+0] = 0 \\
a_prime(2_{10}) &= a_prime(\mathbf{0}10_2) = \frac{1}{\sqrt{2}} \cdot \left[a(010_2) + a(110_2)\right] = \frac{1}{\sqrt{2}} \cdot [0+0] = 0 \\
a_prime(3_{10}) &= a_prime(\mathbf{0}11_2) = \frac{1}{\sqrt{2}} \cdot \left[a(011_2) + a(111_2)\right] = \frac{1}{\sqrt{2}} \cdot [0+0] = 0 \\
a_prime(4_{10}) &= a_prime(\mathbf{1}00_2) = \frac{1}{\sqrt{2}} \cdot \left[a(000_2) - a(100_2)\right] = \frac{1}{\sqrt{2}} \cdot [0-1] = -\left(\frac{1}{\sqrt{2}}\right) \\
a_prime(5_{10}) &= a_prime(\mathbf{1}01_2) = \frac{1}{\sqrt{2}} \cdot \left[a(001_2) - a(101_2)\right] = \frac{1}{\sqrt{2}} \cdot [0-0] = 0 \\
a_prime(6_{10}) &= a_prime(\mathbf{1}10_2) = \frac{1}{\sqrt{2}} \cdot \left[a(010_2) - a(110_2)\right] = \frac{1}{\sqrt{2}} \cdot [0-0] = 0 \\
a_prime(7_{10}) &= a_prime(\mathbf{1}11_2) = \frac{1}{\sqrt{2}} \cdot \left[a(011_2) - a(111_2)\right] = \frac{1}{\sqrt{2}} \cdot [0-0] = 0
\end{aligned} \tag{46}$$

Note that the final results of this operation, as shown in Equation (47), are identical to the previous example except with a sign flip on the 4th element of *a_prime* . In other words, when both example circuits are complete, there is exactly a 50% probability of measuring the register in a classical state of 0, or a classical state of 4.

$$a_prime = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & 0 & -\left(\frac{1}{\sqrt{2}}\right) & 0 & 0 & 0 \end{bmatrix}^T \tag{47}$$

Other single-qubit gates can be applied using the same methodology, as shown in Equations (48)–(50), adapted from [8]. The full Julia code for implementing other single-qubit gates using the in-place algorithm is included with the full code of the simulator in the supplemental code repository.

***X* - gate transformation :**

$$a_prime(*_{\ldots}*0_{(k)}*_{\ldots}*) = \frac{1}{\sqrt{2}}\left[a(*_{\ldots}*0_{(k)}*_{\ldots}*) + i \cdot a(*_{\ldots}*1_{k)}*_{\ldots}*)\right] \tag{48}$$

$$a_prime(*_{\ldots}*1_{(k)}*_{\ldots}*) = \frac{1}{\sqrt{2}}\left[a(*_{\ldots}*1_{(k)}*_{\ldots}*) - i \cdot a(*_{\ldots}*0_{(k)}*_{\ldots}*)\right]$$

***Y* - gate transformation :**

$$a_prime(*_{\ldots}*0_{(k)}*_{\ldots}*) = \frac{1}{\sqrt{2}}\left[a(*_{\ldots}*0_{(k)}*_{\ldots}*) + a(*_{\ldots}*1_{k)}*_{\ldots}*)\right] \tag{49}$$

$$a_prime(*_{\ldots}*1_{(k)}*_{\ldots}*) = \frac{1}{\sqrt{2}}\left[a(*_{\ldots}*1_{(k)}*_{\ldots}*) - a(*_{\ldots}*0_{(k)}*_{\ldots}*)\right]$$

***R*(ϕ)-gate transformation :**

$$a'(*_{\ldots}*0_{(k)}*_{\ldots}*) = a(*_{\ldots}*0_{(k)}*_{\ldots}*) \tag{50}$$

$$a'(*_{\ldots}*1_{(k)}*_{\ldots}*) = e^{i\phi}a(*_{\ldots}*1_{(k)}*_{\ldots}*)$$

The Julia QCS implements the in-place algorithm for multiple-qubit gates using the same methodology as for single-qubit gates. The algorithm iterates through the indices of *a_prime* and computes a value for each index based on both the gate locations and the values in the current *a* buffer. Julia code for the implementation of the *CNOT* gate is included below, where *C* and *T* are the indices of the control and target qubits respectively:

```
for i = 0:((2^N)-1)
    controlQubit = (i >> (N - 1 - C)) & 1
    targetQubit = (i >> (N - 1 - T)) & 1

    if targetQubit == 0
      antiTarget = 1
    elseif targetQubit == 1
```

```
        antiTarget = 0
      end
      if controlQubit == 1
        mask = (2^N-1) $ (1 << (N-1-T))
        targetShift = (antiTarget << (N-1-T))
        index = ((i & mask) | targetShift)+1
        a_prime[i+1] = a[index]
      end
    end
```

For each iteration *i*, the binary representations of the indices of both the control and target qubits are determined using bitshift operators. An intermediate value, called *antiTarget* then is created and defined as the opposite of the classical value of the target qubit. Next, a bitmask is created using the same bitwise *XOR* operation that was used to create the mask for the application of the *H*-gate.

Finally, an *index* value is calculated as follows. First the bitwise *AND* of the bitmask and the iterator *i* is determined, and this value undergoes a bitwise *OR* with the value *targetShift.* The intermediate *targetShift* value is computed by logically left-shifting the *antiTarget* value by (N-1-T) bits. The rationale behind the left shift operation is that for any binary value X, $X \vee 1 = 1$ and $X \vee 0 = X$, (where "$\vee$" indicates the logical *OR* operation) . The possible values of the results of this logical left shift operations are shown for all values of *antiTarget* and *T* in a three-qubit system in Table 5. Finally, 1 is again added to this index due to Julia's 1-based indexing.

Table 5 Values after the Left-Shift Operation in the *CNOT* Gate for *N*=3.

antiTarget	*T* (Index of Target Qubit)	*targetShift* Value
0	0	0_{10} (000_2)
0	1	0_{10} (000_2)
0	2	0_{10} (000_2)
1	0	4_{10} (100_2)
1	1	2_{10} (010_2)
1	2	1_{10} (001_2)

The Julia implementation of the Toffoli gate is unsurprisingly very similar to that of the *CNOT* gate with the addition of a second control qubit. Julia code to iterate through the indices of *a_prime* and update the values after application of a Toffoli gate is shown below:

```
for i = 0:((2^N)-1)
    controlQubit1 = (i >> (N - 1 - C1)) & 1
    controlQubit2 = (i >> (N - 1 - C2)) & 1
    targetQubit = (i >> (N - 1 - T)) & 1

    if targetQubit == 0
      antiTarget = 1
    elseif targetQubit == 1
      antiTarget = 0
    end
    if (controlQubit1 == 1) && (controlQubit2 == 1)
      mask = (2^N-1) $ (1 << (N-1-T))
      targetShift = (antiTarget << (N-1-T))
      index = ((i & mask) | targetShift)+1
      a_prime[i+1] = a[index]
    end
end
```

One small but noteworthy difference in the implementation of our in-place sequential simulator in Julia and the Java simulator from [29] is that our simulator is only able to perform one single-qubit transformation for each line in a QDL file. The Java simulator, however, is able to simulate multiple single-qubit gates in the same line. For example, *H*-gates can be applied to all three qubits in a quantum register in the Java simulator with the following transform:

```
Define Transform1
H H H
```

This transformation is possible using the full-matrix implementation and the simulator simply generates the full 8 x 8 matrix, $H \otimes H \otimes H$. Using the in-place simulation algorithm, however, this is not a valid transform. The three *H*-gates must be applied sequentially, and the buffers swapped after each transformation. A valid QDL command for the in-place simulator to perform the same operation is

```
Define Transform1
H I I

Define Transform2
I H I

Define Transform3
I I H
```

We refer to this behavior as "cascading" the single-qubit gate, since the same transformation is performed to each qubit in sequential order. Parsing the QDL file to apply the unitary transforms is done in constant time, so overall there is little to no performance degradation involved with performing three transformations instead of one. It is, however, a minor inconvenience to have to create a QDL file with three transforms instead of a single line. We note that it is possible to create a parser that can accept multiple single-qubit gates in a single line and simply automate the cascading behavior, but we leave this for future work.

We conducted thorough testing of our in-place implementation of these quantum gates upon known quantum circuits in order to ensure that our implementation was valid. After having established that our program was performing all simulations correctly, we next compared the performance of the full-matrix implementation to that of the in-place, matrix-free algorithm, with somewhat surprising results.

3. Comparison of Full-Matrix and In-Place Algorithms

In order to compare the performance of the full-matrix and in-place algorithms, we developed two separate Julia scripts to easily test both implementations alongside one another across various platforms. Both scripts simulate the same circuit and are executed from the command line with a user-defined number of qubits, *N*, as a command line argument. The simulated circuit simply places the entire *N*-qubit register into superposition, using the transform $H^{\otimes N}$, and then takes the entire register out of superposition, once again using the $H^{\otimes N}$ transform. We will therefore refer to this test circuit as the $2*\left(H^{\otimes N}\right)$ circuit.

The $2*\left(H^{\otimes N}\right)$ circuit is easily understood conceptually, but not particularly useful since it is both very computation-heavy and it always returns the quantum register back exactly into its original state. However, this makes it a quintessential circuit to compare the full-matrix and in-place algorithms because it is trivial to check for correctness, and the high amount of computation means that any performance differences should be easily discernable. To check for correctness, the final classical state is simply compared to the initial state of the register; if the final classical value is different from the initial classical value, then the circuit has been implemented incorrectly.

In order to simulate the $2*\left(H^{\otimes N}\right)$ circuit using the naïve, full-matrix algorithm, we developed a function, *recursive_kron(A, B, i)*, that makes use of Julia's built-in Kronecker product. The built-in Julia Kronecker product function is assumed to be implemented optimally, or at least near-optimally, by Julia's designers.

The commented function is self-explanatory and the Julia code is contained below:

```
function recursive_kron(A, B, i)
    #Function recursively performs (i-1) Kronecker products
              of A and B
    #Inputs:   A and B, same dimension matrices
    #          i, the number of desired Kronecker products
  #Base case. Return the Kronecker product of A and B
  if i == 2
      answer = kron(A, B)
      return answer
  #Recursive case.
  #Kronecker product of A and B, decrement by i.
  else
      B = kron(A, B)
      recursive_kron(A,  B,  i-1)
  end
end
```

The file *naïve_hadamard.jl* uses this *recursive_kron* function to generate the unitary $H^{\otimes N}$ matrix and multiply it by the initial state vector *a* twice. The entire file is contained within the supplemental code, and is called from the command prompt with the following command (where *N* is the number of qubits in the simulated register):

```
julia naïve_hadamard.jl [N]
```

Next, we implemented the Julia script contained in the file *inplace_hadamard.jl* (also in the supplemental code repository). This file uses the in-place algorithm from Section 2 to simulate an identical $2*\left(H^{\otimes N}\right)$ circuit. After running this script, the final state should again be identical to the initial state. The in-place version of this *N*-qubit circuit is called using the following bash command:

```
julia inplace_hadamard.jl [N]
```

We first compared these two algorithms on a laptop computer for various quantum register sizes. Both time and memory performance can be measured easily using Julia's *@time* macro. The laptop used to conduct this test was a 2015 MacBook Air running Mac OS-X El Capitan (Version 10.11.5). The processor onboard was a 1.4 GHz Intel Dual Core i5 with 4GB memory. Simulation results for this computer are contained in Table 6, which indicates the total time required to simulate the $2*\left(H^{\otimes N}\right)$ circuit for both the full-matrix and in-place algorithms. Additionally, the total number of memory allocations made during the simulation, as well as the combined size of all memory allocated during simulation is displayed in the table.

Clearly, the in-place algorithm significantly outperforms the full-matrix algorithm in terms of both time and total size of allocated memory. It was possible to simulate up to 14 qubits in the $2*\left(H^{\otimes N}\right)$ circuit on this computer using the full-matrix implementation. 2.66 GB of total memory allocation was made during the 14 qubit simulation. However, simulation of 15 qubits was not possible on this computer using the full-matrix algorithm, since more than double the amount of memory is required for each additional qubit. When attempting a circuit of 15 or more qubits, the simulation ran for several minutes before returning the following error:

```
LoadError: OutOfMemoryError()
```

The point at which a personal laptop will run out of memory during simulation will of course vary widely from machine to machine. There are many variables that may influence the maximum size of quantum system that can be simulated on a particular computer including the processor, amount of memory, operating system, and other programs running at the time of simulation. This large variance notwithstanding, the in-place algorithm was able to simulate larger quantum systems notably more quickly and efficiently than the naïve algorithm.

Table 6 Naïve versus In-Place Simulation (1.4 GHz Intel Dual Core i5, 4 GB Memory)

Quantum Register Size (N)	Naïve Implementation		In-Place Implementation	
	Time (seconds)	Number of Memory Allocations (Size)	Time (seconds)	Number of Memory Allocations (Size)
10	0.008	10 (680 KB)	0.057	34.76k (1.647 MB)
12	0.161	26 (170 MB)	0.067	34.76k (1.717 MB)
13	1.27	28 (682 MB)	0.068	34.76k (1.811 MB)
14	12.323	30 (2.66 GB)	0.07	34.76k (1.998 MB)
15	Error	Error	0.075	34.76k (2.373 MB)
16	Error	Error	0.104	34.76k (3.123 MB)

Next, we ran both scripts on a single AMD node of the Hamming high performance computing (HPC) cluster. The AMD node was running a 12 core AMD Opteron 6174 CPU at 2.2 GHz. Unsurprisingly the single AMD node was able to simulate larger systems than the laptop for both the full-matrix and matrix-free simulation algorithms. Table 7 displays the time and memory results of this test for the AMD Opteron 6174 CPU. On the AMD node, out-of-memory errors were observed when using the full-matrix algorithm for any register with more than 16 qubits. The in-place algorithm, however, was able to simulate a 17 qubit circuit in a mere 0.216 seconds, with a total of 4.623 MB in memory allocations. The maximum number of qubits in a $2*\left(H^{\otimes N}\right)$ circuit that the AMD node could simulate using the in-place algorithm was 33 qubits, which took 5,092 seconds with a total of 192 GB allocated.

Table 7 Naïve versus In-Place Simulation Performance (AMD Opteron 6174 CPU)

Quantum Register Size (N)	Naïve Implementation		In-Place Implementation	
	Time (seconds)	Number of Memory Allocations (Size)	Time (seconds)	Number of Memory Allocations (Size)
16	48.8	32 (42.66 GB)	0.157	34.76k (3.123 MB)
17	Error	Error	0.216	34.76k (4.623 MB)
20	Error	Error	0.56	34.76k (25.63 MB)
25	Error	Error	16.3	34.76k (769 MB)
26	Error	Error	34.2	34.76k (1.502 GB)
33	Error	Error	5,092	29.02k (192 GB)
34	Error	Error	Error	Error

Finally, we performed the same test using an Intel node on Hamming, which had even better performance than the AMD node. More specifically, this node was running a 16-core Intel Xeon E-5 2698 v3 CPU with a clock rate of 2.30 GHz. Test results for the Intel Xeon processor are included in Table 8. Using the Intel node, it was possible to simulate a maximum register size of 17 qubits using the full-matrix algorithm. With the in-place algorithm, however, an 18 qubit register was simulated in only 0.0747 seconds, with 7.62 MB of total memory allocations. On the Intel node, the maximum quantum register size possible using the in-place algorithm was 33 again qubits, which also required the same 192 GB of total memory allocation as the AMD node. The Intel node, however, performed the 33 qubit simulation of the $2*\left(H^{\otimes N}\right)$ circuit significantly faster than the AMD node, at 1,349 seconds.

Table 8 Naïve versus In-Place Simulation (Intel Xeon E-5 2698 CPU)

Quantum Register Size (N)	Naïve Implementation		In-Place Implementation	
	Time (seconds)	Number of Memory Allocations (Size)	Time (seconds)	Number of Memory Allocations (Size)
16	19.051	34 (42.66 GB)	0.061	34.76k (3.123 MB)
17	76.752	36 (170.69 GB)	0.065	34.76k (4.623 MB)
18	Error	Error	0.0747	34.76k (7.62 MB)
20	Error	Error	0.127	34.76k (25.63 MB)
25	Error	Error	3.62	34.76k (769 MB)
30	Error	Error	136	29.02k (24.02 GB)
33	Error	Error	1,349	29.02k (192 GB)
34	Error	Error	Error	Error

Ultimately, the results across the board show a pronounced increase in both time and memory efficiency of the in-place algorithm as opposed to generating the full transformation matrix.

B. GOALS OF PARALLELIZING A QCS

Although the in-place simulation algorithm significantly outperforms the full-matrix simulation algorithm in terms of both speed and memory, the total amount of

memory available to a single processor is, of course, limited. The fact that each processor has a finite amount of local memory available puts a hard limit on the maximum number of qubits that can be simulated in a $2*\left(H^{\otimes N}\right)$ circuit on a single processor. In order to increase circuit size beyond this limit without increasing the amount of memory available on an individual processor, the simulation must be performed in parallel across multiple processors. The in-place algorithm can be parallelized across multiple nodes by dividing up the simulation and assigning each processor an equal portion of the simulation to store in its local memory.

Parallelizing the simulation across multiple processors is not without cost, however, since communication between processors is extremely expensive when compared with computation on a single processor [17], [21], [22], [24]. This means that, in addition to the time and memory costs that are present when performing simulation sequentially on a single processor, communication costs must also be factored in when performing simulation in parallel. It may in fact be faster to simulate a smaller quantum system on a single processor than to distribute the quantum system across multiple processors due to high communication costs. Therefore, it will be important to balance the communication, computation, and memory costs when parallelizing the in-place simulation algorithm across multiple processors.

C. IMPLEMENTING THE IN-PLACE ALGORITHM USING MPI

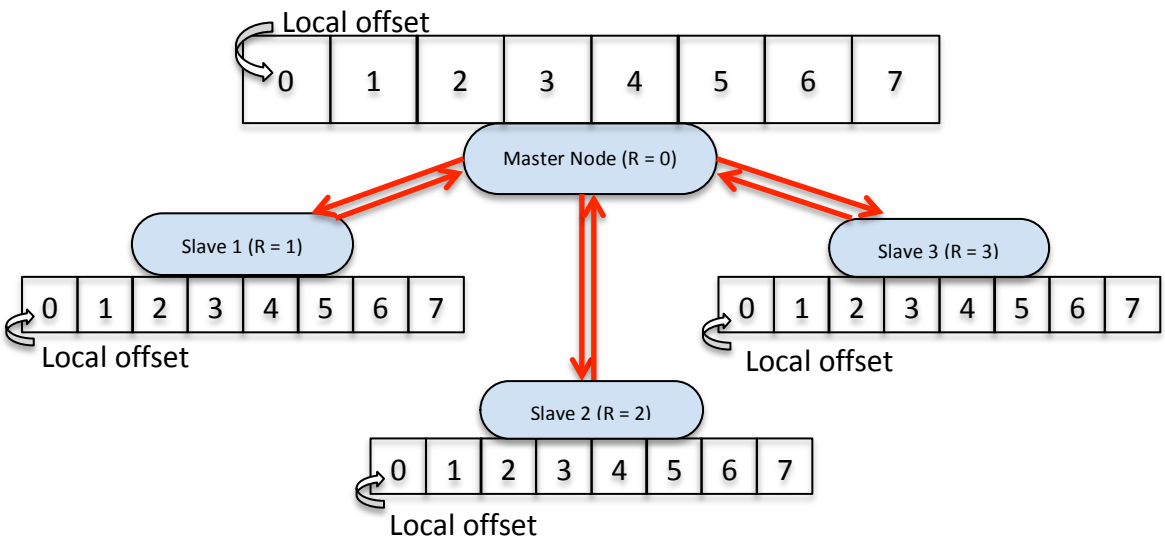
We implemented the matrix-free in-place algorithm across multiple nodes using a master-slave node paradigm. Under the master-slave model, the master node executes all computation and has unidirectional control over all slave nodes, which are used for storage only and perform no computation of their own. When an MPI program to run the simulation is executed across P processors, each processor is given a rank, R. These ranks ordered from $R = 0$ to $R = (P-1)$.

The master node is always defined as the processor with an MPI rank $R = 0$. We assume that the number of processors, P, is a power of two in order to avoid a potentially difficult and (for this project) irrelevant graph partitioning problem. According to the MPI programming model, the master processor, as well as each of the slave processors,

has exclusive access to its own local memory. Therefore, in order to access values on a slave node, the master node must engage in bidirectional MPI communication with that particular slave node.

In order to simulate a quantum register consisting of N qubits sequentially, a single processor must maintain the entire state vector of length 2^N in its own local memory. For parallel execution, however, this state vector is distributed amongst the P processors equally so that each one of the P processors is responsible for storing $2^N/P$ amplitudes of the overall global state vector a. Figure 12 displays an example of the master-slave model for a simulation of $N = 5$ qubits, and $P = 4$ processors. Globally, there are a total of $2^5 = 32$ states in the vector a. Each processor is thus responsible for maintaining $2^5/4 = 8$ states locally and has no direct access to the states of any other processor. MPI communications between processors in Figure 12 are shown as red arrows.

Figure 12 Master-Slave Paradigm Diagram.



The parallel version of the in-place QCS is contained in the *simulator_p.jl* file and can be executed using the following shell command to simulate a user-defined QDL file:

```
mpirun -np [P] julia simulator_s.jl [QDLfilename]
```

Upon execution of this command, P MPI processes are automatically started. The master process (i.e., $R = 0$) begins parsing the indicated QDL file. All slave nodes begin by initializing their storage arrays (local buffers *a* and *a*_prime) to all zeros and then enter an infinite loop where they await commands from the master node.

After parsing the file and determining which transforms to apply, the master node begins performing the transforms in-place just as in the sequential code. The main difference between the sequential and parallel implementations is that the buffers *a* and *a_prime* are now distributed amongst all the nodes. In order to read or write from a certain index, k, of the global *a* buffer, the master node must first determine where k is being stored. The master node can do this by using the integer division and modulo operations.

In the example from Figure 12, each of 4 processors is locally responsible for the storage of 8 elements of the global buffer *a*, which is of length $2^5 = 32$. Each processor has a total of $L = 2^N/\mathrm{P} = 32/4 = 8$ local memory locations. To access the kth element of the global buffer *a*, the master first determines which slave node is responsible for storing index k by integer dividing k by L. In Julia, this is done using the command $div(k, L)$. Next, the master node computes the offset within the local memory of the processor in charge of element k by modulo dividing k by L. The Julia command to perform the modulo operation is $mod(k, l)$.

As a numerical example, if the master node is attempting to read from element 19 of the global buffer *a*, the master node first computes the rank of the processor by computing $R = div(19,8) = 2$. The master node next computes that the offset within the processor of rank 2 is $mod(19,8) = 3$. To read from this position, the master node can now send the appropriate communication to the processor with rank 2.

For another example, consider the master node attempting to read from global position 7. The master computes that the processor with rank $R = div(7,8) = 0$ is assigned to this memory location, and that the local offset is at $mod(7,8) = 7$. Since this memory address is located within the master node's local memory, no MPI communication with other nodes is required to read from or write to this location.

For all transformations from *a* to *a_prime* as described in Section A.2, the master node first computes the rank and offset of the index *i* to be swapped. If the memory location is within the master's local memory, the master node can swap the values locally without any need for MPI communication. If, however, the index *i* is located within some other node's local memory, the master node must communicate with the associated slave node using MPI according to some protocol that prevents deadlocks and/or race conditions.

The communications protocol that we established for this simulator involved both a data and control plane from the master to the slave nodes. The control plane tells the slave which type of activity to perform. The activities that the slave nodes are permitted to execute are *READ*, *STORE*, *SWAP*, and *EXIT*.

The *EXIT* function is self explanatory; slave nodes terminate their infinite loop and print their stored array to an output text file labeled by rank.

If the master node needs to read from a slave node's local memory, it will issue a *READ* control message to the appropriate node. Once a particular slave node has received one of these *READ* control messages, it will temporarily suspend its infinite loop until it has received a data message from the master node. This data message will indicate the particular offset that the master node is attempting to read from. The master node then awaits a data message from the same slave node that will contain the value stored at that particular offset. After sending this value as a data message, the slave node goes back into its infinite loop, and the master node resumes computation.

To perform a write operation to a slave's local memory, the master node first sends a *STORE* message to that particular slave node. The slave node will suspend its infinite loop and await two data messages, the first indicating the offset within the local *a_prime* buffer, and the second indicating the value to store at this offset. After receiving both of these data messages, the slave resumes its infinite loop and the master resumes computation.

When the master node has globally completed a transform from *a* to *a_prime*, it will broadcast a *SWAP* command in the control plane to all slave nodes. Upon receiving a

SWAP command in the control plane, all slave nodes will swap their local *a* and *a_prime* buffers and the next transform will begin. The *SWAP* command is used in conjunction with the *STORE* command so that all buffer swaps occur simultaneously, as coordinated by the master node. This prevents race conditions resulting from values in a slave node's local memory being overwritten by premature buffer swaps.

This particular communication protocol for the master-slave model of parallel QCS is contained in full within the supplemental code repository. The implementation has been tested thoroughly for correctness on $2*\left(H^{\otimes N}\right)$ circuits of many different sizes as well as the smaller circuits described in Chapter I. The results of these tests are included in the next chapter. Overall, the master-slave paradigm increases the amount of memory available to the master node for computation. However, each access of non-local memory is very expensive, since the master node must undergo idle clock cycles while it awaits responses from the slave nodes to its data and control messages. Therefore, while the master-slave implementation is correct and does allow for larger circuits to be simulated, it is certainly not an efficient way to conduct simulation.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. PERFORMANCE ANALYSIS AND OPTIMIZATION OF THE PARALLEL QUANTUM COMPUTER SIMULATOR

A. ASCERTAINMENT OF CORRECTNESS OF THE PARALLEL QUANTUM COMPUTER SIMULATOR

In general, algorithmic performance analysis is tantamount to worthless if the algorithm is not guaranteed to always produce correct results. Additionally, it is notoriously difficult to ensure correctness for programs being executed in parallel across multiple processors due to deadlock and/or race conditions that may arise unpredictably. These deadlocks or race conditions develop because in MPI programing it is impossible to precisely control when a particular processor will execute a specific piece of code in relation to the other processors running the same code. As a result, a parallel program may execute correctly when run once, but may develop a deadlock or race condition when run a second time due to processors accessing shared resources at slightly different intervals between executions. We therefore ensured the correctness of both the parallel and sequential implementations of the matrix-free Julia universal quantum simulator before analyzing their time and memory performance characteristics.

We tested both versions of the simulator for correctness against the Java-based quantum simulator from [29] beginning with several very small circuits and progressing towards much larger and more complicated circuits. On the parallel QCS, each circuit was simulated multiple times in order to ensure that it returned the same results every execution, and that no deadlock or race conditions ever developed. We also tested the parallel simulator using different hardware platforms as well as with various numbers of total processors to ensure that it correctly performed the simulation in all cases. The QDL files describing all of the test circuits in this section are contained in the Appendix. For all test circuits, both the parallel and sequential in-place simulator computed the correct simulation results every time.

We started by simulating a very simple entanglement circuit on both the parallel and sequential Julia simulators. The entanglement circuit is a two-qubit circuit that begins with both qubits in a classical state of $|0\rangle$, resulting in an initial state vector

$a = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^T$. The circuit places the upper qubit into superposition using an *H*-gate and then entangles both qubits with a $CNOT_{1,0}$ gate. Both the parallel and sequential Julia versions of the simulator correctly computed the state of the quantum system after both transforms were applied as $a' = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} \end{bmatrix}^T$, which was identical to the results from the simulator from [29].

Next, we tested both simulators using the three-qubit teleportation circuit, which begins in a classical state of $|4\rangle$. In this circuit, the middle qubit is placed into superposition using an *H*-gate, and then the three qubits are entangled using a $CNOT_{2,1}$ gate followed by a $CNOT_{1,0}$ gate. Next, the upper qubit is placed into superposition using another *H*-gate. Finally, the upper and middle qubits are measured. There are four possible final state vectors for this circuit, depending on the results of the two pseudo-random measurements; these four permutations are shown in Table 9. Both the sequential and parallel Julia implementations return results that are identical to those from the Java simulator in [29] for the teleportation circuit.

Table 9 Teleportation Circuit Final State Vectors

Qubit 0 After Measurement	Qubit 1 After Measurement	Final State Vector
$\|0\rangle$	$\|0\rangle$	$a' = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$
$\|0\rangle$	$\|1\rangle$	$a' = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$
$\|1\rangle$	$\|0\rangle$	$a' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \end{bmatrix}^T$
$\|1\rangle$	$\|1\rangle$	$a' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix}^T$

The next circuits we tested were various three-qubit circuits designed to individually test the implementations of the *H*-, *S*-, *T*-, *X*-, and *Z*-gates. The *S*-gate and *T*-gate are simply special cases of the $R(\phi)$-gate where $\phi = \pi/2$ for the *S*-gate and $\phi = \pi/4$ in

the case of T-gate. Both gates will leave a basis state of $|0\rangle$ unchanged but an S-gate maps a basis state of $|1\rangle$ to $i \cdot |1\rangle$ and a T-gate maps a basis state of $|1\rangle$ to a state of $\frac{|1+i\rangle}{\sqrt{2}}$.

Each of these single-qubit gate test circuits begins in a classical state of $|7\rangle$ and places all three of its qubits into and out of superposition using two of the respective single-qubit gates per qubit, for a total of six transformations. The simulations can be stopped after any particular transformation in order to compare the intermediate state vectors and determine if the different versions of the simulator are producing the correct results. Table 10 displays the correct intermediate state vector after the third transformation for each single-qubit test circuit. Similarly, Table 11 displays the correct final state vector after the sixth and final transformation for each of the single-qubit test circuits. At all steps of each test circuit, both the sequential and parallel simulators matched the correct results computed using the Java simulator from [29].

Table 10 Single-Qubit Gate Test Circuit State Vectors After Third Transform

QDL Circuit (from Appendix)	Intermediate State Vector
H-gate Test	$a' = 2^{-3/2}\begin{bmatrix} 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}^T$
S-gate Test	$a' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & -i \end{bmatrix}^T$
T-gate Test	$a' = \frac{1}{\sqrt{2}}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & (-1+i) \end{bmatrix}^T$
X-gate Test	$a' = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$
Z-gate Test	$a' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T$

Table 11 Single-Qubit Gate Test Circuit Final State Vectors

QDL Circuit (from Appendix)	Final State Vector
H-gate Test	$a' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T$
S-gate Test	$a' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}^T$
T-gate Test	$a' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & -i \end{bmatrix}^T$
X-gate Test	$a' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T$
Z-gate Test	$a' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T$

Having established that both the parallel and sequential Julia QCS were correctly simulating these relatively small circuits, we tested both implementations using the much larger nine-qubit Shor code described in Section I.C.3(3). We created two separate test circuits in QDL to test the Shor code: one to detect and correct a single bit-flip error, and a second to detect and correct a single phase shift error.

The circuit to test for a single bit-flip error begins with a nine-qubit register in a classical state of $|256\rangle$. The qubits are then entangled as depicted in the circuit diagram for the Shor code in Figure 10. A bit-flip is intentionally introduced by applying an *X*-gate at transform 12. If the bit-flip is successfully detected and corrected, the final result will be a classical state of $|448\rangle$. Both versions of the Julia simulator correctly detected and corrected for the single bit-flip error in every simulation.

The circuit to test for a single phase shift error is identical to the bit-flip test circuit except that instead of a bit-flip error, a phase shift error is artificially introduced at the 12th transform using a *Z*-gate. In this circuit, if the phase shift is correctly detected and reversed, the final result will be a classical state of $|292\rangle$. Again, both the parallel and sequential Julia QCS implementations produced equivalent results to the Java simulator from [29].

Having successfully tested the sequential Julia version of the quantum simulator across various quantum circuits, we can state with a high degree of certainty that the simulation algorithms have been implemented correctly. We also tested the MPI simulator using the same QDL circuits multiple times across different hardware platforms while also varying the number of processors and are convinced of the correctness of our parallel implementation of these same quantum simulation algorithms. Moreover, we are convinced that neither deadlocks nor race conditions will develop in the course of simulating any well-formed quantum circuit.

B. PERFORMANCE ANALYSIS OF THE PARALLEL QUANTUM COMPUTER SIMULATOR

With the correctness of both Julia quantum simulators established, we next compared the time and memory performance of the parallel QCS to the in-place sequential QCS on all of the circuits from the previous section. Table 12 depicts the total amount of time required to execute each of the circuits from the previous section on both the sequential and MPI versions of the simulator. For all experiments in the remainder of this thesis, the sequential simulator ran on a single Hamming node with an Intel Xeon ® E-5 2698 CPU, and the parallel simulator ran on two or more of these same nodes. Additionally, to ensure that each Hamming node was not running any other jobs during our performance analysis tests, we executed each simulated circuit using the bash script contained in the appendix. In this bash script, the `#SBATCH --exclusive` line ensures that no other jobs run on the same nodes during our simulations.

The results in Table 12 show that for all tested circuits, the sequential simulator is approximately 10 times faster than the parallel simulator. Since both simulators are running the same in-place algorithm, there is clearly a performance degradation associated with distributing the quantum simulation across multiple nodes for quantum circuits of this size. However, it is common in the field of scientific computing to see a decrease in performance for a fixed size problem when the problem size is below a certain limit [21]. For all but so-called embarrassingly parallel problems (i.e., problems in which there is little to no data dependency or need for communications between processors), the problem size must increase beyond a certain limit in order to realize the

benefit of distributing computation across multiple processors [21], [22]. This is especially true in the case of a memory-bound, rather than computation-bound, problems such as quantum computer simulation. Moreover, since adding additional nodes in the master-slave simulation model only increases available memory and does not add any computation power, such a performance decrease is to be expected.

Table 12 QCS Time Performance Analysis

QDL Circuit (from Appendix)	Sequential Runtime (s)	Parallel Runtime (s)	Parallel to Sequential Runtime Ratio
Entanglement	0.0181	0.175	9.65
Teleportation	0.0181	0.238	13.13
H-gate Test	0.0182	0.154	8.47
S-gate Test	0.0165	0.149	8.97
T-gate Test	0.0180	0.146	8.11
X-gate Test	0.0166	0.141	8.45
Z-gate Test	0.0167	0.144	8.58
Bit-Flip Circuit	0.0215	0.229	10.65
Phase Shift Circuit	0.0220	0.231	10.52

The quantum circuits in Table 12 are all small enough that they can be simulated on a single processor. However, once a quantum system is large enough that the two state-vector buffers (i.e., the vectors *a* and *a_prime* from Section III.A.2) cannot be stored within the memory of a single processor, the sequential QCS becomes unable to simulate that system. The maximum number of qubits in a quantum circuit that can be simulated on a single processor is entirely dependent on the amount of memory available to that processor.

In Section III.A.3, we showed that on both the AMD Opteron™ 6174 and the Intel Xeon ® E-5 2698 CPUs on the Hamming HPC cluster, the maximum number of

qubits that could be simulated in a $2*\left(H^{\otimes N}\right)$ circuit was 33. Using the parallel QCS, however, we were able to increase the simulation size beyond this limit, although the simulations took significantly longer due to the added communications overhead. The total memory requirements for quantum simulation doubles for every additional qubit in the quantum register for the $2*\left(H^{\otimes N}\right)$ circuit. Therefore, we doubled the number of processors for every additional qubit beyond 33 in order to have enough memory to enable simulation.

Table 13 shows a comparison of the runtimes of the sequential simulator to the parallel simulator on the $2*\left(H^{\otimes N}\right)$ circuit as the number of qubits in the quantum register, *N*, increases. Although we have shown that we can increase the size of the simulated quantum register by utilizing memory on additional slave nodes, there is a significant performance slowdown due to the added communication overhead. For a quantum register of 20 qubits, the parallel simulation is 476 times slower than the sequential simulation. This ratio decreased, however, to 112 times slower for a 30 qubit register and 85.2 times slower for a 33 qubit register. The parallel Julia QCS was also able to simulate $2*\left(H^{\otimes N}\right)$ circuits with 34 and 35 qubit registers, but these simulations took over 65 and 135 hours, respectively. Longer simulations are theoretically possible but were not attempted due to resource constraints on the Hamming HPC cluster. The reason for these long simulation times is that in the master-slave paradigm of the parallel QCS, the slave nodes provide only additional memory and no added processing power. Thus, the increased communication costs between the master and slave nodes slow down computation greatly. These communication costs will be analyzed quantitatively in Section C.

Table 13 $2*\left(H^{\otimes N}\right)$ Circuit Sequential and Parallel Simulation Times

Quantum Register Size (N)	Sequential Runtime (s)	Parallel Runtime (s) / Number of Processors	Parallel to Sequential Runtime Ratio
20	0.127	60.5 / 2 processors	476
25	3.62	1,920 / 2 processors	530
30	136	15,300 / 2 processors	112
33	1,350	115,000 / 2 processors	85.2
34	Error	234,000 / 2 processors	-
35	Error	489,000 / 4 processors	-

Next, we studied the strong scaling of the parallel QCS by increasing the number of processors available to simulate a fixed-size $2*\left(H^{\otimes N}\right)$ circuit. We compared the execution times of the parallel QCS for three different $2*\left(H^{\otimes N}\right)$ circuits as the number of MPI processes increased from 2 to 32. Table 14, Table 15, and Table 16 show the experimental strong scaling results for a 10, 12, and 14-qubit circuit, respectively. Each table also compares parallel runtimes to the sequential QCS runtime on the same circuit.

Table 14 Strong Scaling of the QCS on a $2*\left(H^{\otimes 10}\right)$ Circuit

Processors	Runtime (s)	Runtime Ratio to Sequential QCS
1 (Sequential QCS)	0.0411	1
2	0.0613	1.49
4	0.091	2.21
8	0.132	3.21
16	0.214	5.20
32	0.952	23.16

Table 15 Strong Scaling of the QCS on a $2*\left(H^{\otimes 12}\right)$ Circuit

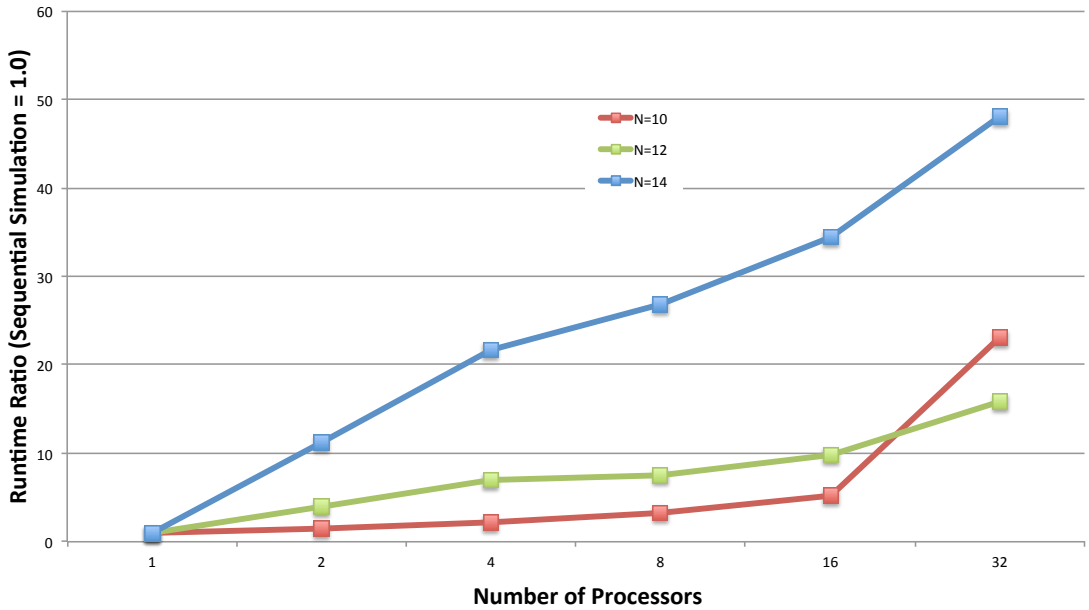
Processors	Runtime (s)	Runtime Ratio to Sequential QCS
1 (Sequential QCS)	0.0588	1
2	0.229	3.89
4	0.408	6.93
8	0.439	7.46
16	0.578	9.83
32	0.936	15.91

Table 16 Strong Scaling of the QCS on a $2*\left(H^{\otimes 14}\right)$ Circuit

Processors	Runtime (s)	Runtime Ratio to Sequential QCS
1 (Sequential QCS)	0.0637	1
2	0.715	11.2
4	1.38	21.7
8	1.71	26.8
16	2.20	34.5
32	3.06	48.0

Figure 13 graphically depicts the ratio of each simulation's runtime to the sequential simulator's runtime for the three different $2*\left(H^{\otimes N}\right)$ circuits as the number of processors increases. Increasing the number of MPI processes involved in the simulation increases the amount of global memory available and thus increases the maximum size of the quantum system that can be simulated. This is clearly not without cost, however, as the increased communications overhead necessarily slows down the simulation significantly. For example, while 32 processors operating in parallel have 32 times the global memory as a single processor, it takes 48.0 times longer to simulate a 14 qubit $2*\left(H^{\otimes N}\right)$ circuit, as shown in Table 16.

Figure 13 Runtime Ratio Compared to Sequential QCS versus Number of MPI Processes for Various $2*\left(H^{\otimes N}\right)$ Circuits

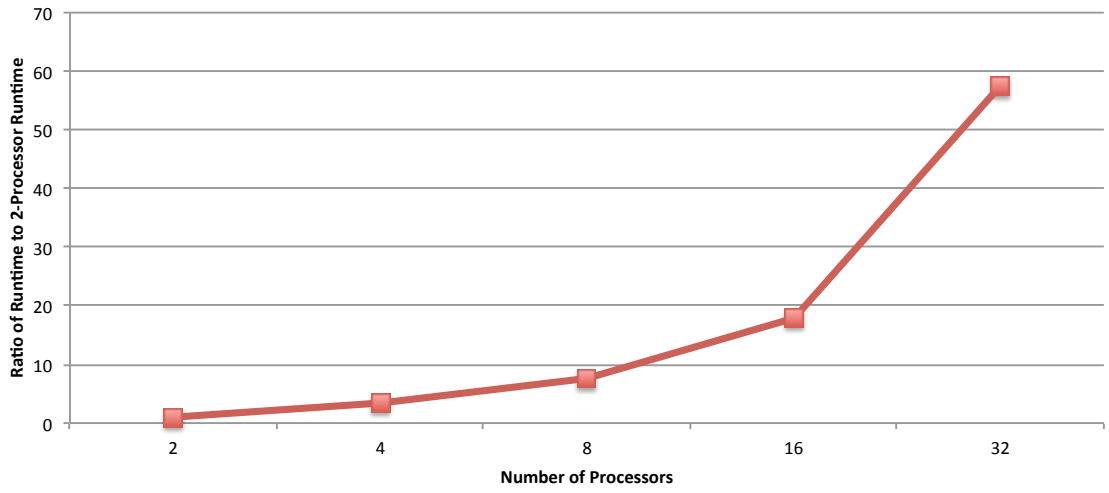


We next studied the weak scaling of the parallel QCS by doubling the number of processors available each time we doubled the size of the problem. This is equivalent to doubling the number of MPI processes for each additional qubit in the quantum system. We began this experiment with a $2*\left(H^{\otimes N}\right)$ circuit with a baseline of $N = 10$ and 2 MPI processes. We then doubled the number of processors for each qubit above the baseline of 10 in the circuit. The results are shown in Table 17, and again indicate the significance of the increased communications cost due to adding more MPI processes. This matches the expected weak scaling results, since adding additional processes in the master-slave QCS model only increases global memory and does not improve computation ability. Figure 14 graphically displays the runtime ratio compared to the dual processor baseline for this experiment.

Table 17 Weak Scaling of the QCS

Processors	Quantum Register Size (N)	Runtime (s)	Runtime Ratio to Baseline
2 (Baseline)	10	0.0534	1
4	11	0.180	3.37
8	12	0.410	7.68
16	13	0.952	17.8
32	14	3.06	57.3

Figure 14 Runtime Ratio to Baseline versus Number of MPI Processes For Various $2*\left(H^{\otimes N}\right)$ Circuits



Ultimately, adding additional MPI processes increases the maximum possible size of quantum register that our QCS can simulate. However, adding MPI processes is very costly because of the increased communications overhead. The sequential in-place Julia simulator is therefore optimal when compared to our distributed memory simulator for all quantum systems of 33 qubits and under (on the Hamming HPC). For systems larger than 33 qubits, however, the distributed memory parallel simulator must be used, despite the poorer performance. Performance of the distributed memory model may be improved substantially if all nodes are able to perform computation, as opposed to the master-slave

model where only one node is performing computation. However, choreographing computation amongst all nodes is a non-trivial problem that we leave for future work.

C. COMMUNICATION COSTS IN THE PARALLEL QUANTUM COMPUTER SIMULATOR

The data from Table 12 and Table 13 show that the sequential version of the Julia quantum simulator clearly outperforms the parallel version for smaller circuits. As previously stated, the reason for this decrease in performance is the addition of communications overhead in the parallel simulator. In the sequential simulator, the entire simulation exists in the local memory of a single processor and no external communications are required. Therefore, all memory allocations can be done locally, which is significantly faster than having to communicate with another processor. Table 18 depicts the total number of memory allocations made as well as the total number of bytes allocated during the simulation of each circuit on the sequential QCS.

In the parallel simulator, however, the state space of the simulation is distributed evenly across multiple nodes as described in Section III.C. Prior to performing any computational step, the master node in the parallel simulator must first determine if all of the amplitudes involved in that computational step are stored in local memory. If all of these amplitudes are indeed stored in local memory, then the new amplitude can be computed and stored locally without any additional communications overhead. However, if any of the amplitudes involved in the computation are not in local memory, the master node must wait for the communications protocol discussed in Section III.C to complete before performing the computation. Depending on the type of transformation being performed, this may result in several *READ* or *STORE* calls to one or more slave nodes for each step of computation. At the end of each transformation, the master node must also broadcast a *SWAP* call to each of the slave nodes used in that computation in order to successfully choreograph the global buffer swap.

Table 18 Sequential QCS Memory Allocations

QDL Circuit (from Appendix)	Total Memory Allocations (Total Bytes Allocated)
Entanglement	11.70K allocations (544 KB)
Teleportation	12.07K allocations (554 KB)
H-gate Test	12.13K allocations (555 KB)
S-gate Test	11.48K allocations (527 KB)
T-gate Test	12.08K allocations (554 KB)
X-gate Test	11.38K allocations (524 KB)
Z-gate Test	11.39K allocations (525 KB)
Bit-Flip Circuit	40.60K allocations (1.2 MB)
Phase Shift Circuit	40.86K allocations (1.2 MB)

From the master node's perspective, each *READ* call results in three total MPI messages being sent and received. First, the master node sends a control message to the appropriate slave node consisting of a single Julia *Int64* (8 bytes of information). The master node then sends a data message with another 8-byte *Int64* to indicate the offset that is being read from. The master node then receives a single message from that node containing two *Float64* values (8 bytes each), corresponding to the real and imaginary components of the complex amplitude being read. Therefore, each *READ* call from the master node's perspective results in a total of 32 bytes being sent and received using MPI communication.

Similarly, a *STORE* call involves a single control message consisting of 8 bytes. The control message is also followed by 8 byte offset message and a 16 byte value message. Each *STORE* call therefore also results in a total communications payload of 32 bytes. Finally, each *SWAP* call consists of a single *Int64* being broadcast to every MPI process other than the master process. Thus, a *SWAP* results in the transfer of $8 \cdot (P-1)$ bytes, where P is the total number of MPI processes.

Table 19 depicts the total number of *READ*, *STORE*, and *SWAP* calls (from the master node's perspective) required to simulate each one of the test circuits described in the previous section. The total size of all data communicated between MPI processes

during each simulation has also been calculated and is shown in Table 19. Finally, the total number of bytes sent and received is divided by the total simulation time from Table 12 in order to determine an average communication bandwidth throughout each simulation. Note that this communication bandwidth only accounts for the data payload encapsulated inside each MPI message, and not the size of the entire message itself.

Table 19 Parallel QCS Communication Cost Analysis

QDL Circuit (from Appendix)	Total Number of *READs* / *STOREs* / *SWAPs*	Data Transmitted	Average Communication Bandwidth
Entanglement	6 / 4 / 2	322 B	1.84 KB/s
Teleportation	44 / 23 / 7	2.16 KB	9.06 KB/s
H-gate Test	48 / 25 / 7	2.35 KB	15.2 KB/s
S-gate Test	24 / 25 / 7	1.58 KB	10.7 KB/s
T-gate Test	24 / 25 / 7	1.58 KB	10.8 KB/s
X-gate Test	24 / 25 / 7	1.58 KB	11.2 KB/s
Z-gate Test	24 / 25 / 7	1.58 KB	11.0 KB/s
Bit-Flip Circuit	6,656 / 5121 / 28	377 MB	1.65 MB/s
Phase Shift Circuit	6,656 / 5121 / 28	377 MB	1.63 MB/s

As Table 19 shows, the number of communications required for a simulation grows with the size and complexity of the circuit being simulated. Therefore, using the master-slave paradigm to implement a quantum simulator in MPI will result in higher overhead for larger circuits. This means that the performance loss in the parallel implementation of the Julia QCS, when compared to the sequential QCS, will increase as the size of the quantum circuit increases.

The communications cost also increases as the number of MPI processes involved in the simulation grows. This can be seen by examining the strong scaling of the communication costs—that is, observing the increase in MPI communications for a fixed problem size as the number of processors increases. To perform this experiment, we

again used the $2*\left(H^{\otimes N}\right)$ circuit for N = 10, N = 12, and N = 14. For each fixed problem size, we doubled the number of MPI processes incrementally from 2 to 32 processes, and tabulated the results in Table 20, Table 21, and Table 22. Clearly, as the number of MPI processes increases, more communications to and from the master node will be required, resulting in the slower simulation times previously seen in Section B.

Table 20 Communications Cost Strong Scaling on a $2*\left(H^{\otimes 10}\right)$ Circuit

Processors	*READs* / *STOREs* / *SWAPs*	Total Transmission Size (MB)	Average Communication Bandwidth (MB/s)
2	20,480 / 10,241 / 21	0.983	16.0
4	30,720 / 15,360 / 63	2.76	30.4
8	35,840 / 17,920 / 147	6.24	47.2
16	38,400 / 19,201 / 315	13.1	61.4
32	39,680 / 19,841 / 651	26.9	28.3

Table 21 Communications Cost Strong Scaling on a $2*\left(H^{\otimes 12}\right)$ Circuit

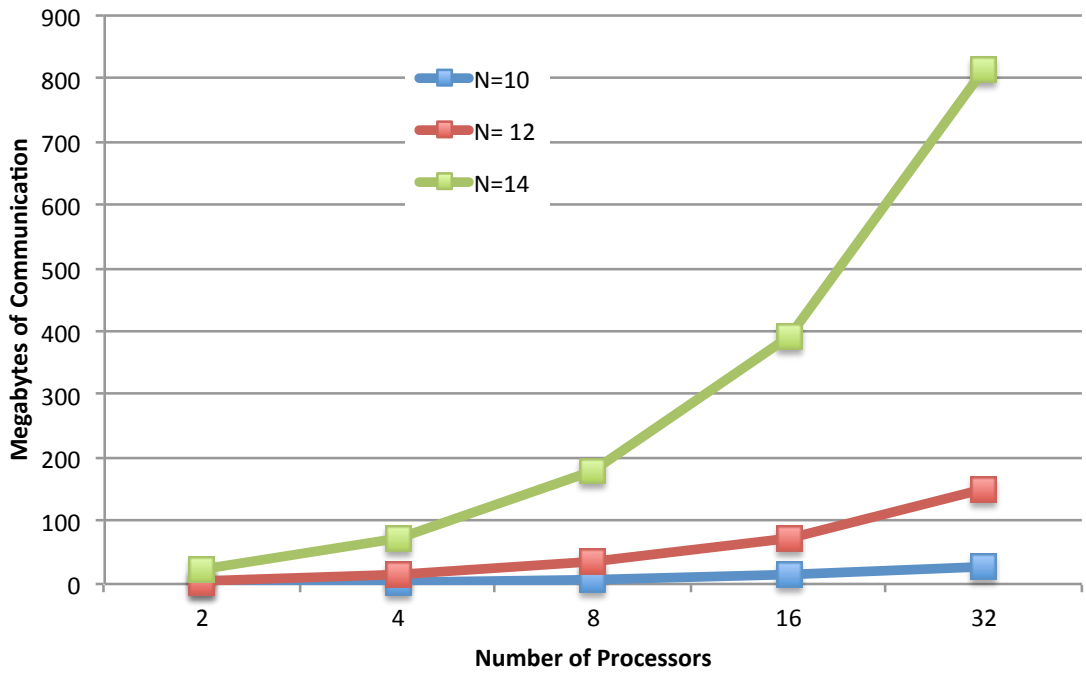
Processors	*READs* / *STOREs* / *SWAPs*	Total Transmission Size (MB)	Average Communication Bandwidth (MB/s)
2	98,304 / 49,153 / 25	4.71	20.6
4	147,456 / 73,729 / 75	14.5	34.4
8	172,032 / 86,017 / 175	34.1	77.6
16	184,320 / 92,161 / 375	73.4	127
32	190,464 / 95,233 / 775	151.9	162

Table 22 Communications Cost Strong Scaling on a $2*\left(H^{\otimes 14}\right)$ Circuit

Processors	*READs* / *STOREs* / *SWAPs*	Total Transmission Size (MB)	Average Communication Bandwidth (MB/s)
2	458,725 / 229,377 / 29	22.0	30.8
4	688,128 / 344,064 / 87	72.9	52.9
8	802,816 / 401,409 / 203	178	104
16	860,160 / 430,081 / 435	390	177
32	888,832 / 444,417 / 899	815	266

Figure 15 graphically depicts the total amount of data transmitted to and from the master node for the three different $2*\left(H^{\otimes N}\right)$ circuits as the number of MPI processes is incrementally doubled.

Figure 15 Total Data Transmitted versus Number of MPI Processes for Various $2*\left(H^{\otimes N}\right)$ Circuits

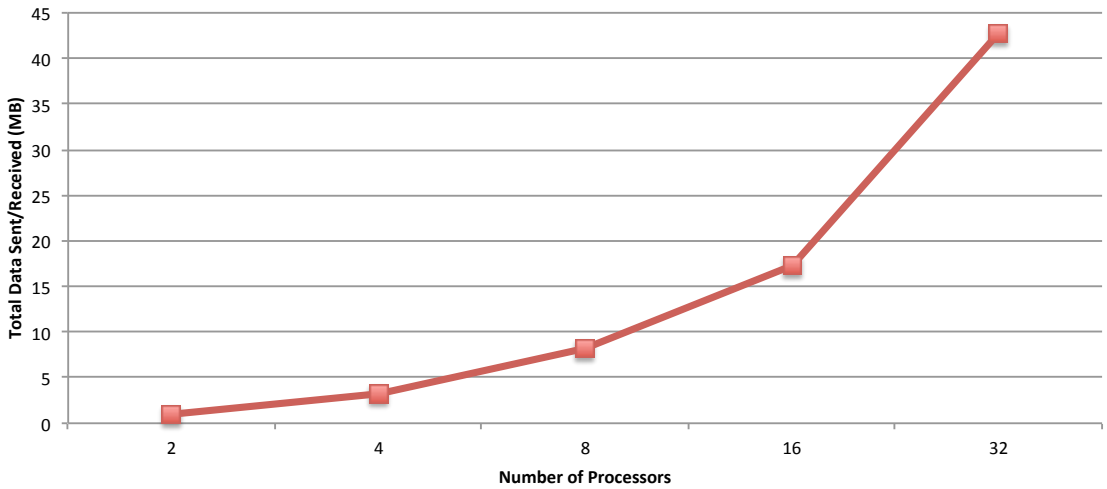


Next, we explored the weak scaling of the communications overhead in our distributed memory QCS using the $2*\left(H^{\otimes N}\right)$ circuit. Again, we began with a baseline of N = 10 and 2 MPI processes. For each additional qubit, we doubled the number of processors and recorded the results in Table 23. As expected, the total size of data transmitted and received by the master node via MPI communications goes up substantially as both the size of the quantum system and the number of processors increase. The total amount of data sent and received by the master node is plotted against the number of MPI processes for this experiment in Figure 16.

Table 23 Communications Cost Weak Scaling of the QCS

Processors	Quantum Register Size (N)	*READs* / *STOREs* / *SWAPs*	Total Transmission Size (MB)	Average Communication Bandwidth (MB/s)
2 (Baseline)	10	20,480 / 10240 / 21	0.983	18.4
4	11	67,584 / 33,793 / 69	3.24	17.9
8	12	172,032 / 86,017 / 175	8.26	20.12
16	13	339,360 / 199,681 / 405	17.3	18.11
32	14	888,832 / 444,417 / 899	42.7	13.94

Figure 16 Total Data Transmitted versus Number of MPI Processes for Various $2*\left(H^{\otimes N}\right)$ Circuits



Ultimately, the distributed memory version of our QCS has been implemented correctly and is free from deadlock and race conditions. It harnesses the added memory associated with extra processors in order to simulate larger quantum systems that were impossible to simulate using the sequential QCS. However, distributing the quantum simulation across multiple nodes is not without cost, and the added communications overhead can be extremely expensive in some cases. While the master-slave model of parallel QCS is reliable in that it produces correct results, it does not scale well due to the unavoidable and exponentially increasing communications costs.

D. OPTIMIZING THE PARALLEL QUANTUM COMPUTER SIMULATOR

Several optimizations were included in the MPI version of the QCS that improved the performance of the distributed memory simulator. We also take note of some opportunities for further optimizations that we leave for future work.

1. Optimizations Included in our Simulator

We included two main optimizations in the final version of our parallel QCS. Each of these optimizations improved performance by a small constant factor. In the case of large simulations that take several days or longer, however, these optimizations are quite valuable.

The first optimization we used was to include the `@inbounds` Julia macro into all for-loops. Julia by default incorporates array bounds checking within all expressions [25]. Thus, when iterating through each of the 2^N amplitudes of an N-qubit quantum register, the Julia just-in-time compiler must check at each iteration whether the array index is located properly inside the array. Adding the `@inbounds` macro before each for-loop eliminates array bounds checking within that loop [25]. We utilized the `@inbounds` Julia macro only after establishing that our simulator was working correctly to ensure that all array indexes in our scripts were indeed located within the bounds of the arrays. We incorporated the `@inbounds` macro into both the sequential and parallel versions of the simulator, and realized approximately a 3% speedup in both cases.

The next optimization we included was to preallocate the *a_prime* buffer at the beginning of the script, as opposed to creating a new *a_prime* buffer for each transformation. Initially, we had only allocated the *a* buffer, and for each unitary transformation performed, we allocated a new *a_prime* buffer consisting of all zeros. At the conclusion of the transformation, the pointer to the *a* buffer was swapped to point to the *a_prime* buffer, and the old *a* buffer was discarded. Reallocating a buffer of zeros of length 2^N for each transform then discarding it at the end of the transform is certainly not without cost. We improved the performance of both the sequential and parallel simulators by declaring the *a_prime* buffer at the same time as the *a* buffer, near the beginning of the script. At the end of each of transformation, the buffer pointers are simply swapped,

and the values in the old *a* buffer can be overwritten in the next transform without reallocation. In the sequential simulator, this resulted in roughly a 4% speedup, but in the parallel simulator it resulted in a variable but always less than 1% speedup. The reason for the smaller performance increase is that in the parallel QCS, the percentage of computational steps in which the local *a* and *a_prime* buffers are used is much smaller. The percentage of calculations that use the master node's local buffers also varies widely depending on the problem size and number of processors.

Overall, we were able to obtain a small but noticeable speedup for both the sequential and parallel simulators by incorporating the `@inbounds` Julia macro and preallocating our working buffers.

2. Potential Future Optimizations

The most obvious optimization, which we leave for future work, is to depart from the master-slave model and enable computation on all nodes. This would alleviate the communications bottleneck that arises as the master awaits communications from slave nodes before being able to perform computation. First, each node can be responsible for parsing the QDL file separately, since this is done in constant time. Each MPI process can then go through every transform independently and determine if one of the global complex amplitudes that it has stored locally is involved in that transformation. If so, that processor can determine which other processors have amplitudes involved in that transformation (in the same manner that our master node determined this), and those processors can engage in pairwise communication. We were unable to devise a working communications protocol to enable computation on all nodes that was free of deadlock and race conditions. Moreover, since the problem of quantum computer simulation is memory bound and not computation bound, it is not clear how much, if any, speedup this would provide.

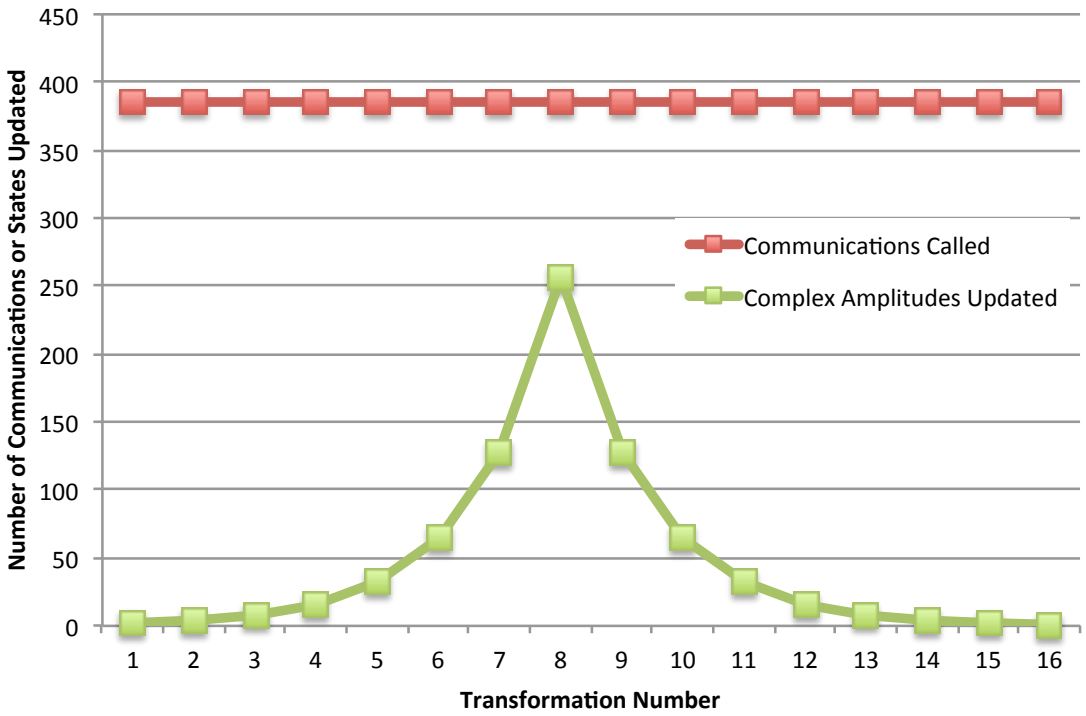
Finally, we examined the number of communications that took place during simulation in our distributed memory QCS in order to attempt to determine if any of the communications were unnecessary. For example, consider the number of communications that take place during each transformation of the of $2*\left(H^{\otimes N}\right)$ circuit.

In our Julia parallel QCS, the number of *READ*, *STORE*, and *SWAP* calls is constant at each step of the transformation. For each transformation, there are 2^N *READ* calls, 2^{N-1} *STORE* calls, and 1 *SWAP* call. Therefore, for each of the $2 \cdot N$ steps, there are a total of $\left(2^N + 2^{N-1} + 1\right)$ combined calls to *READ*, *STORE*, and *SWAP*.

The number of global complex amplitudes updated per transformation is not constant, however. Regardless of the size of the quantum register, the circuit begins in a classical state with only one nonzero complex amplitude in the global *a* vector. Specifically, this amplitude will be equal to $1 + 0 \cdot i$ located at the initial index *k* that corresponds to the decimal representation of initial state of the quantum register, φ_0. In the $2 * \left(H^{\otimes N}\right)$ circuit, the number of nonzero global complex amplitudes always exactly doubles every transformation, until the Nth transformation, after which all global complex amplitudes are nonzero. At this point, the quantum system is in a perfectly balanced superposition between all of its constituent basis states. For the remainder of the transformations, the number of nonzero complex amplitudes is halved for each transformation until there remains exactly one nonzero complex amplitude. If the circuit has been correctly implemented, this nonzero complex amplitude will be an amplitude of $1 + 0 \cdot i$ located at the same index, *k*, at which it began the circuit. Figure 17 displays the total number of combined *READ*, *STORE*, and *SWAP* calls as well as the total number of amplitudes updated for each transformation in a $2 * \left(H^{\otimes N}\right)$ circuit where $N = 8$.

The number of communications is constant every step, although for many of the steps only a small fraction of the complex amplitudes in the global state vector *a* are being updated. This raises the question of whether or not some of the MPI communications that are taking place are unnecessary. It is unclear whether or not some of these MPI communications can be eliminated without loss of generality in the context of a distributed memory universal quantum simulator. We leave further experimentation into eliminating potentially extraneous MPI communications to further work.

Figure 17 Total MPI Communications and Complex Amplitudes Updated per Step of the $2*\left(H^{\otimes 8}\right)$ Circuit



THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION AND FUTURE WORK

A. CONCLUSION

This thesis has produced many important insights and conclusions in both the field of quantum computer simulation as well as parallel scientific computing. We first implemented a sequential matrix-free universal quantum computer simulator using the Julia programming language. For an *N*-qubit quantum register, the matrix-free algorithm allowed us to operate directly on two a 2^N x 1 column vector instead of having to store a 2^N x 2^N unitary matrix for each transformation.

Using the matrix-free, in-place algorithm greatly improved the performance of the sequential QCS over the full-matrix simulation algorithm from [29]. More specifically, the full-matrix simulation algorithm took 76.7 seconds to simulate a 17-qubit quantum circuit on an Intel Xeon E-5 2698 CPU. The sequential in-place implementation took only 0.065 seconds to run the same simulation on the same hardware, for a speedup of 1,180%. The sequential in-place algorithm also increased the number of qubits we could simulate on the same Intel processor from 17 to 33—an increase in the size of the problem by a factor of 2^{16}.

This thesis also showed that quantum computer simulation is memory bound, not computation bound. We alleviated the memory bottleneck by parallelizing our sequential in-place algorithm using a distributed memory approach, although this came at a large performance cost due to the addition of communications overhead. We correctly implemented a method of dividing up the two 2^N x 1 column vectors and distributing their states equally amongst multiple processors. We also showed how to implement unitary transforms with the states divided up and distributed across their different nodes.

Comparison of our parallel and sequential simulators revealed that for circuits small enough to be stored in the memory of a single processor, it is faster to perform quantum computer simulation sequentially rather than in parallel due to high communication costs. However, the distributed memory parallel simulator must be used to simulate any circuits that are too large to be stored on a single processor.

This thesis also produced important academic and educational benefits in the form of open-source sequential and parallel quantum computer simulators. We also developed other scripts designed to study various aspects of quantum computer simulation and parallel scalability, including the parallel and sequential versions of the $2*\left(H^{\otimes N}\right)$ circuit. These scripts are all available through the Dudley Knox Library, and can be studied or improved upon in future work at the Naval Postgraduate School.

B. FUTURE WORK

There are plenty of opportunities for improvement and future work from this thesis. First of all, there are several opportunities for enhancement of the existing parallel quantum simulator. The master-slave node model could potentially be abandoned in favor of all nodes performing choreographed computation. Additionally, eliminating unnecessary communication between nodes can theoretically alleviate some of the communications overhead. There has also been a lot of recent work done in the field of data locality optimization; some form of data locality optimization could perhaps be applied to the method of storing particular quantum states in order to further minimize communication costs.

Finally, different parallel architectures could potentially be exploited in order to improve parallel quantum simulation performance. We do not believe that the "single program, multiple data" parallelization technique, such as Intel's ispc compiler, will provide much useful performance benefit, since quantum computer simulation is memory bound and not computation bound. However, we have not experimented at all with the single program, multiple data paradigm in terms of quantum computer simulation and leave this for future work.

The computations that are performed in the course of quantum computer simulation are quite intensive in terms of the number of floating point operations that are performed. Therefore, we believe that a vector architecture, such as a graphics processing unit, may provide useful performance benefits towards quantum computer simulation. We leave for future work the possibility of translating our existing simulation algorithms to run on vector architecture hardware, perhaps using the open-source OpenCL framework or NVIDIA's CUDA compiler.

APPENDIX. SAMPLE QDL CIRCUITS.

(1) Entanglement Circuit

```
Define N 2
Define Phi0
0 0
Define Transform1
H I
Define Transform2
Control Target
```

(2) Teleportation Circuit

```
Define N 3
Define Phi0
1 0 0
Define Transform1
I H I
Define Transform2
I Control Target
Define Transform3
Control Target I
Define Transform4
H I I
Define Transform5
M I I
Define Transform6
I M I
```

(3) 3-Qubit H-Gate Test Circuit

```
Define N 3
Define Phi0
1 1 1
Define Transform1
H I I
Define Transform2
I H I
Define Transform3
I I H
Define Transform4
I I H
Define Transform6
I H I
Define Transform6
```

```
H I I
```

(4) 3-Qubit X-Gate Test Circuit

```
Define N 3
Define Phi0
1 1 1
Define Transform1
X I I
Define Transform2
I X I
Define Transform3
I I X
Define Transform4
I I X
Define Transform6
I X I
Define Transform6
X I I
```

(5) 3-Qubit S-Gate Test Circuit

```
Define N 3
Define Phi0
1 1 1
Define Transform1
S I I
Define Transform2
I S I
Define Transform3
I I S
Define Transform4
I I S
Define Transform5
I S I
Define Transform6
S I I
```

(6) 3-Qubit Z-Gate Test Circuit

```
Define N 3
Define Phi0
1 1 1
Define Transform1
Z I I
Define Transform2
I Z I
```

```
Define Transform3
I I Z
Define Transform4
I I Z
Define Transform5
I Z I
Define Transform6
Z I I
```

(7) 3-Qubit T-Gate Test Circuit

```
Define N 3
Define Phi0
1 1 1
Define Transform1
T I I
Define Transform2
I T I
Define Transform3
I I T
Define Transform4
I I T
Define Transform5
I T I
Define Transform6
T I I
```

(8) Shor Code Bit-Flip Test Circuit

```
Define N 9
Define Phi0
1 0 0 0 0 0 0 0 0
Define Transform1
Control I I Target I I I I I
Define Transform2
Control I I I I I Target I I
Define Transform3
H I I I I I I I I
Define Transform4
I I I H I I I I I
Define Transform5
I I I I I I H I I
Define Transform6
Control Target I I I I I I I
Define Transform7
I I I Control Target I I I I
Define Transform8
I I I I I I Control Target I
Define Transform9
```

```
Control I Target I I I I I I
Define Transform10
I I I Control I Target I I I
Define Transform11
I I I I I I Control I Target
Define Transform12
X I I I I I I I I
Define Transform13
Control I Target I I I I I I
Define Transform14
I I I Control I Target I I I
Define Transform15
I I I I I I Control I Target
Define Transform16
Control Target I I I I I I I
Define Transform17
I I I Control Target I I I I
Define Transform18
I I I I I I Control Target I
Define Transform19
UT1 UT2 UT3 I I I I I I
Define Transform20
I I I UT1 UT2 UT3 I I I
Define Transform21
I I I I I I UT1 UT2 UT3
Define Transform22
H I I I I I I I I
Define Transform23
I I I H I I I I I
Define Transform24
I I I I I I H I I
Define Transform25
Control I I I I I Target I I
Define Transform26
Control I I Target I I I I I
Define Transform27
UT1 I I UT2 I I UT3 I I
```

(9) Shor Code Phase Shift Test Circuit

```
Define N 9
Define Phi0
1 0 0 0 0 0 0 0 0
Define Transform1
Control I I Target I I I I I
Define Transform2
Control I I I I I Target I I
Define Transform3
H I I I I I I I I
Define Transform4
I I I H I I I I I
Define Transform5
```

```
I I I I I I H I I
Define Transform6
Control Target I I I I I I I
Define Transform7
I I I Control Target I I I I
Define Transform8
I I I I I I Control Target I
Define Transform9
Control I Target I I I I I I
Define Transform10
I I I Control I Target I I I
Define Transform11
I I I I I I Control I Target
Define Transform12
Z I I I I I I I I
Define Transform13
Control I Target I I I I I I
Define Transform14
I I I Control I Target I I I
Define Transform15
I I I I I I Control I Target
Define Transform16
Control Target I I I I I I I
Define Transform17
I I I Control Target I I I I
Define Transform18
I I I I I I Control Target I
Define Transform19
UT1 UT2 UT3 I I I I I I
Define Transform20
I I I UT1 UT2 UT3 I I I
Define Transform21
I I I I I I UT1 UT2 UT3
Define Transform22
H I I I I I I I I
Define Transform23
I I I H I I I I I
Define Transform24
I I I I I I H I I
Define Transform25
Control I I I I I Target I I
Define Transform26
Control I I Target I I I I I
Define Transform27
UT1 I I UT2 I I UT3 I I
```

(10) Slurm Workload Manager Sample Bash Script

```
#!/bin/bash
#SBATCH --job-name=jobname
#SBATCH --output=bashout/filename.out
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2
#SBATCH --time=36:00:01
#    SBATCH --partition=beards
#SBATCH --exclusive
#SBATCH --constraint=intel
##########################################
#                                        #
#   Output some useful job information. #
#                                        #
##########################################

echo -------------------------------------------------------
if [ "${SLURM_NNODES}" -eq "1" ]
then
  echo 'CPUS(xNODES): '${SLURM_JOB_CPUS_PER_NODE}'(x1)'
else
  echo 'CPUS(xNODES): '${SLURM_JOB_CPUS_PER_NODE}
fi
echo 'Job is running on nodes:'
echo $SLURM_JOB_NODELIST
echo -------------------------------------------------------
echo SLURM: submission node:        $SLURM_SUBMIT_HOST
echo SLURM: partition:              $SLURM_JOB_PARTITION
echo SLURM: submission directory:   $SLURM_SUBMIT_DIR
echo SLURM: job identifier:         $SLURM_JOBID
echo SLURM: job name:               $SLURM_JOB_NAME
echo SLURM: current home directory: $HOME
echo SLURM: PATH:                   $PATH
echo -------------------------------------------------------

source /etc/profile
module use /usr/share/Modules/modulefiles
module load compile/gcc/5.2.0
module load  mpi/openmpi/1.10.2
module load julia/0.4.5


cd $SLURM_SUBMIT_DIR

mpirun -np 2 julia simulator_p.jl circuits/bitflip
```

SUPPLEMENTAL

The supplemental tarball file contains five Julia 0.4.5 scripts that were used to conduct the various experiments performed in this thesis. Additionally, a bash script is contained which is designed to run these Julia scripts on HPC cluster using the Slurm Workload Manager. Various quantum circuit files, in the QDL format, are also contained within the `circuits/` subdirectory. Finally, a readme file (entitled simply `README`) is included that contains instructions for how to execute all of the scripts contained in the tarball file.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] N. Yanofsky and M. Mannucci. *Quantum Computing for Computer Scientists*. New York: Cambridge University Press, 2008.

[2] C. Williams. *Explorations in Quantum Computing*. 2nd Edition. New York: Springer Publishing, 2011.

[3] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[4] T. Moses. "Quantum computing and cryptography: their impact on cryptographic practice," Entrust Inc., Dallas, TX, Jan. 2009.

[5] S. Aaronson. "The limits of quantum computers," *Scientific American*, vol. 298, no. 3, pp. 62–69, March2008.

[6] T. Metodi, A. Faruque, and F. Chong. *Quantum Computing for Computer Architects*. San Rafael, CA: Morgan & Claypool, 2006.

[7] S. Anthony. (2014) "IBM cracks open a new era of computing with brain-like chip: 4096 cores, 1 million neurons, 5.4 billion transistors*."* [Online]. Available: http://www.research.ibm.com/articles/brain-chip.shtml.

[8] K. De Raedt, et al. "Massively parallel quantum computer simulator." *Computer Physics Communications,* vol. 176, no. 2, pp. 121–136, 2007.

[9] D. Deutsch. "Quantum theory, the Church-Turing thesis and the universal quantum computer," in *Proc. of the Royal Society of London A: Mathematical, Physical, and Engineering Sciences*, vol. 400, no. 1818, pp. 97–117, Aug. 1985.

[10] D. Deutsch and R. Jozsa. "Rapid solution of problems by quantum computation," in *Proc. of the Royal Society of London A: Mathematical, Physical, and Engineering Sciences*, vol. 439, no. 1907, pp. 553–558, Aug. 1992.

[11] L. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. of the 28thAnnual ACM Symposium on Theory of Computing*, pp. 212–219, 1996.

[12] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. New York: Cambridge University, 2000.

[13] L. Vandersypen, et al. "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance." *Nature*, vol. 414, no. 6866, pp. 883–887, 2001.

[14] E. Martin-Lopez, et al. "Experimental realization of Shor's quantum factoring algorithm using qubit recycling." *Nature Photonics,* vol. 6, no. 11, pp. 773–776, 2012.

[15] D. MacKay. *Information Theory, Inference, and Learning Algorithms*. New York: Cambridge University Press, 2003.

[16] P. Shor, "Scheme for reducing decoherence in quantum computer memory," Physical Review A, vol. 52, no. 4, Oct. 1995.

[17] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Fifth Edition. Waltham, MA: Morgan Kaufmann. 2012.

[18] "Intel 14 nm technology," Intel. [Online]. Available: http://www.intel.com/content/www/us/en/silicon-innovations/intel-14nm-technology.html. [Accessed: 06-Jun-2016].

[19] M. Bohr. "A 30 year retrospective on Dennard's MOSFET scaling paper." *IEEE Solid-State Circuits Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[20] L. Armasu, "Samsung's new $14 billion chip plant to manufacture DRAM, processors in 2017," Tom's Hardware, Aug-2015. [Online]. Available: http://www.tomshardware.com/news/samsung-14-billion-chip-plant,29058.html. [Accessed: 06-May-2016].

[21] T. Rauber and G. Rünger. *Parallel Programming: For Multicore and Cluster Systems*. Berlin: Springer-Verlag, 2010.

[22] D. Culler, et al. *Parallel Computer Architecture: A Hardware/Software Approach*. Burlington, Massachusetts: Morgan Kaufman Publishers. 1999.

[23] G. Torres, "Inside Pentium 4 architecture," Hardware Secrets, 2005. [Online]. Available: http://www.hardwaresecrets.com/inside-pentium-4-architecture/2. [Accessed: 06-Jun-2016].

[24] "Open MPI: open source high performance computing," Open MPI: Open Source High Performance Computing. [Online]. Available: http://www.open-mpi.org/. [Accessed: 06-Apr-2016].

[25] "The Julia language," The Julia Language. [Online]. Available: http://julialang.org/. [Accessed: 06-Mar-2016].

[26] M. Hill and M. Marty "Amdahl's law in the multicore era," *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 2008.

[27] K. Moreland and R. Oldfield, “Formal metrics for large-scale parallel performance,” Lecture Notes in Computer Science High Performance Computing, pp. 488–496, 2015

[28] J. L. Gustafson, “Reevaluating Amdahl’s law,” *Commun. ACM*, vol. 31, no. 5, pp. 532–533, Jan. 1988.

[29] J. Weathers. “Methods for quantum circuit design and simulation,” M.S. thesis, Comp. Sci., Naval Postgraduate School, Monterey, CA, 2010.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California