University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Development and Optimisation of Quantum Computing Simulators with a Study of Quantum Algorithms

Youssef Moawad

April 3, 2019

Abstract

Quantum computing has been a rapidly growing field over the past two decades as it promises dramatic improvements for some algorithms over classical computers. Simulating quantum computers is necessary for researching new quantum computing algorithms because real quantum computers are quite expensive and difficult to build. Simulating a quantum computer on a classical computer is a difficult task because the complexity grows exponentially with the number of qubits required to simulate. In this work, we present two such simulators based on different approaches, a matrix-based approach and a matrix-free approach. Optimisations are developed for the simulations and CPU parallelisation is applied. We compare the performance of the optimised simulators to their initial unoptimised implementations as well as to some third party simulators. We find that some of the developed optimisations give significant performance improvements. We also show that the matrix-free simulator dramatically outperforms the matrix-based simulator, but is outperformed by the existing third-party ones. We also give a study of some quantum algorithms and give their implementations on the developed simulators. Finally, we present an interpreter for a quantum language to make accessing the simulator easier for potential end-users.

Acknowledgements

I would like to thank my supervisors, Dr. Syed Waqar Nabi, Dr. Rene Steijl, and Dr. Wim Vanderbauwhede for providing a great deal of support throughout this project. Without their guidance, this work would not have been possible.

Dedicated to my parents, who supported me through my four years of undergrad at Glasgow University and who always pushed me to achieve the best I can. Mama, Papa, I can never thank you enough!

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Youssef Moawad Date: 27 March 2018

Contents

List of Figures

1 Introduction

1.1 End of Moore's Law

Moore's law is an observation made by Gordon Moore, the co-founder of Intel, which states that the number of transistors that can be fit onto a computing chip doubles every roughly two years, by virtue of our ability to create ever smaller transistors. However, as this continues we eventually start reaching limits set forth by nature on how small we can create transistors. Since electrons have to flow through these transistors, the smaller the transistors are the more likely quantum mechanical effects will cause errors in computation. In fact, this predicts that Moore's law must come to an end (Waldrop 2016).

1.1.1 Quantum Computing as a Mitigating Solution

We now look at Quantum Computing. First suggested by Richard Feynman (1981) at the California Institute of Technology in 1981, it takes advantage of quantum mechanical phenomena such as superposition and entanglement in order to achieve a complexity speedup over classical computers in certain algorithms.

While classical computers use the computational unit known as the *bit*, capable of taking values of either 0 or 1, a quantum computer uses a computational unit called a *qubit*, which is capable of being in a *superposition* of 0 and 1. This concept of superposition is introduced more formally in the next chapter. This is commonly referred to as being in a state of 0 and 1 at the same time; which has an element of truth to it, but certainly does not describe the full picture. This may not appear to have any significant benefit when considering only one qubit, which can only be in a superposition of two states. Two qubits, however, can be in a superposition of four states: 00, 01, 10, and 11; Three qubits allow us to have a superposition of eight states; and so on. *The number of states doubles with each additional qubit.* For a system with n qubits, we can have a superposition of 2^n states.

Quantum Parallelism This notion of superposition allows us to exploit *quantum parallelism*. This refers to the property of quantum computers to effectively evaluate some function (or rather, the quantum analogue of some function) for multiple values at the same time. A function evaluated on a quantum register in a superposition is evaluated for all the components of the superposition. Whereas this may not seem particularly useful as the result will be a superposition of the values of the outputs of the function, some algorithms have been devised which cleverly takes ad vantage of such effects. These algorithms include Shor's algorithm for factoring large numbers, which takes polynomial time, and Grover's search algorithm which is quadratically faster than the classical equivalent.

1.2 Simulating Quantum Computers

The mathematical formalism of quantum computing is largely described by linear algebra. Most importantly, the operations of matrix multiplication and the tensor product are used extensively. This is fairly straightforward to simulate, and there are libraries available which implement such computations efficiently, e.g. Numpy. However, there are some subtleties with this formalism which make it difficult to implement a simulator capable of executing any circuit it is given (see e.g. section 3.1.3).

For this project, Numpy was used to prototype most of the implementation due to its ease of use and how fast such prototypes can be made. However, the prototypes were then ported to C++ to utilise CPU acceleration via OpenMP. A matrix system was written from scratch in C++, as only two matrix operations were required.

In addition to the straightforward approach of directly simulating the mathematical formalism, there is also an alternative approach which proved to be far more efficient and requiring less memory than the first approach. This paradigm is also explored in this paper.

1.3 Goals of the Project

In this section, the fundamental aims and objectives are presented.

1.3.1 Problem Statement

Since it is quite difficult to build an actual quantum computer which can operate on a useful number of qubits, simulating quantum computers in order to test algorithms becomes of significant importance. This is, however, quite memory-demanding on classical computers as each added qubit doubles the amount of memory required to store the state which represents the system.

1.3.2 Aims

The aim of this project is to implement a quantum computer simulator from scratch and explore different optimisations which can be applied to it to improve memory usage and minimise running time. The implemented quantum computer should be *universal*, i.e. capable of running any quantum circuit. We also aim to implement multiple quantum computing algorithms to demonstrate this. Finally, an interpreted language will also be developed in order to provide a more intuitive user experience for a potential end-user.

1.4 Dissertation Outline

The dissertation is structured into eight chapters as follows:

- **Chapter 2** provides some background about quantum information and computation. We also discuss some already existing quantum computer simulators and some approaches to simulation.
- **Chapter 3** discusses the first and most straightforward approach of quantum computer simulation: the **Matrix-based** approach, which is a direct representation of the mathematical formalism behind quantum computing.

- **Chapter 4** discusses a different approach to simulation which provides a massive performance improvement over the matrix-based approach: the **Matrix-Free** approach.
- **Chapter 5** introduces a variety of quantum computing algorithms which were implemented on the developed simulator. This includes algorithms which simply demonstrate quantum computing advantage, such as **Deustch's Algortihm**, as well as algorithms of significant applications including **Shor's algorithm for factoring large numbers** and **Grover's search algorithm**.
- **Chapter 6** introduces the **Quantum Language Interpreter**, an interpreter for a DSL which provides easier access to writing quantum circuits for the simulator.
- **Chapter 7** contains details about the evaluation strategy devised and evaluation results.
- Finally, **chapter 8** concludes the dissertation with a summary of the most important results found.

2 Background

In this chapter, we explore the fundamentals of computation using quantum systems. The main concepts of quantum computers are discussed. The main reference for these concepts is the book by Nielsen and Chuang (2010), *Quantum Computing and Quantum Information*. The quantum circuit diagrams used throughout this work were typeset using the Q-Circuit package by Eastin and Flammia (2004).

2.1 Quantum Computing Concepts

2.1.1 Qubits

A binary quantum state can be represented by the ket vector $|\psi\rangle$:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix},$$

where α and β are the *complex probability amplitudes* of the state. For the purpose of this paper, a *ket vector* can be thought of as simply some quantum mechanical state.

The fundamental computational unit of a quantum computer is the *qubit*. A qubit simply represents a binary quantum state with the properties described here. Our basis states, 0 and 1, have the following state vector representation:

$$|0\rangle \equiv \begin{bmatrix}1\\ 0\end{bmatrix},\ |1\rangle \equiv \begin{bmatrix}0\\ 1\end{bmatrix}$$

2.1.2 Measurement

The two pieces of information that make up a qubit are its complex probability amplitudes. What these really tell us are the probabilities of finding the qubit in either of the states $|0\rangle$ or $|1\rangle$, *when it is observed*. This is a quantum phenomenon known as *the collapse of the wavefunction*. The probability of finding the qubit in either state is given by the square of the magnitude of the probability amplitudes:

$$P(|\psi\rangle = |0\rangle) = |\alpha|^2, P(|\psi\rangle = |1\rangle) = |\beta|^2$$

2.1.3 State Vectors

In the above representation of the qubit, the vector $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$ represents the *state vector* of a quantum system consisting of one qubit. We can have a quantum system which has two qubits, $|q_1\rangle$ and $|q_2\rangle$, such that:

$$|q_1\rangle = \begin{bmatrix}\alpha_1\\ \beta_1\end{bmatrix}, |q_2\rangle = \begin{bmatrix}\alpha_2\\ \beta_2\end{bmatrix}$$

Such a quantum system can take the value of four states: $|00\rangle$, $|01\rangle$, $|10\rangle$, or $|11\rangle$. At any given instant, each possible state will have an associated complex probability amplitude, much like α and β for the single qubit case. We can find these probability amplitudes by getting the state vector which describes the entire state. We do this by taking the tensor product of the state vectors representing each individual qubit:

$$|q_1 q_2\rangle = |q_1\rangle \otimes |q_2\rangle = \begin{bmatrix}\alpha_1\\ \beta_1\end{bmatrix} \otimes \begin{bmatrix}\alpha_2\\ \beta_2\end{bmatrix} = \begin{bmatrix}\alpha_1 \begin{bmatrix}\alpha_2\\ \beta_2\end{bmatrix}\\ \\ \beta_1 \begin{bmatrix}\alpha_2\\ \beta_2\end{bmatrix}\end{bmatrix} = \begin{bmatrix}\alpha_1\alpha_2\\ \alpha_1\beta_2\\ \beta_1\alpha_2\\ \beta_1\beta_2\end{bmatrix}$$

Thus, we can write the combined quantum state as:

$$|q_1 q_2\rangle = \alpha_1\alpha_2 |00\rangle + \alpha_1\beta_2 |01\rangle + \beta_1\alpha_2 |10\rangle + \beta_1\beta_2 |11\rangle$$

This concept is extendable for quantum systems with a higher number of qubits. We say that such a collection of qubits make up a *quantum register*.

2.1.4 Quantum Gates

A *quantum gate* represents some operation which can be run on some number of qubits. This is analogous to classical electronic gates, such as AND and OR. Since we can represent a quantum state using a state vector whose elements are the probability amplitudes of the state, it would make sense to be able to represent operations on the state as matrices which alter the probability amplitudes.

A common quantum gate is the *Pauli-X gate*, more commonly known as the X gate or the NOT gate. This simply has the effect of flipping the value of the qubit, much like the classical NOT gate. The matrix representing this gate is given here:

$$X \equiv \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$$

The circuit representation of the X gate is shown here:

$$|0\rangle \text{ —}\oplus\text{— } |1\rangle$$

We represent gate operation as matrix multiplication by the state vector. Consider the example of applying the X gate to the $|0\rangle$ state:

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

We see that, indeed, applying the X gate to the $|0\rangle$ state gives the expected flipped state, $|1\rangle$.

Applying Quantum Gates to Multi-Qubit State Vectors A state vector representing n qubits has 2^n components representing the probability amplitudes of all the possible states the system can take. However, we cannot multiply a 2×2 matrix by such a state. In order to apply a gate to such a system, we must first expand the gate by utilising the tensor product. This involves applying the tensor product with the identity matrix to left of the required matrix as many times as there are qubits to the left of the target qubit and, similarly, applying the same tensor product to the right of the required matrix as many times as there are qubits to the right of the target qubit. This is seen in the following relation, where a single qubit gate G is applied to the tth qubit and G_t is the expanded matrix:

$$G_t = \bigotimes_{i=1}^{n} \begin{cases} G, & i = t \\ I, & \text{otherwise} \end{cases} \tag{2.1}$$

2.1.5 Superposition

So far the qubits which were discussed were essentially just as useful as classical bits; i.e. we have not taken advantage of any quantum phenomena yet. As mentioned previously, *superposition* refers to the ability of quantum states to exist in a probabilistic state which could be a "combination" of different states. For instance, we could have a single qubit which has a uniform probability of being in the states $|0\rangle$ and $|1\rangle$. Such a state could look like: $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, and would have the state vector: $\frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix}$. Notice that the probability amplitudes are both $\frac{1}{\sqrt{2}}$. We know that we can find the probability of a given state by take the square of the absolute value of its probability amplitude. Thus, for such a state, the probability of finding it to be in the $|0\rangle$ state is $\left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}$. Similarly, we can find the probability of finding the qubit to be in the $|1\rangle$ state to also be $\frac{1}{2}$. Hence, the qubit is indeed equally probable to be found in either of those states.

To put a qubit into such a superimposed state, we introduce the *Hadamard gate* (H):

$$H \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The circuit representation of the H gate is shown here:

$$|0\rangle \text{ —}\boxed{H}\text{—— } \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

We can test this gate on the $|0\rangle$ state:

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

This is the superimposed state we expect. What happens if we apply the Hadamard gate to the other basis state, $|1\rangle$, though? This is demonstrated here:

$$H|1\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}0\\ 1\end{bmatrix} = \begin{bmatrix}\frac{1}{\sqrt{2}}\\ -\frac{1}{\sqrt{2}}\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ -1\end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

This also represents a state in an equally probable superposition, however, the $|1\rangle$ portion of the state is negative. This gives this state a *phase* different to the other one. We explore why this is useful when considering quantum algorithms.

The Pauli-Z Gate These two states form a different basis to the computational basis we described before ($|0\rangle$ and $|1\rangle$):

$$|+\rangle \equiv \frac{|0\rangle + |1\rangle}{\sqrt{2}},\ |-\rangle \equiv \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

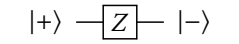
We saw before that we can use a Pauli-X gate to flip a $|0\rangle$ state into a $|1\rangle$ state. Similarly, we can use a *Pauli-Z* gate to flip a $|+\rangle$ state into a $|-\rangle$, and vice versa. The Z gate is given here:

$$Z \equiv \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix},$$

such that:

$$Z|+\rangle = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}\begin{bmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix} = \begin{bmatrix}\frac{1}{\sqrt{2}}\\ -\frac{1}{\sqrt{2}}\end{bmatrix} = |-\rangle\,.$$

The circuit representation of the Z gate is shown here:



We can also similarly show that $Z|-\rangle = |+\rangle$.

2.1.6 Entanglement

This is the second important quantum phenomenon which makes computation on a quantum system of interest. *Entanglement* refers to the experimentally observed coupling of two or more quantum states which are prepared to be in such a state. For example, we could have two qubits prepared such that their state is described by:

$$|q_1\rangle \otimes |q_2\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Notice that this implies that the only two possible states of such a system are $|00\rangle$ and $|11\rangle$. However, we can perform independent measurements on each qubit. This means that if we were to measure one of these qubits, and find it to be in some state, it will be guaranteed that the other qubit is in that same state without measuring it. In effect, this means that an attempt to collapse the wavefunction of one of the qubits will immediately also collapse the wavefunction of the other qubit, because in reality they can only be described by a single wavefunction. We say that the two qubits are *entangled*.

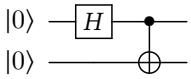
The remarkable thing is that this seems to happen instantly, regardless of the physical distance between the qubits. Hypothetically, we can prepare two entangled qubits, put each of them on a

spaceship and take them to separate ends of the universe, and conduct this experiment (provided they do not decohere during the trip!). This was indeed verified but on the scale of a few hundred miles rather than on different ends of the universe!

In order to create an entangled pair of qubits, we require the *CNOT* (controlled not) gate. This is a multi-qubit gate which has the effect of applying the X gate to some qubit (the target) if another qubit is $|1\rangle$ (the control). This has predictable effects if the control qubit is simply either $|0\rangle$ or $|1\rangle$. Interesting effects happen, however, if the control qubit is in a superposition. The matrix representing this gate is:

$$CNOT \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

To create an entangled pair from two qubits which are not in a superposition, we apply the H gate to one of them then use *CNOT* with the superimposed state as the control:



2.2 Approaches to Simulating Quantum Computers

In this section, three approaches to implementing quantum computer simulators are presented. Two of these approaches were implemented and explored during the project.

2.2.1 Matrix-Based Approach

This is the most straightforward approach to simulating a quantum computer. This simply relies on storing a state vector of the complex probability amplitudes of the quantum system and representing gates by their matrix operators. In order to apply a gate to the quantum system, it either has to be an n-qubit gate, where n is the number of qubits in the system, or it has to be explicitly expanded using the tensor product such that it can be applied to the system.

This presents significant memory load on the system. For starters, the program will always have to store the state vector. For an n-qubit quantum register, this is 2^n complex numbers. A complex number is represented by two 64-bit doubles. This allows us to write a function for the amount of memory required to store such a state vector:

$$M_{SV}(n) = (2 \times 2^n \times 8) \text{ bytes}$$

This means that the memory required for this grows exponentially as we add more qubits to our simulation. In addition, any gate which we would need to run on the quantum system would be represented by a matrix of dimensions $2^n \times 2^n$. Again, we can write a function for the amount of memory this would require:

$$M_G(n) = (2 \times 2^{2n} \times 8) \text{ bytes}$$

We see that this grows even faster than the state vector. During this work, we explored multiple ways to optimise both the performance and memory issues of this approach.

2.2.2 Matrix-Free Approach

This approach attempts to solve the memory shortcomings of the matrix-based paradigm by taking away the requirement to explicitly expand matrices using the tensor product. Instead, for a gate application, it systematically iterates over the amplitudes and updates them as required.

2.3 Existing Simulators

2.3.1 libquantum

libquantum (Butscher and Weimer 2007) is a library initially developed for the purpose of simulating general quantum systems but was later given the functionality to simulate quantum computing circuits, by BjÃűrn Butscher and Hendrik Weimer. It is written in C and requires the user to directly utilise the functions and structures provided to simulate a circuit. It is able to simulate any quantum circuit and includes implementations of two important quantum algorithms: Shor's algorithm for integer factorisation and Grover's search algorithm.

This was chosen as one of the targets with which to compare the developed simulator.

2.3.2 QX Simulator

QX Studio (Khammassi b), developed at QuTech, is a similar simulator to libquantum in that it can simulate any given quantum circuit. QX Studio was also chosen as a comparison target, however, because it also includes an interpreter for a quantum computing language called Quantum Code, usable through a QX Studio which provides a GUI for writing Quantum Code and running the simulation. A user's manual on Quantum Code is available (Khammassi a). A comparison between Quantum Code and the DSL developed as part of this work will be given.

2.4 Acceleration

Many options were considered for accelerating the running of the simulator. CPU, GPU, and FPGA acceleration were all considered. Due to the scope of the project, we constrained ourselves to CPU acceleration. This allowed us to focus instead on exploring some interesting quantum algorithms.

2.4.1 Parallelisation using OpenMP

All of the CPU acceleration utilised throughout this work is done through OpenMP parallelisation. This proved to be very convenient, as much of the linear algebra involved in running the computations is already easily parallelisable. However, in developing optimisations for the matrix-based approach, we found opportunities for parallelising other parts of the program as well, which proved very beneficial. The matrix-free approach also offered interesting opportunities for parallelisation.

3 Matrix-Based Simulator

In this chapter, the implementation of the matrix-based simulator is presented along with the optimisations that were developed for it.

3.1 Numpy-based Implementation

The project started off by writing a prototype for the simulator in python using Numpy. Numpy was chosen for this because it provides easy access to the linear algebra operations that are required to do the simulation. Python would also prove to be an ideal choice for testing preliminary implementations of some of the optimisations that were investigated over the course of the project.

3.1.1 Representing Qubits and State Vectors

The state vector will contain all the information necessary to represent our quantum state at any point in time. Operations on the quantum state are represented by multiplying a matrix which represents the operation to the state vector. The state vector is simply a list of the complex probability amplitude of each possible observable state. For an n-qubit state, this is a 2^n dimensional array in Numpy.

State Vector Expansion Optimisation We now introduce the first optimisation we can apply. This is a minor optimisation which shaves off some processing time off of any algorithm. We can represent a qubit using two complex numbers (representing the probability amplitudes of the states $|0\rangle$ and $|1\rangle$. This allows us to be able to represent n-qubits using $2n$ complex numbers as long as the qubits are not entangled or required to be combined in order to perform a multi-qubit gate.

However, we do eventually require that qubits be combined and/or entangled in order to run meaningful circuits. To accomplish this, we structure our simulator as follows.

StateVector This class represent a collection of one or more *combined* qubits. It does this by maintaining a 2^{n_s} dimensional Numpy array of the probability amplitudes, where n_s is the number of qubits in the state vector. It also maintains a list of integers which represent the *qubit IDs*. The qubit ID represents a qubit's location is the full quantum register. This is necessary because we will not always necessarily combine adjacent qubits. For this optimisation, it is required to be able to combine state vectors into larger state vectors. We do this by creating a new `StateVector` object whose probability amplitudes array is the tensor product of the first and second arrays. The new qubit IDs list is then simply the the qubit ID lists concatenated. The `StateVector` class exposes a number of important methods which will be discussed in the coming subsections.

QRegister The `QRegister` class represents a full quantum register and it does this by maintaining a list of `StateVector` objects. This class is essentially the "user-facing" part of the library. It exposes methods to apply gates to qubits in the register, as well as a measure() method, which returns one of the possible observable states of the register based on the probabilities. This is discussed further in the coming Measurement section. The quantum register object for n qubits is initialised with a list containing n `StateVector` objects, each of which represent one of the qubits with all of them being in the $|0\rangle$ state.

To further expand on why it is necessary to maintain a list of qubit IDs in each `StateVector` object, we present the following example. Consider a quantum register with four qubits. The `QRegister` is initialised with four `StateVector` objects each with a list of qubit IDs, each of which contains only a single integer, the qubit's ID. Assume that it is required to apply e.g. the CNOT gate to the first and third qubits. These qubits are not adjacent but their state vectors are indepedent, so they need to be combined into a single state vector before the gate is applied. This is done using the method described above. Now our `QRegister` contains three `StateVector` objects, with respective qubit IDs of [0, 2], [1], and [3]. If we now need to apply the CNOT gate to the third and second qubits, we need to combine the first and second state vectors in that list. Again, we do this using the algorithm described above and then we get our `QRegister` having `StateVector` objects with [0, 2, 1] and [3] qubit ID lists.

3.1.2 Gate Applications

A single-qubit gate is simply represented by a 2×2 Numpy array. In order to apply the gate to a state vector of a single qubit, the matrix is multiplied by the probability amplitudes array and then the new array is stored in the state vector.

This can be generalised to n-qubit gates. As discussed in the previous chapter, this is a $2^n \times 2^n$ matrix, so again we use a $2^n \times 2^n$ Numpy array to represent such a gate. Applying the gate to an n-qubit state vector (which will have 2^n probability amplitudes) is again simply using Numpy matrix multiplication. However, it is possible that the gate needs to be applied to qubits which are in different state vectors in the register, so it is necessary to make sure all the required state vectors are combined using the method defined above

Preparing n-qubit Gates to operate on m Qubits However, in order to apply an n-qubit gate to a state vector with m *consecutive* qubits, where $m > n$, we need to explicitly expand the gate matrix to fit the number of qubits in the state vector. This is done by repeatedly applying a tensor product with the 2×2 identity matrix for each qubit on the left and right hand side of the qubits on which the gate is required to operate. We demonstrate this with an example.

Consider a 5-qubit state vector on which we need to apply CNOT to the third and fourth qubits. The 4×4 dimensional CNOT matrix will not suit as we have a $2^5 = 32$ dimensional state vector. We need to construct a matrix suitable for this particular gate-qIDs application. We do this as follows:

$$CNOT_{5,(2,3)} = I \otimes I \otimes CNOT \otimes I$$

This will result in a 32×32 dimensional matrix, $CNOT_{5,(2,3)}$, suitable to be applied to the 32-dimensional state vector. We accomplish this using Numpy's kron function (short for Kronecker product, another name for the tensor product). We can then simply multiply the matrix by the state vector to obtain the updated probability amplitudes.

3.1.3 Qubit Adjacency and Swapping

The example above works because the two qubits are adjacent in the state vector. However, what if it is required to apply CNOT to two non-consecutive qubits? Or rather, to generalise: how do we apply an n-qubit gate to an m-qubit state register where the target qubits are not necessarily adjacent in the state vector?

For simplicity, and to have the ability to test algorithms early on in the development period of the project, a naive swapping algorithm was devised. This constituted simply swapping the target qubits (let their count be t) with the first t qubits (while keeping track of the order of swaps), applying the gate to the state vector (keeping into account expansion of gate matrices), and the swapping the qubits back in reverse order. This algorithm works but is rather wasteful of precious processing time. This algorithm was later revised and an updated version implemented which takes advantage of the fact that state vectors store lists of qubit IDs, which means that, when it comes to making measurements, the actual order of qubits in the state vector is irrelevant as they will be rearranged after the measurement is taken.

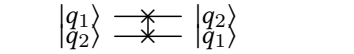
The important thing here is not to change the value of the swapped qubits, i.e. when we perform a swap in the state vector, we must also swap the respective qubit IDs in the qubit IDs list of the `StateVector` object. Thus we can bring the qubits to be adjacent in the vector of probability amplitudes such that we can perform a useful gate on them but when they are measured, we get the correct qubit's measurement in the correct place of the register.

Notice that it is required to do this only if all the five qubits are in the same state vector. So, in the example circuit given above, if the quantum register was newly initialised, then all the qubits would be seperated into their own `StateVector` objects. If we need to perform an important two qubit gate on the first and fifth qubits, we would simply combine the the first and fifth `StateVector` objects into a single and we would not need to perform any swaps (unless the gate requires the fifth qubit to be "above" the first, then a single swap would be required). We only need to perform this sort of complicated if all five qubits are in the same state vector.

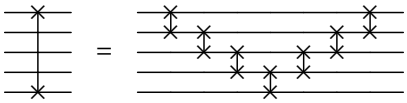
Swapping non-adjacent qubits We now introduce the SWAP gate:

$$SWAP \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This gate swaps the probability amplitudes of two *adjacent* qubits. In a circuit, this gate is represented by the following gate diagram:



Since we are trying to solve the problem of multi-qubit gates only being applicable to adjacent qubits, any gates we use to swap the qubits must only be applied on adjacent qubits. This requires that in order to swap two qubits seperated by k other qubits that we apply $2k + 1$ adjacent SWAP gates. This is demonstrated in the following circuit:



3.1.4 Measurement

We now have a simulator capable of representing any number of qubits (given infinite memory) and of running any quantum gate on any combination of those qubits. We now need to be able to simulate a measurement of the quantum register, i.e. collapse the wavefunction. For the purposes of the simulator, we use a measurement algorithm which does not change the state vector, such that we can take multiple measurements on the same quantum system without having to rerun the quantum circuit after each measurement.

We now simply need to randomly pick out one of the states using a weighted distribution. Python provides functions to do this in the random module. It is just needed to come up with a list of all possible state strings by doing repeated cartesian products of the characters '0' and '1'. Python also provides a function for this.

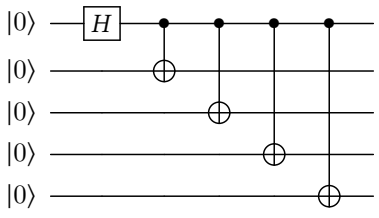
Accumulator Physically, once a measurement has been on a system, its wavefunction collapses and it reduces to a state in which we can measure it. This means that we are unable to take a second measurement on the same system but instead have to prepare it again by rerunning the quantum circuit on a blank system. However, since our measurement algorithm does not alter the probability amplitudes of the system, we are free to take as many measurements as we like. We developed an `Accumulator` system to make it easy to take a large number of measurements. A simple function takes in the quantum register in question and performs some given number of measurements, returning the results in a dictionary.

3.1.5 Demonstrating Entanglement

We can now start to implement any quantum algorithm on our Numpy simulator. While quantum algorithms are covered in their own chapter in this report, we briefly demonstrate entanglement using the Numpy simulator here.

Two-Qubit Entanglement As discussed earlier, entanglement is achieved by applying an H gate to one of the qubits and then a CNOT gate to both qubits, where the control qubit is the one in a superposition. When this circuit is run in our Numpy simulator and a large number of measurements are taken using our Accumulator, we get a roughly equal number of $|00\rangle$ and $|11\rangle$ states, which demonstrates that the pair of qubits is indeed entangled.

Many-Qubit Entanglement Similarly, we can entangle as many qubits as we need. This is accomplished by simply applying as many CNOT gates as there are qubits other than the first superimposed one, always using the superimposed one as the control. The following circuit diagram represents this for five qubits:



Again, when this is run, we get the expected result of an equal likelihood of either the $|00000\rangle$ state or the $|11111\rangle$ state.

3.2 C++ Dense Matrix Implementations

Now that we have a prototype of our simulator written in Python and Numpy, we can move translate it over to C++, which will allow us to explore parallelisation paradigms for the different parts of the simulator.

Most of the components of the Numpy simulator translate directly to C++. The fundamental difference is that instead of using a linear algebra library, we provide a custom implementation of dense matrices with matrix multiplication and tensor product operations. This was chosen in order to not worry about any overhead that might come with a library and to have complete control over the components used in the final simulator.

3.2.1 Parallelisation Opportunities

We now discuss any opportunities for parallelisation which could be applicable to our simulator, as it currently stands. The most obvious ones to start with are for our linear algebra implementations; matrix multiplication and tensor products. These algorithms can be parallelised in standard ways and using simple OpenMP parallel for loop pragmas.

Another opportunity for parallelisation comes in the form of parallelising the gate expansion steps (for applying n-qubit gates to m-qubit state vectors). This is a complex algorithm and is discussed in detail in the coming `CircuitOptimiser` section.

3.3 Sparse Matrix Optimisation

A significant bottle-neck to performance of the current implementation is that it is very memory-expensive to store the entire expanded matrices when applying n-qubit gates to m-qubit state vectors, especially when very few entries in the matrices are nonzero. This is also costly in terms of processing time. This makes it so that using sparse matrices instead of dense matrices to represent gates could provide a significant performance increase as well as save significant memory, allowing for higher number of qubits to be represented.

3.3.1 Implementation

Again, we opted for a custom implementation of sparse matrices. We store an array of complex doubles which represent the nonzero values populating the matrix. We also store an array of pairs of integers, which represents the "keys" of the matrix, i.e. the row and column values of the matrix. This is a variation of the DOK (dictionary-of-keys) implementation of sparse matrices. This makes it easy (and parallelisable) to zip through the nonzero values of the matrix, which allows us to write a very efficient tensor product method.

3.3.2 Difficulties in Parallelising Sparse Matrix Multiplication

Whereas tensor product performance is significantly improved when utilising DOK spare matrices, we trade-off being able to parallelise matrix multiplication at all. This is mainly due to the fact that the number of nonzero values of the resulting matrix is unknown until the nonzero values of both matrices have been iterated on. Some papers regarding this issue were reviewed (Buluç and Gilbert (2012), Wolf et al.), but it was decided it was beyond the scope of this project to try to mitigate this issue.

3.4 Optimising Tensor Products with the Identity Matrix

We recognise that almost all of the tensor products involved in the expansion of gate to fit multi-qubit state vectors are repeated tensor products with the 2×2 identity matrix. An optimisation can be applied here to reduce the number of tensor products required per gate expansion to just two: one for the set of qubits to the left of the target qubit(s) and one for the set of qubits to the right. Let n_l be the number of qubits to the left of the target qubit(s) and n_r be the number of qubits to the right. Then the identity matrices representing each of these sets of qubits are the 2^{n_l}-dimensional and 2^{n_r}-dimensional identity matrices. Thus we reduce the previously discussed Equation 2.1 to:

$$G_t = I_{2^{n_l}} \otimes G \otimes I_{2^{n_r}}, \tag{3.1}$$

where I_n is the nth dimensional identity matrix.

3.5 Optimising the Swap Algorithm

As mentioned above, the initially implemented swapping algorithm used to bring qubits to be adjacent is rather naive and wasteful of processing time. We now introduce some optimisations for this algorithm.

3.5.1 Parallelisation

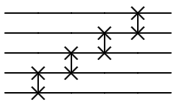
We note that the swapping algorithm, as it currently stands is rather slow, since in order to swap two qubits seperated by k qubits, it is needed to apply 2k-1 SWAP gates. Each of these SWAP gates needs to be explicitly expanded to fit the number of qubits in the state vector. Doing this serially is very processor-consuming. However, since we know all the qubit positions which will need to be swapped, each expanded SWAP gate is independent of any other, so we can prepare all of the SWAP gates initially, in parallel, then apply them to the state vector sequentially.

Initial Idea for the CircuitOptimiser This optimisation also provided the inspiring idea for the `CircuitOptimiser`, discussed in the next section. Since we can implement such parallelisation for swapping, as we know all the gates and target qubits of each gate, then we can apply such an optimisation also to the execution of any circuit, provided we know all the circuit's gates and target qubits in advance.

3.5.2 Removing the Requirement for Swapping: Moving

We now discuss an algorithm which replaces the initially devised swapping algorithm. We recall that the current algorithm for applying an n-qubit gate to an m-qubit state vector involves swapping all the t target qubits with the first t qubits in the state vector, applying the gate, then applying the swaps again in reverse order. The first improvement which we could make here comes from realising that since the `StateVector` object maintains an array of the qubit IDs that its qubits represent in the quantum register of which it is a part, then it does not matter how the qubits are arranged in the actual vector as long as they have the correct label. This makes it so that the step of swapping the qubits back to their original positions unimportant for the purposes of the simulator. This already significantly reduces the processing time it takes to apply such a multi-qubit gate, and this improvement is more noticeable the larger that state vector is.

In addition, in this series of swaps, we notice that all we require is that the k-th target qubit is moved into the k-th position of the state vector; i.e. we do not care about where the qubit it displaces is left. Thus, we do not require that the displaced qubit takes the place of the moved qubit. So we can reduce this swap section to a simple move. This reduces the circuit we need to apply to move e.g. the fifth qubit to the place of the first (the example demonstrated above) to:



This further reduces each swap operation required for preparing to apply a multi-qubit gate (now a move operation) to only $k + 1$ SWAP gates rather than $2k + 1$, for k being the number of qubits seperating the source and target qubit locations.

Reducing the number of moves required Finally, we optimise the requirement of moving all the qubits to the start of the register by instead checking whether there is room in the register after the first target qubit for the rest of the target qubits, and if there is, move them there starting from the last target qubit. If there isn't, we move the first target qubit back enough steps such that there is just enough room after it for the rest of the target qubits, then move the targets in front of it again. Finally, we check if the first target qubit was moved (it would have only moved back and all the qubits that matter would be in front of it). If it was moved, move it back to where it was and if not, then we're done. We now have all the required qubits adjacent and in their required order. We only now need to prepare the gate by using tensor products with the identity matrices (as described above) and simply apply to the state vector. It is important throughout all the move steps to make sure to also update the array of qubit IDs that the state vector maintains.

This algorithm was implemented as part of the `CircuitOptimiser`, described in the next section.

3.6 The CircuitOptimiser

The idea of parallelising the preparation (matrix expansion) of the SWAP gate in the swapping/-moving algorithm for making qubits adjacent can be extended to an entire given circuit. With prior knowledge of all the quantum gates that make up a circuit, we can prepare all the necessary matrices to execute the circuit in parallel. This is costly in memory, as all the matrices will have to be stored in memory at the same time, but gives way to massive performance improvements. The memory cost is also significantly mitigated by using sparse matrices as described above.

We now describe the steps involved in writing this optimiser. We define a `CircuitOptimiser` class, which simply takes a pointer to a `QRegister` and a list of applicable quantum gates (ApplicableGate). An *applicable quantum gate* is a quantum gate with an associated list of qubit IDs to which it is intended to be applied.

Since we can split any of the circuit gates to a series of SWAP gates required to bring the qubits adjacent and then the actual gate, then we can split the entire circuit into a series of *steps*, where each step is an expanded quantum gate applied to an entire `StateVector` object.

Some Formalism In order to utilise the move algorithm described above, we need the `CircuitOptimiser` to track where each qubit will be at each step. Thus, we formalise this idea of a `Step` to be a list of lists describing the current location of each qubit (much like the `QRegister` stores a list of `StateVector`s which keep a list of qubit IDs) and an `ApplicableGate`. Notice that, because of the rules described in this section, the qubits of each Step are always guranteed to

be adjacent, since we perform all the required moves before the gate application and each move is split into a series of SWAP gate performed on adjacent qubits. Finally, we define an *Op*, in the context of the `CircuitOptimiser`, to be a `StateVectorApplicableGate`, which is an expanded quantum gate coupled with an ID of a state vector to which it is meant to be applied, and a step. In a single run of the `CircuitOptimiser`, there are as many Steps as Ops. Steps are constructed serially and then Ops are constructed from the Steps in parallel.

Determining Required Qubit Moves and Tensor Products The first part of the `CircuitOptimiser` algorithm is the determination of the required tensor products, i.e. computing the Steps. This is done serially as it is not very computationally expensive and it is difficult to determine the number of Steps required. At certain steps, we will need to tell the `QRegister` to combine qubits, so we maintain a dictionary of Step ID keys and arrays of qubit IDs as values. When computing a step that requires qubits to be combined, we add a new entry to this dictionary. This dictionary is then used during the final part of the algorithm, gate application.

Parallelising Gate Preparation Having determined which gates need to be applied and their respective *adjacent* qubits, we can go ahead and compute all the necessary matrices. Because of our preparatory step above, this can be easily parallelised using an OpenMP pragma.

Gate Application Now that all the matrices required to execute the circuit are computed, we can go ahead and apply them to the `QRegister` in series.

A summary of the `CircuitOptimiser` algorithm is provided in Appendix A.

4 Matrix-Free Approach

The largest bottle-neck of the matrix-based approach is that it has to explicitly expand and store very large matrices for application to the state vector. This creates significant load on memory, even when using sparse matrices, as the number of qubit increases. In this chapter, we discuss a different paradigm for gate application that implicitly expands the gate matrix.

In this approach, the gates are still represented as matrices, however they are never expanded through the tensor product to fit a large number of qubits.

4.1 Updating the State Vector without Creating a Large Matrix

Instead of utilising matrix multiplication with a full state vector, in this approach we construct several small state vectors that each correspond to only the number of qubits we are targeting in the main state vector. It should noted that throughout this approach we still use the `StateVector` Expansion optimisation developed for the MBS with a `QRegister` contain. However, we are only concerned with how each `StateVector` applies a gate to its qubits, as `StateVector` combination and measurement is just the same as for MBS (except that measurements have to be reversed in the `StateVector`, as will be explained).

4.1.1 Single Qubit Gate Application

We start by considering the simplest case: single qubit gates. Consider we need to apply a single gate to the tth qubit in a state vector. We need to find all the **pairs of states where each pair has the tth qubit as** 0 **and** 1. This problem can be reduced to finding all the integers (up to the size of the state vector) whose tth bits are 0 and 1. We can find define a function which will find the nth integer whose tth bit is 0 and we can loop over n calling the function on the t. Such a function was taken from Kelly (2018), who implemented a quantum computer simulator for OpenCL, and implemented in C++ for the purposes of MFS, and is shown in Listing 4.1.

```
int nthCleared(int n, int t) {
  int mask = (1 << t) - 1;
  int notMask = ~mask;

  return (n & mask) | ((n & notMask) << 1);
}
```

Listing 4.1: *Function returning the nth integer whose tth bit is zero as per Kelly (2018).*

Then, for each such pair, we apply the gate to it by performing normal matrix multiplication. The full algorithm is shown in Algorithm 1.

Algorithm 1 Matrix-Free Single-Qubit Gate State Vector Update as per Kelly (2018)

for $i \leftarrow 0$ **to** 2^{n-1} **do**
  $a \leftarrow$ the ith integer whose tth bit is 0;
  $b \leftarrow$ the ith integer whose tth bit is 1;
  $v_a \leftarrow v_a \cdot G_{0,0} + v_b \cdot G_{0,1}$;
  $v_b \leftarrow v_a \cdot G_{1,0} + v_b \cdot G_{1,1}$;
end for

4.1.2 Controlled-Single Gate Application

Kelly (2018) also introduces an algorithm for applying a single qubit gate with a control. This is more flexible than using predefined controlled gates (as in MBS) as we simply need to have a gate defined and then we can apply a controlled version of it without needing to define its matrix explicitly. Although in MBS we can easily define a function which will generate a controlled version of a given gate, this is rather wasteful in memory as the matrix representing it could be large depending on the number of target qubits, whereas using this approach does not require constructing such a matrix. For instance, the *CNOT* gate is never explicitly defined in MFS, but rather applied as an X gate applied to some qubit controlled by another qubit.

The algorithm is given here:

Algorithm 2 Matrix-Free Controlled Single-Qubit Gate State Vector Update as per Kelly (2018)

for $i \leftarrow 0$ **to** 2^{n-1} **do**
  $a \leftarrow$ the ith integer whose tth bit is 0;
  $b \leftarrow$ the ith integer whose tth bit is 1;
  $c_a \leftarrow ((1 << c_{id})\&a) > 0$;
  $c_b \leftarrow ((1 << c_{id})\&b) > 0$;
  if c_a **then**
    $v_a \leftarrow v_a \cdot G_{0,0} + v_b \cdot G_{0,1}$;
  end if
  if c_b **then**
    $v_b \leftarrow v_a \cdot G_{1,0} + v_b \cdot G_{1,1}$;
  end if
end for

Expanding to multi-controls We can expand this algorithm to allow for single qubit gate applications controlled by multiple qubits. This will allow for useful gates such as the *TOFFOLI* (doubly controlled *NOT*) to be used without having to decompose them into series of single gates and singly-controlled single gates, which is rather tedious. This is simply done by doing the control step in the above algorithm multiple times over all the control qubits.

In C++, this is implemented as:

```
for(int i = 0; i < pow(2,n-1); i++) {
   int zero_state = nthCleared(i, qIndex);
   int one_state = zero_state | (1 << qIndex);

   cxd zero_amp = amplitudes[zero_state];
   cxd one_amp = amplitudes[one_state];
```

```
    bool control_zero = true;
    bool control_one = true;

    for(int c = 0; c < controlIndices.size(); c++) {
      if(! (((1 << controlIndices[c]) & zero_state) > 0)) control_zero = false;
      if(! (((1 << controlIndices[c]) & one_state) > 0)) control_one = false;

      if(!control_zero && !control_one) break;
    }

    if(control_zero) amplitudes[zero_state] = gate[0]*zero_amp + gate[1]*one_amp;
    if(control_one) amplitudes[one_state] = gate[2]*zero_amp + gate[3]*one_amp;
}
```

Listing 4.2: *Multi-Controlled Single Qubit Gate Application. An extension of the algorithm by Kelly (2018).*

4.1.3 Multi-Qubit Gate Application

Finally, we can expand the original algorithm to allow for gates which operate on more that one qubit. Whereas this is not necessarily very important as all multi-qubit gates can be decomposed into series of single qubit gates and CNOTs, it does offer increased convenience. This will allow us for instance to easily apply SWAP gates if required as well as quantum oracles for e.g. Deutsch's Algorithm (covered in the next chapter). This requires us to introduce some new helper functions:

```
int nnthCleared(int n, vector<int> ts) {
  int mask = (1 << 20) - 1;

  for(int i = 0; i < ts.size(); i++)
    mask &= ~(1 << ts[i]);
  int notMask = ~mask;

  return (n & mask) | ((n & notMask) << ts.size());
}

int nthInSequence(int n, vector<int> ts, int s) {
  if(s >= pow(2,ts.size())) throw "Unapplicable";

  int f = nnthCleared(n, ts);

  for(int i = 0; i < ts.size(); i++) {
    if(!(s & (1<<i))) {
      f |= 1 << ts[i];
    }
  }

  return f;
}
```

Listing 4.3: *Helper functions for MFS Multi-Qubit Gate Application.*

We then use this `nthInSequence()` to retrieve all the required qubit indices for the matrix application. The `nnthCleared()` function is the multi-qubit equivalent of the above `nthCleared()` function. `nthInSequence()` cycles through all the required qubit indices for a given matrix multiplication when passed different values of `s`. The full multi-qubit gate application code is then:

```
void StateVector::applyMultiGate(vector<QID> qIDs, Gate gate) {
 vector<int> qIndices;
 qIndices.reserve(qIDs.size());

 for(int i = 0; i < qIDs.size(); i++) {
  qIndices.push_back(distance(this->qIDs, find(this->qIDs, this->qIDs+n,
      qIDs[i])));
 }

 int N = pow(2,qIDs.size()); // size of gate

 for(int i = 0; i < pow(2,n-qIDs.size()); i++) {
  vector<int> states;
  vector<cxd> amps;

  for(int j = 0; j < N; j++) {
   int state = nthInSequence(i, qIndices, j);
   states.push_back(state);
   amps.push_back(amplitudes[state]);
  }

  for(int i = 0; i < N; i++) {
   cxd sum = 0;
   for(int j = 0; j < N; j++) {
    sum += gate[N*i+j]*amps[j];
   }
   amplitudes[states[i]] = sum;
  }
 }
}
```

Listing 4.4: *Multi-qubit MFS gate application.*

4.1.4 OpenMP Parallelisation of MFS

As described in Algorithm 1, to update the state vector in this approach, we loop over each pair of relevant probability amplitudes and update them. Since all the pairs are mutually exclusive (no two pairs share the same probability amplitude index), parallelising this algorithm is relatively straightforward. Much like parallelising the matrix operations for MBS, all that is necessary here is an OpenMP pragma: `#pragma omp parallel for` to parallelise this algorithm across as many threads as are available. The same applies to the controlled single-qubit gate application in Algorithm 2.

5 Quantum Algorithms

In this chapter, we discuss some quantum computing algorithms and their implementations on the developed simulator. Also discussed are some quantum error correction codes.

5.1 Deutsch's Algorithm

This algorithm, developed by Deutsch (1985), was the first example to demonstrate quantum supremacy through parallelism, by solving a specific problem exponentially faster than any classical algorithm.

The problem in consideration is that of finding whether a function $f(x)$, where $f : \{0, 1\} \to \{0, 1\}$, is constant or balanced. Such a function is constant if $f(0) = f(1)$ and is balanced if $f(0) \neq f(1)$. Classically, in order to determine this, we need to evaluate the function twice, once with $x = 0$ and another with $x = 1$. This is because we could, e.g. evaluate $f(0)$ to be 0; this would still leave the possibility of f being either constant or balanced: it is balanced if $f(1)$ is then evaluated to be 1 and constant if $f(1)$ is evaluated to be 0. Deutsch proposed that if we could find a quantum version of f, we would only need to evaluate the function once on a superposition of states to determine whether it is constant or balanced. This quantum version of f, U_f, cannot only operate on a single qubit $|x\rangle$ such that $U_f |x\rangle = |f(x)\rangle$, because a quantum operator must be unitary and thus invertible. But if $f(x)$ is constant, then it is not invertible. And so we construct U_f to act on two qubits, $|x\rangle$ and $|y\rangle$, such that:

$$U_f |x\rangle |y\rangle = |x\rangle |f(x) \oplus y\rangle$$

Deustch suggested that, once we have the operator U_f, we can apply the circuit in Figure 5.1, measure the first qubit and find out whether $f(0) = f(1)$ or not. If $|x\rangle$ was measured to be $|0\rangle$, the function f was constant, otherwise if it was measured to be $|1\rangle$, the function f was balanced.

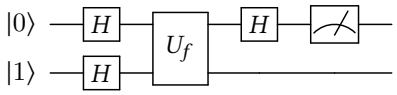


Figure 5.1: *Quantum circuit for Deustch's algorithm.*

We notice here that, we start by applying the H gate to both qubits, putting them in a superposition of all four possible states. This is what allows us to exploit quantum parallelism. We only needed to evaluate the U_f gate here once to find the answer we need, whereas classically we would have needed two evaluations of the function f.

5.1.1 Quantum Oracles

In order to implement this algorithm in the simulator however, it is needed to explicitly know U_f in advance. Generally, such an operator is known as a *quantum oracle*. This term is used in the context of quantum computing to be analogous to a classical blackbox. Since, for the purpose of the problem, no information about f is known until we evaluate it with some value, then it is appropriate to say the same about U_f, assuming we have a way of constructing U_f directly from f. Hence, f is a *blackbox* and U_f is its corresponding quantum oracle.

There are four possibilities for f to be in, which correspond to the following four quantum oracles:

$$U_{f00} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, U_{f01} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, U_{f10} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, U_{f11} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Implementing this circuit in the developed simulators becomes straightforward once we know the required oracle.

5.1.2 Deutsch-Jozsa Algorithm

The Deutsch-Jozsa algorithm is the generalisation of Deutsch's algorithm for functions of n bits. Developed by Deutsch and Jozsa (1992), it allows us to determine whether a function $f : \{0,1\}^n \to \{0,1\}$ is constant or balanced. Such a function would have to be evaluated $2^{n-1} + 1$ times classically to determine this, whereas a quantum oracle equivalent of f would need to be evaluated once. This shows an exponential speedup over classical algorithms, and was one of the first demonstrations of quantum supremacy.

The circuit for the Deutsch-Jozsa algorithm is a natural extension of Deutsch's algorithm and is shown in Figure 5.2.

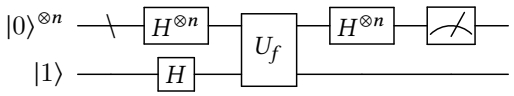


Figure 5.2: *Quantum circuit for the Deutsch-Josza algorithm. In this figure, the backslash across the first wire means that this represents multiple qubits, n qubits to be exact.*

5.2 Quantum Teleportation

Since cloning a quantum state is forbidden (Wootters and Zurek 1982), the closest we can do is *quantum teleportation*. It possible to teleport an arbitrary quantum state from one qubit to another by using an entangled pair of qubits.

Consider we have some quantum state, $|\psi\rangle$, encoded in a qubit. We also have two other qubits, $|x\rangle$ and $|y\rangle$ that are entangled, such that:

$$|xy\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

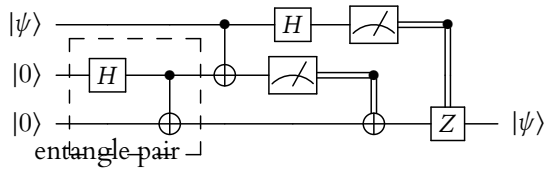


Figure 5.3: *Quantum circuit for single qubit teleportation.*

The state $|\psi\rangle$ is some general state: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, such that the combined state is:

$$|\psi xy\rangle = \frac{1}{\sqrt{2}}(\alpha|0\rangle + \beta|1\rangle)(|00\rangle + |11\rangle) = \frac{1}{\sqrt{2}}(\alpha|000\rangle + \alpha|011\rangle + \beta|100\rangle + \beta|111\rangle)$$

In Figure 5.3, this is the state as it occurs after the "entangle pair" step. The next two gates then encode the quantum state into the entangled pair. The final part of the circuit decodes the state and stores it in the qubit, $|y\rangle$, by performing two classically-controlled gates on the qubit. The classically-controlled gates are demonstrated by the double-lines in the figure. We demonstrate what happens by applying the gates to the state. Applying *CNOT* to the first and second qubit (i.e. flipping the second qubit in all possible states only if the first qubit of the state is $|1\rangle$):

$$|\psi xy\rangle = \frac{1}{\sqrt{2}}(\alpha|000\rangle + \alpha|011\rangle + \beta|110\rangle + \beta|101\rangle)$$

Now applying H to the first qubit, i.e. replacing any $|0\rangle$ in the first qubit with $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and any $|1\rangle$ with $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$:

$$|\psi xy\rangle = \frac{1}{\sqrt{2}}(\alpha|+00\rangle + \alpha|+11\rangle + \beta|-10\rangle + \beta|-01\rangle)$$

$$= \frac{1}{2}(\alpha|000\rangle + \alpha|100\rangle + \alpha|011\rangle + \alpha|111\rangle + \beta|010\rangle - \beta|110\rangle + \beta|001\rangle - \beta|101\rangle)$$

We now measure the first and second qubits, transmit them classically to the location of the third qubit, and conditionally apply *NOT* and *Z* to the third qubit according to their measured values, respectively. The application of *NOT* is simply represented by flipping the qubit's value in each possible state. The application of *Z* is represented by flipping the sign of the state if the qubit's value is $|1\rangle$ in the state, for each possible state.

We have four cases:

$$|00\rangle \rightarrow \text{apply nothing} \rightarrow |00\rangle(\alpha|0\rangle + \beta|1\rangle)$$
$$|01\rangle \rightarrow \text{apply X} \rightarrow |01\rangle(\alpha|0\rangle + \beta|1\rangle)$$
$$|10\rangle \rightarrow \text{apply Z} \rightarrow |10\rangle(\alpha|0\rangle + \beta|1\rangle)$$
$$|11\rangle \rightarrow \text{apply X and Z} \rightarrow |11\rangle(\alpha|0\rangle + \beta|1\rangle)$$

In any of the cases, we see that the final qubit has become the state $|\psi\rangle$, as it was in the first qubit at the start of the teleportation algorithm. We notice that the first two qubits can exist in any of the four possible states that two qubits can take. This is because, in order to teleport a quantum state, the initial state has to be effectively destroyed, i.e. in its most random state.

Because the simulator was developed to only allow measurements on the entire register at any instant, simulating this circuit directly is not possible. So we have to adjust it so as to "simulate" the

measurement outcomes, as per Figure 5.4. This allows us to simulate the teleportation algorithm on either of the developed simulators.

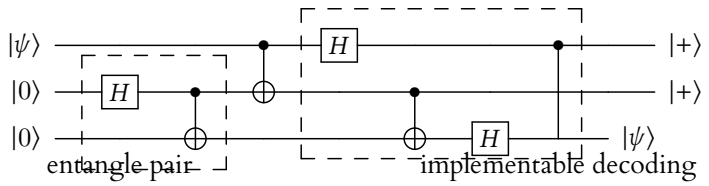


Figure 5.4: *Implementable quantum circuit for single qubit teleportation.*

5.3 Quantum Fourier Transform

The *Quantum Fourier Transform* (QFT) (Nielsen and Chuang 2010, p. 216) is a crucially important operation in many quantum algorithms. The QFT is the quantum analogue of the classical discrete Fourier transform, which converts a set of complex-valued samples of a function into a complex-valued sample of the Fourier transform of that function, i.e. in the frequency domain. Similarly, the QFT converts qubits into the phase domain. <more>

Classicaly, a set of function samples, x_n, of length N, is converted into the frequency domain of the function by applying the DFT as follows:

$$y_k = \sum_{n=0}^{N-1} x_n e^{-\frac{i2\pi}{N}kn}$$

However, by convention, the QFT has the same effect as the *inverse DFT*, so we need to use:

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} y_k e^{\frac{i2\pi}{N}kn}$$

Phase-Shift Gates We now introduce *phase-shift gates*, $R(\phi)$:

$$R(\phi) \equiv \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix} \tag{5.1}$$

For the purposes of the QFT however, it is more convenient to define the gate R_m, as:

$$R_m = \begin{bmatrix} 1 & 0 \\ 0 & \omega_m \end{bmatrix},$$

where ω_m is the 2^m-th root of unity: $\omega_m \equiv e^{\frac{2\pi i}{2^m}}$. This is equivalent to an application of $R(\phi)$ with $\phi = \frac{2\pi}{2^m}$:

$$R_m \equiv R(\frac{2\pi}{2^m}) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{2\pi}{2^m}} \end{bmatrix}$$

The QFT is then implemented as a series of repeated Hadamard gates and controlled phase-shift gates, as in Figure 5.5.

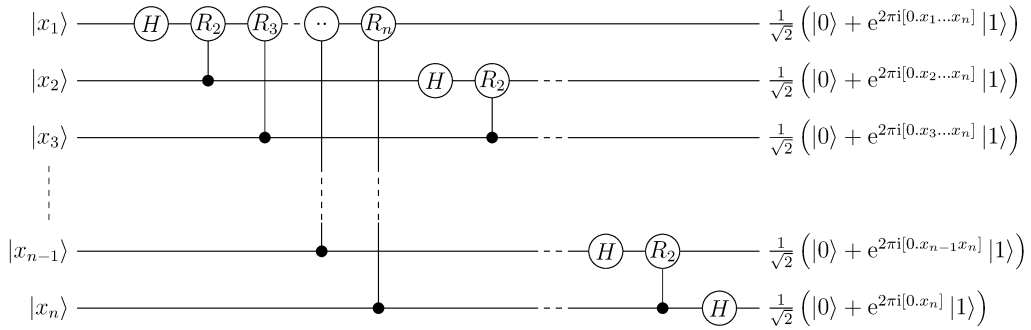


Figure 5.5: *Quantum Fourier Transform on n qubits. Image from Wikipedia, the free encyclopedia (2019)*

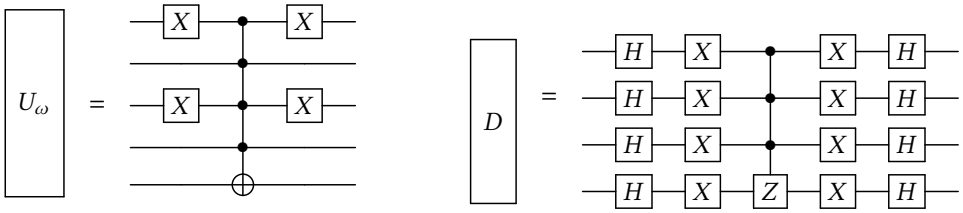
5.4 Grover's Search Algorithm

Grover's search algorithm (Grover 1996) was one of the first to be developed which could be used to perform a practical task on a quantum computer significantly faster than a classical algorithm. In essence, this allows us to perform a database search in $O(\sqrt{N})$ steps; a quadratic speedup compared to the classical equivalent taking $O(N)$ steps. The algorithm operates by iteratively amplifying the probability of the required item to be found while lowering the probability of all others, until the required probability is at its maximum possible value.

Suppose we need to find the index of an item in a database which satisfies some condition. Classically we can represent this by a function $f : \{0,1\}^n \rightarrow \{0,1\}$ which only returns 1 when the correct index of a state is passed. We are then essentially looking for the state $|\omega\rangle$ such that $f(\omega) = \omega$. Similar to the Deustch-Jozsa algorithm, we need to find a quantum oracle equivalent to the function f. Again using one ancillary qubit, we can write:

$$U_\omega |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle$$

This has the effect of negating the probability amplitude of the state which we desire. Such a negation however has no effect on the probability of this state. We now need to apply the *Grover diffusion* operator, which will reflect the negated probability about its original value, thus amplifying it. Throughout the rest of this section, we take the example of performing a search on four qubits (with an additional ancillary qubit), where we are looking for the state where e.g. the first and third qubits are on. We define the quantum oracle, U_ω as shown in Figure 5.6a.



(a) *Grover's oracle for a four qubit search, where the first and third qubits are on.*

(b) *Grover's diffusion operator for a four qubit search.*

Figure 5.6: *Operators required for Grover's search algorithm.*

From Figure 5.6a, we see that the oracle can be implemented in the form of applying the X gate to the target qubits, performing a multi-control controlled gate across the entire register as shown, and finally applying X to the target gate again.

The Grover diffusion operator does not depend on the target qubit and is, as such, the same for any given number of qubits. It also does not need access to the ancillary qubit as it only concerns the database items. For the four qubits example as above, the diffusion operator is shown in Figure 5.6b.

By applying U_ω then D repeatedly, the probability of the desired state is amplified, such that when a measurement is made, it is highly likely that we have the correct result. We do however need to exploit quantum parallelism in the first place, and so the first step is to apply H to all the qubits, to put the system into its maximally-mixed state. Figure 5.7 then shows the final Grover's search circuit. A visualisation of the application of U_ω followed by D is shown in Figure 5.8, from (Nielsen and Chuang 2010, p. 253).

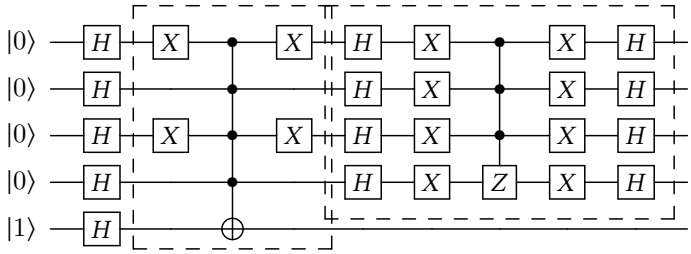


Figure 5.7: *Grover's full search algorithm for a four qubit search, where the first and third qubits are on.*

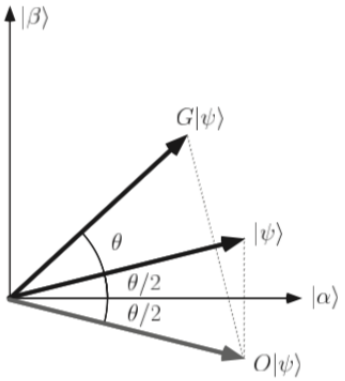


Figure 5.8: *Visualisation of Grover's search from Nielsen and Chuang (2010, p. 253). Here, the operator O is the Grover oracle (U_ω above) and G is Grover's diffusion operator (D above).*

5.5 Quantum Adders

We now discuss some quantum addition circuits. We start by discussing quantum analogues to the half adder and the full adder and then move on to discuss two different implementations of ripple carry adders.

5.5.1 Cuccaro Ripple-Carry Adder

Developed by Cuccaro et al. (2004), this was the first quantum ripple-carry adder, which does not use the QFT, to require only a single ancillary qubit. Such an adder uses $2 \times n + 2$ qubits for

adding two n-bit numbers. A 6-bit version of this adder is shown in Figure 5.9. This circuit was extended to allow any number of bits as input. This algorithm does the addition in place, i.e. the sum replaces one of the input bit strings.

A function was developed which takes two bit strings, runs them through the adder and outputs their sum. It is called like so:

```
cout << NBitCuccaroAdder("1101111","1110011") << endl;
```

Listing 5.1: *Calling the n-bit Cuccaro adder function. This outputs the result: 11100010.*

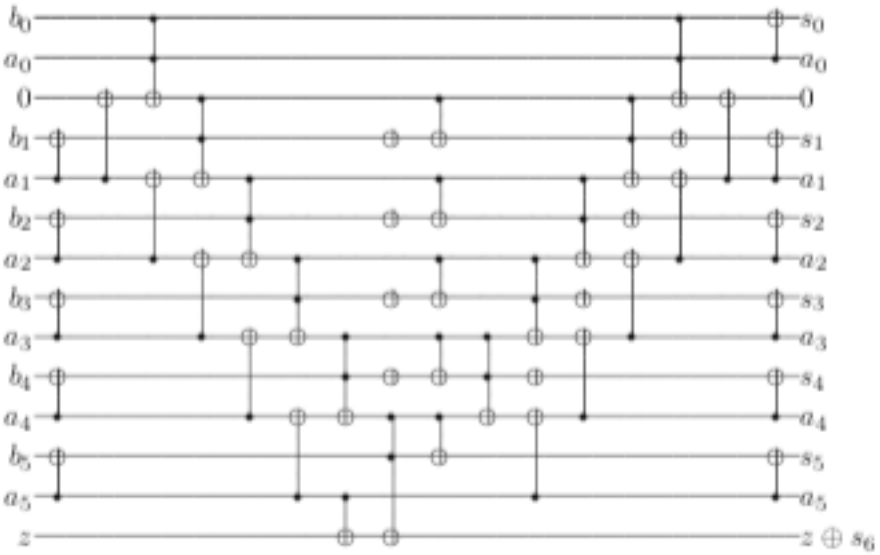


Figure 5.9: *6 Bit Cuccaro Ripple Carry Adder (Cuccaro et al. 2004).*

5.5.2 QFT Adder

In a paper about arithmetic on a quantum computer, Ruiz-Perez and Garcia-Escartin (2017) describe a quantum adder that utilises the phase domain (via the QFT) to perform the addition. This uses $2 \times n + 1$ qubits and requires no ancillary qubits. The QFT adder does the addition in place, like the Cuccaro adder. One of the input bit strings is first put into the phase domain by applying the QFT to it. We then apply a series of controlled phase shift gates (much like in the QFT itself) with the second input bit strings as the controls to perform the addition. The result is the sum of the inputs in the phase domain. This is shown in Figure 5.10. We retrieve the final result by applying the inverse QFT. A function for calling this circuit directly with two bit strings to sum, similar to the one provided for the Cuccaro adder, is also provided for this adder.

5.6 Quantum Error Correction

Much like classical bits, quantum bits are prone to random errors due to interference with the environment. In this section, we investigate three quantum error correction codes and we

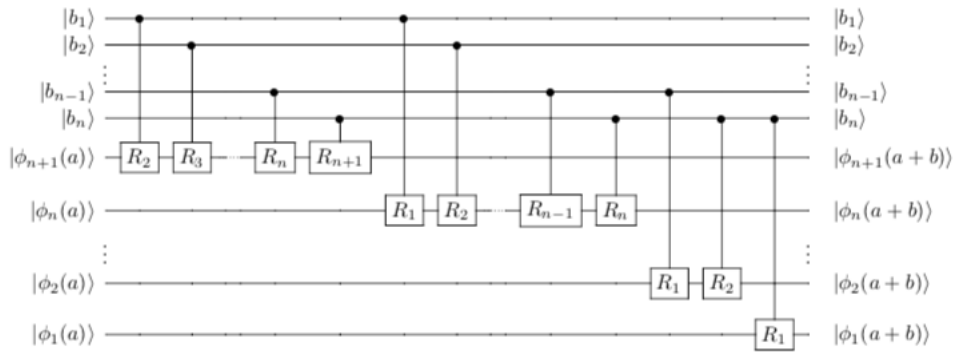
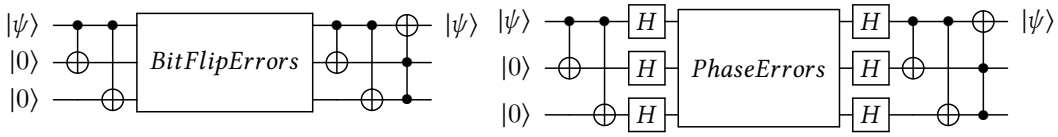


Figure 5.10: *Quantum addition using the QFT. From Ruiz-Perez and Garcia-Escartin (2017).*



(a) *Bit flip quantum error correction code circuit.*

(b) *Phase flip quantum error correction code circuit.*

Figure 5.11: *Bit flip and phase flip QEC codes.*

simulate a environment where qubits are prone to errors after being encoded. A useful reference for the content discussed in this chapter is the tutorial by Steane.

5.6.1 Bit Flip Code

The *bit flip code* is designed to protect qubits against bit flip errors, i.e. errors that would change a $|0\rangle$ state to a $|1\rangle$ state and vice versa. Encoding a state requires two additional qubits and is done as shown in Figure 5.11a. The bit flip errors can simply be simulated by randomised applications of the X gate.

5.6.2 Phase Flip Code

The *phase flip code* protects qubits against errors which would change a $|+\rangle$ state to a $|-\rangle$ state, and vice versa. Such errors can be simulated by randomised applications of the Z gate. Similar to the bit flip code, the phase flip code also requires two additional qubits, and is accomplished as shown in Figure 5.11b.

5.6.3 Shor Code

The *Shor code* (Shor 1995) was one of the first QEC codes to fully protect qubits against environmental errors. This is accomplished by combining the bit flip and phase flip codes and requires nine qubits to encode one qubit. A phase flip code is first applied across the first, fourth, and seventh qubits and then three bit flip codes across the first, second, and third blocks of qubits. Decoding happens in reverse; decoding the bit flip codes first then the phase flip code. This is

shown in 5.12. This code can be used to effectively protect a qubit against any environmental errors.

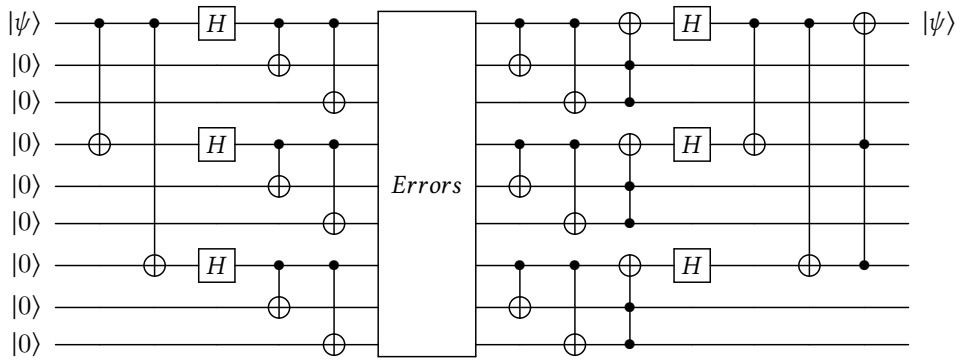


Figure 5.12: *Shor code for full quantum error correction, combining the bit flip and phase flip codes.*

5.6.4 Preserving entanglement using the Shor code

To demonstrate the use of the Shor code to protect qubits from both bit flip and phase flip errors, a simulation was conducted using 18 qubits. The goal is to entangle the first and the tenth qubits and then use the remaining qubits to encode them. We then apply any set of random error gates to them, to simulate environmental noise. Finally, we decode them, take a series of measurements and ensure that the first and the tenth qubits are still entangled. The code for this experiment is provided in Appendix B.

6 Quantum Language Interpreter

The Quantum Language Interpreter (QLI) was developed on top of the simulator to allow for easy access to the simulator without having significant knowledge of its C++ implementation. In this chapter, the development of this interpreter is described and implementation of some quantum algorithms are given in the developed language.

6.1 Language Elements

Initialising a quantum register The first (non-comment) line in a QLI program has to be one which utilises the `init` keyword to initialise a quantum register with a given number of qubits. For example, `init 5` will initialise a quantum register with 5 qubits all in a state of $|0\rangle$.

Expressions An expression in QLI is similar to an integer expression in other languages. These expressions are mainly used as qubit identifiers and loop parameters. Examples of expressions in QLI include `3+4*2` and `3*(n+2)`, given that `n` is defined.

Executing a Gate QLI comes with a set of predefined gates including the most common quantum gates introduced in previous sections. In order to run a gate, it is enough to simply enter the gate's name followed by the location (ID(s)) of the target qubit(s). For instance, performing the *H* on the second gate in a register would look like: `H 1`.

For added convenience, we use the keyword `all` to refer to all the qubits in a register. For instance, given a two qubit register, in order to apply CNOT with the first qubit as the control, we could either write `CNOT 0 1` or `CNOT all`. This convenience becomes of importance when the user defines gates with a large number of inputs.

Taking Measurements The result of a QLI program will always be the measurement of some quantum register. We can specify which qubits to measure using the `return` keyword followed by the IDs of the qubits to measure. We can also use `all` here. Additionally, we can use the `take` keyword followed by a number of measurements which utilises the `Accumulator` to take multiple measurements on the system.

We now provide the first example of a complete QLI program:

```
init 2 -- Initialise the quantum register with two qubits
H 0 -- Put the first qubit in a superposition
CNOT 0 1 -- use CNOT to entangle the qubits
take 1000 -- take 1000 measurements on the system and print out the results
```

Listing 6.1: *QLI program to entangle two qubits and take 1000 measurements on the register.*

Naming a Qubit This is the equivalent of defining a variable or a constant in other languages. Using the `let` keyword followed by a variable name and an expression will create a new variable and initialise it with the given expression. As described above, the expression itself can be in terms of other variables so we can create variable based on other variables. This is useful for instance when using the Cuccaro adder when it is more convenient to define qubits in terms of what they represent in the algorithms (the two input bit strings in the case of adders).

Defining a Gate Gate definition are equivalent to function definitions in other languages: it allows users to define their own gates which they can reuse essentially as black boxes multiple times in a circuit. To define a new gate, the `gate` keyword is used followed by the name of the gate and a block of gate calls. Finally, to terminate the gate, the `endgate` keyword is used. Gates with a specific number of input qubits also provide that number after the gate's name. Otherwise, no number is provided. The qubit identifiers used in a gate are not global qubit IDs, but rather the indices in the passed list of qubits. For instance, in the next example, we define the `entangle2` gate which does the same as the circuit defined above and call it with qubit IDs. This will have the effect of putting the second qubit in a superposition then using *CNOT* with it as the control. Effectively, this makes no difference in terms of entangling the state.

```
init 2 -- Initialise the quantum register with two qubits
gate entangle2 2 -- entangle2 gate for two-qubit entanglement
   H 0
   CNOT all
endgate
entangle2 1 0 -- perform entangle2 on the register, qubits reversed
take 1000
```

Listing 6.2: *Defining a gate for entangling two qubits.*

A dynamic gate is one which takes a variable number of qubits. For the purposes of the gate, the number of arguments is provided in an `argc` variable inside the gate. We will describe an example gate which entangles an arbitrary number of qubits utilising a dynamic gate after loops are introduced.

Looping over Qubits We can define a set of gates to be run multiple times by utilising loops. These are C-style for loops. They are used through the `loop` keyword, which is followed by three required parameters and optionally one extra parameter. The first parameter is the name of the loop parameter (typically `i` in C-style loops). The second and third parameter are the start and end values of the loop parameter, *inclusive*. The optional fourth parameter is the amount to change the loop parameter each iteration. When the fourth parameter is not given, the loop parameter is simply increased by one each iteration. This allows for a very succinct syntax for the purpose of simulating circuits with elements that repeat in complex ways, such as the Cuccaro adder. Finally, the loop code block is terminated with the `endloop` keyword.

An example of loops is provided here, where we utilise a loop to put a register of 10 qubits into a superposition with all states equally probable:

```
init 10
loop i 0 9
   H i
endloop
take 1000
```

Listing 6.3: *Applying H gate to 10 qubits using a QLI loop.*

Finally, we give an example which utilises both loops and dynamic gates to define a gate which entangles an arbitrary number of qubits.

```
init 10
gate entangle
   let n argc -- For convenience, let n be the number of arguments
   H 0
   loop i 0 n-1 -- loop over all the input qubits
      CNOT 0 i
   endloop
endgate
entangle all -- call the entangle gate to entangle all 10 qubits
take 1000
```

Listing 6.4: *Defining a dynamic gate to entangle an arbitrary number of qubits.*

QLL: Quantum Linked Library This is the equivalent of import statements in other languages. Using `.qll` files allows for encapsulating gates in different files to reuse across multiple files. In QLI, `.qli` files are executables and `.qll` can be linked to `.qli` files or other `.qll`s. This is done using the `link` keyword followed by the file name without the extension. The linked files must have the `.qll` extension and must be in the lookup directory of the interpreter.

6.2 Implementing the Language

Due to the time constraints of the project, the developed interpreter does not properly utilise standards of programming language interpretation (e.g. stages of tokenisation, lexing, parsing, etc.), but rather processes the given program line by line. In addition, parsing expressions was not implemented from scratch but rather utilises the `?` C library for evaluating mathematical expressions. Implementation of integer variables (qubit naming) is done using a simple lookup table implemented as a C++ map.

When a variable is defined, it is available to use for its entire gate scope (or in the main circuit). When a variable is referenced in an expression, its name is simply replaced by its value using string operations, then the expression is passed through the tinyexpr parser. User-defined gates are implemented by a map from gate name to an array of strings which represents the lines of the gate. This is required because dynamic gates will require the replacement of their `argc` variable when the gate is called depending on the number of qubits. A loop is handled by first going through it and storing all its lines, then constructing the equivalent C-style loop, and handling all the lines recursively in the loop. Finally, QLL links are handled by going through the entire `.qll` file and handling all its gate definitions as if they were gate definitions in the main file.

Utilising the CircuitOptimiser In the MBS, the implemented QLI utilises the most efficient performance optimisation, the `CircuitOptimiser`. This is achieved by creating a list of all the gate calls that the QLI program makes then passing them through the `CircuitOptimiser`.

6.3 Implementing Quantum Algorithms in QLI

To demonstrate the usefulness of QLI, we implemented a selection of quantum algorithms in it and shown that it works as required. These implementations are provided in Appendix C.

7 Evaluation

In this chapter, we evaluate the performance of each layer of optimisation of the MBS compared to a baseline. We also evaluate the performance of MFS with respect to the version of MBS with the most optimisations. Finally, we compare the performance of MFS against two third party simulators, QX Studio and libquantum.

All time measurements in this chapter were taken on a mid-2014 MacBook Pro with an Intel Core i7-4870HQ processor clocked at 2.50GHz with 4 cores and 16GB of 1600MHz DDR3 memory. To measure the time a block of code runs in C++, we use the `chrono::high_resolution_clock::now()` function, as shown in Listing 7.1. libquantum was run via the terminal and so using `time` was appropriate to time it. QX Studio displays the time taken to run automatically, which was convenient.

```
auto start = chrono::high_resolution_clock::now();
   <code to measure here>
auto end = chrono::high_resolution_clock::now();
auto diff = end - start;
cout << "--- end time: " << (chrono::duration <double, milli>
    (diff).count())/1000 << " s" << endl;
```

Listing 7.1: *Measuring time in C++.*

7.1 Evaluation strategy

MBS In order to see the effect of each applied performance optimisation to the MBS, we evaluate against three different algorithms: 12 qubit QFT, 6 bit Cuccaro adder (which uses 14 qubits), and 8 bit Cuccaro adder (which uses 16 qubits). In order to be able to simulate these feasibly, we choose our baseline to be the simulator after the `StateVector` and sparse matrix memory optimisations have been applied. This leaves us with four levels of optimisations to compare:

- Baseline (StateVector and Sparse Matrix Optimisations)
- + Tensor Product
- + Swap/Move Optimisation
- + `CircuitOptimiser`

MFS vs. MBS We now compare MFS with the MBS. Again we chose three algorithms to target: 6 bit Cuccaro adder, 8 bit Cuccaro adder, and 15 qubit QFT. Our benchmarking targets are:

- MBS + CircuitOptimiser
- Python MFS
- C++ MFS

MFS vs. Third Party Simulators Finally we compare the MFS with the reviewed third party simulators, QX Studio and libQuantum. For this we chose the 15 qubit QFT and the 20 qubit QFT for comparison against QX Studio, and the 10 qubit Grover's search and the 15 qubit Grover's search for comparison against libquantum. The targets for this set of evaluations are:

- C++ MFS vs. QX Studio
- C++ MFS vs. libQuantum

To evaluate QLI, we will compare its language features with QX Studio's Quantum Code.

7.2 Results

For each target, as many time measurements were made as was possible and they were averaged to help reduce random errors if we were to make just one measurement. The graphs in this chapter were made by deciding a baseline target against which to compare in each step, then normalising the time with respect to that target to find the relative performance. All the data collected can be found in Appendix D.

Note regarding the graphs in this section: to save space, we combined some algorithms into the same graph using multi-bar plots. It is important to realise here that the proportionality is only valid for each specific quantum algorithm with itself, e.g. the tallest green bar in Figure 7.1 is 72.7× the shortest green bar, but is not 72.7× the shortest blue or orange bars.

7.2.1 Evaluating MBS Optimisations

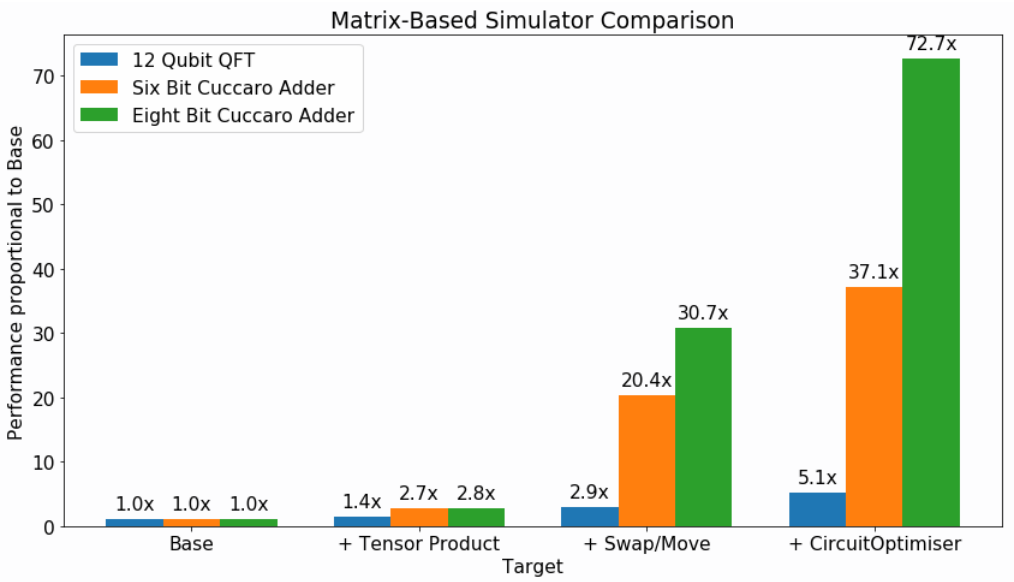


Figure 7.1: *MBS optimisations evaluation results.*

Figure 7.1 shows the results obtained for comparing the optimisations developed for the matrix-based simulator. Starting with the algorithm with the lowest number of qubits (12 Qubit QFT), we can see that the all the optimisations applied result in a 5.1× performance improvement that the base. It can also be seen that as we test on algorithm with a more qubits, this improvement grows exponentially (the 6-bit adder uses 14 qubits and the 8-bit adder uses 16 qubits).

7.2.2 MBS vs. MFS

We now compare the developed matrix-free simulator with the most optimised version of the matrix-based one. Figure 7.2a shows the result of running the Cuccaro adders with these targets. We can see that MFS easily beats MBS here with up to an 11.0× performance improvement.

The 15 qubit QFT results shown in Figure 7.2b show a dramatic performance improvement over MBS. This is interesting because, while there are fewer qubits than there are in the 8-bit adder, there are far more gates in the 15 qubit QFT and yet the QFT performance improvement is two orders of magnitude greater the adder's improvement. This shows that, not only does MFS handle a larger number of qubits more efficiently but also that it handles a larger number of gates far better than MBS.

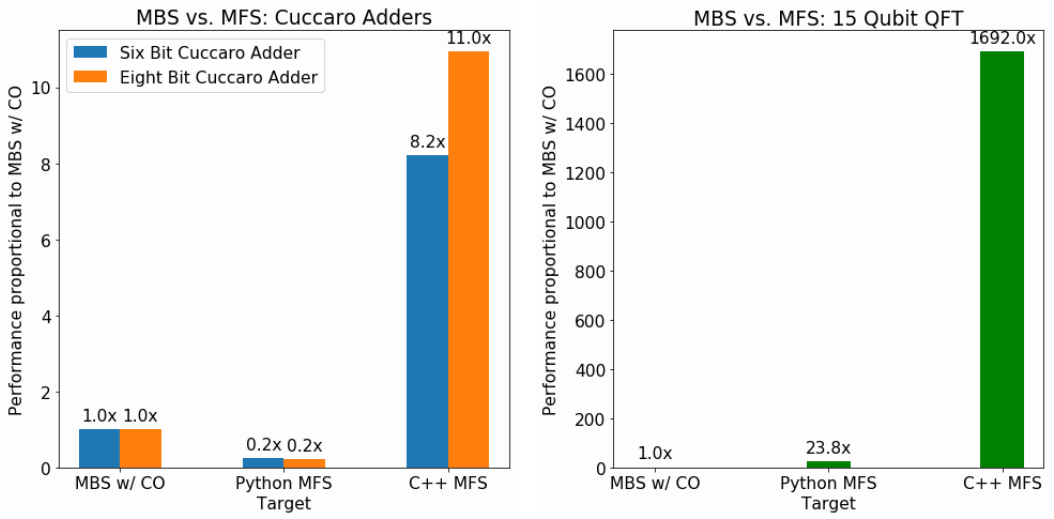


Figure 7.2: MBS vs. MFS evaluation results.

7.2.3 MFS vs. Third Party Simulators

Finally, we compare the matrix-free simulator with the two third party simulators in question. Figure 7.3a shows the evaluation results for MFS vs. QX Studio for the 15 and 20 qubit QFTs. The graph shows that QX Studio still beats the MFS by up to 30.2× for the selected algorithms. Figure 7.2b shows that libquantum outperforms MFS by up to 3.6× for the 15 qubit Grover's search.

Whilst these results show that there remains some work to be done to bring the developed matrix-free simulator up to the same standard as existing work, they are not entirely unexpected as significantly more time was spent in the development of these simulators than the MFS. It is a rather positive result that QX Studio outperforms MFS for the selected algorithms only by up one order of magnitude, and that libquantum does not outperform it by more than 3.6×.

7.2.4 QLI vs. Quantum Code

We perform a qualitative comparison between the developed QLI and QX Studio's Quantum Code. Upon comparing the two, it was found that they are very similar in syntax and functionality

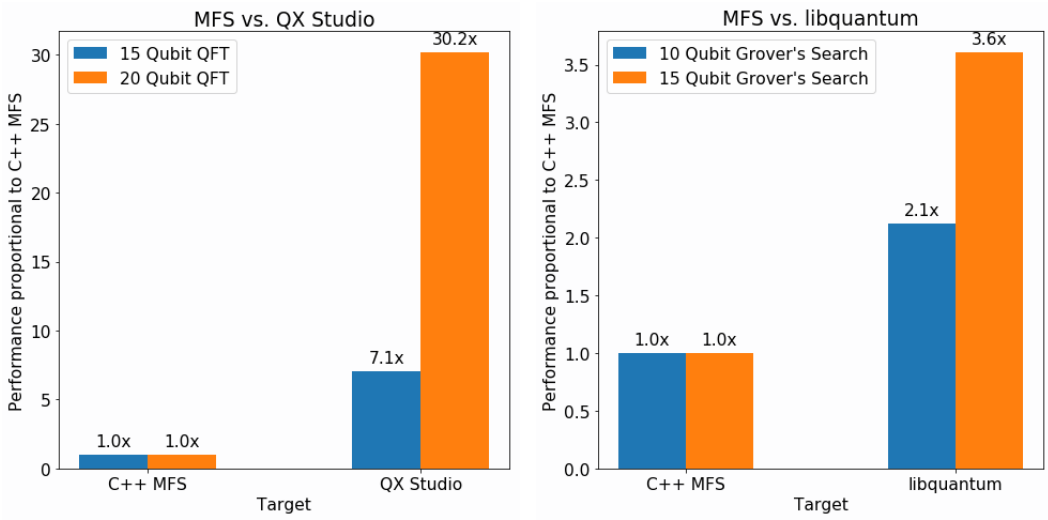


Figure 7.3: *MBS vs. third party evaluation results.*

with slight differences. Most importantly, it seems that Quantum Code does not allow C-style for loops for looping over some sub-circuit with changing loop parameters. Figure 7.4 shows the type of loops permitted by Quantum Code. Only a sort of global repetition over a sub-circuit with fixed qubit IDs is allowed. Additionally, Quantum Code does not have functionality for defining and reusing quantum gates other than in the context of looping. These two points make QLI somewhat more flexible than Quantum Code.

However, Quantum Code has the edge in other aspects. It is more developed that QLI with more predefined gates and better error messages than QLI. It also has a more flexible measuring paradigm allowing measurements to be taken of individual qubits and stored into a binary bit. This binary bit can then be used as a control for a quantum gate. Quantum Code also has the feature of simulating quantum errors implicitly if required. Finally, Quantum Code allows the user to specify sets of gates to be run in parallel.

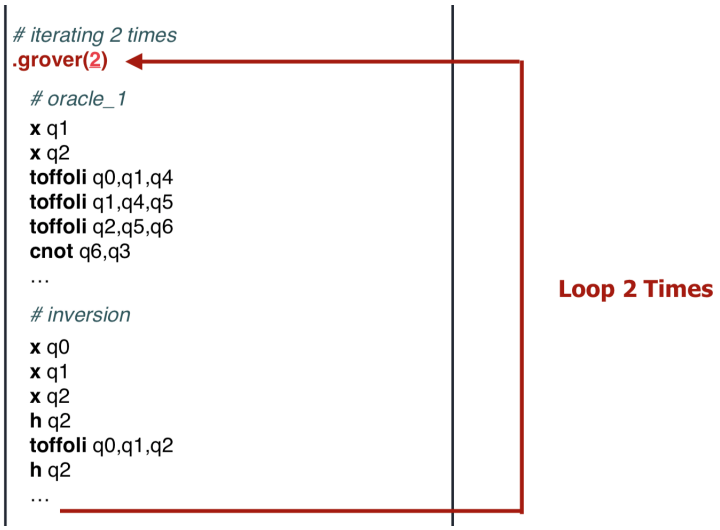


Figure 7.4: *Looping in Quantum Code. From*

8 Conclusion

We conclude this paper with a brief summary of the optimisations developed for the simulators, the quantum algorithms implemented, and the results found.

This project looked at the development of two types of quantum computing simulators: a matrix-based one, which directly simulates the mathematical formalism behind quantum computing, and a matrix-free one, which cleverly mitigates a major bottleneck of the matrix-based approach. We also looked at implementing several quantum algorithms on the developed simulators. In addition, we built an interpreter for a quantum language (QLI) and showed how some of the quantum algorithms could be implemented on it. Throughout the work, we looked for opportunities to take advantage of CPU parallelisation as much as possible.

Several optimisations were developed for MBS. Memory optimisations included the **StateVector** on-demand expansion optimisation and the **sparse matrix** optimisation. The running time of the simulator also benefited significantly from these optimisations. In addition, three performance optimisations were developed: the **tensor product with identity optimisation**, which minimises the number of tensor products required per gate expansion, the **swap/move optimisation**, which replace the initial naive swapping algorithm with one that requires far fewer swapping steps, and the **CircuitOptimiser**, which parallelises gate matrix expansion on the CPU. These optimisations were evaluated and compared to the base implementation and found to be of significant value. In particular, it is important to realise that as the number of qubits required to simulate increases, the performance gain provided by the optimisations also increases (due to the exponential relationship between the number of states and the number of qubits).

The matrix-free simulator was developed as per an initial design by Kelly (2018), and expanded upon by adding additional functionality (multi-controlled single qubit gates and multi-qubit gates). This simulator was compared against the best-optimised version of MBS and was found to be significantly faster, making it a clear choice for any further work.

We implemented several quantum computing algorithms on the simulator demonstrating its ability to simulate any given circuit and showing examples of its use. Implemented algorithms include Deutsch's and the Deustch-Josza algorithm, the Quantum Fourier Transform, the Cuccaro adder and the QFT adder, Grover's search algorithm, and three quantum error correction codes.

Finally, we implemented a Quantum Language Interpreter which gives users the ability to access the simulator using an easy to use language. The language allows the user to define custom gates, loop over any set of gates with any custom loop parameters, define integer variable to act as qubit labels, take multiple measurements of a register at a time, and to import custom-defined gates from other files.

We compared the MFS to two third-party quantum computing simulators: **libQuantum** and **QX Studio**. Both proved to outperform the developed MFS, however not by many orders of magnitude (one order at most), across the tested algorithms. This shows that there remains room to improve the MFS, which is discussed in the next section.

8.1 Future Work

This work can be further improved in a number of ways, which are summarised here.

The **MBS** would benefit from:

- Parellelising sparse matrix multiplication; it was briefly discussed in this work why this was not immediately possible.
- Further improvements to the swap/move algorithm; while it significantly brought down the number of swaps required from the initial naive swapping algorithm, this moving algorithm is still not necessarily optimum.
- Utilise Schmidt Decomposition to break down large state vectors into smaller one when possible.

While the **MFS** proved to be significantly faster than MBS, no unique optimisation were developed for it. Suggestions for optimisations include:

- Parallelising gate applications when possible; i.e. potentially applying more than one gate at the same time.
- The same Schmidt Decomposition suggested for MBS would also be beneficial here as the MFS also uses the `StateVector` on-demand expansion optimisation.
- Potentially using Gray codes to represent the states rather than standard binary; this could have some performance increase especially if the simulator is ported to run on a distributed memory system using, e.g., MPI, as the states of interest could be closer to each other in each MPI node.

It would be of interest to attempt to use different architectures for parallelisation in addition to CPUs. The linear algebra operations for instance, used extensively in MFS, would be very well suited to be accelerated using GPUs. Additionally, recently FPGAs have been shown to have the potential of dramatic acceleration of such applications.

QLI currently has some limitations which could improved by implementing the following:

- Allowing the user to apply a gate to a qubit controlled by an arbitrary number of other qubits. This would make it easy to implement Grover's algorithm in QLI.
- Allowing the user to define parametrised gates, such as the phase shift gate, $R(\phi)$, mentioned in Equation 5.1. This would allow for the implementation of the QFT in QLI, which would open the door more many other algorithm to be implemented in QLI.
- QLI is currently only available for MBS. Making it available for MFS would make it run significantly faster.

It is planned to implement the QLI improvements over summer and publish a bundle with MFS and QLI.

8.2 Personal Reflections

Whilst the results found show that the developed simulators are not up to the same standard as other available options, I personally gained significant experience in solving these types of optimisation problems during my undertaking of this project. I chose to develop a quantum computer simulator from scratch because I wanted to experience such a project for myself, and have complete control over all aspects of the simulator. This let me have complete freedom over the direction of development of the simulators and gave me the responsibility of finding different bottlenecks to target and mitigate with optimisations that I developed.

In addition, I gained plenty of new technical experience as well. Before this project, I had only used C and OpenMP in the context of university coursework, and had never used C++ before.

I felt like this project in particular was very well-suited for me, given that I am a joint honours with physics student. I had studied the quantum mechanics concepts underlying quantum computing throughout my physics studies in universities, and so this complemented the project very well. Also, this project opened a rather amazing opportunity for the future of my career. I will be undertaking a PhD in this field after I graduate this year, supervised by the same esteemed team of academics who supervised this work. The PhD research will look into how combining different architectures can allow us to accelerate such simulators. Using FPGAs in particular will be looked at in far more detail.

A CircuitOptimiser Algorithm Summary

Algorithm 3 CircuitOptimiser

```
qReg ← pointer to the QRegister;
gates ← input list of ApplicableGates;
qubitsToCombine ← empty dictionary;
steps ← list of Steps;
{Compute the steps}
for i ← 0 to length of gates do
    if number of qubits of ith gate is 1 then
        add to steps a new step consisting of the current form of the state vectors and the ith gate;
        go to next loop;
    end if
    index ← combineQubits(qubits of the ith gate);
    add this combination of qubits to the dictionary;
    apply the move algorithm defined above to ensure qubits are adjacent, but instead of per-
    forming a SWAP, add it to the list of Steps;
    add the ith gate to the list of Steps coupled with the current form of the state vectors;
end for
{Prepare the matrices}
ops ← array of the same size as steps;
{This is parallelised using "pragma omp parallel for"}
for i ← 0 to length of steps do
    gate ← the gate of the ith step;
    state ← the state of the ith step;
    li ← the index of the state vector from ‘state‘ in which the first target qubit of ‘gate‘ is;
    if the number of target qubits of ‘gate‘ is 1 and the number of qubits in the li-th state vector
    is 1 then
        ops[i] ← a new StateVectorApplicableGate from ‘gate‘ and ‘li‘;
        go to next loop;
    end if
    startingIndex ← the index of the first target qubit in the li-th state vector;
    preparedGate ← the gate, having been expanded through tensor products with appropriately
    sized identity matrices, as previously described;
    ops[i] ← a new StateVectorApplicableGate from ‘preparedGate‘ and ‘li‘;
end for
{Apply the prepared gates}
for i ← 0 to length of steps do
    if i is in the keys of qubitsToCombine then
        qReg ensures the qubits are combined as per qubitsToCombine[i];
    end if
    qReg directly applies the ith prepared gate to the state vector ID given by the ith entry of
    ops;
    qReg tells the same state vector to reorder its qubit IDs as per the state in the ith step (from
    steps);
end for
```

B Shor code test

```
QRegister reg (18);

reg.entangleQubits({0,9});

vector<ApplicableGate> firstEncode = ShorEncodeGates({0,1,2,3,4,5,6,7,8});
vector<ApplicableGate> firstDecode = ShorDecodeGates({0,1,2,3,4,5,6,7,8});
vector<ApplicableGate> secondEncode =
    ShorEncodeGates({9,10,11,12,13,14,15,16,17});
vector<ApplicableGate> secondDecode =
    ShorDecodeGates({9,10,11,12,13,14,15,16,17});

vector<ApplicableGate> gates;

gates.insert(gates.end(), firstEncode.begin(), firstEncode.end());
gates.insert(gates.end(), secondEncode.begin(), secondEncode.end());

// Error gates
gates.push_back(HAGate({0}));
gates.push_back(XAGate({0}));
gates.push_back(HAGate({9}));
gates.push_back(XAGate({9}));
gates.push_back(ZAGate({9}));

gates.insert(gates.end(), firstDecode.begin(), firstDecode.end());
gates.insert(gates.end(), secondDecode.begin(), secondDecode.end());

CircuitOptimiser co (&reg, gates);
co.executeCircuit();

cout << takeMeasurementsInString(reg, 10, nullptr) << endl;
```

Listing B.1: *Testing the Shor quantum error correction code on preserving two-qubit entanglement.*

This produces the result {"|011100100000100100>":2, "|011100100011000000>":3, "|100000000111000000>":3, "|111100100100100100>":1, "|111100100111000000>":1}, which demonstrates that the two relevant qubits are entangled as required.

C | Quantum Algorithms in QLI

C.1 Entanglement

This section defines some entanglement gates implemented in QLI.

```
-- Entangle 2 qubits
gate entangle 2
  H 0
  CNOT 0 1
endgate

-- Entangle 5 qubits
gate entangle5 5
  H 0
  loop n 1 4
    CNOT 0 n
  endloop
endgate

-- Entangle n qubits
gate entangleAll
  H 0
  loop n 1 argc-1
    CNOT 0 n
  endloop
endgate
```

Listing C.1: *Entanglement gates in QLI.*

C.2 Cuccaro Adder

Listing C.2 defines a dynamic gate that performs the Cuccaro adder algorithm on an arbitrary number of qubits. Listing C.3 shows how this would be used to add two 6-bit numbers.

```
gate nBitAdd

  let N argc

  -- step 1
  loop i 3 N-2-1 2
    CNOT i+1 i
  endloop
```

```
  -- step 2
  loop i 0 N-5-1 2
    CNOT i+4 i+2
    TOFF i i+1 i+2
  endloop

  -- step 3
  CNOT N-2 N-1

  -- step 4
  loop i 3 N-4-1 2
    X i
  endloop

  -- step 5
  TOFF N-4 N-3 N-1

  -- step 6
  loop i 2 N-3-1 2
    CNOT i i+1
  endloop

  -- step 7
  TOFF N-6 N-5 N-4

  -- step 8
  loop i N-8 0 -2
    TOFF i i+1 i+2
    X i+3
    CNOT i+6 i+4
  endloop

  -- steps 9 and 10
  CNOT 4 2
  CNOT 1 0

  -- step 11
  loop i 3 N-2-1 2
    CNOT i+1 i
  endloop

endgate
```

Listing C.2: *Gate defining the n-bit Cuccaro adder.*

```
init 14

link nbitAdder

let a0 1
let a1 4
let a2 6
let a3 8
let a4 10
let a5 12
```

```
let b0 0
let b1 3
let b2 5
let b3 7
let b3 9
let b5 11
let b6 13

X b5
X a5

nBitAdd all

return 13 11 9 7 5 3 0
```

Listing C.3: *Using the nBitGate gate defined above to add two six bit numbers. This shows the addition of 100000 and 100000, returning 1000000.*

C.3 Quantum Teleportation

```
gate teleport 3

 let source 0
 let ancillary 1
 let target 2

 H source
 H target

 CNOT target ancillary

 CNOT source ancillary

 H source
 CNOT ancillary target

 H target

 CNOT source target

 H source
 H ancillary

endgate
```

Listing C.4: *Quantum teleportation algorithm for teleporting one qubit as per Figure 5.4.*

C.4 Quantum Error Correction Codes

```
-- Bit flip encode gate
gate bitFlipEncode 3
  CNOT 0 1
  CNOT 0 2
endgate

-- Bit flip decode gate
gate bitFlipDecode 3
  CNOT 0 1
  CNOT 0 2
  TOFF 2 1 0
endgate
```

Listing C.5: *Bit flip encode and decode gates.*

```
-- Phase flip encode gate
gate phaseFlipEncode 3
  CNOT 0 1
  CNOT 0 2
  loop n 0 2
    H n
  endloop
endgate

-- Phase flip decode gate
gate phaseFlipDecode 3
  loop n 0 2
    H n
  endloop
  CNOT 0 1
  CNOT 0 2
  TOFF 2 1 0
endgate
```

Listing C.6: *Phase flip encode and decode gates.*

```
link bitFlipCode
link phaseFlipCode

-- Shor encode gate
gate shorEncode 9
  phaseFlipEncode 0 3 6

  loop n 0 2
    bitFlipEncode 3*n 3*n+1 3*n+2
  endloop
endgate

-- Shor decode gate
gate shorDecode 9
  loop n 0 2
    bitFlipDecode 3*n 3*n+1 3*n+2
```

```
  endloop

  phaseFlipDecode 0 3 6
endgate
```

Listing C.7: *Shor code encode and decode gates.*

```
-- Test function for Shor code

link shorCode

gate testShorCode 9
  shorEncode all

  X 0
  Z 0
  H 0
  Y 0

  shorDecode all
endgate
```

Listing C.8: *Shor code test example.*

D Evaluation Data

In this appendix, the data that was used to plot the graphs in the evaluation chapter are given. They are provided in the form of python lists, as matplotlib was used to plot the graphs. All measurements are in seconds.

```
## Six Bit Cuccaro Adder
sixBitCuccaro["sparseMatrix"] = [4.97204, 5.2889, 5.34636, 5.40119, 5.57536,
    5.48964, 5.56526, 5.56325, 5.67292, 5.65377]
sixBitCuccaro["tensorProduct"] = [1.86137, 1.93872, 1.91257, 2.01046, 2.10236,
    2.01952, 2.00132, 2.02538, 2.06837, 2.04238]
sixBitCuccaro["swap"] = [0.262744, 0.24673, 0.252698, 0.240938, 0.26379,
    0.266057, 0.272915, 0.307789, 0.280732, 0.279012]
sixBitCuccaro["circuitOptimiser"] = [0.148322, 0.14369, 0.143563, 0.149211,
    0.150211, 0.149439, 0.144998, 0.148753, 0.148075, 0.145343]
sixBitCuccaro["MFSCpp"] = [0.0198845, 0.0171636, 0.0189235, 0.0201423, 0.0158027,
    0.0184635, 0.0167815, 0.0159982, 0.017155, 0.0187834]
sixBitCuccaro["MFSPython"] = [0.5697290897369385, 0.5654201507568359,
    0.6300492286682129, 0.5970900058746338, 0.5961129665374756,
    0.5821170806884766, 0.5827648639678955, 0.6142289638519287,
    0.5938510894775391, 0.6286830902099609]

## Eight Bit Cuccaro Adder
eightBitCuccaro["sparseMatrix"] = [232.562, 201.339, 208.474]
eightBitCuccaro["tensorProduct"] = [75.5074, 76.3749, 77.1541, 77.8872, 77.6576,
    78.707]
eightBitCuccaro["swap"] = [5.86739, 6.70946, 6.66325, 6.82182, 6.81669, 7.05067,
    6.97598, 7.19657, 7.5523, 8.09873]
eightBitCuccaro["circuitOptimiser"] = [2.44596, 2.50828, 2.95763, 3.0352,
    2.97448, 3.1249, 3.04478, 3.08006, 3.16542, 3.12437]
eightBitCuccaro["MFSCpp"] = [0.27889, 0.259421, 0.263479, 0.26166, 0.264794,
    0.263095, 0.276829, 0.273586, 0.276815, 0.27138]
eightBitCuccaro["MFSPython"] = [13.965410947799683, 14.40625, 14.724072933197021,
    14.555179834365845, 14.207228899002075, 14.167850971221924,
    14.081443071365356, 14.1582190990448, 14.15177869796753, 14.023910760879517]

## 12 Qubit QFT
twelveQubitQFT["sparseMatrix"] = [4.40733, 4.62143, 4.91585, 4.79144, 5.14391,
    4.88364, 5.0891, 4.9389, 5.08491, 4.92431]
twelveQubitQFT["tensorProduct"] = [3.01675, 3.15022, 3.60194, 3.40491, 3.42498,
    3.71766, 3.59571, 3.7066, 3.8178, 4.2163]
twelveQubitQFT["swap"] = [1.54241, 1.61553, 1.6314, 1.64145, 1.68148, 1.73271,
    1.73593, 1.72507, 1.7466, 1.82781]
twelveQubitQFT["circuitOptimiser"] = [0.951941, 0.968114, 0.964902, 0.962328,
    0.9416, 0.938109, 0.956593, 0.951849, 0.953703, 0.949491,]
twelveQubitQFT["MFSCpp"] = [0.0074523, 0.00554452, 0.00482314, 0.00507401,
    0.00589206, 0.00527656, 0.00353013, 0.00375474, 0.00368133, 0.003572]
twelveQubitQFT["MFSPython"] = [0.20763802528381348, 0.1992032527923584,
    0.2086200714111328, 0.21040892601013184, 0.21875691413879395,
```

```
    0.20960307121276855, 0.22414803504943848, 0.21168899536132812,
    0.21737909317016602, 0.20654821395874023]
twelveQubitQFT["QX"] = [0.003709, 0.00221845, 0.00227494, 0.00338396, 0.00255132,
    0.0026913, 0.00230753, 0.00230742, 0.00258861, 0.00269052]

## 15 Qubit QFT
fifteenQubitQFT["sparseMatrix"] = [182.159, 184.825, 180.292, 178.117]
fifteenQubitQFT["tensorProduct"] = [148.19, 144.592, 144.113, 144.122, 144.222]
fifteenQubitQFT["swap"] = [80.1081, 73.9688, 77.0958, 77.1211, 77.2852]
fifteenQubitQFT["circuitOptimiser"] = [63.5944, 63.6301, 63.4241, 64.173, 64.8049,
    66.4986]
fifteenQubitQFT["MFSCpp"] = [0.0419233, 0.0398275, 0.0335646, 0.0344345,
    0.0343453, 0.0402635, 0.0408642, 0.0407184, 0.0377606, 0.0366413]
fifteenQubitQFT["MFSPython"] = [2.6014180183410645, 2.632641077041626,
    2.6698648929595947, 2.6839728355407715, 2.6660428047180176,
    2.714071035385132, 2.680086851119995, 2.7392890453338623, 2.758836030960083,
    2.8445377349853516]
fifteenQubitQFT["QX"] = [0.00558594, 0.00533387, 0.00487083, 0.00534395,
    0.00523512, 0.00583799, 0.0058227, 0.00481834, 0.00558143, 0.00545878]

## 20 Qubit QFT
twentyQubitQFT["MFSCpp"] = [1.91858, 2.04494, 2.04633, 2.19634, 2.16873, 2.24077,
    2.33453, 2.30687, 2.2485, 2.29616]
twentyQubitQFT["MFSPython"] = [157.24818420410156, 161.32120323181152,
    163.18367505073547, 162.07471990585327, 155.42757892608643]
twentyQubitQFT["QX"] = [0.0683492, 0.0720555, 0.0725373, 0.0706384, 0.0734309,
    0.0730448, 0.0743534, 0.072605, 0.0716182, 0.0731933]

## 10 Qubit Grover's Search
grovers10["swap"] = [73.7069, 71.8631, 77.3345, 76.8897, 77.2331]
grovers10["circuitOptimiser"] = [36.3459, 36.5641, 36.2699, 36.9646, 36.4745,
    35.8949, 36.8341]
grovers10["MFSCpp"] = [0.0304088, 0.0395423, 0.0309762, 0.0271448, 0.026693,
    0.0290795, 0.0268601, 0.0267796, 0.0275492, 0.0278573,]
grovers10["MFSPython"] = [1.176297903060913, 1.145475149154663,
    1.194793939590454, 1.2254791259765625, 1.2119500637054443,
    1.1534457206726074, 1.1770591735839844, 1.2125027179718018,
    1.2105679512023926, 1.1866638660430908]
grovers10["libQuantum"] = [0.015, 0.013, 0.014, 0.014, 0.014, 0.013, 0.014,
    0.014, 0.013]

## 15 Qubit Grover's Search
grovers15["MFSCpp"] = [8.62755, 8.4205, 8.77716, 9.28922, 9.13607, 8.94614,
    8.65274, 8.66517, 8.84877, 8.46672]
grovers15["libQuantum"] = [2.429, 2.437, 2.423, 2.440, 2.383, 2.433, 2.443,
    2.468, 2.450, 2.427]
```

Bibliography

A. Buluç and J. R. Gilbert. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, Jan. 2012. ISSN 1064-8275, 1095-7197. doi: 10.1137/110848244. URL http://epubs.siam.org/doi/10.1137/110848244.

B. Butscher and H. Weimer. libquantum - Simulation of quantum mechanics, 2007. URL http://www.libquantum.de/.

S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton. A new quantum ripple-carry addition circuit. *arXiv:quant-ph/0410184*, Oct. 2004. URL http://arxiv.org/abs/quant-ph/0410184. arXiv: quant-ph/0410184.

D. Deutsch. Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 400(1818): 97–117, July 1985. ISSN 1364-5021, 1471-2946. doi: 10.1098/rspa.1985.0070. URL http://rspa.royalsocietypublishing.org/cgi/doi/10.1098/rspa.1985.0070.

D. Deutsch and R. Jozsa. Rapid Solution of Problems by Quantum Computation. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 439(1907):553–558, Dec. 1992. ISSN 1364-5021, 1471-2946. doi: 10.1098/rspa.1992.0167. URL http://rspa.royalsocietypublishing.org/cgi/doi/10.1098/rspa.1992.0167.

B. Eastin and S. T. Flammia. Q-circuit Tutorial. *arXiv:quant-ph/0406003*, June 2004. URL http://arxiv.org/abs/quant-ph/0406003. arXiv: quant-ph/0406003.

R. P. Feynman. Simulating physics with computers. 1981.

L. K. Grover. A fast quantum mechanical algorithm for database search. *arXiv:quant-ph/9605043*, May 1996. URL http://arxiv.org/abs/quant-ph/9605043. arXiv: quant-ph/9605043.

A. Kelly. Simulating Quantum Computers Using OpenCL. *arXiv:1805.00988 [quant-ph]*, May 2018. URL http://arxiv.org/abs/1805.00988. arXiv: 1805.00988.

N. Khammassi. QX Quantum Computer Simulator, a. URL http://quantum-studio.net/#users_manual.

N. Khammassi. QX Quantum Code 0.1 User Manual. b.

M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, Cambridge ; New York, 10th anniversary ed edition, 2010. ISBN 978-1-107-00217-3.

OpenMP. Home - OpenMP. URL https://www.openmp.org/.

L. Ruiz-Perez and J. C. Garcia-Escartin. Quantum arithmetic with the Quantum Fourier Transform. *Quantum Information Processing*, 16(6), June 2017. ISSN 1570-0755, 1573-1332. doi: 10.1007/s11128-017-1603-1. URL http://arxiv.org/abs/1411.5949. arXiv: 1411.5949.

P. W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, 52: R2493–R2496, Oct 1995. doi: 10.1103/PhysRevA.52.R2493. URL https://link.aps.org/doi/10.1103/PhysRevA.52.R2493.

A. M. Steane. A Tutorial on Quantum Error Correction. page 24.

M. M. Waldrop. The chips are down for MooreâĂŹs law. *Nature News*, 530(7589): 144, Feb. 2016. doi: 10.1038/530144a. URL http://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338.

Wikipedia, the free encyclopedia. Quantum circuit for quantum-fourier-transform with n qubits, 2019. URL https://commons.wikimedia.org/wiki/File:Q_fourier_nqubits.png#/media/File:Q_fourier_nqubits.png. [Online; accessed March 27, 2019].

M. M. Wolf, E. G. Boman, and B. A. Hendrickson. Optimizing Parallel Sparse Matrix-Vector Multiplication by Corner Partitioning.

W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886):802, Oct. 1982. ISSN 1476-4687. doi: 10.1038/299802a0. URL https://www.nature.com/articles/299802a0.