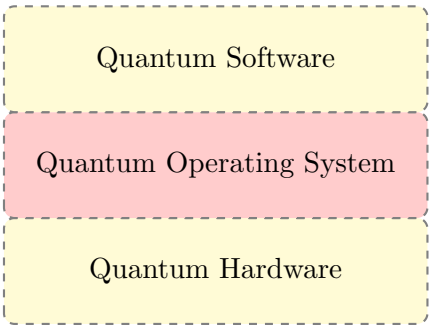
Flinders University

College of Science and Engineering

Design of a Quantum Computer Operating System

by

Reid Honan, B.Eng(Software)(Hons)



supervised by
Dr. Trent Lewis and
Dr. Michael Haythorpe and
Dr. Greg Falzon

Thesis submitted to Flinders University for the degree of
Doctor of Philosophy

May 9, 2024

THESIS SUMMARY

The purpose of this Thesis Design of a Quantum Computer Operating System is to examine the capability of a Quantum Computer and investigate the applicability of an Operating System to improve the efficiency of the devices.

This thesis is composed of ten Chapters, with each Chapter investigating a different facet of the system. Chapter one is introductory and presents the position of the author and defines the direction of the research. Chapter two examines the relevant literature of both Quantum Computing and Operating Systems in order to prepare the reader for the system developed in subsequent chapters. Chapter three encompasses the methodology underpinning the research completed in this thesis, this includes the research plan, experiment choice and philosophical worldview.

Chapter four concentrates on presenting the theoretical design of the Glade OS system which forms the core of this Thesis. The theoretical system demonstrates the capability of a Quantum Computer to process multiple programs concurrently and presents a graph theory approach which elegantly combines the existing information to allow the planning and management of this approach. Chapter five then continues the Glade OS system with a direct focus on the implementation of the system into code. This implementation forms the basis of the testing in subsequent Chapters and demonstrates some optimisations which are not currently available in commercially available alternatives.

Chapter six analyses both the theoretical and implemented systems from Chapters four and five in order to determine the efficiency, performance and accuracy of the system. Chapter seven builds on the analysis by review a series of extra improvements to boost the systems capabilities.

Chapter eight expands the Thesis by considering the application of the Glade OS system in multiple different configurations. This Chapter demonstrates the universality of the proposed system and the adaptability of the approach. Chapter nine completes the review of Glade OS by investigating the cyber security vulnerabilities within the system and the alternative configurations from Chapter eight. A review of this content has not been seen in current Quantum Computing research and highlights interesting approaches.

Conclusions are drawn in Chapter ten as the multifaceted analysis from Chapters six to nine are combined and the answers to the research questions from Chapter one are confirmed. The author suggests that Quantum Computer Operating Systems should be introduced to all Quantum Computers in order to improve their efficiency.

DECLARATION

I certify that this Thesis:

1. does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university
2. and the research within will not be submitted for any other future degree or diploma without the permission of Flinders University; and
3. to the best of my knowledge and belief, does not contain any material previously published or written by another person except where due reference is made in the text.

May 9, 2024

ACKNOWLEDGEMENTS

First and foremost I am extremely grateful to my supervision team, Dr. Trent Lewis, Dr Michael Haythorpe, Dr. Greg Falzon and Prof. David Powers for their limitless patience, invaluable advice and continuous support during my PhD journey. Their immense knowledge and plentiful experience has allowed me to grow and develop as an academic researcher and in life.

I would also like to thank Dr. Scott Anderson and Mr. Jake Cooke for their technical support on my study. It was their kind help and support that has made my study as efficient and enjoyable as it was.

Additionally, I would like to express my gratitude to my parents Brenda and William, my sister Paige, brother Sam and nephew Lincoln and everyone in my Village. Without their tremendous understanding and encouragement in the past few years, it would have been impossible for me to complete my studies.

I would never have been in a position to complete this work without the fantastic introduction and grounding in IT from Kristian Burghof which set me on this path.

Finally, I must acknowledge the contribution of my Flinders University Research and Travel Scholarships in the completion of this Thesis.

PUBLICATIONS

1. R. Honan, T. W. Lewis, S. Anderson, and J. Cooke, “A quantum computer operating system,” in International Conference on Algorithms and Architectures for Parallel Processing, Springer, 2020, pp. 415–431.

Contents

List of Figures

List of Tables

List of Algorithms

List of Notable Equations

1. INTRODUCTION

Traditional computers have exceeded all expectations with their development [1]. To have a palm sized device which connects to the sum total of human knowledge was amongst the wildest future ever forecast [1]. However, traditional computers are not infallible, many problems are currently either beyond the ability of computers or will take such an extreme amount of time such that the results are no longer relevant [2]. This difficulty stems from issues with the traditional model of computing, a binary model based on the deterministic behaviour of electricity according to classical physics.

Quantum mechanics is largely believed to hold the keys to understanding the universe [3], [4]. This is due to the nature of quantum mechanics underpinning reality as humanity interprets it [3], [4]. Quantum mechanics introduces the concept of superposition and entanglement, concepts which run counter to the observable universe. In classical physics distinct states exist and objects can only exist in one of these states at any distinct point in time. In contrast, quantum mechanics permits a superposition of states occuring simultaneously until an observation occurs. For example, A coin is either heads or tails, a person is either wet or dry. Superposition allows the state to exist in more than one state at a time and therefore combine multiple states together which exceeds the options available when using standard binary states in classical computing. These concepts offer a large boost to the computing power, but also introduce significant drawbacks that must be addressed.

To combat the rise of large data sets and computationally difficult problems a new computing paradigm is required. In the early 1980's Richard Feynman explored the theoretical concept of a machine built on quantum mechanics which could outperform traditional computers [5]. This concept grew rapidly following Peter Shor presenting a real world use case with an algorithm to factor large prime numbers using a quantum computer in 1994 [6], [7]. Today, quantum computing is a multi-billion dollar industry with research and businesses covering the globe [8], [9]. Quantum computers owe their success to the introduction of a new computing paradigm, one based on quantum mechanics over the more deterministic classical physics.

Currently users of a quantum computer can create a process (AKA a program composed of logic gates), have it allocated to the computers memory and retrieve the results at the end. This is accomplished by utilising a single job in single job out (SJISJO) philosophy which works well for a technology under development. By testing the technology with a single job at a time it reduces the sources of error and makes research and development easier. While this SJISJO approach works for development, it is a mark of technological maturity that the system move from laboratory conditions to working in a relevant environment.

The Technology Readiness Levels [10], [11] provide a roadmap for transforming technology from a theory (TRL1) through to fully operational technology (TRL9). As a part of the transition from TRL4 onwards, the SJISJO philosophy is inefficient. Classical computing faced a strikingly similar issue to the one described above. Early computers focused on a single task at the exclusion of all else, until IBM's system/360 managed to implement multiprogramming [12, p. 37]. This single task method worked well to perfect the operations, but lacked the versatility that made computers the universal workhorses they are today. Early computer programs were entirely self contained, detailing everything from the necessary operations through to how to interact with peripheral devices. This approach led to limited program portability, due to inconsistencies between different computers and manu-

Table 1.1: Technology Readiness Levels - adapted from [10], [11]

Technology Readiness Level	**Description**
TRL9	Actual system proven in operational environment
TRL8	System complete and qualified
TRL7	System prototype demonstration in operational environment
TRL6	System/subsystem model or prototype demonstration in a relevant environment
TRL5	Component or breadboard validation in relevant environment
TRL4	Component or breadboard validation in laboratory environment
TRL3	Analytical and experimental critical function an/or characteristic proof of concept
TRL2	Technology concept or application formulated
TRL1	Basic principles observed and reported

facturers. To resolve this problem, in 1956 General Motors IBM mainframe operators began to develop a program which could abstract the operation of the computer from the user programs [13]. This approach allowed user programs to focus on the specifics of their program, while relying on the mainframe program to handle the operation of the computer system. Other companies soon followed the General Motors approach, though the resultant operating systems still varied wildly between different companies and computers.

In the 1960's IBM began development on their own operating system, aiming to provide a consistent interface for all IBM machines [12]. Other early pioneers of operating systems include Control Data Corporation, Computer Sciences Corporation, Burroughs Corporation, GE, Digital Equipment Corporation, and Xerox [12]. Unix was also published during the late 1960's with Microsoft's Windows operating system still some time away in 1985 [12]. Today operating systems are core to the use and management of computers, with annual profits of approximately $25 Billion (12% of total revenue) in 2022 [14], [15].

The research presented within this Thesis works to extend the field of quantum computing by examining and implementing the missing pieces of a quantum computer operating system. As will be explored in Figure 2.8 an operating system is composed of 3 pillars:

1. Process Management
    - Scheduling
    - Synchronisation
2. Resource Management
    - Memory Management
    - Context Management
3. Communications Management
    - File Systems
    - Communications Services
    - Security

Current quantum computers can be argued to meet a subset of these requirements. The SJISJO approach does technically provide a means of scheduling processes, accomplished by acting as a queue with a First In First Out algorithm, and the memory management is a simple task of allocating what the task requires from the resource pool before freeing and allocating for the next process. The SJISJO approach removes the need for synchronisation and Context Management by only implementing a single job at any time.

The research presented within this Thesis produces a system which supports full implementations of all components except for the File Systems and Communications Services:

1. Process Management
    - Scheduling
    - Synchronisation
2. Resource Management
    - Memory Management
    - Context Management
3. Communications Management
    - ~~File Systems~~
    - ~~Communications Services~~
    - Security

File systems have been excluded because the memory required to implement them is not available at this time and so quantum computers are physically incompatible with this requirement. Communication Services has been excluded as it is only partially available, quantum computers and quantum networking exist independently but exisiting implementations have yet to be combined. The concept of quantum networking integration is discussed in Chapter 7 but as it is still theoretical this Thesis cannot state it supports the requirement in good faith.

1.1 Research Questions

The research within this Thesis will specifically focus on the following research questions:

RQ1 Can a quantum computer support processing of multiple programs concurrently?

RQ2 Is there a framework which can be implemented on a quantum computer to support processing of multiple programs concurrently?

RQ3 What is the cost and/or effect of adopting this framework on a quantum computer? The cost and effect are considered in terms of:

RQ3A Implementation,

RQ3B Performance,

RQ3C Algorithmic Complexity,

RQ3D Versatility/portability,

RQ3E Security

This Thesis presents each of the research questions in order throughout the remaining chapters while designing a system which could easily be utilised today.

1.2 Chapter Breakdown

The remainder of this Thesis is a discussion covering the art of parallel computing on a quantum computer before progressing into a presentation of a quantum operating system. Specifically, the chapters construct the system in a direct manner with each subsequent chapter building on the preceding content directly.

Chapter 2 reviews the related literature and develops the framework through which the rest of this Thesis will be viewed. This chapter reviews the standard implementations of hardware and software for a quantum computer from the base definitions in order to determine a universal representation to design a framework around. This universal framework was used to allow the system in subsequent chapters to be applied to all currently accepted implementations of a quantum computer.

Chapter 3 formalises the methodological approach taken within this Thesis and describes the criteria which will be used to review the artifact created in the following chapters. The base system developed throughout this Thesis is subsequently reviewed by these criteria in Chapter 7 before continuing to develop the system further.

Chapter 4 addresses the first 2 research questions by first providing a proof by construction that quantum computers can definitively support processing of multiple programs concomitantly (RQ1). With that question addressed, the remainder of the chapter constructs and defends a modular theoretical framework which can be used to augment the existing quantum computer work flow and thus add support for automatically allocating multiple programs concomitantly (RQ2). This framework is deliberately designed using a modular approach to allow for the further augmentation and specialization in future research, this modularity is explored in Chapters 6 and 7.

Chapter 5 further develops the framework constructed in Chapter 4 by outlining the implementation in code, in doing so it presents the implementation cost associated with the third research question (RQ3A). This chapter reviewed popular quantum computer simulators and found each lacking in functionality to nicely implement the framework. Due to the missing features an alternative simulator was developed which natively supported the concept of parallel quantum programs. This new simulator approaches the simulation of a quantum computer from a novel direction and thus this chapter describes the implementation in exhaustive detail to ensure accuracy.

Chapter 6 reviews the framework described in Chapter 4 and identifies modular sections within the framework which can be replaced with alternative algorithms to accomplish the same task. These sections are reviewed independently and the alternative algorithms are discussed with the intention of identifying the recommended algorithm for each section. Following this review, 2 more categories of tests are employed. These include tests to compare the performance of the framework against the existing popular simulators (as found in Chapter 5) and tests to confirm the accuracy of the new simulator developed in Chapter 5. Combined this discussion and subsequent tests presents the solution to Research Question 3B.

Chapter 7 further considers issues with the new framework with the intent of either providing a solution, or exploring possible solutions before highlighting areas which require further research. As these issues form the bottlenecks of the system their algorithm complexity heavily impacts the performance of the final system and this discussion forms the answer to Research Question 3C.

Chapter 8 explores more advanced configurations beyond the 1 quantum computer to 1 classical computer configuration that has been used in the previous chapters. This chapter explores three alternative configurations and explores how the framework outlined in Chapter 4 can be applied to each in turn. Due to resource limitations this chapter is largely theoretical however it serves to demonstrate the universal nature of the framework presented in Chapter 4 and therefore resolves the Research Question 3D.

Chapter 9 reviews the framework presented in Chapter 4 and the alternative configurations presented in Chapter 8 with a focus on the security of each. This chapter constructs a library of cyber-security issues which have the potential to cause problems for quantum computation if not addressed. Each issue is classified and graded before reviewing which configurations are susceptible. Finally each issue is reviewed and either a solution or a series of

mitigation strategies designed to reduce the danger of the issue is presented. With the completion of this chapter the Research Question 3E is discussed thereby resolving all of the research questions.

Chapter 10 completes the Thesis by summarising the content found throughout the previous chapters. This conclusionary chapter demonstrates the key findings, implications and limitations of the work presented within this Thesis. Due to the rapid pace of development in this field this chapter also includes a review of more recent literature (other than Chapter 2) and how it applies to this Thesis. The research questions are summarised and the final overall answers are presented. Finally this Chapter looks to the future in an attempt to forecast what future work is required and where subsequent research could have a significant impact on this area.

2. LITERATURE REVIEW

2.1 Quantum Computing

2.1.1 Definition of a Quantum Computer

Quantum computing has existed in one form or another for approximately 40 years and is popularly traced back to the "Simulating Physics with Computers" presentation by Richard Feynman [5]. In his talk Feynman discussed the difficulties with using a 'universal computer' to simulate classical physics and by extension quantum mechanics. The argument (correctly) hinged on the requirement to move the mathematics from a continuous space to a discrete space, for example considering the continuous flow of time affecting the system as a series of discrete intervals which are applied in a linear fashion [5]. Because of this requirement classical computers can simulate an imitation of a physical system but will never truly simulate the system. The solution (according to Feynman) is to retreat from universal digital computers and return to analog computing and direct simulation of events [5]. The question that Feynman left the audience with was to "try to find out what kinds of quantum mechanical systems are mutually intersimulatable", in other words is there a universal quantum computer that can be created to simulate all other quantum systems [5]. This presentation directly led to the growth of the quantum computing discipline and is widely considered the birth of the field [16]–[19].

In response to the introduction of quantum computing into popular research, the difficulty became defining what abilities a candidate quantum computer would need to demonstrate in order to meet the definition of a quantum computer. Some definitions had been published with a focus on

the mathematics and theory of simulation [17], [20] however these definitions are largely theoretical and lacked consideration of implementation. Due to the drastic differences in the various developing hardware approaches, a hardware agnostic definition was required. This quest led DiVincenzo (2000) to define his 5+2 criteria for a quantum computer. They are as follows [16]:

Required:

R1 A scalable physical system with well characterised qubits

R2 The ability to initialize the state of the qubits to a simple fiducial state, such as $|000...0\rangle$

R3 Long relevant decoherence times, much longer than the gate operation time

R4 A 'universal' set of quantum gates

R5 A qubit-specific measurement capability

Optional:

O1 The ability to interconvert stationary and flying qubits

O2 The ability to faithfully transmit flying qubits between specified sections.

To further elaborate on these criteria, in R1 a quantum computer must be able to differentiate between individual qubits (e.g. q_0 and q_1 should be entirely separate qubits). This allows for qubits to be singularly specified and acted upon, this criterion also extends to individual qubit measurement.

For R2 a quantum computer which cannot be set to a state of $|000...0\rangle$ cannot reliably perform computation. Any computer whether an abacus or a desktop requires the ability to reset the computer to a simple base state.

R3 introduces the concept of Decoherence for Quantum Computers. Decoherence is perhaps the greatest enemy of a quantum computer, the infernal noise constantly corrupting quantum data. While the goal is to eventually

remove this concern, current generation quantum computers AKA Noisy Intermediate Stage Quantum Computers (NISQ) require time to reliably compute the program and measure the result prior to the decoherence corrupting the data [21], [22].

Lastly for R4 a universal gate set must be established, such that programmers can declare their programs in the logic gates ready for execution. Or at least write their programs using an accepted abstraction layer.

The two optional criteria relate directly to networking functionality, which is not directly required for initial quantum computation. In O1 'stationary' qubits are traditional memory, whereas 'flying' qubits are qubits prepared for networking purposes (i.e. transmitting and receiving). Therefore, it is advantageous to be able to convert between the two, thus allowing for simple data transmission with a minimal overhead. While the requirement to faithfully transmit the qubits (O2) allows for users to be assured that the data will arrive as expected, and can therefore continue the use of the program as normal.

2.1.2 Quantum Computer Hardware

2.1.2.1 Standard gate model of quantum computing

The gate model is considered the standard representation of quantum computer algorithms and was touched on in Feynmans original presentation [5]. The model is analogous to the electronic circuits used to represent digital computers. By representing each qubit as a single horizontal line moving from left to right the model allows us to mark the operations we perform as markers along the line in the order they need to be executed. For example a basic 2-qubit circuit with an operation on the 2nd qubit would look similar to Figure 2.1.

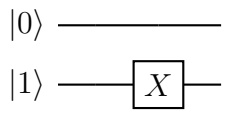


Figure 2.1: Example Quantum Circuit - Each horizontal line represents the path of a single qubit. The $|0\rangle$ on the left represents a qubit starting at state 0, while $|1\rangle$ represents a qubit starting at state 1

Quantum gate array computers are perhaps the closest implementation compared to traditional classical computers. Gate array computers are designed to take a base input state and manipulate it through a circuit composed of predefined logic gates [23]. This approach can be seen in the software offerings by Rigetti [24], IBM [25] and Microsoft [26]. These quantum logic gates include [27]:

1. Pauli-X
2. Pauli-Y
3. Pauli-Z
4. Hadamard
5. Phase Shift
6. $\frac{\pi}{8}$ shift
7. Control Gate
8. Toffoli

While the classical logic gates include:

1. Not
2. And
3. Or
4. Exclusive-Or (XOR)
5. Not-And (NAND)
6. Not-Or (NOR)

Each of the quantum logic gates are unitary, meaning that any transformation can be reversed by applying the conjugate transpose operation ($U^\dagger$) and results in the Identity operation (Equation 2.1).

$$UU^\dagger = U^\dagger U = I \tag{2.1}$$

To this end, each quantum gate has a unitary matrix associated with it [28]. The Control gate is designed to substitute in a 2 x 2 gate matrix in place of the matrix featured in Equation 2.2.

$$\begin{bmatrix} U_{01} & U_{02} \\ U_{03} & U_{04} \end{bmatrix} \tag{2.2}$$

1. Pauli-X $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
2. Pauli-Y $\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
3. Pauli-Z $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
4. Hadamard $\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
5. Phase Shift $\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
6. $\frac{\pi}{8}$ shift $\begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix}$
7. Control Gate $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & U_{01} & U_{02} \\ 0 & 0 & U_{04} & U_{03} \end{bmatrix}$
8. Toffoli $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

These gates are combined with the standard representation of qubits in Section 2.1.4.2 to represent quantum programs.

2.1.2.2 Alternative models of quantum computation

Quantum computers are an expansive research area with researchers agreeing on very little. This has led to the development of multiple alternative methods with which to perform quantum computation. These methods are typically summarised with the main three approaches:

1. One-way Quantum Computer [29]
2. Adiabatic Quantum Computer [30]
3. Topological Quantum Computer [31]

Each of the approaches above are explored in more detail below.

One-way Quantum Computer One-way Quantum Computers are a variation on the quantum gate array that receives a quantum system in a predefined quantum state [29]. This input is then acted upon through the quantum computer via measurements until it resolves to the required output [32]. It has been shown that quantum gate arrays are equivalent to One-Way quantum computers, due to their ability to use logic gates to generate a pre-defined quantum state from any base input state [27].

Adiabatic Quantum Computer Adiabatic Quantum Computers are designed to work using the time dimension. Adiabatic computation is built to take a simple quantum system which can be easily prepared in a ground state which then evolves over time into a complex system which approaches the desired solution [30], [33]. This representation is noted as being particularly good for optimization problems and is currently in use by D-Wave computing in Canada [34] though controversies exist around D-Waves quantum capabilities [35]–[38].

Topological Quantum Computer Topological quantum computing aims to braid quasi-particles together to encode quantum information [31]. These quasi-particles have the interesting property of being their own anti-particles. This approach is noted as being particularly good at error correction, with much of the ability already hard-coded into the braiding [31]. It should be noted that this approach is currently theoretical only, though there is evidence for their existence, the quasi-particles are yet to be discovered [39]. Microsoft originally believed that they had found evidence of the Majorana particle [40] until further examination of their full data set resulted in an outright retraction of that paper.

It should be noted that every alternative approach is capable of being simulated by the 'standard' circuit model [41], [42]. Therefore the choice is largely inconsequential, with the choices merely better suited to perform

certain tasks or be implemented by specific hardware approaches.

2.1.3 Different Implementations and Current Specifications

Now that the theory of a hardware implementation has been introduced, the next step is to consider the various implementations and how they approach the relevant problems. Quantum computer hardware is notoriously difficult to fabricate. This has led to multiple separate technologies being explored in parallel. These technologies include [39]:

- Trapped ion - IonQ
- Superconducting loops - Google, IBM, Quantum Circuits
- Silicon quantum dots - Intel, University of New South Wales
- Topological qubits - Microsoft, Bell Labs
- Diamond vacancies - Quantum Diamond Technologies Inc., Quantum Brilliance, NVision

It should be noted that the current implementations of quantum computing tend to straddle one of two implementations. Either the quantum computer is a single complete system which is purely run through qubits, or a hybrid with a classical computer [28], [43]. Both implementations have their advantages and disadvantages, with the immediate research surrounding the hybrid machines. Hybrid systems continue to work as a classical computer would, with the added bonus of a quantum processor to use as needed [28], [43]. This approach typically approaches a quantum computer as an extension of the GPU (Graphical Processing Unit) concept to formulate a QPU (Quantum Processing Unit). It is expected that while pure quantum computers will eventually exist, the immediate step is to incorporate the QPU into traditional computing.

The different hardware implementations (introduced above) are outlined below and are summarised again in Figure 2.2.

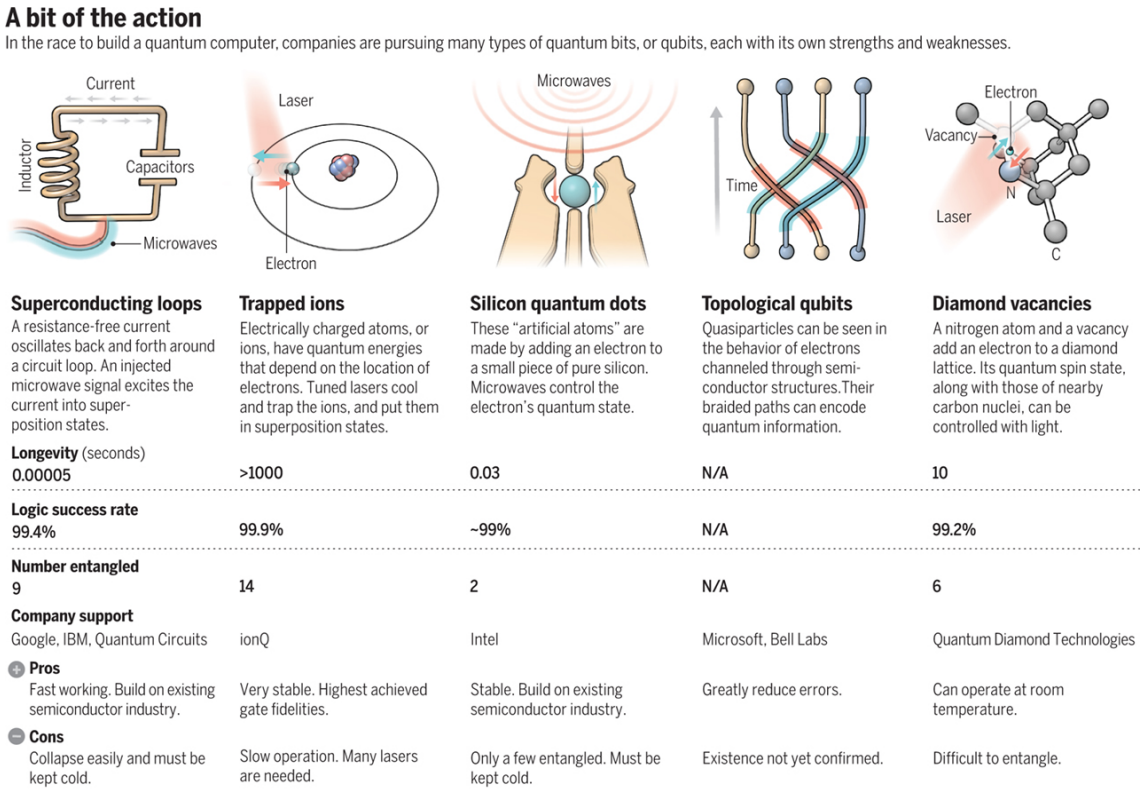


Figure 2.2: Summary of Technologies (Dec, 2016) from [39]. Reprinted with permission from AAAS.

Superconducting Loops Superconducting loops work through use of **Josephson junctions** to create resistance free loops of electrical current [44]. By squishing layers of nonconducting materials between superconducting materials, the system produces a non-linearity within the circuit which defines the 2-level qubits. Manipulation of the Josephson junction through microwaves enables the application of logic gates [44]. This approach benefits from traditional semiconductor fabrication, thereby reducing the cost in both time and money. The downside is most current superconductors require extremely cold temperatures in order to reach peak efficiency, thereby requiring the computer to be extensively cooled in order to properly function.

Trapped Ions Trapped Ion quantum computers work by utilising ions as the qubits, and holding them individually in a trap through the use of an electromagnetic field [45]. These trapped ions are then manipulated with specific lasers to alter their state according to the instructions [45]. These transformations can take many forms depending on the choice of hardware. Variations come in the forms of:

- Use of different ions in the traps.
- New/Different Ion trap design.
- New/Different Ion trap configuration and layout.
- Variations in lasers.

Silicon Quantum Dots Silicon quantum dot quantum computers utilise silicon nanoparticles which are protected from interference through quantum wells [46]. These electrons are then excited by microwaves to transform through the various states [46]. Similar to the superconducting loops technology, this approach can also benefit from its similarity to the traditional semiconductor industry practices.

Topological Quantum Computer Topological quantum computing utilises a unique approach based on quasi-particle braiding which is widely different to the alternative implementations discussed here [31]. Therefore it requires an extensive amount of novel research to be completed. While evidence has been found suggesting the existence of Majorana particles, they have never actually been (officialy) found [39], [40]. Because the required particles have not been verifiably found, there are no statistics on the efficiency of this variation. It is only included to complete the review and because of the theoretical benefits.

Diamond Vacancies In order to combat the difficulties associated with cooling Superconducting loops and Silicon quantum dots, researchers turned towards more stable structures [47], [48]. By exploiting a vacancy within di-

amond lattices, the quantum data is largely stable and can even operate at room temperature [47], [48]. The addition or absence of light can then be used to manipulate the quantum state as desired [47], [48]. Although this approach allows for room temperature computing, it can cause difficulty when attempting to coerce the qubits to work together (especially with entanglement). The manufacturing cost of these machines is also larger than alternatives due to the cost of materials and the specialised nature of their construction.

Now that each of these technologies has a definition, the next logical question is whether they are congruent with the definition of a quantum computer explored in Section 2.1.1.

While Topological quantum computers do not have a physical implementation the theoretical implementations would meet the DiVincenzo criteria [49]. Because of the lack of physical implementations there is significant work to be done, however the current research is compatible due to also meeting the DiVincenzo criteria. The other approaches all suffer from 2 main issues, inter-connectivity and isolation. Isolation is the conflict between properly isolating the qubits to protect against decoherence or opening the qubits to allow them to be measured and manipulated [44], [50]. Similarly interconnectivity problems arise when trying to apply multiple qubit operations. If these issues are resolved then these systems can be said to meet the criteria.

In summary, there are numerous possible options for the hardware of a quantum computer. Each choice of hardware comes with its own problems ranging from issues with the technology to difficulties with control of quantum computations. Because of the numerous variables at play, choosing a specific hardware technology for this software system is foolish. Developing a hardware agnostic approach based purely on the consistent data ensures the viability of the system on whichever hardware technology eventually wins. Developing the system using this approach will require some hardware specific optimisations when implemented, though it will translate to each of the hardware options.

2.1.3.1 Why is this hardware different to traditional computers?

As discussed at the beginning of this Chapter, quantum computers are able to actually simulate quantum processes instead of relying on a classical imitation [5]. All classical computers follow the model of the Turing machine and are restricted to operating in discrete space [5]. Even the 2 competing architectures of Harvard vs Von-Neumann do nothing to change the fundamental model of a system which completes one instruction at a time on a relevant part of the system [5]. The instructions are completed by reading the data into the registers and then performing the different operations in sequential order. A core trick of digital computers is that they can appear to work on multiple programs at the same time because they completed the instructions so quickly and swap contexts so often that it is indistinguishable from the user [51]. A related benefit of this single CPU multiprogramming is that parts of the system that are not being worked on can be left alone (either in the registers or transferred to memory) with the knowledge that the data is consistent when the process returns [51], this knowledge directly underpins the process management concepts discussed in Section 2.2.2 [51].

With quantum hardware, the model has changed significantly. The concepts of programs being composed of instructions remains the same (although the instructions themselves are different) and that the data is stored in some form of memory is also consistent with the standard model. The difference comes when you consider the composition of the memory. As the quantum computer is an analog computer instead of a digital computer the concept of time-blindness no longer applies. The analog approach balances all of the variables constantly and uses time as a continuous force constantly evolving the system [5]. This means that if a process is left unattended it is almost certain that it will not be in the same state that it was left in [5]. The quantum computer is also able to independently evolve each qubit which achieves true parallelism instead of the trick that digital computers use of allocating a small time slice to update each qubit instead of each qubit evolving at the same time [5].

Another consequence of the updated model is that while in a digital computer every piece of memory can communicate and work with every other piece a quantum computer has specific restrictions. Quantum computers come with a map of their qubits which demonstrates which qubits can interact with each other and which qubits cannot. To demonstrate the consequence of this problem consider a digital computer with n-registers but each register will only talk with the registers on either side (e.g. register 4 will only talk with registers 3 and 5). Because of this change, any program you want to execute needs to fundamentally restructure its execution to ensure that all of the operations can still occur and be processed in the correct order.

Because of the above differences the traditional operating systems cannot be simply ported across. While the fundamental tenets of an operating system (process management, resource management and communications management) remain the same, the approaches taken by existing operating systems are not compatible with the components of a quantum computer.

2.1.4 Quantum Software

2.1.4.1 Mathematical Representation of a Quantum Computer

Qubits are the base unit of computation used by quantum computers [28], [43], [52], [53]. Qubits are analogous to classical computer bits, which exist in either high or low voltage states. These states are commonly referenced as 0 and 1, or True and False. Typically a Bloch Sphere is used to best represent the 3-Dimensional nature of qubits [28] (Fig. 2.3). The Bloch Sphere is a natural extension of the unit circle (a circle with a radius of 1) into 3 dimensions.

As shown in Figure 2.3, the binary 0 and 1 states have survived the transition, though now referred to as states $|0\rangle$ and $|1\rangle$ instead. In fact all binary states can be rewritten in the ket notation, $000 \rightarrow |000\rangle$ or $010 \rightarrow |010\rangle$, where $|000\rangle$ means all 3 qubits are in state $|0\rangle$ [28], [43], [52]. This representation is

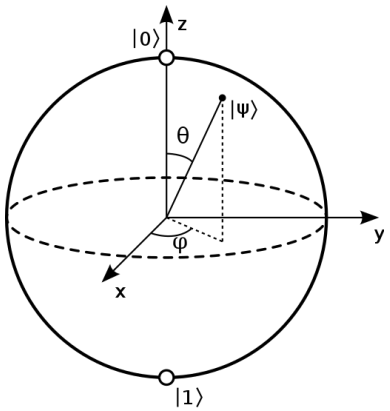


Figure 2.3: Bloch Sphere

related back to the quantum physics waveform representation of a particle. The particle is represented as [28], [43]:

$$|\varphi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2.3}$$

Where the complex number α indicates the amplitude of the state $|0\rangle$ and the complex number β indicates the amplitude of the state $|1\rangle$. Using the wave function the probability of the qubit being in state $|0\rangle$ has the probability of $|\alpha|^2$ and state $|1\rangle$ has the probability of $|\beta|^2$. In these equations the values of α and β are complex-valued. The only requirement is that $|\alpha|^2 + |\beta|^2 = 1$ in order to preserve the laws of probability (explored below). Based on the rules defined above, one method to write a perfectly balanced single qubit superposition is Equation 2.4 [28].

$$|\varphi\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle \tag{2.4}$$

$$\alpha = \frac{1}{\sqrt{2}}, \beta = \frac{1}{\sqrt{2}}, |\alpha|^2 = \frac{1}{2}, |\beta|^2 = \frac{1}{2} \tag{2.5}$$

The wave function can also be used to show the outcome of multiple superpositions. This is accomplished by combining the amplitudes and joining the two states into one, e.g. $|x\rangle \otimes |y\rangle = |xy\rangle$.

A common example of combined states is the four 'Bell States' that are referred to in entanglement (equation 2.6) [28], [54].

$$\begin{aligned}
|B_{00}\rangle &= \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{|00\rangle}{\sqrt{2}} + \frac{|11\rangle}{\sqrt{2}} \\
|B_{01}\rangle &= \frac{|01\rangle + |10\rangle}{\sqrt{2}} = \frac{|01\rangle}{\sqrt{2}} + \frac{|10\rangle}{\sqrt{2}} \\
|B_{10}\rangle &= \frac{|00\rangle - |11\rangle}{\sqrt{2}} = \frac{|00\rangle}{\sqrt{2}} - \frac{|11\rangle}{\sqrt{2}} \\
|B_{11}\rangle &= \frac{|01\rangle - |10\rangle}{\sqrt{2}} = \frac{|01\rangle}{\sqrt{2}} - \frac{|10\rangle}{\sqrt{2}}
\end{aligned} \tag{2.6}$$

These states show that there is a 0.5 probability (50% chance) of getting either state $|00\rangle$ or $|11\rangle$ for the $|B_{00}\rangle$ and $|B_{10}\rangle$ or either state $|01\rangle$ or $|10\rangle$ for $|B_{01}\rangle$ and $|B_{11}\rangle$. This is due to the α and β values resolving the same as Equation 2.5. The importance of this state (which is one of the four Einstein-Podolsky-Rosen states) is that the results of the first quantum qubit dictate the results of the paired qubit. If the first qubit in equation 2.6 is measured at state $|0\rangle$ then the second state must be state $|0\rangle$ [28] and the same in reverse. If the qubits are represented as an independent object then they can be measured independently but if the qubits are entangled then affecting one of them will have repercussions on the linked qubits [28].

No-Cloning Theorem A key difference between the classical computer model and the quantum computer model is the ability to duplicate data. In the classical computer the data can simply be read and a copy can be written. This ability is commonly used throughout classical computing but is mostly lacking in quantum computing [28], [43]. The problem in quantum computing is that the qubits cannot be read without utilising the measurement operation and therefore collapsing the data to one of the 2 base states [28], [43], [52], [55]. Because of this restriction it is only possible to duplicate data that is in a known state like one of the two basis states or one created by the

programmer (as the steps to reproduce it are known) [28], [43].

2.1.4.2 How is the software written? (Languages and Abstractions)

Theoretical Implementation A common tactic for early adopters is to attempt to simulate a new technology, allowing developers to build on the technology while hardware approaches are built. In performing this tactic researchers quickly came across a potential deal breaker, the sheer amount of information required to describe a quantum state made it increasingly difficult to simulate. A single qubit is defined as [28]:

$$|\varphi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2.7}$$

Which requires 2^1 complex values to be stored namely α and β. In order to define two qubits, the waveform requires 2^2 complex values and is represented as either Equation 2.8 or 2.9 [28], [43], [54].

$$|\varphi\rangle = \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \delta\,|11\rangle \tag{2.8}$$

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix} \tag{2.9}$$

Because this relationship continues to grow for each extra qubit, established quantum simulators are limited to approximately 30 qubits (for example Microsoft Q# [26], IBM Qiskit [25] and Rigetti Forest [24]). For n qubits the number of amplitudes that need to be stored are 2^n, with each amplitude being a complex number which when absolute squared is between 0 and 1. This relationship scales exponentially and can rapidly consume all the available memory in the simulator. The typical approach is to store the entire state table of 2^n states for the entire system and performing the manipulations upon that table [28], [43], [54]. As discussed in Section 5.1 this approach does suffer from problems with precision of values (e.g. Pi) how-

ever all computer models suffer from this flaw. The state table works well, and is recognised as a consistent approach which can be used to simulate a quantum system. State tables work well to simulate a single system for a single task [56].

This approach however is not the only option, an alternative approach to simulating the quantum state is *Automatic measure*. Automatic measure is a technique utilised to reduce the state table to only a single output state of qubits instead of growing rapidly to cope with entanglement. This technique performs a quantum logic gate and then instantly measures the output instead of continuing with more logic gates. Because of the measurement operation this approach only needs to keep a single state in memory instead of the entire state table normally required [57], [58]. In order to ensure accuracy this technique must perform optimisations on the provided quantum programs which tends to lead to combining multiple gates together and performing a single overall transformation instead of multiple little ones [57], [58].

As an example, consider the circuit in Figure 2.4. In this circuit the qubit is first transformed by a Hadamard gate before also receiving a Pauli-X gate. Using the state table approach the status of the qubit would be represented in a table representing the possibilities of a $|0\rangle$ and $|1\rangle$ output. Using the Automatic Measure approach, after the first gate (Hadamard) the qubit is measured as either $|0\rangle$ or $|1\rangle$ before continuing to the second gate (Pauli-X). Because of the continuous measuring after each gate this approach only requires storing the current output state instead of all the possible amplitudes. It should be noted that both of these circuits produce the set of output states with equivalent amplitudes, though some optimisations may be required in larger examples.

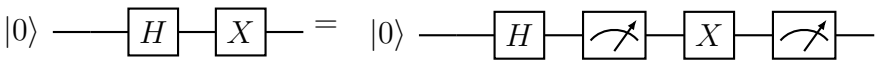


Figure 2.4: An example Quantum Circuit for the Automatic Measure

Overall there is no perfect approach, with each of the discussed approaches failing at varying points. The choice of implementation largely depends on the amount of time available for implementation and the overall purpose of the system. Use of the lookup table approach is simple in it's design and allows the developers to largely ignore the quantum nature of these computations. The lookup table approach with a specified size can be optimised and hard coded for the size, utilising matrix multiplication (e.g. for gate operations) and a prescribed measurement function based on the size. The Automatic Measure approach simplifies the memory management of the system requires a lot of attention in order to properly calculate the resultant memory state. Furthermore, it should be noted that according to the Invariance Thesis [59] these emulations of quantum computers must eventually fail to perfectly emulate the systems unless it can be shown that BQP is equivalent to P which would mean that quantum computers are equivalent to classical computers.

Programming Languages The earliest known discussion of a quantum programming language is from 1996 and includes a comprehensive discussion of quantum psuedocode [60]. Knill [60] discusses the concepts of creating and manipulating quantum registers as well as conditional statements. Since that time, quantum software has evolved according to 3 distinct areas of research [55]:

1. The Design of the Language
2. The Semantics of the Language
3. Verification and Analysis of Programs

Overall, regardless of the research direction the theory of quantum computer programming is typically split between two paradigms [55]:

1. Superposition of Data
2. Superposition of Program

Superposition of Data The majority of initial quantum programs will belong to this paradigm, if only due to the similarity to classical programming. Superposition of data follows the principle of "classical control, quantum data"[55]. Which boils down to the use of classical control structures of *if statements, For/While/Do loops* with quantum data being manipulated in place of the classical data. This method relates very closely to classical programming and is able to take advantage of many of the same advancements found in classical programming[55]. This paradigm is designed to work best with a Hybrid quantum computers, composed of a classical computer executing the program and instructing the quantum computer what operations to perform before returning the result back to the classical computer [55].

Superposition of Program A relatively new paradigm, superposition of program extends the previous paradigm and incorporates quantum control structures. Best summarised with the principle of "quantum control, quantum data" [55]. This paradigm is a recent shift, designed to take full advantage of the previously noted full quantum computers. This paradigm is completely different to classical programming and is therefore less common [55].

Both the superposition of data and superposition of program paradigms can be represented through the use of approved quantum operations (e.g. gates). This leads to any system being able to support the results of both paradigms as quantum circuits. Therefore as long as the system can support a universal set of quantum operations then it can support both programming paradigms equally.

Existing Applications As previously stated, existing quantum simulators are aimed at researching the evolution of quantum programming languages and the appropriate simulation of quantum systems. The later being a means to continue development while eagerly awaiting physical quantum hardware. Regarding the intended purpose, these quantum simulators are excellent implementations. Typically composed of something similar to Figure 2.5.

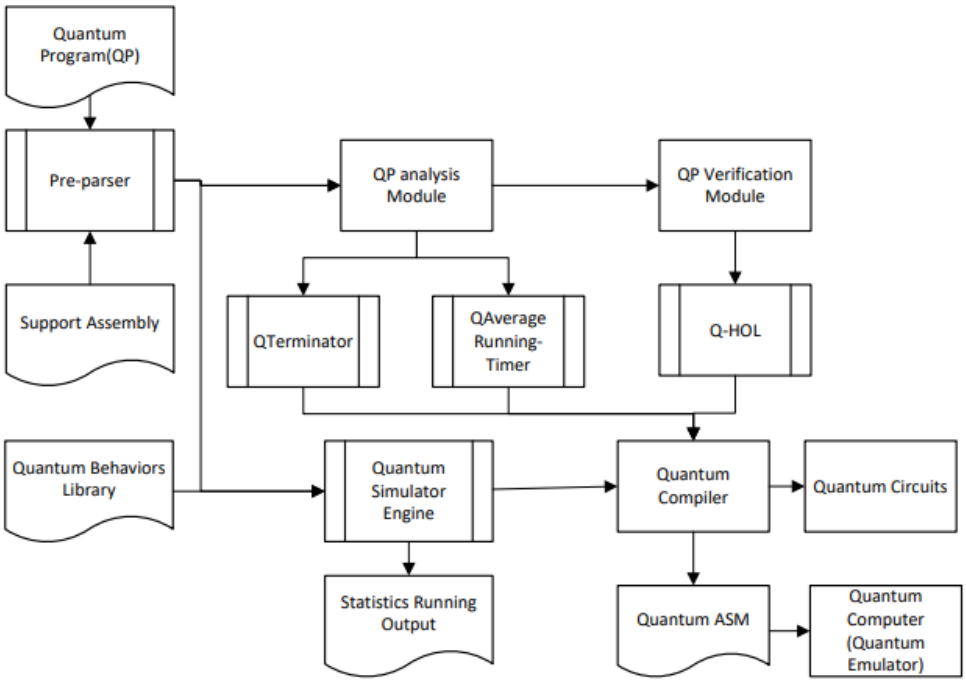


Figure 2.5: $Q\,|SI\rangle$ framework - Reprinted by permission from Springer Nature Customer Service Centre GmbH: Springer Symposium on Real-Time and Hybrid Systems [61] ($Q\,|SI\rangle$: A Quantum Programming Environment, Shusen Liu, Xin Wang, Li Zhou, Ji Guan, Yinan Li, Yang He, Runyao Duan and Mingsheng Ying), 2018 Springer Nature Switzerland AG (2018)

The software stack depicted in Figure 2.5 is the design for $Q\,|SI\rangle$ which is offered from the University of Technology Sydney where the focus was on [62]:

- Quantum algorithms and complexity
- AI applications of quantum computing
- Intermediate quantum computing and architectures
- Quantum programming and verification
- Quantum information theory and security

With these key focus areas, the software stack above is optimised. However this stack suffers from a flaw found in every original quantum simulator currently investigated (IBM Q [25], [63], Microsoft Q# [26], [64], Rigetti [24], $Q\,|SI\rangle$ [61]) it follows a **single job in, single job out** philosophy. This approach is exponentially more efficient and optimised for research regarding quantum algorithms, but ignores the fact that classical computers rarely completely focus on a single task. The typical load for a classical computer stretches into the hundreds of processes with multiple processes being executed simultaneously. Due to this oversight, current quantum simulators are not suited to the research that is being attempted here. It was this observation that led to the development of the simulator as per Chapter 5.

There has been some attempt to multi-thread the simulators, however that threading extends the simulator according to Figure 2.6. Notice that the compiling and setup is handled in a single threaded instance, before providing n copies of the compiled program to a thread pool for execution. By executing the same circuit multiple times the results can be tabulated and then a probability distribution can be estimated. Calculating the probability distribution in this manner saves time, however, it fails to consider the problems associated with concurrent execution of programs on actual hardware.

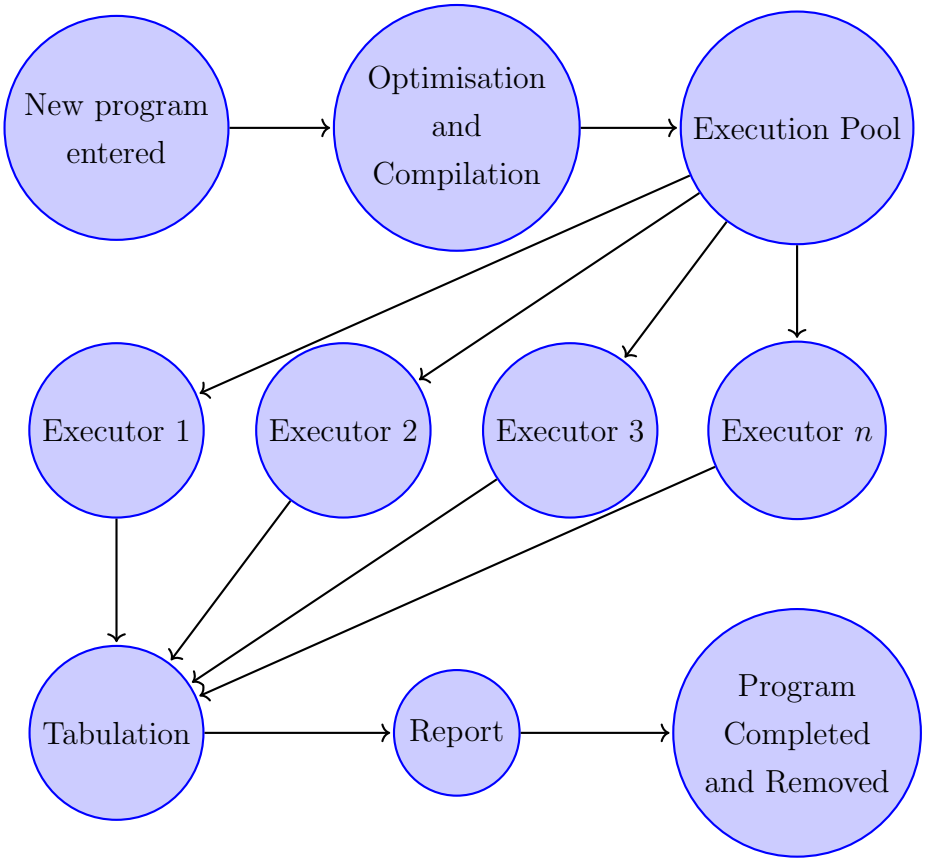


Figure 2.6: Faux Parallel (Thread Pool) Execution. The compiling and setup is handled in a single threaded instance, before providing n copies of the compiled program to a thread pool for execution. By executing the same circuit multiple times the results can be tabulated and then a probability distribution can be estimated.

2.1.4.3 Gate Computing as a middle ground

In order to join the hardware and software approaches together a middle ground must be reached. This is typically found to be in the base operations (i.e. logic gates) that the hardware must support, in order to support everything that the software implementation requires. The software implementations typically apply the logic found in the Divencenzo criteria [16] (explored in Section 2.1.1).

The logic gates are applied by multiplying the matrix representation of the qubit against the matrix representation of the gate. As an example, a qubit in state $|0\rangle = [1, 0]$ when multipled against the Pauli-X gate $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ the result is $|1\rangle = [0, 1]$ which is equivalent to performing a Not Gate.

A universal gate set is a sub collection of the logic gates which together cover all the possible operations of the entire set of logic gates. A Universal Gate Set of a quantum computer is typically associated as [27], [28], [65]:

- Hadamard
- Phase
- $\frac{\pi}{8}$ gate
- Control Not

Though if the Control Not gate is a more universal Control Gate then that can reduce the universal gate set.

While the full suite of logic gates is preferable (as explored in section 2.1.2.1), the universal gate set is what is required to support the software. It is worth noting that the universal gate set is not the preferred method to support software implementation [24], [25], [28], [43]. This is due to the sheer complexity and number of gates from the universal gate set required to implement the other more specialised gates which naturally results in longer execution times, higher error rates and unstable data. Therefore just because you can create every other gate using the above universal set, having access to other gates is recommended [24], [25], [28], [43].

This approach conforms with the established definition of a quantum computer, as explored in Section 2.1.1 and can be used to represent the standard digital computer logic gates if required (see Appendix B).

2.1.4.4 Algorithms

An algorithm is defined as "a set of well-defined logical steps that must be taken to perform a task" [66]. That definition explicitly ignores the technology used to execute the algorithm and focuses on the problem that the algorithm resolves. In the same spirit, researchers began working on quantum algorithms long before any usable quantum hardware was created.

> *"I work on quantum computing hardware and we are at the stage where we are developing small hardware prototypes that are still entirely useless. But in the medium term, what they will be useful for is to help us discover what can be done. Quantum computing is a really special field in the sense that we don't have quantum hardware of the scale that allows us to actually develop quantum computing applications on the basis of the hardware. I find it absolutely mind blowing and a testament to human ingenuity that we do have quantum algorithms."* - Prof. Andrea Morello [67]

Quantum algorithms vary in the specific approach they use but all of them tend towards the same overall process [43]:

1. Initialise the quantum bits to a specific classical state
2. Transform the quantum bits from the classical state into a superposition state
3. Manipulate the superposition through the application of a series of unitary transformations (logic gates)
4. Measure the system.

This process has created many quantum algorithms ranging from string matching [68] to number factoring [6] and everything in between. Quantum algorithms are typically focused solely on the mathematical approach

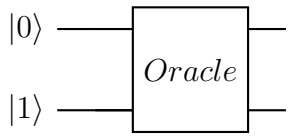


Figure 2.7: Example Quantum Circuit with an Oracle

(e.g. [6] and [69]) required to calculate the end result with the implementation left as an exercise for the reader.

A common tool used in quantum algorithms is an *Oracle*. Oracles are treated as black box components within a circuit which is different for each implementation. For example Grover's search algorithm [69] requires an Oracle which can (through a unitary matrix) result in a 1 for the target state and 0 for all other states [69]. Using this Oracle the algorithm can then manipulate the chosen state directly and leave the other states untouched.

As an example Figure 2.7 demonstrates how the oracle would be represented in the gate format. In practice the oracle found in Figure 2.7 would be replaced with a series of logic gates essentially making the Oracle similar to a method/function or black box component within the algorithm.

2.1.4.5 Combination of Quantum Hardware and Software

Physical vs Logical qubits Equation 2.3 defined a singular Qubit [28] which is then manipulated with operations like the logic gates from Section 2.1.2.1 in order to implement the algorithms specified above. This approach is idealistic but not currently realistic and is one of the largest issues in current quantum computers.

If the Qubits were as precise as the mathematical equation suggests then every time you implement a Hadamard gate you would assume that the qubit would move from Equation 2.3 to:

$$H(|\varphi\rangle) = \frac{\alpha + \beta}{\sqrt{2}} |0\rangle + \frac{\alpha - \beta}{\sqrt{2}} |1\rangle \tag{2.10}$$

However inaccuracies with the hardware (either physical or noise based) cause the results of operations to drift from the mathematical truth. As an exaggerated example the physical Qubit of Equation 2.3 could result in:

$$H(|\varphi\rangle) = \frac{\alpha}{2}|0\rangle + \frac{\sqrt{3}\beta}{2}|1\rangle \tag{2.11}$$

In this exagerated example instead of the qubit moving to a 50/50 balance of probabilities it moves to a 25/75 balance of probabilities which is not what the algorithm expects. Because of the nature of Qubits, it is not possible to retrieve the superposition without using a measurement which destroys the superposition and returns a singular result. Therefore the algorithms must work on blind trust that the operations are done accurately every time that they are used. Even if the quantum systems only differ from the expected result of an operation by a small amount as more and more operations are applied the drift will compound with each inaccurate operation.

In a standard digital computer the results of the operation can be double checked and then adjusted before continuing with the algorithm. Until physical quantum computers have this same capability or assurances the algorithms instead refer to *logical* Qubits instead of *physical* Qubits. Logical Qubits are effectively treated as perfect Qubits where the data never degrades and the operations work perfectly every time. Logical Qubits are currently composed of multiple Physical Qubits intertwined in some error correction mechanism. This exact nature of different error correction codes changes with each code but a simple example is to encode the same logical Qubit into 3 physical qubits and compare the state of each physical Qubit in a majority wins scenario [70], [71].

Noisy Intermediate Scale Quantum Computing (NISQ) A lot of the discussion found within this review has focused on having perfect quantum computers full of logical qubits. Unfortunately this is not the current reality as the field works through the Noisy Intermediate Scale Quantum Computing (NISQ) era [22]. The NISQ era is known for 2 large problems [22]:

1. Noisy (physical) Qubits
2. Difficulty with scaling up to larger computers

The first problem of noisy qubits has been discussed above but the problem of scaling is a new problem that needs to be considered. As explored above, it is possible to intertwine multiple physical qubits together to form a logical qubit so the simple answer is to simply add more qubits until everything is eventually error corrected. The problem with this approach is that the qubits need to be connected in a specific manner (decided by the chosen error correction code) which requires a redesign of the entire computer to accomplish this. Because of the interconnectivity between qubits it is difficult to modularise this approach which makes scaling the technology very difficult. To move beyond the NISQ era more accurate qubits will need to be developed and quantum computers will need to grow in size to be comparable to the digital computers we currently use [22].

2.2 Operating Systems

In order to properly investigate the current approaches of multi-processing a review of classical operating systems is required. This review explores the capabilities of an operating system and how they work together to accomplish their tasks. This grounding enables comparisons of classical operating systems and the system proposed in Chapter 4.

2.2.1 Operating System Definition

Operating systems can be considered through a number of lenses, each with a different set of responsibilities [72]. If you consider the Operating System to be the interface through which the computer and the user interact this obscures the sheer complexity of the operating system and can lead to considering the Operating System as a black box implementation. If you consider the Operating System through the alternative lens of the manager of the resources for the computer you quickly risk optimizing the computer

with a focus on efficiency by ignoring the user. The true definition of an Operating system is a combination of the different lenses, where management of resources is important but only in response to contexts and inputs provided by the user.

Operating Systems are perhaps best summarised as the low-level software that supports a computer's basic functions, such as scheduling tasks and controlling peripherals [51], [72]–[74]. In 1982, JR Mentzner proposed the hierarchical structure of operating systems found in Figure 2.8.

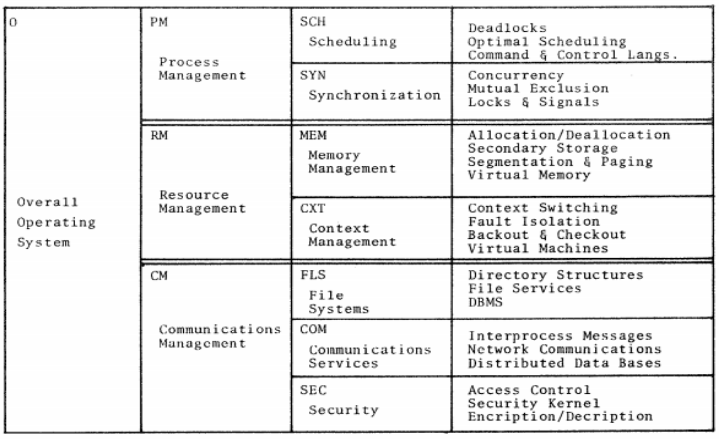


Figure 2.8: Operating System Hierarchy, adapted from [75].

The hierarchical structure is utilised to explore the relevant literature in more depth.

2.2.2 Process Management

Whether discussing an early batch system or a more complex time-sharing system, everything is conducted through processes. An operating system is a collection of processes, which must be carefully managed and executed in order to perform the correct actions. These processes vary from receiving messages from peripheral devices through to adding and deleting characters on the computer display. Often a single action, like a key press, is broken into multiple smaller processes which result in the desired outcome. Processes will

migrate between multiple states during their execution. The standard base representation of the different process states is [72] (Figure 2.9).

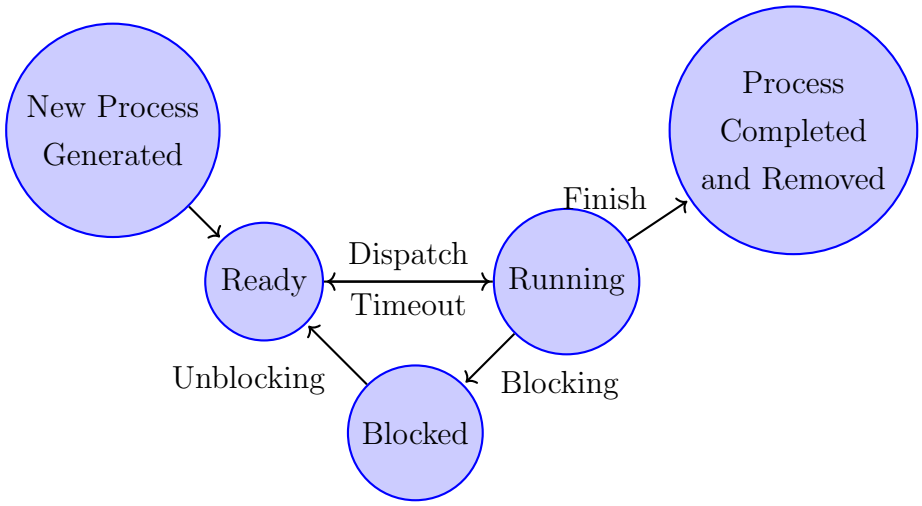


Figure 2.9: Basic Process States [72]

Where the process moves between ready, executing and blocked as required. Though this representation covers most cases, it assumes that all processes are stored in main memory. In the case that processes are too numerous to be stored in main memory, some must be migrated to secondary memory (suspended) while the main memory processes execute. To support this an advanced diagram has been developed which extends the design to include suspended processes [72] (Figure 2.10).

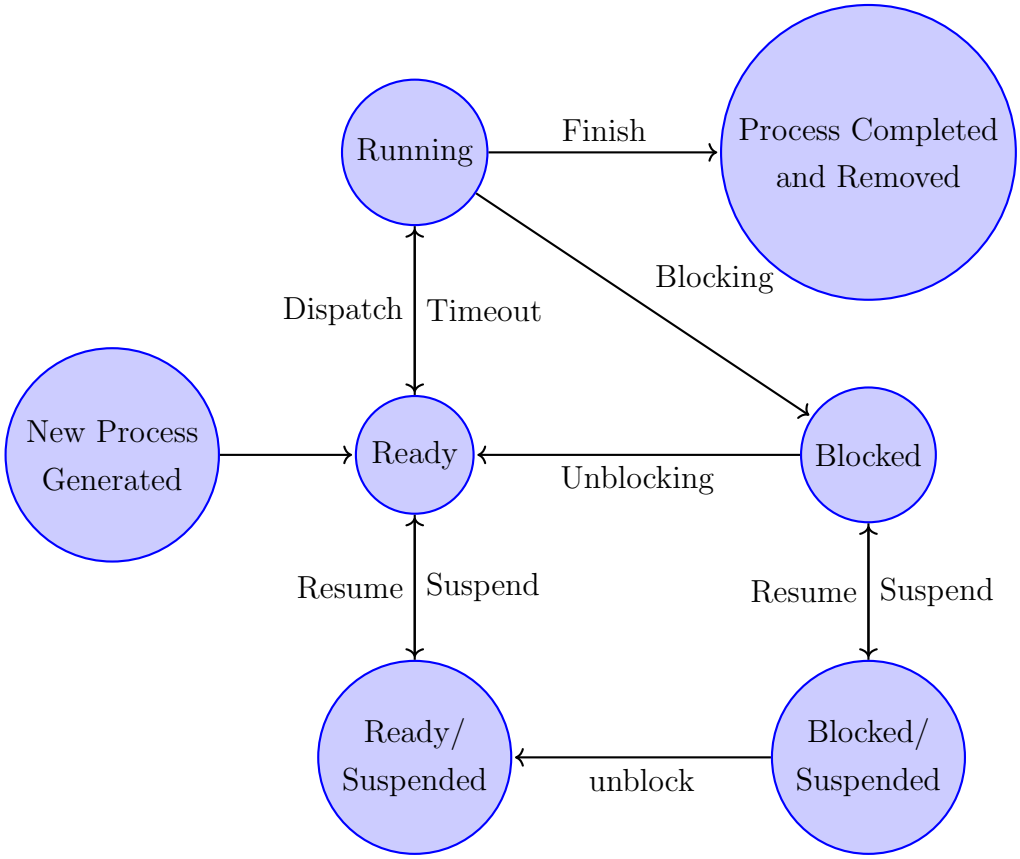


Figure 2.10: Advanced Process States [72]

A third model is used within Unix operating systems [72]. This model enables the distinction between system processes and user processes within the same model (Figure 2.11).

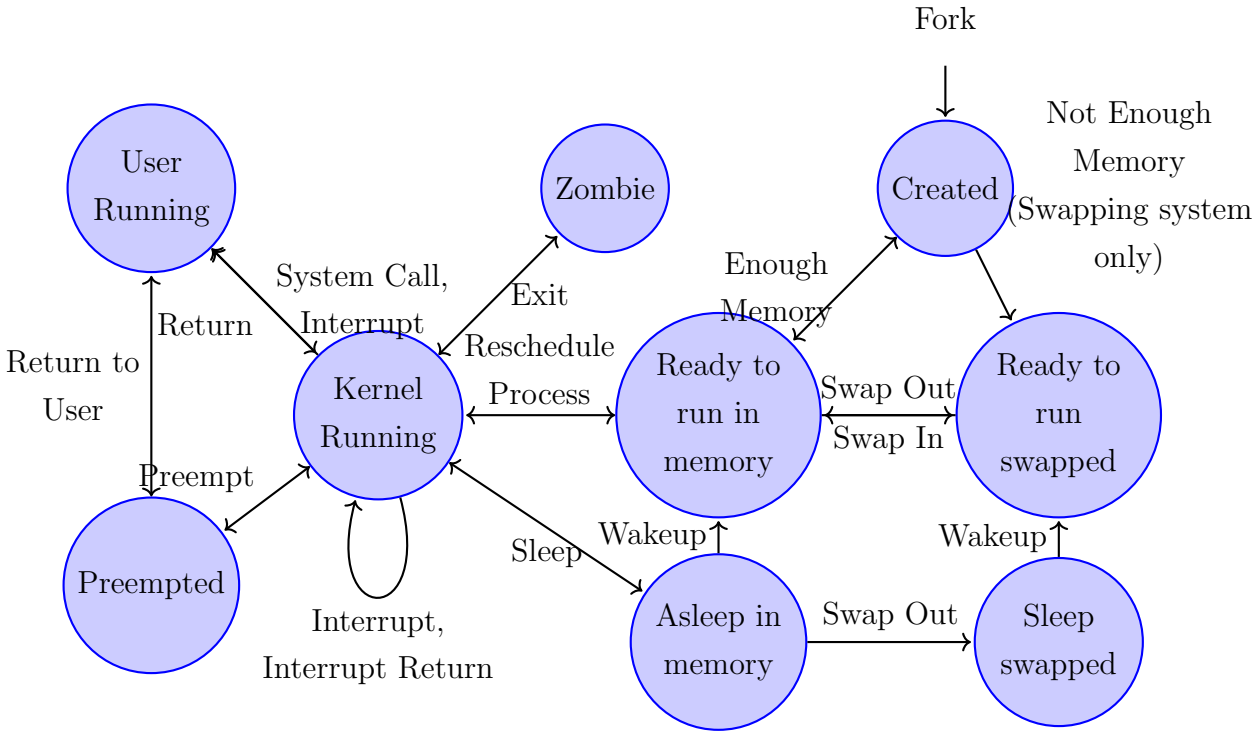


Figure 2.11: Unix Process States [72]

Regardless of which model is implemented into the operating system the consistent parts are the transition between *Ready*, *Running* and *Finished* states. If the system works in a preemptive environment then the system switches from a one way flow to allow for processes to migrate backwards and forwards through the states [72]. The addition of long term computer memory allows the introduction of suspended processes to the model [72]. Understanding whether the new system works in a preemptive or non-preemptive environment and whether long term memory is available dictates which model should be applied.

2.2.2.1 Task scheduling

Within process management is the ability to schedule multiple processes for execution, according to the operation of the computer, and expect that they will be executed in a timely manner [51], [72]–[74]. Processes vary in size and complexity with larger tasks like copying a paragraph of text typically broken down into smaller processes like writing a single character, or deleting a single character. These processes are then compiled into a specific order and executed according to that order. This scheduling is typically reduced to one of the four main algorithms: [51], [72]–[74]

1. First In First Out (FIFO)
2. Shortest Process First / Shortest remaining time
3. Round Robin (RR)
4. Priority Queue (PQ)

Where each algorithm aims to optimise the efficiency of the computer. Another consideration of scheduling is the number of execution streams available to the computer.

Single processor Previous incarnations of classical computers utilised a single constant execution stream. With processes and data being loaded into the stream to be completed. This method simplifies the execution of processes by only having a single process executing at any set time, and therefore only having peripherals and memory locations in use or not in use. The scheduling portion of this implementation is largely limited to the order in which the processes are executed. [51], [72]–[74]

Multiple processors Multiple execution streams allow for the execution of n processes simultaneously. This adds an extra level of complexity with multiple processes vying for different peripherals and memory locations. This

added complexity can lead to issues associated with computer deadlock. Research in this area largely focus on optimal scheduling algorithms, and deadlock detection and resolution. [51], [72]–[74]

With most systems providing only limited implementations of each resource, for example I/O devices and processors, the management of their usage is typically accomplished through the use of the following factors [72]:

- Fairness - All processes competing for resources should be given fair and equal access to resources where able.
- Differential responsiveness - The Operating System should react to the varying requirements of the processes, scheduling them to accommodate the relevant requirements.
- Efficiency - The Operating System should attempt to maximise the amount of processes being processed, while minimising the wait time for processes.

The above factors are in clear conflict with each other, therefore Operating Systems must attempt to find a balance between them.

Another consideration for the Operating System is how to organise the processes for the processors. A common approach is to construct a singular queue (or heap) of processes and simply assign the next process to the available processor as each processor becomes available [74]. This approach works well for similar (generalised) processors however specialised processors (such as the new M1 chipset [76] or I/O processors [72]) do not perform well with this approach. This is due to each specialised processor performing better for their specialised processes but performing poorly for the other processes. An alternative approach is to use multiple queues up to a pairing of 1 queue to 1 processor, thus allocating the load better [72]. The downside with this approach is that the processes are assigned to a subset of the processors and it is easy for some of the processors to become idle after they work through their queues while other processors struggle to complete their

queues [72]. As the alternative approach is an optional improvement to the singular queue system, the system designed in this Thesis will feature the single queue philosophy. This is due to Quantum Computers only featuring a unique processors as resources model while also minimising the complexity of the new system.

2.2.3 Resource Management

The second pillar of operating systems is resource management. Resource management is composed of memory management and context management. Memory management is concerned with the manipulation of data both in and out of memory. While context management is concerned with all things relevant to the individual programs. [51], [72]–[75]

2.2.3.1 Memory Management

The ability to create, access and manipulate values currently stored in memory is a critical component of an operating system. It is the manipulation of these values that are outlined in the processes explored above.

This memory management is typically accomplished according to the following responsibilities [72]:

- Process isolation - Processes are to be kept independent, and are not to interfere with each other.
- Automatic allocation and management - Memory manipulation should be automatic, and based on process requirements.
- Support of modular programming - Programmers can define programs into modules, which are created, deleted and altered dynamically.
- Protection and access controls - Memory must be protected from unauthorised access, and all authorised access must be closely monitored to ensure it does not adversely affect the processes.

- Long-term storage - The operating system must be able to differentiate between long term storage and short term storage, as well as provide long term storage to programs as required.

Modern multiprocessing Memory Management systems tend to have similar requirements, including [72], [74], [77]:

- Relocation - The ability to transport a processes memory from one memory location to another. It is expected that the entire process will be moved safely and this is commonly used when a process grows beyond its memory allocation or when a process is swapped out to the ready state.

- Protection - Processes are to be kept independent except where they are explicitly required to share data which is closely monitored.

- Sharing - There must be a mechanism to allow multiple processes to access the same memory with proper supervision. This is often found in a program that spawns multiple processes with a common data pool.

- Logical Organisation - Programs are segmented into logical components known as modules which are then logically organised within the memory system. This is accomplished by grouping common modules together to allow for faster loading and swapping

- Physical Organisation - This requirement deals with the transfer of data between the different physical memory systems including Main memory and long term storage.

While these requirements have been developed over many years of trial and error, they do contain an assumption towards modern computing technologies. This is evident in the discussion of long term storage and the ability to move memory between locations. As this discussion turns towards quantum computers a series of more technology independent requirements needs to be considered. At a minimum a classical memory management system needs [74]:

- Allocation - The ability to load data and instructions in and out of memory and place them at specific locations.
- Abstraction - Knowledge of the computers memory and what is available or taken.
- Isolation and Sharing - Process isolation and protection from unauthorized interactions.

While the other requirements mentioned above are good to have they are non-essential requirements.

Every computer, whether classical or quantum, has a finite amount of memory to distribute (Classical has bits, quantum has qubits). This memory must be fragmented and allocated to processes as required, before being recalled after the process has completed. This is accomplished through numerous methods, with the typical classical computer approaches of [51], [72]–[74]:

1. First Fit algorithm (Find the first suitable available space in memory)
2. Last Fit algorithm (Find the last suitable available space in memory)
3. Best Fit algorithm (Find the available space in memory closest in size to the requested space)
4. Worst Fit algorithm (Find the largest space in memory)
5. Buddy Fit algorithm (Recursively break the memory into chunks of equal size, then perform best fit)

This practice needs to balance the needs of the processes and the available memory. This can, if not done correctly, lead to conditions of deadlock. Classical computer memory is often considered as an infinite tape containing a series of interchangeable cells, these cells all perform the same functions of store, read and write and the only difference is the location of the cell

within the tape [17]. Conversely, quantum computer memory is a not interchangeable and is often represented in a 3 dimensional graph structure [17]. This fundamental change in underlying structure and lack of universality means these algorithms are not suitable for quantum computer memory management.

2.2.3.2 Context Management

Context management is concerned with maintaining the execution of processes between time slices. When a process is removed from execution before it can complete the assigned work, numerous details must be recorded, these typically include [72]:

- Instruction counter - Current instruction
- Program data/memory
- Program location in memory.

failure to properly record these values will require the process to either restart, or the process could potentially corrupt [51], [72]–[74]. Due to the distinct lack of long term Quantum Memory it is not feasible to store the program data/memory and this component is not included in the Thesis.

2.2.4 Communications Management

Communications management is composed of three main sections, File systems, Communications services, and Security.

2.2.4.1 File Systems

In classical computing, files are designed as collections of related data segments [51] which are coordinated by a file management system. This file management system is responsible for providing access and relevant restrictions, while also maintaining the location and validity of files. This is applied through different approaches dependent on the operating system supporting it. The most common file systems are:

Virtual File System The Linux operating system is typically known for implementing a Virtual File System (VFS), the VFS allows for Linux to build on top of an existing implementation of a file system. The VFS works by taking requests from the user and then converting the requests into the relevant requests for the underlying implementation. This allows Linux to have a vast amount of portability, including running from a USB (Universal Serial Bus) [51], [72]–[74].

New Technology File System Windows operating systems use a proprietary format known as New Technology File System (NTFS), based on the needs of the typical workstation. The design of NTFS is remarkably simple, breaking into the following four regions:

1. partition boot sector
2. master file table
3. system files
4. file area

This approach allows for recoverability of files back to a consistent state, securing and handling larger files [51], [72]–[74].

Due to the distinct lack of long term Quantum Memory it is not feasible to store the program data/memory and this component is not included in the Thesis.

2.2.4.2 Communications Services

Communications services are responsible for all communication throughout the system. This includes messages between distinct processes, networked system and distributed programs/files. This section is critical to the operation of current classical computation, without it, classical computers would not be capable of interacting with the internet, other computers or coordinating parallel processes internally.

Communication between multiple distinct processes can take many forms, though in practice is either temporary or permanent [78]. Temporary communication only exists while it is being transmitted and is therefore suited for synchronous processes, an analog example of this could be pigeon carriers or radio transmissions as these are only detectable or receivable for a brief period before disappearing. Permanent communication however is visible from the message send until it is overwritten by another message, therefore suited for either synchronous or asynchronous communication, an analog example of this could be flying physical flags.

Without the ability to pass messages between concurrent processes, classical computers would be restricted to processing tasks sequentially. For example, if the process needed to step through an array and add +10 to every cell, the only conceivable approach would be to use a single process and move sequentially. Attempting to split the task into multiple processes is doomed to fail because processes cannot inform each other when they have completed.

The concept of computer networking extends the concept of interprocess messages to intercomputer messages. Computer networking began in the late 1950's with the ARPANET project [79], attempting to introduce the redundancy required for the United States to survive damage which could be sustained during wartime. ARPANET operation began in 1969 with only 4 nodes and is responsible for most of the concepts still used today like segmenting messages into 'packets' [79].

Together, interprocess and intercomputer messages combine to allow for distributed computing. Distributed computing is designed to segment operations and allow for distinct elements to be computed either on different parts of the same computer ('cores') or on separate machines entirely. Common issues include, message latency [80], concurrency and partial failures [81].

2.2.4.3 Operating System Security Features

The security section of the operating system is responsible for managing the use of the system. This is accomplished through encrypting and decrypting files as required by the users, and by implementing access control mechanisms such that only authorised users can interact with the relevant data [51], [72]–[74].

Operating systems are designed to manage and provide information to the relevant users. To this effect, a complete system must incorporate a balance between the following categories [72]:

- Availability - The system should always be able to provide the required data, regardless of interruptions. This area is concerned with issues like Denial of Service attacks.
- Confidentiality - Data must be restricted to the authorised parties, unauthorised access must be strictly rejected.
- Data integrity - The integrity of data is paramount, typically coupled with Confidentiality to protect against unauthorised manipulation of data.
- Authenticity - Testing for the authenticity of data, messages and identification of users.

Threats to security are typically either system threats, or malicous software [72], [82].

System Threats System threats are classified as threats that are internal to the base system. In 1980 Anderson [82] classified these threats into three distinct categories:

1. The Masquerader
2. The Legitimate User
3. The Clandestine User

The Masquerader is an entity which appears to be a legitimate user, thus defeating the security countermeasures [82]. The Legitimate User is a legitimate user of the system, however they have engaged in misfeasance [82]. This misfeasance can range from accidentally abusing their access levels, through to deliberately accessing and sharing data with non-cleared parties. Lastly the Clandestine User is an entity who has control of the security countermeasures and can therefore disguise their tracks [82]. A correctly designed system should remain secure in the face of all three of these actors.

Malicous Software Malicous software is commonly referred to as Malware or Viruses. While numerous variations of malicious software exist, they all tend towards similar designs. These designs include: [51], [72]–[74]

- Trojan Horse - a program designed to perform a standard function whilst also breaching the security of the host computer.
- Viruses - a program designed to alter the way a host computer operates and which spreads amongst computers.
- Worms - a program which reproduces itself to spread between computers.
- Spyware - a program designed to gather data about you and your activities before providing it to a third-party without your knowledge or consent.
- Rootkit - a collection of computer software, designed to enable access to parts of a computer system that is not enabled for the user and often masks its existence or the existence of other software.

Both System Threats and Malicous Software must be handled by the operating system. Failure to accomplish this can lead to an outcome between simple user annoyance, through to a completely unusable system. Quantum computers are currently very nascent systems in comparison to classical computers, due to this short lifetime little malicious software exists for a Quantum Computer. Current malicious quantum software works through

improper memory management, thereby initiating uncontrolled communication between processes [83]. A correctly calibrated operating system must, by design, eliminate (or vastly reduce) this threat.

2.3 Existing Quantum Computer Operating Systems

2.3.1 Existing Research (Show the limited research here)

This is not the first time that quantum computer operating systems have been mentioned in literature, that honour belongs to Corrigan-Gibbs et al. [84]. Though their work is the first to analyze the potential of quantum computer operating systems, their work is "necessarily (and shamelessly) speculative (p. 1)" [84]. Corrigan-Gibbs et al., spend their paper discussing the end uses of a quantum operating system, skipping past the actual design and implementation of the system. Due to this skipping of detail, their aspirational paper reads better as an Application Programming Interface (API) reference than an outline of a new system. Whilst the work presented here demonstrates the first attempt to fully describe the parallelisation of quantum programs on quantum hardware. It also appears that no further work has been done on this project since the publication of that paper.

2.3.2 Existing approach

Quantum computers and quantum computer simulators have now moved from specific laboratories to now being available for use by the general public. To accomplish this research groups have elected to use the default Single Job In Single Job Out (SJISJO) approach. The SJISJO philosophy can be seen in figure 2.5 by the absence of supporting technology.

In Figure 2.5 the quantum program is loaded into the system through the Pre-parser before moving directly towards the simulator engine and the inevitable execution. There is clear evidence of quantum program analysis through the analysis and verification modules but there is no point in the design where the concept of a queue is introduced and seemingly no way to process more than one quantum program at a time.

2.3.3 Research Gap

Quantum technologies are set to change the world in numerous ways, both known and unknown. Significant research has already been completed in the fields of quantum computing and related technologies. This chapter reviewed the current literature and assumed knowledge for the reader. Significant problems have been identified in quantum hardware design, including operating temperatures, stability and error management. Conversely, numerous strides have been made in quantum software as algorithms and use cases for quantum computing seem endless.

Currently quantum computers are able to execute quantum programs in a sequential manner. The current approaches work, and have been invaluable for development and testing of quantum hardware technologies. The problem with current approaches is the limitation of sequential execution. Sequential execution leads to the following issues:

1. Increased runtime required to clear the program queue, which by extension requires spending further resources to maintain the quantum computer.

2. Wasted resources due to only using part of the quantum computer.

The research presented in this Thesis attempts to resolve these issues by designing a quantum operating system and then evaluating the proposed solution. After reviewing the current literature, there are a series of capabilities available in traditional computing that are not available in quantum computing. These capabilities include:

- Multiprocessing
- Networking between quantum computers
- Security from cyber threats

Traditional computers have long been able to act on multiple processes at once and have used this to enable the multi-tasking that many take for granted. Without this capability users would be restricted to a single process at a time, no more writing documents while simultaneously performing calculations.

The ability to communicate between traditional computers and therefore co-ordinate their actions is core to the daily usage of traditional computers. From this capability the concepts of sending messages (Email) and documents (File Sharing) were born (among others). Use of this capability can allow for multiple smaller traditional computers to function as a larger composite computer in order to complete the assigned tasks, this is known as a Beowulf cluster.

Existing single operation quantum computers are strongly limited in terms of cyber security threats. This is due to the computer only executing a single process at a time which removes process interference threats and only leaves physical hardware threats [85]. Implementing either multiprocessing or networking will however extend the use cases of quantum computers. These extensions allow for increased performance but also increase the risk of malfeasance occurring. It is expected that the design and implementation of a quantum operating system should include either multiprocessing, networking or both and must therefore attempt to combat the cyber threats that can now occur.

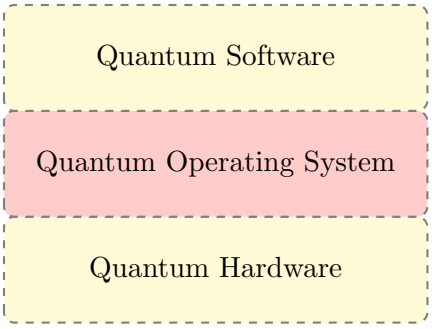


Figure 2.12: An example of the Quantum Software Stack

The Quantum Operating System explored throughout this Thesis is intended to replace the Quantum Operating System layer in Figure 2.12. This inclusion will combine the software and hardware layers that currently exist into a singular software stack ready for use.

3. METHODOLOGY

In order to perform any research an understanding of the methodology is required. This Chapter begins with outlining the strategy being pursued before exploring the research paradigms and the viewpoint used throughout this research. Finally an overall research approach is outlined before this Thesis continues with presenting the research.

3.1 Research Strategy

3.1.1 Design and Creation

The design and creation strategy is focused on designing and subsequently developing new IT systems or products [86]. This strategy is a formalisation of the stereotypical approach featured in IT departments and businesses. This research strategy is composed of the following steps [86]:

1. Awareness - What is the problem?
2. Suggestion - How could this problem be fixed?
3. Development - Implement the solution
4. Evaluation - Examine the solution and assess the worth of the system
5. Conclusion - Consolidate the results, knowledge gained and tie any loose ends together.

These steps do not need to be followed in a strict linear fashion, but rather are typically iterated over recursively to resolve a single problem. This strategy can also be deployed in a stepwise refinement approach to resolve each

sub-problem separately until the overall problem is resolved.

Evaluation of this strategy is complicated due to the wide range of possible outputs. When evaluating the resultant product of this strategy there are numerous criteria one can consider, including the following [86]:

1. Functionality - Does it perform the required function?
2. Completeness - Is the solution complete? Or is it missing parts?
3. Consistency - Does the same input always produce the same output?
4. Accuracy - Is the output of the IT system correct?
5. Performance - How many resources does it consume? (Memory, Time, etc.)
6. Reliability - Does the solution always work?
7. Usability - How user friendly is the solution?
8. Accessibility - Can people get to and use the solution? Is the solution developed in accordance with universal design principles?
9. Aesthetics - Does the means to access the solution appeal to the user? (Colour choice, text size, font, language choice, etc.)
10. Entertainment - Does the solution have any entertainment value to the user?
11. Fit with organisation - Does the solution fit within the organisation values or expectations?

Evaluating the product can take the form of black box testing (looking at input and outputs), white box testing (looking within the product and tracing the execution) or comparisons to alternative programs or scenarios. There are several techniques which can be used for these evaluations, *Proof of Concept* is a technique which focuses on purely producing a product to demonstrate

that something can be produced [86]. This could take the form of testing a new theory or trialing a new method to solve a problem. Alternatively *Proof by Demonstration* focuses on the use of the product by various groups and/or in various contexts [86]. Lastly *Real-world Evaluation* examines the product within the context that it will actually be deployed in, instead of the extensively curated alternative artificial environments [86].

3.1.2 Shanks Theory of Systems

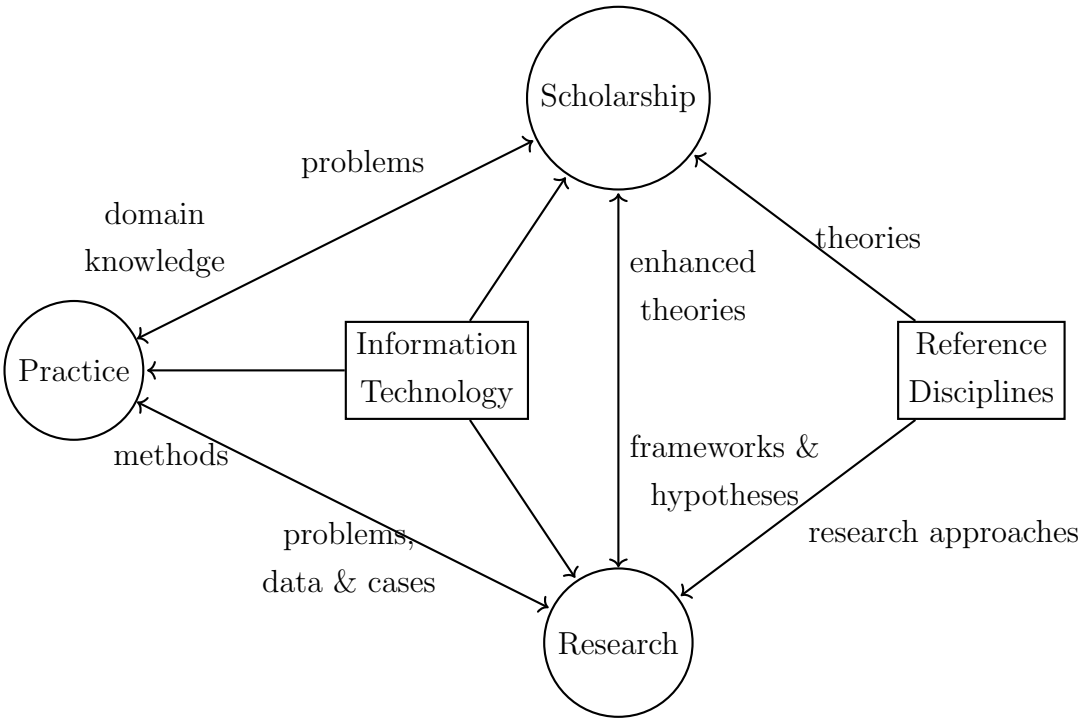


Figure 3.1: Shanks Theory of Information Systems adapted from [87]

Shanks et al. [87] proposed the Shanks Theory of Information System framework found in Figure 3.1 for research surrounding information systems. Shanks defines the terms Scholarship, Research and Practice as follows:

Scholarship "the process of systematizing existing knowledge relevant for a discipline." [87]

Research "A systemic process of acquiring new knowledge" [87]

Practice "The knowledge from practitioners within industry." [87]

Thereby seperating the distinct components of the complete information systems discipline. Shanks [87] continues by expanding research into the following three variants:

Exploratory Initial research, looking to see if an area yields interesting/useful data. (e.g. designing a new programming tool)

Descriptive Research which attempts to describe all parts of a situation. (e.g. UML (Unified Modeling Language) and diagramming of the systems)

Explanatory Research which attempts to explain what happened. (e.g. Debugging/backtracing)

In accordance with the above framework, the research presented within this Thesis is exploratory. The last step of the research strategy is to define the paradigm associated with the research.

3.1.3 Continuum of Research

The continuum (Fig. 3.2) of research [87], [88] is a graphical means to represent how the different research approaches and strategies place regarding quantitative, qualitative and paradigms. The continuum demonstrates that while the concepts of objectivity and subjectivity exist on separate sides of the continuum, there is no truly subjective or truly objective research strategy. Researchers will view their research according to the paradigm (above the continuum) which anchors them in objective or subjective territory, while their choice of research strategy will determine their actual position on the continuum.

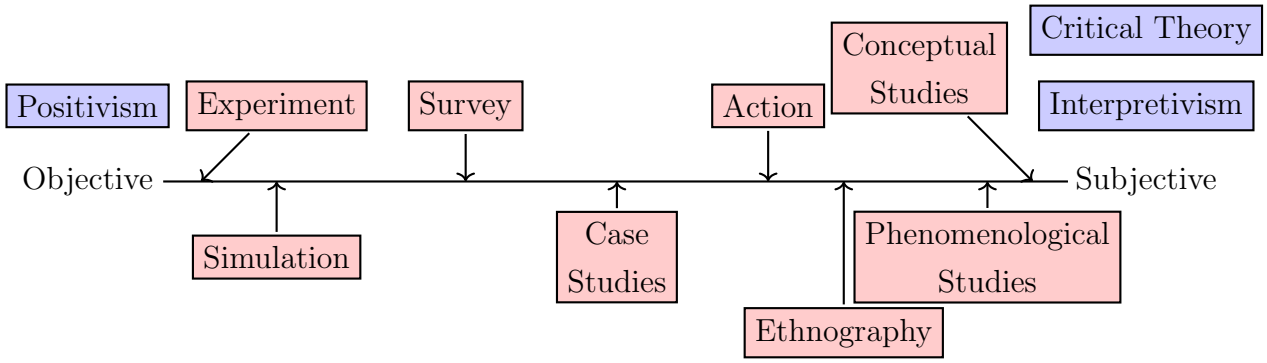


Figure 3.2: Continuum of Research [86]–[88]

3.2 Paradigm and Experiments Choice

Given the gap statement from Section 2.3.3, the research will make extensive use of computer models and programs. These models extend to include rudimentary implementations of principles of quantum mechanics which by their definition are non-deterministic. These models are designed to represent a quantum computer which can receive and execute a quantum program. While the quantum mechanics are non-deterministic, the computer models themselves are deterministic because they co-ordinate and manage the quantum programs. Because of this deterministic nature the **exploratory** research is found within the **positivism paradigm** with the major strategy being **design and creation**. The experiments and specifics of this research are found in Section 3.3.

3.2.1 Critiques of the Positivism Methodology

Werner Heisenberg, Niels Bohr and Wolfgang Pauli had concerns regarding Positivism [3]. Their concerns can be distilled to the following points:

1. What is knowledge and understanding within Positivism?
2. Positivism is inherently blinkered.

The first concern looks at what it is to know or understand anything according to Positivism doctrine. Heisenberg is quoted as saying “The posi-

tivists would probably claim that 'understanding' is tantamount to 'predictive ability'..." (p. 206) [3] an argument supported by Neuman as late as 1991 [89]. The argument being whether because we can predict where a pen will land (after rolling off a table) we can proclaim that we understand the forces at play. Further, if somebody simply takes this equation (for prediction of the pen) and plugs in the variables, can they also say they understand the problem? Heisenberg contends that true understanding cannot be guaranteed from predictive ability citing the example of Ptolemy's astronomy predictions which while accurate, were built on the presupposition that the earth is the center of the universe [3]. It was not until Newton applied the laws of inertia and gravitation that the concept of a helio-centric universe triumphed. Can it be said that Ptolemy understood the planets when his model was so largely invalid?

Niels Bohr simultaneously argued that while he could "... readily agree with the positivists about the things they want, but not about the things they reject." (p. 207) [3]. Bohr argued that positivists were so completely obsessed with their specific facts that they largely ignored the overall picture. Bohr agreed with the use of testing on provable hypotheses, the practice of which served to largely eradicate superstition from scientific inquiry, however disagreed with the rejection of unfalsifiable concepts. Bohr specifically indicted the French Academy for their total rejection of "stones falling out of the sky" (p. 208) [3], because stones by their nature (according to positivists) cannot be in the sky unless someone threw them upwards. This rejection continued until an excessive number of meteorites landed near to Paris, forcing the positivists to adjust their views [90].

Considering the critiques above, the choice to utilise the positivism paradigm does not change. The critique questions whether one can ever truly understand anything, providing compelling arguments to that effect. This part of the critique can be ignored for this specific research because the research does not seek to understand new phenomena, rather to apply researched and documented phenomena in a more effective manner. The second critique

questions the nature of positivism to reject data and phenomena prematurely. This research stays within the bounds of already accepted research, but accepts that new/updated phenomena may yield higher efficiency methods. Therefore the paradigm and experiments mentioned above in Section 3.2 is confirmed as the approach for this research.

3.3 Research Approach

The research presented in this thesis begins with an exploratory constructive approach, by designing and constructing a system to fill the lack of multiprocessing, networking and cyber security threats (see Section 2.3.3). As this is a novel problem and has therefore not been studied before, there is limited literature and data sets available for accurate comparisons.

An initial quantitative analysis is provided in Chapter 4. Then moving to an experimental review of this system and its distinct parts in order to better quantify the power and capability of this new system.

Chapter 4 explores the missing capabilities identified above. This chapter outlines the theoretical design of the system constructed in this research. This system is designed to be self-containing and able to execute on whatever quantum computer is available. Chapter 5 then focuses on the design, structure and development of the test program used in the subsequent chapters.

Chapter 6 combines the theoretical and practical elements presented in Chapters 4 and 5 for an in depth analysis of the overall system. This chapter also features a comparison test between the system developed in chapter 5 and other external systems currently available in the market, featuring Microsoft Q#, IBM Qiskit and Rigetti Quilc/QVM. Data for this analysis is generated according to the follow procedure:

1. Outline the different approaches to solving this problem

2. Generate data sets according to the expected input data and the ex-

pected variability within that data.

3. Execute each algorithm (individually and sequentially) over the data sets, recording results according to the measurements specified in the analysis.

4. Compare and contrast the measured results in order to evaluate the appropriateness of each solution, before recommending solutions according to the evidence.

Chapter 7 expands the base system by investigating improvements for the base system.

Chapter 8 reviews alternative configurations for the operating system, exploring how the base system would respond to the various configurations commonly seen in traditional computing.

Chapter 9 analyses the previous chapters product with a focus on system security.

Finally, Chapter 10 completes with a review of the research conducted and a discussion of future work.

3.4 Chapter Summary

This Chapter outlined the approach and considerations taken with this research. The research performed through this Thesis takes the form of **exploratory** research within the **positivism paradigm** with the major strategy being **design and creation** as established in Section 3.2. The following chapters outline the developed system, the implementation of that system and then moves into an ongoing discussion about the various features and considerations of the system.

4. BASE OPERATING SYSTEM DESIGN

The majority of this chapter has already been published [91].

4.1 Introduction

Quantum computing stands as the next evolution in computing paradigms, able to augment current computing with a fundamentally new approach. Current work in quantum computing focuses on either creating the physical hardware (from a range of possibilities)[16], [63], [92]–[95] or looks at utilising this new paradigm through software applications [6], [96]. Work has been done on optimising quantum programs [27], [97], [98] and mapping their resource allocation [99]–[104] so that it can be executed on the quantum hardware. Using the current state of the art implementations, quantum computers only execute a single program before moving onto the next program. This approach is reminiscent of batch processing in classical computing however it fails to actively utilise the complete power of the quantum hardware.

The current brute force cost to optimise and map a quantum program is so expensive that to map multiple quantum programs together is seemingly intractable. The approach presented in this chapter tackles this problem and presents an approach which results in simple parallel execution of quantum programs. The approach utilises the information required for sequential execution including program dependencies, program mappings and qubit connectivities, reshaping it to elicit novel data which enables the simple parallel execution. In the presentation of this approach Research Questions 1 and 2 are resolved.

4.2 Limitations of Quantum Computers

Some components of a classical operating system cannot be transferred to a quantum operating system due to current limitations with quantum computing. Qubits are currently limited to small execution times due to decoherence and other sources of error [28], [43], [54]. This limited execution time results in a lack of long term quantum memory ensuring that no quantum file systems can be implemented [28], [43], [54]. Due to the low execution time it is currently advised to treat quantum programs as singular blocks, this approach removes the luxury of context switching and limits the synchronisation to allocating qubits to programs without currently considering shared qubits.

While quantum networking [54] expands the capability and functionality of a quantum computer in a similar fashion to current computing systems, it is not required for a quantum computer as per the definition found in Section 2.1.1. Quantum networking has therefore been intentionally omitted from the approach presented below.

4.3 Quantum Process States

Traditional computing processes, maneouver through various states as they compute. This is demonstrated in Figures 2.9, 2.10 and 2.11. These states demonstrate the various evolutions of a process during its journey throughout the computer. Figure 4.1 demonstrates these states for a quantum computer process. The major difference between Figure 4.1 and Figures 2.9, 2.10 and 2.11 is the directionality of the diagram. Figures 2.9, 2.10 and 2.11 utilise a bidirectional system where processes can be loaded on and off of the computer essentially pausing part way through computation, Figure 4.1 alternatively demonstrates that once a quantum process is loaded, it must run through to completion. Figure 4.1 could be updated by allowing running processes to be removed, though unlike Figures 2.9, 2.10 and 2.11 the data associated with that process could not be saved and restarted later due to the no cloning

theorem [28], [43] (see Section 2.1.4.1) and the lack of available quantum memory, it would instead need to be completely wiped. This would require the entire program to be started over, essentially wasting the time previously spent computing.

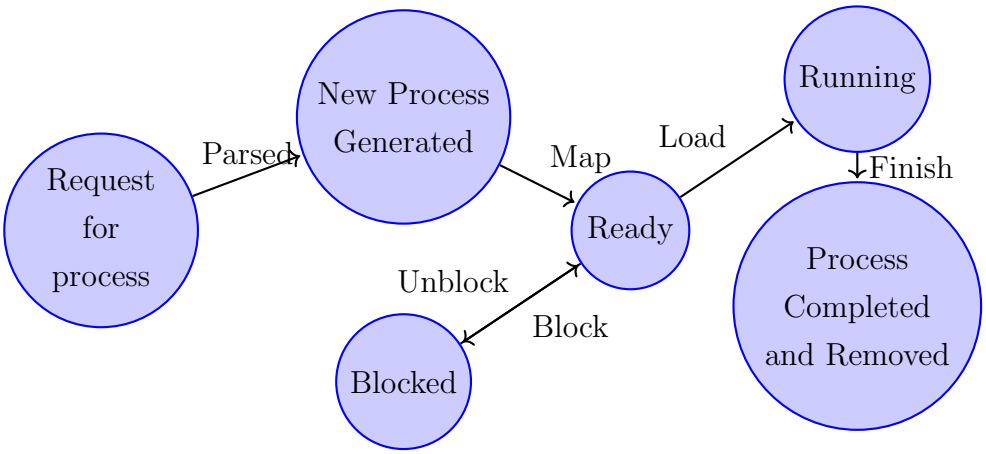


Figure 4.1: Basic Quantum Process States

4.4 Validity of Concurrency

It is well established that quantum computers are capable of executing a quantum program on their hardware, depending on error and connectivity (the connections between qubits) [28], [43]. What is typically overlooked is the ability for a quantum computer to perform parallel, or concurrent computations. The basic structure of a quantum program is a circuit similar to Figure 4.2 which is a toy example of entanglement but serves the purpose of an example quantum circuit with which to demonstrate concurrency.

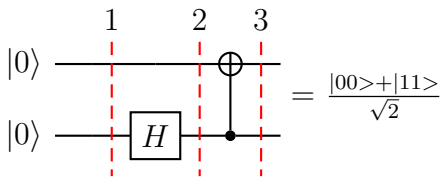


Figure 4.2: An example Quantum Circuit, specifically an example of generating a basic entanglement state, commonly known as a Bell State ($|B_{00}\rangle$). The horizontal lines represent an individual physical qubit and track the operations performed on each qubit. The left hand side of the circuit demonstrates that both qubits begin in state 0 ($|0\rangle$) and end in state $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$. The vertical red dotted lines are 1) the original state of the circuit, 2) the circuit after applying the single Hadamard (state = $\frac{|00\rangle+|10\rangle}{\sqrt{2}}$) and 3) the circuit after applying the Control Not operation (state = $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$).

There are two main approaches to concurrency on a quantum computer, either the system attempts to execute multiple instances of the same circuit and thus minimise the computation time or the system attempts to execute multiple distinct programs at the same time. Regardless of the approach, the concept hinges on the ability to execute more than one program at once. Quantum hardware is unique in its design as it only allows certain physical qubits to interact with each other [63], this is typically expressed in a connectivity graph like Figure 4.3. The connectivity graph in Figure 4.3 comes from an IBM quantum computer known as ibmq_5_yorktown - ibmqx2 v2.0.5 [63], and includes the error rates on the connections.

Quantum computers are typically not fully connected, therefore not all programs will fit at every location (as evidenced by the lack of connections in Figure 4.3). This also means that there can be extensive difference between distinct physical quantum computers as well the same quantum computer at a different period of time. This connectivity graph is only half of the problem, this example also requires the *programs* activity graph. The activity graph specifies the qubits required by the process, and the communication between them that the algorithm requires is denoted as an edge. The ac-

tivity diagram is generated by parsing the instruction set of the process and generating a node for every qubit specified, and a directed edge for every controlled operation specified. An activity graph of the program described in Figure 4.2 is shown in Figure 4.4.

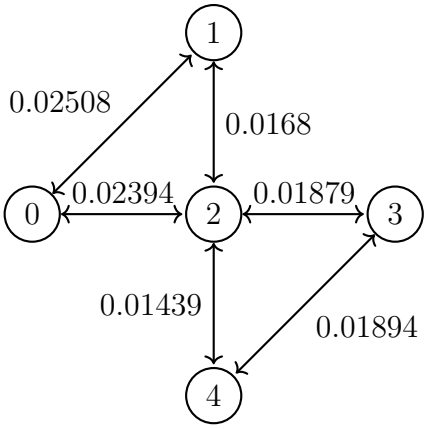


Figure 4.3: Connectivity Graph for ibmq_5_yorktown - ibmqx2 v2.0.5 [63].

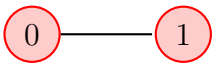


Figure 4.4: Activity Graph generated from Quantum Circuit in Figure 4.2

Ignoring the problem of finding the optimal qubit mapping for the purposes of this example, it can be assumed that the system could choose qubits 0 and 1 to host the quantum program. Note that the change between Figure 4.5 and Figure 4.6 still leaves a configuration of 3 connected qubits. These 3 connected qubits could then be used to host an additional 2 qubit quantum program, thus allowing for concurrent hosting of quantum programs (Figure 4.7).

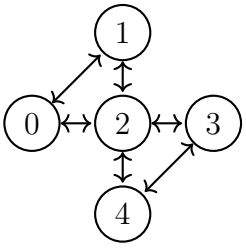


Figure 4.5: Initial System

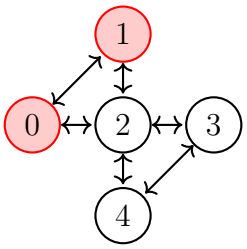


Figure 4.6: Single Map

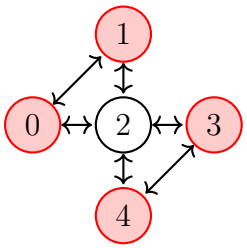


Figure 4.7: Double Map

The quantum circuit of the two split quantum programs is found in Figure 4.8, with the overall result of:

$$\frac{|00000> + |11000> + |00011> + |11011>}{2} \tag{4.1}$$

Upon measurement of the qubits, the resultant bitstring (for example: $|00011>$) should be segregated into the relevant results for each quantum program (for example: $|00>$ and $|11>$). Provided the qubit measurement is fine enough, in accordance with DiVincenzo criteria #5 [16], it is simple enough to measure the individual qubits for that program alone, instead of measuring the entire system and then subdividing.

4.5 Scheduling and Memory Management

Determining which programs can and should be parallelised through some form of a scheduling algorithm is key to achieving parallelisation. The proposed solution to this is to construct an intricate and complicated data structure which enables relative simplicity in the algorithms. The data structure is

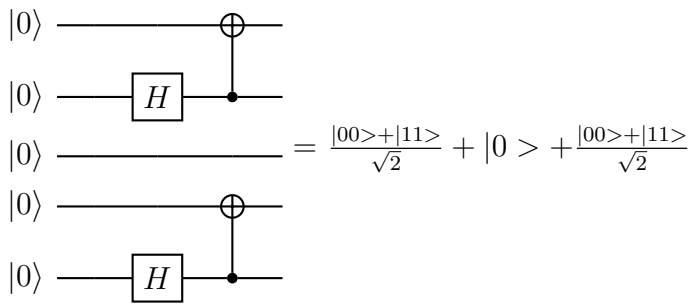


Figure 4.8: Split Quantum Circuit. The top two lines (qubits 0 and 1 in Figure 4.7) host one version of the circuit in Figure 4.4 and the bottom two lines (qubits 3 and 4) host another copy of the circuit

built using 5 separate graphs which are combined into a single multi-layered graph with links between the layers where relevant. The layers are as follows:

1. Qubit Connectivity Map (Figure 4.9a)
    - Nodes = Qubits,
    - Edges = Connections between qubits
2. Program Mappings (Figure 4.9b)
    - Nodes = Program Mapping,
    - Edges = N/A (This graph by default does not include edges because each mapping is individual, however edges will be added later in this chapter)
3. Program Dependency Map (Figure 4.9c)
    - Nodes = Quantum Programs,
    - Directed Edges = $A \rightarrow B$ where A is dependent on B

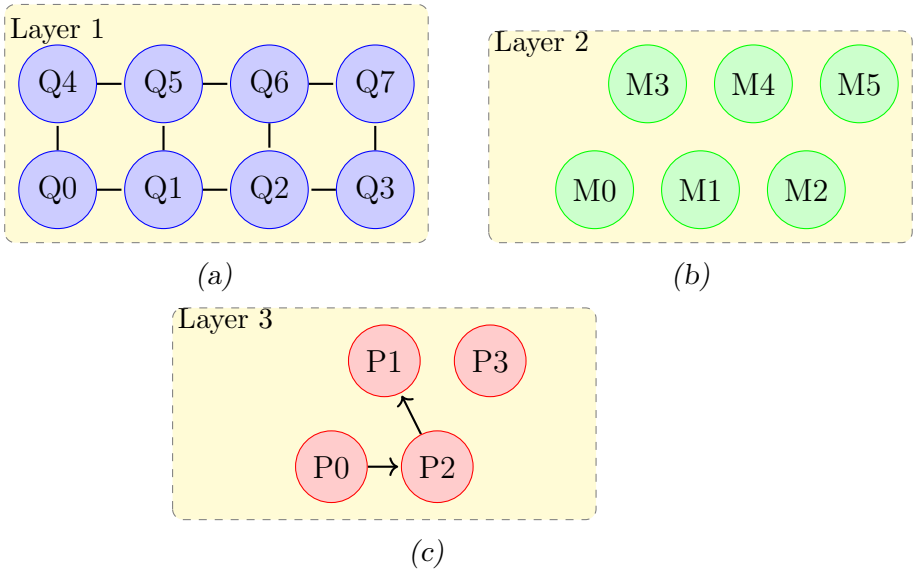
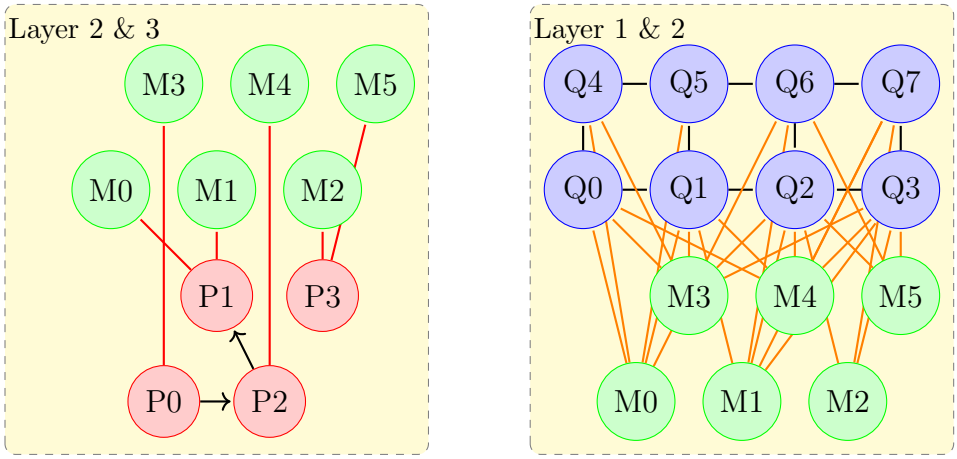


Figure 4.9: An example of single layer components a) An example of the qubit connectivity map, b) An example of the Mapping graph, c) An example of the process dependency graph

These layers form the base design of the data structure, with the links connecting the layers as:

1. Connection between the program mappings (Figure 4.9b) and the program dependency map (Figure 4.9c) where the mapping belongs to that program. This should result in a $1..n$ mapping. see Figure 4.10a

2. Connection between the qubit connectivity map (Figure 4.9a) and the program mappings (Figure 4.9b) where the qubits belong to the program mapping. This should result in a $n..n$ mapping. see Figure 4.10b



(a) (b)

Figure 4.10: Examples of Multilayer components a) An example of the programs and their mappings, b) An example of the mappings connected to their resources.

Overall this data structure combines all the required information for parallelism in a single concise structure. All of the information mentioned above was also required in some capacity for sequential execution. An example of this data structure can be found in Figure 4.11, with 5 graphs connected together. While care has been taken to simplify the example, the complexity of the data structure cannot be ignored.

Storing the nodes from Layer 3 in a priority queue allows for simple application of a scheduler algorithm, for simplicity priority based on arrival time also known as First In First Out (FIFO) is recommended. When the scheduler determines the next program to execute, the edges between layers 3 and 2 can be followed to attempt each of the applicable mappings (preferably starting with 0 cost (perfect mapping) and increasing from there). Mapping cost is determined from the amount of swaps and tweaks required to force the program to execute using that mapping. Swaps are defined as switching the relative place of 2 connected qubits, for example qubits #1 and #2 in Figure 4.3. Swaps are utilised in cases where qubit A needs to connect with other qubits but is unable according to the current mapping, by changing the relative position of A with B these connections can now be made. Tweaks

are defined as edits to the program code which still retain the qubit positions, for example reversing controlled operations allow the programs to be written as $B \rightarrow A$ instead of $A \rightarrow B$. Mappings which perfectly mirror the Activity graphs cost 0 to apply, with each swap increasing the cost by a fixed amount [104]. In the sequential mode, the first mapping will always fit and execute, before the system moves onto the next program.

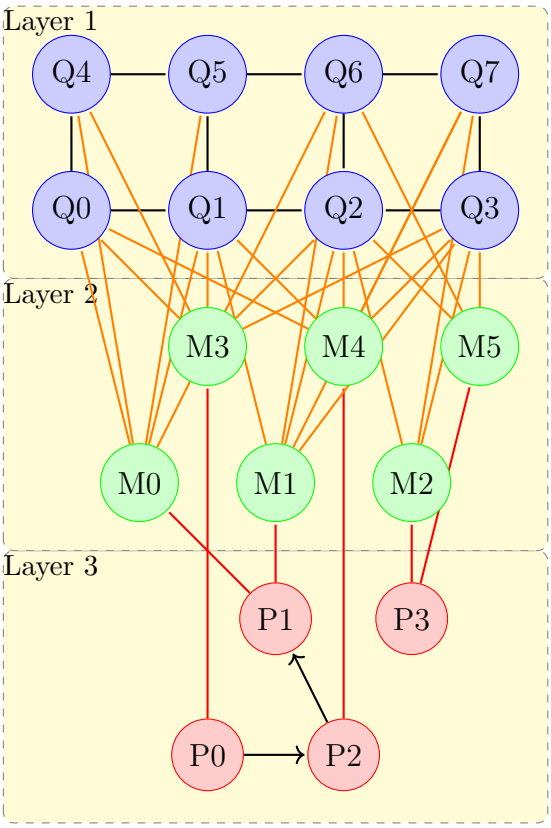
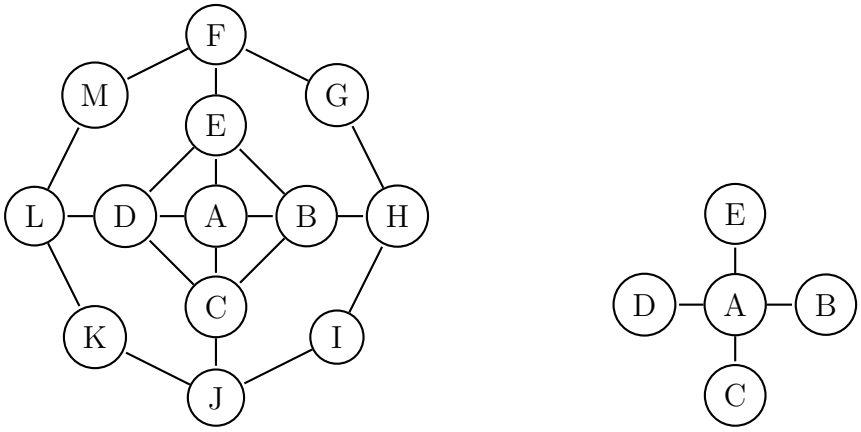


Figure 4.11: An example of the multigraph data structure
Program dependency graph represented as the red nodes (P*) and black directed edges, program mappings represented as the green nodes (M*) with no edges, qubit connectivity map with the blue nodes (Q*), programs and the connection to their mappings represented with red edges and the mappings and which qubits they lock represented with orange edges.

In the parallel mode, the next mapping may not fit (due to other programs currently consuming resources) and may require checking further, less efficient mappings or even halting until other programs free their resources (assigned qubits). Using the multi-layer data structure, unassigned layer 1 nodes (representing qubits) can be investigated as to whether mappings that will use them are able to be executed (all required nodes available). This approach allows the system to actively seek to execute more programs, however it ignores the current position of the scheduler. This can lead to issues of starvation (where programs never receive enough resources) for the next scheduled process because the nodes continue seeking processes which can run now instead of waiting for the required memory. A resolution to this is to have the scheduler 'reserve' nodes which will not be allocated to other programs and will wait for the scheduled task.

The parallel option mentioned above requires a large amount of execution and program management to allow for the free qubits to be searched for applicable programs to execute. A better approach is to employ the use of a serialization conflict graph (Figure 4.13 and 4.14) commonly found in database query processing [105]. This graph is used to determine when programs (or queries in databases) can be run in parallel or if they must be executed sequentially by reviewing what resources are required by the program (query) and whether other programs (queries) require the same resources [105].

In the case of a quantum computer, currently the only resource is the qubits. By treating the programs as single indivisible blocks, the program can be considered as 'locking' the relevant qubits until it has completed its execution. By considering which programs lock which qubits (found within the qubit connectivity and Mapping graph), edges can be introduced into layer 2 which denote which programs do not lock the same qubits and can thus be executed in parallel.



(a) Whole Graph *(b)* 1-ego network from A

Figure 4.12: Ego-Network Graph Example.

Specifically in the layered data structure, the conflict graph [105] will inform which mappings can be executed together which may include multiple possible mappings of the same program. If the edges dictate two mappings which cannot be executed together (Figure 4.13), then the number of edges expands greatly and determining the parallel mappings to execute becomes much more complicated. Alternatively using the edges to indicate two mappings which can be executed together (Figure 4.14) results in smaller edge sets, while determining which mappings to execute refines to a maximum clique problem [106] of the 1-egocentric network (a sub-network we define by selecting a node and only including all of its connections. For example, Figure 4.12b is the 1-egocentric network for node A from the network described in Figure 4.12a) [106] for the scheduled mapping.

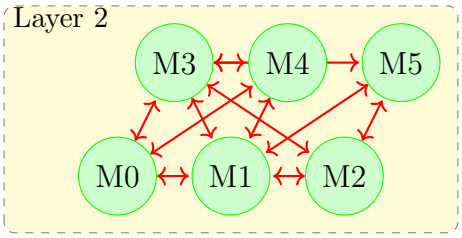


Figure 4.13: An example of the conflict graph where edges mean they **cannot** be executed in parallel

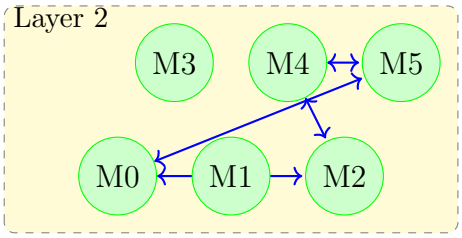


Figure 4.14: An example of the conflict graph where edges mean they **can** be executed in parallel

4.6 Overall System Design

The overall execution of the operating system is to continuously execute the next program according to the scheduler. The executing program then generates a 1-egocentric network and retrieves the maximum clique from said network to execute alongside the scheduled program.

The stages of adding a quantum program to be executed are as follows:

1. Quantum Program (P_Q) submitted to operating system
2. P_Q is added as a node in layer 3 of the multigraph data structure (MG)
3. P_Q is optimised by the circuit optimisation engine
4. P_Q is mapped to the quantum hardware by the mapping engine
    - As each mapping is discovered it is added as a node in layer 2 of MG and connected to the program node in layer 3 of MG
    - As each mapping is discovered the layer 2 node in MG is connected to the relevant layer 1 qubits.
5. All mappings are now added to the conflict graph

The stages of removing a finished quantum program are:

1. Return the results to the program for the user.
2. Return to the layer 1 program node from the completed layer 2 mapping node
3. Delete all layer 2 children (mappings)
    - Remove all layer 3 qubit links from the layer 2 mapping nodes
    - Remove the layer 2 mapping from the layer 2 conflict graph

If there is no programs to be executed it is recommended that the scheduler sleeps for a small period to allow for new programs to be added to the queue. The cost of the above implementation is largely reliant on the performance of the three largest system bottlenecks explored below.

4.6.1 System Bottlenecks

The largest bottlenecks with this system are all designed to be modular and can be replaced with a better algorithm/library when they are discovered. The system bottlenecks are:

1. Quantum program optimisation engine [97]
2. Quantum program mapping engine [104]
3. Parallel execution problem [106]

4.6.1.1 Quantum Program Optimisation Engine

Quantum program optimisation is the process of eliminating redundant operations and attempting to simplify the overall program [97]. For example if a quantum program performs a NOT operation followed by a second NOT operation (see Figure 4.15 and Equation (4.2)) then the NOT operations cancel each other resulting in the same output as though they were never computed. These redundant operations result in time wasted and increases the amount of error in the system.

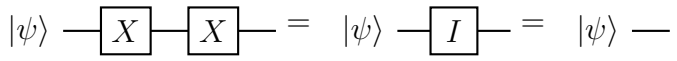


Figure 4.15: Redundant Quantum Circuit

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.2}$$

The optimisation process is not compulsory as sub-optimal programs can still be mapped and executed, however optimised programs should result in faster executions and/or a higher accuracy [97]. The balancing act with this process is to ensure that the benefit gained from optimising the program is at least as long as the time taken to optimise.

4.6.1.2 Quantum program mapping engine

Mapping a quantum program to quantum resources is an ongoing research question [99]–[104]. At the core of the problem is the NP-Hard subgraph isomorphism problem [104]. Locating a mapping that works can usually be accomplished with relative speed, finding the optimal mapping is a different story. A better mapping results in a more accurate result and a faster execution time [104]. Multiple attempts have been made to resolve this problem but the current leading research stems from Siraichi et al [104]. This process has to be completed for every quantum program that is entered into the system. Any delay taken in the mapping stage affects each quantum program, such that a 10 second delay to map a process results in a 100 second delay to map 10 processes.

Siraichi et al. [104] present a method which does not require any underlying knowledge of the qubit connectivity map and does not require any underlying structure within the qubit connectivity map. Siraichi [104] approaches the problem by mapping the program onto the hardware one qubit at a time, placing the qubit in response to already allocated qubits. This approach has the added benefit of providing all possible mappings as a re-

sult, which can then be used in the scheduling and memory management of the system. The algorithm [104] suffers from exponential growth during the search, where allocating qubits can result in growth which approximates $\prod_{k=0}^{q-1}(n-k)$ where n is the number of qubits in the computer and q is the number of qubits being mapped. In practice this limit is typically lower because of the relationship between qubits limiting the applicable matches.

4.6.1.3 Parallel execution problem

The conflict graph approach greatly simplifies the search for programs which can be executed in parallel and the maximal set of programs that can be executed in parallel. Following the conflict graph approach (assuming an edge represents 2 mappings that can be executed in parallel), the task is to determine a clique (a set of nodes where all the nodes are connected through edges) such that the clique is maximal (no more nodes are able to be added to the clique). It is also recommended to find a maximum clique so as to parallelise as many programs as possible.

Because there exists a scheduled node which we must include, a 1 or 1.5 egocentric network [106] may be employed to reduce the search space to only those nodes which are neighbours of the scheduled node.

The parallel execution presented in this chapter is simple and basic, akin to a structure built of blocks. A more advanced parallelism can be employed through more specific application of the blocks. This approach yields better parallelism, however improper implementation will result in exploitable attack vectors. If the system considers a measurement as the trigger to free a qubit then programs can be properly interwoven, however this requires partial mapping and assumes that the qubit is not reused by the program. By incorrectly employing the above advanced approach, the system is then specifically open to CWE-416: Use After Free [107] and CWE-200: Exposure of Sensitive Information to an Unauthorized Actor [108] attacks. Discussion of general security concerns can be found in Chapter 9.

4.7 System Scaling

Now that the system has been specified and explored, the final inquiry is how well the system scales. In pursuit of an answer to this inquiry, the scaling of a quantum computer has been seperated into vertical and horizontal scaling. Vertical scaling considers the system still running on a single quantum computer but of a larger size. Leaving horizontal scaling to consider keeping the quantum computer the same size, but expanding over multiple quantum computers to extend the computing power available. A further discussion of system scaling can be found in Chapter 8, which expands on the introductory analysis below.

4.7.1 Vertical Scaling

The designs and algorithms discussed within this paper will continue to produce accurate results for any sized quantum computer. This is due to simply needing to search a larger graph structure which requires more processing time. However due to inbuilt and inherent inefficiencies in the discussed algorithms, the cost of deploying them on larger quantum computers will continue to increase in accordance with the details of the quantum computer. A discussion on whether all qubits or connections need to be included can be found in Section 10.6.2.

4.7.2 Horizontal Scaling

An alternative method for increasing the size of the system is to employ multiple distinct quantum computers under the purview of a single instance of the system. Each quantum computer provides a connectivity graph of its qubits and can be instructed individually. One can consider all n quantum computers as a single graph of n connected components. Alternatively you can store it as n graphs each of 1 connected component. Searching a single disconnected graph is faster than searching each individual graph, however the cost of considering all n graphs at once is greatly increased from considering them each individually.

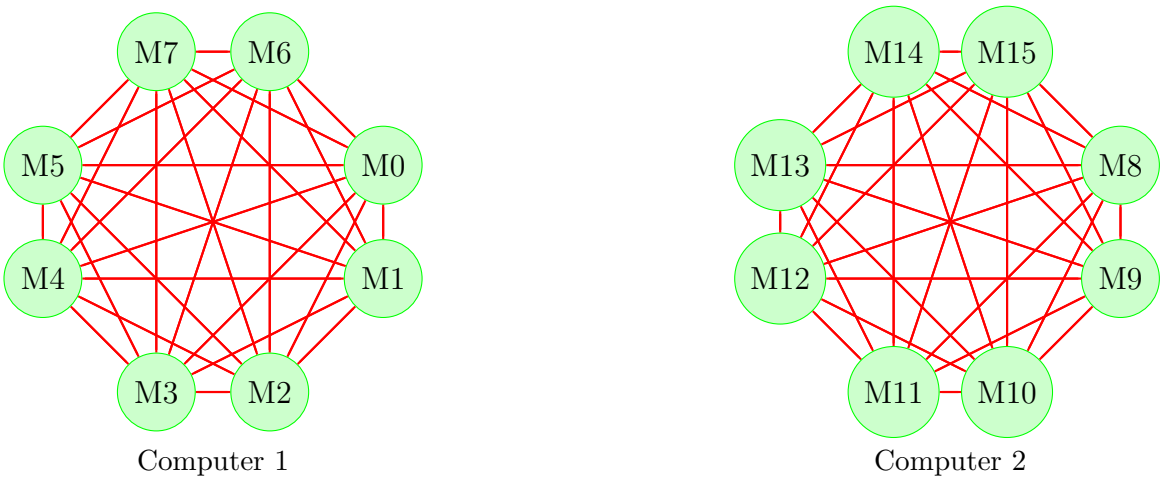


Figure 4.16: An example of multiple Quantum Computer graphs for the system to manage

Without access to a quantum network connection a quantum program can only be split over multiple distinct quantum computers provided that the activity graph is made of multiple connected components with each component on a separate quantum computer.

4.8 Chapter Summary

This chapter demonstrated that not only is quantum computation parallelisable (Research Question 1), and there exists a system which can co-ordinate and organise the concurrency (Research Question 2). The system proposed in this chapter highlights the ongoing problems that need to be addressed in order to fine-tune the system performance. These problems include Quantum program mapping and the Clique identification problem. The algorithms discussed above are intentionally generic, therefore applicable to as many quantum computers as possible.

As quantum computers continue to grow, connect and mature, the algorithms introduced here will require revisions for performance reasons. It is

expected that as quantum computers increase the number and connectivity of their qubits, the multigraph data structure will increase in density. Access to quantum hardware would enable a more specialised implementation to be discussed, though that implementation would then suffer from issues with portability and maintenance.

5. GLADEOS DESIGN

Following the system outline in Chapter 4, the next stage of the research is to model that system and review the behaviour. Following a brief discussion of the issues with computer models, this chapter is dedicated to the design of the computer model used for generating results in subsequent chapters. The detailed information is provided so that any potential errors or assumptions can be identified and accounted for and provides a solution for Research Question 3A.

5.1 Issues with Computer Models

Using a computer based model to simulate a phenomena is a common approach to research for 2 reasons:

1. To confirm that the suggested model accurately simulates the phenomena (e.g. the damage caused by a specific explosive) [109]

2. To test how the phenomena will react to specific inputs (e.g. how a building reacts in strong winds)[110]

The major problem with computer models is that the results are only as accurate as the model itself (as mentioned in Section 3.2.1). If the model equates $B = 2A$ when in reality $B = A + C$, then the results will be correct provided that $A = C$. However the moment that $A \neq C$, the model is incorrect and therefore all results must be double checked for validity or outright dismissed, regardless of the precision of the computer that model is implemented on. Therefore to ensure that the results are correct and valid, effort must be undertaken to ensure that the model is accurate.

A secondary problem with computer models is the war between precision and accuracy. The precision of a solution comes from comparing results to each other and measuring how close the measurements are to each other. For example, if a computer model returns the result $A \pm 10$ from an expected range of [0,20] then it is not very precise because the recorded results vary between measurements. A further consideration for precision is the units used, for example the result may be $A \pm 10$ however if the measurement is in nanometers then the difference is vastly smaller than if the unit was meters.

In contrast, accuracy is measured as how close the reported result is to the accepted true value. For example if a scale weighed a 1kg packet of peanuts as 10kg then that scale is not accurate. The best results are both accurate and precise, though this can be difficult to achieve with some computer models.

Normal people commonly mistake computer models as both accurate and precise by default, often taking the returned solution as the accepted truth. Errors in computer models can cause multiple issues, including imprecise or incorrect results. These errors include:

Rounding errors Errors encountered due to a lack of significant figures resulting in approximations, for example 3.4 and 3.6 round to 3 and 4 respectively thereby changing the difference from 0.2 to 1 [111].

Logic errors Fundamental errors in the algorithm or equations which result in incorrect results, for example $A + B$ instead of $A * B$. This type of error can result in correct results under specific inputs (for example $A = B = 2$) but overall leads to incorrect results [111].

Algorithm Stability Algorithm stability is a measure of how the inputs affect the outputs of an algorithm. To be considered stable, a small change in inputs should result in a small update in results. Instability of the algorithm can be caused by problems with the algorithm or problems with the underlying mathematical problem being solved by the algorithm [111].

Truncating errors Truncation errors also stem from a lack of significant figures resulting in removing the extra digits, for example 2.41 and 2.49 can both be truncated to 2.4 [111]. A common source of truncation is using incorrect data types, for example integers can only store whole numbers and will truncate or refuse other data types.

Irrational Errors Irrational numbers (numbers like π and $\sqrt{2}$ which cannot be represented as $\frac{A}{B}$) require an infinite number of digits to accurately represent them. Because no data type can support that infinite digits, all irrational numbers can only be represented by approximations. Because of this all calculations with these numbers will produce approximate results.

The effect of individual errors may be considered small e.g. $2.123456 \rightarrow 2.12346$. However errors found in early stages of the computation flow downstream to affect the later stages [111]. For example if the computation only uses integers (whole numbers) then $\frac{10}{3} = 3$ and $3 * 3 = 9$ where $\frac{10}{3} * 3 = 10$. This error propagation affects the accuracy of the computation, though the precision should remain relatively constant. Some approaches to reduce errors include using better suited data types, increasing the number of significant digits or algebraic manipulation of the formulas ($\frac{x}{3} * 3 \rightarrow x$) to minimise possible sources of error.

5.2 Quantum Simulator: GladeOS

5.2.1 Requirements

The design of the Quantum Simulator (GladeOS) was created, in part, due the limitations of the already existing tools. To test the feasibility of classical scheduling algorithms, a simulator must adequately portray the mechanics of a quantum computer. Each of the Quantum Simulators (simulators) identified are either:

- Not capable of handling multiple circuits simultaneously (a key contribution of the Thesis to be tested) [24]–[26], or
- Not capable of executing partial circuits [24]–[26], or
- Not capable of being extended [24], [26].

In the case of all known simulators, they are either unable to test or perform the necessary operations to test classical scheduling algorithms, as they are either:

- An interface to physical hardware (IBMQ, Riggetti Forest)
- A Single-User simulator (Qiskit, Q#, Rigetti Forest)

and were deemed unsuitable. As such, to answer the research questions outlined in Chapter 3, one must first create the simulator that can handle scheduling quantum operations with classical algorithms.

This scheduling is two-fold. Not only must the program share computing time, it must also test memory (Qubit) allocation in this shared, scheduled computing space. Given the expensive, specialised hardware (Quantum computers and High Performance Computers) cannot feasibly be purchased or obtained easily the simulator must thus run on commodity hardware. To do so, there are several key concepts that were designed, Qubit simulation design, Simulated Qubit Types and State on Demand.

5.2.2 Qubit Simulation Design

Due to the lack of commercial quantum hardware to develop with, simulating the qubits is a necessity. Simulating the qubits also enables the amount and configuration of the qubits to be changed and updated as required, thereby helping to develop and test a universal solution. After investigating the various equivalent means to represent qubits and their associated costs (Table 5.1), it was decided that Dirac vector notation [28], [43] was the optimal representation for our means. Using this notation every qubit can be represented

using two complex floats, taken as alpha (α) and beta (β) appropriately (Equation 2.3, reproduced below as Equation 5.1 for completeness). Using this approach allows all gates to be simplified to their matrix representations and thus saving on both compute power and time.

$$|\varphi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{bmatrix}\beta\\ \alpha\end{bmatrix} \tag{5.1}$$

Utilising dirac vector notation the computation of any given point upon a Bloch sphere (Figure 2.3) can be performed with minimal memory overhead. These vectors are stored in the form of a complex float. As these are complex numbers, the distinction of standard computing types 'float' and 'double' must be addressed. As GladeOS is designed to run on commodity hardware, memory usage must be kept to a minimum without sacrificing accuracy. As such, complex floats allow for an acceptable level of accuracy, as the single-precision nature is both adequate for dirac vector notation and allows for reduced memory footprint which is significant when calculating state arrays. As GladeOS stores only two complex floats, the memory footprint can be kept to a minimal level until the entanglement operations occur.

Table 5.1: Qubit representation decision matrix

Representation	Positive	Negative
ϕ and θ (Bloch sphere representation)	Equations are defined for single qubit operations, $\|\psi\rangle = cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}sin\frac{\theta}{2}\,|1\rangle$. Requires storage of 2 real floating point values to represent any position.	Operations for multiple qubits are not defined and would require conversion to the Dirac representation

Boolean Data	Because a measured quantum bit returns a single Boolean datum, therefore it should be possible to simulate the system in this manner. Gate operations only have 2 options, a) flip the boolean value or, b) leave the boolean value as is. Logic gate operations like Pauli-X (NOT) gate are very simple (logically).	This method requires a lot of analysis of the quantum program instructions to ensure that the results are accurate. The large problem with this method is that after every gate, the qubit must (essentially) be measured in order to quantify the change to the Boolean. Because of this sudo-measurement chains of gates are difficult to employ. The major issue with this strategy is that it is impossible to hold a superposition state as a qubit, therefore a system which is built using this approach cannot prepare an entanglement state.

X, Y and Z Co-ordinates	Using co-ordinates allows for simple (π or 2π) rotations around the sphere. This representation removes the requirement for using imaginary numbers. Requires storage of 3 real floating point values to represent any position.	The co-ordinate system extends far beyond the Bloch sphere therefore very easily leading to invalid co-ordinates which do not appear on the sphere. Complex rotations around the sphere are complicated and typically require conversion to alternative representations.
α and β (Dirac representation)	Extensive literature exists, including clearly defined Gates and behaviors for multiple qubit system.	Difficult to represent visually. Requires 2 complex floating point values to represent any position.

5.2.3 Simulated Qubit Types

This leads to a logical separation of stationary qubits [16] into an 'open' and an 'entangled' mode. 'Open' qubits are simply two complex floats that GladeOS can perform operations on. If not performing entanglement operations, a classical system with 8GB of Random Access Memory (RAM) could easily address and manipulate millions of the 'open' qubits. For example, each individual 'open' qubit is expected to consume approximately 48 Bytes of memory, therefore 20,800,000 qubits would consume 1 GB of memory. 'Entangled' qubits become a separate logical flow of operations.

Entangled qubits require a state array to perform any operations, given they are now treated as a single unit. This "state array" (array of complex floats e.g. $[0f, 0f, \frac{1}{\sqrt{2}}f, \frac{1}{\sqrt{2}}f]$) does not have to be active in RAM until required. As GladeOS reads in a known "controller" program, it can reconstruct the entangled state at any given point, including historical operations. This state array is both large and grows exponentially. In accordance with the operations of GladeOS, using complex floats, the memory consumption of the entanglement operation is:

$$Memory = 2^{(Qubits+6)} + (Qubits \times 2^7) \tag{5.2}$$

This equates to 33 entangled qubits requiring approximately 68 GB of RAM to hold the state array. This state-on-demand leads to a known and mitigatable issue with GladeOS - calculating this state array will take a large amount of computing power. The benefit is that, for a system with 128GB of RAM, it is possible, in theory, to entangle up to approximate 500 qubits across multiple distinct sets of entangled qubits (allowing for system overhead). The system is not capable of supporting a 34 qubit entangled set, but it can support a 33 qubit entangled set and a 32 qubit entangled set concurrently (as seen in Figure 5.1). The system can support multiple distinct sets provided that each of the entanglement operations does not contain more qubits than the descending series of applicable maximum (33, 32, 31...).

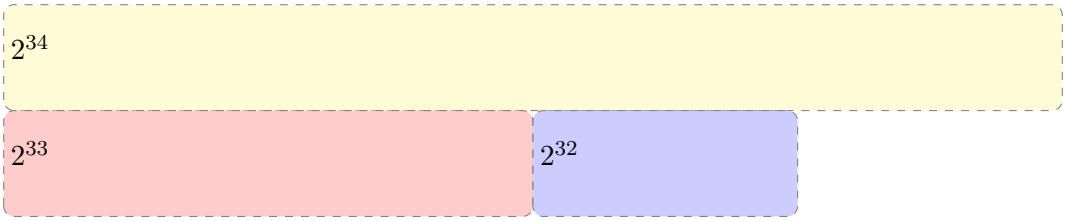


Figure 5.1: Example of the Relative Size Required by the Entangled Sets

5.2.4 Global vs Local State array

Two methods were explored for managing the state of the quantum system. The first option was to instantiate a global state array, assign qubits to programs and perform operations on the global state array. This approach had the benefit of only managing a single instance of the state array, however subdividing the state array made performing and checking operations complex in addition to creating a major bottleneck for the simulator. The alternative method of granting each program their own individual state array allows for operations to be performed in parallel as well as greatly increasing the number of qubits that can be simulated (as explored in Section 5.2.3 and Figure 5.1).

5.2.5 State on Demand

This per-operation limit is not a common occurrence in quantum computer simulators, due to concurrent program execution not being a standard requirement or capability. This limit lends itself nicely to multi-user scheduling testing, as each user can execute freely without co-ordinating with other users. This is handled in GladeOS, by allowing the Scheduler to "know" when it is executing "quantum operations" versus "preparing to perform" said quantum operations. This distinction is invisible to the program executing and is handled internally by GladeOS.

Importantly whilst this "Prep Time" is occurring, Quantum Phenomena, like decoherence are in stasis. This split of program flow mitigates the extended time period of creating and calculating state arrays on low-powered machines. Thus, maintaining the accuracy of the quantum simulation (e.g. decoherence) by accounting for non-quantum simulator operations being performed.

5.3 System Stack

This section covers the system stack of GladeOS. Figure 5.2 is a schematic overview of the relationships between the different components of the stack.

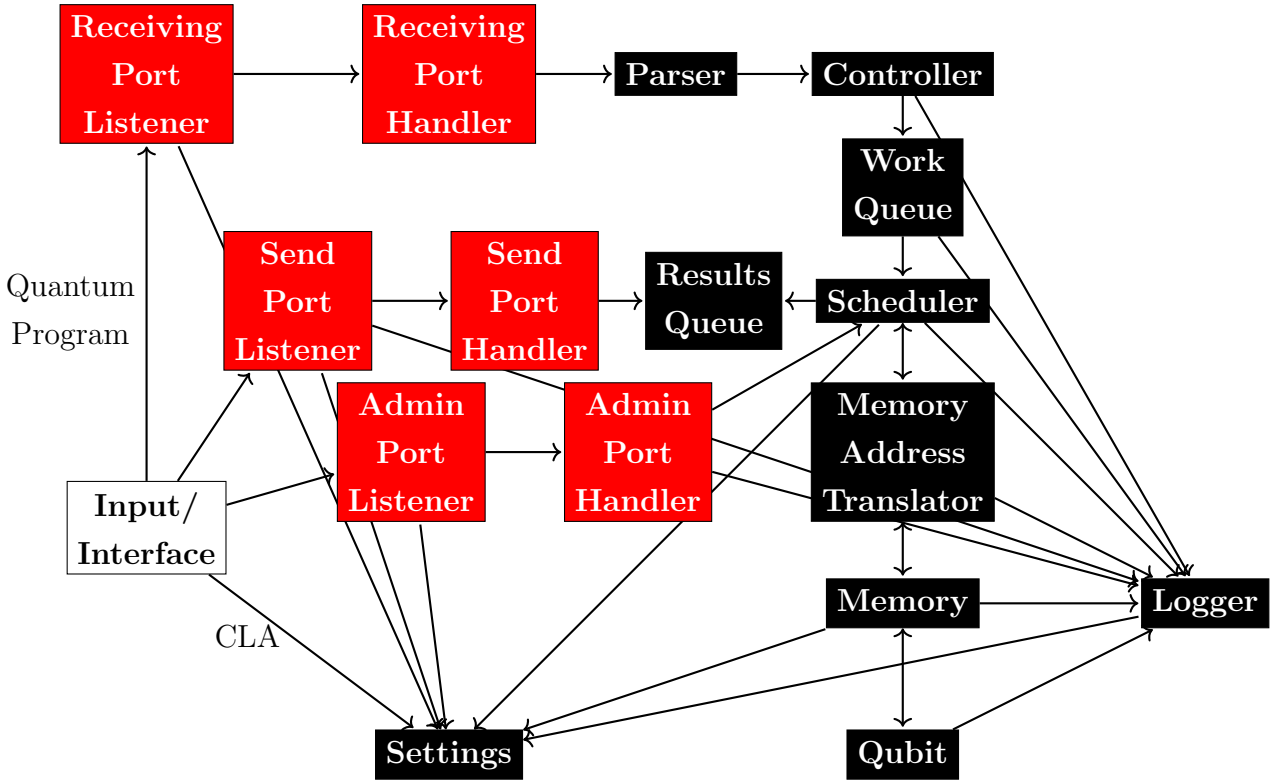


Figure 5.2: GladeOS System Stack (Advanced). The White node indicates where user input is entered into the system, Red nodes indicate networking functionality while Black nodes indicate internal components of the system.

5.3.1 Qubit

The Qubit class is designed to handle all single qubit facilities. It maintains the α and β of the qubit (using Dirac Vector Notation). These data points are acted upon by the logic gates which adjust the α and β accordingly. Also included in this class is an independent timer which counts down until the α and β are reset, thus simulating decoherence rendering the qubit erroneous.

5.3.2 Memory

The Memory class handles all overreaching organising and managing of the qubits as a block. This class treats the qubits as a set array from 0 through to n and handles the interaction between programs and qubits.

5.3.3 Memory Address Translator

Because of the issues associated with observing quantum data the 'memory address translator' was implemented for better management of the memory. This class works similar to a page table in a classical computer OS, taking the internal address used in the program (e.g. q[0]) and the program itself in order to map that address to the correct qubit. This methodology allows all programs to handle their internal addressing as normal ($0..n$) while ensuring that the shared memory space remains properly segmented. This memory address translator is only necessary because this system handles multiple quantum programs at the same time (unlike other simulators currently available and a key advantage of GladeOS). This class can also be extended to test a large number of memory allocation algorithms, including but not limited to the Siraichi algorithm[104].

5.3.4 Scheduler

GladeOS extends the submission queue found in alternative systems and allows the use of a variety of algorithms. The scheduler class is responsible for taking the incoming quantum programs and allocating the execution time to each program. This can be accomplished through a variety of algorithms, including but not limited to:

- First In First Out (FIFO) [72], [74]
- Last In First Out (LIFO) [72]
- Priority Queue (PQ) [72], [74]
- Qbogo (random ordering) (proprietary)

This class **must** also pay close attention to the decoherence time (discussed in Section 5.7), waiting too long to execute a specific program can lead to a need to restart the execution. It should be noted that monopolisation of resources is a very real possibility for larger programs, especially if they fail to complete in the time slots allocated. This can lead to starvation for all other processes in the scheduler. Currently the system continues to execute the process even if the decoherence time elapses, future work is needed to fully address this issue. Upon completion the process is placed in the Results Queue.

5.3.5 Controller

Critically, the controller class is responsible for handling a single quantum program. This program which consists of an instruction list must also track its progress and output. This controller is then cycled through according to the scheduler and executes when it gains access to the processor unit. Once the controller instance has been initialised, the system places the instance into the work queue for immediate execution.

5.3.6 Networking Stack

This system is designed to function as a stand alone system and to interact with users through the networking stack. This approach allows for a single system to support multiple users at once. The different components of the networking include:

Receiving The Receiving listener and handler is responsible for receiving quantum programs from the user. The handler then submits the program to the Parser before it is fully loaded as a controller in the work queue. The only feedback returned to the user is an error message if it fails to parse or a hash of the program if it succeeds.

Result The Result listener and handler is responsible for querying the results queue and returning results to the user as a pair of hash and result. After the results are returned they are removed from the system queue.

Admin The Admin listener and handler is responsible for providing an update of the system status which includes the number of results in the queue. Continuously polling the Admin listener can be used to wait until the number of results increases which indicates that a job has been completed.

Together the networking components provide the various interactions that the user would require from the system.

5.4 Command Line Arguments

As this system is designed as a test bench to gather raw data regarding the performance of different components it has a large amount of flexibility inbuilt. This flexibility is controlled through command line arguments which are provided at initialisation, the list of commands is:

Table 5.2: Command Line Arguments

Command	**Description**
--maxqubits or -m	Changes the max number of 'Open' Qubits that GladeOS will simulate. This disables all automatic detection. Must be greater than -e if specified.
--maxentangles or -e	Changes maximum number of 'Entangled' Qubits that GladeOS will simulate. This disables all automatic detection. Your system must have the Physical RAM available, or GladeOS will abort.
--controller or -c	Expects a fully qualified path to a valid directory, containing controller files.
--help or -h	Shows this table
--clocks or -c	Prints nanosecond scale clocks to screen
--maxthreads or -t	Specifies number of processing threads for the networking thread pool

Table 5.2: Command Line Arguments

Command	**Description**
--recvport or -rp	Specified port for GladeOS to listen on for new jobs. Default: 7200
--sendport or -sp	Specified port for GladeOS to listen on for all outbound traffic. Default: 5200
--adminport or -ap	Specifies the port that GladeOS will use for all admin status queries. Default: 6200
--logalltoconsole	Enable printing all debug/trace information to console, as well as disk.
--tracequbitops	Enabled logging in Qubit ops and State-Array tracing to disk
--logdir or -ld	Directory for GladeOS to store its log-files. Must be writable. Can be an aboslute or relative path. Default: logs/
--logthreads or -lt	Override the number of threads in the logging system thread-pool. Default: 4
--logqueuedepth or -lq	Override the number of messages that can be stored in the log-buffer at a given time. Default: 10,000
--scheduler or -sm	Set the Single-Threaded Controller Job Scheduler Mode. Valid Modes are: 'FIFO', 'FILO', 'QBOGO', 'PQ'
--multischeduler or -msm	Set the Multi-Threaded Controller Job Scheduler Mode. Valid Modes are: 'FIFO', 'FILO', 'QBOGO', 'PQ'
--sleeptime or -st	Set how many MILLI-seconds the scheduler will sleep for when there is no work in the Queue. Default: 250 MILLI-seconds. DO NOT set this too low, or you will pin a CPU to 100 with all the sleeps, and things WILL break.

Table 5.2: Command Line Arguments

Command	**Description**
--adminsecret	Set the secret string required to access the Administration data of the system.
--sendsecret	Set the secret string required to access the results returned by the system.

5.5 Supported Gates

Quantum simulators (including Qiskit, Q# and Rigetti) utilise quantum logic gates in order to construct quantum programs. The exact set of logic gates can differ between simulators, though each supports an accepted universal set of quantum logic gates. For this simulator the selected gates have been chosen from the Quantum Computation and Quantum Information textbook by Nielsen and Chuang [28] with additional rotation gates added to complete the set.

5.5.1 Single qubit

The quantum logic gates which require a single qubit to execute are:

Table 5.3: Supported single qubit gates

Name	**GladeOS Command**	**Operation**	**Matrix**
Pauli X (Not gate)	X(1)	π rotation around the X axis	$\begin{vmatrix} 0 & 1 \\ 1 & 0 \end{vmatrix}$

Table 5.3: Supported single qubit gates

Name	**GladeOS Command**	**Operation**	**Matrix**
Pauli Y	Y(1)	π rotation around the Y axis	$\begin{vmatrix} 0 & -i \\ i & 0 \end{vmatrix}$
Pauli Z	Z(1)	π rotation around the Z axis	$\begin{vmatrix} 1 & 0 \\ 0 & -1 \end{vmatrix}$
Hadamard	H(1)	$\frac{\pi}{2}$ rotation around the Y axis and π rotation around the Z axis	$\frac{1}{\sqrt{2}}\begin{vmatrix} 1 & 1 \\ 1 & -1 \end{vmatrix}$
Phase	S(1)	$\frac{\pi}{2}$ rotation around the X axis	$\begin{vmatrix} 1 & 0 \\ 0 & i \end{vmatrix}$
$\frac{\pi}{8}$	T(1)	$\frac{\pi}{4}$ rotation around the X axis	$\begin{vmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{vmatrix}$
Rotation X	RX([10.041],6) RX([θ],6)	Rotation around X axis (angle specified in degrees)	$\begin{vmatrix} \cos(\frac{\theta}{2}) & -i\sin(\frac{\theta}{2}) \\ -i\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{vmatrix}$

Table 5.3: Supported single qubit gates

Name	**GladeOS Command**	**Operation**	**Matrix**
Rotation Y	RY([236.461],6) RY([θ],6)	Rotation around Y axis (angle specified in degrees)	$\begin{vmatrix} \cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{vmatrix}$
Rotation Z	RZ([56.025],6) RZ([θ],6)	Rotation around Z axis (angle specified in degrees)	$\begin{vmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{vmatrix}$
Free Rotation	R([10.041,236.461, 56.025],6) R([θ,μ,ρ],6)	Rotation around X, Y and Z axes (angles specified in degrees)	$\begin{vmatrix} cos(\frac{\theta}{2}) & -isin(\frac{\theta}{2}) \\ -isin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{vmatrix}$ and $\begin{vmatrix} cos(\frac{\mu}{2}) & -sin(\frac{\mu}{2}) \\ sin(\frac{\mu}{2}) & cos(\frac{\mu}{2}) \end{vmatrix}$ and $\begin{vmatrix} e^{-i\frac{\rho}{2}} & 0 \\ 0 & e^{i\frac{\rho}{2}} \end{vmatrix}$

5.5.2 Multiple qubits

The other class of supported gates are gates which require multiple qubits to enact. These gates are as follows:

Table 5.4: Supported multiple qubit gates

Name	GladeOS Command	Input	Operation	Matrix
Controlled Gate	C([1,2],X(3))	Control qubit(s), target qubit	Perform the specified operation on the target qubit if and only if the control qubit(s) are $\|1\rangle$. Supported operations include Pauli X, Pauli Y, Pauli Z or Hadamard gates.	$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & x_1 & x_2 \\ 0 & 0 & x_3 & x_4 \end{vmatrix}$

The controlled gate can be extended for use with any of the supported single qubit gates.

5.5.3 Proof of all gates

All of the single qubit gates have been built using the approved matrix implementations as specified in the book "Quantum Computation and Quantum Information" [28]. To demonstrate this, each of the single qubit gates is included below with the approved matrix [28] and the resulting state assuming Equation 5.3 is the start state.

$$|\varphi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{5.3}$$

5.5.3.1 ***Qubit class***

The header of the qubit class (which stores the single qubit gates) from GladeOS can be seen in Algorithm 5.1. It should be noted that the Alpha and Beta values are stored as private variables within the qubit class and that each of the single qubit gates are implemented as functions directly within the qubit class (the specification for these gates can be found in Section 5.5). This approach enables the system to manage each qubit independently and

it only needs to combine multiple qubits when the gates require it.

```
class Qubit {
  private:
  /// Alpha value of the qubit. Alpha|0> + beta|1>
  std::complex<float> Alpha;

  /// Beta value of the qubit. Alpha|0> + beta|1>
  std::complex<float> Beta;

  /// Decoherence time limit for the qubit.
  std::chrono::duration<double, std::milli> decoherence
   = std::chrono::seconds(500);

  /// Asynchronous variable to facilitate the usage of
   asynchronous timers.
  std::future<bool> asyncFuture;

  /// Unique ID value to differentiate qubits.
  std::uint16_t qubitID = 0;
  public:
  Qubit();
  ~Qubit();
  void UpdateQubitID(size_t ID);
  [[nodiscard]] auto GetQubitID() const noexcept ->
   size_t;

  ///Gate Operations
  void RotateX() noexcept;
  void RotateY() noexcept;
  void RotateZ() noexcept;
  void RotatePhase() noexcept;
  void RotatePiEight() noexcept;
  void Hadamard() noexcept;
  void FreeRotate(Instruction<std::uint16_t, float>::
```

```
    MultiAxisAngle theta) noexcept;
31  void RotateX(float theta) noexcept;
32  void RotateY(float theta) noexcept;
33  void RotateZ(float theta) noexcept;
34  static float rad2deg(float radians) noexcept;
35  static float deg2rad(float degree) noexcept;
36  auto Timer() noexcept -> bool;
37
38  [[nodiscard]] auto ZeroProb() const noexcept -> std::
    complex<float>;
39  [[nodiscard]] auto OneProb() const noexcept -> std::
    complex<float>;
40
41  /// Method to set the qubit to state |0>
42  /// @return Nothing
43  constexpr void SetZero() noexcept {
44    Alpha = std::complex<float>(1.0F, 0.0F);
45    Beta = std::complex<float>(0.0F, 0.0F);
46  }
47
48  /// Method to set the qubit to state |1>
49  /// @return Nothing
50  constexpr void SetOne() noexcept {
51    Alpha = std::complex<float>(0.0F, 0.0F);
52    Beta = std::complex<float>(1.0F, 0.0F);
53  }
54
55  /// Collapses the current state of the qubit into a
    boolean output value
56  /// @return Boolean of resultant collapsed state.
57  auto Measure() noexcept -> bool;
58
59  /// Method to retrieve the (|1>) Beta values for the
    qubit.
60  /// @return Beta
```

```
61    [[nodiscard]] constexpr auto GetBeta() const noexcept
       -> std::complex<float> { return Beta; };
62
63    /// Method to retrieve the (|0>) Alpha values for the
       qubit.
64    /// @return Alpha
65    [[nodiscard]] constexpr auto GetAlpha() const noexcept
        -> std::complex<float> { return Alpha; };
66  };
```

Algorithm 5.1: Example implementation of the Qubit Class

5.5.3.2 ***Pauli X***

The Pauli X operation is described by Equation 5.4, the final state is shown in Equation 5.5 with the accompanying code in Algorithm 5.2

$$\begin{vmatrix} 0 & 1 \\ 1 & 0 \end{vmatrix} \tag{5.4}$$

$$|\varphi\rangle = \beta\,|0\rangle + \alpha\,|1\rangle \tag{5.5}$$

```
1     /// Qubit 180X rotation gate. Also known as the
       Pauli-X gate or the not gate.
2   void Qubit::RotateX() noexcept {
3     const std::complex<float> temp = Alpha;
4     Alpha = Beta;
5     Beta = temp;
6   }
```

Algorithm 5.2: A Pauli-X operation

5.5.3.3 *Pauli Y*

The Pauli Y operation is described by Equation 5.6, the final state is shown in Equation 5.7 with the accompanying code in Algorithm 5.3

$$\begin{vmatrix} 0 & -i \\ i & 0 \end{vmatrix} \tag{5.6}$$

$$|\varphi\rangle = -\beta i\,|0\rangle + \alpha i\,|1\rangle \tag{5.7}$$

```
1  /// Qubit 180Y rotation gate. Also known as the Pauli-
    Y gate.
2  void Qubit::RotateY() noexcept {
3    const std::complex<float> temp = Alpha;
4    Alpha = Beta * (ComplexStatics::comp_NegOne *
     ComplexStatics::comp_I);
5    Beta = temp * ComplexStatics::comp_I;
6  }
```

Algorithm 5.3: A Pauli-Y operation

5.5.3.4 *Pauli Z*

The Pauli Z operation is described by Equation 5.8, the final state is shown in Equation 5.9 with the accompanying code in Algorithm 5.4

$$\begin{vmatrix} 1 & 0 \\ 0 & -1 \end{vmatrix} \tag{5.8}$$

$$|\varphi\rangle = \alpha\,|0\rangle - \beta\,|1\rangle \tag{5.9}$$

```
1  /// Qubit 180Z rotation gate. Also known as the Pauli-
     Z gate.
2  void Qubit::RotateZ() noexcept {
3    Beta = Beta * (ComplexStatics::comp_NegOne);
4  }
```

Algorithm 5.4: A Pauli-Z operation

5.5.3.5 ***Hadamard***

The Hadamard operation is described by Equation 5.10, the final state is shown in Equation 5.11 with the accompanying code in Algorithm 5.5

$$\frac{1}{\sqrt{2}}\begin{vmatrix}1 & 1\\ 1 & -1\end{vmatrix} \tag{5.10}$$

$$|\varphi\rangle = \frac{\alpha+\beta}{\sqrt{2}}|0\rangle + \frac{\alpha-\beta}{\sqrt{2}}|1\rangle \tag{5.11}$$

```
1  /// Qubit Hadamard rotation gate. Also known as coin-
     flip gate or 90Y-Rotation and 180X-Rotation.
2  void Qubit::Hadamard() noexcept {
3    const std::complex<float> temp = Alpha;
4    Alpha = (Alpha + Beta) * ComplexStatics::comp_Onesqrt2
       ;
5    Beta = (temp - Beta) * ComplexStatics::comp_Onesqrt2;
6  }
```

Algorithm 5.5: A Hadamard operation

5.5.3.6 **Phase**

The Phase operation is described by Equation 5.12, the final state is shown in Equation 5.13 with the accompanying code in Algorithm 5.6

$$\begin{vmatrix} 1 & 0 \\ 0 & i \end{vmatrix} \tag{5.12}$$

$$|\varphi\rangle = \alpha\,|0\rangle + i\beta\,|1\rangle \tag{5.13}$$

```
1  /// Qubit Phase rotation gate. Also known as Pi/4 gate
   .
2  void Qubit::RotatePhase() noexcept {
3    Beta = Beta * ComplexStatics::comp_I;
4  }
```

Algorithm 5.6: A Phase operation

5.5.3.7 $\frac{\pi}{8}$

The $\frac{\pi}{8}$ operation is described by Equation 5.14, the final state is shown in Equation 5.15 with the accompanying code in Algorithm 5.7

$$\begin{vmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{vmatrix} \tag{5.14}$$

$$|\varphi\rangle = \alpha\,|0\rangle + \beta e^{\frac{i\pi}{4}}\,|1\rangle \tag{5.15}$$

```
1  /// Qubit Phase/2 rotation gate. Also known as Pi/8
   gate.
2  void Qubit::RotatePiEight() noexcept {
3    Beta = Beta * exp(ComplexStatics::comp_I * static_cast
   <float>(0.78539816339744830961));}
```

Algorithm 5.7: A Pi/8 operation

5.5.3.8 *Rotation X*

The Rotate X operation is described by Equation 5.16, the final state is shown in Equation 5.17 with the accompanying code in Algorithm 5.8

$$\begin{vmatrix} cos(\frac{\theta}{2}) & -isin(\frac{\theta}{2}) \\ -isin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{vmatrix} \tag{5.16}$$

$$|\varphi\rangle = (cos(\frac{\theta}{2})\alpha - isin(\frac{\theta}{2})\beta)\,|0\rangle + (-isin(\frac{\theta}{2})\alpha + cos(\frac{\theta}{2})\beta)\,|1\rangle \tag{5.17}$$

```
/// Free rotation on the X-Axis
/// @param theta Degree to rotate.
void Qubit::RotateX(float theta) noexcept {
  theta = deg2rad(theta);
  const std::complex<float> temp = Alpha;
  Alpha = (Alpha * (cosf(theta / 2.0F))) + (Beta * (
   ComplexStatics::comp_NegI * (sinf(theta / 2.0F))));
  Beta = (temp * ComplexStatics::comp_NegI * (sinf(theta
    / (2.0F)))) + (Beta * (cosf(theta / (2.0F))));}
```

Algorithm 5.8: A Rotate-X operation

5.5.3.9 *Rotation Y*

The Rotate Y operation is described by Equation 5.18, the final state is shown in Equation 5.19 with the accompanying code in Algorithm 5.9

$$\begin{vmatrix} cos(\frac{\theta}{2}) & -sin(\frac{\theta}{2}) \\ sin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{vmatrix} \tag{5.18}$$

$$|\varphi\rangle = (cos(\frac{\theta}{2})\alpha - sin(\frac{\theta}{2})\beta)\,|0\rangle + (sin(\frac{\theta}{2})\alpha + cos(\frac{\theta}{2})\beta)\,|1\rangle \tag{5.19}$$

```
  /// Free rotation on the Y-Axis
/// @param theta Degree to rotate.
void Qubit::RotateY(float theta) noexcept {
  theta = deg2rad(theta);
  const std::complex<float> temp = Alpha;
  Alpha = (Alpha * (cosf(theta / (2.0F)))) + (Beta * (
   ComplexStatics::comp_NegOne * sinf(theta / (2.0F))));
  Beta = (temp * (sinf(theta / (2.0F)))) + (Beta * (cosf
   (theta / (2.0F)))); }
```

Algorithm 5.9: A Rotate-Y operation

5.5.3.10 ***Rotation Z***

The Rotate Z operation is described by Equation 5.20, the final state is shown in Equation 5.21 with the accompanying code in Algorithm 5.10

$$\begin{vmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{vmatrix} \tag{5.20}$$

$$|\varphi\rangle = e^{-i\frac{\theta}{2}}\alpha\,|0\rangle + e^{i\frac{\theta}{2}}\beta\,|1\rangle \tag{5.21}$$

```
  /// Free rotate on the Z-axis
/// @param theta degree to rotate the Z-axis.
void Qubit::RotateZ(float theta) noexcept {
  theta = deg2rad(theta);
  Alpha = Alpha * (exp((ComplexStatics::comp_NegI * (
   theta) / (2.0F))));
  Beta = Beta * (exp((ComplexStatics::comp_I * (theta) /
    (2.0F))));
}
```

Algorithm 5.10: A Rotate-Z operation

5.5.3.11 **Free Rotation**

The Free Rotate operation is described by Equation 5.22, the final state is shown in Equation 5.23 with the accompanying code in Algorithm 5.11

$$\begin{vmatrix} cos(\frac{\theta}{2}) & -isin(\frac{\theta}{2}) \\ -isin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{vmatrix} \begin{vmatrix} cos(\frac{\mu}{2}) & -sin(\frac{\mu}{2}) \\ sin(\frac{\mu}{2}) & cos(\frac{\mu}{2}) \end{vmatrix} \begin{vmatrix} e^{-i\frac{\rho}{2}} & 0 \\ 0 & e^{i\frac{\rho}{2}} \end{vmatrix} \tag{5.22}$$

$$\begin{aligned} |\varphi\rangle = \\ (\alpha e^{-i\frac{\rho}{2}}(cos(\frac{\theta}{2})cos(\frac{\mu}{2}) + isin(\frac{\theta}{2})sin(\frac{\mu}{2})) + \beta e^{-i\frac{\rho}{2}}(-cos(\frac{\theta}{2})sin(\frac{\mu}{2}) - isin(\frac{\theta}{2})cos(\frac{\mu}{2}))) |0\rangle + \\ (\alpha e^{i\frac{\rho}{2}}(-cos(\frac{\theta}{2})sin(\frac{\mu}{2}) - isin(\frac{\theta}{2})cos(\frac{\mu}{2})) + \beta e^{i\frac{\rho}{2}}(cos(\frac{\theta}{2})cos(\frac{\mu}{2}) + isin(\frac{\theta}{2})sin(\frac{\mu}{2}))) |1\rangle \end{aligned} \tag{5.23}$$

```
1  ///Free rotation on the X,Y and Z axis.
2  /// @param theta Rotation degree around all axis
3  void Qubit::FreeRotate(Instruction<std::uint16_t, float
       >::MultiAxisAngle theta) noexcept {
4    RotateZ(theta[2]);
5    RotateY(theta[1]);
6    RotateX(theta[0]);
7  }
```

Algorithm 5.11: A Free Rotate operation

These single qubit gates have been extended into 2 qubit gates by implementing a control gate variation. The control gate implementations are designed to take any number of control bits, which are all required to be **1** before execution.

Following the use of a control gate the qubits in question are considered entangled, and can no longer be modelled correctly by separating them. A clear example of this is the four Bell States, which are generally used to prove entanglement. Because of this performing manipulations on entangled

states of n qubit length is a complicated endeavour. The following proof uses a generic representation of a 3 qubit system but can be expanded to any length.

5.5.3.12 Multiple Qubit logic proof

The three qubits used for this system are defined as:

$$|\varphi_1\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$
$$|\varphi_2\rangle = \gamma\,|0\rangle + \delta\,|1\rangle$$
$$|\varphi_3\rangle = \epsilon\,|0\rangle + \rho\,|1\rangle$$

Their entangled qubit system is therefore represented as:

$$State = \begin{vmatrix} \alpha\gamma\epsilon \\ \alpha\gamma\rho \\ \alpha\delta\epsilon \\ \alpha\delta\rho \\ \beta\gamma\epsilon \\ \beta\gamma\rho \\ \beta\delta\epsilon \\ \beta\delta\rho \end{vmatrix} \tag{5.24}$$

The main requirement to perform a single qubit operation is to determine the paired states. Paired states are states which differ only by a single value. For example $State[0] = \alpha\gamma\epsilon$ and $State[4] = \beta\gamma\epsilon$ differ by a single value, namely α and β. By identifying these paired states for the relevant qubit, the single qubit operations can all be performed on the entangled states. To identify the paired states one can use the following equation:

$$Pair = Index + 2^{n-(q+1)} \tag{5.25}$$

Where $Index$ is the current position in the state matrix (e.g. Equation 5.24), n is the number of distinct qubits that form the 2^n state matrix. Lastly

q is the relative position (read left to right) of the target qubit, starting at 0 and running through to $n-1$. Using Equation 5.25 one can simply calculate the state which is paired to the current state. For example in a state table of $n = 3$ qubits state 0 is paired with:

$$Pair = 0 + 2^{3-(0+1)} = 0 + 2^2 = 4 \tag{5.26}$$

In processing it is important to ensure that states are not processed twice. Using the above example, when state 4 has been processed it is imperative that states 4,5,6 and 7 be skipped. Failure to skip already processed states will result in incorrect values. A simple recourse is to jump the same length as the pair equation. For example, 3 is paired with 7, so jumping 4 states will result in state 8. This method should, if applied correctly, ensure that each state is only processed once. These operations will now be explored for each gate.

5.5.3.13 ***Standard Code***

The majority of the code for controlled operations is code for preparation. The template is as follows:

```
1  void Memory::ControlledMTPauliX(Controller &c, const std
       ::vector<uint16_t> qubitIndexes) {
2    //Temp Index for StateCalc
3    const auto target = c.instructions.at(c.
       instructionIndex).getMTarget();
4    std::vector<uint16_t> workingIndexes(qubitIndexes);
5
6    //Calc States
7    bool lastSpot = false;
8    if (std::find(workingIndexes.begin(), workingIndexes.
       end(), target) == workingIndexes.end()) {
9      workingIndexes.emplace_back(target);
10     lastSpot = true;
11   }
```

```
12  auto stateArrayIT = CalcAllTheMTThings(c,
     workingIndexes);
13
14  const auto len = stateArrayIT->first.size();
15  const auto iterIndex = std::distance(stateArrayIT->
     first.begin(),
16    std::find(stateArrayIT->first.begin(), stateArrayIT
     ->first.end(), target));
17
18  const auto value = static_cast<std::uint32_t>(std::pow
     (2.0F, static_cast<float>(len - (iterIndex + 1UL))));
19
20  auto iter1 = stateArrayIT->second.begin();// iterator
     at the start of the state array
21  auto iter2 = stateArrayIT->second.begin();// iterator
     at the start of the state array
22  std::advance(iter2, value);//Move iter2 to the pair
     value
23  auto iter3 = iter2;//iterator to indicate when the
     system is going to start repeating
24  std::complex<float> tmp = 0.0F;//Create a temp value
     to hold the value at iter1 for computation purposes.
25  bool performOP = false;
26  float Cindexvalue = 0;
27  std::uint16_t CiterIndex = 0;
28  for (auto i = 0UL; i < std::pow(2.0F, len - 1); i++) {
29    performOP = true;
30    for (auto cqubit : qubitIndexes) {
31      //need to check that all the qubits are 1.
32      CiterIndex = std::distance(stateArrayIT->first.
     begin(),
33        std::find(stateArrayIT->first.begin(),
     stateArrayIT->first.end(), cqubit));
34
35      if (std::fmod(std::floor(Cindexvalue / (std::pow
```

```
    (2.0F, static_cast<float>(len - (CiterIndex + 1UL))))
    ), 2.0F) == 0) {
36          performOP = false;
37        }
38      }
39      if (performOP) {
40        std::iter_swap(iter1, iter2);//swap the iterators
41      }
42      Cindexvalue++;
43      std::advance(iter1, 1);
44      std::advance(iter2, 1);
45      if (iter3 == iter1) {
46        std::advance(iter1, value);
47        std::advance(iter2, value);
48        iter3 = iter2;
49        Cindexvalue = Cindexvalue + value;
50      }
51    }
52  }
```

Algorithm 5.12: A controlled operation template

The different part of each operation is found on line 39-41. That specific if statement holds the code necessary to execute the operation, by changing that segment of code the operation changes entirely. For simplicity (and conciseness) the following snippets focus on this **if** statement.

5.5.3.14 ***Pauli X***

Swap the paired states.

```
1     if (performOP) {
2       std::iter_swap(iter1, iter2);//swap the iterators
3     }
```

Algorithm 5.13: A controlled Pauli-X template

5.5.3.15 **Pauli Y**

Multiply the first state by i. Multiply the second state $-i$. Swap the paired states.

```
if (performOP) {
  *iter1 *= ComplexStatics::comp_NegI;
  *iter2 *= ComplexStatics::comp_I;
  std::iter_swap(iter1, iter2);
}
```

Algorithm 5.14: A controlled Pauli-Y template

5.5.3.16 **Pauli Z**

Leave the first state alone. Multiply the second state by -1

```
if (performOP) {
  *iter2 *= ComplexStatics::comp_NegOne;
}
```

Algorithm 5.15: A controlled Pauli-Z template

5.5.3.17 **Hadamard**

This gate is the most complicated out of the available selection. Recall that

$$|\varphi_1\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{5.27}$$

will be manipulated into

$$|\varphi_1\rangle = \frac{\alpha+\beta}{\sqrt{2}}\,|0\rangle + \frac{\alpha-\beta}{\sqrt{2}}\,|1\rangle \tag{5.28}$$

Then by using the paired states $State[0] = \alpha\gamma\epsilon$ and $State[5] = \beta\gamma\epsilon$, one can create $\frac{(\alpha\gamma\epsilon+\beta\gamma\epsilon)}{\sqrt{2}}$ and by moving the like terms to the front of the equation it results in $\gamma\epsilon(\frac{(\alpha+\beta)}{\sqrt{2}})$. Which due to the commutive property of multiplication is re-arranged to $\frac{\alpha+\beta}{\sqrt{2}}\gamma\epsilon$ which is the correct result for the first

half of the paired state. The second half of Equation 5.28 follows the same steps as the first half, however it alters the sign to result in $\frac{\alpha-\beta}{\sqrt{2}}\gamma\epsilon$.

```
1    if (performOP) {
2      tmp = *iter1;
3      *iter1 = (tmp + *iter2) / static_cast<float>(sqrt
   (2));
4      *iter2 = (tmp - *iter2) / static_cast<float>(sqrt
   (2));
5    }
```

Algorithm 5.16: A controlled Hadamard template

5.5.3.18 **Phase**

Leave the first state alone. Multiply the second state by i

```
1    if (performOP) {
2      *iter2 *= ComplexStatics::comp_I;
3    }
```

Algorithm 5.17: A controlled Phase template

5.5.3.19 **Pi/8**

Leave the first state alone. Multiply the second state by $e^{\frac{i\pi}{4}}$

```
1    if (performOP) {
2      *iter1 *= std::exp(ComplexStatics::comp_I *
   static_cast<float>(0.78539816339744830961));
3    }
```

Algorithm 5.18: A controlled Pi/8 template

5.5.3.20 **Rotation X**

Replace the first state with $(cos(\frac{\theta}{2})\alpha - isin(\frac{\theta}{2})\beta)$ and the second state with $-isin(\frac{\theta}{2})\alpha + cos(\frac{\theta}{2})\beta$.

```
1        if (performOP) {
2            if (iter1 == stateArrayIT->second.end()) {
   throw QException("BAD ITERATOR ACCESS!"); }
3            *iter1 = (*iter1 * (cosf(theta / 2.0F))) +
   (*iter2 * (ComplexStatics::comp_NegI * (sinf(theta /
   2.0F))));
4            *iter2 = (*iter1 * ComplexStatics::comp_NegI
    * (sinf(theta / (2.0F)))) + (*iter2 * (cosf(theta /
   (2.0F))));
5        }
```

Algorithm 5.19: A controlled Rotate-X operation

5.5.3.21 **Rotation Y**

Replace the first state with $(cos(\frac{\theta}{2})\alpha - sin(\frac{\theta}{2})\beta)$ and the second state with $sin(\frac{\theta}{2})\alpha + cos(\frac{\theta}{2})\beta$.

```
1        if (performOP) {
2            temp = *iter1;
3            *iter1 =
4                    (*iter1 * (cosf(theta / (2.0F)))) +
   (*iter2 * (ComplexStatics::comp_NegOne * sinf(theta /
    (2.0F))));
5            *iter2 = (temp * (sinf(theta / (2.0F)))) +
   (*iter2 * (cosf(theta / (2.0F))));
6        }
```

Algorithm 5.20: A controlled Rotate-Y operation

5.5.3.22 ***Rotation Z***

Multiply both state 1 by $e^{-i\frac{\theta}{2}}$ and state 2 by $e^{i\frac{\theta}{2}}$. (Note the negative sign)

```
1        if (performOP) {
2            temp = *iter1;
3            *iter1 = *iter1 * (exp((ComplexStatics::
   comp_NegI * (theta) / (2.0F))));
4            *iter2 = *iter2 * (exp((ComplexStatics::
   comp_I * (theta) / (2.0F))));
5        }
```

Algorithm 5.21: A controlled Rotate-Z operation

5.5.3.23 ***Free Rotation***

```
1       if (performOP) {
2           //performing each rotation, the order doesn't matter it should end up at the same spot.
3           //Rotate X
4           temp = *iter1;
5           *iter1 = (*iter1 * (cosf(theta / 2.0F))) + (*iter2 * (ComplexStatics::comp_NegI * (sinf(theta / 2.0F))));
6           *iter2 = (temp * ComplexStatics::comp_NegI * (sinf(theta / (2.0F)))) + (*iter2 * (cosf(theta / (2.0F))));
7
8           //Rotate Y
9           temp = *iter1;
10          *iter1 = (*iter1 * (cosf(lambda / (2.0F)))) +
11                  (*iter2 * (ComplexStatics::comp_NegOne * sinf(lambda / (2.0F))));
12          *iter2 = (temp * (sinf(lambda / (2.0F)))) + (*iter2 * (cosf(lambda / (2.0F))));
13
14          //Rotate Z
15          temp = *iter1;
16          *iter1 = *iter1 * (exp((ComplexStatics::comp_NegI * (phi) / (2.0F))));
17          *iter2 = *iter2 * (exp((ComplexStatics::comp_I * (phi) / (2.0F))));
18      }
```

Algorithm 5.22: A controlled Free Rotate operation

Please note that each of the values prescribed above have been extracted directly from their relative matrix. The other operation which can be performed on an entangled set is a controlled operation. This is a much more complicated operation opposed to the single gate operations explored above. The control gate requires the ability to manipulate the state only when the control qubit is in the $|1\rangle$ state. While this is simple for a single qubit, it gets more expensive with every qubit to parse through every state to check whether it is in the correct state. To combat this expense, the Equation 5.29 was devised by building upon the equations created in Section 5.5.3.12 (proof in Section 5.6).

$$\left\lfloor \frac{Index}{2^{n-(q+1)}} \right\rfloor \%2 \tag{5.29}$$

In Equation 5.29 $Index$ is the current position in the state table (decimal), n is the number of distinct qubits that form the 2^n state table. q is the relative position of the control qubit, starting at 0 before running through to $n-1$ and $\%2$ applies the modulus division operation between the result and the constant 2. To demonstrate the accuracy of this equation the following example has been provided.

$$n = 3, q = 1$$

A three qubit state table appears as:

$$State = \begin{vmatrix} \alpha\gamma\epsilon \\ \alpha\gamma\rho \\ \alpha\delta\epsilon \\ \alpha\delta\rho \\ \beta\gamma\epsilon \\ \beta\gamma\rho \\ \beta\delta\epsilon \\ \beta\delta\rho \end{vmatrix} \tag{5.30}$$

Table 5.5: Example results for 3 qubit state, checking value of bit 1 (LtR)

Index	State	Equation	Output
0	$\alpha\gamma\epsilon$	$\lfloor\frac{0}{2}\rfloor\%2$	0
1	$\alpha\gamma\rho$	$\lfloor\frac{1}{2}\rfloor\%2$	0
2	$\alpha\delta\epsilon$	$\lfloor\frac{2}{2}\rfloor\%2$	1
3	$\alpha\delta\rho$	$\lfloor\frac{3}{2}\rfloor\%2$	1
4	$\beta\gamma\epsilon$	$\lfloor\frac{4}{2}\rfloor\%2$	0
5	$\beta\gamma\rho$	$\lfloor\frac{5}{2}\rfloor\%2$	0
6	$\beta\delta\epsilon$	$\lfloor\frac{6}{2}\rfloor\%2$	1
7	$\beta\delta\rho$	$\lfloor\frac{7}{2}\rfloor\%2$	1

iterating over the state table with $\lfloor\frac{Index}{2^{3-(1+1)}}\rfloor\%2 = \lfloor\frac{Index}{2}\rfloor\%2$ yields the following output:

$$State = \begin{vmatrix} \alpha\gamma\epsilon \\ \alpha\gamma\rho \\ \boldsymbol{\alpha\delta\epsilon} \\ \boldsymbol{\alpha\delta\rho} \\ \beta\gamma\epsilon \\ \beta\gamma\rho \\ \boldsymbol{\beta\delta\epsilon} \\ \boldsymbol{\beta\delta\rho} \end{vmatrix} \tag{5.31}$$

Where indexes 2,3,6,7 are $|1\rangle$, demonstrated in Table 5.5.

Remember that this is only useful in determining the states that the chosen control bit is in $|1\rangle$ state. Recursive tests can be used for multiple control bits. To actually apply the controlled gate it is necessary to find the paired states for the target qubit. This search is conducted using the methodology presented in Equation 5.31.

5.6 Logic

Any binary state is represented as a string of length n composed of "0" or "1" 's as the alphabet. A pair of binary states is defined as 2 binary states which differ by exactly a single character. For example states 000 and 010 are paired states which differ by the central character. To simplify this, binary strings can be represented as A0B and A1B where A and B are binary strings of their own. For example, if $A = 101$ and $B = 010$ then $A0B = 1010010$ and $A1B = 1011010$ are paired states. The difficulty comes from determining the pair state from the original state, that is, $\text{pair}(B_1) = B_2$.

An iteration of a binary string is defined as all possible values of a binary state of length m given by 2^m values. For example an iteration of $m = 2$ is $[00, 01, 10, 11]$. The difference between A0B and A1B is a single iteration of B. To determine the length of B one could parse the entire binary representation and count the distance, alternatively it can be calculated by subtracting the length of $A + 1$ (+1 to account for the differing character) from the length of the full binary string. It may appear that we have simply moved the problem, however the length of A is given by the index of the differing character (supplied by the user). For example the state 01101 is of length 5, and with the index of the differing character as 3 (011**0**1), A is 011 and B is 1. The jump value to the paired state is calculated by Equation 5.32 where n is the length of the full state, and I is the index of the differing character. Given a decimal value 13 = 01101 and an Index of 3 the pair state is $13 + jumpValue = 13 + length(B) = 13 + 2^{n-(I+1)} = 13 + 2^{5-(3+1)} = 13 + 2^1 = 15$.

$$length(B) = 2^{n-(I+1)} \tag{5.32}$$

This approach can be expanded to consider differing substrings instead of differing characters. This expansion only requires altering the jump value of $n - (I + 1)$ to $n - (I + C)$ where C is the length of the substring. Given any decimal index L, $L \pm jumpValue$ returns the index where only the specified substring is different (moved 1 value along the iteration of the substring).

An application of this jump value equation is to determine what the specific value of the specified substring is for a particular decimal index. For example, given decimal index 13 = 01101, substring index 2 of length 2 we are retrieving 01**10**1. This is demonstrated in equation 5.33

$$\left\lfloor \frac{Index}{2^{n-(I+C)}} \right\rfloor \%2^C \tag{5.33}$$

The Equation 5.33 is composed of 3 sections:

1. Count the number of iterations.
2. Convert that number to an integer.
3. Determine the substring value from the number.

Step 1 is achieved through the $\frac{Index}{2^{n-(I+C)}}$ the division returns how many iterations of B have been completed. Step 2 is achieved by flooring the returned division. Finally step 3 is achieved by performing a modulus division which returns the decimal value. To complete the above example

$$\left\lfloor \frac{Index}{2^{n-(I+C)}} \right\rfloor \%2^C = \left\lfloor \frac{13}{2^{5-(2+2)}} \right\rfloor \%2^2 = \left\lfloor \frac{13}{2^{5-4}} \right\rfloor \%4 = \left\lfloor \frac{13}{2} \right\rfloor \%4 = \lfloor 6.5 \rfloor \%4 = 2 \tag{5.34}$$

The end result of the equation is the decimal value of 2, which in binary is 10. Using the above equation it is simple to mathematically extract the decimal representation of the specified binary substring from the decimal representation of a binary string.

5.7 Decoherence

Another issue which other simulators address in different ways is the issue of decoherence. This is a phenomenon which affects all long term quantum data. For this reason it has been built into the system. Because different quantum hardware is known to have significantly different decoherence times the system decoherence time of GladeOS is a variable which can be specified at run time (it can also be randomly set).

Decoherence is the effect of interference on the quantum state [43]. An example to explain this concept further is the release of a helium balloon. If there is absolutely no interference then the balloon will rise straight up until it cannot be seen anymore. However this is rarely the case, typically a multitude of wind currents are swirling around which manipulate the balloon and drag it off it's original course. One could model these winds and then account for them if they had enough accuracy and compute power, but as this software is required to utilise commodity hardware this is not a valid approach. Instead the implemented decoherence effect is to resolve the qubit to the zero state at the end of the decoherence timer (simulating a hard projective measurement and reset of the value). Using this method allows for the timer to be paused during preparation time (i.e. time associated with setting up the simulator, but not the simulation itself), allowing for an accurate representation while still maintaining minimal memory costs.

Given that the feature is not entirely accurate in the representation of decoherence, it is expected that quantum program developers (users) should not rely on the outcome of the decoherence timer for any part of their algorithm or outcomes. Rather, they should instead develop their algorithms to combat the decoherence.

5.8 Virtual Memory Addressing

If given a specific hardware mapping and a program to map, the simple approach is to hardcode the qubit id's (physical memory address) into the program. This approach allows the program to execute on this specific hardware, and simplifies the processing required by the system to execute the program. This approach can then be extended to encode a second process to execute on an alternate set of qubits. Using this method requires a large amount of planning time to fit the programs and is not scalable as quantum computers continue to grow and develop. The problem with this approach is that the program and their mapping is intertwined and therefore not translatable to alternate quantum computers.

To ensure that the quantum programs can be executed on any hardware map, an algorithm must be employed to map the program to a relevant section in memory. Using this approach leaves the developer with a choice to make, either:

- Translate the program in place and update the qubit ID's with the mapped qubits (Compilation), or
- Construct a 'lookup table' mapping the program qubit ID to the physical qubit ID, to be used when attempting to perform an operation (Interpretation).

Both of these approaches will succeed and when there is only a single active program it is largely an academic exercise to choose between them. Translating in place will result in slightly faster execution times as the few clock cycles spent perusing the lookup table (in the second approach) are shaved off. However using the lookup table allows the original program to be maintained in case of accidental corruption through the mapping process. The lookup table approach is reminiscent of the virtual memory addressing technique found in classical computing.

When supporting multiple active programs at a single time the above approaches need to be reconsidered. The lookup table approach still works, however, as it is a single source of truth, it must be protected from concurrent table accesses and/or data corruption. This can be easily accomplished by locking the table and only allowing a single program to use the table at any point. This approach works, however it struggles due to constant lock thrashing as every operation in each active program must wait on this lock.

The much better approach when supporting multiple active programs is the 'in place translation'. This approach is performed at the very end of the memory allocation procedure, meaning that the only time active programs may need to wait is when they are being allocated. Due to concerns over memory allocation corrupting the program, both the original instruction string and the parsed components of the instruction are stored. The translation process alters the parsed components leaving the original instruction string untouched.

5.9 Measurement

Measurement of a quantum system is a complex endeavour which is essential for receiving results out of the system. Measurement of a single qubit (with no entanglement) is a relatively simple task, perform a random number generation of 0 or 1 based on the α and β respectively. Random number generation is a known problem for standard computers, therefore to accomplish this generation a Mersenne twister engine [112], [113] has been employed (Algorithm 5.23). Measuring an entangled state however is a much more complex operation. Standard practice for measuring a single bit from an entangled set is to retrieve the probability of that state being a 0 or 1 and perform the standard measurement, before renormalising the state array according to the result [28], [43]. This approach can then be extended to measure each qubit independently, thereby measuring the entire system. This approach works well for single qubit measurement or measurement of small entangled states, however for relatively large entangled states (> 10) this approach is com-

putationally intensive and cannot be assisted by including more execution threads.

```
1   /// Collapses the current state of the qubit into a
     boolean output value
2 /// @return Boolean of resultant collapsed state.
3 auto Qubit::Measure() noexcept -> bool {
4   if (settings->GetQubitTracing()) { GLADE_INFO("Measure
     , Before Op: Alpha: {0}, Beta: {1}", this->Alpha,
     this->Beta); }
5   //These data types all exist within <random>
6   //Mersenne Twister Engine; Must call the functor on an
      existing std::random_device in order to function,
7   //will break if parenthesis balance is altered
8   std::mt19937_64 e((std::random_device())());
9   //Compute bernoulli distribution with a probability of
      true being the absolute value of beta
10  //where state = alpha |0> + beta |1>
11  std::bernoulli_distribution d(pow(std::abs(Beta), 2));
12  return d(e);//Generate a single boolean from the
     distribution
13 }
```

Algorithm 5.23: Measurement operation for a single qubit

The alternative approach for full state measurement that has not been utilised in this simulator is to generate a random number (r) between a lower bound (e.g. 0) and an upper bound (e.g. 100), then summing the relative probabilities (s) of each state until $s > r$ whereby that state is chosen. For example, in Table 5.6 the state table shows that all states have equal probability (25%). After generating r, the value is compared to the probabilities in the state table. If $0 < r <= 25$ then the measured value would be $m = |00\rangle$, likewise if $25 < r <= 50 \therefore m = |01\rangle$, if $50 < r <= 75 \therefore m = |10\rangle$ and if $75 < r <= 100 \therefore m = |11\rangle$. As $r = 80$ (Equation 5.35), the returned result is $m = |11\rangle$. To manage the precision of this alternative

method the boundary values could be adjusted and the random number could move from an integer value to a decimal notation. This process reduces the amount of computation being performed while still remaining true to the core amplitudes of the state array. As the number of qubits in the state array grows, the benefit of this approach continues to grow.

Table 5.6: Example State Table

State	Probability
$\|00\rangle$	25%
$\|10\rangle$	25%
$\|01\rangle$	25%
$\|11\rangle$	25%

$$r = 80 \therefore m = |11\rangle \tag{5.35}$$

5.10 Using GladeOS

Now that the system has been outlined, the last missing piece is how to interact with this simulator. The simulator was designed to be a standalone server program (similar to Rigetti [24]), where multiple users can interact with a single instance of the program. This decision was made to allow for better testing of multiple quantum programs and to prepare the system for future research into multiple users. To allow for this, the preferred method for communicating with the program is through specified network ports. GladeOS supports 3 specific ports:

- Receiving Port (Default: 7200). This port is used by users to transmit quantum programs to the simulator to be executed.
- Admin Port (Default: 5200). This port is used by users to query the current status of the simulator.

- Send Port (Default: 6200). This port is used by users to receive output from the simulator.

These ports each have a very specific task, and are adjustable using the command line arguments found in Table 5.4. Using network ports to facilitate communication allows for users to create custom front end applications to best suit their audience. In the interest of security, the Admin Port and the Send Port require the user to authenticate before data will be transmitted. This authentication takes the form of a 'secret' string which must be transmitted to the system. These secrets can be left as default, or set to a custom secret through command line arguments.

5.10.1 Program structure

Quantum programs are written according to the language specification outlined in Table 5.5.1 and 5.4. Programs are written with a single gate on each line, with an 'END' on the final line (as seen in Listing 5.24). Anything after the 'END' is treated as superfluous and ignored.

```
1 X(0)
2 Y(1)
3 H(2)
4 X(2)
5 Z(2)
6 X(1)
7 END
```

Algorithm 5.24: Example Quantum Program

Programs are added to the system queue by transmitting them to the Receiving Port. The system parses the program and immediately adds it to the queue for execution as soon as possible. When the program is submitted, assuming everything is copacetic, the system will return a hash of the program text. This hash is used to identify the program later and is provided along with the eventual result by the Send Port e.g. hash:result.

5.10.2 Interacting with GladeOS using Python

The majority of the tests performed throughout this Thesis take advantage of the simplicity of python networking. In Algorithm 5.25, python methods have been provided which access each of the 3 ports. The provided methods allow for use through the requests library, or standard http networking.

```
ADMIN_URL = 'http://localhost:5200'
CONTROLLER_URL = 'http://localhost:6200'
RESULTS_URL = 'http://localhost:7200'
ADMIN_SECRET = "REDACTED"
RESULTS_SECRET = "REDACTED2"

def get_results():
    results = requests.post(url=RESULTS_URL, data=
   RESULTS_SECRET)
    time.sleep(5)
    return results.json()

def get_jobs_in_progress():
    admin = requests.post(url=ADMIN_URL, data=
   ADMIN_SECRET)
    time.sleep(5)
    return admin.json()["JobsInProgress"]

def enqueue_new_job(controller):
    controller = requests.post(url=CONTROLLER_URL, data=
   controller)
    time.sleep(5)
    if controller.json()["EnqueState"] != 1:
        return False
    return True

def get_results_http():
    conn = http.client.HTTPConnection("localhost",7200)
    payload = RESULTS_SECRET
```

```
27      conn.request("POST", "undefined", payload)
28      res = conn.getresponse()
29      time.sleep(5)
30      data = res.read()
31      data = data.decode('utf-8')
32      data = json.loads(data)
33      conn.close()
34      return data
35
36  def get_jobs_in_progress_http():
37      conn = http.client.HTTPConnection("localhost",5200)
38      payload = ADMIN_SECRET
39      conn.request("POST", "undefined", payload)
40      res = conn.getresponse()
41      time.sleep(5)
42      data = res.read()
43      data = data.decode('utf-8')
44      data = json.loads(data)
45      return data["JobsInProgress"]
46
47  def enqueue_new_job_http(controller):
48      conn = http.client.HTTPConnection("localhost",6200)
49      payload = controller
50      conn.request("POST", "undefined", payload)
51      res = conn.getresponse()
52      data = res.read()
53      data = data.decode('utf-8')
54      data = json.loads(data)
55      time.sleep(0.5)
56      if data["EnqueState"] != 1:
57          return False
58      return True
```

Algorithm 5.25: Example interaction with GladeOS

5.11 Chapter Summary

The server program GladeOS works according to the details supplied above. GladeOS incorporates 10 single qubit gates and all again as multi-qubit variations. This required the development and implementation of unique mathematics (found in Section 5.6) in order to properly apply the gates and perform the relevant tasks. With the implementation of GladeOS (Research Question 3B), this Thesis can now explore the effect of multiprocessing in a quantum computer.

The system explored above in exhaustive detail is included to provide substance behind the results featured in Chapter 6. The development of GladeOS provides practical and foundational software on which the results of the later chapters can be obtained. This system was also meticulously outlined to ensure that any faults in the logic or mathematics of the system could be found, thus assisting in validating the later results.

6. ANALYSIS OF GLADEOS

Now that the base system (Chapter 4) has been outlined and the implementation (Chapter 5) has been presented, the analysis of these systems can be presented. This analysis focuses on 4 areas of performance (Research Question 3B):

1. Algorithm efficiency
2. Comparison with existing solutions
3. Accuracy tests
4. Design and Creation methodology evaluation

Section 6.1 reviews the different components of the presented systems. This review investigates the different algorithmic approaches and the performance of each. This review ends with a recommendation of which algorithm to employ for each of the identified bottlenecks.

Section 6.2 moves beyond the system presented within this Thesis and compares the performance to the other commercially available simulators. These simulators are tested by providing a library of quantum programs to be executed. This execution is tracked in terms of time required and in terms of memory usage. The accuracy of the simulators is not tested because the commercial simulators are accepted to be accurate and the accuracy of GladeOS is tested in Section 6.3.

Section 6.3 demonstrates the accuracy of the GladeOS implementation by performing a Chi-Squared test on each of the base gates (and some other small programs). Demonstrating that a quantum simulator is accurate could

be an entire extra research project due to the probabilistic nature of the simulators. Because of this the tests work by calculating the expected output for the provided programs and comparing that probability distribution to the results of executing that program 10000 times.

Finally, Section 6.4 reviews the systems presented in Chapters 4 and 5 and discusses in this Chapter against the criteria outlined in Chapter 3. This review considers the system from the many facets of the Design and Creation methodology. Together with the other sections these reviews present an overview of the performance of the system and provide a solution to Research Question 3B.

6.1 Algorithm efficiency

Built using a modular approach, the base design of the operating system is established in Chapter 4 and implemented in Chapter 5. This modular approach outlines the goal of each segment, with the actual algorithmic implementation up for debate. This modular approach highlights the main areas which can greatly impact the performance of the system. These areas are:

1. Quantum program mapping
2. Maximal clique detection
3. Program Scheduler

These bottlenecks are each discussed below, these discussions utilise graph theory approaches to ensure the outcomes remain applicable to the Glade OS system outlined in Chapter 4. Improvements in any of the the above areas will yield improvements to the entire system.

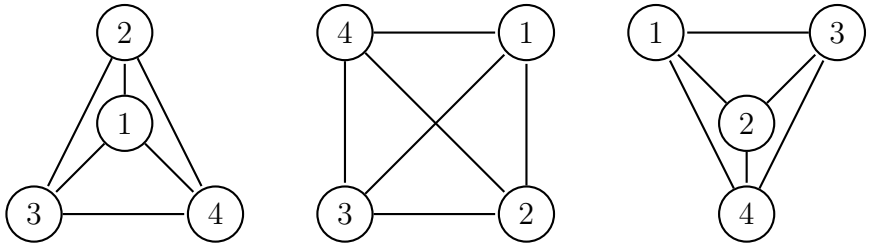
6.1.1 Quantum Program Mapping

The example provided in Section 4.4 takes the program requirements as an activity graph (Figure 4.4) and allocates available resources from the connectivity graph (Figure 4.3) in the form of a mapping (Figures 4.5, 4.6 and 4.7). Regardless of whether the quantum computer is restricted to executing a single quantum program or seeking to execute multiple quantum programs simultaneously, all programs must be mapped to qubits. This problem is similar to the register allocation problem found in classical computing where a finite set of resources are available for use and must be coordinated in order to complete the intent of the program. Currently the only available resources are qubits which are intrinsically linked to the hardware and have limited interactions between them.

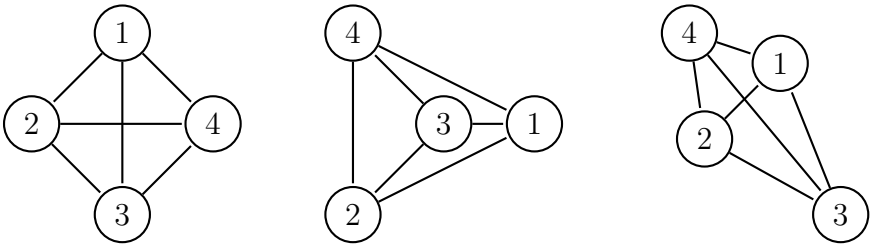
In order to map a quantum program onto the hardware we need to locate a copy of the activity graph within the connectivity graph (see Section 4.4). Locating (isomorphic subgraph) copies of the activity graph within the connectivity graph is an example of the subgraph isomorphism NP-problem. Isomorphic graphs are graphs which are structurally the same, the position of the nodes is irrelevant and what matters is the connections between them. The graphs found in Figure 6.1 are an example of this isomorphism. The subgraph component of subgraph isomorphism relates to searching for Graph G_1 inside of Graph G_2 where $G_1 \subseteq G_2$

It is not always possible to perfectly fit G_1 inside of G_2. In the case where $G_1 > G_2$ then the G_1 cannot be found within that hardware, due to insufficient resources. In the other case where $G_1 \leq G_2$ then if G_1 cannot be found within G_2, then G_1 can normally be carefully edited in order to fit the hardware graph. These edits typically take the form of a $SWAP$ function switching the state of two qubits (Figure 6.2)

This operation increases the number of logic gates and the time required which raises the error rate of the program. Inaccurate logic gates can result in errors propagating through the computation, therefore including more gates



(a) Example # 1 *(b)* Example # 2 *(c)* Example # 3



(d) Example # 4 *(e)* Example # 5 *(f)* Example # 6

Figure 6.1: Isomorphic Arrangements of the Same Distinct Graph

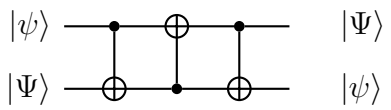
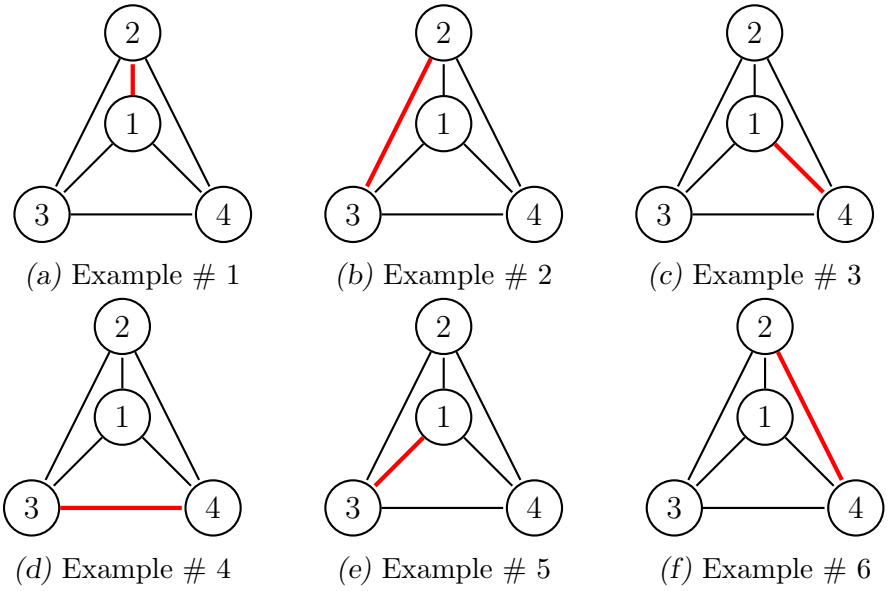


Figure 6.2: SWAP Quantum Circuit

will inevitably result in less accurate computations. Furthermore, the effect of random noise on the system which over time results in system decoherence and unusable results must also be considered. In Noisy Intermediate-Scale Quantum (NISQ) technologies error rates are naturally high and excessive swaps will only serve to raise the error rate of the program [21], [22]. There are numerous mapping algorithms, though most tend to suffer from restricting themselves to only a specific device or set of devices, or excessively poor performance in worst case (Table 6.1). The Siraichi algorithm found in Section 6.1.1.1, was chosen due to its generality, consistent performance and suitability for extension.

6.1.1.1 Siraichi Algorithm summary

The full details of this algorithm can be found in "Qubit Allocation as a combination of subgraph isomorphism and token swapping" [104], for completeness/simplicity a short summary has been included here. The Siraichi algorithm steps through the quantum program and considers all possible mappings simultaneously in order to find the lowest cost. The cost is defined according to the amount of swap and redirection operations the mapping uses and the extra operations required to transform between mappings. If the quantum program consists of a single connection, then Figure 6.3 shows all the possible edges that can be mapped.



(a) Example # 1 *(b)* Example # 2 *(c)* Example # 3

(d) Example # 4 *(e)* Example # 5 *(f)* Example # 6

Figure 6.3: All 6 Possible Edges for A → B

The Siraichi algorithm [104] has three options when considering a connection A → B:

1. If both the qubits (A and B) are unknown, then create a seperate mapping for every available edge in the hardware graph. (Fig. 6.3)

2. If one of the qubits (A or B) is unknown, then map the unknown qubit in terms of the known qubit

3. If both qubits are known (A and B) and there is an edge between them, then do nothing.

If at any point the unknown qubits are unable to be mapped or if an edge does not exist between two known qubits, then the mapping can be considered invalid. If all mappings are considered invalid, then the current mappings are saved and the next connection should be considered as a brand new mapping. This cycle repeats until the entire program has been mapped, then the algorithm reviews the different saved mappings and investigates the cost to transform from Mapping $M_1 \rightarrow M_2$ through $SWAP$ operations.

Alternative algorithms do exist, however the analysis presented in [104] and Table 6.1 demonstrates the superiority of this algorithm. New algorithms may further improve upon the performance of [104], however for now it is selected as the default model. In Table 6.1 a ✓ indicates that the algorithm has this flaw, while a ✗ indicates the opposite.

Table 6.1: Mapping algorithm comparison, different algorithms suffer from similar problems.

Algorithm	**Hardware Restrictions**	**Swaps only**	**Poor Runtime Complexity**
Shafei [103]	✓	✓	✓
Gerard [114]	✗	✓	✗
Siraichi [104]	✗	✗	✗
Lao [115]	✗	✓	✓

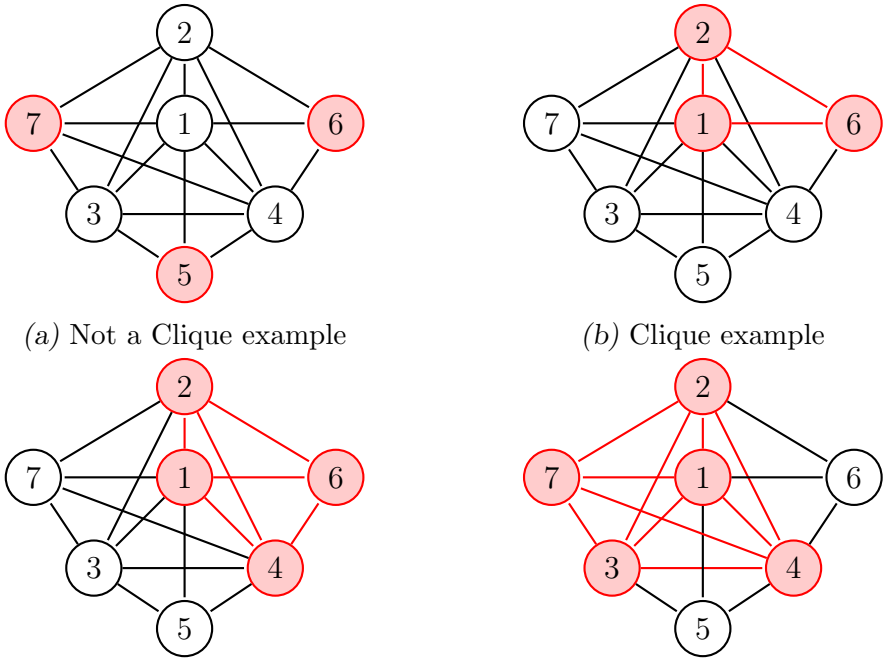
6.1.2 Maximal Clique Algorithms

A core component of the operating system presented in Chapter 4 is the ability to determine parallelisable quantum programs through the use of the dependency graph. This component relies on the ability to determine a maximal clique from within the dependency graph. To enhance this discussion, a few terms need to be defined.

A *clique* is a set of nodes within a graph, which are all directly connected to each other by edges (a distance of 1). Cliques can be made of any size or shape, the only requirement is that every node must be directly connected to every other node in the clique. An example of a set of nodes which do not form a clique can be found in Figure 6.4a. An example of a valid clique can be found in Figure 6.4b.

A *maximal clique* is a set of nodes within a graph that form a clique and cannot be extended to include any other nodes. An example of this is the difference between Figure 6.4b and Figure 6.4c. Figure 6.4b is a valid clique, however it can be extended further to include Node 4. After including Node 4, there are no more nodes which can be added to the clique and still form a valid clique.

A *maximum clique* is the largest possible clique which can be found within the given graph. A maximum clique is by definition a maximal clique, as if the clique could be extended any further then it would not be the maximum clique.



(a) Not a Clique example

(b) Clique example

(c) Maximal Clique

(d) Maximum Clique

Figure 6.4: Clique Examples

To find a clique within a graph is a relatively simple task by simply building the clique 1 node at a time and checking the edges each time. Once a clique has been identified it can then be checked as to whether it can be extended in order to confirm it as a maximal clique. There is currently no fast algorithm to locate a maximum clique [116], often requiring searching almost the entire graph in order to confirm that the maximal clique they identified is the maximum clique.

6.1.2.1 Original Algorithm

The original algorithm is designed to mimic the current implementation of executing one program at a time. To accomplish this the algorithm reads the vertices from the edge list in the order they are specified. Figure 6.5 demonstrates the execution of the algorithm, with the following key:

- Black and White nodes are uncatagorised
- Red nodes are selected
- Gray nodes are previosuly selected nodes not able to be selected again.

and the legend for Figure 6.5 is contained in Psuedocode 1 where each step results in a single node that is ready for further processing.

Psuedocode 1 Original Example Legend

1: no nodes are selected.
2: node 1 is selected and processed.
3: node 1 is deselected and node 2 is selected.
4: node 2 is deselected and node 3 is selected.
5: node 3 is deselected and node 4 is selected.
6: node 4 is deselected and node 5 is selected.
7: node 5 is deselected and node 6 is selected.
8: node 6 is deselected and node 7 is selected.

The algorithm is $O(n)$ time-space complexity, essentially just a simple loop stepping through each node one at a time (according to lexicographical ordering). There is limited calculation in this algorithm, thereby acting as the control group for this analysis.

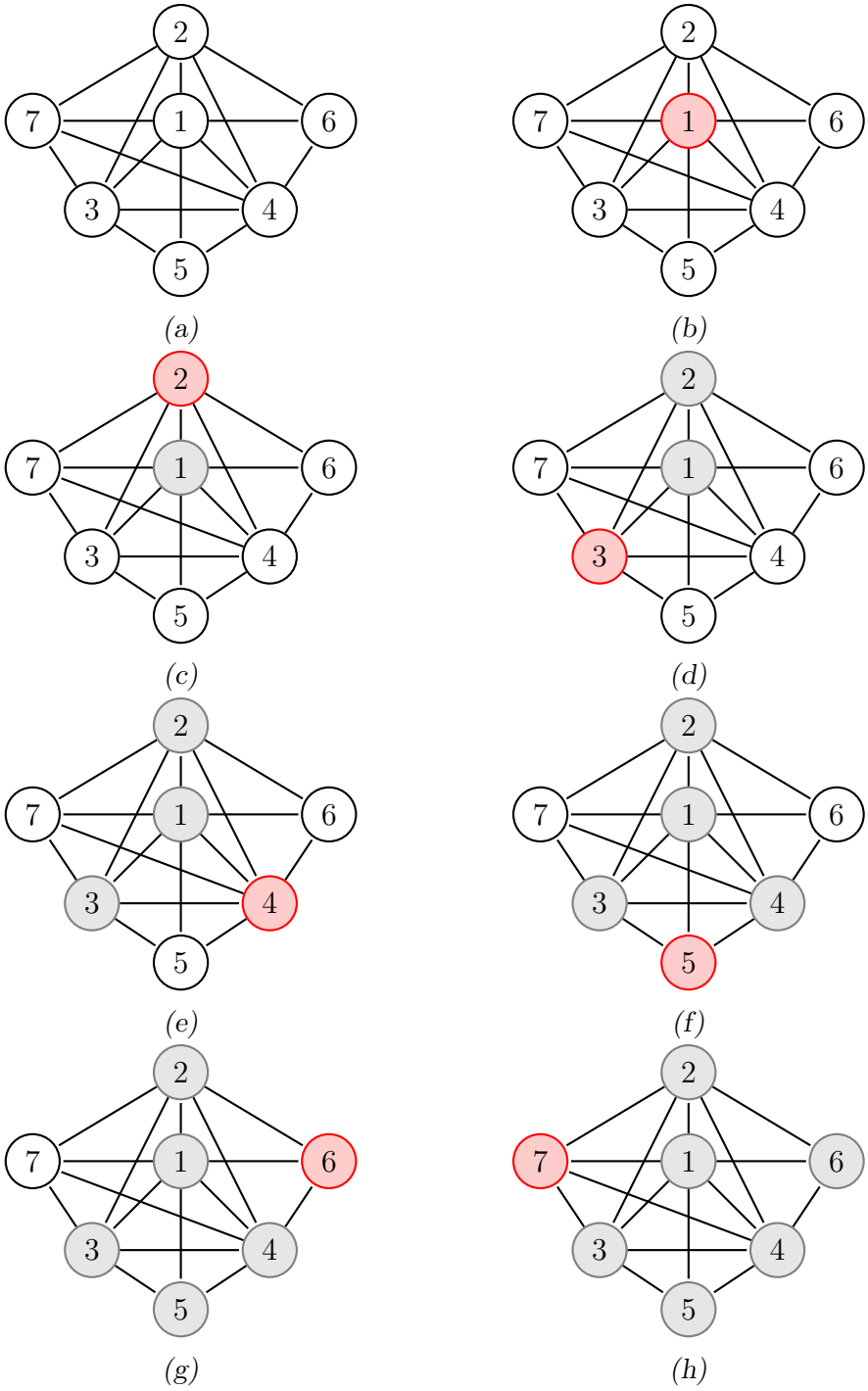


Figure 6.5: Original (Brute Force) Algorithm walkthrough. Nodes are identified in lexicographical order and processed one at a time. Black and white nodes are uncatagorised, red nodes are selected, gray nodes are previously selected nodes not able to be selected again.

```
1
2   def runAlgo3(graphs, file):
3     print("Algorithm 3")
4     print("An emulation of the existing architecture")
5     resultList = []
6     for graph in graphs:
7         cliques = []
8         clique = []
9         controller = readGraphFromFile(graph)
10        while not controller.fully_allocated():
11            source = controller.get_source()
12            clique.append(source.id)
13            source.allocate_vertex()
14            cliques.append(list(clique))
15            clique.clear()
16        resultList.append(list(cliques))
17
18    return resultList
19
20
```

Algorithm 6.1: Original Algorithm

6.1.2.2 Greedy Local Search - Set

This algorithm searches from the known node out, using only the edges from that known node as the search space. This approach searches the graph vertex list in lexicographical ordering, meaning that it favours cliques containing nodes earlier in the ordering. This also means that this approach may not find the maximum graph, but it will provide a maximal clique (dependent on what order the nodes are interacted with). The algorithm has been demonstrated in Figure 6.6, where Figures 6.6b to 6.6h demonstrate constructing the first clique from the original graph, with the following key:

- Black and White nodes are uncatagorised
- Yellow nodes are available options
- Red nodes are selected
- Gray nodes do not fit within the clique

and the legend for Figure 6.6 is contained in Psuedocode 2 which results in a clique of size 5 $\{1, 2, 3, 4, 7\}$ ready for further processing.

Psuedocode 2 Greedy Local Search Example Legend

1: Original Graph.
2: Node 1 is chosen (according to lexicographical ordering). Edges from node 1 are illuminated.
3: Node 1 has been selected, following the edges node 2 is selected and added to the clique.
4: Following the lexicorgaphical ordering of the nodes directly connected to the chosen node (#1), node 3 is assessed. As node 3 is directly connected to all nodes in the existing clique, it as added to the clique.
5: Following the lexicorgaphical ordering of the nodes directly connected to the chosen node (#1), node 4 is assessed. As node 4 is directly connected to all nodes in the existing clique, it as added to the clique.
6: Following the lexicographical ordering of the nodes directly connected to the chosen node (#1), node 5 is assessed. As node 5 is not directly connected to all nodes in the existing clique, it as left out of the clique.
7: Following the lexicorgaphical ordering of the nodes directly connected to the chosen node (#1), node 6 is assessed. As node 6 is not directly connected to all nodes in the existing clique, it as left out of the clique.
8: Following the lexicorgaphical ordering of the nodes directly connected to the chosen node (#1), node 7 is assessed. As node 7 is directly connected to all nodes in the existing clique, it as added to the clique.

A python implementation has also been provided in listing 6.2, with a time-space complexity of $O(n^2)$.

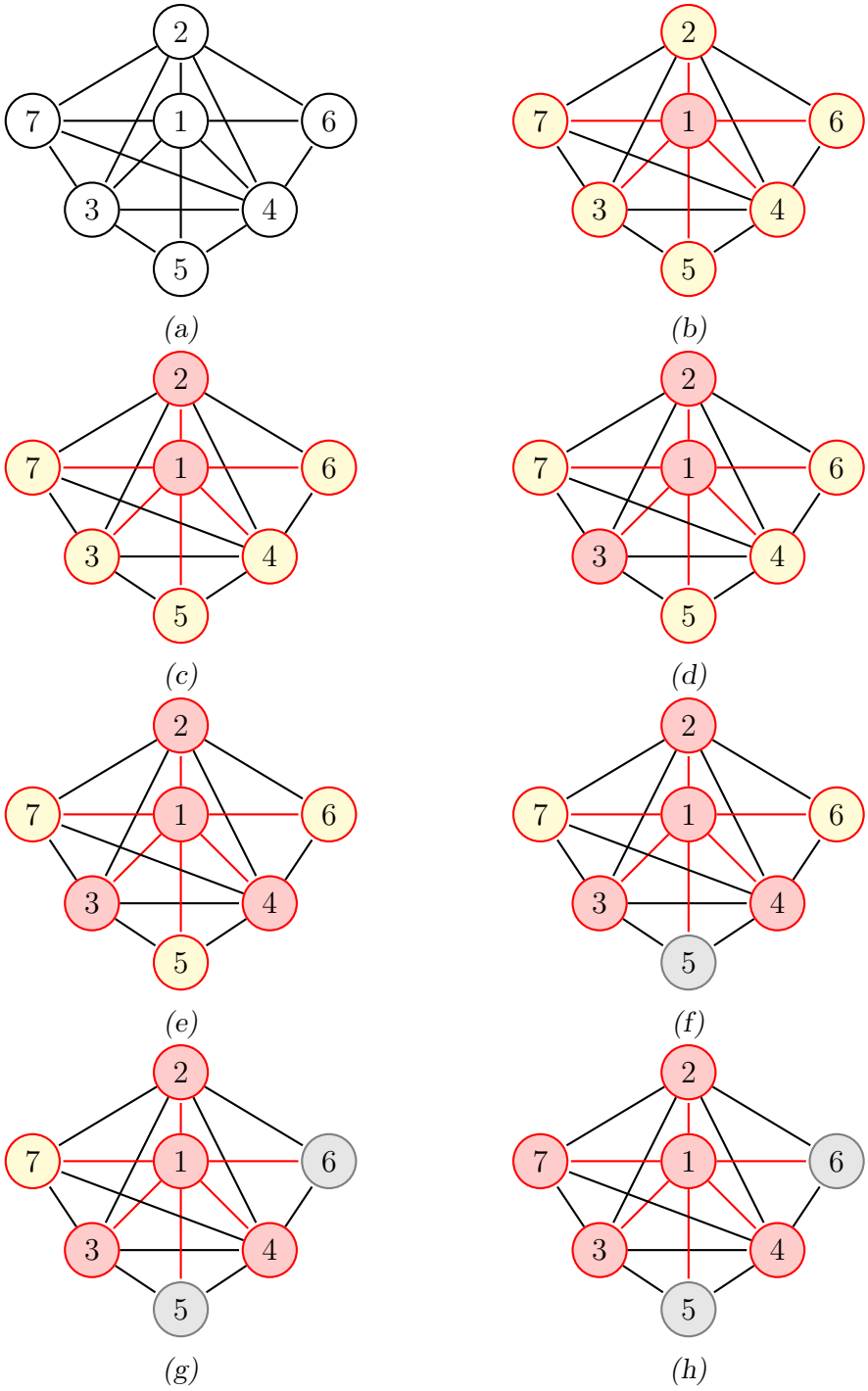


Figure 6.6: Greedy Set Algorithm Walkthrough. This algorithm processes nodes in cliques, each clique is generated by stepping through each node in a lexicographical order. Once chosen, the Nodes edges are stepped through adding new nodes to the clique one at a time, skipping past nodes which don't fit. Black and White nodes are uncatagorised, Yellow nodes are available options, Red nodes are selected and Gray nodes do not fit within the clique.

```
1
2   def runAlgo1(graphs, file):
3     print("Ego Network Approach #1")
4     print("This approach only considers the ego network
    from the specified node in a set order,")
5     print("this reduces the search space making the
    clique detection faster")
6
7     # a data structure to hold the clique
8     resultList = []
9
10    for graph in graphs:
11        # create the graph from the file
12        controller = readGraphFromFile(graph)
13        # create the required data structures
14        # to store the nodes that compose the clique
15        clique = []
16        # to store the ID of each node in clique (in the
     same order)
17        cliqueIDs = []
18        # to store the list of cliques which cover the
    graph
19        cliques = []
20        while not controller.fully_allocated():
21            # specify that the scheduled node must be
    included
22            source = controller.get_source()
23            clique.append(source)
24            cliqueIDs.append(source.get_id())
25            # Core algorithm
26            # we can ignore the nodes which are not part
     of the ego network from the source. So we step
    through all the connections from the scheduled node.
27            for possible in source.get_connections():
```

```
28                # a variable to check against to confirm
     that all the members of the clique are neighbours
29                # it is reset here for each run.
30                allneighbours = True
31                # if the node is already allocated then
    we can ignore it.
32                if possible.is_allocated():
33                    allneighbours = False
34                if allneighbours:
35                    for checkagainst in clique:
36                        # if any member of the clique is
     not connected to the others then it is not a clique
37                        if checkagainst not in possible.
    get_connections():
38                            allneighbours = False
39                # if all the neighbours is satisfied
    then add the node to the clique.
40                if allneighbours:
41                    clique.append(possible)
42                    cliqueIDs.append(possible.get_id())
43            # Mark each node in the clique as allocated
44            for vertices in clique:
45                vertices.allocate_vertex()
46            # append the clique to the list and reset
    the other lists
47            cliques.append(list(cliqueIDs))
48            clique.clear()
49            cliqueIDs.clear()
50        resultList.append(list(cliques))
51     return resultList
52
53
```

Algorithm 6.2: Greedy local search - set algorithm

6.1.2.3 Greedy Local Search - Random

This algorithm searches from the known node out, using only the edges from that known node as the search space. This approach searches the graph vertex list in random ordering, meaning that it attempts to find the optimal cliques instead of favouring lexicographical ordering. This alteration could also result in suboptimal cliques which would have been avoided by considering the nodes according to lexicographical ordering. This also means that this approach may not find the maximum clique, but it will provide a maximal clique. The algorithm has been demonstrated in Figure 6.7, where Figures 6.7b to 6.7h demonstrate constructing the first clique from the original graph, with the following key:

- Black and White nodes are uncatagorised
- Yellow nodes are available options
- Red nodes are selected
- Gray nodes do not fit within the clique

and the legend for Figure 6.7 is contained in Psuedocode 3 which results in a clique of size 5 $\{1, 2, 3, 4, 7\}$ ready for further processing.

Psuedocode 3 Greedy Local Random Search Example Legend

1: Original Graph
2: Node 1 is chosen and nodes 2,3,4,5,6 and 7 are highlighted as possibilities.
3: Node 3 is added to the chosen clique, leaving nodes 2,4,5,6 and 7 as possibilities.
4: Node 6 is tested, found to be invalid and removed from the group of possibilities.
5: Node 7 is added to the chosen clique leaving 2,4 and 5 as possibilities.
6: Node 4 is added to the chosen clique leaving 2 and 5 as possibilities.
7: Node 5 is tested, found to be invalid and removed from the group of possibilities.
8: Node 2 is added to the chosen clique leaving no further possibilities, thus ending the algorithm.

A Python implementation has also been provided in Algorithm 6.3, with a time-space complexity of $O(n(n+n)) = O(n^2)$ (assuming that the random shuffle [117] used to select a node performs similar to the Fisher-Yates shuffle [118], [119] that was used in this code, alternative algorithms will result in alternative complexities).

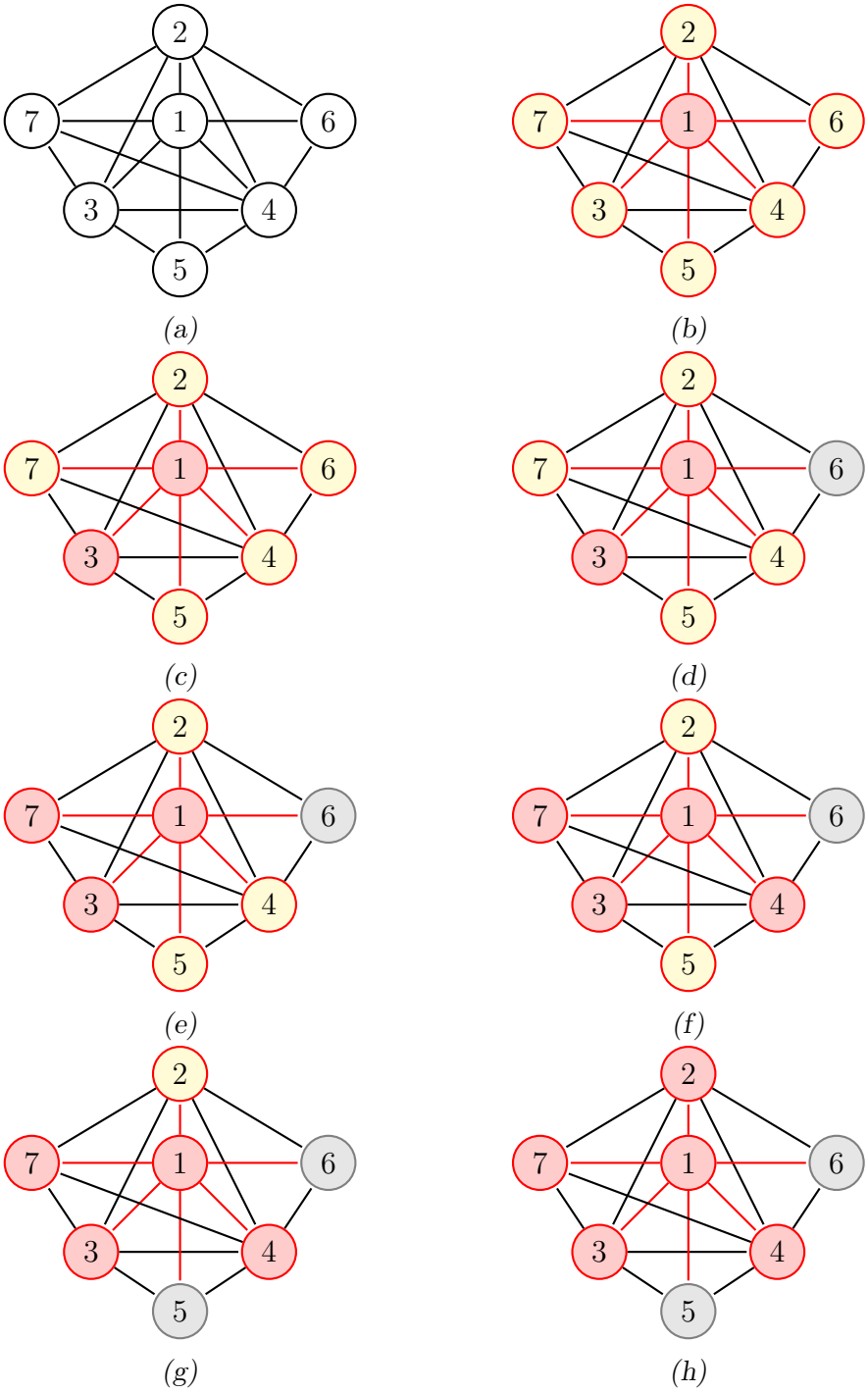


Figure 6.7: Greedy Random Algorithm Walkthrough.

```
1
2   def runAlgo2(graphs, file):
3     print("Ego Network Approach #2")
4     print("This approach only considers the ego network
    from the specified node in a random order,")
5     print("this reduces the search space making the
    clique detection faster")
6     print("")
7     resultList = []
8     for graph in graphs:
9         controller = readGraphFromFile(graph)
10        clique = []
11        clique2 = []
12        cliques = []
13        while not controller.fully_allocated():
14            # specify that the scheduled node must be
    included
15            source = controller.get_source()
16            clique.append(source)
17            clique2.append(source.get_id())
18            forLoopOutput = source.
    get_connections_random()
19            # Core algorithm
20            # we can ignore the nodes which are not part
     of the ego network from the source.
21            for possible in forLoopOutput:
22                # a variable to check against to confirm
     that all the members of the clique are neighbours
23                # it is reset here for each run.
24                allneighbours = True
25                if possible[0].is_allocated():
26                    allneighbours = False
27                if allneighbours:
28                    for checkagainst in clique:
```

```
29                      # if any member of the clique is
    not connected to the others then it is not a clique
30                      if checkagainst not in possible
    [0].get_connections():
31                          allneighbours = False
32
33              # if all the neighbours is satisfied
    then add the node to the clique.
34              if allneighbours:
35                  clique.append(possible[0])
36                  clique2.append(possible[0].get_id())
37
38          # print the clique that was found
39          for vertices in clique:
40              vertices.allocate_vertex()
41          cliques.append(list(clique2))
42          clique.clear()
43          clique2.clear()
44      resultList.append(list(cliques))
45
46   return resultList
47
48
```

Algorithm 6.3: Greedy local search - random algorithm

6.1.2.4 Maximum Graph Search

In opposition to the previous algorithms, this approach is guaranteed to find a maximum clique. This is accomplished by generating the powerset (all possible combinations of nodes) and determining all the cliques within the powerset. Then searching from the bottom (largest set possible AKA set of all nodes) until a set is found that is a clique and contains the known node. Because this approach needs to generate all possible cliques it requires much more upfront processing, though is guaranteed to return the maximum clique

at all times. The implementation found in Algorithm 6.4 features a time-space complexity of $O(n^3)$ which exceeds the other algorithms by an entire power. The performance of this algorithm is based largely on two factors, the number of nodes in the graph and the density of the graph. The more nodes present in the graph result in a larger powerset, thereby requiring more time to step through that set. As the density of the graph increases the number of powerset elements which double as a valid clique will increase and therefore require storing, overall requiring more space.

```
1  def runAlgo4(graphs, file):
2      print("Algorithm 4")
3      print("Determine all the maximal cliques, then
       select the largest clique that contains the chosen
       vertex")
4      # find the biggest clique (to maximise the number of
       programs)
5      # if the clique contains the specified node then
       great,
6      # else the clique can be disregarded and restart the
       procedure.
7      resultList = []
8      for graph in graphs:
9          controller = readGraphFromFile(graph)
10         confirmedcliques = []
11         cliques = []
12         # need some code to generate all the possible
       combinations
13         for subset in powerset(controller.get_vertices()
       ):
14             clique = True
15             # need some code to check that the
       combination is a clique
16             for index in range(len(subset)):
17                 for index2 in range(len(subset)):
```

```
18                      if index == index2:
19                          continue
20                      if controller.get_vertex(subset[
    index]) not in controller.get_vertex(
21                              subset[index2]).
    get_connections():
22                          clique = False
23              if clique:
24                  cliques.append(subset)
25          while not controller.fully_allocated():
26              confirmed = []
27              source = controller.get_source()
28              # need to confirm that the clique contains
    the necessary vertex.
29              for possible in cliques:
30                  confirmMe = True
31                  # print(possible)
32                  if source.id not in possible:
33                      confirmMe = False
34                  for node in possible:
35                      if controller.get_vertex(node).
    is_allocated():
36                          confirmMe = False
37                  if confirmMe:
38                      confirmed.append(possible)
39              for V in confirmed[len(confirmed)-1]:
40                  controller.get_vertex(V).allocate_vertex
    ()
41              confirmedcliques.append(confirmed[len(
    confirmed)-1])
42          resultList.append(list(confirmedcliques))
43
44      return resultList
```

Algorithm 6.4: Maximum Graph Algorithm

6.1.2.5 Data Set Details

In order to assess Algorithms 6.1, 6.2, 6.3 and 6.4 a series of trials is required. Because it is difficult to accurately identify the exact scenario that the algorithms will run on multiple test data sets have been established. The test files are designed to work with 2 main parameters: Nodes and Density. Nodes is defined as the number of nodes in the data set and Density is defined as the percentage of edges between the nodes. For example if we have 10 Nodes and 100% Density then we have 10 Nodes with edges between all of them (fully connected graph).

The Nodes can be found in one of five sets:

- Nodes 10-50 (small nodes)
- Nodes 50-100 (large nodes)
- Nodes 10-100 (average nodes)
- Nodes 100-1000 (extra large nodes)
- Nodes 1000-10000 (extra extra large nodes)

The Density can be found in one of three sets:

- Density 10-50 (small density)
- Density 50-100 (large density)
- Density 10-100 (average density)

To best compare the above algorithms, the data sets outlined in Table 6.2 were generated with each composed of 100 files (1 program per file). The data sets vary in number of nodes and the density of edges. The number of nodes varies because the number of quantum programs in the queue is expected to vary. Similarly because an edge indicates programs which can be run in parallel, the density of edges is expected to vary over the lifetime of the system.

Table 6.2: Data Set details

Data Set	**Nodes**	**Density**
SNSD	10-50	10-50
SNLD	10-50	50-100
SNAD	10-50	10-100
LNSD	50-100	10-50
LNLD	50-100	50-100
LNAD	50-100	10-100
ANSD	10-100	10-50
ANLD	10-100	50-100
ANAD	10-100	10-100
XLNSD	100-1000	10-50
XLNLD	100-1000	50-100
XLNAD	100-1000	10-100

6.1.2.6 Data Results

The purpose of these experiments are not to determine the best algorithm for a specific use case, rather an attempt to determine a general best practice. Because the goal is to review the results generally, the results have been averaged across the entire data set. The results are composed of two parts, *cliques* which count the average number of cliques found and *size* which marks the average size of each clique.

Note: to appropriately simulate the ego random algorithm the algorithm is executed 10 times and then averaged. This average value is used due to the fluctuations in the results due to the random shuffle. The other algorithms have a consistent performance so they are not averaged.

Table 6.3: Maximal clique algorithm comparison

Data Set	**Details**	**Algorithm 1 - Imitation**	**Algorithm 2 - Ego Set**	**Algorithm 3 - Ego Random**
SNSD	Files: 100 $\mu(nodes)$: 28.84 $\sigma(nodes)$: 13.7 $\mu(edges)$: 295.56 $\sigma(edges)$: 267.4	Cliques: 28.84 Size: 1.0	Cliques: 8.88 Size: 3.25	Cliques: 8.89 Size: 3.24
SNLD	Files: 100 $\mu(nodes)$: 24.69 $\sigma(nodes)$: 14.39 $\mu(edges)$: 626.81 $\sigma(edges)$: 626.14	Cliques: 24.69 Size: 1.0	Cliques: 2.72 Size: 9.069	Cliques: 2.7 Size: 9.12
SNAD	Files: 100 $\mu(nodes)$: 24.93 $\sigma(nodes)$: 14.66 $\mu(edges)$: 398.92 $\sigma(edges)$: 409.34	Cliques: 24.93 Size: 1.0	Cliques: 5.57 Size: 4.47	Cliques: 5.58 Size: 4.47
LNSD	Files: 100 $\mu(nodes)$: 73.16 $\sigma(nodes)$: 15.79 $\mu(edges)$: 1609.36 $\sigma(edges)$: 955.96	Cliques: 73.16 Size: 1.0	Cliques: 17.85 Size: 4.098	Cliques: 17.79 Size: 4.11
LNLD	Files: 100 $\mu(nodes)$: 75.47 $\sigma(nodes)$: 16.21 $\mu(edges)$: 4460.38 $\sigma(edges)$: 1899.83	Cliques: 75.47 Size: 1.0	Cliques: 5.07 Size: 14.89	Cliques: 5.05 Size: 14.94

Table 6.3: Maximal clique algorithm comparison

Data Set	**Details**	**Algorithm 1 - Imitation**	**Algorithm 2 - Ego Set**	**Algorithm 3 - Ego Random**
LNAD	Files: 100 $\mu(nodes)$: 73.18 $\sigma(nodes)$: 15.85 $\mu(edges)$: 3132 $\sigma(edges)$: 2115.35	Cliques: 73.18 Size: 1.0	Cliques: 10.4 Size: 7.04	Cliques: 10.47 Size: 6.99
ANSD	Files: 100 $\mu(nodes)$: 49.13 $\sigma(nodes)$: 27.68 $\mu(edges)$: 904.44 $\sigma(edges)$: 967.11	Cliques: 49.13 Size: 1.0	Cliques: 13.22 Size: 3.72	Cliques: 13.11 Size: 3.75
ANLD	Files: 100 $\mu(nodes)$: 50.63 $\sigma(nodes)$: 28.96 $\mu(edges)$: 2552.91 $\sigma(edges)$: 2308.99	Cliques: 50.63 Size: 1.0	Cliques: 4.06 Size: 12.61	Cliques: 4.01 Size: 12.61
ANAD	Files: 100 $\mu(nodes)$: 44.71 $\sigma(nodes)$: 27.4 $\mu(edges)$: 1380.49 $\sigma(edges)$: 1714.16	Cliques: 44.71 Size: 1.0	Cliques: 8.51 Size: 5.25	Cliques: 8.51 Size: 5.25
XLNSD	Files: 100 $\mu(nodes)$: 535.57 $\sigma(nodes)$: 270.85 $\mu(edges)$: 83322.06 $\sigma(edges)$: 79419.67	Cliques: 535.57 Size: 1.0	Cliques: 86.58 Size: 6.18	Cliques: 86.26 Size: 6.2

Table 6.3: Maximal clique algorithm comparison

Data Set	**Details**	**Algorithm 1 - Imitation**	**Algorithm 2 - Ego Set**	**Algorithm 3 - Ego Random**
XLNLD	Files: 100 $\mu(nodes)$: 545.12 $\sigma(nodes)$: 239.99 $\mu(edges)$: 161606.89 $\sigma(edges)$: 123920.41	Cliques: 545.12 Size: 1.0	Cliques: 20.37 Size: 26.89	Cliques: 20.27 Size: 26.89
XLNAD	Files: 100 $\mu(nodes)$: 526.21 $\sigma(nodes)$: 275.54 $\mu(edges)$: 127983.31 $\sigma(edges)$: 118264.02	Cliques: 526.2 Size: 1.0	Cliques: 43.99 Size: 11.96	Cliques: 43.78 Size: 12.02

6.1.2.7 Analysis of Results

Analysis of the results found in Table 6.3 reveals a handful of verifiable trends. The original (imitation) algorithm currently being used to schedule quantum programs will always complete and return a result within a standard time. The greedy local search algorithms returns a smaller set of resultant cliques for largely the same execution time. The difference between the random and set variations of this algorithm are largely minute, though on average the random variation does tend towards slightly larger cliques thereby resulting in slightly less cliques overall.

Conversely the Maximal clique algorithm (brute force implementation) is guaranteed to return an optimal strategy, but suffers from an increase in time which parallels the growth in the factorial series. The maximum graph search algorithm has been omitted from Table 6.3 due to excessive execution times. For instance, attempts to simulate data set SNSD and SNAD required multiple hour computations including some graphs in excess of 648 hours ($\sim$27 days) on hardware detailed in Table 6.6. These excessive execu-

tion times render this algorithm ill suited to this task.

The graphs this algorithm is expected to process over is difficult to estimate. The number of nodes in the graph will be equal to either the entire length of the queue or a defined subset of the queue. The number of edges (density of the graph) will largely be determined by the size of the programs in the queue and the size of the quantum computer being utilised. The worst case scenario of a large (or extra large) queue of small jobs being executed on a large quantum computer is expected to mirror the XLNLD data set, which is featured in Table 6.3. Reducing the queue length or only taking a subset of the queue at a time will drop the XL $\rightarrow$ S, while jobs which closely fit the quantum computer will lower the density of the graph.

Because of the varying sizes of the graph it is difficult to perfectly prescribe an algorithm. The original and maximum search algorithm can easily be excluded because of their excessive output and execution times, leaving only the two greedy local search algorithms. There are cases where the greedy local search algorithm performs better than the random variation however those times are few and far in between (only 3 of the results in Table 6.3 show this). It is for these reasons that moving forward, the recommendation is to adopt the greedy local search algorithm with a random variation.

6.1.3 Scheduling Algorithms

After the quantum programs have been optimised and mapped, the system must determine which program to execute first. The major difference between scheduling for a classical computer and a quantum computer is that due to decoherence and a lack of functioning quantum memory, preemptive scheduling algorithms are not recommended. Leaving a quantum process partially complete and unattended results in errors corrupting the data and without quantum memory, results cannot be saved for reuse later. This means that once you start a program, the wisest choice is to complete the execution. Combined with the maximal clique detection outlined in Section

6.1.2.4 the programs will not be executed in a static schedule, depending on the relationships between the programs, but it will continue to execute all available programs. The different scheduling algorithms for this analysis are:

First In First Out (FIFO) As each program is sent into the operating system it is added to the tail of the queue, the next programs are selected from the head of the queue [72], [74]. In the worst case scenario, where no programs can be processed concurrently, all programs will be executed in the order they arrive in [72], [74]. This algorithm has no chance of program starvation (a program never executing) [72], [74]. FIFO focuses on treating every program equally and to best ensure that programs are executed in order to minimise the wait time for all programs [72], [74].

Last In First Out (LIFO) As each program is sent into the operating system it is added to the head of the queue, the next programs are selected from the head of the queue [72]. In the worst case scenario, where no programs can be processed concurrently, all programs will be executed in the reverse order they arrive in [72]. This algorithm has a high chance of program starvation while the amount of new programs being added outweighs the number of programs being executed [72]. This algorithm relies on the number of new programs slowing down and allowing the stored programs to execute. LIFO is designed to focus on the most recent programs as the programs which the user wants to execute at this time, the other programs can be left for later (if the user has waited this long, they can wait even longer) [72]. This design decision is more accurate when only considering a single users programs, instead of multiple users all attempting to take advantage of the computing power.

Priority Queue This algorithm adds an extra layer of consideration by treating programs as individual programs instead of merely one of a set [72], [74]. By treating programs as individuals, it becomes possible to consider the priority of the program as the deciding factor. Program priority can be found from a variety of factors including arrival time (if the only factor then it is

akin to FIFO or LIFO), which user submitted it, or how long it has been idle [72], [74]. This algorithm commits to executing programs based on how important they are, it can suffer from program starvation for low priority programs [72], [74]. This algorithm does not provide a set execution plan, instead capitulating to the priorities as they adapt. For the test performed in Table 6.4 priority was simulated by performing a random shuffle (using the Fisher-Yates shuffle [118], [119]) on the queue (originally ordered according to FIFO principles).

Testing Rationale Because the quantum programs are stored within a graph data structure, it is difficult to evaluate the efficiency of scheduling algorithms without properly considering the effect of the graph. Due to this, the scheduling algorithm analysis is combined with the clique detection algorithms considered in Section 6.1.2. The programs are ordered according to the scheduling algorithm being tested before completing the same dataset with the greedy local clique algorithm with a random variant prescribed above.

The difference between scheduling algorithms is expected to be relatively minor, and it is therefore expected that larger data sets will demonstrate this difference more clearly. The result of these estimations are that the data set being used for this experiment is the XLN* datasets (XLNSD, XLNLD and XLNAD). The data in Table 6.4 represents the average (mean) of processing the graphs according to the algorithms selected.

Table 6.4: Scheduling algorithm comparison

Data Set	**Details**	**FIFO**	**LIFO**	**Priority Queue**
XLNSD	Files: 100 $\mu(nodes) = 535.57$ $\sigma(nodes) = 270.85$ $\mu(edges) = 83322.06$ $\sigma(edges) = 79419.67$	Cliques:86.36 Size:6.2	Cliques:86.28 Size: 6.2	Cliques:86.29 Size:6.2
XLNLD	Files: 100 $\mu(nodes) = 545.1$ $\sigma(nodes) = 239.99$ $\mu(edges) = 161606.89$ $\sigma(edges) = 123920.41$	Cliques:20.33 Size:26.81	Cliques:20.31 Size:26.82	Cliques:20.31 Size:26.82
XLNAD	Files: 100 $\mu(nodes) = 526.20$ $\sigma(nodes) = 275.54$ $\mu(edges) = 127983.30$ $\sigma(edges) = 118264.02$	Cliques:43.74 Size:12.03	Cliques:43.76 Size:12.02	Cliques:43.79 Size:12.02

Reviewing the data from Table 6.4, it appears that regarding the number of cliques LIFO consistently performs as either the best option or the second best. Conversely, the FIFO and Priority Queue implementations swing from being the best in some cases to the complete worst in others. It should be noted that the difference between the best and worst algorithm is typically only a couple of decimal points, for example $43.74 \rightarrow 43.78$. Because of this small swing of 0.04 and the inherent starvation risk associated with LIFO it is the recommendation of this analysis that FIFO be the scheduling algorithm employed, though the impact of this choice appears relatively minor. Priority Queue was strongly considered however this approach has an added cost of managing the priority of the programs which is not required for FIFO or LIFO.

In the worst case situation, each of the scheduling algorithms considered above will perform by processing a single quantum program before continuing onto the next one. This limit forms an upper bound for the execution time with the clique detection algorithms acting to lower that bound where possible. Because of this, the choice of scheduling algorithm has little to no impact on how long it takes to execute the entire program queue. Any scheduling algorithm can be used with this system, with the only consideration being that it should be starvation-free.

6.2 Overall Simulator Comparison

6.2.1 Data Sources

The data used in the tests is generated according to the script found in Appendix A.1. The odds of getting any one of the base gates is $\frac{1}{10}$ and any of the controlled gates is $\frac{1}{90}$. The difference is due to the controlled gates only being actionable when the 'C' gate is selected from the initial random generation. By editing the script in Listing 6.5, the following variables can be defined:

1. Number of files.
2. Number of qubits for each file.
3. Number of gates per file.
4. Restricting the gates for each file.

The testing procedure for this system is explored in Figure 6.8. Test files are generated according to the generation script above, before being funneled into the appropriate simulators. Finally the results are recorded using the Python memory-profiler to chart the time cost against the memory cost.

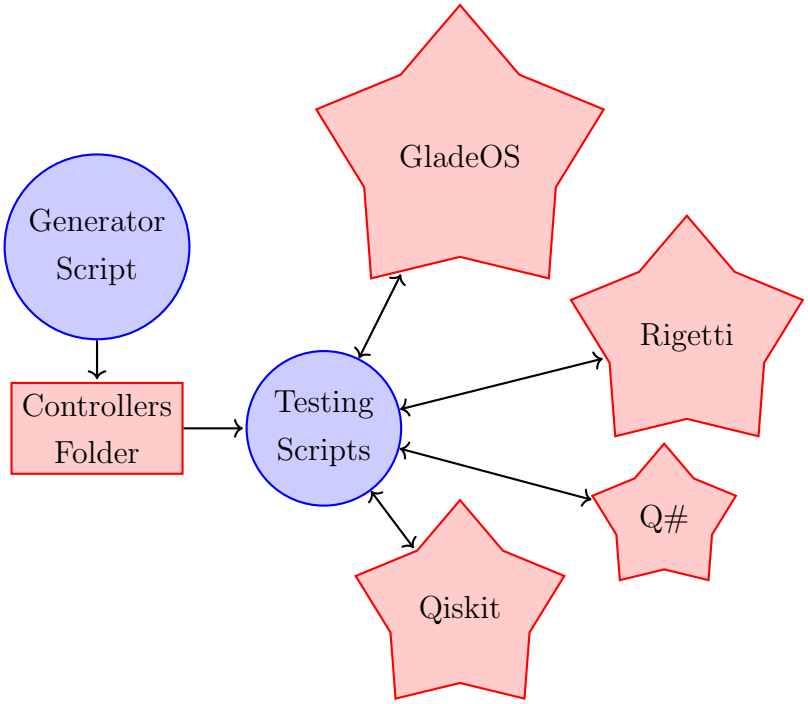


Figure 6.8: Testing Procedure

The testing script below is also editable, with the following options:

1. Which subset of simulators you want to execute the files through
2. How many times each file is executed.

```
# this file is to run all the tests
# run GladeOS
# run Qiskit
# run rigetti
# etc....

import datetime
import os
import subprocess
import sys
import time
```

```
12 import qsharp
13 from memory_profiler import profile  # comment this out
   for mprof plot
14
15 from glade.Glade import GladeTester
16 from qiskitDirectory.QISKIT import QiskitTester
17 from rigetti.RIGETTI import RigettiTester
18 import qsharpDirectory
19 from qsharpDirectory.QSHARPparser import QsharpParser
20 from qsharpDirectory.QSHARP import QsharpTester
21 import importlib
22
23 def getTime():
24     return datetime.datetime.now()
25
26 @profile
27 def runQiskit(controllers, num_shots):
28     for controllerName in controllers:
29         qiskittester = QiskitTester(controllerfilenames=
   controllerName, shots=num_shots)
30         qiskittester.start()
31         del qiskittester
32
33 @profile
34 def runRigetti(controllers, num_shots):
35     pqvm = subprocess.Popen(["qvm", "-S"])
36     pquilc = subprocess.Popen(["quilc", "-S"])
37
38     rigettiTester = RigettiTester(controllerfilenames=
   controllers, shots=num_shots)
39     rigettiTester.start()
40     del rigettiTester
41     pqvm.kill()
42     pquilc.kill()
43
```

```
44 @profile
45 def runQsharp(controllers, shots):
46     # shots = 1
47 
48     for cf in controllers:
49         parser = QsharpParser()  # create a parser
50         parser.start(controllerfilename=cf)  # create a
   new operation.qs file
51 
52         qsharp.reload()
53         tester = QsharpTester()  # tester to run the
   operation.qs file
54         for x in range(shots):
55             tester.start()  # actually run the operation
   .qs file
56         printNewlines(2)
57         del parser
58         del tester
59 
60 @profile
61 def runGlade(controllers, num_shots):
62     subs = 0
63     directory = 'Controllers/'
64     # open glade with default args
65     args = ['glade/Glade_Executable/GladeOS', '-msm', '-
   t 1', '--logalltoconsole'] #, '-mt 5', '-msm FIFO',
   '--logalltoconsole'] # , '--logalltoconsole']
66     returnCode = subprocess.Popen(args)
67     time.sleep(5)
68     # Make us a 'GladeOS' Object to call stuff with.
69     gt = GladeTester(controllerfilenames=controllers,
   shots=num_shots)
70     # Start it.
71     gt.start()
72     returnCode.kill()
```

```
73      del gt
74
75  def printNewlines(number):
76      for x in range(number):
77          print("")
78
79  @profile
80  def runMe():
81      print()
82      print("========================================")
83      print("     TestRunner Python file V1.0")
84      print("========================================")
85      print("Script start")
86
87      runGladeTrigger = False
88      runQiskitTrigger = False
89      runRigettiTrigger = False
90      runQsharpTrigger = True
91
92      sleepTime = 3
93
94      if len(sys.argv) > 1:
95          shots = sys.argv[1]
96      else:
97          shots = 5
98
99      preTime = getTime()
100
101     time.sleep(int(sleepTime))
102     directory = 'Controllers/'
103     controllers = []
104     for filename in os.listdir(directory):
105         if filename.endswith(".txt"):
106             controllers.append(directory+filename)
107             continue
```

```
108         else:
109             continue
110     controllers.sort()
111     time.sleep(int(sleepTime))
112     print(controllers)
113     # run Glade.
114     if runGladeTrigger:
115         runGlade(controllers, int(shots))
116     gladeTime = getTime()
117     printNewlines(3)
118     time.sleep(int(sleepTime))
119 
120     # run qiskit.
121     if runQiskitTrigger:
122         runQiskit(controllers, int(shots))
123     qiskitTime = getTime()
124     printNewlines(3)
125     time.sleep(int(sleepTime))
126 
127     # run rigetti.
128     if runRigettiTrigger:
129         runRigetti(controllers, int(shots))
130     rigettiTime = getTime()
131     printNewlines(3)
132     time.sleep(int(sleepTime))
133 
134     # run qsharp
135     if runQsharpTrigger:
136         runQsharp(controllers, int(shots))
137     qsharpTime = getTime()
138     printNewlines(3)
139     time.sleep(int(sleepTime))
140 
141     totalTime = getTime()
142     print("========================================")
```

```
143     print("Global script time taken: " + str(totalTime -
        preTime))
144     print("Qiskit script time taken: " + str(qiskitTime
    - preTime))
145     print("Rigetti script time taken: " + str(
    rigettiTime - qiskitTime))
146     print("Qscript time taken: " + str(qsharpTime -
    rigettiTime))
147     print("Glade script time taken: " + str(qsharpTime -
        gladeTime))
148     print("Script Completed")
149
150 runMe()
```

Algorithm 6.5: Testing Script

6.2.2 Issues Encountered

The gathering of data has not been without issue. The following sections cover those issues including issues with some of the simulators.

6.2.2.1 General problems

As shown in the feature matrix (Table 6.5), the simulators differ considerably in their design. This leads to difficulties comparing two simulators together as each is built to their own specification and for their own purpose. Qiskit and Q# were developed to research and develop quantum computer programming. Due to this design choice, while the program optimisation and execution are extremely rapid they seem to forget simulating critical aspects of quantum hardware, ignoring potential bottlenecks and thus being overly optimistic on performance measures. Rigetti conversely was designed to fully simulate a quantum computer and everything that entails. To that end, Rigetti utilises server programs to handle the simulation load.

Table 6.5: Quantum simulator feature matrix

Feature	GladeOS	Qiskit	Q#	Rigetti
Program Execution	✓	✓	✓	✓
Program optimisation	✗	✓	✓	✓
Hardware emulation	✓	✓	✗	✓
Program mapping	✓	✗	✗	✓
Multithreaded execution	✓	✓	✓	✓
Decoherence emulation	✓	✗	✗	✗

Due to inconsistencies in the quantum circuit language between simulators, some instructions have had to be added or composed of multiple smaller instructions to enable equal testing of all the simulators. All the simulators support a completely connected qubit connectivity graph thus enabling all qubits to be entangled with all others, this does simplify the previously introduced 3 layer graph structure by removing the mappings and simply allocating qubits to programs.

6.2.2.2 IBM Qiskit

Qiskit is a very popular choice of quantum computer simulator with extensive libraries of code to better support that execution. Qiskit libraries range from bare metal hardware programming to high level artificial intelligence libraries. Running a standard instance of Qiskit, is very user friendly though deceptive with the options. Qiskit employs a version of OpenMP embedded within itself. While this is not hidden from the user, it is not made clear either. In order to get the best comparison possible, equivalent settings and options are required. Qiskit naturally defaults to a multi-threaded version which undermines the nature of the comparison. This issue was combated by restricting the "max_parallel_shots=1" [120] and "max_parallel_threads=1" [120] thereby restricting the system to only executing one 'shot' (circuit) at a time and allowing for a valid comparison.

6.2.2.3 Rigetti

Rigetti is perhaps the most accurate simulator on the market, sacrificing speed for accuracy. The difficulty with Rigetti's QVM and QUILC server programs are the antiquated software stack required to support it. The pre-requisites are:

1. Standard UNIX build tools
2. SBCL (a recent version, but not SBCL 1.5.6): Common Lisp compiler
3. Quicklisp: Common Lisp library manager
4. ZeroMQ: Messaging library required by RPCQ. Development headers are required at build time.

These pre-requisites prove (in my experience) to be require excessive precision to correctly install (on a base Ubuntu 20.04 install).

6.2.2.4 Microsoft Q#

Microsoft's quantum ambitions have recently been set back with the retraced article on the Majarona particles [40]. During this storm, they have continued to further develop their quantum simulator. The major limitation of Q# is the requirement to compile the code which takes longer depending on the length of the files. This limitation appears when trying to execute multiple distinct programs in sequential order.

Previous attempts to profile the Q# simulator have required new programs to overwrite a specific file (on disk) which was then imported into the program and executed. To change to the next program, the new program had to overwrite the existing program and then re-import the file so that the changes would be detected and recognised. This method proved to be extensively difficult to import modules and files correctly without constantly importing the qsharp library unnecessarily.

This limitation has been addressed by using the compile command and instead of writing to a file, merely holding the string in memory and compiling it when required.

6.2.3 Results

The formula to calculate the standard error is found in Equation 6.1, with the following legend:

- SE is the standard error of the sample
- σ is the standard deviation of the sample
- n is the number of samples

$$SE = \frac{\sigma}{\sqrt{n}} \tag{6.1}$$

Equation 6.1 is used throughout this section to calculate the error rate of the time and memory measurements for the different simulators. To accomplish this the samples are composed of the time each distinct execution required and the maximum amount of memory that was required in each distinct execution respectively.

Table 6.6: Details of the system which conducted the experiments

Software	**Version**
Operating System	Ubuntu 20.04
RAM	16GB
CPU	Intel i7-6770HQ

6.2.3.1 Qiskit

Table 6.7: Packages and versions used to evaluate the Qiskit platform

Software	**Version**
Python	3.8.5
Qiskit	0.23.5
Qiskit Aer	0.7.4
Qiskit Aqua	0.8.2
Qiskit IBMq Provider	0.11.1
Qiskit Ignis	0.5.2
Qiskit Terra	0.16.4

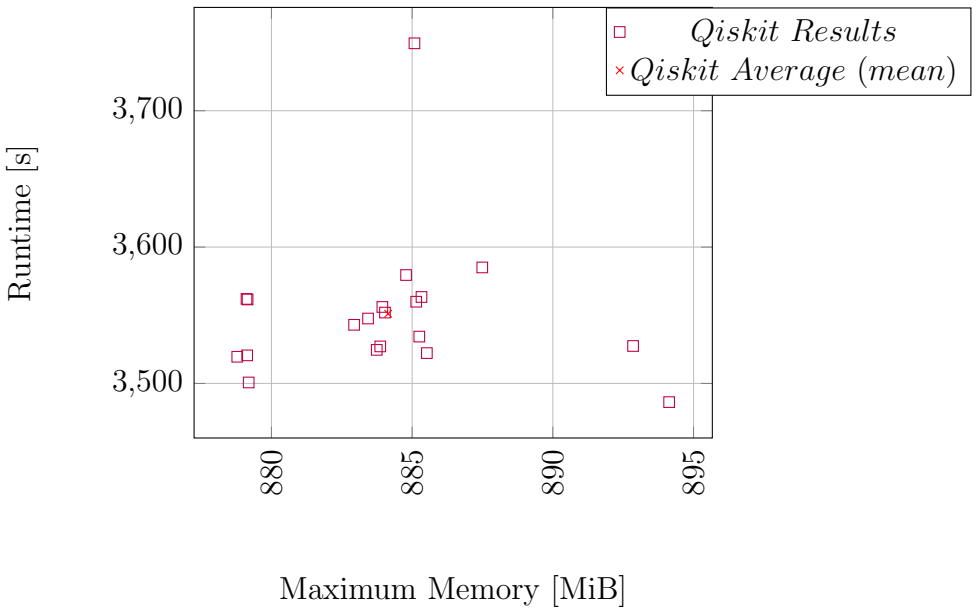


Figure 6.9: A scatterplot of the time and memory measurements recorded over 20 executions of the 100 program test using the Qiskit platform

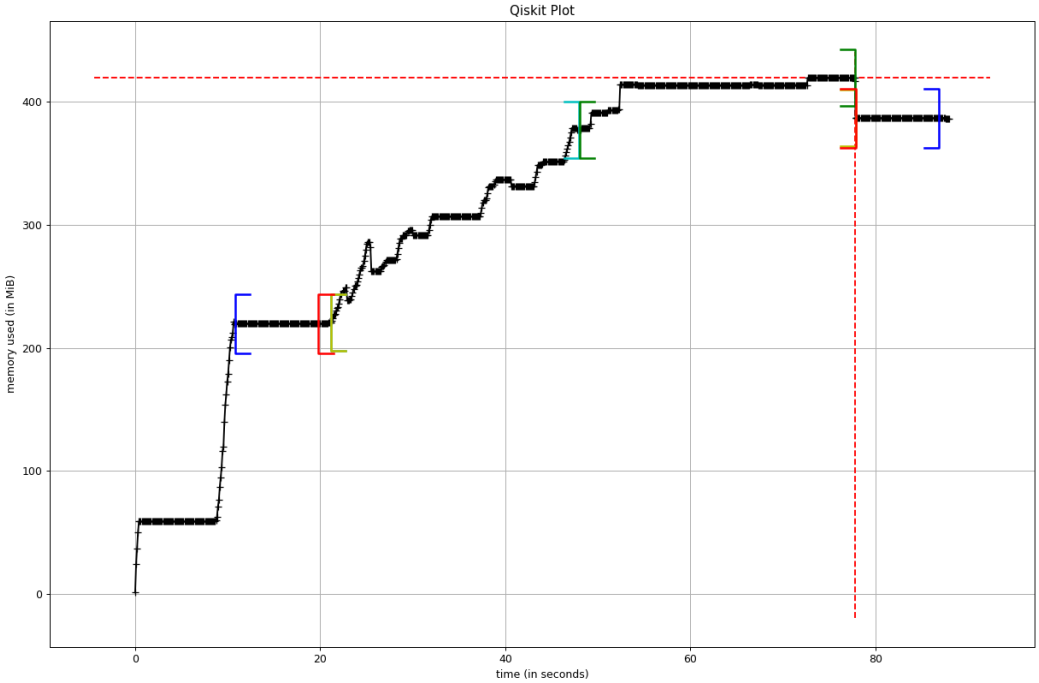


Figure 6.10: A plot of the memory usage across time using the Qiskit platform

Legend:

[ ] compile_all 26.847s

[ ] start 56.694s

[ ] run_all_controllers 29.797s

[ ] runQiskit 58.063s

[ ] runMe 76.082s

The data captured in Figure 6.9 demonstrates that the Qiskit quantum simulator takes on average 3551.12 ± 11.86 seconds and consumes 884.15 ± 0.924 MiB. The data in Figure 6.9 ranges between 3486.3 and 3749.45 in execution time (difference of 263.14) and between 878.785 and 894.13 in memory consumption (difference of 15.35).

The Qiskit simulator is provided as open source Python package available through github (`https://github.com/Qiskit`) and common Python package managers like pip. Qiskit benefits from a modular approach which enables users to be hyper-specific about the relevant components or to leave it largely up to the package to handle it for you. For the purposes of this test the specifics have been largely left to Qiskit, so as to simulate a standard instance. The only specified piece is the use of the *'qasm_simulator'* as the backend, while everything else has been left to the package.

IBMs' Qiskit packages form an ecosystem attempting to cover all parts of quantum computing (Table 6.8). Which extends the standard simulation package extensively, providing options not found in either of the four other simulators. For this reason the test was limited to only include functionality that was consistently available on all platforms.

The graph found in Figure 6.10 demonstrates the amount of memory used by a typical execution of the Qiskit platform over time. It has a series of useful sections marked to assist in the understanding of the information. The sections correspond with methods within the Python script:

- runMe - This method is responsible for reading the data set and then starting the relevant simulator.
- runQiskit - This method is responsible for preparing and executing all parts of the Qiskit script.
- start - This method is responsible for compiling each of the programs (see below) and executing them on the Qiskit platform.
- Compile_all - This method is responsible for compiling each of the programs found in the data set for execution with the rigetti platform.
- run_all_controllers - This method is responsible for executing each of the compiled programs on the Qiskit platform

It can be seen that most of the memory growth stems from the compilation of Qiskit programs before slowly increasing throughout the programs execution. There are also 2 red dashed lines (1 horizontal and 1 vertical) which mark the point where the memory consumption is at its peak. The system originally launches to a base consumption of roughly $219.67MiB$ before growing linearly throughout the compilation routine. Following the compilation routine the memory growth slows greatly. While the programs are executed, the memory wavers slightly while slowly creeping upwards. Following the execution, the memory usage scales back down to a level approximately $419.48MiB$ as the Qiskit ecosystem is closed down.

Table 6.8: Qiskit components

Package	**Coverage**
Aer	Simulation of quantum computers
Aqua	NISQ algorithms
Ignis	Error Correction and noise reduction
Metal	Development of quantum hardware
Terra	Core component, basic elements like quantum circuits, pulses and managing interfaces between components

The graph found in Figure 6.10 demonstrates a largely variable runtime graph. The graph separates the test into compilation and execution routines, which combine to form the test. The compilation routine features little consistency, instead favoring a significant swing up followed immediately by a small swing down and a period of constant memory usage. This pattern continues throughout the compilation and results in steady growth upwards. The interesting part comes from within the execution routine. The memory usage continues to grow during the execution routine, though it ignores the pattern from the compilation routine and instead slowly grows as the executions require it. Following the test, the system still retains a large growth

from approximatelty 219.67MiB up to approximately 419.48MiB. It is assumed that the retained growth stems from storing the compiled programs and the results from the executions.

6.2.3.2 Rigetti

Table 6.9: Packages and versions used to evaluate the Rigetti platform

Software	**Version**
Python	3.8.5
QVM	1.17.1
QUILC	1.23.0
PyQuil	2.28.0
SBCL	2.0.1.debian
ZeroMQ	4.3.2-2ubuntu1 amd64
Quicklisp	2021-02-13
Magicl	0.9.1
RPCQ	3.8.0

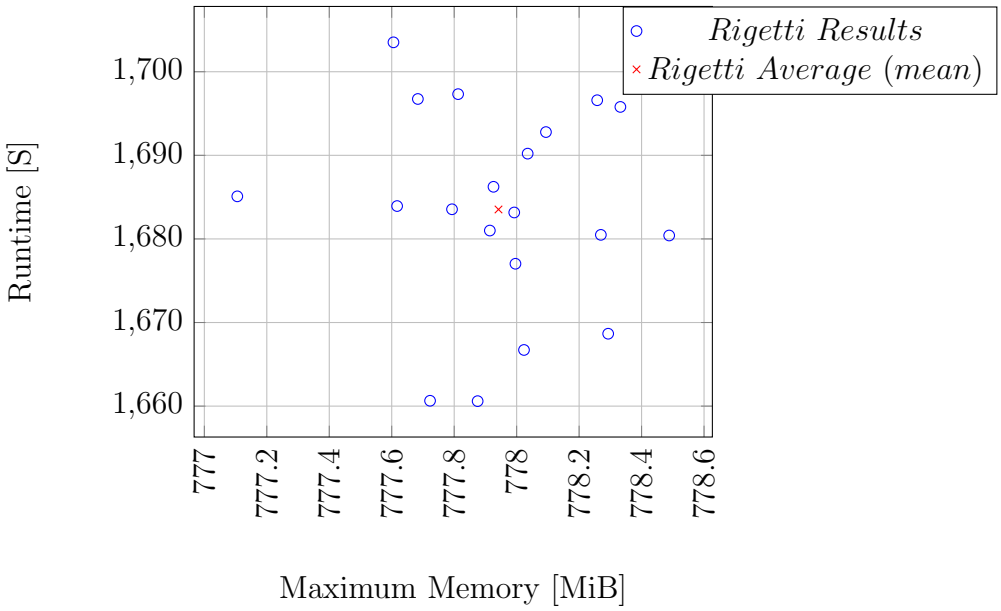


Figure 6.11: A scatterplot of the time and memory measurements recorded over 20 executions of the 100 program test using the Rigetti platform

The Rigetti simulator contains perhaps the most difficult build requirements of the four simulators tested here. The requirements include:

1. The QVM (quantum virtual machine) program (built from source) (`https://github.com/quil-lang/qvm`)
2. The pyquil python package (available through pip)
3. The QUILC (Quantum Instruction Langauage Compiler) program (built from source). (`https://github.com/quil-lang/quilc`) Extra requirements:
   (a) Standard UNIX build tools
   (b) SBCL (Steel Bank Common Lisp) (not SBCL 1.5.6) a common Lisp compiler.
   (c) QuickLisp (a common Lisp library manager)
   (d) ZeroMQ (Messaging library required by RPCQ)

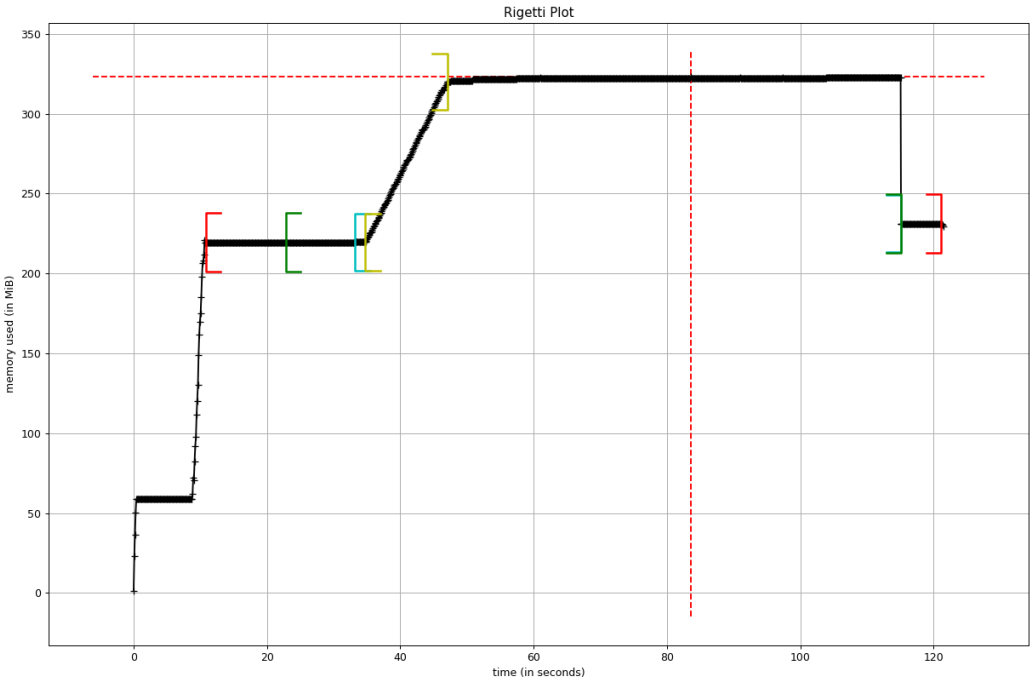


Figure 6.12: A plot of the memory usage across time using the Rigetti platform

Legend:

[ ] start 81.881s

[ ] compile_all 12.334s

[ ] runRigetti 92.319s

[ ] runMe 110.336s

The QVM and pyquil installation are simple, provided that QUILC has been properly installed. The pyquil package provides a user-friendly interface to the QVM/QUILC

The data captured in Figure 6.11 demonstrates that the Rigetti quantum simulator takes on average 1683.52±2.737 seconds and consumes 777.94±0.07 MiB. The data in Figure 6.11 ranges between 1660.59 and 1703.517 in execution time (difference of 42.92) and between 777.1 and 778.49 in memory

consumption (difference of 1.38).

The graph found in Figure 6.12 demonstrates the amount of memory used by a typical execution of the Rigetti platform over time. It has a series of useful sections marked to assist in the understanding of the information. The sections correspond with methods within the Python script:

- runMe - This method is responsible for reading the data set and then starting the relevant simulator.
- runRigetti - This method is responsible for preparing and executing all parts of the Rigetti script.
- start - This method is responsible for compiling each of the programs (see below) and executing them on the Rigetti platform.
- Compile_all - This method is responsible for compiling each of the programs found in the data set for execution with the Rigetti platform.

It can be seen that most of the memory growth stems from the compilation of Rigetti programs and then remains relatively consistent throughout the execution. There are also 2 red dashed lines (1 horizontal and 1 vertical) which mark the point where the memory consumption is at its peak. The system originally launches to a base consumption of roughly $219.5MiB$ before growing linearly throughout the compilation routine to a maximum of $322.93MiB$. Following the compilation routine the memory growth slows greatly. While the programs are executed, the memory wavers slightly while remaining largely consistent. Following the execution, the memory usage scales back to the original level as the Rigetti ecosystem is closed down.

6.2.3.3 Q#

Table 6.10: Packages and versions used to evaluate the Q# platform

Software	Version
Python	3.8.5
Qsharp	0.15.2103.133969
Qsharp Chemistry	0.15.2103.133969
Qsharp Core	0.15.2103.133969
Jupyter Core	4.7.1
Jupyter Packaging	0.7.12
Jupyter Server	1.5.1
JupyterLab	3.0.12
JupyterLab Pygments	0.1.2
JupyterLab Server	2.4.0

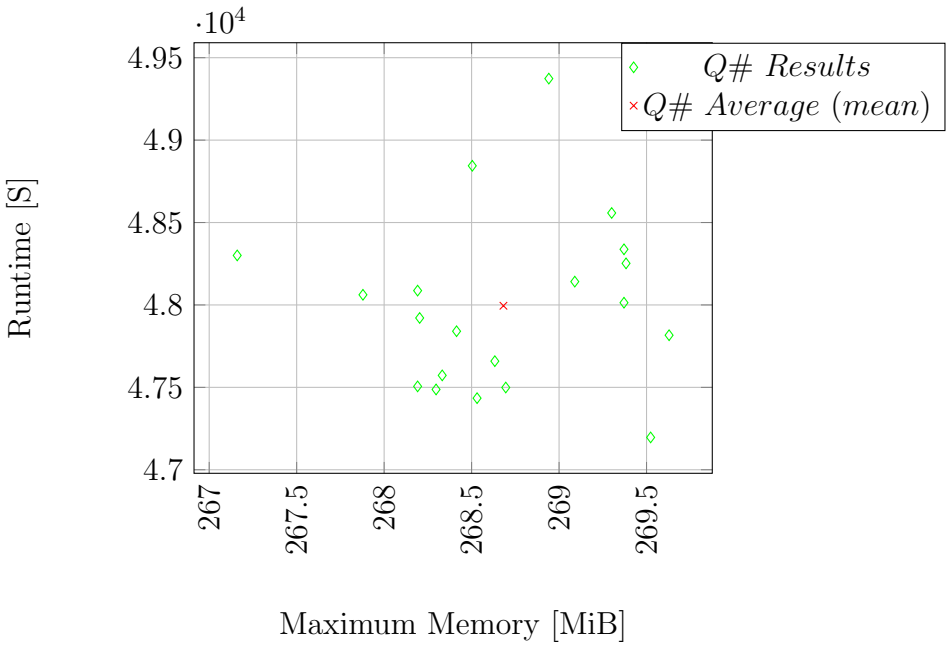


Figure 6.13: A scatterplot of the time and memory measurements recorded over 20 executions of the 100 program test using the Q# platform

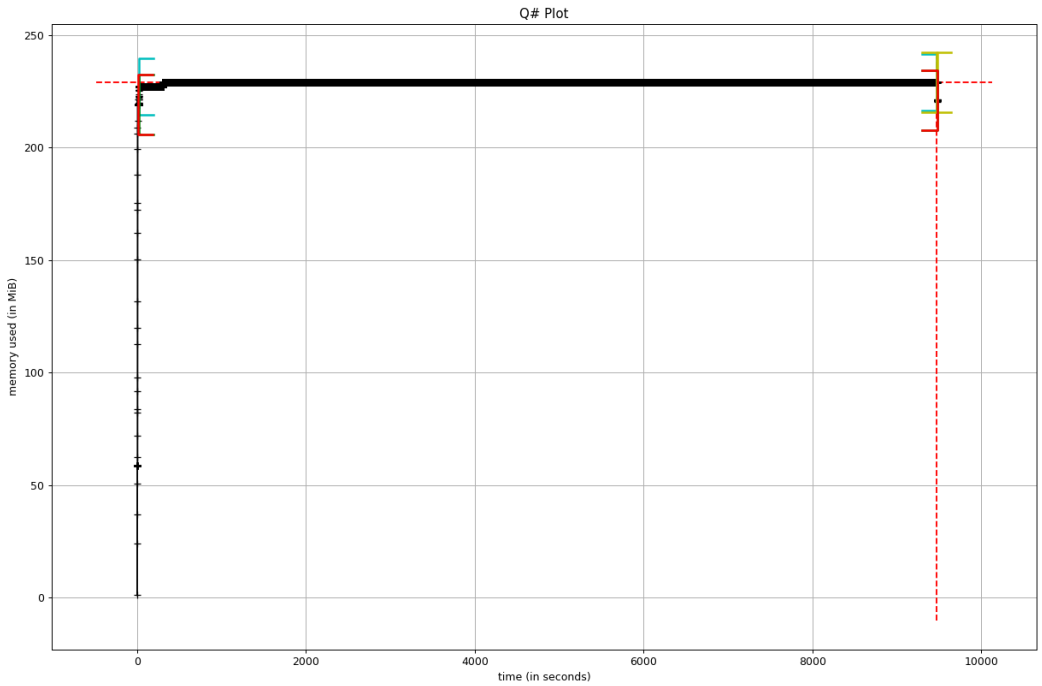


Figure 6.14: A plot of the memory usage across time using the Q# platform

Legend:

[ ] compileAll 9444.652s

[ ] fireAll 7.899s

[ ] runQsharp 9455.044s

[ ] runMe 9473.063s

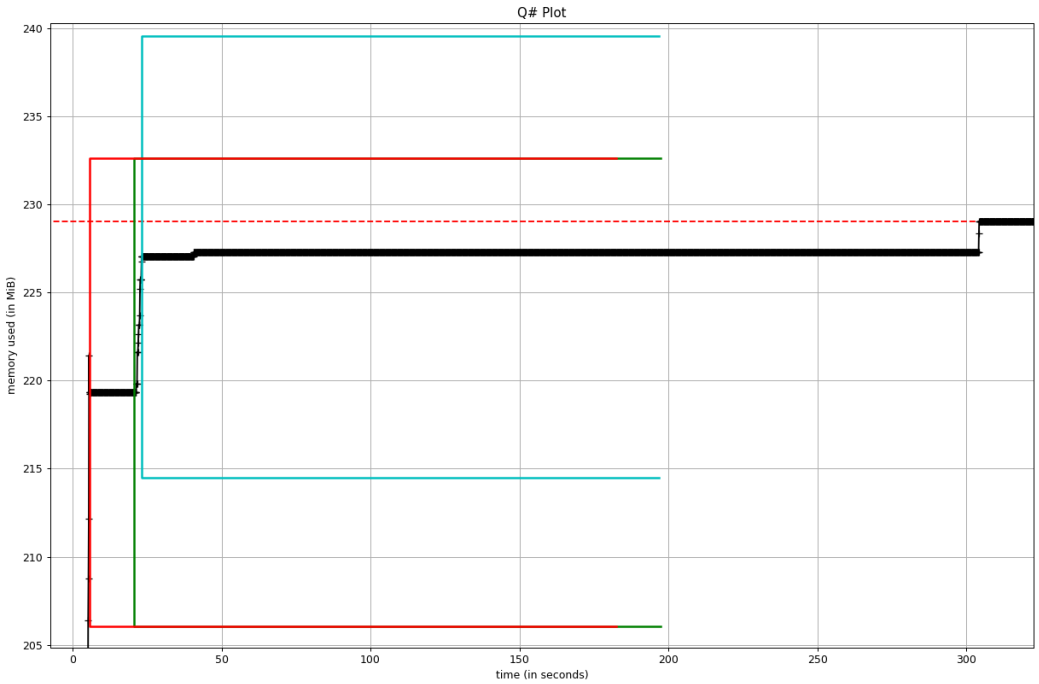


Figure 6.15: A plot of the memory usage across time using the Q# platform concentrated on the initial warm up stage.

Legend:

[ ] compileAll 9444.652s

[ ] fireAll 7.899s

[ ] runQsharp 9455.044s

[ ] runMe 9473.063s

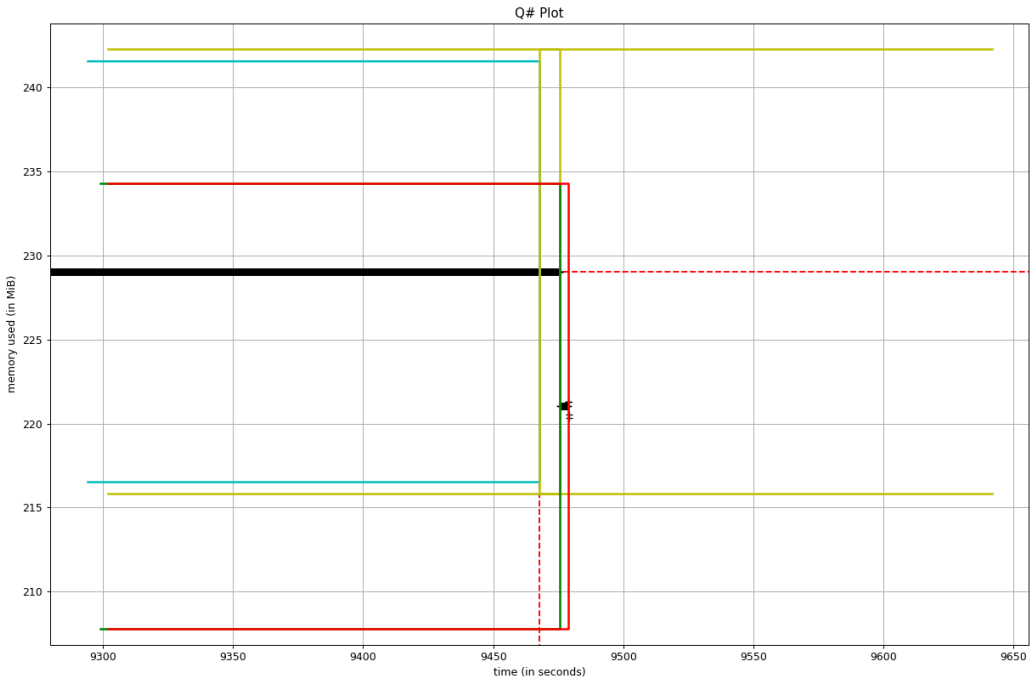


Figure 6.16: A plot of the memory usage across time using the Q# platform concentrated on the final conclusion stage

Legend:

[ ] compileAll 9444.652s

[ ] fireAll 7.899s

[ ] runQsharp 9455.044s

[ ] runMe 9473.063s

The data captured in Figure 6.13 demonstrates that the QSharp quantum simulator takes on average 47995.06 ± 118.06 seconds and consumes 268.68 ± 0.14 MiB. The data in Figure 6.13 ranges between 47196.298 and 49373.153 in execution time (difference of 2176.85) and between 267.16 and 269.63 in memory consumption (difference of 2.47).

The graph found in Figure 6.14 demonstrates the amount of memory

used by a typical execution of the Q# platform over time. The graph is visibly distorted because of the excessive time spent performing the test. Figures 6.15 and 6.16 have been included to demonstrate the start up and shut down components of Figure 6.14. Figures 6.14, 6.15 and 6.16 has a series of useful sections marked to assist in the understanding of the information. The sections correspond with methods within the python script:

- runMe - This method is responsible for reading the data set and then starting the relevant simulator.
- runQsharp4 - This method is responsible for preparing and executing all parts of the Q# script. (3 variations were tested to try and get the best performance from the Q# system).
- CompileAll - This method is responsible for compiling each of the programs found in the data set for execution with the Q# platform.
- fireAll - This method is responsible for executing them on the Q# platform.

It can be seen that most of the memory growth stems from the compilation of Q# programs and then remains relatively consistent throughout the execution (Figure 6.15). There are also 2 red dashed lines (1 horizontal and 1 vertical) which mark the point where the memory consumption is at its peak. The system originally launches to a base consumption of roughly $219.33MiB$ before growing linearly throughout the compilation routine. It should be noted that the time spent during the compilation is excessive and it is responsible for the time required to perform the test. The amount of time spent actually executing the program is 0.083% of the time taken to complete the test. Following the compilation routine the memory growth slows greatly. While the programs are executed, the memory wavers slightly while remaining largely consistent. Following the execution, the memory usage scales back to the original level as the Q# ecosystem is closed down. The amount of memory consumed during the test remains largely stagnant, holding at approximately $229.05MiB$ which indicates that the method of

compilation is not expensive in terms of memory but expensive in terms of time. This expense hints at either a sub-optimal compilation routine or the inclusion of some NP problems therefore delaying the compilation while it solves the computationally difficult (NP) problems.

6.2.3.4 GladeOS

Table 6.11: Packages and versions used to evaluate the GladeOS platform

Software	**Version**
Operating System	Ubuntu 20.04
Python	3.8.5
Requests	2.22.0
Requests NTLM	1.1.0
Requests Unixsocket	0.2.0
GladeOS	3.0
gcc	11.1
g++	11.1

The data captured in Figure 6.17 demonstrates that the GladeOS quantum simulator takes on average 29.89 ± 0.02 seconds and consumes 226.76 ± 0.12 MiB. The data in Figure 6.17 ranges between 30.0977 and 29.72 in execution time (difference of 0.377) and between 227.855 and 226.14 in memory consumption (difference of 1.71).

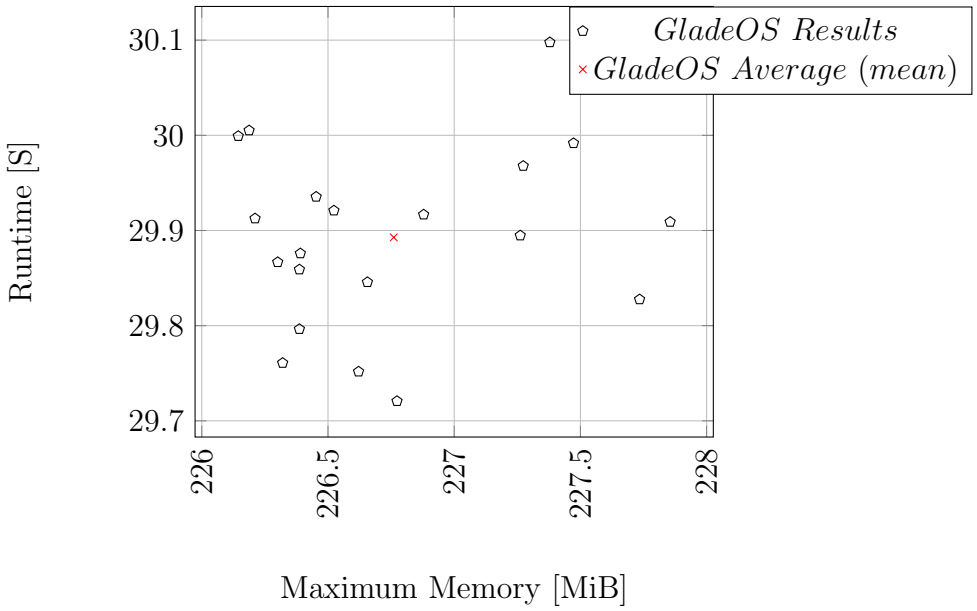


Figure 6.17: A scatterplot of the time and memory measurements recorded over 20 executions of the 100 program test using the GladeOS platform

The graph found in Figure 6.18 demonstrates the amount of memory used by a typical execution of the GladeOS platform over time. It has a series of useful sections marked to assist in the understanding of the information. The sections correspond with methods within the python script:

- runMe - This method is responsible for reading the data set and then starting the relevant simulator.
- runGlade - This method is responsible for preparing and executing all parts of the GladeOS script, which involves starting the external GladeOS server program with the specified arguments (pausing to ensure it has actually started) and then actually beginning the test.
- start - This method is responsible for compiling each of the programs (see below) and executing them on the GladeOS platform.

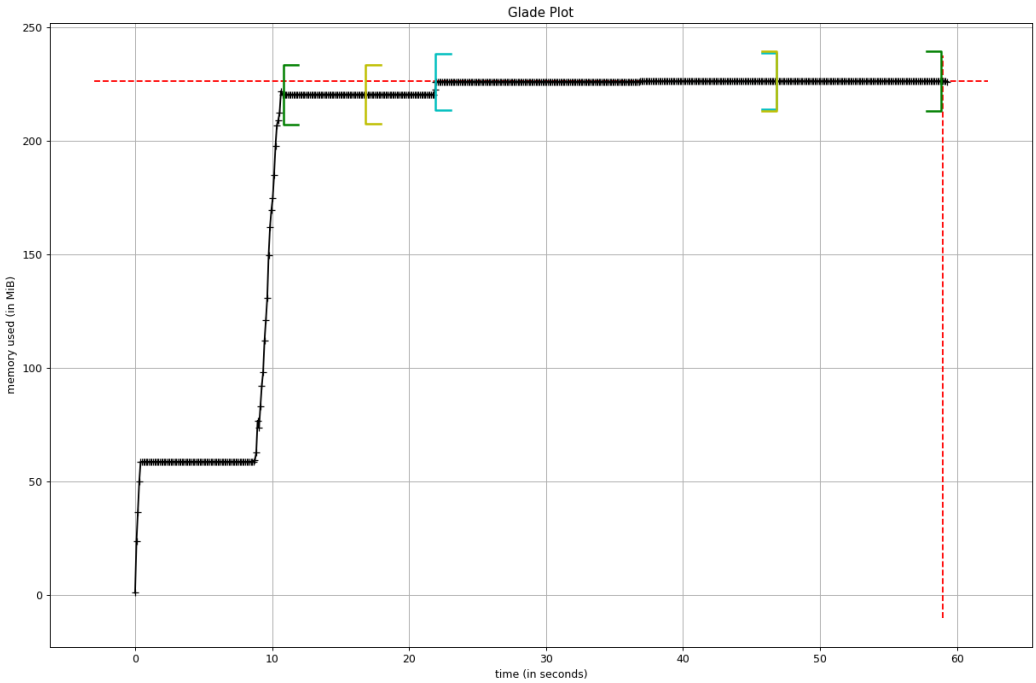


Figure 6.18: A plot of the memory usage across time using the GladeOS platform

Legend:

[ ] start 24.882s

[ ] runGlade 30.005s

[ ] runMe 48.026s

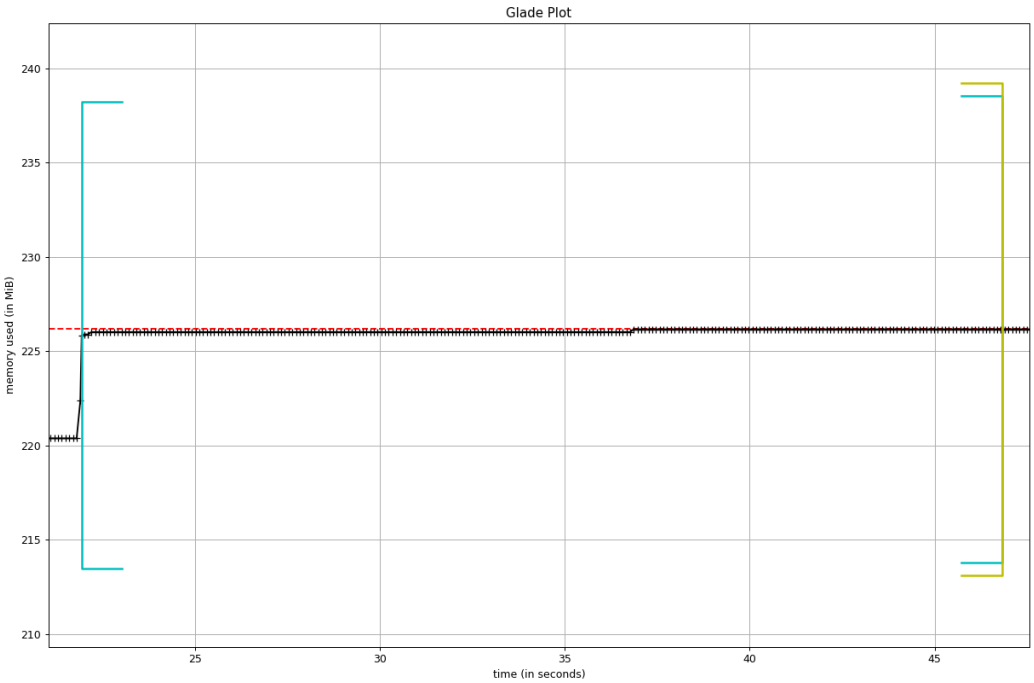


Figure 6.19: A concentrated view of the memory usage during the runtime of the GladeOS platform

Legend:

[ ] start 24.882s

[ ] runGlade 30.005s

[ ] runMe 48.026s

There are delays inbuilt into the testing script to ensure clear delineation between each component of the script. It can be seen that most of the memory growth stems from the compilation of GladeOS programs and then remains relatively consistent throughout the execution. There are also 2 red dashed lines (1 horizontal and 1 vertical) which mark the point where the memory consumption is at its peak. The system originally launches to a base consumption of roughly $220.355MiB$. While the programs are executed, the memory wavers slightly while remaining largely consistent with a

maximum of 226.19MiB. It should be noticed that the plot in Figure 6.18, varies (with regards to Memory Consumption) very slightly throughout the test and profiles a largely consistent usage (Figure 6.19). Because of how GladeOS was designed, it is very difficult to extrapolate compilation times compared to execution times and it shall therefore be treated as joint. Following the execution, the memory usage remains at the higher level as the GladeOS ecosystem is closed down.

Part of the problem with measuring GladeOS performance, is that all of the compilation is done on the GladeOS program and is only accessed through network connections. The code within GladeOS is so heavily optimised that the Python requests library cannot send the programs fast enough to actually occupy the GladeOS program.

The times found in Table 6.12 demonstrate the average speed of each operation, indicating that a program which features only 1000 Pauli-X gates would on average take 0.14 microseconds (μs) to execute. This speed means that the GladeOS system is having to pause and wait while the Python script can provide the next program. For a more precise test of GladeOS's capabilities please refer to Section 6.3.

Table 6.12: Logic gate operation times for the GladeOS system

Operation	**Time Taken [ns]**
Pauli-X	138.61
Pauli-Y	238.57
Pauli-Z	150.65
Phase	154.73
Half-Phase	325.21
Rotate-X	357.9
Rotate-Y	337.95
Rotate-Z	515.41
Free-Rotate	15619.4

6.2.3.5 Combined results

Table 6.13: All versions

Software	**Version**
Operating System	Ubuntu 20.04
RAM	16GB
CPU	Intel i7-6770HQ
Python	3.8.5
Qiskit	0.23.5
Qiskit Aer	0.7.4
Qiskit Aqua	0.8.2
Qiskit IBMq Provider	0.11.1
Qiskit Ignis	0.5.2
Qiskit Terra	0.16.4
QVM	1.17.1
QUILC	1.23.0
PyQuil	2.28.0
SBCL	2.0.1.debian
ZeroMQ	4.3.2-2ubuntu1 amd64
Quicklisp	2021-02-13
Magicl	0.9.1
RPCQ	3.8.0
Qsharp	0.15.2103.133969
Qsharp Chemistry	0.15.2103.133969
Qsharp Core	0.15.2103.133969
Jupyter Core	4.7.1
Jupyter Packaging	0.7.12
Jupyter Server	1.5.1
JupyterLab	3.0.12
JupyterLab Pygments	0.1.2
JupyterLab Server	2.4.0
Requests	2.22.0
Requests NTLM	1.1.0
Requests Unixsocket	0.2.0
GladeOS	3.0
gcc	11.1
g++	11.1

Table 6.14: Average Results

Simulator	**Time Taken** [s] $\mu \pm \sigma_{\bar{X}}$	**Maximum Memory** [MiB] $\mu \pm \sigma_{\bar{X}}$
GladeOS (1T)	29.89 ± 0.021	226.76 ± 0.12
Q#	47995.065 ± 118.0585	268.682 ± 0.14
Qiskit	3551.12 ± 11.86	884.15 ± 0.924
Rigetti	1683.52 ± 2.737	777.94 ± 0.07

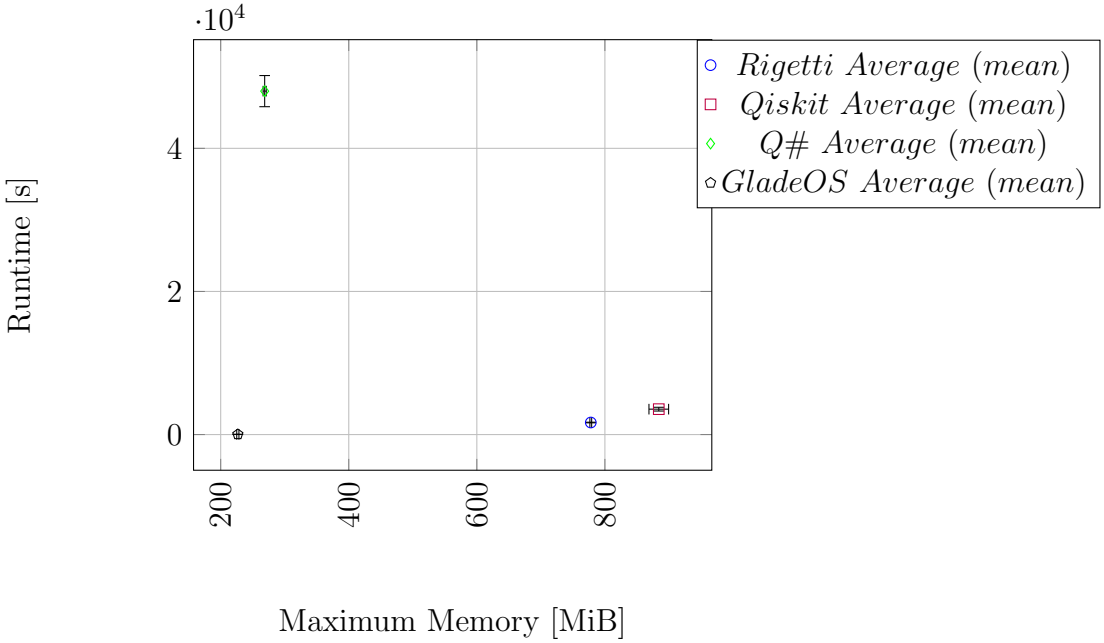


Figure 6.20: Scatterplot Results Global (20 executions of 100 Program test). Range of values is indicated with the error bars.

Comparing the results from the varying simulators reveals interesting results. Simulators either follow one of two trends, either relatively large consumption of memory but minimal execution time or small memory footprint with a large execution time. Q# (microsoft) exemplifies the second trend, taking between 13 and 14 hours to complete the random test while maintaining the second smallest memory footprint (approximately $268.68MiB$). Qiskit and

Rigetti demonstrate the first trend, taking 1 hour with $884.15MiB$ and 30 minutes with $777.94MiB$ respectively.

The results from the varying simulators indicates that GladeOS outperforms the 3 commercial simulators by varying margins. It is believed that this is due to a number of reasons including the specialised matrix approach (see Chapter 5), a smaller amount of support libraries and a small core development. The simulators tend to either implement the entire state array as a single large matrix or employ alternative means to perform their calculations. The seperation of the state array into smaller matrices is an approach that has not been seen outside of GladeOS. Also worth noting is that each of the comparison simulators go beyond the standard requirement of simulating the state array by employing quantum programming libraries or hardware emulation to attempt to reach a higher level of accuracy or to benefit the end user. Lastly, each of the comparison simulators has been developed by a large and ever changing development team. They are either open-sourced (Qiskit) or the product of a department within Microsoft or Rigetti and therefore suffer from the inevitable penalties of ever changing design and implementation of new research. GladeOS by comparison was developed in a small stretch of time with a specialised and consistent design that includes numerous optimisations like using pointer arithmetic and regular expressions in order to reduce the memory usage and processing time.

6.3 GladeOS Specific Tests

As alluded to during the multiple simulator tests, the performance of GladeOS was an upper bound which included multiple internal waits. Based on this, more precise tests were desired, designed and developed. Because the other simulators tested in Section 6.2.3 were so far apart from GladeOS results they have been ignored for these tests, otherwise the GladeOS results would have been compared to the nearest neighbour simulator.

6.3.1 Known Limitations

The only times that GladeOS fails is when the program queue grows faster than the program processes them or the system requires too much memory to calculate and exceeds the provided RAM. In these case the program will eventually ask for more memory than the operating system can provide, forcing the operating system to kill the program.

To counter the system requesting too much memory there exists a hard limit as to the number of qubits that can be entangled in a single group. This unfortunately is a known issue and is readily acknowledged by the qubit limits of other simulators [24]–[26]. Programs may have multiple entangled groups, but if any group exceeds the maximum size then the system will be reaped by the host OS. As the maximum size is determined by the amount of RAM that the host computer has at that specific point in time, it is not possible to restrict submissions based on this limit. Further, because the system measures qubits individually except for the entangled groups, programs which require significantly more than the entanglement limit can be computed without crashing.

To counter the program queue problem the following variables have been defined:

P The time taken to parse a single gate

N The number of gates in the program (averaged over the file set)

E The time taken to process a single gate

H The amount of execution threads provided to the program.

G The number of graphs in the file set

S_c The cost of storing a single graph.

T The amount of time that has passed.

On average it takes approximately $3 - 6ns$ to parse a single qubit gate and approximately $6 - 9ns$ to parse a control gate. Meanwhile the times to process a gate can be found in Table 6.12 with a maximum of approximately $1570ns$. Therefore given that E is significantly larger than P it is a clear given that $EN > PN$.

$$f(T) = min\left(\left\lfloor \frac{T}{PN} \right\rfloor, GPN\right) \tag{6.2}$$

$$g(T) = min\left(\left\lfloor \frac{TH}{EN} \right\rfloor, GEN\right) \tag{6.3}$$

$$Q = (f(t) - g(t))S_c \tag{6.4}$$

Equation 6.2 calculates the amount of files parsed, while Equation 6.3 calculates the amount of files completed. Combining both equations into Equation 6.4 yields the approximate size of the queue (Q) at at that point in time (T).

Equations 6.2 and 6.3 where devised by calculating the amount of time it takes to parse and execute a gate respectively. That value multiplied by the number of gates returns the amount of time required to parse (or execute) that gate. That value is the divisor of T to allow the value to plotted over time, as T grows the equations grow in proportion to the PN and EN. The Floor operations where included because parsed (or executed) gates are counted as discrete values instead of continuous, this means that until T reaches P $\frac{T}{PN} = 0$ and once T reaches P then $\frac{T}{PN} = 1$. Unfortunately, T continues to grow infinitely so to include a cap on the growth the minimum function has been used with the GPN or GEN acting as the maximum allowable value. If T grows beyond that value then the function will always return GPN or GEN as they are no less than the left hand side. These

concepts were combined in Equation 6.4 to calculate the size of the queue by subtracting the amount executed from the amount currently parsed and multiplying the result by the storage cost (S_c).

Each program that is parsed into the system is composed of (S_c):

- A vector of Instructions
- A $uint16\_t$ instruction counter (2 bytes)
- a 'State array map' - vector of Pair(unordered set ($uint16\_t$), vector of complex floats)
- A hash of the controller file (4 bytes)

The size of the state array is determined by the number of qubits allocated to the controller. Each entangled group of qubits will have a pair within this outer vector, for n entangled qubits, the unordered set will have n entries and the inner vector will hold 2^n values. Therefore overall the state array costs $n * 2 + 2^n * 4$ bytes (where 2 and 4 are the relevant data sizes).

The instruction counter is a fixed value of 2 bytes while the instruction size is significantly more complicated. This is due to the instructions consisting of:

- Instruction type enum (4 bytes)
- Qubit identification number ($uint16\_t$, 2 bytes)
- Vector of control bits ($2 * C$ bytes, where C is the number of entries)
- Vector of angle rotations (maximum $3 * 4$ bytes)

If we assume that every instruction is a controlled rotate (utilising all 3 angles) with a single control bit, then each instruction has a maximum size of $4 + 2 + 2 + 12 = 20$ bytes. Which results in a size of $20 * I$ bytes (where I is the number of Instructions). Every controller file that is processed by the system grows the system by $S_c = (20 * I) + (n * 2) + (2^{n+2}) + (4)$ bytes. Note:

$N \approx I - 1$ because of the final "END" instruction.

Following these formulas, provided that the Equation 6.5 does not exceed the amount of available memory then the system will sustain the load.

$$mem = \left(min\left(\left\lfloor \frac{T}{PN} \right\rfloor, GPN \right) - min\left(\left\lfloor \frac{TH}{EN} \right\rfloor, GEN \right) \right) * ((20I)+(2n)+(2^{n+2})+(4)) \tag{6.5}$$

If the following assumptions are made:

- $P = 10$ nano seconds
- $N = 1000$ gates
- $E = 1500$ nano seconds
- $H = 1$
- $G = 500$
- $S_c = (20I) + (2n) + (2^{n+2}) + (4)$ where:
  - $I = 1001$
  - $N = 10$

then Figure 6.21 represents the memory consumption of the system and the upper limit. The blue line represents the trend described by Equation 6.5 and the red line equates to 5Mb of memory.

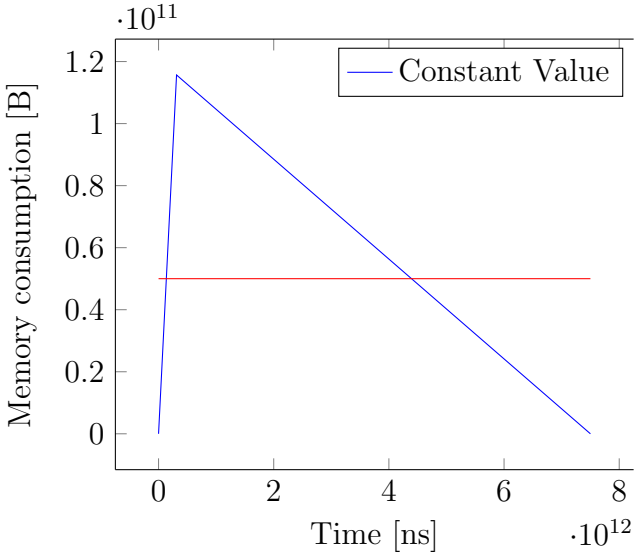


Figure 6.21: Memory Usage Estimation

6.3.2 Expected Output Tests

The following tests are designed to test the accuracy of the systems programs. Each test is composed of:

- The program file.
- A histogram showing the expected results (as calculated), against the execution results (averaged over 10,000 executions).

These tests are included to demonstrate the correctness and accuracy of the developed system. While it would be possible (with reworking of code) to retrieve the program state from the simulator, it would be ignoring the measurement operation and therefore result in an incomplete evaluation. The results collected are retrieved as a 3-bit string, thereby producing values between 000 (0) and 111 (7) inclusive.

In order to properly evaluate a measured probability distribution against an expected probability distribution a 'goodness of fit' test is required. This test compares the measured distribution to the expected distribution and returns a statistic that demonstrates how probable it is that the 2 distributions

match. There are multiple 'goodness of fit' tests available but for this research the Fisher's Exact Test and Chi Squared tests were considered as they work with any expected distribution instead of requiring a normal distribution for the expected distribution. Overall the Chi Squared test was chosen due to issues with scaling the Fishers Exact test to larger distributions. The Chi Squared formula can be seen in Equation 6.6 where O_i is the observed value and E_i is the expected value for that group (state) in the data.

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i} \tag{6.6}$$

Where there is more than 1 correct (valid) answer a χ^2 test (Equation 6.6) [121] has been utilised to determine the goodness of fit between the calculated and the simulated results. The χ^2 test was chosen because of its ability to scale to varying sizes and number of elements as opposed to other tests like Fischers Exact Test which is limited in the data sizes it supports. The problem with the χ^2 test is that it cannot handle cells with 0's in them, the following methods were explored to deal with this limitation:

1. Find an alternative test
2. Jitter the results by 0.000000001.
3. Ignore the 0's entirely

The first option is to source an alternative test, one that can handle the 0 cells. The only goodness of fit tests that could handle 0 cells were very specifically limited to set sizes (typically a 2x2 table) which would not suit the data presented below. The second option was to jitter the results by a significantly small value. This value would remove cells from holding 0's while also minimising the impact on the calculations. Ultimately this approach will work, however it introduces noise into the data leading to test results which are inaccurate (to a small degree). The final option considered was to ignore the 0 cells. This option was proposed due to the cells which store the 0 values having a probability of appearing in the output equal to 0% (read impossible). Therefore if the values appeared then the goodness of

fit test was irrelevant as the simulator would be clearly producing incorrect values. Therefore this option was chosen to handle the 0 columns for the purposes of the χ^2 test.

For the Chi-square goodness-of-fit tests below each program was executed 10000 times to allow the distribution to regress to the mean and allow for an accurate comparison. It is hypothesised for each test that the distributions will be considered consistent. A p-value less than 0.05 will indicate statistical significance and therefore infer that the generated results are not consistent with the calculated results which means the quantum gate implementations are flawed.

6.3.2.1 Pauli X Gate

This program is included to demonstrate the application of the Not (Pauli-X) gate and the related accuracy.

```
1  X(0)
2
```

Algorithm 6.6: Pauli-X Test Program

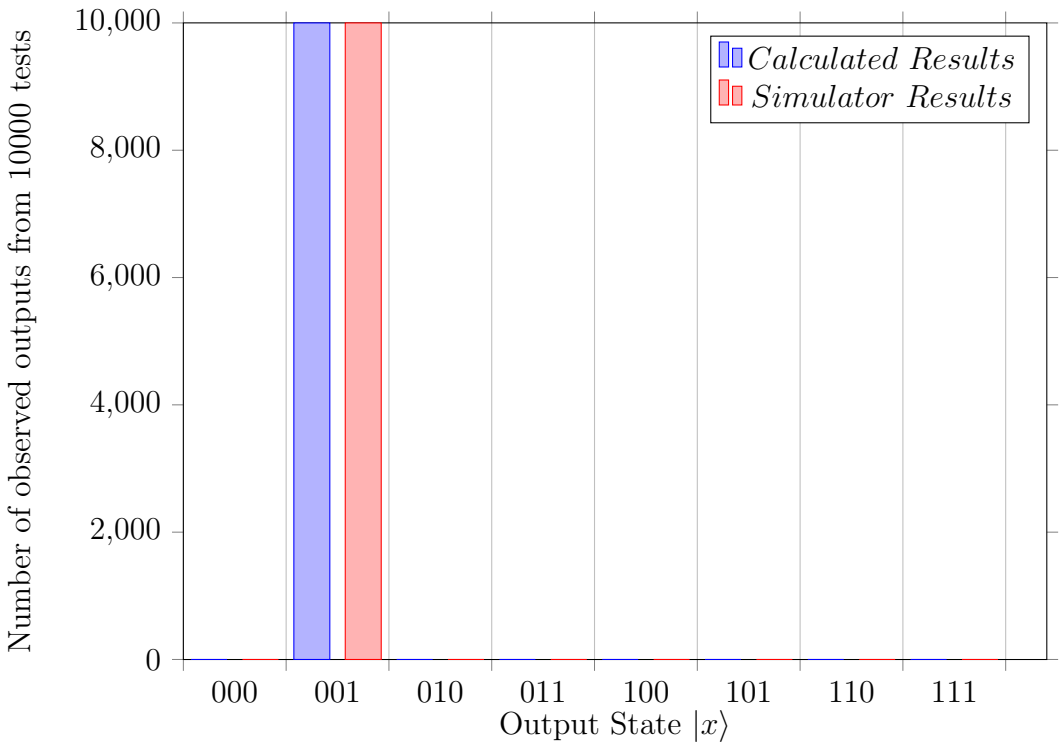


Figure 6.22: Comparitive histogram of the expected (calculated) results and those generated from executing the Pauli-X test program (Algorithm 6.6) in the GladeOS system

6.3.2.2 Pauli Y Gate

This program is included to demonstrate the application of the Pauli-Y gate and the related accuracy.

```
1 Y(0)
2
```

Algorithm 6.7: Pauli-Y Test Program

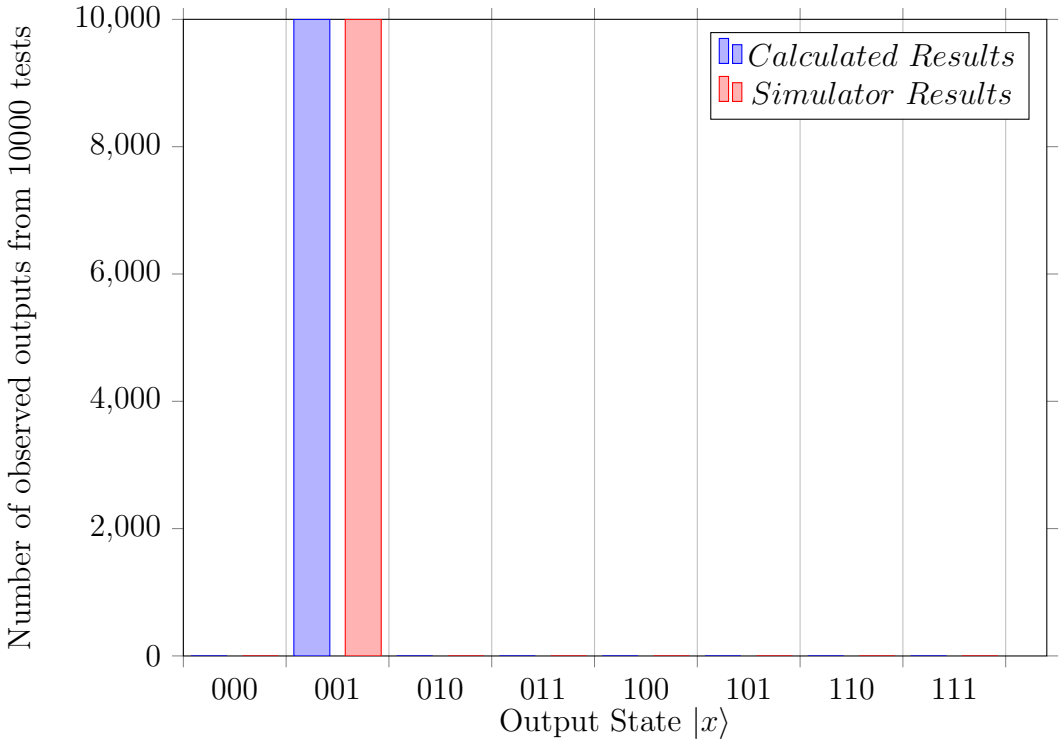


Figure 6.23: Comparative histogram of the expected (calculated) results and those generated from executing the Pauli-Y test program (Algorithm 6.7) in the GladeOS system

6.3.2.3 Pauli Z Gate

This program is included to demonstrate the application of the Pauli-Z gate and the related accuracy.

```
1  H(0)
2  Z(0)
3
```

Algorithm 6.8: Pauli-Z Test Program

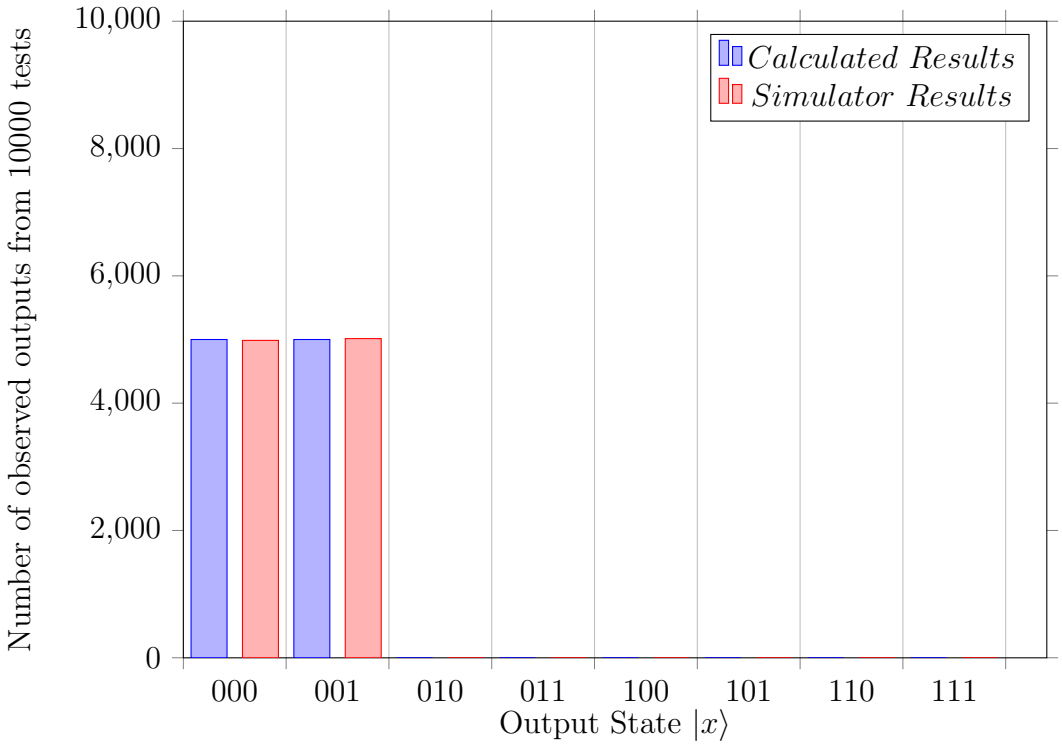


Figure 6.24: Comparative histogram of the expected (calculated) results and those generated from executing the Pauli-Z test program (Algorithm 6.8) in the GladeOS system

Utilising a Chi Square test with the measured and expected results, the Chi Square statistic is 0.0784, with a corresponding pvalue of 0.78 using 1 degree of freedom. Based on this p-value, there is no statistical significant difference and the system has a high probability of being accurate.

6.3.2.4 Hadamard Gate

This program is included to demonstrate the application of the Hadamard gate and the related accuracy.

```
1  H(0)
2
```

Algorithm 6.9: Hadamard Test Program

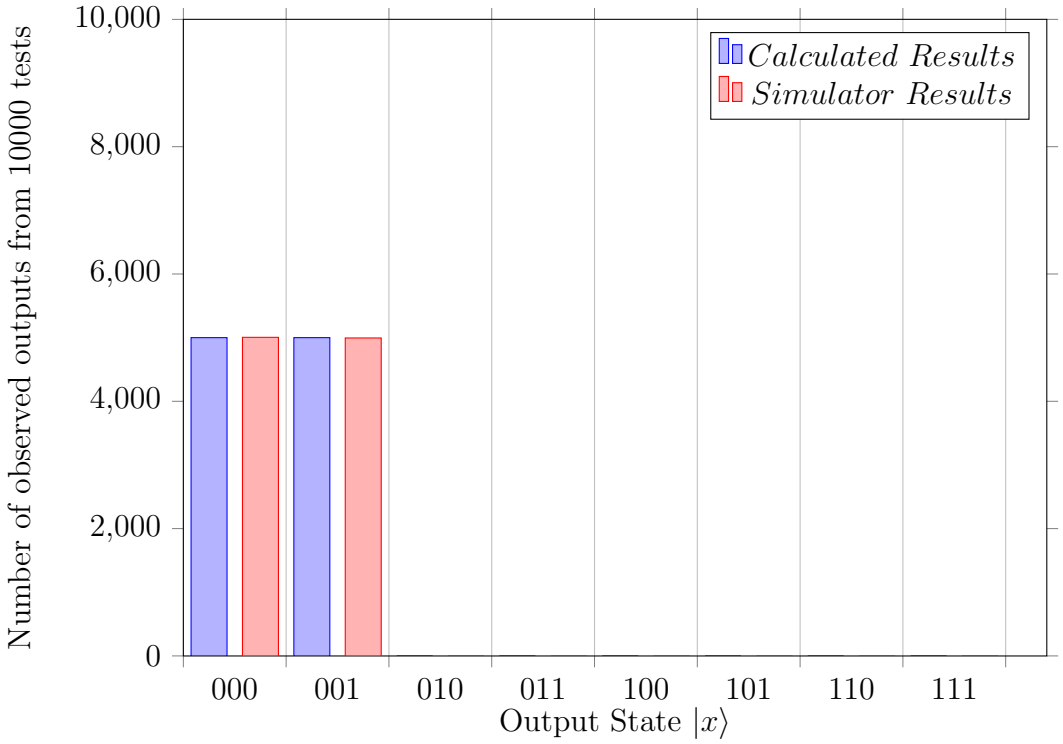


Figure 6.25: Comparative histogram of the expected (calculated) results and those generated from executing the Hadamard test program (Algorithm 6.9) in the GladeOS system

Utilising a Chi Square test with the measured and expected results, the Chi Square statistic is 0.01, with a corresponding p-value of 0.92 using 1 degree of freedom. Based on this p-value, the system is not statsitically significantly different from expected and so has a strong probability of being accurate.

6.3.2.5 Hadamard Gate 2

This program is included to demonstrate the application of the hadamard gate and the related accuracy.

```
1  H(0)
2  H(1)
3
```

Algorithm 6.10: Dual Hadamard Test Program

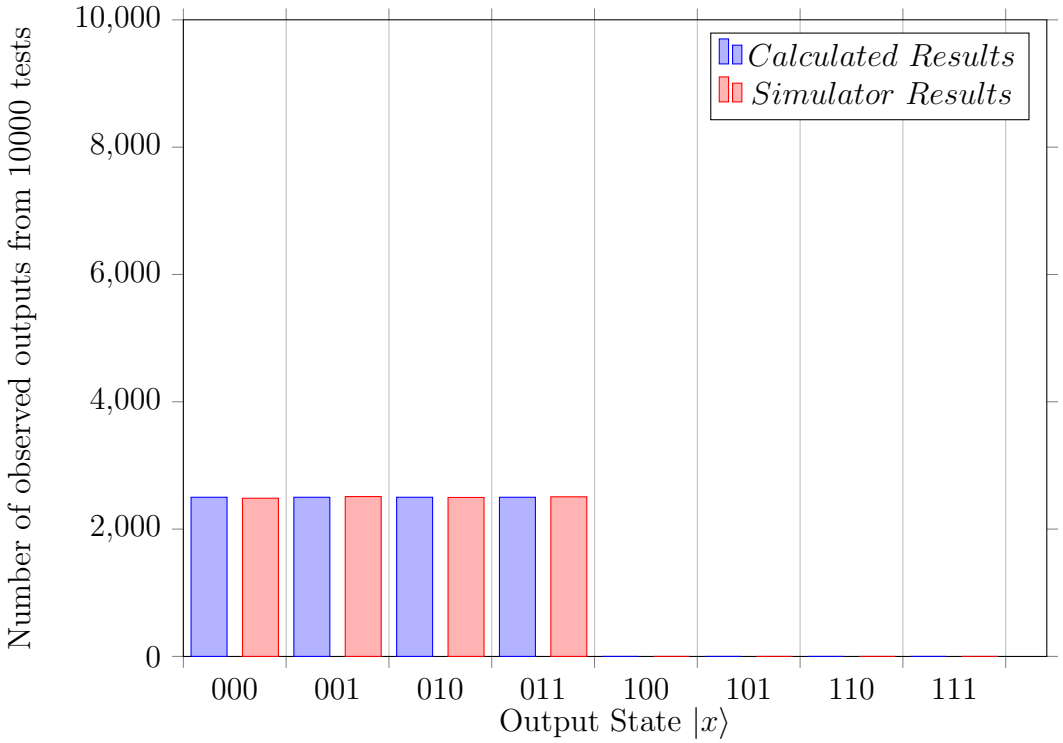


Figure 6.26: Comparative histogram of the expected (calculated) results and those generated from executing the Dual Hadamard test program (Algorithm 6.10) in the GladeOS system

Utilising a Chi Square test with the measured and expected results, the Chi Square statistic is 0.01616, with a corresponding pvalue of 0.98 using 3 degree of freedom. Based on this p-value, the system is not statistically significantly different from expected and so has a strong probability of being accurate.

6.3.2.6 Phase Gate

This program is included to demonstrate the application of the Phase gate and the related accuracy.

```
1  H(0)
2  S(0)
3
```

Algorithm 6.11: Phase Test Program

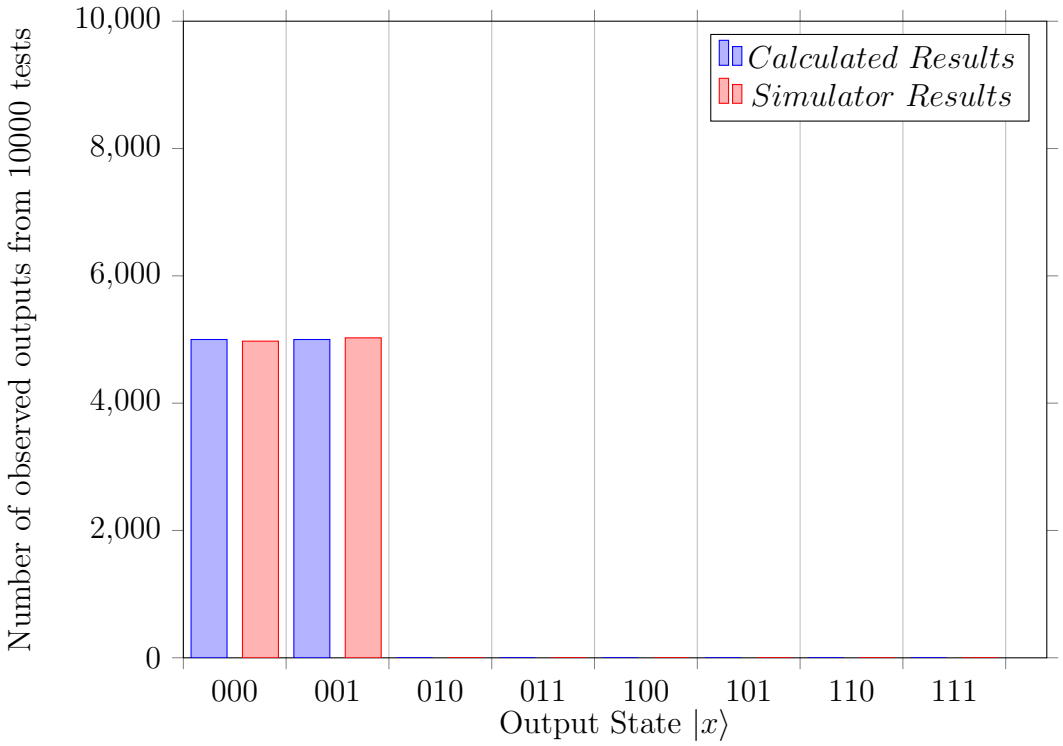


Figure 6.27: Comparative histogram of the expected (calculated) results and those generated from executing the Phase test program (Algorithm 6.11) in the GladeOS system

Utilising a Chi Square test with the measured and expected results, the Chi Square statistic is 0.27, with a corresponding pvalue of 0.6 using 1 degree of freedom. Based on this p-value, there is no statistical significant difference and the system has a high probability of being accurate.

6.3.2.7 $\frac{\pi}{8}$ Gate

This program is included to demonstrate the application of the $\frac{\pi}{8}$ gate and the related accuracy.

```
1  H(0)
2  T(0)
3
```

Algorithm 6.12: $\frac{\pi}{8}$ Test Program

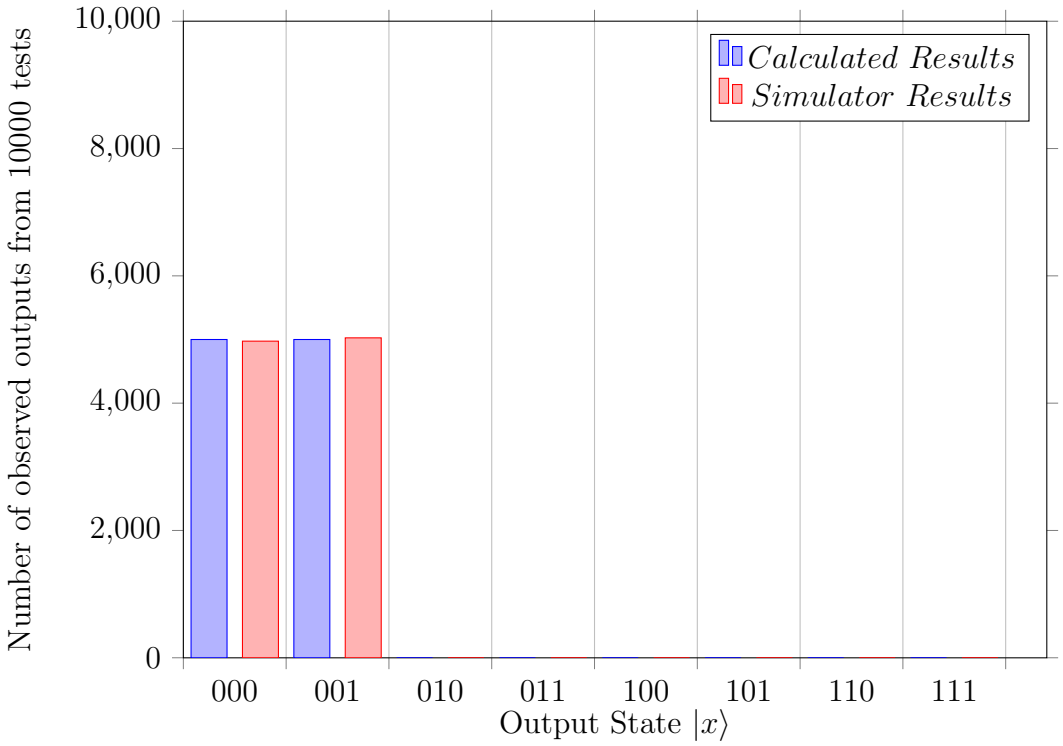


Figure 6.28: Comparative histogram of the expected (calculated) results and those generated from executing the $\frac{\pi}{8}$ test program (Algorithm 6.12) in the GladeOS system

Utilising a Chi Square test with the measured and expected results, the Chi Square statistic is 0.27, with a corresponding pvalue of 0.6 using 1 degree of freedom. Based on this p-value, there is no statistical significant difference and the system has a high probability of being accurate.

6.3.2.8 Rotate X Gate

This program is included to demonstrate the application of the Rotate-X gate and the related accuracy.

```
1 RX([90],0)
2
```

Algorithm 6.13: Rotate-X Test Program

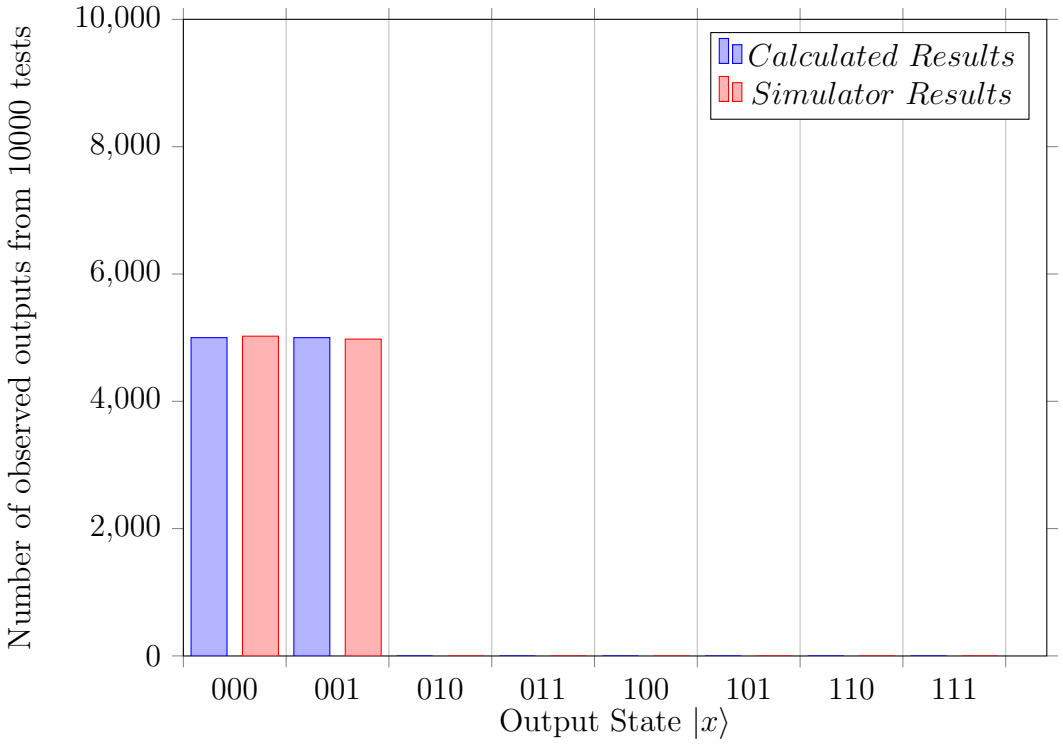


Figure 6.29: Comparative histogram of the expected (calculated) results and those generated from executing the Rotate-X test program (Algorithm 6.13) in the GladeOS system

Utilising a Chi Square test with the measured and expected results, the Chi Square statistic is 0.2116, with a corresponding pvalue of 0.65 using 1 degree of freedom. Based on this p-value, there is no statistical significant difference and the system has a high probability of being accurate.

6.3.2.9 Rotate Y Gate

This program is included to demonstrate the application of the Rotate-Y gate and the related accuracy.

```
1 RY([90],0)
2
```

Algorithm 6.14: Rotate-Y Test Program

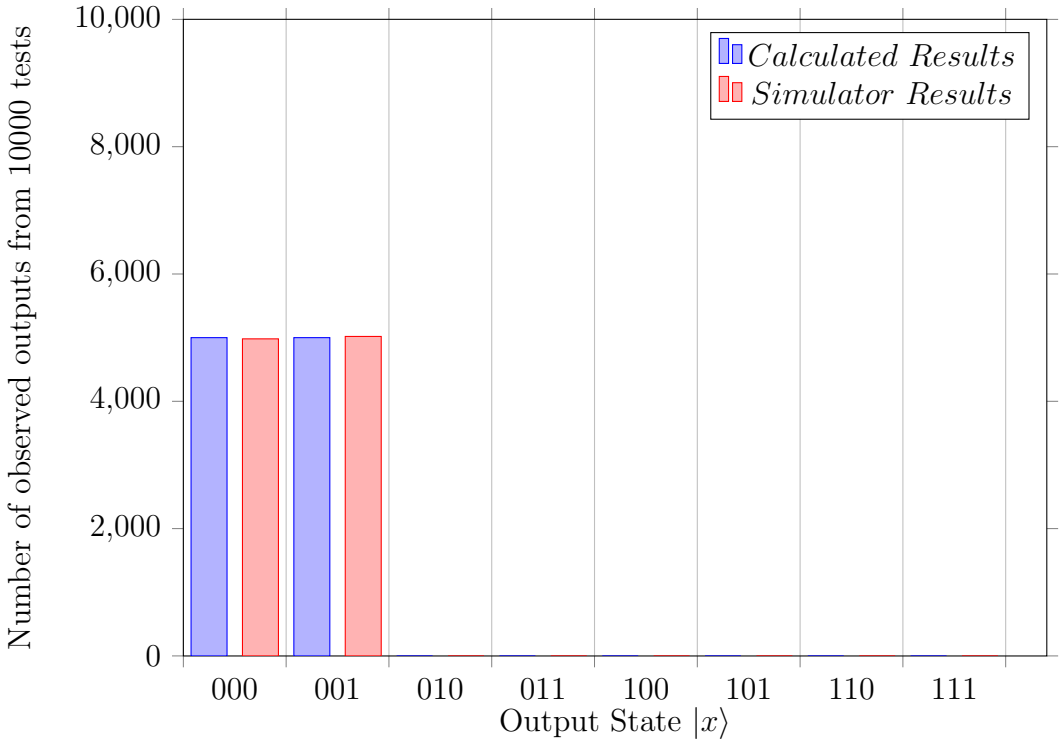


Figure 6.30: Comparative histogram of the expected (calculated) results and those generated from executing the Rotate-Y test program (Algorithm 6.14) in the GladeOS system

Utilising a Chi Square test with the measured and expected results, the Chi Square statistic is 0.1444, with a corresponding pvalue of 0.7039454151516744 using 1 degree of freedom. Based on this P-value there is no indication of a statistically significant difference.

6.3.2.10 Rotate Z Gate

This program is included to demonstrate the application of the Rotate-Z gate and the related accuracy.

```
1 RZ([90],0)
2
```

Algorithm 6.15: Rotate-Z Test Program

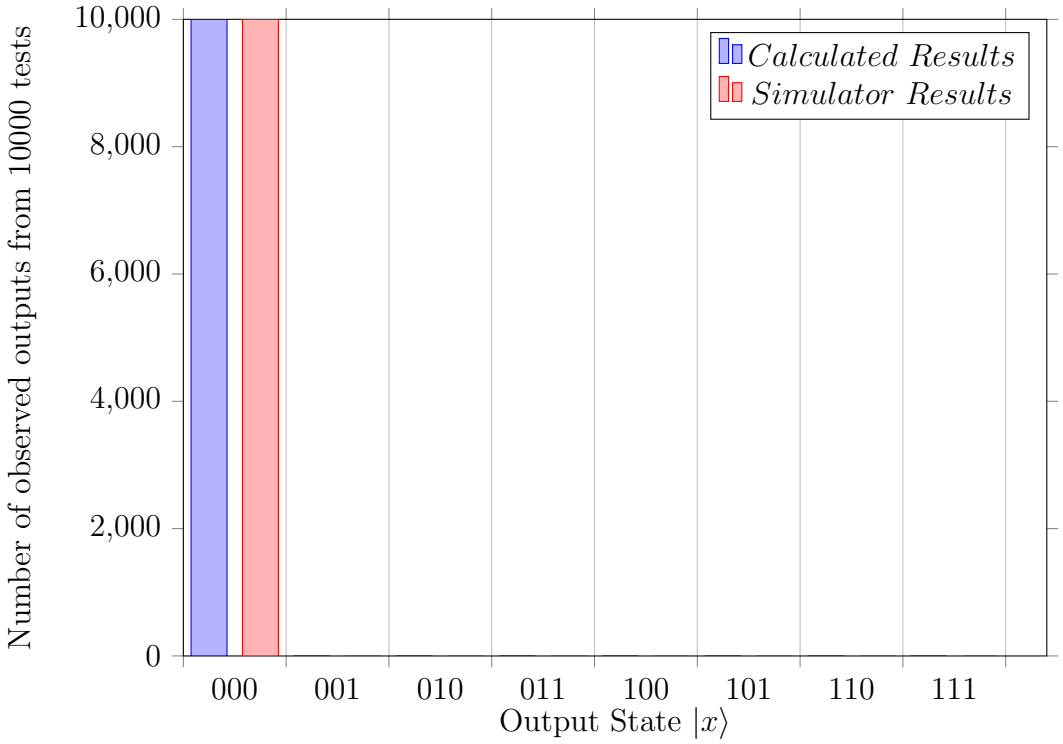


Figure 6.31: Comparative histogram of the expected (calculated) results and those generated from executing the Rotate-Z test program (Algorithm 6.15) in the GladeOS system

6.3.2.11 Free Rotate Gate

This program is included to demonstrate the application of the Free Rotate gate and the related accuracy.

```
1 R([90,90,90],0)
2
```

Algorithm 6.16: Free Rotate Test Program

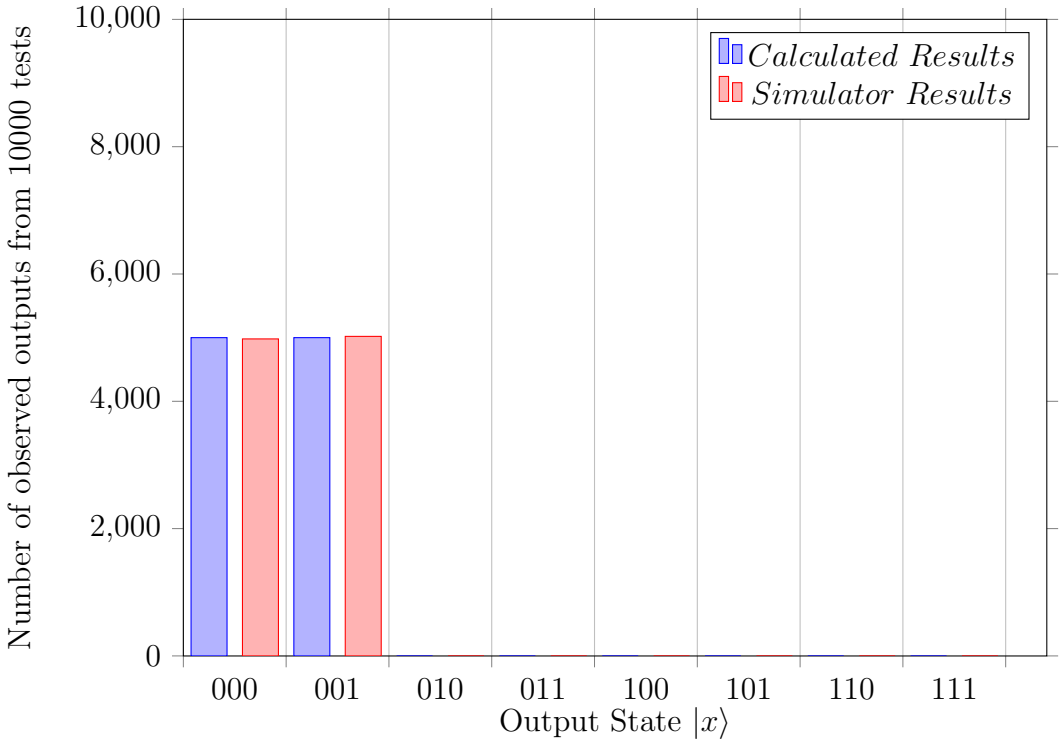


Figure 6.32: Comparative histogram of the expected (calculated) results and those generated from executing the Free Rotate test program (Algorithm 6.16) in the GladeOS system

Utilising a Chi Square test with the measured and expected results, the Chi Square statistic is 0.16, with a corresponding pvalue of 0.69 using 1 degree of freedom. Based on this p-value, there is no statistical significant difference and the system has a high probability of being accurate.

6.3.2.12 Basic Entanglement

This program is included to demonstrate the application of the Basic entanglement setup and the related accuracy.

```
1  H(0)
2  C(0,X(1))
3
```

Algorithm 6.17: Basic Entanglement Test Program

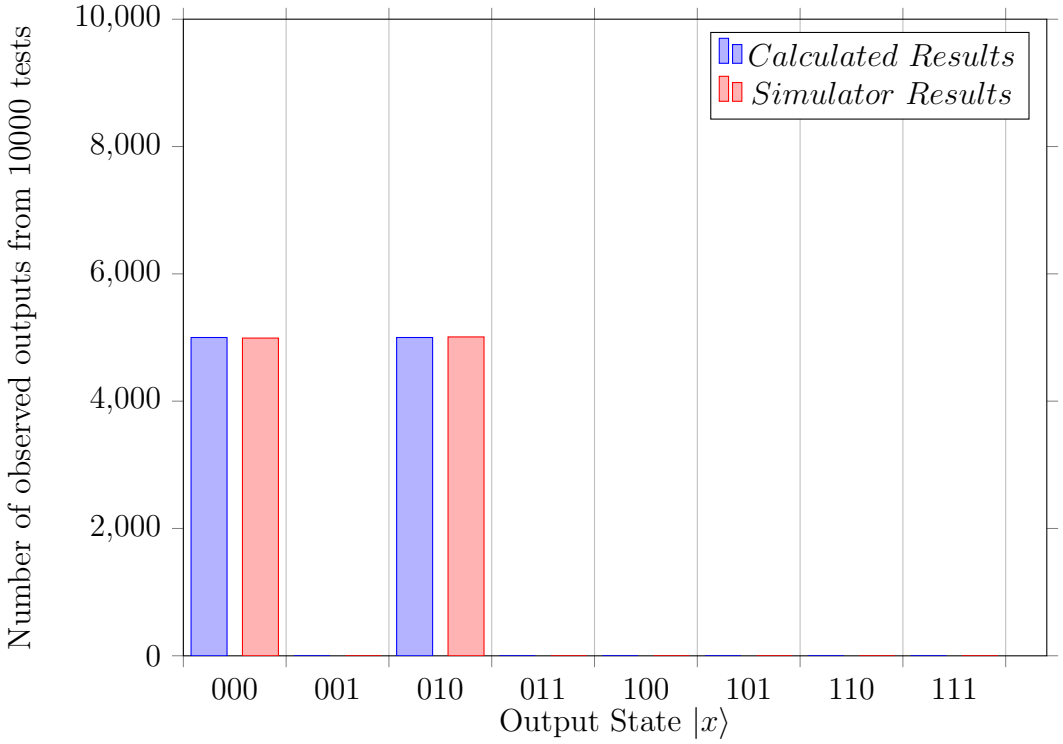


Figure 6.33: Comparative histogram of the expected (calculated) results and those generated from executing the Basic Entanglement test program (Algorithm 6.17) in the GladeOS system

Utilising a Chi Square test with the measured and expected results, the Chi Square statistic is 0.0324, with a corresponding pvalue of 0.86 using 1 degree of freedom. Based on this p-value, there is no statistical significant difference and the system has a strong probability of being accurate.

6.3.2.13 W Entanglement state

This program is included to demonstrate the application of the W Entanglement gate and the related accuracy.

```
1  H(0)
2  C(0,X(1))
3  C(0,X(2))
4
```

Algorithm 6.18: W Entanglement Test Program

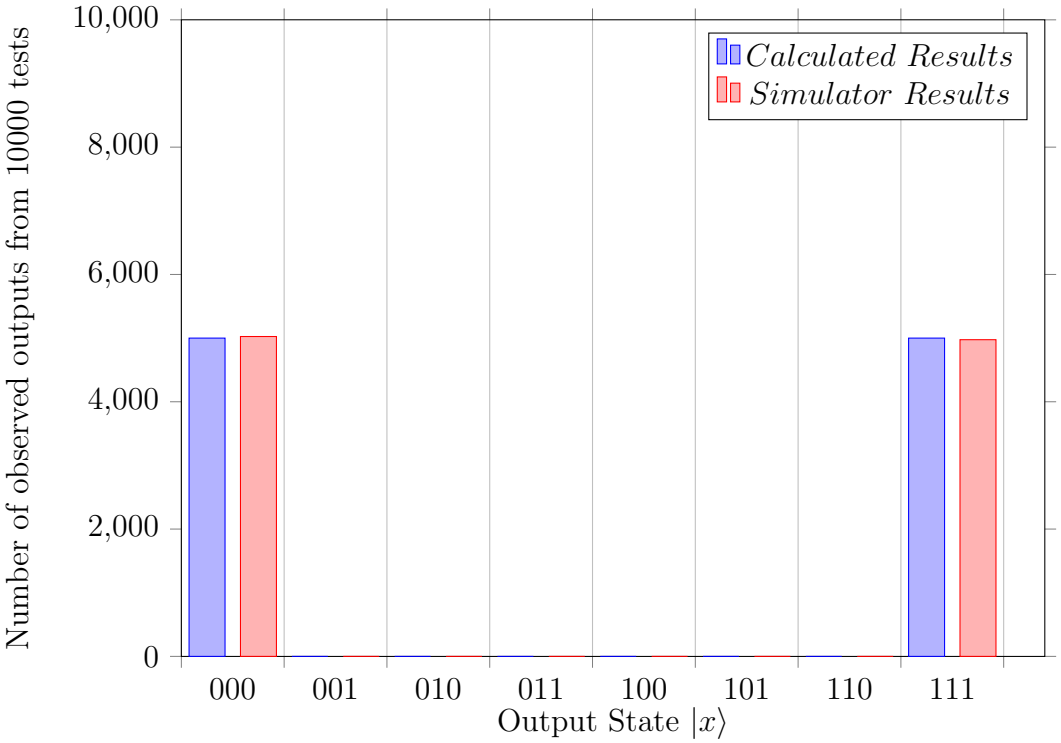


Figure 6.34: Comparative histogram of the expected (calculated) results and those generated from executing the W Entanglement test program (Algorithm 6.18) in the GladeOS system

Utilising a Chi Square test with the measured and expected results, the Chi Square statistic is 0.25, with a corresponding pvalue of 0.62. Based on this p-value, the system is not statistically significantly different from expected and so has a high probability of being accurate.

6.3.2.14 Summary

The results of the previous tests have been summarised in Table 6.15. It should be noted that none of the tests failed to produce the expected results and that the worst test cases were only ±26 away from the expected 5000/5000 split. These tests demonstrate that Glade is both accurate and precise and validates the design presented back in Chapter 5.

Table 6.15: Summary of Glade Specific Tests

Test Name	**Chi-Squared statistic**	**P-Value**	**Difference**
Pauli-X	0	1	±0
Pauli-Y	0	1	±0
Pauli-Z	0.0784	0.78	±14
Hadamard	0.01	0.92	±5
Hadamard 2	0.01616	0.98	±15
Phase	0.27	0.6	±26
$\frac{\pi}{8}$	0.27	0.6	±26
Rotate X	0.2116	0.65	±23
Rotate Y	0.1444	0.7	±19
Rotate Z	0	1	±0
Free Rotate	0.16	0.69	±20
Basic Entanglement	0.0324	0.86	±9
W Entanglement	0.25	0.62	±25

6.4 Design and Creation Evaluation

Following the construction and testing of the system (described in Chapter 4 and built in Chapter 5), the next stage is an evaluation in accordance with the principles outlined in Section 3.1.1. Those criteria are:

1. Functionality
2. Completeness
3. Consistency
4. Accuracy

5. Performance
6. Reliability
7. Usability
8. Accessibility
9. Aesthetics
10. Entertainment
11. Fit with organisation

Together, the above criteria explore the system from multiple facets providing a multidimensional review.

Functionality The overall functionality of the software system is to receive and process quantum programs. In this aspect the system performs according to the specification. The novel functionality of the system is the ability to perform parallel processing, in this regard the system further conforms to specification. Regarding the parallel processing the functionality is a poor mans imitation of the true parallel processing available to quantum computers, a true representation of the parallel processing would be a thread for every qubit and performing processing through that. Developing this simulator to function on existing commercially available hardware would require limiting the functionality according to the standard number of cores. To create a truly parallel simulator, one could limit the number of qubits in the simulation to the number of cores in the available hardware and execute all the threads in a truly parallel manner. Given that computer programs do not execute in isolation, it would be almost impossible to guarantee that qubit to core ratio of 1 instead of alternative programs, therefore performing parallel processing according to this design is not feasible. For the purposes of this system the parallel processing is accomplished by utilising a thread (from a pool of size n) per quantum program. This approach demonstrates

the benefit of multi-processing on a quantum computer though does suffer from slight inaccuracies (just like every simulation).

Completeness As explored in Section 5.1 computer models always suffer from inaccuracies, resulting in outputs which suffer from the summation of errors. In this regard very few software models can ever be considered truly complete and certainly not this model (which utilises a number of irrational numbers). Quantum programs are composed of logic gates, which combine to form a complete program. Should a simulator support a universal gate set then any unsupported gates can be created as a composition of the supported gates. In this capacity, the simulator supports the universal gate set of $Hadamard, Phase, \frac{\pi}{8}$ and $Control$ gates as well as a complement of extra commonly available gates.

Consistency The question of consistent results with quantum programs is a difficult one to answer. The output of a quantum program is a single bit string of length n. At the conclusion of the program runtime however the result is a probability distribution of various bit strings of length n. The act of reading results 'collapses' the distribution down to a single result. Because of this, the same program can be executed multiple times and result in x unique bit strings. This difference in output does not mean that the program is producing inconsistent results, it merely requires further examination. Analysis of the probability distribution is required to confirm the consistency of the system. When examination of the distribution is accomplished, the execution of the same program results in an equivalent distribution. In this capacity, the output of the system is strongly consistent, as shown in Section 6.3.2.

Accuracy System accuracy suffers from the same problems as system consistency, in addition to the inaccuracies presented in Section 5.1. Because of the issues presented with measuring system consistency, it is not suitable to measure system accuracy at the end of the system. Further, because of the issues presented in Section 5.1 perfect system accuracy is impossible. Knowing this, the system has been made as accurate as possible by utilising

complex float data types and ensuring adequate data type sizes to reduce truncation and rounding errors.

Performance Performance of a software system can be difficult to measure, composed of any of a number of criteria:

- Execution time
- Resource Usage
- Scalability
- Result correctness
- Result accuracy
- Impact of bias

among others. Therefore the best measures of performance are the tests conducted above.

Reliability The hallmark of a reliable system is one that can successfully handle erroneous conditions without needing a restart. As evidenced from Section 6.3.1, the system has 2 known limitations. Regardless of the speed that programs are received, the amount of programs that are sent or the submission of invalid programs the system continues to compute. The only environments where the system is shown to fail is where the size of the submitted programs was so great that it overwhelmed the host operating system by consuming all available RAM even on a test computer with 256GB of RAM, or such that multiple sources flood the system with programs faster then they can be processed, resulting in forced termination from the operating system.

Usability The system is designed to execute as a server program with no graphical presence. Depending on the command line arguments everything including the logs are written to file automatically with no user involvement. To submit quantum programs or read results, users will need to communicate

with the program on the specified ports. All parts of the system, including logging and ports are able to specified at start time. Because of this design, the system can be utilised through many different methods ranging from user friendly to pure network programming.

Accessibility As explored under usability, the system can be accessed by a range of different methods with varying accessibility. Because this system is not designed with a typical front end, user accessibility is a feature left to the end user. Tools like Postman can simplify the process while providing a graphical user interface, alternatively simple programs can be written to submit programs and retrieve results which can be customised to individual users requirements.

Aesthetics Unlike most products developed with the design and creation methodology the system exists in a bubble insulated from the end user. Due to this bubble, the system has minimal aesthetics to evaluate. The only conceivable aesthetics that can be evaluated are the choice of language with the command line arguments (CLA). The CLA were chosen either for their usage of technical terminology (e.g. port, qubit) or because they were phrases which detailed their function. Because of these reasons, the choice of language was never really available for discussion, though it does assume some familiarity with the technical terminology.

Entertainment The system is not intended to perform an entertaining function and therefore is unable to be evaluated according to this criteria.

Fit with organisation The system is not designed to be the product of any single organisation and was not produced as a commercial product. Instead the system exists as a research tool to be utilised as a testing ground for novel and interesting theorems. As a research tool, the system exhibits the reliability, accuracy and flexibility expected of such a tool. In this capacity the system is a clear fit within the values and expectations of research.

6.5 Chapter Summary

The results featured above in this chapter complete the overview of the original system constructed through Chapters 4 and 5. The above results present the recommended algorithms for various bottlenecks of the system in an attempt to alleviate them. It has been demonstrated that the GladeOS system accurately performs the specified quantum programs and in doing so performs well when compared with other more established simulators. Lastly, now that the system has been established, it has been assessed according to the criteria discussed in Chapter 3 and has been found to perform favorably. This system is not without its flaws, as Chapters 7, 8 and 9 will now illuminate.

7. IMPROVEMENTS FOR THE QUANTUM OPERATING SYSTEM

7.1 List of Improvements

While there is always improvements that can be made to any system, this Section will focus on a short list which each have a large impact. These improvements attempt to address the following issues:

1. Runtime Difference Issue
2. Quantum Networking Integration
3. Inter-programs Synchronisation

The first problem relates to the co-ordination of executing multiple programs at the same time, failure to resolve this issue can result in increased execution time and a strong possibility of increased error rates. The second problem investigates the different ways that quantum networks could be integrated into quantum computers and therefore into the Operating System. Lastly, the final problem investigates the concept of synchronising the execution of multiple programs so that they can work with each other instead of merely working side by side. Each of these problems is defined and discussed below before a recommended solution is provided. Together these improvements demonstrate the complexity of the system and provide an overview of the systems algorithmic complexity (Research Question 3C).

7.2 Runtime Difference Issue

There is a known issue in the base system (Chapter 4) where quantum programs may be able to work alongside each other, but vary significantly on their execution time. For example a program (A) that takes 1 second to execute and a program (B) that takes 10 seconds to execute could potentially perform concurrently, however there will be a large portion of dead time (9 seconds) between the conclusion of program A and B. Because of the difficulties associated with stopping and starting quantum programs, the solution to this problem must be implemented prior to the programs commencement of execution.

It is possible to estimate the length of a quantum program by counting the number of gates on the critical path and multiplying them by the appropriate multipliers (different for each hardware system and so not discussed here). This will provide a base time cost for the program which can then be adjusted for each implementation according to the additional swaps required to make the implementation work (according to the Siraichi algorithm [104] which has been discussed in Section 6.1.1.1).

Using the cost (generated from the procedure in the previous paragraph), it is elementary to compute the difference between two quantum program mappings with a simple subtraction. The problem then shifts to constructing a clique where the sum of the difference is minimal. Which is a simple extension of already implemented algorithms (see Section 6.1.2) to consider the weightings of the edges when composing the cliques. In Figure 7.1 a random example has been included to demonstrate the concept, in this case the optimal clique is M0 and M5 because their time difference is 0.5 which is less than any of the other available cliques (for example M0 and M2 has a cost of 4). In general this approach adds a small piece of data onto the base system with the benefit of presenting a solution to the time difference issue which scales with the number of programs.

If the system can perform measurements of independent qubits then there are no more steps to follow. Otherwise following the identification of the clique, the larger programs begins execution and the smaller program can be delayed in order to synchronise the measurement operations accordingly.

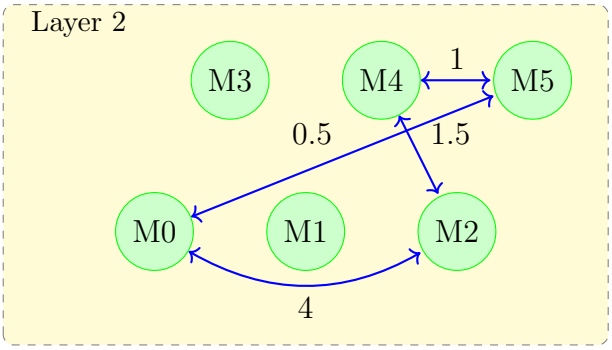


Figure 7.1: An example of the conflict graph (from Section 4.5) with edge values indicating difference between the lengths.

By utilising this approach to generate graphs (like Figure 7.1) the system can then select similarly timed programs to execute. This has the benefit of solving the runtime difference issue by removing the disparity in length between the grouped programs. Using this approach ensures that the programs with the most similar execution time are grouped together and by adding padding (empty space) to the smaller programs these can be artifically lengthened to be equivalent to the larger programs in the group.

7.3 Quantum Networking Integration

Traditional networking involves multiple interconnecting pieces which each transform the data as required. As different technologies interconnect they typically require transforming the data from stored bits into digital bits, and then digital bits to microwave signals or pulses of light [122]–[124]. These transformations require that the data is read bit by bit and then converted, sometimes requiring multiple passes over the same data. Unfortunately quantum data cannot be measured without losing information or cloned at all [28]. Because of this converting quantum bits to alternate representations with-

out loss of information is a very complicated endeavour and the subject of much research [54], [125]. Different hardware implementations of quantum computers have innate properties that lend themselves well to quantum networking. For example, encoding quantum information in photons allows for relative simple transmission down fibre optic cabling [125].

Integrating quantum networking into the Operating System requires great precision. Entanglement links are expensive and should be used carefully to ensure the system receives a maximum benefit. Entanglement links work by sharing a quantum state over a period of space, this link is only usable before either end is measured [54]. Recreating these broken links requires the creation and distribution of a quantum state, the cost (in terms of time and resources) of which is dependent on the technologies chosen. A quantum network connection is commonly used for quantum key distribution [54], [126]–[128], though a natural extension is to enable teleportation of qubits between two distinct quantum computers [54]. An example of linking two distinct quantum computers can be found in Figure 7.2 or Figure 7.3.

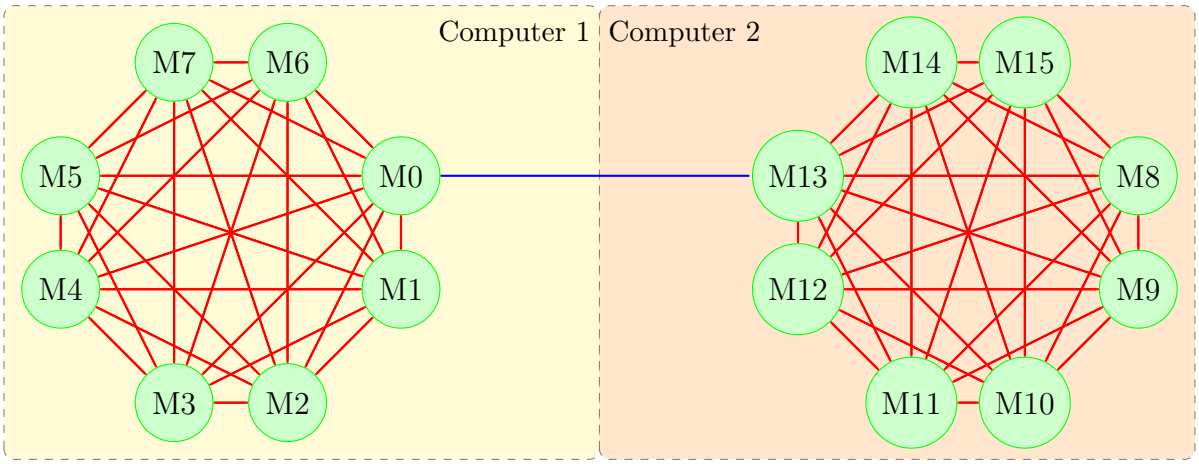


Figure 7.2: An example of Quantum Networking between 2 distinct fully connected computers. The connection is formed through M0 and M13 as an entangled pair

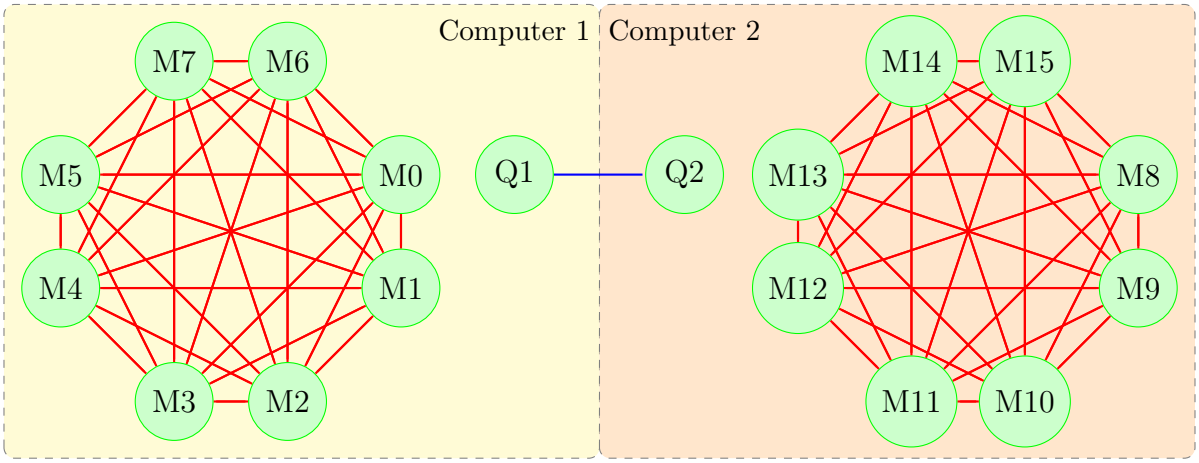


Figure 7.3: A second example of using Quantum Networking to connect two quantum computers. The connection is formed through Q1 and Q2 as an entangled pair, which are distinct from the actual computers themselves (similar to a NIC in traditional computing).

There are at least 2 approaches to network integration, either integration between 2 computers memory directly (Figure 7.2) or connecting quantum networks seperate from the computers memory (similar to a NIC in traditional computing) (Figure 7.3). Regardless of how the networking is implemented, the networked devices can be integrated together.

Quantum network links are delicate with a single usage lifetime, measuring either of the entangled pair will cause the alternate to resolve itself. Improper usage of entangled networking pairs can lead to unexpected interference on programs which utilise the entangled qubits. Figure 7.4 illustrates a quantum circuit where without knowledge of the initial entanglement (B_{00}) the output of the circuit is flawed. The expected output of the circuit is to generate a state of $|11\rangle$. Instead, the quantum circuit in Figure 7.4 results in a largely different outcome (Equation 7.1) which results in the bottom 2 qubits in a superposition of $|00\rangle$ and $|11\rangle$ which is also dependent on the remaining top qubit.

$$\frac{|011\rangle + |100\rangle}{\sqrt{2}} \tag{7.1}$$

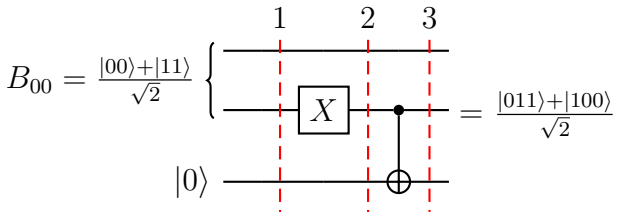


Figure 7.4: An example Quantum Circuit which suffers from errors due to entanglement

Traditional usages of quantum networking links include assisting with quantum key distribution [126]–[128] and accommodating qubit teleportation [54]. The operation of quantum teleportation is performed through the circuit found in Figure 7.5.

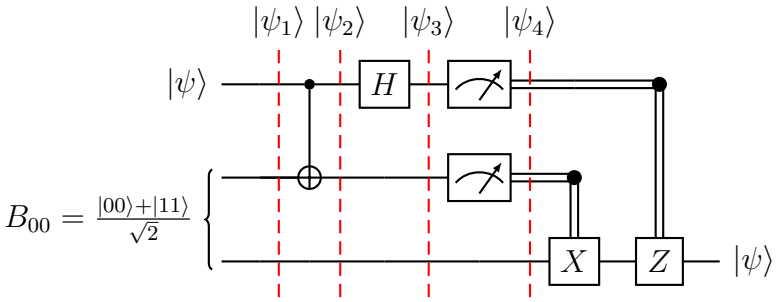


Figure 7.5: Quantum Circuit for the Teleportation Operation

The Quantum Teleportation operation (Figure 7.5) works by 'teleporting' a single qubits state through an entangled pair. To the user this has the effect of moving a single qubits state from one location to another. This transmission is certified against interception [28], [54], [125] and allows the user to detect if someone is attempting to intercept the communication [28], [54], [125]. The ability to transmit the state from 1 qubit to another computer emulates the traditional networking used worldwide and allows quantum computers to work together if they are properly synchronised. An unfortunate part of the Quantum Teleportation operation is that the entangled qubit pair is destroyed during the operation, this means that the pair can only be used once and will need to be regularly replaced in order to be effective [28], [54], [125].

Co-ordinating a program to execute over multiple computers is a standard feature of classical computing, but rather difficult in quantum computing. The method described below borrows heavily from the Siraichi [104] approach, merely extending it to increase the effectiveness. A quick recap of the Siraichi method can be found in Section 6.1.1.1. The Siraichi approach can largely be implemented as is, with two upgrades/updates:

1. When mapping qubits, expand the considered space to include multiple distinct graphs which each represent their quantum computer
2. When joining the distinct sets together, there are two stages:
   (a) Using the entanglement links to get the qubits to the relevant graphs (at great cost to the mapping).
   (b) The standard swapping stage presented in Siraichi.

The first upgrade of expanding the considered area is elementary to implement, as it merely provides more options to sort through. The crux of the difference is in the second upgrade. The following section will explore this change in detail:

7.3.1 Qubit Swapping Outline

Once the qubits have been placed according to the Siraichi algorithm, the algorithm employs 'token swapping' algorithms to move the qubits into the new positions. Token swapping algorithms use a standard graph data structure and place small pieces of data (colours, strings, integers, etc.) known as tokens (see Figure 7.6a). The goal is to move the tokens from their initial placement to the desired final placement in the least swaps possible (see Figure 7.6b). This is accomplished according to the Miltzow algorithm [129], [130] which moves the qubits to their end position one by one. By prioritising swaps where both 'tokens' want to swap with each other (in order to reach their goals), the end resultant sequence is reduced.

Token swapping was originally designed such that each token was unique and there were no repeats [129], [130]. This was then extended to include

the variation where repeats of tokens was allowed (it does not matter which orange token lands on the node, so long as one does) as seen in Figure 7.6c [129], [130]. In both of these cases it doesn't matter how many swaps you perform or how many times you utilise each edge, it only matters that the tokens end up at the correct node. When concerned with moving the qubits within a quantum computer the standard algorithms are sufficient.

When considering moving qubits between quantum computers the problem increases in difficulty. Transporting qubits is best achieved through Quantum Teleportation [54] which can switch the states of 2 entangled qubits, though this is a one sided operation which is difficult to model. It is much easier to consider the problem of Quantum Teleportation if we consider connections between computers as bi-directional, using 2 entanglements in order to send and receive a qubit. This approach then mirrors the token swapping problem mentioned before, with one major difference. That difference is that each teleportation edge can only be used once before it has to be renewed. Because of the difficulty associated with establishing the entanglement links, it is considered that once a teleportation edge has been used it is no longer available for the subsequent operations. Depending on the specific hardware this operating system is implemented on, the renewal of entanglement links may be quite simple and cheap though this is considered to be the minority of cases. In that specific case the links may be considered as reusable resources and treated accordingly.

There is the very real opportunity that not all quantum computers need to move qubits. This is modeled by the satisfied quantum computers already having the correct token for their position. The satisfied computers are still present in the entanglement graph, because they may need to be used, but represented as already solved. There are a number of approaches to solve this problem, which are explored in Sections 7.3.2 to 7.3.5.

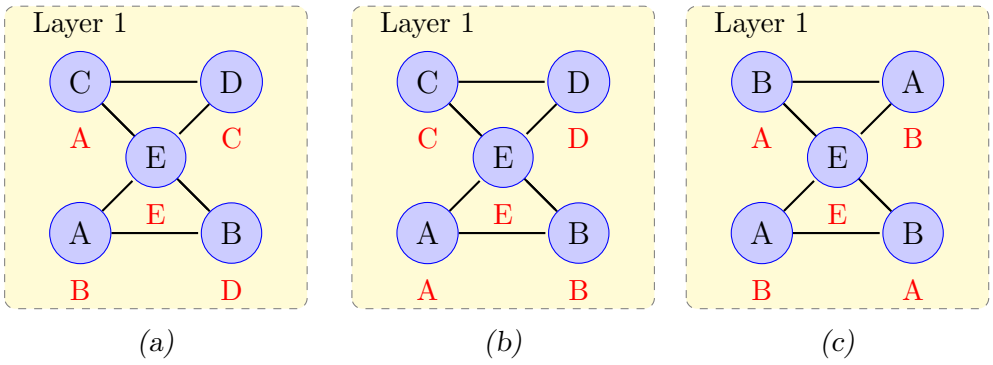


Figure 7.6: An example of a token swapping graph, token's are represented as red letters and must end at the node of the same label. (a) demonstrates an unsolved token swapping graph while (b) depicts a solved version of the same graph and (c) demonstrates a token swapping graph with duplicates allowed.

7.3.2 Brute Force

The first step in attempting to solve any problem is to evaluate the brute force solution for its suitability. The brute force approach for this problem is to attempt every possible permutation of the edge set of all possible lengths and see if that permutation solves the system. By starting from permutations of length 1 and working up to length n, it ensures that the system finds the smallest possible solution. The downside to this approach is that for any suitably large system, the number of permutations is excessive.

$$1 + \sum_{r=1}^{n} \frac{n!}{(n-r)!} \tag{7.2}$$

For example, for a system of 10 computers the number of permutations to step through are $10+90+720+5040+30240+151200+604800+1814400+3628800+3628800 = 9864101$ which at a rate of 10 permutations every second it equates to 986410.1 seconds or, 16440.1683 minutes or, 274 hours or, 11.4 days which is not feasible for use within this algorithm.

7.3.3 Graph Theory Approach

The next approach is to test each swap as it is performed on the graph. This approach is similar to a depth first search and builds a chain of swaps as it progresses. An inline optimisation can be made with this approach by testing after each swap to see if the graph is still balanced (Fig. 7.7). Balanced in this context is defined as every token being able to reach their target node, as seen in Figure 7.7 where each token (red number) can be moved to match the qubit number (e.g. 0 can reach/is linked to Q0). When the system is unbalanced (Fig. 7.8), the system cannot be solved, as 0 is unable to reach the qubit Q0. Therefore when the system becomes unbalanced there is absolutely no sequence of swaps that can make the system solvable and the search can stop and retreat back to the next chain.

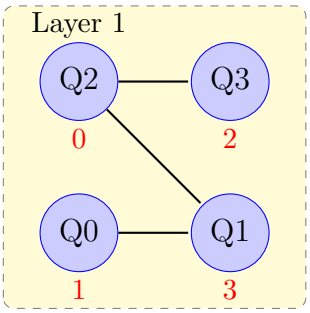


Figure 7.7: An example of a balanced map as every token (red number) can reach/is connected to the relevant qubit number ($0 \rightarrow Q0$)

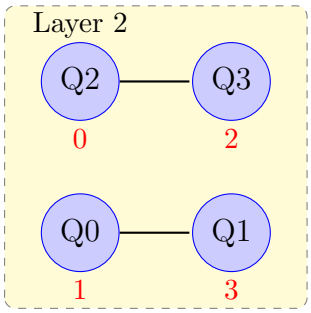


Figure 7.8: An example of an unbalanced map as tokens (red number) are unable to reach the relevant qubit number ($0 \nrightarrow Q0$)

This approach in the worst case will be equivalent to the brute force approach as it needs to test all permutations. In the best case the system is already solved and this will be identified at the start while the average case is not able to calculated as it is largely dependent on the connectivity and size of the graphs. The benefit of using this system, is that it identifies impossible states faster and can find unsolvable systems quicker than brute force.

7.3.4 Matrix Approach

Another approach that can be utilised is to utilise matrix manipulation to resolve the problem. By representing the tokens as a matrix, and the final position as a matrix then a transformation matrix can be used to interconnect the two. Representing the tokens as a matrix is accomplished by listing the tokens currently placed on the nodes in the order of the nodes, the final position is also written in the same order but transposed. Using these three matrices the following equation holds true:

$$\begin{bmatrix} 3 & 1 & 2 \end{bmatrix} \times T = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \tag{7.3}$$

Where T is the transformation matrix, in this specific example:

$$T = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{7.4}$$

Because the transformation matrix is simply moving the values around without editing them at all we can consider this Transformation Matrix as a Permutation Matrix [131]. A permutation matrix is a more specialised transformation matrix and is categorised by being square and having a only a single 1 in each row and column [131]. Taking the permutation matrix, it can be decomposed into a series of elementary matrices using the Gauss-Jordan elimination method [131], [132]. Elementary matrices are matrices which differ from the identity matrices by only a single row operation, in this case swap operations [132]. The process to decompose the matrix is to augment the transformation matrix against the identity matrix and then solve for the inverse [132]. The row operations that are performed to generate the inverse are then used as the series of swaps. Because permutation matrices are by definition composed of a series of swaps they are guaranteed to have an inverse, this approach can be used with any size permutation matrix [131].

The first step in this example is to align the permutation matrix alongside the identity matrix. A solid bar has been placed between them to better delineate the 2 matrices.

$$\left[\begin{array}{ccc|ccc} 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \tag{7.5}$$

The second step in this example is to switch the first and second rows. This swap was chosen first to ensure that the first line of the permutation matrix is now the same as the original identity matrix.

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right] R_1 \leftrightarrow R_2 \tag{7.6}$$

The third (and final) step in this example is to switch the second and third rows. This swap finishes the transformation from the original permutation matrix to the identity matrix. After this swap the sequence of required swaps is now known.

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{array}\right] R_2 \leftrightarrow R_3 \tag{7.7}$$

This method yields the inverse of the matrix and the series of operations that produce the matrix. This approach will always yield a finite sequence of at most n swaps for a $n * n$ matrix, though the swaps may not be allowed with the current entanglement graph. The sequence that is generated is based purely on how the solver approaches finding the inverse (e.g. left to right, right to left, top to bottom etc...). This approach completely ignores the current edge list and struggles to take advantage of a provided graph.

7.3.5 Abstract Algebra Approach

The final approach explored to resolve this problem is through utilising a branch of Abstract Algebra known as Group Theory. Group Theory utilises the cyclic notation for permutations, for example, the transformation used in the Section 7.3.4 would be written like Equation 7.8.

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \tag{7.8}$$

Where Equation 7.8 demonstrates both the 2-line and 1-line notation as the left hand side and right hand side of the Equation respectively. The 2-line notation is read as the top line becomes the bottom line, so 1 becomes 2, 2 becomes 3 and 3 becomes 1. The 1-line notation is read as each number becomes the next one in the chain, it is common to have multiple cycles when written in the 1-line notation. Equation 7.9 demonstrates the multiple cycles perfectly, it should be noted that each cycle starts and ends at the brackets.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 4 & 1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 2 & 4 \end{pmatrix} \begin{pmatrix} 1 & 5 & 3 \end{pmatrix} \tag{7.9}$$

Every permutation can be written in the 1-line notation, as a product of disjoint cycles. Meaning that each element is contained in only one cycle and only appears once. Because the cycles do not conflict with each other they can be written in any order. The 1-line permutation in Equation 7.9 can be decomposed further, as demonstrated by Equation 7.10. Equation 7.10 also highlights a key insight in that while every cycle can be written as a series of transpositions (swaps), it is not a unique series of transpositions.

$$\begin{pmatrix} 2 & 4 \end{pmatrix} \begin{pmatrix} 1 & 5 & 3 \end{pmatrix} = \begin{pmatrix} 2 & 4 \end{pmatrix} \begin{pmatrix} 3 & 5 \end{pmatrix} \begin{pmatrix} 1 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 4 \end{pmatrix} \begin{pmatrix} 1 & 5 \end{pmatrix} \begin{pmatrix} 1 & 3 \end{pmatrix} \tag{7.10}$$

The tricky part with this approach is to ensure that the sequence of

transpositions only includes the swaps that are allowed according to the graph (swaps can only occur if pairs of elements are in the edge list). This can be accomplished through the careful application of some simple transformation rules:

1. (1 2) = (2 1)
2. (1 2) (1 2) = (1) (2) = $\emptyset$
3. (1 2) (3 4) = (3 4) (1 2)
4. (1 2) (2 3) =
   (a) (1 3) (1 2) or
   (b) (2 3) (1 3)
5. (1 2) (3 4) = (1 2) (2 3) (2 3) (3 4)

It is recommended that while rule 5 works with any insertion, the swap that is inserted is two of either (1 3), (2 3), (1 4) or (2 4) this means that both (1 2) and (3 4) can interact with the inserted transpositions (as per rule 4). The parity of a permutation is defined as the number of transpositions (swaps) modulus 2, in other words *odd* or *even*. The parity of a permutation is constant, therefore in order to represent it at a higher level the permutation must grow or shrink by a factor of 2 (as per rule 5). Any attempts to alter a permutation without maintaining the permutation parity will result in a new permutation that is not equivalent to the original.

7.3.6 Recommendations

The Siraichi algorithm will handle the qubit mappings within the quantum computers, while any of the above approaches (brute force → abstract algebra) can determine the swaps required in order to move the qubits around for the next series of operations. This approach extends the Siraichi algorithm by adding in some extra edit commands and extra costs. The swaps can be fit to take advantage of existing entanglement connections, using Section

7.3.2 or 7.3.3 will fit using only existing connections. While Section 7.3.5 can fit but may require new connections to be constructed and Section 7.3.4 has little to no relation to the original graph and will determine an optimal solution that will regularly require multiple new connections.

Recommending a single algorithm is not a simple task, as the algorithmic choice is largely dependent on the hardware that is being used. If the network links are preset and cannot be changed easily, then the recommendation is the Graph Theory approach (Section 7.3.3) as it is guaranteed to provide either a valid solution or a correct impossible ruling. However, if the network links are able to be altered or extended then the Matrix approach (Section 7.3.4) will provide a more optimal solution. It is expected that the Abstract Algebra approach may in time surpass the Graph Theory approach, though determining an algorithmic approach to apply the aforementioned rules is a complex task. Further analysis and research is required in order to determine a singular optimal approach to this distributed computing problem, however the different approaches presented in Sections 7.3.2 to 7.3.5 demonstrate that the problem is at least solvable, if not efficiently.

7.4 Synchronization

The parallel processing presented in Chapter 4 and implemented in Chapter 5 is a primitive variety. By simply segmenting the quantum programs, the quantum programs operate in parallel though they cannot communicate. Part of the strength of classical parallel programming is the communication which allows sharing resources and data [51], [72]–[74]. Two approaches to sophisticated quantum parallel processing are either implementing a shared computing space or employing ancillary qubits.

7.4.1 Shared Computing Space

One concept that is currently not explored is the idea of shared computing space in quantum computing. This ability to share data between programs is an ability found in classical computing via threaded programming models with shared memory space or more complex arrangements with message passing interfaces between processes on the same or different computers. This method is not without risks, these risks can be collected into the following three categories:

1. Qubits entangled with another program and one of them measures.
2. Edit the qubit value (single qubit gates).
3. Overwriting the qubit.

Entangled with another program, one of them measures This category covers the condition where 2 or more programs entangle with the same qubit. This can be used to great effect to link the programs together and share the data between them. However, when done un-intentionally this can lead to premature collapse of superpositions. Which causes incorrect solutions to be generated by the programs. This issue is one that will be extremely hard to diagnose due to the random nature of the race condition, where we cannot guarantee the order of execution of two parallel programs [133].

Edit the qubit value This category covers the condition where data is stored by a single program but edited by another program. This can be used for positive effect by using the result of the second program to influence the first program. The issue stems from cases where neither program is expecting the contact. Here program 1 is expecting their data to remain consistent (or perhaps some third program to connect with it). While program 2 is expecting the data slot to be free and open for storage. This leads to program 1 working with incorrectly edited data, and program 2 has failed to correctly store their data. This leads to both programs failing to execute as expected, though this issue is much easier to debug. A subset of this category is the

measurement case. By measuring the data already stored in the shared compute space, it irrevocably changes the data and can force the original program to experience critical failure.

Overwriting the qubit Shared computing space means that both parties have some degree of access to the same data space. It is a simple process for a program to measure the current state and then set the data to the correct state for storage. This irrevocably overwrites the data and one cannot recover the original state. In this case, the second program has the correct data stored while the first program proceeds with the overwritten data which leads to largely incorrect outputs [134].

7.4.2 Ancillary Qubits

An alternative approach to placing quantum programs on to quantum computers is to place the program onto any qubits and to utilise ancillary qubits in order to forge a symbolic link between the target and the source.

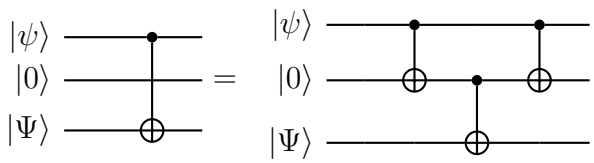


Figure 7.9: Quantum Circuit using Ancillary Qubits

This relationship is demonstrated in Figure 7.9, with a single control not operation being replaced by 3 consecutive operations. Use of Ancillary qubits requires programs to expand their requirements and also requires careful scheduling of each operation. There is no requirement to perform the 3 replacement control not operations consecutively, however they must be completed in order prior to using the ancillary qubit or using the original qubit in another operation.

This approach allows quantum programs to execute on most quantum hardware, provided there is a path of unallocated qubits between each pair of qubits. The major issue with this approach is that every multi-qubit operation now requires $n = 1 + 2 * A$ extra operations to complete (where A stands for the number of ancillary qubits used). Each operation requires time to act which steadily increases the risk of decoherence and other errors within the system.

Another issue with the use of ancillary qubits is it exponentially increases the number of possible mappings to search for. Because of these reasons, the use of ancillary qubits has been abandoned in favour of using mapping techniques like those featured within the Siraichi method[104].

7.4.3 Synchronisation recommendations

Because of the difficulties associated with controlling a single quantum computer, let alone 2 interconnected quantum computers, there are no known quantum algorithms which are designed to work with each other. Without these algorithms, even if a protocol could be established there would be no method to validate the protocol. Based on current quantum computing technology both the shared computing space and ancillary qubit methods increase the number of gates and therefore increasing the error rate of the programs. Therefore it has been decided that as the field currently stands, synchronisation stands as a desideratum.

7.5 Chapter Summary

To summarise, there are a variety of optimisations that can be made to tweak the the original system outlined in Chapter 4 to suit the users specific implementation. The original system outlined in Chapter 4 will work as originally designed; however effective optimisation results in improved performance and efficiency. Implementing the time difference extension is easily integrated into the algorithms found in Chapter 6.

The quantum networking integration requires further research, namely because integration between a universal quantum computer (Section 2.1.1) and quantum networking technology is not readily available. Both universal quantum computers and quantum networks exist in isolation, however their joint application is only theorised about. While the theories enable research to be conducted on their use cases, it is difficult to prepare an answer with the limited resources.

The synchronisation using ancillary qubits greatly simplifies the initial mapping of programs to qubits, however it results in a large growth of overall complexity for runtime. It is for this reason that, while this area may recieve further research, it is deemed ill suited for use in current technology.

8. ALTERNATIVE CONFIGURATIONS

The focus of this Thesis has been the development of an operating system for a specific configuration of 1 classical computer to 1 quantum computer. This configuration was chosen because of the inherent simplicity while also accurately representing the inherent problems. This Chapter extends the discussion on system scaling by introducing and discussing 3 alternate configurations:

1. 1 classical computer - * (n) quantum computer
2. * (n) Classical computer - 1 quantum computer
3. * (n) classical computer - * (n) quantum computer

These configurations are extensions of the already discussed system (Chapters 4 and 7) and aim to apply that system into these configurations. Alternative strategies for handling access within each configuration will exist and this Chapter was not intended to be a universal list of configurable options, merely an introduction.

*8.1 1 Classical Computer - * (n) Quantum Computer*

In the current NISQ (Near Intermediate State Quantum) environment scaling quantum computers to include more qubits is a recognised challenge [135]. An alternative approach to quantum computer scaling is to integrate multiple smaller quantum computers together to form a larger pool of available qubits. This configuration is similar in design to a Beowulf cluster [136] (This has been touched on in Section 7.3). This configuration of multiple distinct

quantum computers can differ based on the inclusion of quantum networking. Section 8.1.1 and Figure 8.1 discuss the implementation where quantum networking is not feasable, while Section 8.1.2 and Figure 8.2 discuss the implementation where quantum networking can be considered.

8.1.1 Quantum Networking Disabled

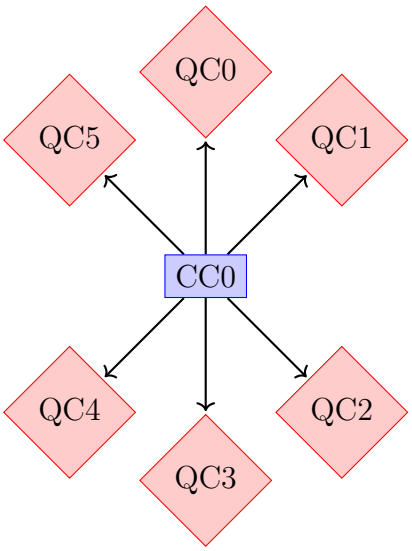


Figure 8.1: 1 Classical Computer (Purple Rectangle) to Multiple Quantum Computers (Red Diamond)

By employing multiple distinct quantum computers under the purview of a single instance of the system, each quantum computer provides a connectivity graph of its qubits and can be instructed individually. One can consider all n quantum computers as a single graph of n connected components thereby allowing for searching over multiple quantum computers. Alternatively you can store it as n graphs each of 1 connected component thereby limiting the search to programs that only fit within a single quantum computer. Searching a single disconnected graph is faster than searching each individual graph due to searching a single albeit larger graph over multiple smaller graphs, however the cost of considering all n graphs is greatly increased from sequentially considering them individually.

Without access to a quantum network connection a quantum program can only be split over multiple distinct quantum computers provided that the activity graph is made of multiple connected components where each component is allocated to a separate quantum computer. If the search is restricted to considering each quantum computer individually then it will ignore the mappings which span multiple computers. A search can be included at the end of the entire system, however this creates double handling of the same search space. Alternatively programs could be partially mapped and then combined with other partial mappings, though this greatly increases the complexity of the search and increases the complexity of the scheduling and mapping space.

8.1.2 Quantum Networking Enabled

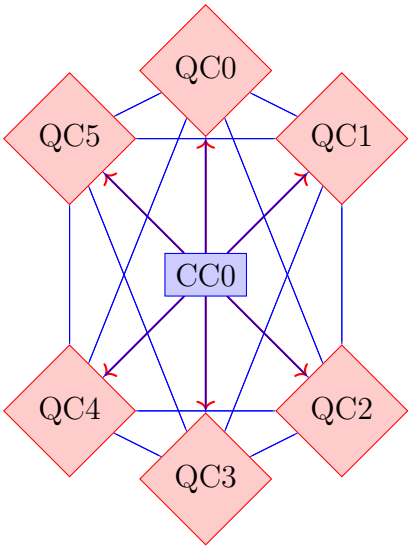


Figure 8.2: 1 Classical Computer (Purple Rectangle) to Multiple Quantum Computers (Red Diamond)

Using a quantum network connection allows programs to be spread over multiple quantum computers through entanglement connections and the use of teleportation operations. As shown in Chapter 7 (Section 7.3) where quantum entanglement links are established, the system cannot be modeled as

completely seperate quantum computers. Quantum computers that are entangled must be searched as a single device, failure to do so can lead to incorrect or invalid mappings. The networking shown in Figure 8.2 is an upper bound, the system can be anywhere in between Figures 8.2 and 8.1. This networking may result in multiple distinct groups of entangled computers which can be considered as a single graph of n entangled components or as n graphs each of 1 entangled components.

Another consideration is that the entangled qubits may be mapped for use within a program. It must be remembered that entangled qubits are not in the base state $|0\rangle$ like the rest of the system, instead they are likely in a Bell State (as provided in Equation 2.6). If this fact is not considered, then the program that utilises those entangled qubits will not return the correct result due to unexpected interference.

8.2 * (n) Classical Computers - 1 Quantum Computer

A configuration that already exists within the multiple online quantum computing platforms [63] is multiple classical computers, all submitting their programs to the same quantum computer. This configuration has two approaches, either the quantum computer talks with a single classical computer directly and the others through the classical network (trusted master nodestar topology [137]) or all quantum computers can talk directly to the classical computer (decentralised network [138], [139] approach). It is assumed that due to the ubiquitous nature of classical networking that all of the classical computers are connected through a network, if this network is the traditional internet then further steps will be required to ensure data security.

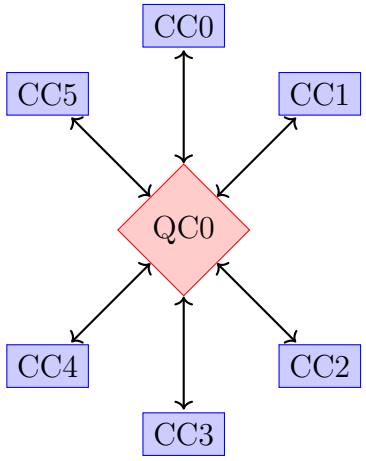


Figure 8.3: Multiple Classical Computers (Purple Rectangle) to 1 Quantum Computer (Red Diamond)

8.2.1 Free for all Implementation

The worst approach to sharing the utilisation of a quantum computer is to implement a *free for all*. This approach features a configuration similar to that shown in Figure 8.3 and has no (or limited) communication between the classical computers, with each of them sending instructions to the quantum computer without considering the effect on the other submissions. This approach will almost certainly result in completely untrustable results due to unintentional interaction between various submissions. Because this approach is such a bad concept and will almost always fail it will be ignored for the rest of this Thesis and is only mentioned for completeness.

8.2.2 Trusted Node (Master computer) Implementation

The trusted node approach hosts the operating system on 'classical computer 0' (CC0), and asks that all submitted programs are transmitted to the trusted node. The trusted node then maps and executes the programs on the quantum computers before returning the relevant results to the relevant origin node.

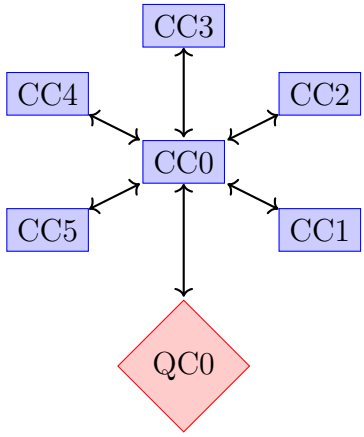


Figure 8.4: Master Classical Computer Implementation

The trusted node approach requires devout and complete trust in the elected node. The trusted node can be used to authenticate and/or block other users in order to ensure that only valid users can gain access. The trusted node could be used to return incorrect or invalid results to the users in order to deceive them. The trusted node requires access to the source code (at minimum the gate operations) of each submitted program, otherwise it cannot pass the submission to the quantum computer.

This access means that the trusted node can duplicate or save the programs for review at another date. This allows the owner of the trusted node to effectively steal intellectual property from the other nodes with minimal trail. In order to combat this, licensing and other agreements can be deployed, though these are out of the scope of this thesis. Blind quantum computing [140], a technique which obscures the purpose of the quantum computation from even the owner of the quantum computer, could also be used to circumvent not trusting the trusted node, however that will require a quantum networking connection.

8.2.3 Decentralised (Blockchain) Implementation

In order to circumvent the required trust in the previous configuration, a decentralised implementation can be employed. This approach is reminiscent of blockchain technologies [141], whereby a single blockchain [141] exists for the quantum computer and nodes may add quantum programs onto the blockchain [141]. With the inherent security of blockchain [141] a firm trail is included to protect intellectual property, though it does require that all users have a copy of the blockchain [141] which will spread the source code further.

This approach still requires a specified node to execute the scheduler and co-ordinate the quantum computer, however the actions of that scheduler can easily be reviewed by the other nodes by observing the manipulations to the blockchain [141].

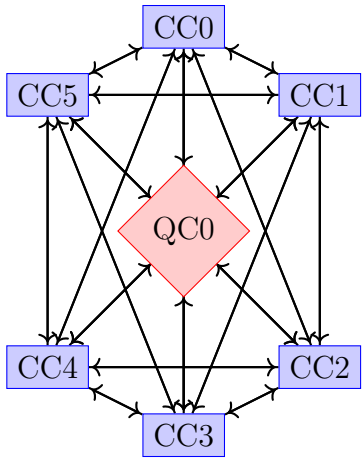


Figure 8.5: Multiple Classical Computers to 1 Quantum Computer

8.3 * (n) Classical Computers (purple rectangle) - * (n) Quantum Computers (red diamond)

The final alternate configuration stems from a blend of the two previous configurations, where an organisation has procured a handful of quantum computers for their multiple staff members to engage. This is analogous to High Performance Computing (HPC) configurations.

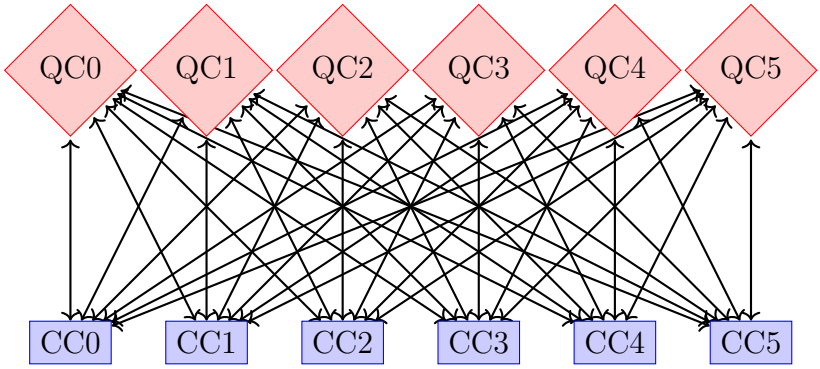


Figure 8.6: Multiple Classical Computers (Purple Rectangle) to Multiple Quantum Computers (Red Diamond)

8.3.1 High-Performance Computing Approach

The HPC approach borrows from the approach utilised by universities and other research institutions with their HPC installations. This approach borrows from the trusted node approach (Section 8.2.2) and provides a specific node which receives the submissions from other classical computers. This trusted node then maps and executes the process on some subset of the available quantum computers. This approach also inherits the weaknesses and cyber security vulnerabilities of Section 8.2.2.

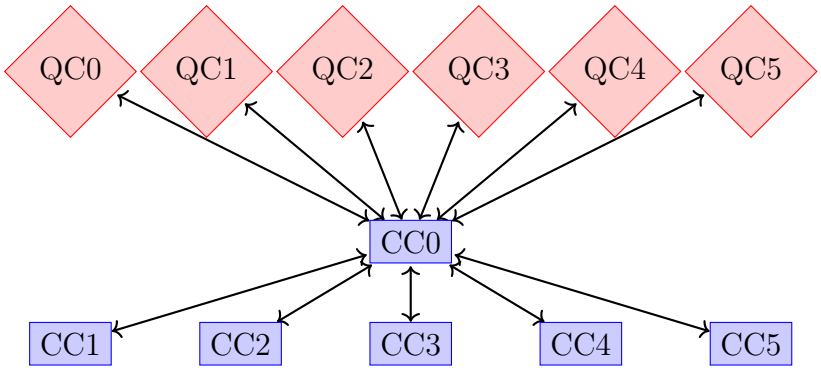


Figure 8.7: Multiple Classical Computers (Purple Rectangle) to Multiple Quantum Computers (Red Diamond) - HPC Approach

8.3.2 Subdivision Approach

Another approach is to break the current groups up into smaller groups, and can even be groups as small as 1-1 groups. This approach can be useful if an organisation wishes to provide for instance 3 quantum computers for their research division (n classical computers) and the rest should be divided amongst all the remaining departments. Alternative use cases could be to provide access for different subscription tiers, though in this case it should be noted that the current system does not support a classical computer belonging to 2 different subdivisions at this stage.

It is not required to provide access to all quantum computers from all classical computers. Where C is the number of classical computers and Q is the number of quantum computers, there are 3 possible outcomes:

1. $C = Q$
2. $C < Q$
3. $C > Q$

By subdividing off the computers the capabilities of any singular program execution are limited, however the complexity of program compilation and assignment is greatly simplified due to the restriced search space. An approach is to subdivide the computers (both classical and quantum) into groups, for example a group of 6 CC and QC can be divided into 2 groups of 3 computers each (Figure 8.8).

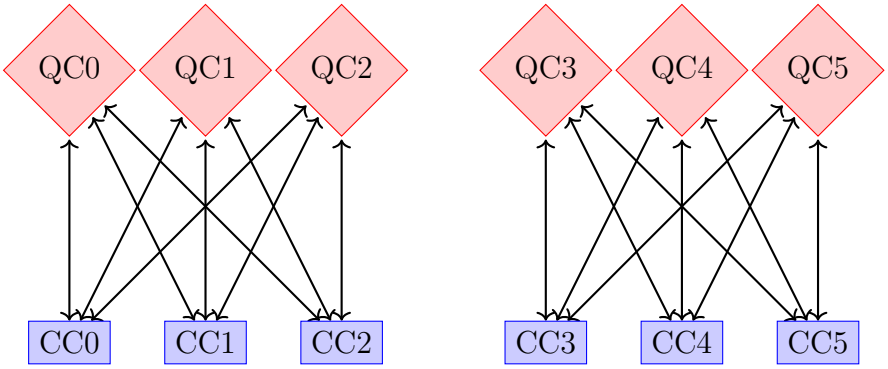


Figure 8.8: Seperate Groups Subdivided from the Original Set.

These subdivided groups do not need to be perfectly split in half, they can be split any which way the user wishes. The sub-divided groups can then be split recursively or treated as distinct groups. After subdividing, the system will resemble one of 3 states. Depending on the resultant state of the system, it is recommended to review the relevant section according to the list:

1. 1 CC and N QC (see Section 8.1)
2. N CC and 1 QC (see Section 8.2)
3. N CC and N QC (see Section 8.3)

8.4 Chapter Summary

This Chapter introduced the various different configurations that quantum computers could be established within networks. The system established in Chapter 4, GladeOS, works with all of the setups described above. These configurations are outlined in order to better discuss the different approaches one could take to fit the Chapter 4 system onto the hardware. These configurations are also introduced here in order to correctly assess the potential cyber security threats in Chapter 9.

9. CYBER SECURITY ANALYSIS

The current research surrounding Quantum computers is focused on the use of quantum computers over the management of and results from quantum programs. Existing research of cyber security vulnerabilities and quantum computers has focused on either issues within the algorithms [127], physical apparatus issues (e.g. photon detector dead time [85]) or using algorithms to break current security standards [6], [7], [142]. Issues between the operating system and the quantum computers have not been as strongly assessed. The vulnerabilities identified below are the result of an initial assessment, with a focus on issues that the introduction of an operating system creates (Research Question 3E). It is expected that there are vulnerabilities beyond what has been identified below and that they will be identified as future research continues.

9.1 Cyber security analysis of GladeOS

The cyber security analysis of the system outlined in this Thesis is separated into destructive and constructive attacks. Destructive attacks do not achieve any tangible benefit to the attacker, they simply disrupt the other quantum programs, a classical example would be DOS (Denial Of Service)/DDOS (Distributed Denial Of Service) attacks [143]. While constructive attacks aim to achieve a beneficial outcome to the attacker, a classical example would be Buffer Overflow attacks [144]. Each of the vulnerabilities identified below are either existing classical security issues, issues that were experienced during this research project or concepts that were generated from the discussions throughout this Thesis.

9.2 Templates

Table 9.1: Vulnerability Template

Vulnerability Name	A short descriptive name
Vulnerability Description	A sentence or two describing the vulnerability
Vulnerability Likelihood	How likely is it that the vulnerability can occur. Low - High
Vulnerability Danger	How dangerous is the vulnerability (how much damage would happen if it was triggered) Low - High
Vulnerability Classification	Does this attack benefit the attacker (Constructive) or just cause chaos (Destructive)? Constructive/Destructive
Attacker Cost	How much effort/resources does the attack require from the attacker? Low - High
Attacker Benefit	How much benefit does the attacker gain from causing this attack? Low - High
Vulnerability Mitigation	A sentence or two exploring how to mitigate the vulnerability
Vulnerable OS Versions	A list of which OS versions can be affected by the vulnerability: 1, 2, 3, 4, 5, 6, 7, 8 (versions according to Table 9.2)

Table 9.1 demonstrates the various components used throughout the cyber security analysis. The overall approach is to:

1. Describe the vulnerability (name and description)
2. Examine the risk of the vulnerability (likelihood and danger)
3. Classify the vulnerability (constructive/destructive)
4. Determine if this attack is worth performing? (cost and benefit)

5. Describe how to mitigate the vulnerability

6. Outline what operating system versions are susceptible to this vulnerability.

These components have been compiled into Table 9.1 with a small description for each component.

9.3 Vulnerability assessment

Now that the standard template has been defined and the various terms explained the vulnerability assessment can begin. This assessment begins by providing an overview of the different operating systems versions in Table 9.2. Following this overview Table 9.3 outlines the complete list of identified vulnerabilities and which operating systems versions are susceptible to each. Each vulnerability is then explored in detail before a summary is provided in Section 9.4.

Table 9.2: Operating System Versions

Version Number	**Version Name**	**Source**
1	Base	Chapter 4
2	Improved	Chapter 7
3	1..n without Networking	Section 8.1.1
4	1..n with Networking	Section 8.1.2
5	Trusted Node	Section 8.2.2
6	Decentralised	Section 8.2.3
7	HPC	Section 8.3.1
8	Subdivision	Section 8.3.2

Due to space considerations the versions in the Table 9.3 are referred to by their version number in Table 9.2. In Table 9.3 a ✓ indicates that the system is susceptible for this vulnerability, while a ✗ indicates that the system is not susceptible to this vulnerability.

Table 9.3: Attack legend

OS_Version	1	2	3	4	5	6	7	8
Invalid File	✓	✓	✓	✓	✓	✓	✓	✓
Slow Loris	✓	✓	✓	✓	✓	✓	✓	✓
Communication Disrupt	✓	✓	✓	✓	✓	✓	✓	✓
Denial of Service	✓	✓	✓	✓	✓	✓	✓	✓
Over allocated memory	✓	✓	✓	✓	✓	✓	✓	✓
Out of bounds memory	✓	✓	✓	✓	✓	✓	✓	✓
Incorrect amplitude	✓	✓	✓	✓	✓	✓	✓	✓
OS interference	✓	✓	✓	✓	✓	✓	✓	✓
False Flag	✗	✗	✓	✓	✓	✓	✓	✓
Improper Subdivision	✗	✗	✗	✗	✗	✗	✗	✓
Untrustworthy leader	✗	✗	✗	✗	✓	✗	✓	✗
Asynchronous Computing	✗	✗	✓	✓	✓	✓	✓	✓
Network Fail	✗	✗	✓	✓	✓	✓	✓	✓
QC System Crash	✗	✗	✓	✓	✓	✓	✓	✓
OS System Crash	✗	✗	✓	✓	✓	✓	✓	✓
Man in the Middle	✓	✓	✓	✓	✓	✓	✓	✓
OS slowdown	✓	✓	✓	✓	✓	✓	✓	✓

Tables 9.4 to 9.20 represent a detailed look at each attack listed in Table 9.3. Each attack is presented using the Template from Section 9.2 with some attacks receiving multiple mitigation sections to encompass the different available approaches. After reviewing each attack in detail a summary can be found in Section 9.4.

Table 9.4: Invalid File attack

Vulnerability Name	Invalid File attack
Vulnerability Description	By supplying an invalid file to the system, it could attempt to force the system to fail in repeatable ways. Invalid files include incorrect qubit numbers, use of `char` for qubits instead of numbers etc...
Vulnerability Likelihood	Low
Vulnerability Danger	Minimal
Vulnerability Classification	Destructive
Attacker Cost	Low
Attacker Benefit	Low
Vulnerability Mitigation	By parsing the files as they are received the system ensures that only valid files are admitted to the system.
Vulnerable OS Versions	1, 2, 3, 4, 5, 6, 7, 8

Table 9.5: Slow Loris

Vulnerability Name	Slow DoS attack
Vulnerability Description	The slow DoS attack involves opening communication with the Operating system and then never completing the communication [145]. This approach can be focused on either a slow READ which uses multiple packets to transmit data or slow GET which utilises all the (finite) available sockets for the system and results in no communication being possible [145]. In this case, the slow GET attack can stop people from being able to submit programs at all by taking all the sockets and holding onto them.
Vulnerability Likelihood	Low
Vulnerability Danger	Low
Vulnerability Classification	Destructive
Attacker Cost	Low
Attacker Benefit	Low
Vulnerability Mitigation #1	Limit the number of connections from a single user (fails against a distributed attack) [145]

Vulnerability Mitigation #2	Restricting the time a client can stay connected for. (Still impacted by the attack, just to a lesser amount. Can also cause valid submissions to fail due to poor network for example).
Vulnerability Mitigation #3	Restricting the minimum bandwidth required for a valid connection. (Fails against users with inadequate, intermittent, limited or faulty connections/internet access)
Vulnerable OS Versions	1, 2, 3, 4, 5, 6, 7, 8

Table 9.6: Communication disruption attack

Vulnerability Name	Communication disruption between OS and Quantum Hardware
Vulnerability Description	If the attacker can gain access to the hardware and therefore intercept the messages between the operating system and the hardware then the attacker can impersonate the OS and launch invalid requests. The attacker could also return false results to the system. NOTE: attack requires access to hardware
Vulnerability Likelihood	Low
Vulnerability Danger	High
Vulnerability Classification	Constructive and/or Destructive
Attacker Cost	High
Attacker Benefit	Medium/High
Vulnerability Mitigation	Proper security considerations must be taken with the physical hardware and its location, there is no method to resolve this through the operating system.
Vulnerable OS Versions	1, 2, 3, 4, 5, 6, 7, 8

Table 9.7: Denial of service attack

Vulnerability Name	(Distributed) Denial Of Service
Vulnerability Description	By spamming excessive amounts of 'fake' programs, the attacker can overload the queue thereby extending the processing time for loading new programs as the serialisability conflict graph scales excessively. This attack can also extend the amount of time it would take to process a valid program that a user submitted.
Vulnerability Likelihood	Medium
Vulnerability Danger	Low
Vulnerability Classification	Destructive
Attacker Cost	Low
Attacker Benefit	Low
Vulnerability Mitigation # 1	The first approach is to place a time limit on each program as they enter the queue. This approach automatically removes a program from the queue if it exceeds the time limit stopping the attack, but will also result in user programs returning an unable to execute error message. This approach will also happen when the system is under a completely valid heavy load.

Vulnerability Mitigation #2	The memoization [146] approach is to take the hash of each program file as it appears, then record the results of already executed programs and respond to duplicate programs with the result from the previous execution. This hashing approach can fix the attack, though if the attacker generates random programs each time then the system will still need to treat them as valid requests. The hashing approach will also interfere with multiple executions of the same program sometimes required to reach multiple distinct results (or generate a probability distribution) and should be used with care. This approach can be used in simulators where they can return the probability distribution as calculated instead of requiring multiple executions (e.g. Qiskit, Q#)
Vulnerability Mitigation #3	The final approach is to profile the source of these programs and cut off access to users who are abusing their upload privileges. This approach will also impact power users in their day to day activities. This approach will fail against a distributed denial of service attack because the source of the programs constantly changes.
Vulnerable OS Versions	1, 2, 3, 4, 5, 6, 7, 8

Table 9.8: Overallocation of memory attack

Vulnerability Name	Over allocation of memory
Vulnerability Description	If the program requests n qubits but requires m qubits (where $n > m$) then the program would reserve extra memory thereby limiting the computational capability of the quantum computer.
Vulnerability Likelihood	Low
Vulnerability Danger	Low
Vulnerability Classification	Destructive
Attacker Cost	Low
Attacker Benefit	Medium
Vulnerability Mitigation	Instead of the program forming the request for computer resources, the request is generated by the system upon review of the program. This approach ensures that the requests for resources are valid and utilise valid program files.
Vulnerable OS Versions	1, 2, 3, 4, 5, 6, 7, 8

Table 9.9: Out of bounds memory access attack

Vulnerability Name	Out of bounds memory access
Vulnerability Description	If the program requests n qubits but requires m qubits (where $n < m$) then the program would attempt to access memory that is not allocated to it.
Vulnerability Likelihood	Low
Vulnerability Danger	Low
Vulnerability Classification	Constructive and/or Destructive
Attacker Cost	Low
Attacker Benefit	Medium
Vulnerability Mitigation	Instead of the program forming the request for computer resources, the request is generated by the system upon review of the program.
Vulnerable OS Versions	1, 2, 3, 4, 5, 6, 7, 8

Table 9.10: Incorrect Original Amplitude Attack

Vulnerability Name	Incorrect Original Amplitude
Vulnerability Description	The simplest destructive attack is to move the qubits to an alternative starting amplitude $\|\psi\rangle$ instead of $\|0\rangle$, this attack causes the next program to use those qubits to provide the incorrect output. This can be accomplished by performing a quantum program composed of a handful of operations which results in $\|\psi\rangle$, and fail to measure the qubits.
Vulnerability Likelihood	Low
Vulnerability Danger	Medium
Vulnerability Classification	Destructive
Attacker Cost	Low
Attacker Benefit	Low
Vulnerability Mitigation	To defeat this attack the operating system should employ a sweep of reset $\|0\rangle$ operations to destroy any quantum data left at the completion of a quantum program.
Vulnerable OS Versions	1, 2, 3, 4, 5, 6, 7, 8

Table 9.11: Program Interference Attack

Vulnerability Name	Operating system/Program Interference
Vulnerability Description	The operating system assumes that it has complete control over the quantum computer, if this is incorrect it can lead to race conditions or multiprogram interference at the quantum computer level.
Vulnerability Likelihood	Low
Vulnerability Danger	Medium
Vulnerability Classification	Destructive
Attacker Cost	Low
Attacker Benefit	Low
Vulnerability Mitigation	There is no known mitigation to this attack.
Vulnerable OS Versions	1, 2, 3, 4, 5, 6, 7, 8

Table 9.12: False Flag Attack

Vulnerability Name	False flag (Digital Twin) attack
Vulnerability Description	The attacker can imitate a quantum computer digitally, and when the operating systems detects the extra quantum computer it will attempt to connect with it. If the attacker is successful in connecting with the operating system then it could do whatever it wanted with the requests sent to it.
Vulnerability Likelihood	Low (requires access to the network/premises)
Vulnerability Danger	High
Vulnerability Classification	Constructive and/or Destructive
Attacker Cost	High
Attacker Benefit	High
Vulnerability Mitigation	Ensuring proper security controls on the network and premises, as well as proper authentication between the operating system and devices. Another approach is to preset the quantum machines that the operating system will control (reducing the dynamic scaling of the system) however the attacker then just needs to impersonate one of the preset machines.
Vulnerable OS Versions	3, 4, 5, 6, 7, 8

Table 9.13: Improper subdivision vulnerability

Vulnerability Name	Improper subdivision
Vulnerability Description	Failure to correctly split into subdivisions leads to multiple operating systems attempting to manage the same quantum computers. This leads to crossover and interference between programs. NOTE: This is a vulnerability without an attacker, it is caused by mismanaged resources.
Vulnerability Likelihood	Low
Vulnerability Danger	Low
Vulnerability Classification	Destructive
Attacker Cost	N/A
Attacker Benefit	N/A
Vulnerability Mitigation	The only complete approach to stop this vulnerability is to have a seperate management program on each quantum computer to ensure their is no crossover (and to manage it when it happens).
Vulnerable OS Versions	8

Table 9.14: Untrustworthy leader attack

Vulnerability Name	Untrustworthy leader
Vulnerability Description	The trusted node approach requires a single 'trusted' leader node to co-ordinate the tasks and host the OS. If this node is not trustworthy then that node can easily implement a 'man in the middle' attack.
Vulnerability Likelihood	Medium
Vulnerability Danger	High
Vulnerability Classification	Constructive and Destructive
Attacker Cost	Low
Attacker Benefit	Medium to High
Vulnerability Mitigation	There is no known mitigation to this attack. Test programs with known expected results can be sent over the connection to profile the connection. A possible resolution can be to elect a different trusted node to act instead of the untrustworthy leader.
Vulnerable OS Versions	5

Table 9.15: Asynchronous Computing Attack

Vulnerability Name	Asynchronous execution
Vulnerability Description	When working with multiple quantum computers on a single problem, the communication delay between the OS and the quantum computers can cause executions to become asynchronous. This delay causes gates to be executed out of sequence and therefore corrupts the program execution. This problem can be exacerbated by attackers intercepting and delaying network traffic between the computers.
Vulnerability Likelihood	Medium / High
Vulnerability Danger	Medium
Vulnerability Classification	Destructive
Attacker Cost	Low
Attacker Benefit	Low

Vulnerability Mitigation	The traditional response to this problem is to pause computation while waiting for the secondary machine to catch up and synchronise [147]. Unfortunately due to the constraints of quantum mechanics this approach does not work. The easiest approach is to account for the delay on the lines and delay the quicker message so that it should arrive at the same time as the slower message. This vulnerability is strongly correlated with the 2 generals problem [148] in computer networking in that there is no perfect solution to this problem.
Vulnerable OS Versions	3, 4, 5, 6, 7, 8

Table 9.16: Network Fail Attack

Vulnerability Name	Network Fail
Vulnerability Description	Communication between the Operating System and the Quantum Computer is facilitated through traditional networking, even quantum networking channels must be supported by classical networking channels [54]. Should the classical network channels fail (for any reason) then the quantum networking channels they support are either delayed or outright useless [54]. Likewise, if the quantum computer can no longer communicate with the operating system then the entire stack collapses [54].
Vulnerability Likelihood	Low
Vulnerability Danger	Medium
Vulnerability Classification	Destructive
Attacker Cost	Low/Medium
Attacker Benefit	Low
Vulnerability Mitigation	By following traditional computer networking standards the majority of these issues should be mitigated. Other issues like cable failure can be very difficult to predict and/or mitigate and must be dealt with as they occur.
Vulnerable OS Versions	1,2, 3, 4, 5, 6, 7, 8

Table 9.17: Operating System Crash Attack

Vulnerability Name	Operating System Crash
Vulnerability Description	Causing the computer executing the operating system to crash results in the entire software stack collapsing. This crash can be caused at the operating system level itself, or on the physical machine itself (either through networking or physical interaction).
Vulnerability Likelihood	Low
Vulnerability Danger	High
Vulnerability Classification	Destructive
Attacker Cost	High
Attacker Benefit	Low
Vulnerability Mitigation	Controlling access to the physical machine is the major priority. If the only method to access the computer is through the submission of jobs, then provided that the system validates job submissions it should be relatively impregnable. Securing the aforementioned will result in mitigation of this vulnerability.
Vulnerable OS Versions	1, 2, 3, 4, 5, 6, 7, 8

Table 9.18: Quantum Computer System Crash Attack

Vulnerability Name	Quantum Computer System Crash
Vulnerability Description	Overloading any part of the fragile quantum computers can cause them to crash and fail to compute anything. This requires precise knowledge of the hardware and/or access to the machine either physical or over the network.
Vulnerability Likelihood	Medium
Vulnerability Danger	Medium
Vulnerability Classification	Destructive
Attacker Cost	Medium
Attacker Benefit	Low
Vulnerability Mitigation	Securing access to the quantum computers is priority number 1. The next step is to ensure that all editable variables are validated before being executed. These 2 steps should result in a complete mitigation. More specific mitigation will depend on the quantum computers utilised in the system.
Vulnerable OS Versions	1,2, 3, 4, 5, 6, 7, 8

Table 9.19: Man in the Middle

Vulnerability Name	Man in the Middle
Vulnerability Description	Intercepting and editing user requests before they arrive at the quantum computer can allow attackers to edit the request to either stymie the legitimate user, benefit the attacker or just to eavesdrop on the calculation and results.
Vulnerability Likelihood	Medium
Vulnerability Danger	High
Vulnerability Classification	Constructive and/or Destructive
Attacker Cost	Medium
Attacker Benefit	Low and/or High
Vulnerability Mitigation	Because the vulnerability occurs prior to communication with the OS, there is limited recourse for the OS to take. Using classical standards like encrypting communication between the OS and the client, will attempt to minimise the occurrence of this attack.
Vulnerable OS Versions	1, 2, 3, 4, 5, 6, 7, 8

Table 9.20: OS slowdown attack

Vulnerability Name	OS slowdown
Vulnerability Description	Spamming the system with requests which only consume 1 qubit will be easily added to the queue and therefore simply mapped to every qubit in the system. This will result in n mappings (for a system with n qubits) and will greatly increase the number of mappings saved in the system. When the computer next attempts to load programs, it will have to parse an extra dense conflict serialisability graph which will take extra time. This attack will slow down the computer at the very least if not reach denial of service levels. NOTE: just like the slow loris attack, this attack could entirely be caused through legitimate means.
Vulnerability Likelihood	Medium
Vulnerability Danger	Low
Vulnerability Classification	Destructive
Attacker Cost	Low
Attacker Benefit	Low

Vulnerability Mitigation	It would be trivial to exclude these smaller process requests, however they will not always be malicious requests. There is no mitigation except to process the requests as received. To reduce the time spent searching through the enourmous conflict serialisability graph, a dynamic graph algorithm could be employed which will update its choice as new mappings are received and be ready to provide the OS with 'a' clique when requested. This way the runtime of these quantum computations is utilised as processing time for the clique algorithm.
Vulnerable OS Versions	1, 2, 3, 4, 5, 6, 7, 8

9.4 Attack Summary

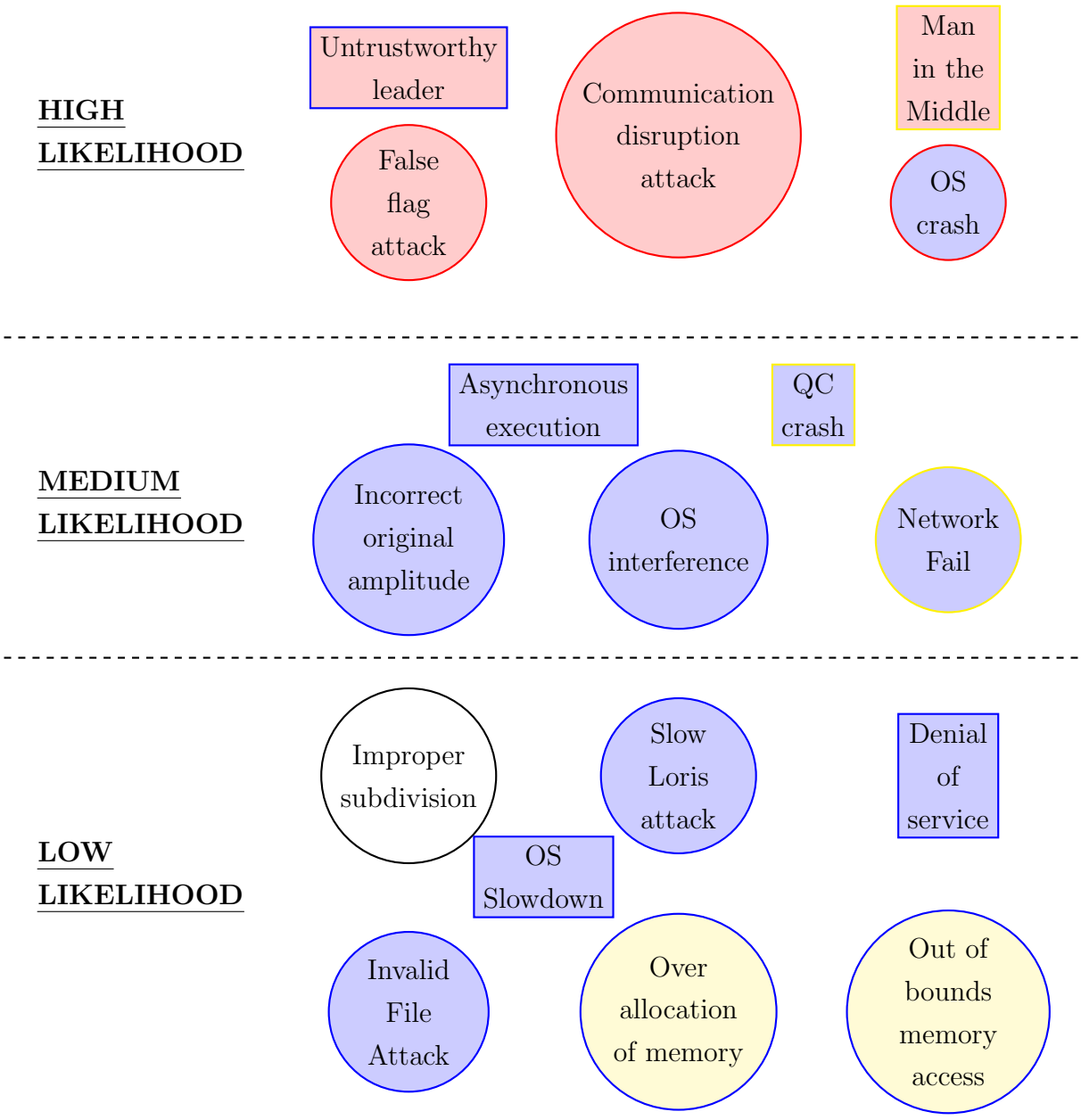


Figure 9.1: An overview of all of the attacks explored above. Y-axis Height indicates Likelihood, Shape indicates Danger (Circle=Low, Rectangle=Medium), Shape Border colour indicates Attacker Cost (Blue=Low, Yellow=Medium, Red=High) and Shape Fill colour indicates Attacker Benefit (Blue=Low, Yellow=Medium, Red=High).

9.5 Chapter Summary

This Chapter explored a series of possible vulnerabilities within the system as described in Chapter 4 as a solution to Research Question 3E. The discussion was extended to include whether the alternate configurations, with and without networking, were susceptible to the vulnerabilities as well. It is assumed that there are vulnerabilities that were not identified in this Thesis, as the field grows and develops over time. As seen in Table 9.3, different variations of the operating system are vulnerable to different attacks. The base and improved systems appear to be the most secure, however that is because the vulnerabilities they are not susceptible to are vulnerabilities related to the integration between multiple machines. Increasing the complexity of the system (adding more computers) increases the attack surface and results in more vulnerabilities.

Mitigation strategies have been identified for every attack listed in Table 9.3, though 8 of the 17 strategies acknowledge that they do not completely solve the problem. Most of the identified vulnerabilities have a mitigation identified and assuming that the mitigations are applied then these vulnerabilities can mostly be treated as solved. Should the mitigations not be correctly applied, then these vulnerabilities can still be considered as active attack vectors into the users system. A proper security analysis will require access to physical hardware for a consistent period of time in order to accurately test these and other vulnerabilities.

10. DISCUSSION AND CONCLUSION

10.1 Discussion

At the beginning of this Thesis 3 research questions were presented to the reader.

RQ1 Can a quantum computer support processing of multiple programs concurrently?

RQ2 Is there a framework which can be implemented on a quantum computer to support processing of multiple programs concurrently?

RQ3 What is the cost and/or effect of adopting this framework on a quantum computer? The cost and effect are considered in terms of:

RQ3A Implementation,

RQ3B Performance,

RQ3C Algorithmic Complexity,

RQ3D Versatility/portability,

RQ3E Security

These research questions were developed throughout the Thesis with solutions presented throughout the document.

Chapter 4 presented a 'validity of concurrency' (Section 4.4) which directly confirmed Research Question 1. This proof demonstrated that it was possible to support processing of more than 1 programs concurrently. This proof was then augmented by presenting a framework which addressed Research Question 2. This system demonstrated the ease with which multiple

program management can be developed into a quantum system.

Chapter 5 built on the framework presented in Chapter 4 by implementing it into code. This implementation provides a solution for Research Question 3A and works as the base implementation for the other Chapters and results in the Thesis. Overall the major cost in the implementation was the simulation of the qubits and this would be removed when the system is implemented on physical hardware.

Chapters 6 and 7 worked to present Research Questions 3B and 3C. Because there are no other multi-program quantum systems currently available a theoretical check is not possible, therefore the performance comparison is made between the implementation from Chapter 5 and commercially available alternatives. It is difficult to check the performance of a quantum simulator directly due to the probabilistic nature of the computation. Therefore the testing was performed in terms of processing speed and memory usage, with separate tests to ensure the accuracy of the implementation. In all of the criteria the implementation excelled and this was especially clear in Figure 6.20 where the results were vastly different.

For Research Question 3C the underlying algorithmic complexity of the framework is minimal with the majority of the bottlenecks being program mapping, clique detection or program scheduling algorithms. The program mapping algorithm is currently recommended to utilise the Siraichi algorithm [104] which has a performance of $O(|Q|!^2 \times |Q| \times |\psi|)$ where $|Q|$ is the number of qubits in the hardware graph and $|\psi|$ is the number of edges in the activity graph. The recommended clique detection algorithm is to employ a random search of nodes with a complexity of $O(2n^2)$ for n mappings and the recommended scheduler is FIFO which contains a minimal overhead, merely requiring a queue to be maintained. Together these complexities demonstrate that ongoing work is needed to optimise this framework but an initial standard has now been created that other research can be benchmarked against.

Chapter 4 had a minor discussion on the versatility of the initial framework which was then scaled up in Chapter 8. These discussions demonstrated that the answer to Research Question 3D is resoundingly positive. The framework presented not only scales for 1 classical computer and n quantum computers, but can by extended to also include multiple classical computers alongside multiple quantum computers. The modular nature of the framework does mean that the system will suffer performance losses if the quantum computers get too large or numerous, however better algorithms and other optimisations can work to reduce these losses. Because the framework was developed with hardware agnosticism as a core principal this framework is applicable to all situations making it as versatile as possible.

Finally Chapter 9 built on the existing cyber security issues and analysed the vulnerabilities found within the framework (and the listed variations). These vulnerabilities go well beyond the existing set of vulnerabilities which focused on issues within the algorithms, physical apparatus issues or using algorithms to break current security standards. The vulnerabilities presented in this Chapter compose an initial understanding of the risks of Quantum Systems Software and a solution or mitigation has been presented for each of the issues. This discussion demonstrates that the answer to Research Question 3E is that yes there are risks and issues but these issues are equivalent or less than current classical computer cyber security issues.

Also earlier in this Thesis, the gap statement (Section 2.3.3) introduced 3 areas that were missing from the literature:

1. Multi-processing
2. Networking between quantum computers
3. Security from cyber threats

This Thesis has demonstrated novel contributions and addressed these research questions through the implementation of the Operating System contained within this Thesis.

The first area of multi-processing was proven feasible in Section 4.4 and is the basis around which the entire operating system is based. This Thesis has not only demonstrated that multi-processing is possible in theory, but in Chapter 6 it has clearly shown the benefit of this approach.

Quantum Networking is an already acknowledged field and is well researched [54], [125], [149]–[151]. The component that is missing is exactly how the already explored quantum networks can interconnect with the quantum computers. This problem could easily be the focus of an entire extra PhD program and so simplifications needed to be made to facilitate the discussion found within this Thesis. Section 7.3 explored two possible implementations of quantum networking before shifting to the underlying problem of how to integrate the networking capability with existing quantum programs and a discussion on how to resolve this problem. It is clear that this area requires further study to be ready for implementation however the discussion presented in Section 7.3 demonstrates that this is a solvable problem by employing a variant of the token swapping algorithm within the program mapping algorithm.

Lastly, the third area of security from cyber threats is one that has, until now, not appeared in the public repositories. There is significant research around using quantum computers to facilitate cyber security attacks by performing feats like breaking encryption algorithms[6], [7], [142]. There has even been attacks designed to disrupt quantum networking connections between quantum devices [85], [127]. While fascinating there has not been a discussion of the numerous problems that can occur when quantum computers are processing data. Until the implementation of multi-processing these effects were largely theoretical, now they are almost tangible. The discussion found in Chapter 9 demonstrates the wealth of possible methods of attack and should highlight the need for a further discussion of these effects.

10.2 Key findings

Within this Thesis a nascent merging of the fields Quantum Computing and Operating Systems has been discussed. The result of this is a Quantum Computer Operating System which has been designed according to hardware agnostic principles. This design was then implemented before being critically evaluated according to numerous benchmarks including outperforming existing simulators.

The results presented within this Thesis demonstrates a practical implementation of a newly designed multi-processing quantum operating system. This system performs to a similar standard as the existing single-processing systems with limited overhead. This system is designed to horizontally or vertically scale to work across a wide variety of quantum computing systems.

10.3 Interpretations

The system implemented in Chapter 5 was developed in accordance with accepted definitions within the unique approach developed within this Thesis. For a system developed under these conditions to automatically produce similar results when compared to commercial variants like Qiskit, Rigetti and Q# strongly indicates that this multi-processing approach should continue to yield strong positive impacts. With further development from specialised developers and implementation onto actual quantum hardware it is expected that this trend should only continue.

It is readily acknowledged that bottlenecks exist within GladeOS including the quantum program mapping and searching for maximal cliques. In fact Chapters 6 and 7 were dedicated to directly examining some of them and offering potential adjustments for them like adding weights to the clique graph to assist with the runtime difference issue. At this stage some bottlenecks have been resolved to acceptable standards while others will require further study and subsequent research to resolve.

10.4 Implications

The research presented in this Thesis has demonstrated that the previous systems of quantum computing were inefficient. The existing simulators are perfectly viable for the current small devices, with the inefficiencies growing as the devices scaled. Quantum computing is already an incredibly expensive field ($674 million dollars US [8]) and any reduction in the operating cost has potential for considerable cost reductions.

The research presented in this Thesis demonstrates that parallel processing can be included with their standard operating procedure. Further to this new capability the research also considered a multitude of security concerns surrounding this parallel system (and the variants), highlighting where vulnerabilities exist and recommendations of how to alleviate them.

Accurate implementation of GladeOS as outlined in this Thesis should result in notable benefits. These benefits include savings in the cost of running and maintaining these expensive machines and in the turnaround time of quantum jobs submitted by clients and users. For example, by executing multiple quantum programs the overall execution time is reduced thereby requiring less liquid nitrogen to cool the computer.

As discussed in Chapter 9, this system is not without risks. These risks may be a negative for some interested parties, but of great interest to others. Eventually parallel processing on a quantum computer will be implemented and a thorough discussion will need to be built on the introduction provided in Chapter 9. This Thesis has barely scratched the possibilities for the effect of controlled interaction between multiple parallel quantum programs.

10.5 Limitations

The research presented in this Thesis explores a field which has received limited coverage. This research has been completed over a 3 year period which

greatly limited the amount of long term research options. The focus of this research was on theoretical and simulated results given the limited accessibility to quantum computing hardware. The establishment of these results provides a setting for additional experiments and verification on quantum computing hardware.

The simulator (Chapter 5) was designed to use the α and β approach along with the accepted logic gate representations. This technique allows users to simulate quantum computation, however it fails to consider the following:

- Accurate simulation of decoherence [28]
- Precision of complex floats compared to complex numbers
- Cross talk in quantum systems [152]
- Mixed states (and density operators) [28]
- The idiosyncratic tendencies of specific hardware implementations. [28], [39]
- The speed of quantum computers [28]
- Accuracy of the logic gates (always assuming 100% accuracy)
- Accuracy of the measurement operation
- Quantum Error Correction [153], [154]
- Inherent parallel operations of quantum computers.
- Multiple quantum computers interacting through networking (either traditional or quantum) [54]

The simulator was designed specifically to audit the concept of executing concurrent quantum programs and it accomplished that. Further experiments will be required to fully determine the effect that parallel quantum programs

have on a physical quantum computer.

Without the ability to compare our results with other approaches at resolving the same parallel programming problem, there is no accepted method to critically critique the research output. Within the research a comparison has been made to commercial sequential programming simulators, though that comparison is limited in it's effectiveness because the simulators are all attempting to prioritise different variables and details (including accuracy, speed, simplicity, etc.). Because of this missing method, a variety of tests were executed in Chapter 6, with the hope that one or a combination of these could act as a stand-in for this benchmarking method.

10.6 Research conducted since this Thesis began

As this field continues to rapidly evolve there is additional recent literature which needs to be considered when discussing the future directions.

10.6.1 Quantum Hardware Updates

IBM initially released their quantum roadmap around November 2021 [155] before updating it in 2022 [156] with a focus on:

1. Increase performance of the processor
2. Develop a better understanding of errors
3. Simplify how quantum computers are programmed

IBM argues that they can increase the performance of the processor by joining together multiple smaller quantum chips instead of attempting to directly scale the technology like Google. IBM's approach works based on the implementation of short range (high fidelity) connections and long range (up to 1m long) (low fidelity) connections [156]. IBM also claims to be implementing dynamic circuits which allow the user to send commands to the computer and receive feedback. This feedback can then be used to perform further computation which is a feature that has not been seen before [156].

This feature further extends to the proposed implementation of using classical multi-threading to segment quantum circuits based on their trivially parallelisable form so that the execution can be spread over multiple distinct quantum computers [156]. Combining the 2 above features with the last concept of quantum circuit knitting (breaking a singular circuit down into an equivalent series of smaller circuits, executing them and then recombining the results) IBM is threatening to radically alter the quantum landscape from a batch processing system and turn it into a real-time system [156].

In comparison, Google released their quantum roadmap back in May 2021 [157] with a focus on further developing the quantum hardware and control software compared to IBM's approach of developing support for quantum software support. Googles approach hinges on the concept of combining between 100 and 1000 physical qubits and converting it into a single error corrected logical qubit [158]. In future hardware versions Google plans to combine these long lived qubits together in order to function as a long term quantum computer [158].

While there is nothing explicitly wrong with either roadmap, it will be interesting to see which technology lasts the test of time. Both roadmaps are on track to complete and each promises unique benefits. If the two technology giants can each follow their roadmap and achieve their milestones then the field of quantum computing is about to receive a large shake up. A combination of the larger computers from Google coupled with the software techniques coming from IBM would see quantum computers becoming much more reliable and much more efficient.

10.6.2 Parallel Processing Updates

- Qubit interference
- Break the qubit map into segments based on error rates.
- Simultaneous execution of circuits.

Other researchers have begun working on the concept of executing multiple quantum circuits on distinct pieces of the hardware. This is being pursued for a couple of reasons:

- Removal of high error rate sections.
- Increase processing speed due to smaller search space.
- Qubit Interference.

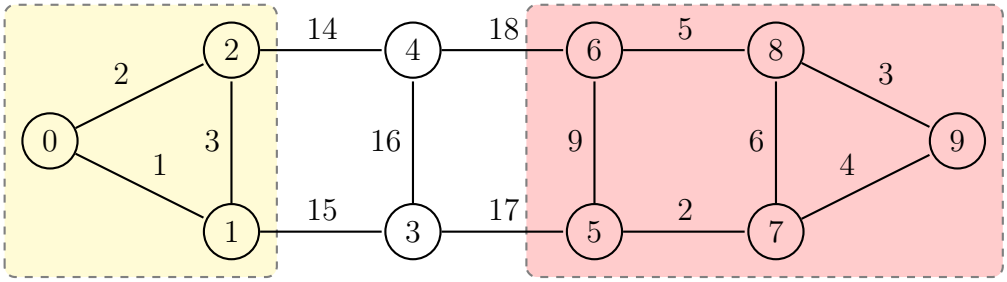


Figure 10.1: Segmented connectivity graph for a hypothetical quantum computer. The number on the edge indicates the error rate for that edge (higher is worse). The number on the node indicates the unique node ID.

In [159] the researchers have recognised the same inefficiency that has been highlighted in this Thesis and have come to a similar conclusion regarding the adoption of parallel processing. The research attempted to address parallel processing through 3 main points [159]:

1. "We advocate the use of multi-programming to improve the utilization and throughput of NISQ computers, whereby the qubits are used to concurrently run multiple workloads."
2. "We develop Fair and Reliable Partitioning (FRP) algorithms that try to split the qubit resources into multiple groups in a fair manner, while avoiding the qubits/links that have extremely high error rates."
3. "We develop the Delayed Instruction Scheduling (DIS) policy to mitigate the interference of measurement operations of one program on the gate operations of the co-running programs."

4. "We propose an Adaptive Multi-Programming (AMP) design that monitors the reliability impact at run time and reverts the system to isolated execution mode if the reliability impact is high."

As the first point is addressed by the implementation of the second point this Thesis considers them as a singular point. With regard to these points, the first point of multi-programming is a large portion of this Thesis and while Das et al [159] encourage the study of this field they set an arbitrary limit in themselves of only 2 programs at a time. This limitation seems to have been imposed for no other reason than because 2 is an easy starting point. It should be recognised that Das et al's approach does cross check the error rates of the 2 program mapping against the singular program mappings. This cross check increases the required processing but does work towards ensuring that every program receives optimal results.

To the second point of Das's DIS policy, this is only an issue if the quantum computer is not compliant with the DiVincenzo criteria (established in [16]. If the computer complies with the criteria than distinct/independent measurement of a qubit is available and can be used to run programs at any rate. If the computer does not comply with the criteria then the measurements must be co-ordinated as Ohkura et al [160] wrote. This restriction has also been noted within this Thesis in Chapter (7) and both researchers have reached a consistent solution.

To the final point of Das's AMP design, while the comparisons between the results of their individual and shared trials is appreciated the approach suffers from the presupposition that the individual test is more accurate than the shared test. A true measure of reliability is only possible if the system is continuously running some pre-determined diagnostic program for which the true result is known ahead of time. This is because without knowledge of the true solution there is no way to determine if the computer has performed an erroneous computation or simply returned an unlikely result. Because of this oversight, the proposed framework has a sharply reduced efficiency when used in a 'production' environment. Users of a quantum computer service

rarely know the solution to their calculations prior to excution of their code, if they knew the solution then the code would not need to be executed. The only way to utilise the framework proposed by Das et al [159] is to build in a calibration test every few runs (to test the reliability) which takes time away from the queue the computer is processing but helps to ensure the accuracy of the results. An alternative to the AMP framework is to multiply the results from the individual run by 2 instead of running the parallel runs. This would have the added benefit of halving the processing time and ensuring that the results of the shared trials neatly match the individual trial results.

The proposed framework in Das's paper is clever in its attempt to continuously adjust the 'partitions' according to the 2 programs that want to run on the computer (like Figure 10.1). For example, one run of the framework could group qubits 0, 1 and 2 as group 1 and build group 2 of qubits 5, 6, 7 and 8. A subsequent run of the framework may realign the partitions to be qubits 1, 2, 3, 4, 5 and 6 for group 1 and 7, 8 and 9 for group 2. This approach allows the system to be flexible and adjust depending on the programs that are attempting to execute. The problem with this approach is the significant computational overhead associated with this approach, for every run of the framework the partitions need to be recalculated (from scratch) and only then can the mapping process for the quantum algorithm begin.

In [161] Niu and Todri-Sanial have again recognised the need for multi-programming and even refer to [159] as inspiration. The focus in this research is regarding the effect of crosstalk on their parallel circuits and how best to work around that. Curiously, in this paper the researchers included an approach to calculate the maximum amount of throughput that can be performed on a quantum computer with reliable results. At the time of research the upper limit was computed to be approximately 38% [161], and though this has likely changed and is computer dependent this approach could easily be adopted for future research.

In [162] Deshpande et al raises the concept that parallel circuit execution

could have a negative impact due to the inclusion of crosstalk. This is not an error featured in my cyber security analysis (Chapter 9) however there are numerous considerations I have identified that have not made it into Deshpande's research. As this research is marked as 'in progress' there is little in the way of results to review. With that said, the research completed here seems to indicate pairing a very specific (almost tailored) malicious circuit with the target circuit in order to have an effect. The research conducted within this Thesis uses a random allocation algorithm designed to fill spaces rather than placing specific circuits with each other thereby relegating the issue to an unfortunate coincidence over a targeted attack. The notion of a quantum antivirus raises interesting questions though I do not think they will bear closer examination. Quantum computers are still too nascent and individual with different qubit layouts and interference patterns for a set of malicious code patterns to be compiled within a database.

In [160] Ohkura et al present the concept that the amount of crosstalk present in a circuit is related to the distance (defined in qubits) between parallel quantum circuits. While this relationship was not able to be directly quantified it was experimentally demonstrated that as the distance between circuits increased the rate of crosstalk fell.

10.6.3 Effect of current literature on the research completed in this Thesis

The literature and roadmaps presented above strongly indicate that parallel quantum execution is (as argued in this Thesis) one of the next natural evolutions of the technology. Due to current resource limitations the literature presented here is limited to discussions of NISQ devices the research presented within this Thesis can be adjusted to account for NISQ devices or deployed as outlined on a more stable quantum computer (using logical qubits over physical qubits).

The recent literature presented above has highlighted a handful of things that were not able to be considered within the research conducted in this

Thesis. The largest unfaced issue is the idea of crosstalk and errors being generated because of parallel circuits. This issue was not able to be properly addressed because (as noted in Chapter 5) no appropriate simulator was identified and so one was created. Because of this crosstalk was not able to be considered as it is an effect only found in physical quantum computer hardware. To address crosstalk in the approach outlined in this Thesis the only change is an extra process occurring on the hardware graph. Instead of merely using the hardware graph as a means to link qubits to program mappings, it can be extended to allow qubits to be disabled thereby disabling all program mappings which utilise that qubit. Alternatively, if crosstalk sections are identified then one could include those sections in the program mappings and not actually use them. This would isolate the sensitive region of the computer and allow for better computation without needing to disable qubits.

10.7 Recommendations and Future Work

It is expected that research will continue in the field of quantum systems software, with great strides still to be made as the hardware develops further. Implementing this system onto physical quantum hardware is the obvious next stage of this research, which will provide much richer data sets. Reducing the run time of these initial quantum computers (which rely heavily on hard to come by resources) is a goal that must be encouraged. Whether through program optimisation, multi-processing or some other means the end result of reducing execution time is a positive impact.

The GladeOS simulator outlined and utilised throughout this Thesis is by no means finalised. There are still numerous avenues to improve the simulator or extend the capabilities to support further research. Some examples of these avenues include:

1. GPU support for quantum gates, this could be accomplished through either:
    - CUDA Compute Unified Device Architecture, or

- OpenCL (Open Computing Language)

2. Enable multiple networked instances (simulation of multiple quantum machines)

3. Enable support for multithreaded scheduling (improve execution speed by removing unnecessary bottlenecks)

Currently all development has been built to run on the processor and memory of the host system directly. In order to alleviate some of the burden and increase the processing capabilities of hosts an extension into GPU calculation is required. Use of the GPU would allow the host to offload matrix operations and process multiple qubits simultaneously. This extension is designed to work with CUDA (Computer Unified Device Architecture) and OpenCL (Open Computing Language) thereby allowing the extension to use most GPU systems.

The current system is designed to utilise a single quantum computer, however most computing devices today take advantage of multiple processors to execute programs faster and more efficiently. Replicating this behaviour to utilise multiple quantum computers is the next major update to the system. This has been left out of the original version described in this Thesis in order to simplify the original testing. This update also requires significant progress in the field of quantum computing and for a networking standard to be defined before it can be completed.

10.8 Thesis Summary

Quantum computers are a nascent field of study which is rapidly making strides in both hardware and software areas. At the beginning of the Thesis 3 Research questions were posed:

RQ1 Can a quantum computer support processing of multiple programs concurrently?

RQ2 Is there a framework which can be implemented on a quantum computer to support processing of multiple programs concurrently?

RQ3 What is the cost and/or effect of adopting this framework on a quantum computer? The cost and effect are considered in terms of:

RQ3A Implementation,

RQ3B Performance,

RQ3C Algorithmic Complexity,

RQ3D Versatility/portability,

RQ3E Security

These questions were answered throughout the Thesis with each Chapter building on the preceding Chapters to provide answers. These answers were consistently positive with Research Question 1 and 2 being demonstrated in Chapter 4 and the analysis being broken over the remaining Chapters.

During the review of available literature it was noted that the concept of managing the operation of quantum programs was underdeveloped with a specific focus on:

1. Multi-processing
2. Networking between quantum computers
3. Security from cyber threats

and whether the above features were feasible and if they could make a noticeable effect compared to the existing approaches. These identified gaps closely mirror the Research Questions and in answering those this Thesis has provided contributions to these gaps as well.

It was recognised that the existing approaches of single program execution worked and were a good case for testing and evaluating the hardware implementation. It was further recognised that the information required for

sequential execution including program dependencies, program mappings and qubit connectivities which can also be used to extend the functionality of the system. It was also recognised that there is significant discussion around the preferred quantum hardware implementation and the system was therefore designed to work with a hardware agnostic approach so as to be applicable to whichever hardware was eventually crowned the standard.

The information found in the program dependencies, program mappings and qubit connectivities was combined in an intricate web of graph data structures. This novel graph structure allowed for the system to determine multi-processing opportunities which could then be actioned by the scheduling system.

It was found that the existing quantum simulators (Qiskit, Q# and Rigetti Forest) suffered from a fundamental design flaw in that they only allowed for a single quantum program to be executed at once. Due to this flaw a new quantum simulator (GladeOS) was designed and implemented which supported multi-processing and was also designed to allow for greater flexibility regarding the 30 qubit limit for most simulators. This flexibility required the development of some extended math to determine exactly how to apply the quantum gates to the relevant quantum bits.

Using the new GladeOS simulator, a series of bottlenecks were evaluated with the aim of recommending algorithms to best suit the needs of the user. Following this evaluation, GladeOS was compared and contrasted against the aforementioned simulators from Microsoft, IBM and Rigetti and returned surprising results. Figure 6.20 demonstrated that the GladeOS system outperformed the other commercial systems (by an order of magnitude in some cases). These results were further encouraged after testing the individual gates presented in the system and every gate returning the expected distributions.

With positive results from the analysis section the Thesis continued onto

reviewing possible updates and the effect they would have on the system. This began with a review of serious bottlenecks and possible approaches to resolve them. It was discovered that some of these bottlenecks are solvable, while others are equivalent to NP-hard problems and a simple solution is therefore not expected. This discussion then pivoted to consider the various alternative configurations for quantum computers and traditional computers which are all supported by the designed system.

The final component to this Thesis is the cyber security analysis which considered a wide variety of potential threat actors and scenarios. These vulnerabilities were then categorised according to likelihood and danger. Most of the vulnerabilities that were discovered were able to be patched in principle, though a few are still outstanding.

10.9 Conclusion

Overall, the research presented in this Thesis is intended to be as complete of a system as possible. The research enables interested parties to implement this parallel quantum computer system with minimal issues. The influx of researchers into this area [159]–[162] speaks to the benefit of this research and approach. Currently, one could be forgiven for thinking that this research area is not strictly necessary for current stage quantum devices. However, as the quantum systems continue to mature the number of qubits will exceed the requirements of the algorithms which will then strictly require a system similar to the one described throughout this Thesis.

11. REFERENCES

[1] L. Madden, M. Dusay, K. MacLane, and W. Martindale, *1999 A.D.* Short, Sci-Fi, IMDb ID: tt6294456 event-location: United States.

[2] S. Bae, "Big-O Notation," in *JavaScript Data Structures and Algorithms: An Introduction to Understanding and Implementing Core Data Structure and Algorithm Fundamentals*, S. Bae, Ed., Berkeley, CA: Apress, 2019, pp. 1–11, ISBN: 978-1-4842-3988-9. DOI: 10.1007/978-1-4842-3988-9_1. [Online]. Available: https://doi.org/10.1007/978-1-4842-3988-9_1.

[3] W. Heisenberg, *Physics and beyond: encounters and conversations* (World perspectives no. 23), engger. London: G. Allen & Unwin, 1971, ISBN: 978-0-04-925020-8.

[4] W. Heisenberg, *Physics & philosophy : the revolution in modern science*, eng. Harmondsworth: Penguin, 1989, ISBN: 0-14-022859-4.

[5] R. P. Feynman, "Simulating physics with computers," en, *International Journal of Theoretical Physics*, vol. 21, no. 6-7, pp. 467–488, Jun. 1982, ISSN: 0020-7748, 1572-9575. DOI: 10.1007/BF02650179. [Online]. Available: http://link.springer.com/10.1007/BF02650179 (visited on 06/22/2020).

[6] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997, arXiv: quant-ph/9508027, ISSN: 0097-5397, 1095-7111. DOI: 10.1137/S0097539795293172. [Online]. Available: http://arxiv.org/abs/quant-ph/9508027 (visited on 04/12/2017).

[7] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, Nov. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[8] E. Gibney, "Quantum gold rush: The private funding pouring into quantum start-ups," en, *Nature*, vol. 574, no. 7776, pp. 22–24, Oct. 2019, Number: 7776 Publisher: Nature Publishing Group. DOI: 10.1038/d41586-019-02935-4. [Online]. Available: https://www.nature.com/articles/d41586-019-02935-4 (visited on 04/07/2021).

[9] CSIRO, *Growing Australia's Quantum Technology Industry - Positioning Australia for a four billion-dollar opportunity*, English, May 2020. [Online]. Available: https://www.csiro.au/en/work-with-us/services/consultancy-strategic-advice-services/csiro-futures/futures-reports/quantum (visited on 07/04/2021).

[10] I. Tzinis, *Technology Readiness Level*, en, Publisher: Brian Dunbar, May 2015. [Online]. Available: http://www.nasa.gov/directorates/heo/scan/engineering/technology/technology_readiness_level (visited on 04/25/2023).

[11] *What are Technology Readiness Levels (TRL)?* en-GB. [Online]. Available: https://www.twi-global.com/technical-knowledge/faqs/technology-readiness-levels.aspx (visited on 04/25/2023).

[12] C. Boyer, *The 360 Revolution*. English, Publisher: Citeseer, 2004. [Online]. Available: https://www.vm.ibm.com/history/360rev.pdf (visited on 10/17/2023).

[13] *How To Program Your Very Own Operating Systems (OS)*, en-US, Feb. 2018. [Online]. Available: https://www.whoishostingthis.com/resources/os-development/ (visited on 05/17/2019).

[14] MICROSOFT CORPORATION, "FORM 10-K," SEC, United States Securities and Exchange Commision form 10-K 001-37845, Jul. 2022. [Online]. Available: https://www.sec.gov/Archives/edgar/data/

789019/000156459022026876/msft-10k_20220630.htm (visited on 10/12/2023).

[15] K. Franek, *Microsoft Revenue Breakdown by Product, Segment and Country - KAMIL FRANEK Business Analytics*, en, Sep. 2022. [Online]. Available: https://www.kamilfranek.com/microsoft-revenue-breakdown/ (visited on 10/12/2023).

[16] D. P. DiVincenzo and IBM, "The Physical Implementation of Quantum Computation," *Fortschritte der Physik*, vol. 48, no. 9-11, pp. 771–783, Sep. 2000, arXiv: quant-ph/0002077, ISSN: 00158208, 15213978. DOI: 10.1002/1521-3978(200009)48:9/11<771::AID-PROP771>3.0.CO;2-E. [Online]. Available: http://arxiv.org/abs/quant-ph/0002077 (visited on 04/22/2017).

[17] D. Deutsch and R. Penrose, "Quantum theory, the Church–Turing principle and the universal quantum computer," *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, vol. 400, no. 1818, pp. 97–117, Jan. 1997, Publisher: Royal Society. DOI: 10.1098/rspa.1985.0070. [Online]. Available: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1985.0070 (visited on 09/29/2023).

[18] I. M. Georgescu, S. Ashhab, and F. Nori, "Quantum simulation," *Rev. Mod. Phys.*, vol. 86, pp. 153–185, 1 Mar. 2014. DOI: 10.1103/RevModPhys.86.153. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.86.153.

[19] H.-S. Zhong, H. Wang, Y.-H. Deng, *et al.*, "Quantum computational advantage using photons," *Science*, vol. 370, no. 6523, pp. 1460–1463, 2020. DOI: 10.1126/science.abe8770. eprint: https://www.science.org/doi/pdf/10.1126/science.abe8770. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.abe8770.

[20] P. Benioff, "Quantum mechanical hamiltonian models of turing machines," en, *Journal of Statistical Physics*, vol. 29, no. 3, pp. 515–

546, Nov. 1982, ISSN: 1572-9613. DOI: 10.1007/BF01342185. [Online]. Available: https://doi.org/10.1007/BF01342185 (visited on 09/29/2023).

[21] M. Brooks, "Beyond quantum supremacy: The hunt for useful quantum computers," en, *Nature*, vol. 574, no. 7776, pp. 19–21, Oct. 2019, Bandiera_abtest: a Cg_type: News Feature Number: 7776 Publisher: Nature Publishing Group Subject_term: Chemistry, Computer science, Quantum physics, Quantum information. DOI: 10.1038/d41586-019-02936-3. [Online]. Available: https://www.nature.com/articles/d41586-019-02936-3 (visited on 11/29/2021).

[22] J. Preskill, "Quantum Computing in the NISQ era and beyond," en, *Quantum*, vol. 2, p. 79, Aug. 2018, arXiv: 1801.00862, ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79. [Online]. Available: http://arxiv.org/abs/1801.00862 (visited on 11/12/2020).

[23] R. P. Feynman, "Quantum Mechanical Computers," en, *Optics News*, vol. 11, no. 2, p. 11, Feb. 1985, ISSN: 0098-907X. DOI: 10.1364/ON.11.2.000011. [Online]. Available: https://www.osapublishing.org/abstract.cfm?URI=on-11-2-11 (visited on 11/29/2021).

[24] Rigetti, *Home*, Library Catalog: rigetti.com. [Online]. Available: https://rigetti.com/ (visited on 05/27/2020).

[25] Qiskit contributors, *Qiskit: An open-source framework for quantum computing*, 2023. DOI: 10.5281/zenodo.2573505.

[26] cjgronlund, *Microsoft Quantum Documentation and Q# API Reference - Microsoft Quantum*, en-us, Library Catalog: docs.microsoft.com. [Online]. Available: https://docs.microsoft.com/en-us/quantum/ (visited on 05/27/2020).

[27] A. Barenco, C. H. Bennett, R. Cleve, *et al.*, "Elementary gates for quantum computation," *Phys. Rev. A*, vol. 52, pp. 3457–3467, 5 Nov. 1995. DOI: 10.1103/PhysRevA.52.3457. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.52.3457.

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

[29] R. Raussendorf and H. J. Briegel, "A One-Way Quantum Computer," *Physical Review Letters*, vol. 86, no. 22, pp. 5188–5191, May 2001, Publisher: American Physical Society. DOI: 10.1103/PhysRevLett.86.5188. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.86.5188 (visited on 11/29/2021).

[30] B. Apolloni, C. Carvalho, and D. de Falco, "Quantum stochastic optimization," en, *Stochastic Processes and their Applications*, vol. 33, no. 2, pp. 233–244, Dec. 1989, ISSN: 0304-4149. DOI: 10.1016/0304-4149(89)90040-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304414989900409 (visited on 11/29/2021).

[31] A. Y. Kitaev, "Fault-tolerant quantum computation by anyons," en, *Annals of Physics*, vol. 303, no. 1, pp. 2–30, Jan. 2003, ISSN: 0003-4916. DOI: 10.1016/S0003-4916(02)00018-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0003491602000180 (visited on 11/29/2021).

[32] P. Walther, K. J. Resch, T. Rudolph, *et al.*, "Experimental one-way quantum computing," en, *Nature*, vol. 434, no. 7030, pp. 169–176, Mar. 2005, Bandiera_abtest: a Cg_type: Nature Research Journals Number: 7030 Primary_atype: Research Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/nature03347. [Online]. Available: https://www.nature.com/articles/nature03347 (visited on 11/29/2021).

[33] T. Albash and D. A. Lidar, "Adiabatic quantum computation," *Rev. Mod. Phys.*, vol. 90, p. 015 002, 1 Jan. 2018. DOI: 10.1103/RevModPhys.90.015002. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.90.015002.

[34] C. S. Calude, E. Calude, and M. J. Dinneen, "Guest Column: Adiabatic Quantum Computing Challenges," en, *ACM SIGACT News*,

vol. 46, no. 1, pp. 40–61, Mar. 2015, ISSN: 0163-5700. DOI: 10.1145/2744447.2744459. [Online]. Available: https://dl.acm.org/doi/10.1145/2744447.2744459 (visited on 11/29/2021).

[35] *The Most Controversial Quantum Computer Ever Made Just Got an Upgrade*. [Online]. Available: https://futurism.com/the-most-controversial-quantum-computer-ever-made-just-got-an-upgrade (visited on 03/25/2024).

[36] E. G. magazine Nature, *D-Wave: Scientists Line Up for World's Most Controversial Quantum Computer*, en. [Online]. Available: https://www.scientificamerican.com/article/d-wave-scientists-line-up-for-world-rsquo-s-most-controversial-quantum-computer/ (visited on 03/25/2024).

[37] *D-Wave's Quantum Computing Claim Disputed Again - IEEE Spectrum*, en. [Online]. Available: https://spectrum.ieee.org/dwaves-quantum-computing-claim-disputed (visited on 03/25/2024).

[38] "Quantum computing star D-Wave faces cash crunch, investors say," en-CA, *The Globe and Mail*, Feb. 2023. [Online]. Available: https://www.theglobeandmail.com/business/article-d-wave-infinity-machine-cash-crunch/ (visited on 03/25/2024).

[39] G. Popkin, "Quest for qubits," en, *Science*, vol. 354, no. 6316, pp. 1090–1093, Dec. 2016, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.354.6316.1090. [Online]. Available: http://www.sciencemag.org/cgi/doi/10.1126/science.354.6316.1090 (visited on 08/23/2018).

[40] H. Zhang, C.-X. Liu, S. Gazibegovic, *et al.*, "RETRACTED ARTICLE: Quantized Majorana conductance," en, *Nature*, vol. 556, no. 7699, pp. 74–79, Apr. 2018, Number: 7699 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/nature26142. [Online]. Available: https://www.nature.com/articles/nature26142 (visited on 03/16/2021).

[41] E. Farhi, J. Goldstone, S. Gutmann, J. Lapan, A. Lundgren, and D. Preda, "A Quantum Adiabatic Evolution Algorithm Applied to Random Instances of an NP-Complete Problem," *Science*, vol. 292, no. 5516, pp. 472–475, Apr. 2001, Publisher: American Association for the Advancement of Science. DOI: 10.1126/science.1057726. [Online]. Available: https://www.science.org/doi/10.1126/science.1057726 (visited on 09/12/2023).

[42] W. van Dam, M. Mosca, and U. V. Vazirani, "How powerful is adiabatic quantum computation?" *Proceedings 2001 IEEE International Conference on Cluster Computing*, pp. 279–287, 2001.

[43] N. S. Yanofsky and M. A. Mannucci, *Quantum computing for computer scientists*. Cambridge: New York : Cambridge University Press, 2008, OCLC: ocn212859032, ISBN: 978-0-521-87996-5.

[44] C. Padurariu and Y. V. Nazarov, "Theoretical proposal for superconducting spin qubits," *Physical Review B*, vol. 81, no. 14, p. 144 519, Apr. 2010, Publisher: American Physical Society. DOI: 10.1103/PhysRevB.81.144519. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.81.144519 (visited on 11/29/2021).

[45] J. I. Cirac and P. Zoller, "Quantum Computations with Cold Trapped Ions," *Physical Review Letters*, vol. 74, no. 20, pp. 4091–4094, May 1995, Publisher: American Physical Society. DOI: 10.1103/PhysRevLett.74.4091. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.74.4091 (visited on 11/29/2021).

[46] M. A. Reed, "Quantum Dots," *Scientific American*, vol. 268, no. 1, pp. 118–123, 1993, Publisher: Scientific American, a division of Nature America, Inc., ISSN: 0036-8733. [Online]. Available: https://www.jstor.org/stable/24941343 (visited on 11/29/2021).

[47] A. Gruber, A. Dräbenstedt, C. Tietz, L. Fleury, J. Wrachtrup, and C. v. Borczyskowski, "Scanning Confocal Optical Microscopy and Magnetic Resonance on Single Defect Centers," *Science*, vol. 276, no. 5321, pp. 2012–2014, Jun. 1997, Publisher: American Association

for the Advancement of Science. DOI: 10.1126/science.276.5321.2012. [Online]. Available: https://www.science.org/doi/10.1126/science.276.5321.2012 (visited on 11/29/2021).

[48] E. Gibney, "Quantum physics: Flawed to perfection," en, *Nature*, vol. 505, no. 7484, pp. 472–474, Jan. 2014, ISSN: 1476-4687. DOI: 10.1038/505472a. [Online]. Available: https://www.nature.com/articles/505472a (visited on 11/29/2021).

[49] B. Field and T. Simula, "Introduction to topological quantum computation with non-Abelian anyons," en, *Quantum Science and Technology*, vol. 3, no. 4, p. 045 004, Jul. 2018, Publisher: IOP Publishing, ISSN: 2058-9565. DOI: 10.1088/2058-9565/aacad2. [Online]. Available: https://dx.doi.org/10.1088/2058-9565/aacad2 (visited on 03/25/2024).

[50] J. M. Elzerman, R. Hanson, L. H. Willems van Beveren, B. Witkamp, L. M. K. Vandersypen, and L. P. Kouwenhoven, "Single-shot read-out of an individual electron spin in a quantum dot," en, *Nature*, vol. 430, no. 6998, pp. 431–435, Jul. 2004, Number: 6998 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/nature02693. [Online]. Available: https://www.nature.com/articles/nature02693 (visited on 09/15/2023).

[51] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*, eng, 8. ed., international. student version. Hoboken, NJ: Wiley, 2010, OCLC: 465175615, ISBN: 978-0-470-23399-3.

[52] P. Kaye, R. Laflamme, and M. Mosca, *An Introduction to Quantum Computing*, eng. Oxford: Oxford Univ. Press, 2007, OCLC: 255412511, ISBN: 978-0-19-857049-3 978-0-19-857000-4.

[53] J. Preskill, "Reliable quantum computers," en, *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 454, no. 1969, pp. 385–410, Jan. 1998, ISSN: 1364-5021, 1471-2946. DOI: 10.1098/rspa.1998.0167. [Online].

Available: https://royalsocietypublishing.org/doi/10.1098/rspa.1998.0167 (visited on 10/01/2021).

[54] R. Van Meter, *Quantum networking*. Hoboken, NJ: ISTE Ltd/John Wiley and Sons Inc, 2014, ISBN: 978-1-84821-537-5.

[55] M. Ying, *Foundations of quantum programming*, 1st edition. Cambridge, MA: Elsevier, 2016, ISBN: 978-0-12-802306-8.

[56] D. Gosset and J. Smolin, "A Compressed Classical Description of Quantum States," in *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*, W. van Dam and L. Mancinska, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 135, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 8:1–8:9, ISBN: 978-3-95977-112-2. DOI: 10.4230/LIPIcs.TQC.2019.8. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10400.

[57] M. P. Frank, U. H. Meyer-Baese, I. Chiorescu, L. Oniciuc, and R. A. van Engelen, "Space-efficient simulation of quantum computers," en, in *Proceedings of the 47th Annual Southeast Regional Conference on - ACM-SE 47*, Clemson, South Carolina: ACM Press, 2009, p. 1, ISBN: 978-1-60558-421-8. DOI: 10.1145/1566445.1566554. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1566445.1566554 (visited on 05/28/2019).

[58] D. Bohm, "A Suggested Interpretation of the Quantum Theory in Terms of "Hidden" Variables. I," *Physical Review*, vol. 85, no. 2, pp. 166–179, Jan. 1952, Publisher: American Physical Society. DOI: 10.1103/PhysRev.85.166. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRev.85.166 (visited on 10/02/2023).

[59] J. v. Leeuwen, Ed., *Handbook of theoretical computer science*. Amsterdam ; New York : Cambridge, Mass: Elsevier ; MIT Press, 1990, ISBN: 978-0-444-88075-8 978-0-444-88071-0 978-0-444-88074-1 978-0-262-22040-8 978-0-262-22038-5 978-0-262-22039-2.

[60] E. Knill, "Conventions for quantum pseudocode," English, Los Alamos National Lab. (LANL), Los Alamos, NM (United States), Tech. Rep. LA-UR-96-2724, Jun. 1996. DOI: 10.2172/366453. [Online]. Available: https://www.osti.gov/biblio/366453 (visited on 08/31/2023).

[61] S. Liu, X. Wang, L. Zhou, *et al.*, "$Q|SI\rangle$: A Quantum Programming Environment," in *Symposium on Real-Time and Hybrid Systems: Essays Dedicated to Professor Chaochen Zhou on the Occasion of His 80th Birthday*, C. Jones, J. Wang, and N. Zhan, Eds., Cham: Springer International Publishing, 2018, pp. 133–164, ISBN: 978-3-030-01461-2. DOI: 10.1007/978-3-030-01461-2_8. [Online]. Available: https://doi.org/10.1007/978-3-030-01461-2_8.

[62] *Overview | University of Technology Sydney*. [Online]. Available: https://www.uts.edu.au/research-and-teaching/our-research/centre-quantum-software-and-information/research/overview (visited on 05/06/2019).

[63] IBM, *IBM Quantum Experience*, en, Dec. 2020. [Online]. Available: https://quantum-computing.ibm.com (visited on 02/12/2020).

[64] QuantumWriter, *The Q# Programming Language*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/quantum/quantum-qr-intro (visited on 09/04/2018).

[65] D. Deutsch, "Quantum computational networks," en, *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, vol. 425, no. 1868, pp. 73–90, Sep. 1989, ISSN: 0080-4630. DOI: 10.1098/rspa.1989.0099. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rspa.1989.0099 (visited on 02/24/2021).

[66] T. Gaddis, *Starting out with Python*, eng, 4th edition, global edition. New York, NY: Pearson, 2019, ISBN: 978-1-292-22575-3.

[67] Prof Andrea Morello, Dr Vikram Sharma, Prof Steven Rolston, Dr Paolo Bianco, Dr Mark Clampin, and Katherine Bennell, *Quantum engineering and communication technology for space based applications*, English, Adelaide Convention Centre, Mar. 2021. [Online]. Avail-

able: https://www.youtube.com/watch?v=SiFR19UpFOk (visited on 11/06/2021).

[68] H. Ramesh and V. Vinay, *String Matching in $\tilde{O}(\sqrt{n}+\sqrt{m})$ Quantum Time*, Nov. 2000. DOI: 10.48550/arXiv.quant-ph/0011049. [Online]. Available: http://arxiv.org/abs/quant-ph/0011049 (visited on 08/31/2023).

[69] L. K. Grover, "A fast quantum mechanical algorithm for database search," en, in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*, Philadelphia, Pennsylvania, United States: ACM Press, 1996, pp. 212–219, ISBN: 978-0-89791-785-8. DOI: 10.1145/237814.237866. [Online]. Available: http://portal.acm.org/citation.cfm?doid=237814.237866 (visited on 09/03/2018).

[70] J. Roffe, "Quantum error correction: An introductory guide," *Contemporary Physics*, vol. 60, no. 3, pp. 226–245, Jul. 2019, Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/00107514.2019.1667078, ISSN: 0010-7514. DOI: 10.1080/00107514.2019.1667078. [Online]. Available: https://doi.org/10.1080/00107514.2019.1667078 (visited on 03/18/2024).

[71] S. J. Devitt, W. J. Munro, and K. Nemoto, "Quantum error correction for beginners," en, *Reports on Progress in Physics*, vol. 76, no. 7, p. 076 001, Jun. 2013, Publisher: IOP Publishing, ISSN: 0034-4885. DOI: 10.1088/0034-4885/76/7/076001. [Online]. Available: https://dx.doi.org/10.1088/0034-4885/76/7/076001 (visited on 03/18/2024).

[72] W. Stallings, *Operating systems: internals and design principles* (Pearson international edition), eng, 6. ed., internat. ed. Upper Saddle River, NJ: Pearson/Prentice Hall, 2009, OCLC: 255116549, ISBN: 978-0-13-603337-0.

[73] A. S. Tanenbaum, *Modern operating systems*, eng, 3. ed., international ed. Upper Saddle River, NJ: Pearson/Prentice Hall, 2009, OCLC: 254320777, ISBN: 978-0-13-813459-4.

[74] G. J. Nutt, *Operating systems*, eng, International ed., 3. ed. Boston, Mass.: Pearson Addison-Wesley, 2004, OCLC: 248877291, ISBN: 978-0-321-18955-4 978-0-201-77344-6.

[75] J. Metzner, "Structuring operating systems literature for the graduate course," en, *ACM SIGOPS Operating Systems Review*, vol. 16, no. 4, pp. 10–25, Oct. 1982, ISSN: 01635980. DOI: 10.1145/850726.850728. [Online]. Available: http://portal.acm.org/citation.cfm?doid=850726.850728 (visited on 06/21/2019).

[76] *Apple unleashes M1*, en-US. [Online]. Available: https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/ (visited on 10/08/2023).

[77] A. M. Lister and R. D. Eager, *Fundamentals of operating systems* (Macmillan computer science series), eng, 4. ed., repr. London: MacMillan, 1990, ISBN: 978-0-333-46986-6 978-0-333-46987-3.

[78] L. Lamport, "On interprocess communication," en, *Distributed Computing*, p. 16, 1986.

[79] P. J. Denning, "The Science of Computing: The ARPANET after Twenty Years," *American Scientist*, vol. 77, no. 6, pp. 530–534, 1989, Publisher: Sigma Xi, The Scientific Research Society, ISSN: 0003-0996. [Online]. Available: https://www.jstor.org/stable/27856002 (visited on 11/30/2021).

[80] J. M. Johansson, "On the impact of network latency on distributed systems design," en, *Information Technology and Management*, p. 12, 2000.

[81] Garcia-Molina, "Elections in a Distributed Computing System," *IEEE Transactions on Computers*, vol. C-31, no. 1, pp. 48–59, Jan. 1982, Conference Name: IEEE Transactions on Computers, ISSN: 1557-9956. DOI: 10.1109/TC.1982.1675885.

[82] J. P. Anderson, "Computer Security Threat Monitoring and Surveillance," en, National Institute of Standards and Technology, Technical 79F296400, 1980, p. 56.

[83] A. Ash-Saki, M. Alam, and S. Ghosh, "Experimental characterization, modeling, and analysis of crosstalk in a quantum computer," *IEEE Transactions on Quantum Engineering*, vol. 1, pp. 1–6, 2020. DOI: 10.1109/TQE.2020.3023338.

[84] H. Corrigan-Gibbs, D. J. Wu, and D. Boneh, "Quantum Operating Systems," en, in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems - HotOS '17*, Whistler, BC, Canada: ACM Press, 2017, pp. 76–81, ISBN: 978-1-4503-5068-6. DOI: 10.1145/3102980.3102993. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3102980.3102993 (visited on 11/29/2018).

[85] H. Weier, H. Krauss, M. Rau, M. Fürst, S. Nauerth, and H. Weinfurter, "Quantum eavesdropping without interception: An attack exploiting the dead time of single-photon detectors," *New Journal of Physics*, vol. 13, no. 7, p. 073 024, Jul. 2011, ISSN: 1367-2630. DOI: 10.1088/1367-2630/13/7/073024. [Online]. Available: http://stacks.iop.org/1367-2630/13/i=7/a=073024?key=crossref.9a50411e96e60e52c44860284e8d16a7 (visited on 06/15/2017).

[86] B. J. Oates, *Researching information systems and computing*. London ; Thousand Oaks, Calif: SAGE Publications, 2006, ISBN: 978-1-4129-0223-6 978-1-4129-0224-3.

[87] G. Shanks, Anne Rouse, and David Arnott, *A review of approaches to research and scholarship in information systems* (Working paper series (Monash University. Dept. of Information Systems) ; 93/3 1993), eng. Caulfield East, Vic.: Dept. of Information Systems, Faculty of Computing and Information Technology, Monash University, Apr. 1993. (visited on 03/05/2021).

[88] P. Williams, "Making Research Real: Is Action Research a Suitable Methodology for Medical Information Security Investigations?" *Australian Information Security Management Conference*, Jan. 2006.

[89] W. L. Neuman and L. Kreuger, *Social work research methods: qualitative and quantitative approaches*, eng, 1st ed. Boston: Allyn and Bacon, 2003, OCLC: 611256892, ISBN: 978-0-205-29914-0.

[90] M. Gounelle, "The meteorite fall at L'Aigle and the Biot report: Exploring the cradle of meteoritics," en, *Geological Society, London, Special Publications*, vol. 256, no. 1, pp. 73–89, 2006, ISSN: 0305-8719, 2041-4927. DOI: 10.1144/GSL.SP.2006.256.01.03. [Online]. Available: http://sp.lyellcollection.org/lookup/doi/10.1144/GSL.SP.2006.256.01.03 (visited on 03/15/2021).

[91] R. Honan, T. W. Lewis, S. Anderson, and J. Cooke, "A quantum computer operating system," in *International Conference on Algorithms and Architectures for Parallel Processing*, Springer, 2020, pp. 415–431.

[92] D-Wave, *The D-Wave 2000q Quantum Computer Technology Overview*, English. (visited on 05/27/2019).

[93] G. Kalai, "How Quantum Computers Fail: Quantum Codes, Correlations in Physical Systems, and Noise Accumulation," en, *ArXiv*, p. 17, 2011.

[94] C. Lee, *Bigger is better: Quantum volume expresses computer's limit*, May 2017. [Online]. Available: https://arstechnica.com/science/2017/05/quantum-volume-one-number-to-benchmark-a-quantum-computer/ (visited on 05/31/2017).

[95] S. K. Moore, *IBM Edges Closer to Quantum Supremacy with 50-Qubit Processor - IEEE Spectrum*. [Online]. Available: https://spectrum.ieee.org/tech-talk/computing/hardware/ibm-edges-closer-to-quantum-supremacy-with-50qubit-processor (visited on 07/05/2019).

[96] J. Proos and C. Zalka, "Shor's discrete logarithm quantum algorithm for elliptic curves," *arXiv:quant-ph/0301141*, Jan. 2003, arXiv: quant-ph/0301141. [Online]. Available: http://arxiv.org/abs/quant-ph/0301141.

[97] Matthew Treinish, *"Building a Compiler for Quantum Computers" - Matthew Treinish (LCA 2020)*, English, Conference Presentation, Gold Coast, Australia, Jan. 2020. [Online]. Available: https://www.youtube.com/watch?v=L2P501Iy6J8 (visited on 12/02/2020).

[98] E. Pius, "Automatic Parallelisation of Quantum Circuits Using the Measurement Based Quantum Computing Model," en, *ArXiv*, p. 73, 2010.

[99] A. Javadi-Abhari, P. Gokhale, A. Holmes, *et al.*, "Optimized Surface Code Communication in Superconducting Quantum Computers," *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-50 '17*, pp. 692–705, 2017, arXiv: 1708.09283. DOI: 10.1145/3123939.3123949. [Online]. Available: http://arxiv.org/abs/1708.09283 (visited on 05/17/2020).

[100] C.-C. Lin, S. Sur-Kolay, and N. K. Jha, "PAQCS: Physical Design-Aware Fault-Tolerant Quantum Circuit Synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 7, pp. 1221–1234, Jul. 2015, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2014.2337302.

[101] D. Maslov, S. M. Falconer, and M. Mosca, "Quantum Circuit Placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 4, pp. 752–763, Apr. 2008, arXiv: quant-ph/0703256, ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2008.917562. [Online]. Available: http://arxiv.org/abs/quant-ph/0703256 (visited on 05/17/2020).

[102] M. Pedram and A. Shafaei, "Layout Optimization for Quantum Circuits with Linear Nearest Neighbor Architectures," en, *IEEE Circuits and Systems Magazine*, vol. 16, no. 2, pp. 62–74, 2016, ISSN:

1531-636X. DOI: 10.1109/MCAS.2016.2549950. [Online]. Available: http://ieeexplore.ieee.org/document/7476978/ (visited on 05/17/2020).

[103] A. Shafaei, M. Saeedi, and M. Pedram, "Qubit placement to minimize communication overhead in 2D quantum architectures," en, in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Singapore: IEEE, Jan. 2014, pp. 495–500, ISBN: 978-1-4799-2816-3. DOI: 10.1109/ASPDAC.2014.6742940. [Online]. Available: http://ieeexplore.ieee.org/document/6742940/ (visited on 05/17/2020).

[104] M. Y. Siraichi, V. F. d. Santos, C. Collange, and F. M. Q. Pereira, "Qubit allocation as a combination of subgraph isomorphism and token swapping," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, 120:1–120:29, Oct. 2019. DOI: 10.1145/3360546. [Online]. Available: https://doi.org/10.1145/3360546 (visited on 02/10/2020).

[105] T. M. Connolly and C. E. Begg, *Database systems: a practical approach to design, implementation, and management* (Always learning), eng, 6. ed., global ed. Boston, Mass.: Pearson, 2015, OCLC: 894742602, ISBN: 978-1-292-06118-4 978-0-13-294326-0.

[106] J. Golbeck, *Analyzing the social web*, First edition. Waltham, MA: Morgan Kaufmann is an imprint of Elsevier, 2013, ISBN: 978-0-12-405531-5.

[107] MITRE, *CWE - CWE-416: Use After Free (4.0)*. [Online]. Available: https://cwe.mitre.org/data/definitions/416.html (visited on 05/18/2020).

[108] MITRE, *CWE - CWE-200: Exposure of Sensitive Information to an Unauthorized Actor (4.0)*. [Online]. Available: https://cwe.mitre.org/data/definitions/200.html (visited on 05/18/2020).

[109] R. Pula, F. I. Khan, B. Veitch, and P. R. Amyotte, "A Grid Based Approach for Fire and Explosion Consequence Analysis," en, *Process Safety and Environmental Protection*, vol. 84, no. 2, pp. 79–91, Mar. 2006, ISSN: 0957-5820. DOI: 10.1205/psep.05063. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957582006713088 (visited on 11/29/2021).

[110] V. N. Alekhin, A. A. Antipin, S. N. Gorodilov, L. I. Avdonina, and A. M. Budarin, "Computer simulation-based analysis of wind load effect on structural capacity of skywalk between Hyatt Regency hotel and multipurpose building complex Iset Tower in Ekaterinburg," en, *IOP Conference Series: Materials Science and Engineering*, vol. 456, p. 012 009, Dec. 2018, Publisher: IOP Publishing, ISSN: 1757-899X. DOI: 10.1088/1757-899X/456/1/012009. [Online]. Available: https://doi.org/10.1088/1757-899x/456/1/012009 (visited on 11/29/2021).

[111] E. Kreyszig, *Advanced engineering mathematics*, eng, 8. ed. New York: John Wiley, 1999, ISBN: 978-0-471-33328-9 978-0-471-15496-9.

[112] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, Jan. 1998, ISSN: 1049-3301. DOI: 10.1145/272991.272995. [Online]. Available: https://doi.org/10.1145/272991.272995 (visited on 11/16/2021).

[113] *Std::mersenne_twister_engine - cppreference.com*. [Online]. Available: https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine (visited on 11/16/2021).

[114] B. Gerard and M. Kong, "String Abstractions for Qubit Mapping," *arXiv:2111.03716 [quant-ph]*, Nov. 2021, arXiv: 2111.03716. [Online]. Available: http://arxiv.org/abs/2111.03716 (visited on 12/09/2021).

[115] L. Lao, B. van Wee, I. Ashraf, *et al.*, "Mapping of Lattice Surgery-based Quantum Circuits on Surface Code Architectures," *Quantum*

Science and Technology, vol. 4, no. 1, p. 015 005, Sep. 2018, arXiv: 1805.11127, ISSN: 2058-9565. DOI: 10.1088/2058-9565/aadd1a. [Online]. Available: http://arxiv.org/abs/1805.11127 (visited on 12/09/2021).

[116] R. V. Book, "Richard M. Karp. Reducibility among combinatorial problems. Complexity of computer computations, Proceedings of a Symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Center, Yorktown Heights, New York, edited by Raymond E. Miller and James W. Thatcher, Plenum Press, New York and London 1972, pp. 85–103.," en, *The Journal of Symbolic Logic*, vol. 40, no. 4, pp. 618–619, Dec. 1975, Publisher: Cambridge University Press, ISSN: 0022-4812, 1943-5886. DOI: 10.2307/2271828. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-symbolic-logic/article/abs/karprichard-m-reducibility-among-combinatorial-problems-complexity-of-computer-computations-proceedings-of-a-symposium-on-the-complexity-of-computer-computations-held-march-20-22-1972-at-the-ibm-thomas-j-watson-center-yorktown-heights-new-york-edited-by-millerraymond-e-and-thatcherjames-w-plenum-press-new-york-and-london1972-pp-85103/FD45CA83152D6CCF921B25C81DA31C39# (visited on 10/01/2023).

[117] *Cpython: 2e8b28dbc395 Lib/random.py*. [Online]. Available: https://hg.python.org/cpython/file/2e8b28dbc395/Lib/random.py#l276 (visited on 11/18/2021).

[118] R. A. Fisher, R. A. Fisher, and F. Yates, *Statistical tables for biological, agricultural and medical research*, eng, 6. ed., rev. and enlarged, reprinted. London: Longman, 1979, ISBN: 978-0-582-44525-3. [Online]. Available: https://digital.library.adelaide.edu.au/dspace/handle/2440/10701.

[119] D. E. Knuth, *The art of computer programming*, 3rd ed. Reading, Mass: Addison-Wesley, 1997, ISBN: 978-0-201-89683-1 978-0-201-89684-8 978-0-201-89685-5.

[120] *QasmSimulator — Qiskit 0.32.0 documentation*. [Online]. Available: https://qiskit.org/documentation/stubs/qiskit.providers.aer.QasmSimulator.html (visited on 11/18/2021).

[121] W. Mendenhall, J. E. Reinmuth, and R. J. Beaver, *Statistics for management and economics* (The Duxbury series in statistics and decision sciences), 6th ed. Boston: PWS-KENT Pub. Co, 1989, ISBN: 978-0-534-91658-9.

[122] M. M. Alani, "OSI Model," en, in *Guide to OSI and TCP/IP Models*, ser. SpringerBriefs in Computer Science, M. M. Alani, Ed., Cham: Springer International Publishing, 2014, pp. 5–17, ISBN: 978-3-319-05152-9. DOI: 10.1007/978-3-319-05152-9_2. [Online]. Available: https://doi.org/10.1007/978-3-319-05152-9_2 (visited on 12/10/2021).

[123] A. Levi, "Optical interconnects in systems," *Proceedings of the IEEE*, vol. 88, no. 6, pp. 750–757, Jun. 2000, Conference Name: Proceedings of the IEEE, ISSN: 1558-2256. DOI: 10.1109/5.867688.

[124] M. A. Taubenblatt, "Optical Interconnects for High-Performance Computing," *Journal of Lightwave Technology*, vol. 30, no. 4, pp. 448–457, Feb. 2012, Conference Name: Journal of Lightwave Technology, ISSN: 1558-2213. DOI: 10.1109/JLT.2011.2172989.

[125] A. S. Cacciapuoti, M. Caleffi, F. Tafuri, F. S. Cataliotti, S. Gherardini, and G. Bianchi, "Quantum Internet: Networking Challenges in Distributed Quantum Computing," *IEEE Network*, vol. 34, no. 1, pp. 137–143, Jan. 2020, Conference Name: IEEE Network, ISSN: 1558-156X. DOI: 10.1109/MNET.001.1900092. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8910635?casa_token=4eYyUz4bom0AAAAA:DDuccGSAtfMNFUoM7x3pIlybRxI2FU_-bGgh3XgXfx-qYj5TEAbQaCqoz8P2CY_hOVwt-lulzrrzAMc (visited on 10/16/2023).

[126] A. Shenoy-Hejamadi, A. Pathak, and S. Radhakrishna, "Quantum Cryptography: Key Distribution and Beyond," *Quanta*, vol. 6, no. 1, p. 1, Jun. 2017, ISSN: 1314-7374. DOI: 10.12743/quanta.v6i1.57.

[Online]. Available: http://quanta.ws/ojs/index.php/quanta/article/view/57 (visited on 06/21/2017).

[127] L. A. Lizama-Pérez, J. M. López, and E. De Carlos López, "Quantum key distribution in the presence of the intercept-resend with faked states attack," *Entropy*, vol. 19, no. 1, 2017, ISSN: 1099-4300. DOI: 10.3390/e19010004. [Online]. Available: https://www.mdpi.com/1099-4300/19/1/4.

[128] J. Ouellette, "Quantum key distribution," *Industrial Physicist*, vol. 10, no. 6, pp. 22–25, 2004. [Online]. Available: http://people.cs.vt.edu/~kafura/cs6204/Readings/QuantumX/QuantumKeyDistribution.pdf (visited on 05/31/2017).

[129] T. Miltzow, L. Narins, Y. Okamoto, G. Rote, A. Thomas, and T. Uno, "Approximation and Hardness for Token Swapping," *arXiv:1602.05150 [cs]*, Aug. 2016, arXiv: 1602.05150. DOI: 10.4230/LIPIcs.ESA.2016.185. [Online]. Available: http://arxiv.org/abs/1602.05150 (visited on 02/20/2020).

[130] K. Yamanaka, E. D. Demaine, T. Horiyama, *et al.*, "Sequentially Swapping Colored Tokens on Graphs," en, *Journal of Graph Algorithms and Applications*, vol. 23, no. 1, pp. 3–27, 2019, ISSN: 1526-1719. DOI: 10.7155/jgaa.00482. [Online]. Available: http://jgaa.info/getPaper?id=482 (visited on 02/19/2020).

[131] G. A. Seber, *A matrix handbook for statisticians*. John Wiley & Sons, 2008.

[132] L. Oliveira, *Linear algebra*, eng, First edition. Boca Raton London New York Abingdon: CRC Press, 2022, ISBN: 978-1-351-24345-2 978-1-03-228781-2 978-0-8153-7331-5. DOI: 10.1201/9781351243452.

[133] R. H. B. Netzer and B. P. Miller, "What are race conditions? Some issues and formalizations," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 1, pp. 74–88, Mar. 1992, ISSN: 1057-4514. DOI: 10.1145/130616.130623. [Online]. Available: https://doi.org/10.1145/130616.130623 (visited on 11/23/2021).

[134] A. Kumar Pati and S. L. Braunstein, "Impossibility of deleting an unknown quantum state," en, *Nature*, vol. 404, no. 6774, pp. 164–165, Mar. 2000, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/404130b0. [Online]. Available: http://www.nature.com/articles/35004532 (visited on 01/24/2019).

[135] S. S. Tannu and M. K. Qureshi, "Not All Qubits Are Created Equal: A Case for Variability-Aware Policies for NISQ-Era Quantum Computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 987–999, ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304007. [Online]. Available: https://doi.org/10.1145/3297858.3304007 (visited on 03/28/2021).

[136] J. C. Adams and T. H. Brom, "Microwulf: A beowulf cluster for every desk," en, in *Proceedings of the 39th SIGCSE technical symposium on Computer science education - SIGCSE '08*, Portland, OR, USA: ACM Press, 2008, p. 121, ISBN: 978-1-59593-799-5. DOI: 10.1145/1352135.1352178. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1352135.1352178 (visited on 03/29/2021).

[137] L. G. Roberts and B. D. Wessler, "Computer network development to achieve resource sharing," in *Proceedings of the May 5-7, 1970, spring joint computer conference*, ser. AFIPS '70 (Spring), New York, NY, USA: Association for Computing Machinery, May 1970, pp. 543–549, ISBN: 978-1-4503-7903-8. DOI: 10.1145/1476936.1477020. [Online]. Available: https://doi.org/10.1145/1476936.1477020 (visited on 12/06/2021).

[138] M. Kahani and H. W. P. Beadle, "Decentralised approaches for network management," en, *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 3, pp. 36–47, Jul. 1997, ISSN: 0146-4833. DOI: 10.1145/263932.263940. [Online]. Available: https://dl.acm.org/doi/10.1145/263932.263940 (visited on 12/03/2021).

[139] P. Baran, "On Distributed Communications: XI. Summary Overview," *The RAND Cor*, 1964.

[140] S. Barz, E. Kashefi, A. Broadbent, J. F. Fitzsimons, A. Zeilinger, and P. Walther, "Experimental Demonstration of Blind Quantum Computing," *Science*, vol. 335, no. 6066, pp. 303–308, Jan. 2012, arXiv: 1110.1381, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.1214707. [Online]. Available: http://arxiv.org/abs/1110.1381 (visited on 08/29/2018).

[141] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck, "Blockchain," en, *Business & Information Systems Engineering*, vol. 59, no. 3, pp. 183–187, Jun. 2017, ISSN: 1867-0202. DOI: 10.1007/s12599-017-0467-3. [Online]. Available: https://doi.org/10.1007/s12599-017-0467-3 (visited on 10/17/2023).

[142] E. Lucero, R. Barends, Y. Chen, *et al.*, "Computing prime factors with a Josephson phase qubit quantum processor," *Nature Physics*, vol. 8, no. 10, pp. 719–723, Aug. 2012, ISSN: 1745-2473, 1745-2481. DOI: 10.1038/nphys2385. [Online]. Available: http://www.nature.com/doifinder/10.1038/nphys2385 (visited on 10/12/2017).

[143] Cybersecurity and infrastructure security agency (CISA), *Understanding Denial-of-Service Attacks | CISA*. [Online]. Available: https://www.us-cert.gov/ncas/tips/ST04-015 (visited on 05/25/2020).

[144] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2, Hilton Head, SC, USA: IEEE Comput. Soc, 1999, pp. 119–129, ISBN: 978-0-7695-0490-2. DOI: 10.1109/DISCEX.2000.821514. [Online]. Available: http://ieeexplore.ieee.org/document/821514/ (visited on 05/05/2021).

[145] M. Sikora, T. Gerlich, and L. Malina, "On detection and mitigation of slow rate denial of service attacks," in *2019 11th International Congress on Ultra Modern Telecommunications and Control Systems*

and Workshops (ICUMT), 2019, pp. 1–5. DOI: 10.1109/ICUMT48472.2019.8970844.

[146] D. Michie, ""Memo" Functions and Machine Learning," en, *Nature*, vol. 218, no. 5136, pp. 19–22, Apr. 1968, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/218019a0. [Online]. Available: https://www.nature.com/articles/218019a0 (visited on 12/03/2021).

[147] M. E. Conway, "A multiprocessor system design," in *Proceedings of the November 12-14, 1963, fall joint computer conference*, ser. AFIPS '63 (Fall), New York, NY, USA: Association for Computing Machinery, Nov. 1963, pp. 139–146, ISBN: 978-1-4503-7883-3. DOI: 10.1145/1463822.1463838. [Online]. Available: https://doi.org/10.1145/1463822.1463838 (visited on 12/02/2021).

[148] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber, "Some constraints and tradeoffs in the design of network communications," in *Proceedings of the fifth ACM symposium on Operating systems principles*, ser. SOSP '75, New York, NY, USA: Association for Computing Machinery, Nov. 1975, pp. 67–74, ISBN: 978-1-4503-7863-5. DOI: 10.1145/800213.806523. [Online]. Available: https://doi.org/10.1145/800213.806523 (visited on 11/22/2021).

[149] *Quantum Information and Applications [IQA]*, en. [Online]. Available: https://www.telecom-paris.fr/en/research/laboratories/information-processing-and-communication-laboratory-ltci/research-teams/quantum-information-applications (visited on 03/03/2023).

[150] Sheng-Tzong Cheng, Chun-Yen Wang, and Ming-Hon Tao, "Quantum communication for wireless wide-area networks," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 7, pp. 1424–1432, Jul. 2005, ISSN: 0733-8716. DOI: 10.1109/JSAC.2005.851157. [Online]. Available: http://ieeexplore.ieee.org/document/1461505/ (visited on 05/29/2017).

[151] N. Metwally, "Entanglement Routers via Wireless Quantum Network Based on Arbitrary Two Qubit Systems," *Physica Scripta*, vol. 89, no. 12, p. 125 103, Dec. 2014, arXiv: 1405.0387, ISSN: 0031-8949, 1402-4896. DOI: 10.1088/0031-8949/89/12/125103. [Online]. Available: http://arxiv.org/abs/1405.0387.

[152] M. Sarovar, T. Proctor, K. Rudinger, K. Young, E. Nielsen, and R. Blume-Kohout, "Detecting crosstalk errors in quantum information processors," en-GB, *Quantum*, vol. 4, p. 321, Sep. 2020, Publisher: Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften. DOI: 10.22331/q-2020-09-11-321. [Online]. Available: https://quantum-journal.org/papers/q-2020-09-11-321/ (visited on 12/08/2021).

[153] Daniel Gottesman, *Daniel Gottesman - Quantum Error Correction and Fault Tolerance (Part 1) - CSSQI 2012*, English, Lecture, University of Waterloo, Jun. 2012. [Online]. Available: https://www.youtube.com/watch?v=ltJ1jXQeDl8 (visited on 06/15/2021).

[154] J. M. Gertler, B. Baker, J. Li, S. Shirol, J. Koch, and C. Wang, "Protecting a bosonic qubit with autonomous quantum error correction," en, *Nature*, vol. 590, no. 7845, pp. 243–248, Feb. 2021, Number: 7845 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/s41586-021-03257-0. [Online]. Available: https://www.nature.com/articles/s41586-021-03257-0 (visited on 04/06/2021).

[155] IBM Research, *The IBM Quantum Summit Keynote with Darío Gil, Senior Vice President and Director of IBM Research*, Nov. 2021. [Online]. Available: https://www.youtube.com/watch?v=kR0y5Ps-7Kw (visited on 09/28/2023).

[156] IBM Research, *IBM Quantum 2022 Updated Development Roadmap*, May 2022. [Online]. Available: https://www.youtube.com/watch?v=0ka20qanWzI (visited on 09/28/2023).

[157] Google Quantum AI, *The Quantum Frontier*, May 2021. [Online]. Available: https://www.youtube.com/watch?v=FdlaQjYYbwo (visited on 09/28/2023).

[158] Google Quantum AI, *Our quantum computing journey*, en. [Online]. Available: https://dreamcoat-guggenheim.uc.r.appspot.com/learn/map/ (visited on 09/28/2023).

[159] P. Das, S. S. Tannu, P. J. Nair, and M. Qureshi, "A Case for Multi-Programming Quantum Computers," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52, New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 291–303, ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358287. [Online]. Available: https://doi.org/10.1145/3352460.3358287 (visited on 11/06/2022).

[160] Y. Ohkura, T. Satoh, and R. Van Meter, "Simultaneous Execution of Quantum Circuits on Current and Near-Future NISQ Systems," *IEEE Transactions on Quantum Engineering*, vol. 3, pp. 1–10, 2022, Conference Name: IEEE Transactions on Quantum Engineering, ISSN: 2689-1808. DOI: 10.1109/TQE.2022.3164716.

[161] S. Niu and A. Todri-Sanial, "How Parallel Circuit Execution Can Be Useful for NISQ Computing?" In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, ISSN: 1558-1101, Mar. 2022, pp. 1065–1070. DOI: 10.23919/DATE54114.2022.9774512.

[162] S. Deshpande, C. Xu, T. Trochatos, Y. Ding, and J. Szefer, "Towards an Antivirus for Quantum Computers," in *2022 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, Jun. 2022, pp. 37–40. DOI: 10.1109/HOST54066.2022.9840181.

Appendix A

ANALYSIS APPENDIX

The full data generation script from Chapter 6

```
import random
from datetime import datetime

GATES = ['X', 'Y', 'Z', 'T', 'S', 'C', 'R', 'RX', 'RY', 'RZ']
GATE = ['X', 'Y', 'Z', 'T', 'S', 'R', 'RX', 'RY', 'RZ']
ROTATEGATES = ['X', 'Y', 'Z']

def generate_circuit(numQubits, numGates):
    print("Generating Circuit. qubits:" + str(numQubits) + " gates: " + str(numGates))
    circuit = ''
    histogram = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    SET = random.choices(GATES, k=numGates)
    for s in SET:
        histogram[indexof(s)] = histogram[indexof(s)] + 1
        if s == 'C':
            extra = random.choice(GATE)
            histogram[indexofControlled(extra)] = histogram[indexofControlled(extra)] + 1
            target = random.randrange(numQubits)
            control = random.randrange(numQubits)
            while control == target:
```

```
21                  target = random.randrange(numQubits)
22              if extra == 'RX' or extra == 'RY' or extra
    == 'RZ':
23                  circuit = circuit + s + '([' + str(
    control) + '],' + extra + '([' + str(
24                      random.random() * random.randrange
    (360)) + '],' + str(
25                      target) + '))\n'
26              elif extra == 'R':
27                  circuit = circuit + s + '([' + str(
    control) + '],' + extra + '([' + str(
28                      random.random() * random.randrange
    (360)) + ',' + str(
29                      random.random() * random.randrange
    (360)) + ',' + str(
30                      random.random() * random.randrange
    (360)) + '],' + str(
31                      target) + '))\n'
32              else:
33                  circuit = circuit + s + '([' + str(
    control) + '],' + extra + '(' + str(
34                  target) + '))\n'
35          elif s == 'RX' or s == 'RY' or s == 'RZ':
36              circuit = circuit + s + '([' + str(
37                  random.random() * random.randrange(360))
     + '],' + str(
38                  random.randrange(numQubits)) + ')\n'
39          elif s == 'R':
40              circuit = circuit + s + '([' + str(
41                  random.random() * random.randrange(360))
     + ',' + str(
42                  random.random() * random.randrange(360))
     + ',' + str(
43                  random.random() * random.randrange(360))
     + '],' + str(
```

```
                random.randrange(numQubits)) + ')\n'
        else:
            circuit = circuit + s + '(' + str(random.
    randrange(numQubits)) + ')\n'
    circuit = circuit + 'END\n'
    print(circuit)
    return circuit, histogram


def indexof(s):
    if s == 'X':
        return 0
    if s == 'Y':
        return 1
    if s == 'Z':
        return 2
    if s == 'T':
        return 3
    if s == 'S':
        return 4
    if s == 'C':
        return 5
    if s == 'R':
        return 6
    if s == 'RX':
        return 7
    if s == 'RY':
        return 8
    if s == 'RZ':
        return 9


def indexofControlled(s):
    jumpvalue = 10
    if s == 'X':
```

```
78          return 0+jumpvalue
79      if s == 'Y':
80          return 1+jumpvalue
81      if s == 'Z':
82          return 2+jumpvalue
83      if s == 'T':
84          return 3+jumpvalue
85      if s == 'S':
86          return 4+jumpvalue
87      if s == 'R':
88          return 5+jumpvalue
89      if s == 'RX':
90          return 6+jumpvalue
91      if s == 'RY':
92          return 7+jumpvalue
93      if s == 'RZ':
94          return 8+jumpvalue
95
96
97  def ascii_histogram(data, today):
98      labels = ['X', 'Y', 'Z', 'T', 'S', 'C', 'R', 'RX', '
    RY', 'RZ', 'CX', 'CY', 'CZ', 'CT', 'CS', 'CR', 'CRX',
     'CRY', 'CRZ']
99      for i in range(len(data)):
100         # printout(labels[i] + ":\t" + '|' + ('+' * data
    [i]), today)
101         printout(labels[i] + ":(" + str(data[i]) + ")" +
     (" " * (10 - (len(str(labels[i])) + len(str(data[i]))
    ) + 3))) + "\t" + '|' + ('+' * data[i]), today)
102
103
104 def printout(output, today):
105     print(str(output))
106     fileName = "output" + str(today) + ".txt"
107     outF = open(fileName, "a+", encoding='ascii',
```

```
    newline='\r\n')
108     outF.write(str(output) + '\n')
109     outF.close()
110 
111 
112 numberFiles = 100
113 qubitArray = []
114 gateArray = []
115 histogramArray = []
116 TotalQubits = 0
117 TotalGates = 0
118 today = datetime.now()
119 for a in range(1, numberFiles+1):
120     fileName = "Controllers/generated" + str(a) + ".txt"
121     outF = open(fileName, "w", encoding='ascii', newline
    ='\r\n')
122     numbQubits = random.randrange(2, 10)
123     numbGates = random.randrange(1, 10000)
124     qubitArray.append(numbQubits)
125     gateArray.append(numbGates)
126     TotalQubits = TotalQubits + numbQubits
127     TotalGates = TotalGates + numbGates
128     CIRCUIT, histogram = generate_circuit(numbQubits,
    numbGates)
129     histogramArray.append(histogram)
130     for line in CIRCUIT:
131         # up to 10 qubits, and 20 gates.
132         # write line to output file
133         outF.write(line)
134     outF.close()
135 printout("---------------------------------", today)
136 printout("Statistics for research:", today)
137 printout("Qubit array: " + str(qubitArray), today)
138 printout("Gate array: " + str(gateArray), today)
139 printout("Average number of qubits: " + str(TotalQubits/
```

```
    numberFiles), today)
140 printout("Average number of gates: " + str(TotalGates/
    numberFiles), today)
141 printout("Number of files: " + str(numberFiles), today)
142 printout("Gate distribution: ", today)
143 printout("Key: ['X', 'Y', 'Z', 'T', 'S', 'C', 'R', 'RX',
     'RY', 'RZ', 'CX', 'CY', 'CZ', 'CT', 'CS', 'CR', 'CRX
    ', 'CRY', 'CRZ']", today)
144 histogramoverall = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0]
145 for h in histogramArray:
146     printout(str(h), today)
147 for h in histogramArray:
148     printout(str(h), today)
149     ascii_histogram(h, today)
150     for i in range(len(h)):
151         histogramoverall[i] = histogramoverall[i] + h[i]
152 printout("Overall: ", today)
153 printout(str(histogramoverall), today)
154 ascii_histogram(histogramoverall, today)
155 printout("---------------------------------", today)
```

Algorithm A.1: Random Data Files Generation Script

Appendix B

ALTERNATE GATE REPRESENTATIONS

This appendix outlines how to construct the standard digital computer gates from the minimum gate set.

B.1 Standard Logic gates

B.1.1 NOT

Figure B.1: Equivalent Quantum NOT circuit

B.1.2 AND

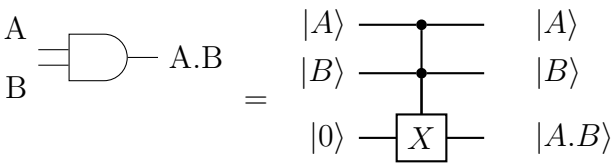


Figure B.2: Equivalent Quantum AND circuit

B.1.3 OR

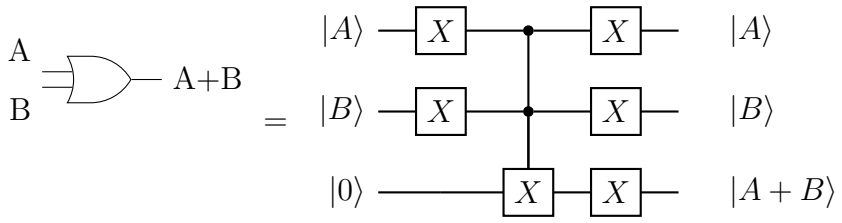


Figure B.3: Equivalent Quantum OR circuit

B.1.4 XOR

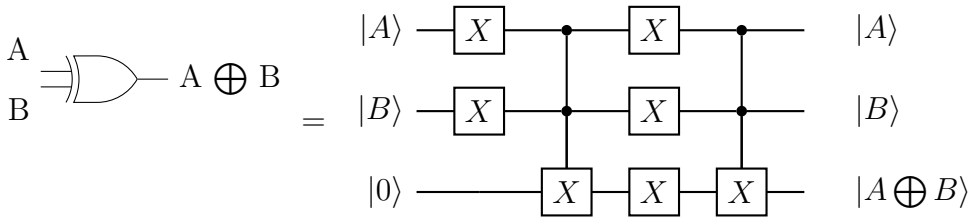


Figure B.4: Equivalent Quantum XOR circuit

B.1.5 NAND

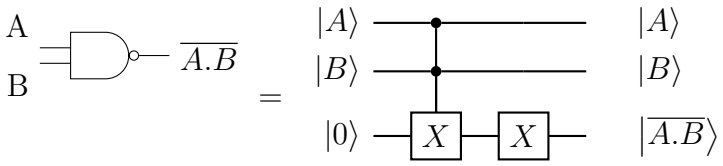


Figure B.5: Equivalent Quantum NAND circuit

B.1.6 NOR

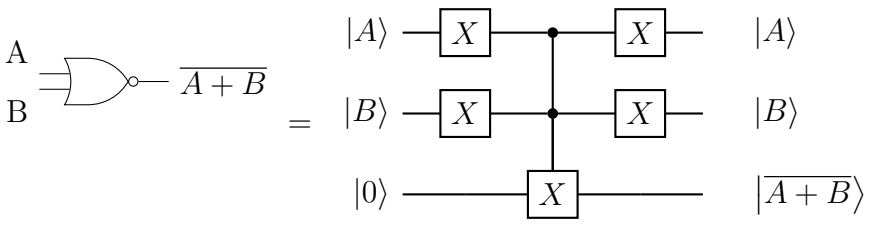


Figure B.6: Equivalent Quantum NOR circuit

B.1.7 XNOR

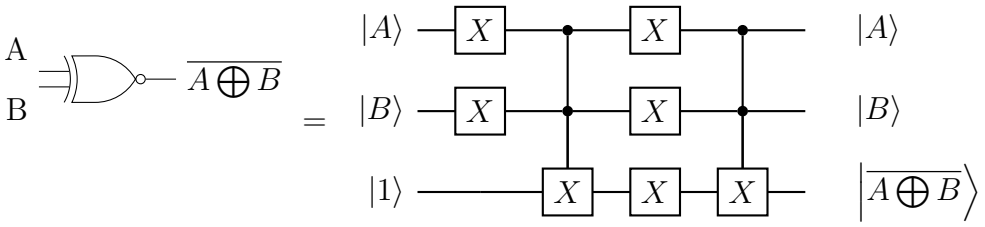


Figure B.7: Equivalent Quantum XNOR circuit