Kandidatexamen

A Review of Freely Available Quantum Computer Simulation Software

Johan Brandhorst-Satzkorn

LiTH-MAT-EX–2012/07–SE

A Review of Freely Available Quantum Computer Simulation Software

Department of Applied Mathematics, Linköpings Universitet

Johan Brandhorst-Satzkorn

LiTH-MAT-EX–2012/07–SE

Examensarbete: **16 hp**

Level: **G2**

Supervisor: **Jan-Åke Larsson**,
Department of Applied Mathematics, Linköpings Universitet

Examiner: **Jan-Åke Larsson**,
Department of Applied Mathematics, Linköpings Universitet

Linköping: **June 2012**

Abstract

A study has been made of a few different freely available Quantum Computer simulators. All the simulators tested are available online on their respective websites. A number of tests have been performed to compare the different simulators against each other. Some untested simulators of various programming languages are included to show the diversity of the quantum computer simulator applications.

The conclusion of the review is that LibQuantum is the best of the simulators tested because of ease of coding, a great amount of pre-defined function implementations and decoherence simulation support among other reasons.

Keywords: Quantum Computer Simulation, Quantum Programming Languange, Library extension, Quantum Computers.

Acknowledgements

I would like to thank my supervisor, Jan-Åke Larsson, for providing help with the structure and scope of the document.

I would also like to thank the authors of the software herein reviewed, for providing the possibility of simulating a quantum computer in a familiar environment and test their work against each other.

Contents

Chapter 1

Introduction

Chapter 1: This text is written as a bachelor of science final thesis at Linköpings University by Johan Brandhorst with Jan-Åke Larsson as supervisor and examiner, with LaTeX in 2012.

This first chapter will give a brief introduction to quantum computers and other information necessary to perform the comparisons in the review.

1.1 Background

Since the emergence of the field of Quantum Computing in the early 1980s by Richard Feynman [1] and others, there has been a wish to simulate the behavior of a quantum computer without the need to build one, as the construction of a quantum computer is both very expensive and practically complicated.

As a result of this problem there are today many different quantum computer simulators available, which are run on a classical computer. This paper will choose between them and compare a few different freely available quantum computer simulators created in the form of programming language extension libraries or stand-alone programs.

1.2 Quantum Computers

A quantum computer is similar to a normal computer in many ways; they are essentially built up of the same 3 parts: a memory, which holds the current information about the system, a processor, which performs operations on the current state of the computer, and some sort of input/output port where information can be put into the computer and a result can be extracted.

The main difference between a normal computer and a quantum computer is that quantum computers use a qubit to store the current state of the system as opposed to the normal bit. A qubit is different for a few different reasons, one of them is that whereas a classical bit is either 0 or 1, a qubit can be 0 or 1 or a superposition of both. The ket notation is often used to describe the state of a qubit:

$$A = |0\rangle$$

This describes the qubit A with a value of 0. This state corresponds to the classical bit state 0. An example of a superpositioned qubit B is shown below:

$$B = \alpha|0\rangle + \beta|1\rangle$$

where α and β are generally complex numbers and $|\alpha|^2$ and $|\beta|^2$ respectively signify the probability that B is found as either 0 or 1 and naturally

$$|\alpha|^2 + |\beta|^2 = 1.$$

Another interesting difference between a classical bit and a qubit is that while we always know the value of the classical bit, the value of the qubit can only be determined by measurement, and this measurement destroys any information in the qubit. For example, let:

$$C = \alpha|0\rangle + \beta|1\rangle, \quad |\alpha|^2 = 0.3, \quad |\beta|^2 = 0.7.$$

This qubit has a 30% probability of being measured as 0, and a 70% probability of being measured as 1. Once a measurement has been made, it will be either 0 or 1 and α and β will assume either 0 or 1 depending on what value the measurement yielded. As a result of this, normally you cannot view a qubit while an algorithm is running as measuring it would result in the superposition being destroyed. This is called collapsing the superposition.

Another special characteristic of the qubit is that it can exhibit quantum entanglement between itself and another qubit. This is an effect unique to quantum computers and it is used extensively in many algorithms and in quantum cryptography [3]. For example:

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

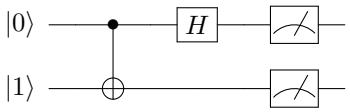
This is an entangled state where if we look at the first qubit we have a 50% chance to measure either 0 or 1 if the second qubit has not been measured. However, if a measurement is performed on the second qubit, the probability to measure 0 or 1 on the first qubit is either 100% or 0% depending on what the second qubit was measured as [4]. The measurement of one of the qubits affects the value of the other. This is a prime example of entanglement. This state in particular is one of the Bell states, useful for things such as quantum teleportation [1].

1.3 Quantum Gates

Quantum gates transform the value of a qubit into something different, depending on the properties of the gate. This is the basic building block of all quantum algorithms and can be very much compared to the basic logical gates of classical computers.

Each quantum gate can be expressed through its unitary matrix, which among other things shows how the gate will act on the input. The size of the input to the gate determines the size of its unitary matrix, a gate with one input has a 2x2 matrix while a gate with two inputs has an 4x4 matrix and so on, by increasing powers of 2.

In order to graphically describe quantum algorithms, a notation called quantum circuitry is often used. The number of qubits used is described by a horizontal line for each qubit and every quantum gate has a special graphical representation which usually looks like a box or a line between two or more qubits;



Shown above are, in order from left to right, the input, the CNOT gate, the Hadamard gate and a quantum measurement on an algorithm with 2 different qubits with initial values 0 and 1 respectively.

The gates used in the comparison of the quantum simulation algorithms in this paper are described below.

1.3.1 The Controlled-NOT gate (CNOT)

The CNOT quantum gate is named after Controlled-NOT, and can be compared to the classical NOT gate with one extra dependency added. It acts on two qubits and inverts the second qubit if and only if the first qubit is 1. The first qubit is called the control while the second is called the target.

Shown below is the 4x4 unitary matrix for the CNOT gate. One can see how the upper left of the matrix is in fact the identity and the lower right is an inversion.

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

The graphical representation of the CNOT gate is shown below, the first line is the control and the second is the target.

1.3.2 The Toffoli gate (CCNOT)

The Toffoli gate is named after Tommaso Toffoli [16] and is in effect an extension of the CNOT gate, and it is also often called a Controlled-Controlled-NOT gate, or CCNOT gate. Just as it sounds, the Toffoli gate is a CNOT gate with 2

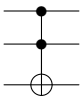
control bits. That is, the 3rd qubit will be inverted if both qubit 1 and qubit 2 are 1. Sometimes an n-Toffoli gate is referenced, this means a CNOT gate with n controlling qubits.

The Toffoli gate is proven to be universal for classical computation, that is, it can be used to simulate any classical gates, and so any classical circuits can be implemented on a quantum computer using Toffoli gates [1]. The Toffoli gate itself can also be implemented using two-qubit gates.

Shown below is the 8x8 unitary matrix for the Toffoli gate. Again one can see how the upper left of the matrix is in fact the identity and the lower right is an inversion, much like the CNOT gate.

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

The graphical representation of the Toffoli gate is shown below.



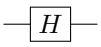
1.3.3 The Hadamard gate

The Hadamard gate is named after Jacques Hadamard, a french mathematician. It is one of the most useful gates in quantum physics [1] and creates a superpositioned state out of a normal 1 or 0. It is used in many quantum computer algorithms [3].

Shown below is the 2x2 unitary matrix for the Hadamard gate.

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

The graphical representation of the Hadamard gate is shown below.



1.4 Quantum Computer Algorithms

The field of quantum computers has developed through the discovery of algorithms rather than the actual use of quantum computers. Shor's proposed number factorization algorithm [11] renewed the field of quantum computing algorithm research when it was shown in 1994 [4], and since then many algorithms have been proposed, among them Grover's database search [12], which together with Shor's number factorization algorithm are the two most famous quantum computer algorithms known.

Modern quantum computer algorithms differ from classical computing algorithms in the relative low-level implementations they are built, as opposed to the high level programming that is performed in modern day classical computer algorithms, often operating on the qubit-level in the operations.

In this paper we will also take the example of the Deutsch-Josza algorithm, which is the first quantum computer algorithm that was proven to give a better result than what is possible on a classical computer [1]. It laid as a base for both Shor's and Grover's algorithms[11, 12].

We will also give a short introduction to computational complexity classes that are relevant to the algorithms presented.

1.4.1 Computational complexity classes

The classes relevant to the algorithms presented include: P, EQP, NP, BPP, BQP.

- P: The set of decision problems that can be solved by a deterministic *classical* machine in polynomial time. This class contains problems which can be effectively calculated.
- EQP: The set of decision problems that can be solved by a deterministic *quantum* machine in polynomial time. It is the quantum analogue of P. Also sometimes called QP.
- NP: The set of decision problems that can be solved by a non-deterministic classical machine in polynomial time. These problems are basically hard to calculate but easy to verify. P is contained within NP.
- BPP: The set of descision problems that can be solved by a probabilistic *classical* machine in polynomial time, with an error probability of at most 1/3 for all cases.
- BQP: The set of dscision problems that can be solved by a probabilistic *quantum* machine in polynomial time, with an error probability of at most 1/3 for all cases. The quantum analogue of BPP.

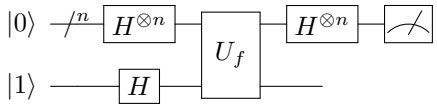
1.4.2 The Deutsch-Josza algorithm

The Deutsch-Josza algorithm was proposed by David Deutsch and Richard Jozsa [1] as a generalisation of Deutsch's algorithm with a quantum register of size n instead of 1. The purpose of the algorithm is to determine whether a function is balanced or constant with as few measurements as possible. In the classical case with n bits, $2^{n-1} + 1$ measurements would need to be performed to be

sure to have the right answer, because it is possible to recieve 2^{n-1} 0s before recieveing a 1 when evaluating the function classically. However with the use of the Deutsch-Josza Algorithm the answer can always be found in exactly one evaluation. It is a deterministic algorithm which is solvable in polynomial time, and is therefore in EQP.

Although of little practical use, the algorithm proves that quantum computers are capable of outperforming classical computers [1].

Shown below is the graphical representation of the algorithm with an input of n qubits.

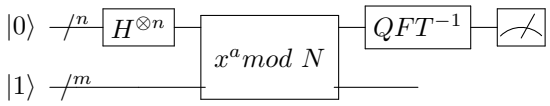


U_f is the quantum implementation of the function being tested. It must be either balanced or constant.

1.4.3 Shor's number factorization algorithm

Shor's algorithm is the first algorithm of any sort (classical or quantum) that have been proven to be able to prime factor a number in polynomial time; this is of great importance to cryptography as the most widely used system today relies on the fact that no known classical computer algorithms are able to prime factor in polynomial time.

It was also the first quantum computer algorithm to have a practical use and is as such possibly the most important quantum algorithm found to this date [1]. It relies on the use of the quantum Fourier transform [11].



Above is the quantum circuit for Shor's number factorization algorithm for the number N, where $n = 2\ \log N$ and $m = \log N$, $x^a mod\ N$ is calculated for a random chosen $a < N$ which has no common factors with N. QFT^{-1} denotes the inverse quantum Fourier transform, which due to intereference causes the important terms to have a higher amplitude than others [11]. From the final measurement either a result is extracted or the algorithm is re-run for a different value of a.

Shor's algorithm is probabilistic and the running time is $O((\log n)^2 * \log\log n)$ for the quantum computing part and it must perform $O(\log n)$ steps of post-processing classical computation, resulting in an overall polynomial time algorithm, placing it in BQP [20].

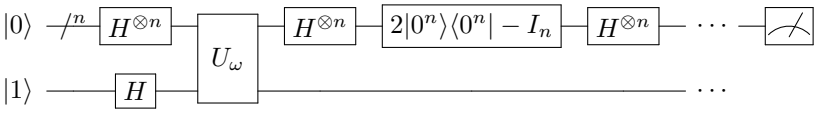
Shor first showed that the factoring problem can be reduced to period finding. This is the first part of the algorithm, and it is usually implemented classically. The second part of the algorithm uses the quantum Fourier transform to find the period, which is where the quantum parallelism is used to outshine classical computers. The third part includes finding the factors backwards from there, granted that the period is correct, which is also done clasically.

1.4.4 Grover's database search algorithm

Grover's database search is the second discovered quantum algorithm with great practical use. It is proven to be able to find an item in a database in $O(\sqrt{N})$ time, where N is the size of the database, as opposed to classical algorithms that need linear time to complete any database search, a quadratic speedup in time.

It is also unique in that it has been proven to be optimal, that is, it is the best possible database search algorithm that can be implemented through a quantum computer, so any other good quantum computer database search algorithm will have at least as many steps as Grover's database search[13]. Since the algorithm is probabilistic it is run with several iterations to give an answer with higher probability[12].

Shown below is the graphical representation of the algorithm where the number of qubits is n and the number of elements being searched is $N = 2^n$.



U_ω is Grover's Oracle function that returns 1 if and only if the input maps to a marked element (the element we are searching for). The part from the oracle until the dots is repeated $O(\sqrt{N})$ times for the best accuracy.

An interesting detail of Grover's database search algorithm is that if it is run too many times, eventually the result will diverge from the answer. As a result it must not only be run enough times, it cannot either be run it too many times.

Chapter 2

Quantum Computer Simulators

Chapter 2: Today there are over 100 different quantum computer simulators available freely over the Internet [18], many based on popular programming languages such as Java, C++ or Python. This is a brief introduction to each of the quantum computer simulators tested, as well as a few others considered for testing but discarded in the process.

2.1 Summary

Name	Computer Language	Last update
LibQuantum	C	2008
QCL	C/Pascal/Own	2006
Eqcs	C++	2012
Q++	C++	2009
jQuantum	Java	2011
PyQu	Python	2009
QFC	MATLAB	2009

2.2 LibQuantum

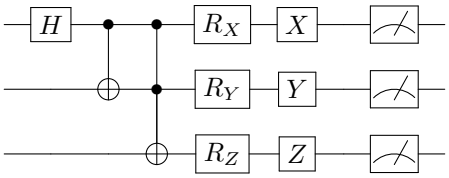
LibQuantum is based on C and was first released in 2003, and has since been continuously updated by the creators, Björn Butscher and Hendrik Weimer. It is today available in the version 1.0.0, released 2008, which not only supports quantum computer simulations but also general quantum algorithm simulation. It has been used as a simulator for a number of papers [8].

2.2.1 Documentation

The documentation for LibQuantum is very well written with examples for all the functions defined and an explanation for the mathematical theory behind them. It is an invaluable reference when programming with LibQuantum.

2.2.2 Included functions

It has pre-defined functions for the following quantum gates: Hadamard, CNOT, Toffoli (CCNOT), Axis rotation gates, Pauli spin gates, and many others



as well as already implemented functions for the QFT, QFT^{-1} and the calculation of $x^a mod\ N$, which as previously explained are all part of Shor's number factorization algorithm. A function exists for the creation of custom quantum gates through their unitary matrix.

Also included is an example program created with the functions in the program, to show how the functions can be used. The included quantum computer algorithms are implementations of Shor's factorization algorithm and Grover's database search algorithm.

2.2.3 Other features

Something useful for applications is the decoherence and quantum error correction simulation part of the library. The decoherence operator is included in all elementary gates provided by libquantum so as to simulate the behavior of a real quantum computer. The decoherence parameter can be set before any calculations are done if there is a known decoherence for a certain experimental setup. A source is provided for a list of values of the decoherence parameter[9]. Quantum error correction encoding can be applied before a operation and decoded afterwards. The QEC uses Steane's 3-qubit code [10, 21].

Additional features of LibQuantum include density operator formalism to deliver high performance of systems with a big number of qubits and support for decoherence simulation as well as time evolution by numerically integrating the Schrödinger equation through the use of the fourth-order Runge-Kutta scheme, which is useful for simulating arbitrary quantum systems, although less so for quantum computer simulation.

LibQuantum also includes a particularly useful set of tools for understanding the procedure of the programs, quobdump and quobprint, the first one creates an object code file from a program run (e.g. one calculation of Shor's algorithm) and the second one takes the object file created by the first and prints it. The output is a list of all the calculations performed by the program. This makes it very easy to perform debugging of your own programs.

2.3 QCL - Quantum Computer Language

QCL is one of the older languanges, first released in 1998 and developed by Bernhard Ömer[2]. It is written in C like LibQuantum. Unlike LibQuantum QCL attempts to make itself a basis for a new quantum programming language rather than work as an extension of the standard C libraries. The programs are run through the terminal rather than compiled and executed on their own. This means there is little programming experience outside of the knowledge of QCL needed to successfully create programs, however it also means that you cannot use standard C functions inside the program code, thus limiting the freedom of the programmer.

The installation of QCL is unfortunately currently broken. The standard installation method through the makefile is not working and as such the only way to run the program is to download a pre-compiled version from the website. The source code still serves as a possible way of analyzing the structure.

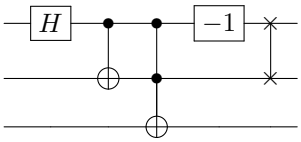
2.3.1 Documentation

The documentation for QCL is contained within a Masters thesis in Computer Science written by Bernhard Ömer[2] and it also contains a thorough introduction to quantum computers and quantum physics in general. The paper is filled with examples to show how the program is used so the user has plenty of information on how to construct their simulation. QCL has support for everything included in a basic programming language, such as all standard data types including complex numbers and strings. It includes all the operators expected to be found as well as logical operations. For a more thorough list of features of QCL see [2].

The use of QCL is harder than LibQuantum if you are used to programming in C or C++ at first since there is a new programming standard to adjust to. The documentation may be overwhelming but the frequent use of examples to illustrate the information is very useful.

2.3.2 Included functions

QCL has pre-defined functions for the following quantum gates: Hadamard, CNOT, Toffoli, NOT, Swap and the ability to define other gates through their unitary matrix.



Also included are implementations of Shor's factorization algorithm, Grover's database search algorithm and Deutsch-Jozsa's algorithm.

2.3.3 Other features

The structure of QCL is such that it is launched through the terminal together with the algorithm to be run, and it is possible to set how many qubits of memory should be allocated upon launch, through the use of launch parameters.

2.4 Eqcs

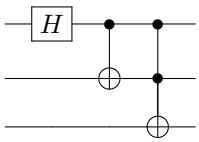
Eqcs is developed by Peter Belkner and is currently in version 0.0.8, last updated in 2012. Most of the code however is written in 1999 and as such is quite dated. The last update of the code came as a fix to a problem that the author of this report was experiencing and Peter Belkner is the only simulator author to have replied to any correspondence sent by the author. He is currently working on Eqcs version 0.1 which is a revision of the code, however it is not ready for testing yet.

2.4.1 Documentation

The documentation for Eqcs is found on the website [10] and contains an introduction to how the quantum state information is handled and how to create a qubit or a set of qubits. It continues to describe in detail much of what can be done to a set of qubits, such as logical operators between two sets, shifting and comparisons. Unfortunately it lacks description of anything related to simulating quantum circuits, such as example implementations of different quantum gates, with the exception of the example code wherein a cnot gate is defined. As such, any implementations of quantum algorithms are hard and it is only with the help of Belkner that the author has been able to write any algorithms.

2.4.2 Included functions

Eqcs comes with the following gates defined in examples: CNOT, CCNOT and with the help of Belkner, Hadamard. It also shows how the gates are defined, and there is as such a possibility of defining other gates.



There is an implementation of a quantum plain adder as described by Vedral et al. [15].

2.4.3 Other features

The quantum gate representation in Eqcs is based on the work of Barenco et al. [14] and Vedral et al. [15] where it is shown that all unitary operations on arbitrarily many bits can be expressed as a composition of a set of gates that consist of all one-bit quantum gates and the two-bit xor gate, and it is also shown how to construct a plain adder from quantum gates.

2.5 Q++

Q++ was first developed in 2003 by Chris Lomont. There is a program written on top of the Q++ library that allows for graphical creation of quantum circuits and simulations, although windows is the only supported platform and it is also quite buggy.

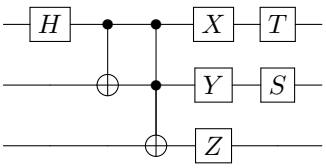
2.5.1 Documentation

Q++ comes with a brief documentation of its contents, without any examples at all which makes it hard to understand the way it is supposed to be used. In general the documentation is the least possible, it contains every function within the library and a short description of what it does but the lack of examples makes it very hard to follow.

There are a few examples included, outside of the documentation, that can be used to determine how to use the various functions. A lot of examples from [1] are included which gives a good guide if you own or have access to the book.

2.5.2 Included functions

It has pre-defined functions for the following quantum gates: Hadamard, CNOT, Toffoli, Pauli spin gates, T and S gate and the ability to define other gates through their unitary matrix.



Q++ also includes a function for the quantum fourier transform as well as its inverse.

2.5.3 Other features

Q++ has a very useful function called the Quantum Operator. A Qop is defined as a set of gates and as such construction of quantum circuits from the circuits diagram is very easy. This is shown through the programming of some exercises available in [1], included in the source code of the simulator.

Unfortunately as of the time of writing this paper the author has been unable to compile the code and therefore unable to test the strenghts and weaknesses of the library.

2.6 Other simulators

jQuantum, PyQu and QFC have not be tested but are still provided in this review to show that there are simulators written in many different computer languages.

Chapter 3

Practical Evaluation

Chapter 3: The testing consists of, where available, the timing of the implementations of the quantum algorithms for growing input numbers to determine the scaling in time with bigger inputs. This is a look at the simulators with existing implementations of Shor's number factorization algorithm and Grover's database search algorithm. All the test data is found in Appendix A.

All tests were conducted on a computer with a AMD 3.2 GHz Quad Core processor with 4096 MB of RAM running Ubuntu 11.10.

3.1 Shor's factorization algorithm

3.1.1 LibQuantum

The implementation of Shor's factorization algorithm in LibQuantum takes one input, a number. When it is run it reports throughout the calculation what is being done, as shown below for $N = 111$:

```
$ ./shor 111
N = 111, 37 qubits required
Random seed: 52
Measured 2276 (0.138916), fractional approximation is 5/36.
Possible period is 36.
Unable to determine factors, trying again.
Random seed: 4
Measured 6372 (0.388916), fractional approximation is 7/18.
Possible period is 18.
111 = 37 * 3
```

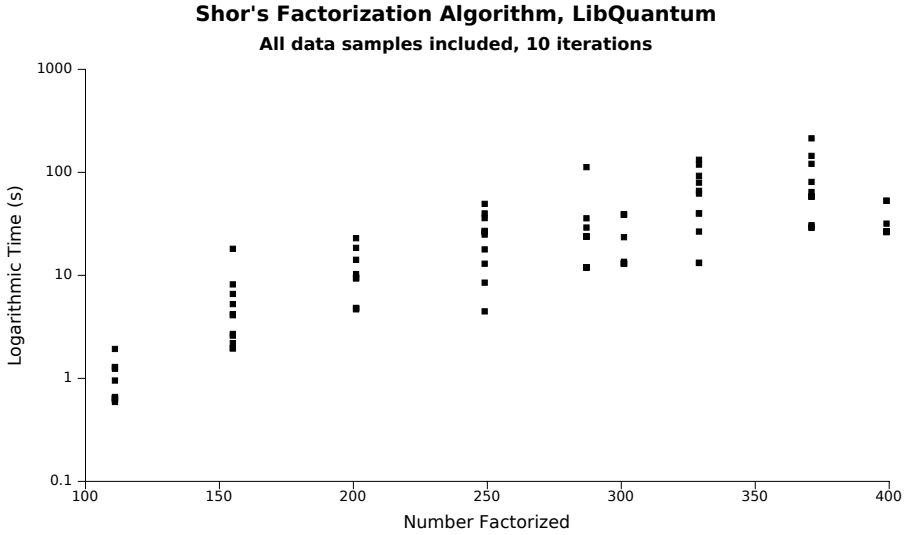
For testing purposes the algorithm was rewritten to restart the calculation if the measurement was unsuccessful as well as take another input, the number of iterations that wishes to be run, and instead output the time taken to complete each iteration, as shown below for $N = 155$ and iterations $= 10$:

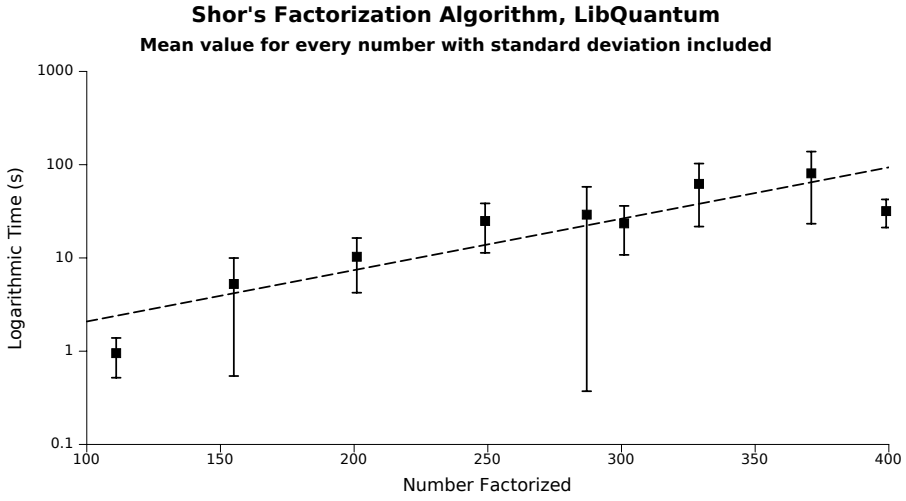
```
$ ./shorloop 155 10
N = 155, 41 qubits required
Time: 4 seconds and 30 milliseconds
Time: 4 seconds and 0 milliseconds
Time: 3 seconds and 950 milliseconds
Time: 10 seconds and 120 milliseconds
Time: 2 seconds and 40 milliseconds
Time: 2 seconds and 50 milliseconds
Time: 5 seconds and 690 milliseconds
Time: 1 seconds and 980 milliseconds
```

```
Time: 3 seconds and 810 milliseconds
Time: 4 seconds and 40 milliseconds
```

The practical limitations for the implementation comes at $N > 400$ as the time of the execution scales out of proportion. This is due to the program getting stuck in a loop where it cannot determine the factors of the number from the modular exponentiation. Having run the program for various numbers above 400 it can be said that for the majority of them the implementation cannot determine the factors. Since the program cannot reliably factor every number in the range $400 < N < 500$, it will be considered the practical limit of the program. A separate memory allocation problem is encountered for $N > 1000$.

Below is shown a number of graphs created with the use of the loop program. All different numbers were run with 10 iterations.





As is evident of the first graph, the degree of difference in the time range is consistently quite large. This is due to the probabilistic nature of Shor's algorithm,

and quantum computers in general, where if the desired value cannot be found, you simply run the calculation again. As a result the time elapsed is quite regularly found to be in groups, so that one group completed the algorithm in one pass, the next in two passes etc.

It is peculiar that sometimes when the number factorized is increased, the best calulation time will be similar or faster than it was for the smaller number. As can be seen in the second graph, the time for $N = 301$ is on average better than the time taken for $N = 271$. This can be greatly attributed to one large sample value for the latter, seen in the diagrams above, which brings up the average by a great deal. Another factor for the complex growth in calculation time is the how the frequency distribution of $x^a \bmod N$ varies with N.

In conclusion it can be seen that the calculation time for LibQuantum's implementation of Shor's factorization algorithm grows exponentially with time.

3.1.2 QCL

As QCL is not a library for using with C/C++ standard libraries, it is impossible to properly time the execution time of the algorithm inside QCL. However, the library QCL uses for simulations is found within the source code and includes an implementation of Shor's factorization algorithm. That implementation was modified in this experiment. The modified code can be found in appendix B.

The implementation takes one input, a number. When it is run it reports throughout the calculation what is being done, as shown below for $N = 111$:

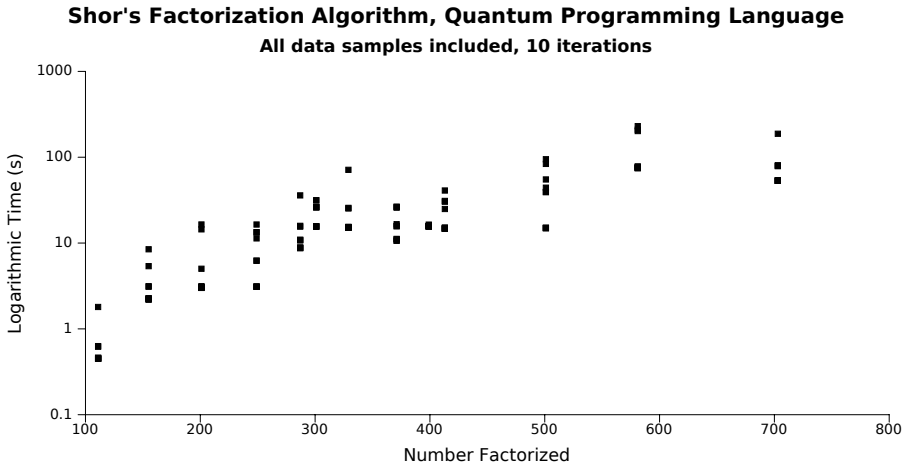
```
$ ./shor 111
factoring 111: random seed = 1332083009, tries = 3.

RESET:   reseting state to |0,0>
FFT:     performing 1st fourier transformation.
EXPN:    trying x = 68. |a,0> --> |a,68^a mod 111>
MEASURE: 2nd register: |*,73>
FFT:     performing 2nd fourier transformation.
MEASURE: 1st register: |8192,73>
rational approximation for 8192/2^14 is 1/2, possible period: 2
111 = 3 * 37
```

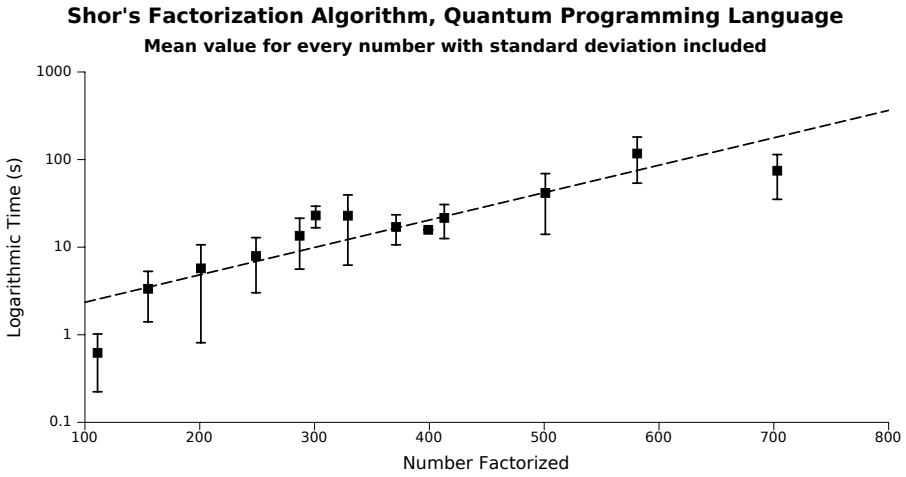
For testing purposes the algorithm was rewritten to take another input, the number of iterations that wishes to be run, and instead output the time taken to complete each iteration, as shown below for $N = 155$ and iterations $= 10$:

```
$ ./shorloop 155 10
Time taken 2 seconds 340 milliseconds
Time taken 2 seconds 320 milliseconds
Time taken 3 seconds 380 milliseconds
Time taken 2 seconds 390 milliseconds
Time taken 8 seconds 70 milliseconds
Time taken 6 seconds 900 milliseconds
Time taken 3 seconds 440 milliseconds
Time taken 2 seconds 420 milliseconds
Time taken 2 seconds 490 milliseconds
Time taken 2 seconds 470 milliseconds
```

The modified code can be found in appendix B. The practical limitations for the implementation are mostly felt by the execution time growing for bigger numbers. At $N = 703$ the mean time to complete the factorization is 75 seconds, while peaking as high as 188 seconds. A memory allocation error is encountered for $N > 1000$.

Below is shown a number of graphs created with the use of the loop program. All different numbers were run with 10 iterations.



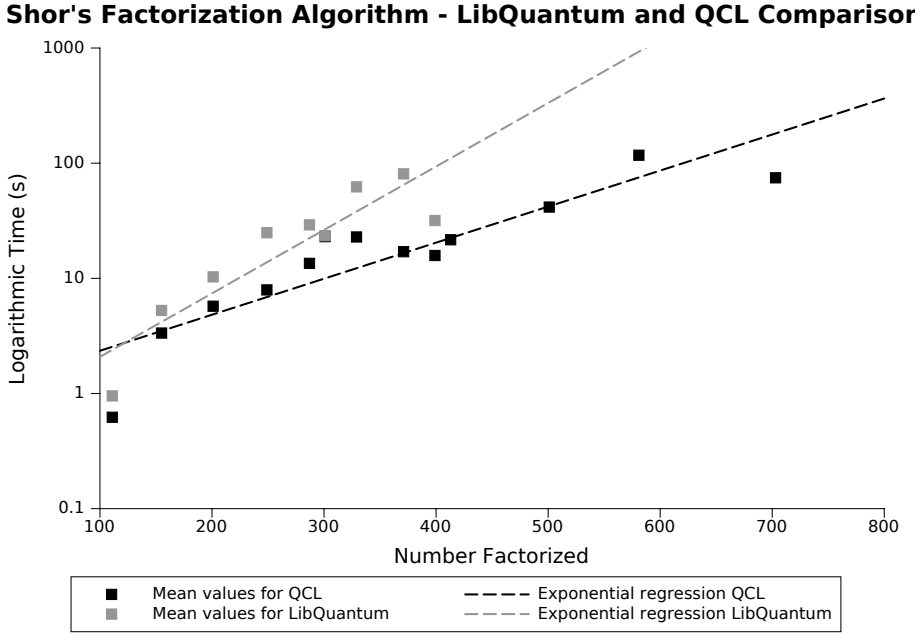


Compared to the graph for LibQuantum, here we have a greatly increased sample size due to the fact that the algorithm can factor numbers all the way up to 703 without running into memory errors. We can again note that the degree of difference in the time range is consistently large, for the same reasons as stated in the graph for LibQuantum.

Something interesting occurs for $400 > N > 300$, where the average time drops down from the previous value and then steadily rises up again. It is quite remarkable that while LibQuantum cannot handle numbers above 300 very well, QCL's implementation instead gets better for this number range.

In conclusion it can be seen that the calculation time for QCL's implementation of Shor's factorization algorithm also grows exponentially with time.

Finally show below is the graph for the mean values compared to each other.



It is interesting to note that the mean of QCL is better than LibQuantum for every value of N, although in some measurements the single fastest measurement of the two was LibQuantum. This suggests that the QCL implementation is less prone to single large sample values while LibQuantum is more so.

3.2 Grover's database search algorithm

3.2.1 LibQuantum

The implementation of Grover's database search algorithm on LibQuantum takes two inputs, the first is the number that should be searched for, and the second is the size of the qubit database, as shown for 42 and database size = 8 qubits ($N = 2^8 = 256$ positions).

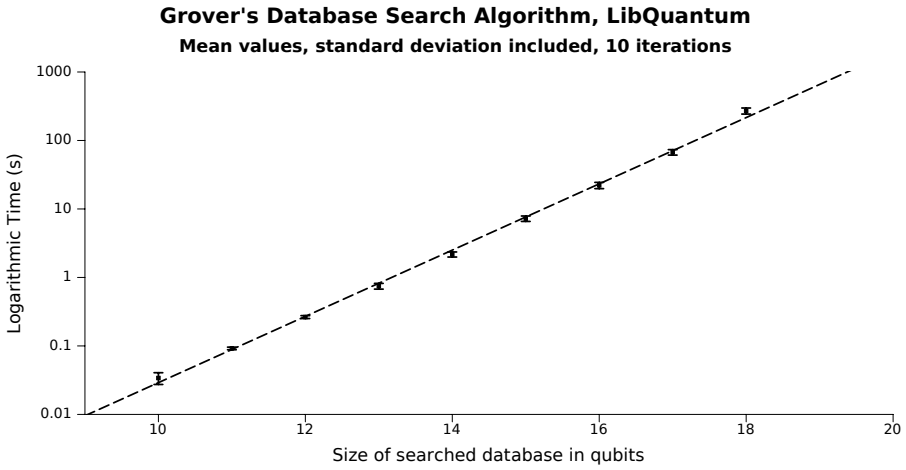
```
$ ./grover 42 8
Iterating 12 times
Iteration #1
Iteration #2
Iteration #3
Iteration #4
Iteration #5
Iteration #6
Iteration #7
Iteration #8
Iteration #9
Iteration #10
Iteration #11
Iteration #12

Found 42 with a probability of 0.999947
```

As seen the function iterates $\frac{\pi}{4}\sqrt{N}$ times and returns the probability of success. With bigger numbers the number of iterations increases as well as the time for every iteration.

For testing purposes the program was rewritten to take another input, the number of iterations that wishes to be run, and instead output the time taken to complete each iteration, as shown below for 42, database size 13 and 5 iterations:

```
$ ./groverloop 42 13 5
Time: 0 seconds and 700 milliseconds
Time: 0 seconds and 710 milliseconds
Time: 0 seconds and 740 milliseconds
Time: 0 seconds and 730 milliseconds
Time: 0 seconds and 780 milliseconds
```

The modified code can be found in appendix B. The time taken for each iteration includes all the function iterations. Included below is a graph showing the time scaling of LibQuantums implementation of Grover's database search algorithm.



As seen in the graph the statistical values almost perfectly align with the exponential trend line, and the standard deviation is comparatively small for all database sizes.

The practical limitations of the algorithm is for database size values of > 20, where the execution time of each iteration becomes too large. It returns a segmentation fault for database size values of > 25.

As a conclusion it can be said that the implementation of Grover's database search algorithm in LibQuantum grows exponentially in time with bigger qubit values of the database size, with about a tripling in computation time per qubit increase in database size. The reason for this is two-fold, since not only is the computation time increased for every iteration when the database is bigger, there is also a need for more iterations to give a sufficiently accurate result.

3.2.2 QCL

Although QCL has an implementation for Grover's database search algorithm, it is as previously stated currently impossible to time the execution time of the algorithm using standard library timing functions. Therefore no direct comparison in terms of execution time can be made, however sampled below is the use of the implementation for a few different values.

The program is launched through the terminal, first by launching QCL and second by calling the function grover, below shown searching for the value 25 in a database of 8 qubits ($N = 2^5 = 32$ positions):

```
$ ./qcl grover -i
QCL Quantum Computation Language (64 qubits, seed 1336898397)
[0/64] 1 |0>
qcl> grover(25)
: 5 qubits, using 3 iterations
: measured 25
[0/64] 1 |0>
```

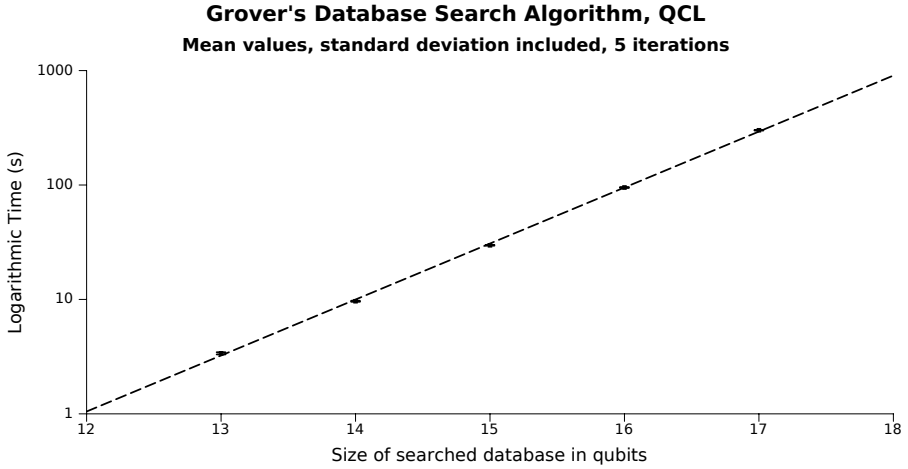
Compared to the LibQuantum implementation, a lot less information is provided. There is no indicator of when an iteration is completed which makes it unnecessarily frustrating to wait for a big computation to complete, as opposed to for LibQuantum where the progress is shown through the completion of each

iteration. It is worth noting at this stage that the QCL implementation uses half the iterations of LibQuantum and give as reason for this that the desired probability is only $p > \frac{1}{2}$ [2].

An edited version of the program was used for testing, which prints every iteration of the algorithm and takes an extra parameter, the size of the database in qubits and, for a fair comparison, the number of iterations increased to gain the same probability of correctness as LibQuantum (iterations $= \frac{\pi}{4}\sqrt{N}$):

```
qcl> grover(25,8)
: 8 qubits, using 13 iterations
: iteration 1 completed
: iteration 2 completed
: iteration 3 completed
: iteration 4 completed
: iteration 5 completed
: iteration 6 completed
: iteration 7 completed
: iteration 8 completed
: iteration 9 completed
: iteration 10 completed
: iteration 11 completed
: iteration 12 completed
: iteration 13 completed
: measured 25
[0/64] 1 |0>
```

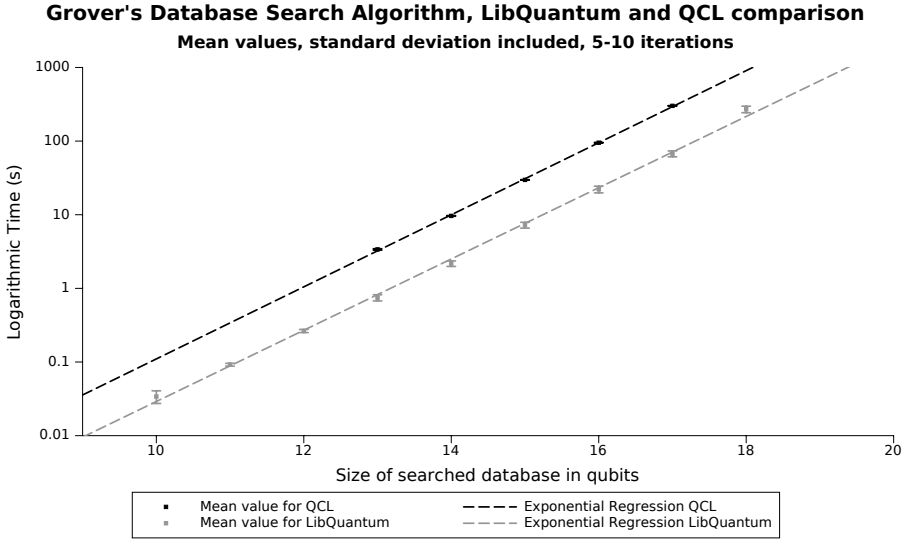
For smaller sizes of the database it is impossible to time the execution manually, but given a large enough database size and iterations a fair comparison should be possible. Manual timing using a stopwatch was used for database sizes of 13-17 qubits. There were 5 measurements performed for each database size. The human error in timing can account for roughly 0.5 seconds. Shown below are the results of the test.



As seen the growth is again even and without much deviation. The execution time is roughly tripled for every one qubit increase in database size. This is, again, down to both the increased number of iterations and the increased time in each iteration. In conclusion it is shown that the execution time grows exponentially for increasing database sizes.

The practical limitations of the algorithm is for database size values of > 18, where the execution time of each iteration becomes too large. It, like LibQuantum, returns a segmentation fault for database size values of > 25. This suggests that they use the same or similar size of temporary memory.

Finally shown below is the comparison graph between LibQuantum and QCL.



As seen the growth of the two is very similar, almost identical however there is a static relative difference between the two with LibQuantum outperforming QCL in every measurement. The static difference implicates that there is some part or function in the QCL implementation that takes much longer than the LibQuantum equivalent, but with the same scaling pattern. In conclusion it can be said that the LibQuantum implementation of Grover's database search algorithm is superior to the QCL implementation, being roughly 4,5 times faster for all sizes of the database.

Chapter 4

Programming with the simulators

Chapter 4: For all tested simulators an implementation of the Deutsch-Jozsa algorithm has been coded to compare the difficulty of creating an arbitrary quantum algorithm and to see how the simulators express the quantum states. All the code can be found in appendix B

4.1 LibQuantum

There was already an implementation of the Deutsch-Jozsa algorithm included in LibQuantum, which has been modified, the entire code can be found in appendix B. Writing the code is very easy and very similar to any other C code in the structure, making it easy to come from a background of C programming and dive straight into the functions LibQuantum has to offer. A look through the documentation gives a good overview of available functions.

For this particular algorithm the possibility to make an if-statement based on a measurement is great since it is useful for displaying the result as text rather than as a result of a measurement.

4.2 QCL

There is another implementation of Deutsch-Jozsa's algorithm included in QCL but a revised version has been created to get a feel of how the programming in QCL works.

At first it is hard and everything has to be referenced back to the documentation, which is thorough and a good source of help when constructing the algorithms. It is immediately obvious that the inability to set the initial state of the quantum computer is a lacking feature. The lack of a proper state printing function is also annoying as doing any kind of troubleshooting relies heavily on the changes of the quantum states from operation to operation.

The implementation does in fact not seem to work, and the author has been unable to pin the problem down other than noticing that the first Hadamard gate does not seem to give the desired state. If this is error in the code or error

in the program is not evident, although after much searching it would seem that there is some problem with the program itself.

4.3 Eqcs

After having sent several mails to the author of Eqcs and with the help of his Hadamard gate definition the algorithm was finally completed. The main problem was the lack of documentation. If Eqcs had documentation like LibQuantum it would be quite easy to use. As it stands even the definition of new gates is something that has to be guessed unless you are helped, as the examples provided on the website give some hint as to how to do it but no clear direction.

The inclusion as a library into the C++ family of libraries is welcome as it means that the use of other C++ functions is made possible.

Something else lacking from Eqcs is the ability to create an if-statement based on a quantum measurement. The best you can do is return the result of the measurement in ket-notation.

Chapter 5

Conclusions

Chapter 5: Conclusions regarding the testing and programming with the simulators, pros and cons of each design and suggestions on improvements further analysis possibilities.

5.1 The practical evaluation

The result of the practical evaluation weights heavily in favor of QCL when it comes to time scaling of Shor's algorithm. LibQuantum leaves something to desire both in terms of memory usage and execution time of its implementation.

The result of the testing of Grover's algorithm however weights in favor of LibQuantum, as it was shown to be consitently 4,5 times faster than the QCL implementation. This result points to show problems with the implementation on the QCL side as the scaling of the two is very similar. There should be an identifiable bottleneck in QCL that is much more efficient on LibQuantum.

In conclusions the pratical evaluation leaves something to desire from both the contenstants that have predefined implementations of Shor's and Grover's algorithms.

5.2 The programming

The result of the programming test is by far in favor of LibQuantum. The ease of programming and likeliness to standard C syntax makes it very easy to jump into and use, even with little experience in quantum computer simulation. The troubleshooting tools available make it very easy to find where your algorithm is doing something wrong.

QCL is hard to get used to and lacks many of the functions that exist in C/C++. The syntax is somewhat similar to C but with subtle changes that makes it sometimes unintuitive to use. With more use QCL might become easier and more comfortable to use but the lack of general functions from the standard computer languages will always stand out as a problem when constructing programs. Execution of your own algorithms within the qcl program is also a source of annoyance, as the program has to be relaunched every time you want a different random time seed in your algorithm. The lack of a good quantum state print function for troubleshooting is also noticeable.

Eqcs is, like LibQuantum, quite easy to jump into once you know how to use the basic functions, such as the gate constructor. The biggest problem is the lack of documentation and lack of predefined functions. With some more work on the documentation and with more gates defined the potential of Eqcs is great.

5.3 Conclusions

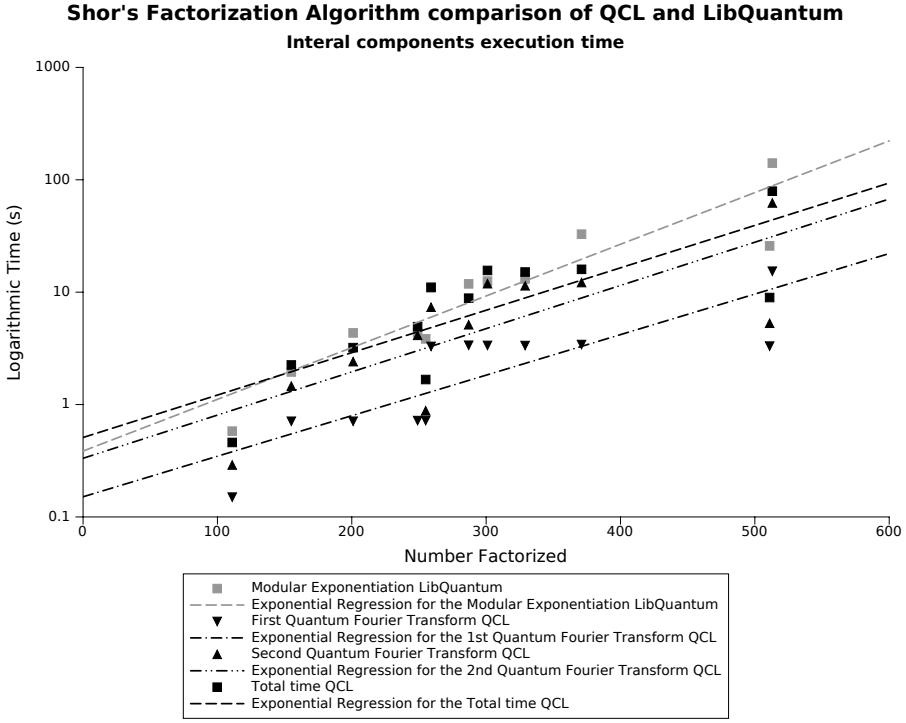
The winner of this review is LibQuantum, for its ease of use, high amount of predefined functions, as well as the troubleshooting software included and the decoherence simulation support, which although not tested, provides a more realistic simulation of quantum computer behaviour.

QCL is worth a special mention for its efficient memory management and good time scaling in its implementation of Shor's factorization algorithm. EQCS still has a long way to go to reach the standard of documentation and implementations that LibQuantum has achieved.

5.4 Improvements

It is obvious to say that LibQuantum can improve its memory management and speed up the execution time of its implementation of Shor's algorithm. After further investigating the differences between the QCL implementation and the LibQuantum implementation using standard library timing functions , it is shown that the LibQuantum included function for Modular Exponentiation is responsible for the biggest execution time increase, and it scales much worse than the QCL counterpart. As such, there should be room for improvement in that function, however it is important to note that the QCL implementation is slightly different from the LibQuantum one, utilizing two separate quantum Fourier transformations. Further work in this field could look through the LibQuantum included functions and attempt to find where the bottleneck is and what can be done about it.

Show below is the execution time and scaling of the various modules of the two implementations. The total time of LibQuantum's implementation is only around 1,5% bigger than the execution time of the Modular Exponentiation alone, as such the comparison between total time is made with that as guide in the case of LibQuantums implementation. The two biggest components in QCL's implementation are shown as well as the total time for the whole program. Notice the jumps at $N = 255$ and $N = 511$. This is due to the prime numbers factors just before a power of 2 being easier to calculate, as well as the needed increase in register size for the numbers above individual powers of 2. One can also notice how the total time for the LibQuantum implementation is below the QCL implementation for low values of N. As the time on that scale is measured in less than a second the result is not very important.


Shor's Factorization Algorithm comparison of QCL and LibQuantum
Interal components execution time
Logarithmic Time (s)
1000
100
10
1
0.1
0
100
200
300
400
500
600
Number Factorized
Modular Exponentiation LibQuantum
Exponential Regression for the Modular Exponentiation LibQuantum
First Quantum Fourier Transform QCL
Exponential Regression for the 1st Quantum Fourier Transform QCL
Second Quantum Fourier Transform QCL
Exponential Regression for the 2nd Quantum Fourier Transform QCL
Total time QCL
Exponential Regression for the Total time QCL

Bibliography

[1] Michael A. Nielsen, Isaac L. Chuang (2000), *Quantum Computation and Quantum Information*, Cambridge University Press. ISBN 0-521-63235-8.

[2] Bernhard Ömer (2000), *Quantum Programming in QCL*, Vienna University of Technology,
`http://tph.tuwien.ac.at/~oemer/doc/quprog.pdf`

[3] Eleanor Rieffel (2008), *An Overview of Quantum Computing for Technology Managers*,
`arXiv:0804.2264v2[quant-ph]`

[4] Eleanor Rieffel, Wolfgang Polak (2000), *An Introduction to Quantum Computing for Non-Physicists*,
`arXiv:quant-ph/9809016v2`

[5] Bernhard Ömer (1998), *A Procedural Formalism for Quantum Computing*, Vienna University of Technology,
`http://tph.tuwien.ac.at/~oemer/doc/qcldoc.pdf`

[6] Bernhard Ömer (2009), *Structured Quantum Programming*,
Vienna University of Technology.
`http://tph.tuwien.ac.at/ oemer/doc/structquprog.pdf`

[7] Bernhard Ömer (2003), *Classical Concepts in Quantum Programming*,
`arXiv:quant-ph/0211100v2`

[8] Björn Butscher, Hendrik Weimer (2011),
`http://www.libquantum.de/bibliography`

[9] David P. DiVincenzo (1994), *Two-bit gates are universal for quantum computation*,
`arXiv:cond-mat/9407022v1`

[10] Andrew Steane (1996), *Multiple Particle Interference and Quantum Error Correction*,
`arXiv:quant-ph/9601029v3`

[11] Peter Shor (1996),
Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer,
`arXiv:quant-ph/9508027v2`

[12] Lov Grover (1996), *A fast quantum mechanical algorithm database search*, arXiv:quant-ph/9605043v3

[13] Charles H. Bennet, Ethan Bernstein, Gilles Brassard, Umesh Vazirani (1997), *Strenghts and Weaknesses of Quantum Computing*, arXiv:quant-ph/9701001v1

[14] Adriano Barenco, Charles H. Bennet, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John Smolin, Harald Weinfurter (1995), *Elementary gates for quantum computation*, arXiv:quant-ph/9503016v1

[15] Vlatko Vedral, Adriano Barenco, Artur Ekert (1995), *Quantum Networks for Elementary Arithmetic Operation*, arXiv:quant-ph/9511018v1

[16] Tommaso Toffoli (1980), *Reversible Computing*, http://pm1.bu.edu/~tt/publ/revcomp-rep.pdf

[17] Jonathan Barret, Stefano Pironio (2005), *Popescu-Rohrlich Correlations as a Unit of Nonlocality* http://link.aps.org/doi/10.1103/PhysRevLett.95.140401

[18] Quantiki, Quantum wiki page (2011), *List of QC simulators* http://www.quantiki.org/wiki/List_of_QC_simulators

[19] Eqcs (2012), *Eqcs-0.0.8* http://home.snafu.de/pbelkner/eqcs/index.html

[20] Matthew Hayward (2008), *Quantum Computing and Shor's Algorithm* http://students.imsa.edu/ matth/quant/299/paper.pdf

[21] LibQuantum (2012), *Quantum Error Correction* http://www.libquantum.de/api/1.1/Quantum-Error-Correction.html

Appendix A

Statistical Data

A.1 Shor's Factorization Algorithm

A.1.1 LibQuantum

N/Iteration	111	155	201	249	287	301	329	371	399
1	0,59	2,7	4,73	26,43	23,86	13,55	13,26	144,5	26,47
2	0,64	2,2	14,17	27,03	23,65	13,11	132,51	58,09	26,47
3	1,93	2,6	4,68	49,45	23,97	13,03	39,85	213,9	52,97
4	0,62	18,11	9,48	35,94	11,93	13,01	66,14	121,04	26,5
5	0,64	4,1	18,5	4,47	23,85	13	118,94	30,66	26,54
6	1,24	4,2	4,83	17,86	11,92	39,03	79,52	64,43	26,5
7	0,64	1,95	4,81	8,5	35,79	39,04	40,04	58,16	26,62
8	1,28	1,97	22,94	26,36	11,96	38,98	26,63	58,23	53,06
9	0,66	8,17	9,5	39,91	11,93	38,94	92,56	29,58	26,56
10	1,29	6,6	9,32	12,98	112,9	12,98	13,18	28,88	26,23
Mean	0,953	5,26	10,3	24,89	29,96	23,47	62,263	80,75	31,792
Std. Dev	0,434	4,718	6,057	13,553	31,190	12,682	40,564	57,469	10,612
Median	0,65	3,4	9,4	26,39	23,75	13,33	53,09	58,19	26,52

A.1.2 Quantum Computer Language

N/Iteration	111	155	201	249	287	301	329	371	399	413	501	581	703
1	1,8	2,19	3,01	16,44	15,87	15,47	15,46	11,18	16,4	30,97	15,01	209,25	78,9
2	0,45	3,14	16,46	3,11	10,93	25,97	25,69	25,86	15,7	30,33	15,11	76,36	79,01
3	0,63	3,1	3,03	13,35	35,91	26,13	15,32	15,68	15,74	14,84	94,66	230,05	187,71
4	0,45	2,24	3,05	6,24	8,7	26,3	71,3	11,08	15,82	24,94	55,06	75,32	53,98
5	0,45	8,46	3,1	11,33	8,88	31,52	15,03	16,57	15,7	14,76	14,92	74,45	53,2
6	0,45	2,23	3	3,09	10,73	26,36	25,29	26,3	15,6	40,94	44,19	74,77	53,6
7	0,46	5,38	3,14	3,12	8,81	15,68	15,07	26,54	15,87	14,76	39,29	75,37	53,02
8	0,62	2,21	14,44	13,36	10,85	31,43	15,28	10,66	15,75	14,82	14,89	201,74	53,04
9	0,45	2,27	5,02	6,22	15,53	15,63	15,06	15,78	15,65	15,19	39,27	78,12	53,25
10	0,46	2,27	3,03	3,09	8,78	15,64	15,25	10,67	15,63	14,65	83,59	77,27	80,61
Mean	0,622	3,349	5,728	7,935	13,5	23,01	22,875	17,03	15,79	21,62	41,599	117,27	74,632
Std. Dev	0,3985	1,9443	4,9167	4,9177	7,8861	6,3500	16,643	6,3921	0,219	9,0765	27,588	63,463	39,469
Median	0,455	2,27	3,075	6,23	10,79	26,05	15,3	15,73	15,72	15,02	39,28	76,815	53,79

A.2 Grover's Database Search Algorithm

A.2.1 LibQuantum

Qubits/Iteration	10	11	12	13	14	15	16	17	18
1	0,03	0,09	0,25	0,69	2,21	7,32	20,44	68,44	284,2
2	0,03	0,09	0,27	0,71	1,65	7,7	21,27	50,25	294,2
3	0,04	0,09	0,25	0,77	2,22	7,36	22,56	70,28	291,5
4	0,04	0,09	0,27	0,79	2,16	7,6	18,12	68,31	222,7
5	0,02	0,1	0,28	0,82	2,23	7,25	24,35	68,07	285,7
6	0,04	0,09	0,26	0,63	2,11	5,32	23,56	64,42	293,2
7	0,03	0,09	0,29	0,81	2,24	7,23	24,37	67,96	284,9
8	0,04	0,1	0,26	0,8	2,33	7,52	18,38	73,02	214,6
9	0,03	0,09	0,25	0,82	2,23	7,4	24,47	69,9	266,5
10	0,04	0,09	0,26	0,63	2,33	7,64	23,64	72,78	267
Mean	0,034	0,092	0,264	0,747	2,171	7,234	22,116	67,34	270,4
Std. Dev	0,007	0,004	0,013	0,072	0,185	0,657	2,317	6,170	27,528
Median	0,035	0,09	0,26	0,78	2,225	7,38	23,06	68,38	284,5

A.2.2 Quantum Computer Language

Qubits/Iteration	13	14	15	16	17
1	3,5	9,5	30,1	95	301,6
2	3,4	9,8	29,5	96,5	299,5
3	3,4	9,6	29,7	95,4	298,4
4	3,3	9,7	29,8	93,6	301,3
5	3,3	9,6	29,5	94,9	304,2
Mean	3,38	9,64	29,72	95,08	301
Std. Dev	0,075	0,102	0,223	0,933	1,985
Median	3,4	9,6	29,7	0,932	301,3

A.3 Internal timing comparison of Shor's Algorithm

A.3.1 LibQuantum

N / Operation	111	155	201	249	255	259	287	301	329	371	511	513
Modular Exponentiation	0,58	1,95	4,34	4,35	3,84	11,17	11,85	12,54	13,12	32,81	25,8	140,72
Measure	0,02	0,05	0,1	0,09	0,09	0,21	0,21	0,21	0,22	0,43	0,43	0,98
Quantum Fourier Transform	0,02	0,01	0,11	0,05	0,01	0,14	0,07	0,07	0,12	0,19	0,19	1,96

A.3.2 Quantum Computer Language

N / Operation	111	155	201	249	255	259	287	301	329	371	511	513
1st Quantum Fourier Transform	0,15	0,71	0,71	0,72	0,72	3,29	3,37	3,36	3,35	3,42	3,31	15,36
Modular Exponentiation	0,01	0,07	0,07	0,06	0,06	0,3	0,3	0,3	0,31	0,32	0,31	1,39
Measure	0,01	0,01	0,02	0,02	0,01	0,05	0,04	0,05	0,05	0,05	0,05	0,2
2nd Quantum Fourier Transform	0,29	1,46	2,41	4,13	0,88	7,35	5,13	11,9	11,4	12,18	5,29	62,21

Appendix B

Programming code

B.1 LibQuantum

B.1.1 Shor loop

```
1  /*
2    shorloop.c, modified version of shor.c by
3    Bjoern Butscher and Hendrik Weimer made for
4    the purpose of iterating and measuring effiency.
5
6    By Johan Brandhorst.
7  */
8
9  #include <stdlib.h>
10 #include <stdio.h>
11 #include <math.h>
12 #include <time.h>
13
14 #include <quantum.h>
15
16 int main(int argc, char **argv)
17 {
18   quantum_reg qr;
19   int j,i;
20   int width, swidth;
21   int x = 0;
22   int N;
23   int iter;
24   int c,q,a,b, factor;
25
26   if(argc == 1 || argc == 2 || argc > 3)
27     {
28       printf("Usage: shorloop [number] [iterations]\n\n");
29       return 3;
30     }
31
32   N=atoi(argv[1]);
33   iter = atoi(argv[2]);
34
35   if(N<15 || N % 2 == 0)
36     {
37       printf("Invalid number\n\n");
38       return 3;
39     }
40
41   width=quantum_getwidth(N*N);
42   swidth=quantum_getwidth(N);
43
44   printf("N = %i, %i qubits required\n", N, width+3*swidth+2);
```

```
45
46    // The iterations needed
47    for (j = 0; j < iter; j++)
48      {
49        // Start the timing function
50        clock_t start = clock(), diff;
51
52      retry:
53        srand(time(0));
54        // To make sure a new random seed is selected
55        x = 0;
56
57        if(argc >= 3)
58    {
59      x = atoi(argv[2]);
60    }
61        while((quantum_gcd(N, x) > 1) || (x < 2))
62    {
63      x = rand() % N;
64    }
65
66        qr=quantum_new_qureg(0, width);
67
68        for(i=0;i<width;i++)
69    quantum_hadamard(i, &qr);
70
71        quantum_addscratch(3*swidth+2, &qr);
72
73        quantum_exp_mod_n(N, x, width, swidth, &qr);
74
75        for(i=0;i<3*swidth+2;i++)
76    {
77      quantum_bmeasure(0, &qr);
78    }
79
80        quantum_qft(width, &qr);
81
82        for(i=0; i<width/2; i++)
83    {
84      quantum_cnot(i, width-i-1, &qr);
85      quantum_cnot(width-i-1, i, &qr);
86      quantum_cnot(i, width-i-1, &qr);
87    }
88
89        c=quantum_measure(qr);
90
91        // If 0 is measured, try again
92        if (c == 0)
93    {
94      printf("Measured 0, trying again");
95      goto retry;
96    }
97
98        q = 1<<(width);
99
100        printf("Measured %i (%f), ", c, (float)c/q);
101
102       quantum_frac_approx(&c, &q, width);
103
104        printf("fractional approximation is %i/%i.\n", c, q);
105
106       if((q % 2 == 1) && (2*q<(1<<width)))
107   {
108      printf("Odd denominator, trying to expand by 2.\n");
109     q *= 2;
110   }
111
112       if(q % 2 == 1)
113   {
114     printf("Odd period, try again.\n");
115     goto retry;
116   }
117
118       printf("Possible period is %i.\n", q);
```

```
119
120         a = quantum_ipow(x, q/2) + 1 % N;
121         b = quantum_ipow(x, q/2) - 1 % N;
122
123         a = quantum_gcd(N, a);
124         b = quantum_gcd(N, b);
125
126         if(a>b)
127     factor=a;
128         else
129     factor=b;
130
131         if((factor < N) && (factor > 1))
132     {
133       printf("%i = %i * %i\n", N, factor, N/factor);
134       diff = clock() - start;
135
136       // Print the time taken
137       int msec = diff * 1000 / CLOCKS_PER_SEC;
138       printf("Time: %d seconds and %d milliseconds\n"
139        ,msec/1000, msec%1000 );
140     }
141         else
142     {
143       printf("Factors could not be determined\n");
144       goto retry;
145     }
146       }
147
148
149     // Clean up
150     quantum_delete_qureg(&qr);
151
152     return 0;
153   }
```

B.1.2 Grover Loop

```
1  /*
2    groverloop.c, modified version of grover.c by
3    Bjoern Butscher and Hendrik Weimer made for
4    the purpose of iterating and measuring effiency.
5
6    By Johan Brandhorst.
7  */
8
9  #include <quantum.h>
10 #include <stdio.h>
11 #include <math.h>
12 #include <stdlib.h>
13 #include <time.h>
14
15 #ifdef M_PI
16 #define pi M_PI
17 #else
18 #define pi 3.141592654
19 #endif
20
21 void oracle(int state, quantum_reg *reg)
22 {
23   int i;
24
25   for(i=0;i<reg->width;i++)
26     {
27       if(!(state & (1 << i)))
28   {
29     quantum_sigma_x(i, reg);
30   }
31     }
32
33   quantum_toffoli(0, 1, reg->width+1, reg);
34
35   for(i=1;i<reg->width;i++)
36     {
37       quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
38     }
39
40   quantum_cnot(reg->width+i, reg->width, reg);
41
42   for(i=reg->width-1;i>0;i--)
43     {
44       quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
45     }
46
47   quantum_toffoli(0, 1, reg->width+1, reg);
48
49   for(i=0;i<reg->width;i++)
50     {
51       if(!(state & (1 << i)))
52   quantum_sigma_x(i, reg);
53     }
54
55 }
56
57 void inversion(quantum_reg *reg)
58 {
59   int i;
60
61   for(i=0;i<reg->width;i++)
62     quantum_sigma_x(i, reg);
63
64   quantum_hadamard(reg->width-1, reg);
65
66   if(reg->width==3)
67     quantum_toffoli(0, 1, 2, reg);
68
69   else
70     {
```

```
71        quantum_toffoli(0, 1, reg->width+1, reg);
72
73        for(i=1;i<reg->width-1;i++)
74    {
75      quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
76    }
77
78        quantum_cnot(reg->width+i, reg->width-1, reg);
79
80        for(i=reg->width-2;i>0;i--)
81    {
82      quantum_toffoli(i, reg->width+i, reg->width+i+1, reg);
83    }
84
85        quantum_toffoli(0, 1, reg->width+1, reg);
86      }
87
88    quantum_hadamard(reg->width-1, reg);
89
90    for(i=0;i<reg->width;i++)
91      quantum_sigma_x(i, reg);
92  }
93
94
95  void grover(int target, quantum_reg *reg)
96  {
97    int i;
98
99    oracle(target, reg);
100
101   for(i=0;i<reg->width;i++)
102     quantum_hadamard(i, reg);
103
104   inversion(reg);
105
106   for(i=0;i<reg->width;i++)
107     quantum_hadamard(i, reg);
108
109 }
110
111 int main(int argc, char **argv)
112 {
113   quantum_reg reg;
114   int iter, i, j, N, width=0;
115
116   srand(time(0));
117
118   if(argc==1)
119     {
120       printf("Usage: grover [number] [[qubits]] [iterations] \n\n");
121       return 3;
122     }
123
124   N=atoi(argv[1]);
125   iter=atoi(argv[3]);
126
127   if(argc > 2)
128     width = atoi(argv[2]);
129
130   if(width < quantum_getwidth(N+1))
131     width = quantum_getwidth(N+1);
132
133   reg = quantum_new_qureg(0, width);
134
135   for (j = 0; j < iter; j++)
136     {
137       // Start the timer
138       clock_t start = clock(), diff;
139
140       quantum_sigma_x(reg.width, &reg);
141
142       for(i=0;i<reg.width;i++)
143   quantum_hadamard(i, &reg);
144
```

```
145         quantum_hadamard(reg.width, &reg);
146
147         for(i=1; i<=pi/4*sqrt(1 << reg.width); i++)
148     {
149       grover(N, &reg);
150     }
151
152         quantum_hadamard(reg.width, &reg);
153
154         reg.width++;
155
156         quantum_bmeasure(reg.width-1, &reg);
157
158         // Stop the timer and print the time taken
159         diff = clock() - start;
160
161         int msec = diff * 1000 / CLOCKS_PER_SEC;
162         printf("Time: %d seconds and %d milliseconds\n",msec/1000, msec%1000 ↩
                ;
163
164       }
165
166     // Clean up
167     quantum_delete_qureg(&reg);
168
169     return 0;
170 }
```

B.1.3 Deutsch-Jozsa's algorithm

```
1  /*
2   Implementation of Deutsch-Josza's algorithm, created by Bjoern Butscher
3   and Hendrik Weimer and modified by Johan Brandhorst.
4  */
5
6  #include <quantum.h>
7  #include <math.h>
8  #include <time.h>
9  #include <stdlib.h>
10 #include <stdio.h>
11
12 void f (quantum_reg *reg, int N)
13 {
14   // Performs the CNOT with first qubit as control and qubit N+1 as target
15   quantum_cnot(0, N, reg);
16 }
17
18 int main(int argc, char **argv)
19 {
20   // Perform random seed for the simulation of quantum behaviour
21   srand(time(0));
22
23   int verbose = 0;
24
25   if (argc < 2 || !atoi(argv[1]))
26     {
27       printf("Usage: ./deutsch [number of qubits] v [for verbose mode]");
28       return 0;
29     }
30
31   int N = atoi(argv[1]);
32
33   if (N < 1)
34     {
35       printf("The Number of Qubits must be > 0");
36       return 0;
37     }
38
39   if (argc > 2 && *argv[2] == 'v')
40     verbose = 1;
41
42   // Create new Quantum Registry, N+1 qubits, start value binary 1(N*0).
43   quantum_reg reg;
44   reg = quantum_new_qureg(1 << N, N+1);
45   printf("The Input:\n");
46   quantum_print_qureg(reg);
47
48   // Perform Hadamard on the qubits
49   quantum_walsh(N+1, &reg);
50   if (verbose)
51     {
52       printf("Hadamard(N+1):\n");
53       quantum_print_qureg(reg);
54     }
55
56   // The function to be tested
57   f(&reg, N);
58   if (verbose)
59     {
60       printf("CNOT(1->N+1):\n");
61       quantum_print_qureg(reg);
62     }
63
64   // Perform hadamard on qubits 0-N.
65   quantum_walsh(N, &reg);
66   if (verbose)
67     {
68       printf("Hadamard(N):\n");
69       quantum_print_qureg(reg);
70     }
```

```
71
72    // Measure qubit N+1 to get rid of it from the registry
73    // (bmeasure counts qubits from 0, so qubit N+1 is acessed as N)
74    quantum_bmeasure(N, &reg);
75
76    // Measure the remaining Quantum Registry, if 1 the function is balanced,
77    // if 0 constant.
78    if (quantum_measure(reg))
79      printf("Result: Function is Balanced\n");
80    else
81      printf("Result: Function is Constant\n");
82
83    return 0;
84  }
```

B.2 QCL

B.2.1 Shor loop

```
1  /*
2   Implementation of Shor's algorithm by Bernhard Oemer.
3   Slightly edited by Johan Brandhorst for comparison purposes.
4 */
5
6 #include <math.h>
7 #include <time.h>
8 #include <unistd.h>
9 #include <stdio.h>
10
11 #include "operator.h"
12
13 extern char *optarg;
14 extern int optind;
15
16 // global variables
17
18 int seed=time(0); // random seed value
19 int quiet=0;    // quiet mode
20 int verbose=0;    // verbose mode
21 int show=0;   // show state spectrums
22 int dump=0;   // dump quantum state
23 int maxtries=3;   // max. number of selections
24 int maxgates=-1;  // max. number of cond. phase gates per bit in FFT
25 int iter=0;   // total number of tries
26
27
28 // returns 0 and sets *a and *b if n = (*a) * (*b)
29 // returns 1 if n is a prime number
30
31 int factorize(word n,word *a,word *b) {
32   word i,m;
33
34   m=(word)ceil(sqrt((double)n));
35   for(i=2;i<=m;i++) {
36     if(n%i==0) {
37       *a=i;
38       *b=n/i;
39       return 0;
40     };
41   };
42   return 1;
43 }
44
45 // returns 1 if p is a power of b and 0 otherwise
46
47 int testpower(word p,word b) {
48   if(p<b) return testpower(b,p);
49   if(p==b) return 1;
50   if(p%b) return 0;
51   return testpower(p/b,b);
52 }
53
54 // returns x^a mod n
55
56 word powmod(word x,word a,word n) {
57   word u,y;
58   int i;
59   y=1; u=x;
60   for(i=0;i<BPW-1;i++) {
61     if(a & (1<<i)) { y*=u; y%=n; };
62     u*=u; u%=n;
63   };
64   return y;
65 }
66
67 // returns the greatest common divisor of a and b
```

```
68
69  int gcd(int a,int b) {
70    if(b>a) return gcd(b,a);
71    return a%b ? gcd(a,a%b) : b;
72  }
73
74  // returns a random number 1 < r < (n-1) coprime to n
75
76  int randcoprime(int n) {
77    int x;
78
79    while(1) {
80      x=qc_lrand()%(n-3)+2;
81      if(gcd(x,n)==1) return x;
82    }
83  }
84
85  // finds the best rational approximation (*p)/(*q) to x with
86  // denominator < qmax and sets *p and *q accordingly.
87
88  void approx(double x,word qmax,word *p,word *q) {
89    word p0,p1,p2;
90    word q0,q1,q2;
91    word a;
92    double y,z,e;
93
94    e=1.0/(2.0*(double)qmax*(double)qmax);
95    y=x; a=(int)floor(y);
96    p0=1; p1=a;
97    q0=0; q1=1;
98
99    while(1) {
100     z=y-floor(y);
101     if(z<e) break;
102     y=1/z;
103     a=(int)floor(y);
104     p2=a*p1+p0;
105     q2=a*q1+q0;
106     if(q2>qmax) break;
107     p0=p1; p1=p2;
108     q0=q1; q1=q2;
109   };
110   *p=p1; *q=q1;
111 }
112
113 // performs a fast fourier transformation on qs using
114 // Coppersmith's algorithm
115
116 opVar opFFT(int n) {
117   int i,j,m;
118   opVar op;
119
120   for(i=0;i<n;i++) {
121     if(maxgates>0) { m=i-maxgates; if(m<0) m=0; } else m=0;
122     for(j=m;j<i;j++) op *= opX(n,n-i-1,n-j-1,i-j+1);
123     op *= opEmbedded(n,n-i-1,new opU2(PI/2,PI/2,PI/2,PI));
124   };
125   for(i=0;i<(n/2);i++) op *= opSwap(n,1,i,n-i-1);
126   return op;
127 }
128
129 // prints usage message
130
131 void usage() {
132   cerr << "USAGE: shor [options] number\n";
133   cerr << "Options: -s<seed> set random seed value\n";
134   cerr << "         -t<maxtries> set max. no. of selections from same state↩
        .\n";
135   cerr << "         -g<gates> set max. no. of cond. phase gates per bit in ↩
        FFT.\n";
136   cerr << "         -q operate quietly, -v verbose output\n";
137 }
138
139 // main program
```

```
140
141 int main(int argc,char **argv) {
142
143   word number;    // number to be factored
144   word factor;    // found factor
145   int width;    // length of N in bits
146
147   number = atoi(argv[1]);
148
149   { // testing number
150
151     word a,b;
152
153     if(number%2==0) {
154       cerr << "number must be odd !\n";
155       exit(1);
156     };
157     if(factorize(number,&a,&b)) {
158       cerr << number << " is a prime number !\n";
159       exit(1);
160     };
161     if(testpower(b,a)) {
162       cerr << number << " is a prime power of " << a << " !\n";
163       exit(1);
164     };
165   };
166
167   cout.setf(ios::fixed,ios::floatfield);
168   cout.precision(4);
169
170   for (int i = 0; i < atoi(argv[2]); i++)
171     {
172       // Start the timer
173       clock_t start = clock(), diff;
174
175       width=duallog(number);
176       if(verbose) cout << "allocating " << (3*width) << " quBits with " <<
177         (1<<(2*width)) << " terms.\n";
178
179       { // Shors's algorithm
180
181   int nreg1=2*width,nreg2=width;
182   quBaseState qubase(nreg1+nreg2,1<<nreg1);
183   quWord reg1(nreg1,0,qubase);
184   quWord reg2(nreg2,nreg1,qubase);
185   opVar op;
186
187   word x;   // base value
188   word mreg1,mreg2; // mesaurements of 1st and 2nd register
189   word pow;   // pow^2==1 mod number
190   word a,b;   // possible factors
191   word p,q;   // fraction p/q for rational approximation
192   double qmax;  // period and maximal period
193   int tries;    // number of selections
194
195   while(1) {
196
197
198     qubase.reset();     // reseting state
199
200     opFFT(nreg1)(reg1);   // 1st fourier transformaion
201
202     x=randcoprime(number);    // selecting random x
203
204     opEXPN(nreg1,nreg2,x,number)(qubase); // modular exponentiation
205
206     mreg2=reg2.measure().getword(); // measure 2nd register
207
208     opFFT(nreg1)(reg1);   // 2nd fourier transformation
209
210     qmax=1<<width;
211     tries=0;
212
213   reselect:
```

```
214
215     mreg1=reg1.select().getword();  // measure 1st register
216
217     tries++;
218     iter++;
219     if(mreg1==0) {
220
221       if(tries<maxtries) {
222         goto reselect;
223       } else {
224         continue;
225       };
226     };
227
228     // finding rational approximation for mreg1/rmax^2
229     approx((double) mreg1/(qmax*qmax),(int) qmax,&p,&q);
230
231     if(q&1) {
232       if(2*q<qmax) {
233         q*=2;
234       } else {
235         if(tries<maxtries) {
236     goto reselect;
237         } else {
238     continue;
239         };
240       };
241     };
242
243     pow=powmod(x,q/2,number); // pow = x^(q/2) mod number
244     a=(pow+1)%number;   // candidates with possible
245     b=(pow+number-1)%number;  // common factors with number
246
247     // testing for common factors with number
248     if(a>1 && (factor=gcd(number,a))>1) break;
249     if(b>1 && (factor=gcd(number,b))>1) break;
250
251     if(tries<maxtries) {
252       goto reselect;
253     } else {
254       continue;
255     };
256   };
257
258   // Calculate time
259   diff = clock() - start;
260   int msec = diff * 1000 / CLOCKS_PER_SEC;
261   printf("Time taken %d seconds %d milliseconds\n", msec/1000, msec%1000);
262
263       };
264     }
265
266   return 0;
267 }
```

B.2.2 Grover loop

```
// Edited by Johan Brandhorst for the purpose
// of measuring execution time. Increased
// certainty to > 99%

 qufunct query(qureg x,quvoid f,int n) {
  int i;
  for i=0 to #x-1 {      // x -> NOT (x XOR n)
    if not bit(n,i) { Not(x[i]); }
  }
  CNot(f,x);             // flip f if x=1111..
  for i=0 to #x-1 {      // x <- NOT (x XOR n)
    if not bit(n,i) { !Not(x[i]); }
  }
}

operator diffuse(qureg q) {
  H(q);                  // Hadamard Transform
  Not(q);                // Invert q
  CPhase(pi,q);          // Rotate if q=1111..
  !Not(q);               // undo inversion
  !H(q);                 // undo Hadamard Transform
}

operator search(qureg q,int n) {
  int i;
  qureg f[1];
  for i=1 to ceil(sqrt(2^#q)) {
    query(q,f,n);
    CPhase(pi,f);
    !query(q,f,n);
    diffuse(q);
  }
}

procedure grover(int n, int l) {
  //int l=floor(log(n,2))+1;        // no. of qubits
  int m=ceil(pi/4*sqrt(2^l));     // no. of iterations
  int x;
  int i;
  qureg q[l];
  qureg f[1];
  print l,"qubits, using",m,"iterations";
  {
    reset;
    H(q);                // prepare superposition
    for i= 1 to m {      // main loop
      query(q,f,n);      // calculate C(q)
      CPhase(pi,f);      // negate |n>
      !query(q,f,n);     // undo C(q)
      diffuse(q);        // diffusion operator
      print "iteration" , i, "completed";
    }
    measure q,x;         // measurement
    print "measured",x;
  } until x==n;
  reset;                 // clean up local registers
}
```

B.2.3 Deutsch-Jozsa's algorithm

```
// Deutsch-Jozsa Algorithm with CNot as the function evaluated,
// by Johan Brandhorst

procedure deutsch(int N) {
  qureg x[N+1]; // N+1-qubit register
  int n = 0;

  reset;       // Set the state to |0(N*0)>
  Not(x[N]);  // Set the state to |1(N*0)>
  print "The Input:";
  dump x;

  H(x);      // Hadamard on all qubits
  print "Hadamard(N+1):";
  dump x;

  // Function to be tested
  // CNot with N as target and 0 as control.
  CNot(x[0],x[N]);
  print "CNOT(1->N+1):";
  dump x;

  H(x[0:N-1]);  // Hadamard on qubit 0-N
  dump x;

  // Measure the first N qubits to determine if function
  // is constant or balanced
  measure x[0:N-1],n;

  reset;       // Clean up

  if n == 0 {
      print "Function is constant";
    } else {
    print "Function is balanced";
  }
}
```

B.3 Eqcs

B.3.1 Deutsch-Jozsa's algorithm

```
1  // Deutsch-Josza Algorithm implementation by Johan Brandhorst.
2  // Implemented into EQCS 0.0.8
3
4  #include "eqcs_state.h"
5  #include "eqcs_qc.h"
6  #include "eqcs_lambda.h"
7  #include "gates.cc"
8  #include <iostream>
9  #include <cstdlib>
10 #include <cctype>
11
12 int main(int argc, char** argv)
13 {
14   // Simulate random behaviour of QC
15   randomize();
16
17   int verbose = 0;
18
19   if (argc < 2 || !atoi(argv[1]))
20     {
21       cout << "Usage: ./deutsch [number of qubits] v [for verbose mode on]"
22      << endl;
23       return 0;
24     }
25
26   int N = atoi(argv[1]);
27
28   if (N < 1)
29     {
30       cout << "The Number of Qubits must be > 0" << endl;
31       return 0;
32     }
33
34   if (argc > 2 && *argv[2] == 'v')
35     verbose = 1;
36
37   // Create the state |1(N*0)> (that is, 1 followed by N zeros).
38   unsigned long init = (1u << N);
39   EqcsState state(init);
40
41   // Create the Quantum Computer in state |1(N*0)>
42   EqcsQc qc(state);
43
44   // Create the GateArrays
45   EqcsGateArray deutsch1, deutsch2, deutsch3;
46
47   // Run Hadamard on all qubits and put into the GateArray deutsch
48   hadamard(deutsch1, N+1);
49
50   // Run the CNOT gate, the function to be determined whether it is
51   // balanced or constant. Qubit N is target and Qubit 0 is control.
52   cnot(deutsch2,0,N);
53
54   // Run Hadamard on the N first qubits
55   hadamard(deutsch3, N);
56
57   // Run the algorithm
58   qc.perform(deutsch1);
59   if (verbose)
60     cout << "Hadamard(N+1): " << bits(qc.state(), 1, N+1) << endl;
61
62   qc.perform(deutsch2);
63   if (verbose)
64     cout << "CNOT(1->N+1):  " << bits(qc.state(), 1, N+1) << endl;
65
66   qc.perform(deutsch3);
67   if (verbose)
```

```
68      cout << "Hadamard(N):   " << bits(qc.state(), 1, N+1) << endl;
69
70    // Return the result of the test (Measure the N first qubits together)
71    // 2^N - 1 = The Binary mask used for calling what qubits to measure.
72    // N = 1; measure(1), N = 2; measure(3) (qubit 1 and 2 simultaneously).
73    cout << "The result; If 1 the function is balanced, if 0 it is constant"
74         << endl << bits(qc.measure(pow(2,N) - 1), 1, 1) << endl;
75
76    return 0;
77  }
```

B.3.2 Quantum gate definitions

```
1  // Help functions for the simulation of Algorithms in Eqcs-0.0.8
2
3  #include "eqcs_state.h"
4  #include "eqcs_qc.h"
5  #include "eqcs_lambda.h"
6  #include <iostream>
7  #include <cstdlib>
8  #include <ctime>
9
10 // Hadamard Gate by Peter Belkner
11 // Applies the Hadamard Gate on n qubits, n >=1.
12 void hadamard (EqcsGateArray &a, int n)
13 {
14   // Coefficient of the Hadamard Gate.
15   const static complex<double> u = 1.0/sqrt(2.0);
16
17   // Create a sequence of Hadamard gates on qubit i.
18   for (int i = 0; i < n; i++)
19     {
20       EqcsLambda h(u,u,u,-u,0);
21
22       h.set(0,i);
23       a.push_back(h);
24     }
25 }
26
27 // Simple CNOT with configurable control and target qubits.
28 void cnot(EqcsGateArray &a, int control, int target)
29 {
30   EqcsLambda cnot(0.0, 1.0, 1.0, 0.0, 1);
31   cnot.set(0, target);
32   cnot.set(1, control);
33
34   a.push_back(cnot);
35 }
36
37 // Applies a Controlled-NOT operation with qubit i as control and i+1 as ↩
       target
38 // for n qubits, n >= 1.
39 void cnotgate(EqcsGateArray &a, int n)
40 {
41   for (int i = 0; i < n; i++)
42     {
43       EqcsLambda c(0.0, 1.0, 1.0, 0.0, 1);
44
45       c.set(0,i+1); // Target
46       c.set(1,i);   // Control
47       a.push_back(c);
48     }
49 }
50
51 void toffoli (EqcsGateArray &a, int N)
52 {
53   // Qubit N+1 is target and Qubit 0 - N-1 are controls.
54   EqcsLambda Toffoli(0.0, 1.0, 1.0, 0.0, N);
55
56   // The target qubit is N (0:N)
57   Toffoli.set(0, N);
58
59   // All the other bits are controlling
60   for (int i = 0; i < N; i++)
61     Toffoli.set (i+1, i);
62
63   // Insert into GateArray
64   a.push_back(Toffoli);
65 }
66
67 // Creates a chain of logical quantum Not-gates and puts them into the
68 // input GateArray. N determines how many NOTs are inserted, from
69 // qubit 0 to qubit N.
```

```
70  void notgate (EqcsGateArray &a, int n)
71  {
72    for (int i = 0; i < n; i++)
73      {
74        EqcsLambda c(0.0, 1.0, 1.0, 0.0, 0);
75
76        c.set(0,i);
77        a.push_back(c);
78      }
79  }
80
81  void randomize ()
82  {
83    // Simulate random behaviour of quantum computer
84    srand(time(NULL));
85    int n = rand() % 8;
86
87    // Runs the pseudo-random function a random number of times to combat
88    // the likeliness of random valued created next to each other.
89    for (int i = 0; i < n; i++)
90      rand();
91  }
```

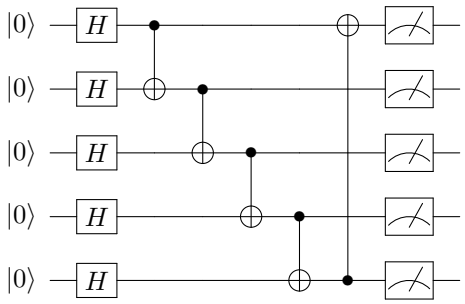
Appendix C

5 CNOT-algorithm

Appendix C: 5 CNOT-algorithm This algorithm was originally planned to be used to compare the performance of the different simulators, but was scrapped after further consideration. The work performed is presented below.

C.1 Five CNOT-gate testing algorithm

This is an algorithm where we put five CNOT-gates in a row after eachother and perform a measurement. This was interesting because it has been shown that a five qubit cluster state such as the one created cannot be simulated properly through classical bit correlation [17]. Unfortunately the configuration tested did not lead to the state desired and as such the algorithm was not as special as hoped. Shown below is the graphical representation of the algorithm that was implemented on the different simulators.



C.1.1 LibQuantum

The construction of the algorithm was simple, the functions needed were thoroughly explained in the documentation and with the standard C functions it was easy to implement other functions around the primary objective to allow for configurability such as different inputs allowing for more detailed output of quantum states between operations.

```
1  /*
2    Implementation of a chain of 5 CNOT-gates in LibQuantum.
3    By Johan Brandhorst
4  */
5
6  #include <quantum.h>
7  #include <time.h>
8  #include <stdlib.h>
9  #include <stdio.h>
10
11 int main(int argc, char **argv)
12 {
13   // Perform random seed for the simulation of quantum behaviour
14   srand(time(0));
15   int N;
16
17   int verbose = 0;
18
19   if (argc > 2)
20     {
21       printf("Usage: ./5cnot v [for verbose mode]");
22       return 0;
23     }
24
25   if (argc > 1 && *argv[1] == 'v')
26     verbose = 1;
27
28   // Create new Quantum Registry, 5 qubits, start value 0.
29   quantum_reg reg;
30   reg = quantum_new_qureg(0, 5);
31
32   // Print Registry Value
33   printf("The Input:\n");
34   quantum_print_qureg(reg);
35
36   // Perform hadamard transform on qubits
37   quantum_walsh(5, &reg);
38   if (verbose)
39     {
40       printf("Hadamard:\n");
41       quantum_print_qureg(reg);
42     }
43
44   // Perform 5 cnot operations
45   // 0-3 control, 1-4 target
46   for (N = 0; N < 4; N++)
47     quantum_cnot(N, N+1, &reg);
48   // 4 control, 0 target
49   quantum_cnot(4, 0, &reg);
50
51   if (verbose)
52     {
53       printf("CNOT:\n");
54       quantum_print_qureg(reg);
55     }
56
57   // Print the result
58   printf("The Resulting State:\n");
59   quantum_print_qureg(reg);
60
61   return 0;
62 }
```

C.1.2 QCL

The construction was in this case extremely simple, although again the lack of a proper quantum state print function is frustrating, and there is lack of certain functions that are standard in the big programming languages and that would make the program easier to use for the user.

```
1  // 5cnot-program by Johan Brandhorst
2
3  procedure cnot() {
4    qureg x[5]; // 5-qubit register
5
6    reset;   // Reset state
7    H(x);  // Hadamard
8
9    CNot(x[1],x[0]);
10   CNot(x[2],x[1]);
11   CNot(x[3],x[2]);
12   CNot(x[4],x[3]);
13   CNot(x[0],x[4]);
14
15   // Print the state
16   dump x;
17 }
```

C.1.3 Eqcs

Straightforward implementation with the use of some C++ functions to make the program more fun to use for the user. Once you've gotten used to the syntax in Eqcs it is quite straightforward to use.

```
1  // 5cnot program by Johan Brandhorst. Implemented into EQCS 0.0.7
2
3  #include "eqcs_state.h"
4  #include "eqcs_qc.h"
5  #include "eqcs_lambda.h"
6  #include <iostream>
7  #include <cstdlib>
8  #include <ctime>
9  #include "gates.cc"
10
11 int main(int argc, char** argv)
12 {
13   // Simulate random behaviour of QC
14   randomize();
15
16   int verbose = 0;
17
18   if (argc > 2)
19     {
20       cout << "Usage: ./5cnot v [for verbose mode on]"  << endl;
21       return 0;
22     }
23
24   if (argc > 1 && *argv[1] == 'v')
25     verbose = 1;
26
27   EqcsGateArray cnotprogram1, cnotprogram2;      // A gate array.
28
29   // Run the Hadamard gate on all 5 qubits and put into
30   // the cnotprogram GateArray
31   hadamard(cnotprogram1, 5);
32
33   // Run the CNOT gate on 4 qubits with i as control and i+1 as target
34   // and put into the cnotprogram GateArray
35   cnotgate(cnotprogram2, 4);
36
37   // Run the specialized CNOT gate with 5th as control and 1st as target.
38   cnot(cnotprogram2,4,0);
39
40   // Create state all qubits = 0;
41   EqcsState state = 0ul;
42
43   // Create a quantum computer initially in state |00000>.
44   EqcsQc qc(state);
45
46   // Print the input
47   cout << "The input:  " << bits(qc.state(), 1, 5) << endl;
48
49   // Run Hadamard.
50   qc.perform(cnotprogram1);
51   if (verbose)
52     cout << "Hadamard: " << bits(qc.state(), 1, 5) << endl;
53
54   // Run CNOTs.
55   qc.perform(cnotprogram2);
56   if (verbose)
57     cout << "5 CNOT:   " << bits(qc.state(), 1, 5) << endl;
58
59   // Print the result.
60   cout << "The result: " << bits(qc.state(), 1, 5) << endl;
61
62   return 0;
63 }
```