Formal Report for Assignment 2: Quantum Computing

Adam Alderton
Physics Department, University of Exeter
(Dated: March 26, 2021)

Quantum computers show promise to revolutionise fields such as chemistry and cryptography by the ability to use algorithms that leverage the effects of quantum mechanics [1]. One such algorithm is Shor's algorithm which is able to factorise large integers more efficiently than classical computers in terms of both speed and size [1]. Presented here is a discussion regarding a simulation built to investigate this algorithm. The simulations were able to factorise integers on the order of 10^1 and the results were shown to be physical as the total probability was conserved to a deviation of 2.4×10^{-15} which can be attributed to the precision of the double data type [2]. The limiting resource to the simulation was found to be computation time and hence the simulation could be improved by the introduction of parallelism.

I. INTRODUCTION

Many discuss future computing paradigms such as general purpose parallel computing, photonic computing and quantum computing which may have impactful roles to play in the future of computing [3]. Quantum computing is the field in which the principles of quantum superposition and entanglement are leveraged to be able to perform certain computations with entities called *qubits* [1]. Qubits are the quantum analogue of classic bits being that they are capable of holding data superposed within them [1]. That is, they can hold either 1 or 0 (as in the classical case) or a hold a superposition between 1 and 0. It is important to note that by putting a qubit into superposition, we are not just masking our view of whether a 1 or a 0 is inside. Instead, the qubit physically is partly both, which can be demonstrated by observing the effects it imposes on other qubits when undergoing an interaction [1].

Modern research and development in the field of quantum computing generally is in one of two threads. One thread addresses the engineering challenge of realising physical quantum computers and the other thread concerns the development of applications of quantum computers [1]. A sub-genre within the development of applications is the development of quantum algorithms and circuits [1, 4]. That is, algorithms which leverage the capabilities of quantum computers to out-perform classical computers, and the design of quantum circuits with which to realise these algorithms [4]. Perhaps the most famous algorithm developed is *Shor's algorithm* which can factorise large numbers which otherwise may lie beyond the capabilities of classical computers [1, 4]. Factorising incredibly large numbers in a reasonable timescale has many implications on the modern world as much of the basis of secure communications is built on the fact that it is virtually impossible to classically factorise some incredibly large numbers [1]. Should these sophisticated encryption algorithms be able to be cracked by quantum computing, secure communication in the modern world may be difficult to maintain [1].

II. SHOR'S ALGORITHM

This implementation of a simulation of a quantum circuit capable of executing Shor's algorithm was carried out by following the instructions and guidance as laid out by D. Candela in their 'Undergraduate Computational Physics Projects on Quantum Computing' [4]. The following introduction to Shor's algorithm is derived from discussion within the mentioned paper and is mentioned here in the interest of clarity and consistent notation moving forward. The basic steps to Shor's algorithm are as follows:

1. Consider a number C to be factored. Ensure that C cannot be trivially factorised by classical means by considering whether it is even or a power of a small integer. If it is even or a power of a small integer, factors have been found and the algorithm can exit.
2. Pick a 'trial integer' a in the interval $1 < a < C$.
3. Find the greatest common divisor between a and C classically via Euclid's algorithm [4]. If this shared divisor (factor) is greater than 1, a non-trivial factor has been found and hence the algorithm can exit here.
4. Find the smallest integer $p > 1$ such that $a^p = 1(\text{mod } C)$. This step is often called *period finding.*
5. If p is odd, or if p is even and $a^{\frac{p}{2}} = -1(\text{mod } C)$, go back to step 2 and trial a different a.
6. If p has passed the mentioned requirements, non-trivial factors of C will be the greatest common divisors between C and $a^{\frac{p}{2}} \pm 1$.

This report concerns the implementation and the discussion of a simulation of a quantum circuit designed to perform Shor's algorithm to factor positive integers. The very thing that makes quantum computers powerful, parallel computing by superposition, is what makes simulating a quantum computer so difficult classically. The number of states which can be represented by N qubits grows as O(2^N), which is excellent for the interest of quantum computers but can mean that classical simulations must consume an exponential amount of memory as a function of N [1, 4]. Additionally, as a quantum computer essentially facilitates a large number of parallel classical computations, the number of classical computations to simulate increases exponentially [1, 4]. Therefore, memory efficiency and compute efficiency is pertinent in the implementation of a simulation of a quantum computer. The crux of this project was how best to develop the simulation within the guidance. The completed simulation should simulate the quantum circuit denoted in FIG. 1, which, when executed, should result in a measurable state from which the factors of a number passed to the program can be found [4].

III. METHODS

Following are short discussions and justifications regarding some implementation details within the produced simulation, and how they complement the outlined guidance and instructions [4].

A. Tailoring Shor's Algorithm

Due to the memory consumption and compute power needed as discussed Section I, the simulation is only (realistically) capable of reliably factorising numbers less than 100 [4]. With this in mind, the classical checks outlined in step 1 of Shor's algorithm have been largely omitted such that the use of the quantum gate array compute is forced.

Additionally, a decision had to be made regarding how to choose a trial integer a in step 2. The trial integer can be specified by the user but if it is not then the possible trial integers are looped over, starting from 2 and ending with $C - 1$. Raising smaller integers to large powers is less likely to cause an overflow error than raising large integers to large powers, so with luck a period hence factors can be found with smaller trial integers.

Warnings are implemented into the program to advise the user as to the minimum register sizes by which to be confident that the algorithm will correctly factorise the given integer. The M register needs to be able to hold the binary representation of $f(x) = a^x(\text{mod } C)$, so the size of the M register must satisfy $2^M >= C$ [4]. Additionally, to be confident in finding a period hence factors, the size of the L register should satisfy $2^L >= C^2$ [4].

B. The Wavevector Pointer Swapping

In-place sparse matrix-vector multiplications are very difficult to achieve hence an additional vector of storage is needed to store the result of the multiplication [5, 6]. In the simulation, these quantities are named *current_state* and *new_state* in the program respectfully. Naturally, a gate operation matrix is operated on *current_state* and the result is then stored in *new_state*. The contents of *new_state* should then be stored in *current_state* in preparation for the next gate operation. One means to consider achieving this is to copy the memory across as GSL neatly provides a function to achieve this [5]. However, the choice was made to actually make *current_state* and *new_state* pointers to (the pointers to) the vector storage quantities. Therefore, on the operation of a matrix, the pointers can simply be swapped to give the correct *current_state*. The benefits of this slightly more complex approach are clear when many qubits are considered as the unnecessary copying of 2^N doubles is circumvented.

C. Building Matrices with Bitwise Operations

As per the guidance by Candela, the tensor product operations to build the gate operation matrices can be achieved by a combination of Dirac deltas [4]. The binary representations indices i and j of a matrix element M_{ij} should be compared. If any of the digits in the binary representations do not match (aside from the *qubit_num*'th), then the element will be 0. Otherwise, the element can be extracted from a 'basis' matrix [4].

As opposed to complex string manipulation or similar, the actual binary values stored in memory representing the integers i and j were used in the simulation. C is often used for low level development and other memory precious tasks hence it has bitwise operations built-in [2]. To mimic the effects of the combined Dirac deltas an exclusive or bitwise operation is applied to i and j. For clarity, a bitwise NOT operation is then applied to yield the final full effect of the combined Dirac deltas. If any binary digit in the result is 0 (apart from the *qubit_num*'th), then M_{ij} is zero.

The indices i and j are stored as unsigned long integers such that the program can accommodate up to 32 qubits instead of 16 as available with the integer data type.

D. Using Sparse Matrices

As advised by Candela, sparse matrices can be used as to not have to store and compute many matrix elements, many of which would otherwise be zero [4]. Sparse matrices can be stored in many formats, three of the most popular being coordinate based (COO), compressed column based (CSC) and compressed row based (CSR) [5].

The CSR format is efficient in matrix-vector multiplications due to individual dot-products between each row and the column vector, especially if the process is parallelised [5, 7]. However, building sparse matrices in this format is challenging [5]. Therefore, in deciding which format to store and operate the matrices in, a balance had to be struck between building efficiency, operating efficiency, simplicity and the cost to convert between the

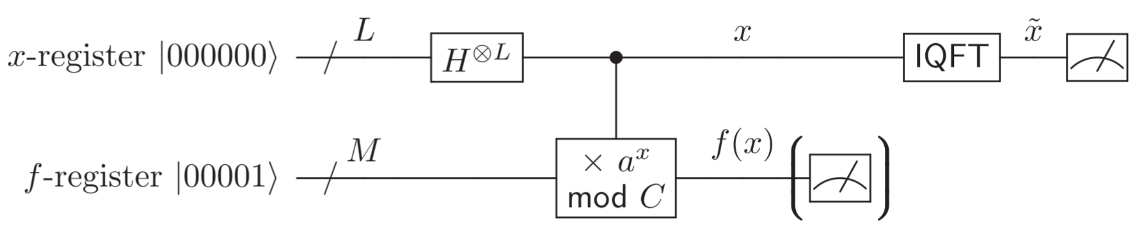

FIG. 1. The quantum circuit used to leverage quantum computing to find the period of the function $f(x) = a^x (\text{mod } C)$, as per the discussion of Shor's algorithm in Section I. The x register contains L qubits and is subject to both Hadamard gates and an inverse quantum Fourier transform which itself consists of Hadamard gates and phase-shift gates [4]. The f register contains M qubits and is subject to the $f(x)$ function dependent on the state of the x register, which of course is in superposition of all possible x due to the application of the Hadamard gates. These operations are performed left-to-right in the diagram. On completion of the circuit, the $\tilde{x}$ register should contains a value approximately equal to $2^L \times \frac{s}{p}$, where s is an unknown integer and p is the period to be found [4].

storage types, if applicable. Due to many matrices in the simulation being particularly sparse, such as the $f(x)$ matrices, the balance struck resulted in the COO format being the format of choice within the simulation. This prevented the need to store two matrices for subsequent compression. Additionally, GSL does not provide a complex sparse matrix - complex vector multiplication function so the use of the COO format enabled a function to do the multiplication to be written easily and simply.

E. Period Guessing

As outlined by the guidance in the Candela paper, the measured 'frequency' $\omega = \tilde{x}/2^L$ should be close to some integer harmonic of the inverse of the period [4]. That is, $\omega \approx \frac{n}{p}$ [4]. A means by which to attempt to convert this continuous quantity to an integer period is by using the continued fractions expansion of ω to guess the period [4]. By building the expansion, a series of denominators of these fractions can be built. Multiples of these denominators (potential periods) can be tested against the condition $a^p = 1(\text{mod } C)$.

This is done as to avoid repeatedly measuring the state of the register to find the period. As per the basics of quantum mechanics, any subsequent measurements of a state will not yield different results so allowing the repeated measurement of the state without repeated quantum computation would not only be against the spirit of the task, but also against the physics of real quantum computers [4].

F. Datatype Notes

In the interest of memory and compute efficiency, considerations were given to the data types of quantities in the simulation. As discussed in Section III C, matrix indices are stored and computed as unsigned long integer data types when building the matrices.

As to be discussed in Section IV, the limiting factor in the computations seemed to overwhelmingly be processing speed. Therefore, one of the reasons the state vectors are stored as double precision is to prevent the repeated casting of floats to doubles which can slow the calculations [2]. Consequently, the dramatic reduction in truncation/round-off that the double precision provides comes at almost no expense which assists in keeping the vector states normalised and hence physical.

A discussion was had regarding the use of something other than complex matrices but complex matrices were eventually settled upon for simplicity and future flexibility alongside the reasons mentioned concerning the storage of double precision complex vectors.

IV. INVESTIGATION

A. Conservation of Probability

Due to the high precision of the double datatype, normalisation checks have not been implemented within the simulation in the interest of processing power. Assuming the correct implementation of the matrices and vectors, the total probability should deviate from 1.0 only by negligible amounts due the truncation of the irrational numbers used within the program such as π and $\sqrt{2}$ and due to round-off error associated with the double precision floating point calculations.

To test this deviation, the total probability was tracked over the course of the factorisation of 39 into 13 and 3, with 6 qubits in the L register and 6 in the M register. The total probability during the application of the matrices is shown in FIG. 2. The probability deviates to a maximum of $1.0 + (2.4 \times 10^{-15})$. An argument can be made to attribute this deviation to the round-off associated with the double type as the IEEE 754 specification of the double type has a precision up to approximately 10^{-16} [2]. The deviation increases up to its maximum as the inverse quantum Fourier transform is carried out indicating that the truncation and round-off error asso-

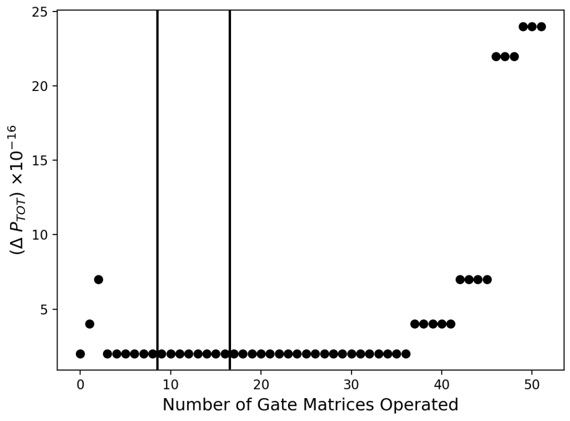

FIG. 2. The deviation in the total probability during the factorisation of 39 using 12 qubits, with 6 being in the L register. The operations before the first vertical line correspond to applying Hadamard gates to the L register. The matrices applied between the two vertical lines correspond to the application of the $f(x)$ gates, and the matrices after the second vertical correspond to the operation of the inverse quantum Fourier transform. The deviation has an initial minimum of 2×10^{-16} and reached a maximum of 2.4×10^{-15}.

ω	(Candela) $N_S = 100$ [4]	$N_S = 100$ (Avg.)	σ
0.0	27	25.6	3.93
0.25	25	23.2	1.47
0.5	30	25.4	3.00
0.75	18	25.6	2.87

TABLE I. The results of running Shor's algorithm to factorise 15 with register sizes $L = 3$ and $M = 4$ and with a trial integer of $a = 7$, compared to the results of Candela running a simulation with the same parameters [4]. The simulation was run 100 times by Candela and 100 × 5 times here to compare. Given are the average occurrences of $\omega = \frac{\tilde{x}}{2^L}$ which are harmonics of $\frac{1}{4}$ giving a correct period of 4. Also given are the standard deviations of the occurrences over the five runs of 100.

ciated with the phase-shift matrices may be having an increasingly relevant effect.

The probability deviation shows an initial spike of 7×10^{-16} as the first three matrices are applied. Further investigation and discussion would be needed to find the cause of this spike, and why it resets itself to 2×10^{-16}.

B. Accuracy

The program is capable of factorising integers on the order of 10^1 assuming the register sizes satisfy the conditions outlined in Section III A including those provided in Table V. of the Candela guidance paper [4]. To test the physicality and accuracy beyond this, results were compared to results given in the Candela guidance [4]. Namely, 15 was factored with the registers $L = 3$ and $M = 4$ with the forced trial integer $a = 7$ such that a period of 4 is given. The results of 100 computations and measurements, run five times and averaged, and how they compare to the results of Candela, can be observed in TABLE I. As the Candela guidance implies it should be, the measured ω is evenly distributed between the harmonics of $\frac{1}{p} = \frac{1}{4}$. Interestingly, the average occurrence of $\omega = 0.25$ is slightly less than that of the other harmonics, and has a slightly lower standard deviation. More investigation may be needed to deduce as to whether this is a statistical anomaly, an issue with the simulation, or an artifact of the time seeded Mersenne twister based random number generation used to measure the states.

C. Execution Times

As expected, and with agreement to the results achieved by Candela, the simulation computing time increases rapidly with increasing qubit register sizes [4]. Particularly, the computation time increases very quickly for increasing L as denoted in FIG. 3. Due to the linked list storage of the sparse complex matrices with the factorially based computation of the inverse quantum Fourier transform it may be difficult to postulate the time complexity of the computation as a function of the register sizes [4]. On inspection, one could argue that increasing the size of the M register would introduce a time complexity similar to the form $O(M \cdot 2^N \cdot 2^N) = O(M2^{2N})$ where $N = L + M$ is the total number of qubits. This time complexity contains the contribution of the M needed $f(x)$ gates along with the contribution of the extra qubits to the matrix-vector multiplications. Also on inspection, one could argue that increasing the size of the L register could introduce a time complexity comparable to $O(L \cdot L \cdot L! \cdot 2^{2N})$. These contributions would be the application of L Hadamard gates, L $f(x)$ gates and the inverse quantum Fourier transform and the increased size of the matrix-vector multiplications respectively.

These assumptions would agree with the quicker increase of execution time as a function of L shown in FIG. 3. However, due to the fact that only a very limited number of qubits could be tested in a reasonable time, alongside the fact that analysis of the time complexity of the multiplication of the linked list based sparse matrices is difficult, further simulation and investigation would be needed to be able to deduce conclusively how the execution time grows as a function of L and M.

D. Parallelisation Discussion

The investigations have shown that computing time is a much more precious resource to the calculation than memory. Analogous to the benefits that physical quantum computing provides, the simulation would benefit from the implementation of parallelism. This is apparent when one considers the number of matrix-vector cal-

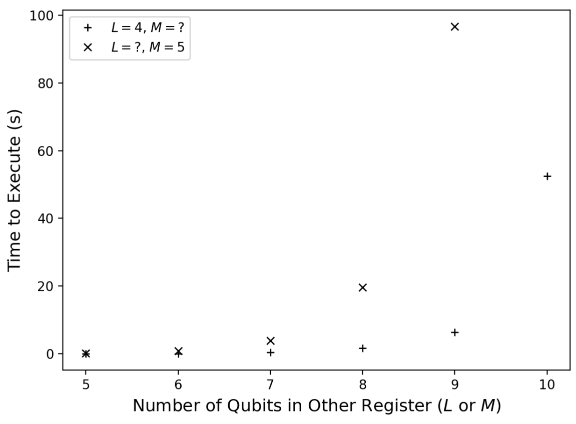

FIG. 3. Execution times of computations factorising 21 into 3 and 7 using Shor's algorithm using a forced trial integer of $a = 2$. Measured are the execution times due increasing the size of a sub-register L or M, keeping the size of the other constant. Shown are the average times of the calculations when run five times. As expected, computation time increases much quicker with increasing L, due to the application recursive inverse quantum Fourier transform.

culations that have to be done and the embarrassingly parallel nature of this talk. Namely, the multiplication of each row of the matrix with the vector is completely independent of any other row-vector multiplications.

Parallelism may be difficult to introduce into the simulation as it may struggle to interface with GSL which itself does not inherently support any kind of parallelisation [5]. However, it promisingly advertises that all of its functions a thread safe so introducing parallelisation may not involve a complete overhaul [5]. The first part of the program to be parallelised would be the sparse matrix vector multiplications and perhaps a relatively simple kernel/multithreaded program could be written to be able to build the elements of the gate matrices in parallel, as of course each of the elements are independent of each other.

V. CONCLUSION

The program built is able to run Shor's algorithm to factorise positive integers and includes the simulation of the quantum gate array compute to parallelise the 'period finding' step to Shor's algorithm. The results produced by the simulation can be justified to be physical due to measurements of the qubit register matching what is physically expected, and justified by the fact that the normalisation of the quantum states only deviates by a maximum of 2.4×10^{-15} which can be attributed to truncation and round-off error [2, 4].

It was found that the limiting resource to the simulation was the processing power/time as opposed to memory consumption therefore the clear next step in improving the simulation is to parallelise tasks such as matrix-vector multiplication and matrix construction.

The simulation project was an intuitive insight into the workings of quantum mechanics and highlights the benefits and future potential that quantum computers demonstrate.

[1] A. Hagar and M. Cuffaro, 'Quantum Computing', *The Stanford Encyclopedia of Philosophy*, (2019).

[2] S. Oualline, *Practical C Programming*, (O'Reilly, 2000).

[3] P. E. Ceruzzi, *A History of Modern Computing*, (MIT Press, London, 2003).

[4] D. Candela, 'Undergraduate Computational Physics Projects on Quantum Computing', *American Journal of Physics* **83**, 688 (2015).

[5] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, F. Rossi and R. Ulerich, 'GNU Scientific Library 2.6 Reference Manual', (2019).

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, (Cambridge University Press, Cambridge, 1992).

[7] N. Bell and M. Garland, 'Implementing Sparse Matrix-Vector Multiplication', *NVIDIA Research*, (2009).