SHANGHAI JIAO TONG UNIVERSITY

Shor Algorithm

Yuwei Wu, Hanjing Wang, Yan Hao

August 7, 2018

ABSTRACT

This is an algorithm operating prime decomposition using quantum algorithm in Q#. For the basic work, we finished coding the whole algorithm in two parts: the traditional part and the quantum part. For the advanced assignments, we first concluded some tips in debugging the project. Secondly, the Parallel Computation accelerates the whole process, consuming only one fifth time of computation without this technique. Thirdly, the Cross-platform technique enables users to get access to this algorithm using Python.

CONTENTS

1 OUR WORK

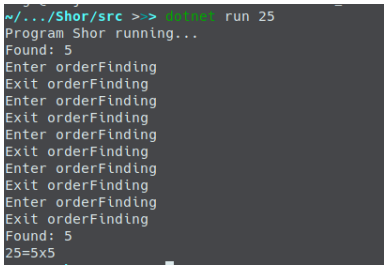
1.1 ALGORITHM

1.1.1 TRADITIONAL PART

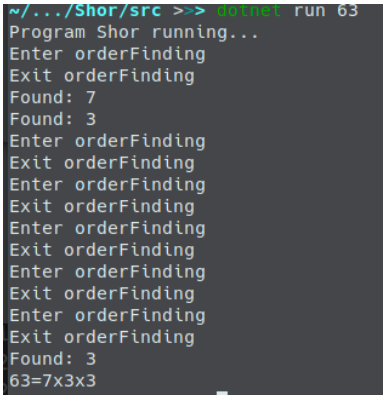
Traditional Algorithms are included in the Driver.cs module. The hierarchy of the file and its main purpose are listed as follows.

Main	The entry of the algorithm, accept sequential integers as input accept sequential integers as input
Procedure	The reduction of factoring to order-finding algorithm(following the instruction from *Quantum-Computation*
orderFinding	The order-finding algorithm, interacting between classcial and quantum computation.
ContinueFractionAlgorithm	Apply the continued fraction expansion algorithm and find the order r of $a^r \, mod N$
FpowMod	Fast algorithm for calculation the power of a mod N
Gcd	Greatest common divisor algorithm

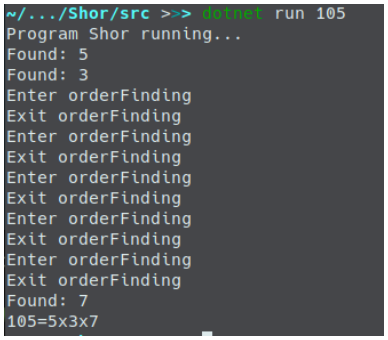
What's special in our Driver.cs?
The most significant part of our implemention of the classcial algorithm lies in the dynamic of our choice of L, T in phase estimation algorithm, which determined by the length of input binary integer and expected precision of result. Compared to other's implemention, we determine the l and t dynamically in the calculating procedure. That partly results from the fact that others' implemention only get single factor of the input and get another from division. That's a risky choice because if the quotient can be further factored. We take the factoring of quotient as a iterative task. It promises high probability of successful execution of our code.
And we optimize the algorithm by reduce the unnecessary code in the loops. For example, we break the loop getting a factor from random choice or order-finding and replace multiplication and division by bit operation. As a result of all these optimization, our program runs much faster than that of Kaiyi Zhang. *Additionally, our program has been proved by experimenets to safely factor integers less than* 2^6 *in a short period of time(less than 5 minutes).* The factoring of small integers like 15 or 21 only take several seconds. Number larger than 2^6 like 105, as long as given enough time(10 minutes or so), can be factored correctly. It's a significant advantage in performance over Kaiyi Zhang's implemention.



(a) Factoring 25



(b) Factoring 63



(c) Factoring 105

Figure 1.1: Examples of factoring with single thread.

1.1.2 Quantum Part

In Quantum Part, we first write the circuit for quantum Fourier transform using Hadamard gates and the controlled R_k gates to achieve a final state and finally using CNOT gates to do swap operations. We use Q# to perform operations and qiskit to visualize the circuit. The circuit we simulate for quantum Fourier transform algorithm is shown in the part 2 Result. For order finding algorithm, we then need to complete the inverse quantum Fourier transform circuit. It is quite easy after finishing the circuit for quantum Fourier transform since we just need to reverse the operations and use a reverse phase for the controlled R_k gates.
Next, we try to make the circuit for the controlled unitary operations, namely the controlled Ua which turn the state $|x>$ to $|a^j x \bmod N>$ given a, N. In the extensions for Q#, we find function ModularMultiplyByConstantLE that can turn an arbitrary state $|x>$ to $|ax \bmod N>$ given a, N, which implies that to get a circuit for the controlled unitary operation we only need to provide a way to calculate a^j. The fastest and safest way to do that is to do the fast power algorithm. Since the Q# does not have while loop, we use a 15-iteration for loop instead and to make sure the number not grow too big we do mod operations during each iteration. Also, because the function for power returns double type value and the fast power algorithm requires int type value, we write a function called power2 to calculate the power of 2 and return int type results. After referring to some paper[1], we get to know the controlled Ua gate consists of a series of ϕADD$(2^i a)$ Mod N gates and these gates are made up of doubly controlled ϕADD gates and QFT and QFT†. We then tried to draw the circuit for the controlled unitary operations using qiskit. And we finished building the gate for the ϕADD using qiskit but the qiskit does not have a visualization gate for the doubly controlled ϕADD which means we have to do a ABC decomposition on the unitary operation of ϕADD and draw a equivalent circuit for the doubly controlled ϕADD gate. However, it is too cumbersome for the whole circuit of the controlled Ua as the ϕADD is just a small unit of the circuit so we gave up drawing the circuit for this part in the end.
Finally, with all the tools prepared, it is not hard for us to perform the order finding algorithm. For the first place, we prepare the initial state by flipping the state of the second register ($qubits[t..t+l-1]$). Then apply $H^{\otimes t}$ on the state of the first register ($qubits[0..t]$).After that, we apply the controlled unitary operation on the second register as the target qubits and the first register as the control qubits. Next, we perform the inverse quantum Fourier transform on the first register. Finally, we do the measurement on the first register, which just finishes the quantum part.

1.2 Advanced Work

1.2.1 Debug

With the aid of Visual Studio, Q# supports some useful debugging capabilities

- setting line breakpoints
- stepping through code using F10

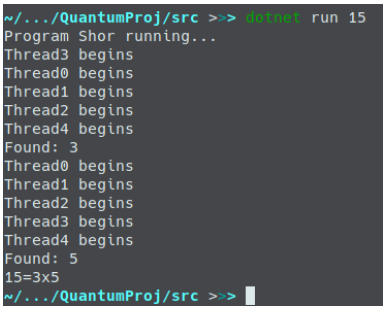
[1] *Circuit for Shor's algorithm using 2n+3 qubits* by Stephane Beauregard

- inspecting values of classic variables

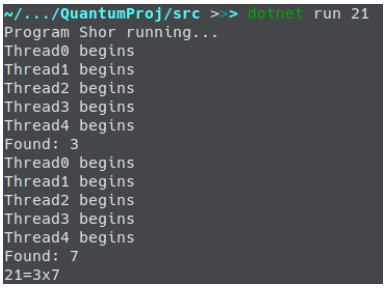
Another technique we used for reduce bugs is every time we write a operation, we add an unit test to test different value and check the answer to make sure every operation we write is right.
Also, we together maintain a debug log (see the file "project.log") to record every mistake we made during the program period. One of the biggest difficulties we confront is that some bugs will not come up immediately we run the program and after several times of running, they come up sometimes even for the number we tested that has succeeded factoring before. That makes the bugs hard to find and then hard to debug as well.

1.2.2 Parallel Computation

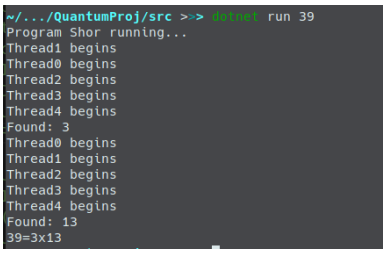
In our python interface(Shor.py) of the program, we using multi-threading techniques to factor sequential inputs parallelly. Furthermore, we have attached great importance to distangle the procedure of the factoring a single input, and we have achived it.
In the experimenets, we have successfully factor $15, 21, 39$ in less than $30s$ as is shown in the following pictures. However, when it comes to larger integers like 63, the algorithm consumes more time and are more likely to show a trend of failure, it may result in the relatively small times of trying in randomly choosing x and more bits of registers to simulate. Furthermore, because we apply synchronized multithreading technique to the factoring procedure, it is more likely to be infected by a terrible value of randomly chosed x. (Source code avaliable in the 'MultiThread' folder)



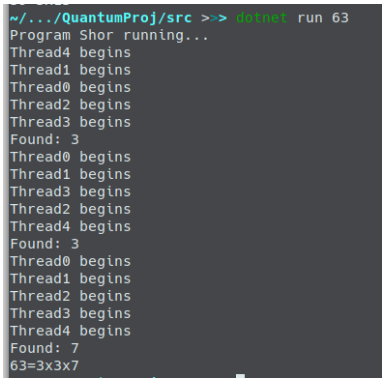
(a) Factoring 15



(b) Factoring 21



(c) Factoring 39



(c) Factoring 63

Figure 1.2: Examples of factoring with multi-threading.

1.2.3 Cross-platform Calls

We provide a python interface of Shor algorithm. To be featured, it accepts two special parameters "–v"(verbose) and "–p"(parallel) with the latter one determining whether it support parallel calculation. It factors sequential inputs parallelly.

Figure 1.3: Python Interface of Shor Algorithm

And a python function *ShorWrapper* is implemented. One can easily call it by transporting a list of integers that are supposed to be factored and getting the result in the list form [factored

integer, factor1, factor2,...].

2 RESULTS

The output circuit generating by qiskit is shown as follows. In the following example, we simulate a circuit consisting of 6 qubits for quantum register1 and 5 qubits for quantum register2 and 6 bits for the classic register.
In the first figure below, the circuit first prepare the initial state and the perform $H^{\otimes t}$. The next two figures show the implement of the inverse quantum Fourier transform.

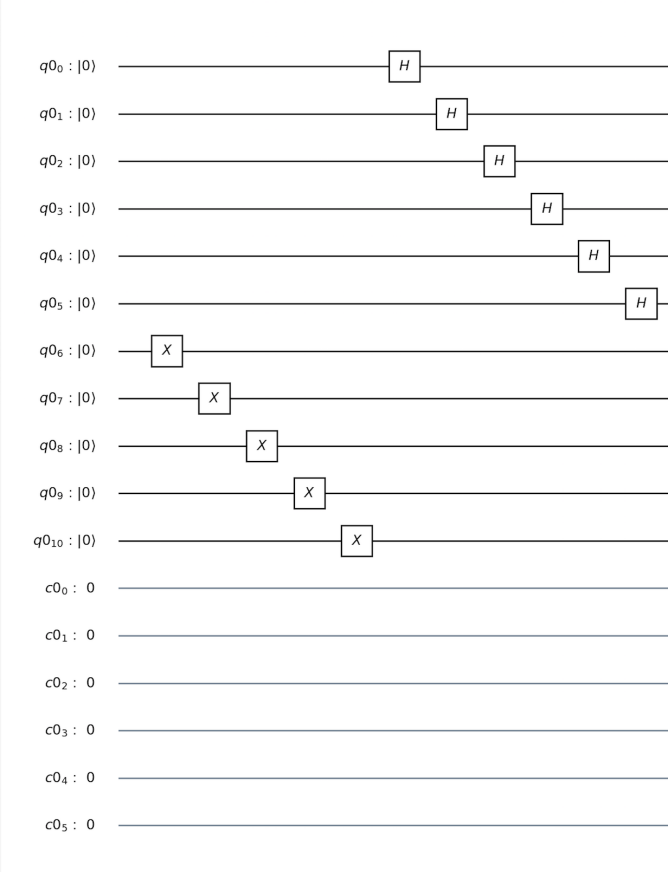


Figure 2.1: First part of the circuit

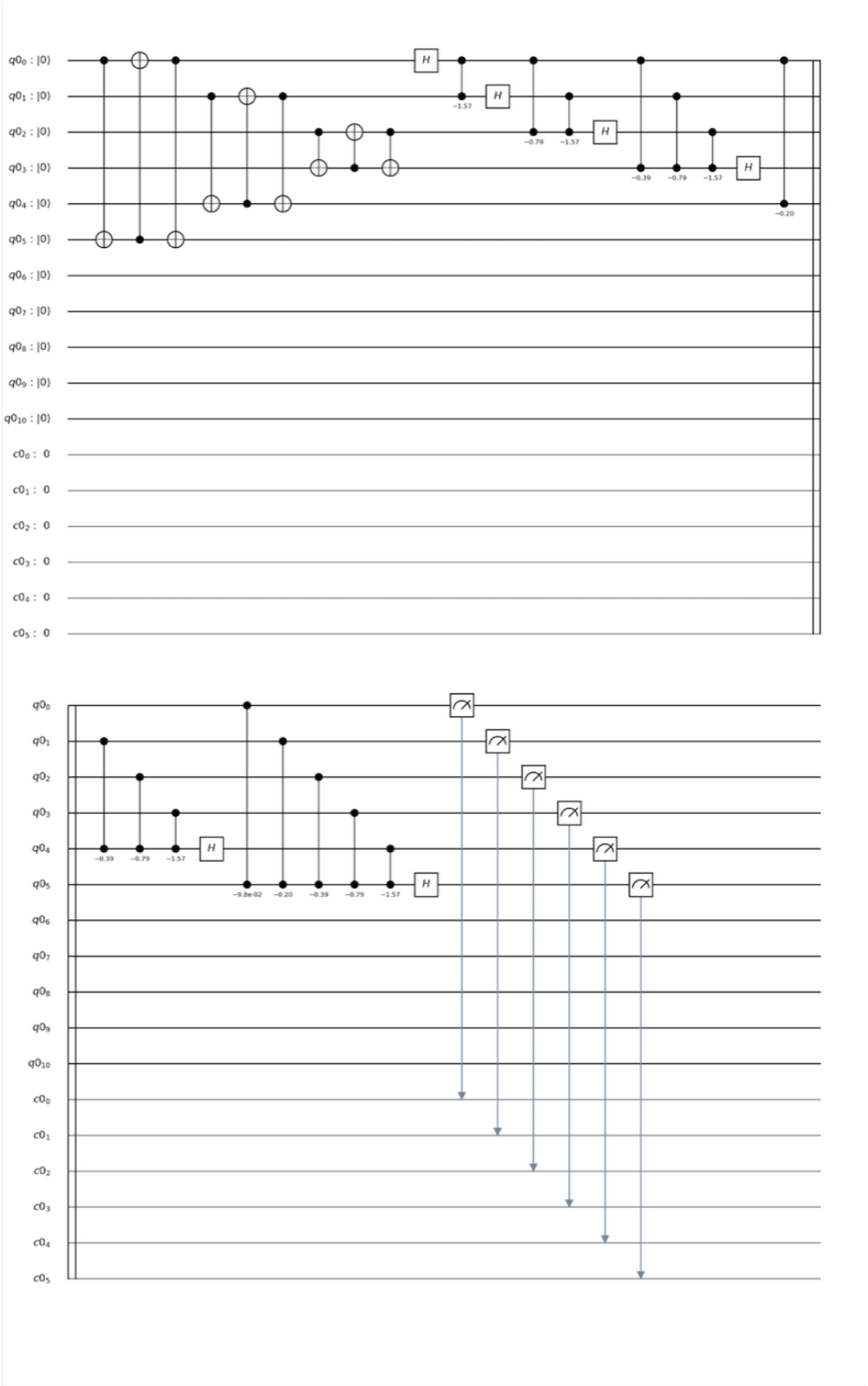


Figure 2.2: Third part of the circuit

The following are some examples of our decomposition.

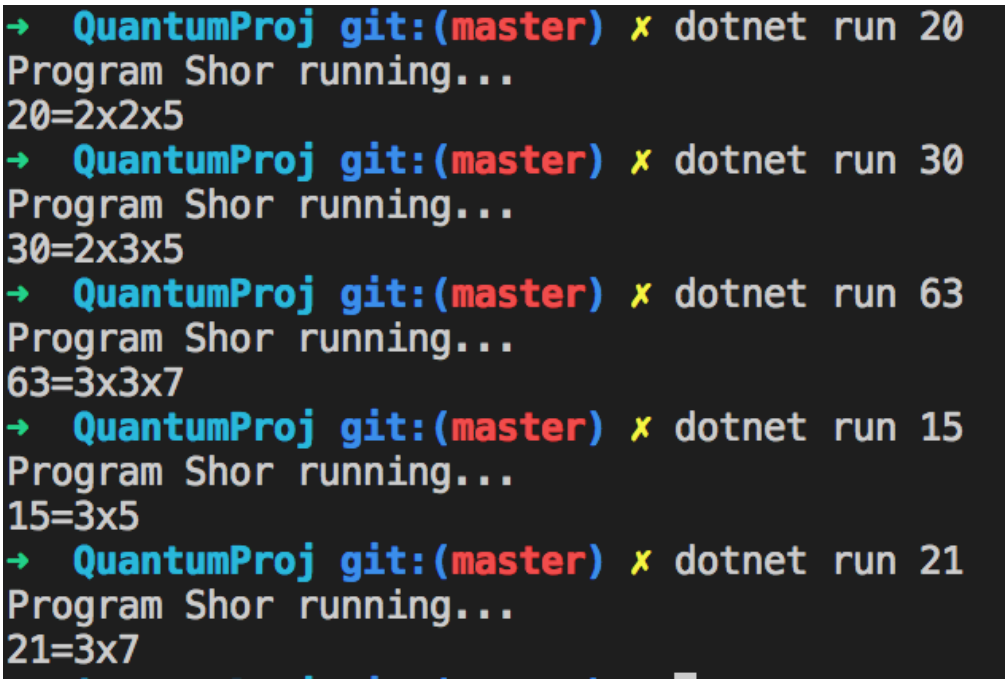


Figure 2.3: results of decomposition