Amanda Turney
ENGR-517
12/9/2022

Final Project Report

Moore's Law describes the observation made by Gordon Moore on the phenomenon of the number of transistors doubling about every two years on microchips. More generally though, this is the observation of computing power growing exponentially since about 1970 and can be attributed to not just transistors, but also an increase in clock rates, newer microarchitectures and architectures, faster networks, increased memory, and more efficient algorithms. These impressive strides in technology have all been built around a core concept of the Von Neumann architecture. Even more foundational though has been the dependence on semiconductor technology, Boolean operations, and the concept of a "bit" for the last 50+ years. However, quantum computing has become a revolutionary technology in not just its ability to shatter the current trend in growth of computing power but also in its departure from many of the long-standing foundational concepts found in classical computing. Quantum computing is built on the concept of a qubit, which holds information about the state of the smallest unit of data. But unlike a classical bit, a qubit isn't limited to 0 and 1; it can be in a superposition, or linear combination, of both 0 and 1 at the same time. The exponential performance gains that quantum computing can offer rely on a phenomenon called entanglement that happens between qubits and can increase processing speed. Quantum computing also comes with a set of quantum gates, or operations, that are performed on qubit(s) and can be used to create entirely new algorithms. Ultimately, quantum computing offers new ways of tackling problems that would be either impractical or incapable of being solved by classical computers.

Shor's Algorithm is *the* quantum algorithm, meaning it was the first non-trivial solution to be developed that had practical application to real-world problems and is what ignited this area of research. At the core, Shor's algorithm is a quantum algorithm for finding the period of a function (a^x mod N) which delivers polynomial speed up over exponential classical algorithms. Period finding is a critical step in efficiently finding prime factors of an integer. Thus, this project will focus on implementing a high-performance quantum simulation of Shor's algorithm in order to find p1*p2 = N where p1 and p2 are prime factors for a given N.

Building a quantum algorithm requires quantum gates; some of which have analogs to classical computing gates, but one of the most important and unique quantum gates is called the Hadamard. This gate operates on a single qubit and puts it into superposition, meaning that if the qubit originally measured in state 0, it would have a 50% chance of being measured in state 0 and a 50% chance of measuring in state 1 after applying the Hadamard. The power of these operators is that they create a fork within an algorithmic path, where one scenario continues with a state of 0 and another scenario continues with a state of 1 for the specific qubit that was transformed. A subsequent operation can be implemented just once, but it will be applied across all possible paths that the state(s) had taken up to that point. This ability to apply the same operations on different inputs can be thought of as same instruction on multiple data (SIMD) parallel is one of the ways that quantum computers get their speed. However, unlike classical computers, this parallelization comes for free as it doesn't require any hardware. Ultimately, the goal of building quantum algorithms is to apply phase shifts to the qubits in such of all the possible states that make it to the end, the incorrect ones will have deconstructive interference and the correct answer(s) will have constructive interference.

Amanda Turney
ENGR-517
12/9/2022

One of the initial design considerations was what the nature of a single unit of work will look like for this application that could be parallelized. This implementation considers the single state of qubit register that progresses through the algorithm as one possible path as the smallest unit of work. For an algorithm with 64 possible paths, there will be 64 parallel processes. However, it may not be known a priori what the quantum circuit will look like due to the dynamic input value of N and thus how many resulting processes will be needed. Additionally, the application will initialize with 1 process and will branch off as it progresses through the quantum circuit. Some of these paths that have many 0 states may not ultimately undergo as many operations as others if there are a lot of controlled-X gates within the circuit, which leads to non-uniform workloads. Lastly, when a single state undergoes a Hadamard operation, the resulting superposition states are a transformed form of the singular state before the operation. In other words, there is a communication step needed such that the newly spawn process has a state that is based off the state of the parent process. With all these characterizations of a single unit of parallelizable work in mind, it was decided that this implementation would better utilize the OpenMP library over the Message Passing Interface (MPI) library. Qubit register states can run as processes in threads, which can be dynamically generated and queued, and the threads can be recycled as some processes finish earlier than others. Spawning child processes from a Hadamard operation that require state information from the parent process also do not involve any explicit communication as would be required in MPI where a worker mostly likely communicate to a manager rank that it needs to initialize a new worker rank with the newly generated qubit register state. OpenMPs library has tasking pragmas that easily allow for that work generation/allocation pattern within the shared memory environment.

The Shor's algorithm application defines several quantum operations that act on a qubit register's state (000 for example for a register of 3 qubits all in state 0) and its amplitude (which the magnitude squared is the probability of it being measured). It also defines the quantum circuit for finding the period of 4^x mod 15 as an array, comprising of lambda functions that call the above-mentioned quantum gates with the specific inputs needed for each step. A function called `applyCircuit` will loop through the quantum circuit and apply the operations to the specific qubit register state of that given thread/process, given any offset to the circuit (such as for a newly generated state that comes up halfway through the circuit, it should only process the 2nd half of the circuit's operations). The application's main block will then initialize the starting qubit register with all 0 states and then will use OpenMPs pragmas to create a section of parallel code and with the single pragma so that `applyCircuit` is applied to the initialized register by only one thread. This one thread will loop through the circuit, applying the operations, until it hits a Hadamard gate. The Hadamard gate function creates a new qubit state structure and uses the `omp task` pragma for a new thread to run `applyCircuit` to this new qubit register state with the circuit offset based off the step at which it was generated within the circuit. Most importantly, the application uses OpenMP's `task_ reduction` pragma to store the accumulated final amplitude of each state within an array where each element holds the amplitude for the state that is equal to the index. For example, a final state of 101 would add its amplitude in the 5th element within the array. A quantum circuit can generate many more processes than there are unique states though, and therefore, many threads may be trying to modify the same element within this resulting array of amplitudes at the same time. For this reason, OpenMP's `task_reduction` pragma is imperative as it creates thread-private versions of this array and then handles accumulating these sums across the threads efficiently. A custom reduction pragma was also declared in the application to handle the addition operator for complex double data

types as well which is the form of the quantum state amplitudes. Lastly, the application utilizes the random device number generator in order to randomly pick the state which is measured from the final set of resulting states given their individual probabilities. That state, if not 0, can be used to calculate the period and ultimately the prime numbers of 15. Below is an output of the application:

```
turneya@j-login2:~/engr_e517/final_project/build/release$ ./shors
Performing Shor's Algorithm to factor 15 with 12 threads
The estimate of the factors of 15 are 3, 5.
Number of tries: 3, using 12 threads, time spent processing: 2741675 ns
turneya@j-login2:~/engr_e517/final_project/build/release$
```

Figure 1: Shor Algorithm application output. The number of retries of the quantum algorithm, processing time, and number of threads are outputted at the end as well as the estimates for the factors of 15.

This specific circuit results in 32 states in superposition and thus 32 threads. Using OpenMP's environment variable `OMP_NUM_THREADS`, a series of experiments can be run to test the application with varying number of threads. Below is the performance of the application in terms of time to solution in nanoseconds.

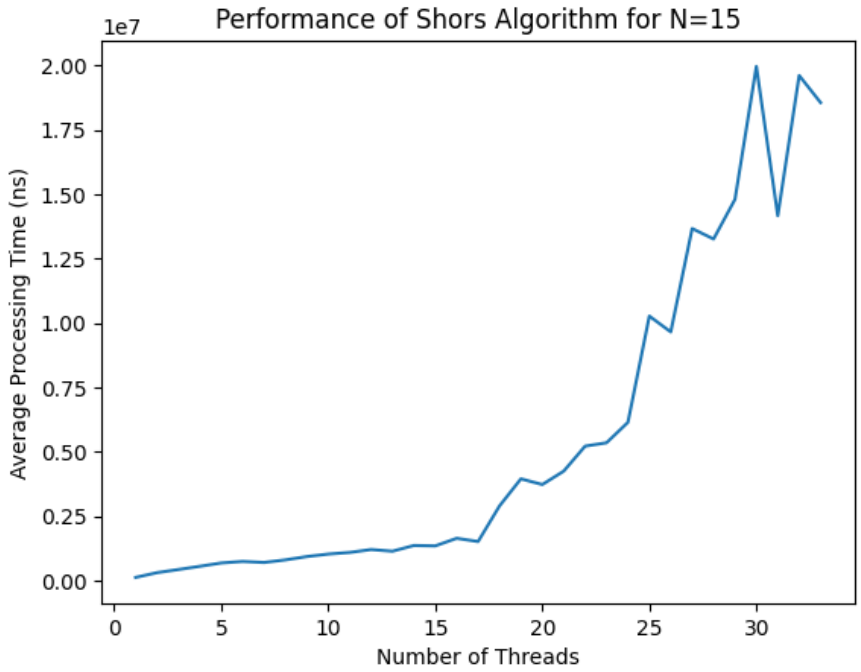

Figure 2: Performance of the Shor's Algorithm application for N=15 using OpenMP and tasking.

One aspect of the application, and of quantum algorithms in general, is that they do not result in the correct answer 100% of the time. Quantum mechanics can be utilized but not controlled so the correct answer cannot be guaranteed for each run of a circuit. For this specific implementation of 4^x mod 15, there are 2 resulting states that are ultimately observed for a measured period of 0 (50% chance) and 2 (50% chance). A period of 0 is not a valid result so in those cases, the application will run the circuit again until it gets a valid result (as seen by the number of retries printed out in the output above). The above plot shows the average processing time across 10 runs of the application under varying number of

threads. While some fluctuation of the performance is expected due to the number of retries for each thread across all the trials, the trend is clear that performance degrades slowly after adding just an additional thread all the way up until about 18 where it starts to curve upwards at a steeper increase. In this case, the workload of a single thread is almost negligibly small as it only 10 operations in total and consist of gates such as NOTs and CNOTs. Therefore, the performance is most likely reflecting the overhead in managing and creating threads over the lifecycle of the application. While the performance does not benefit from strong scaling, running with 32 threads does mirror how the quantum algorithm works in the quantum world.

The completed application does implement a small, working example of Shor's Algorithm for a specific N. However, it would be desirable to instead develop an application that could dynamically build the quantum circuit for a given a and N of the general function a^x mod N. A large circuit that can be built dynamically could also give itself to benefitting from performance gains through parallelization. To implement this, one first needs to develop a framework of a dynamic circuit and circuit building operations. Next, the application should take user input for an N and will randomly select the a. Last, the application should call the circuit builders that will dynamically add the correct operations to the quantum circuit that is required to carry out the period finding function for the specific input values. In order to build the algorithm for a generalized a^x mod N function, one can start with its most basic component: a quantum adder. One good choice for the quantum adder is Draper's Adder, which requires transforming into the Quantum Fourier Transform (QFT) basis, applying controlled phase gates, and then doing the Inverse QFT to get back to the computation basis [4]. To build an adder with a modulus N, an inverse adder is used to subtract off N. Multiplication of a mod N is achieved by repeating the adder with a modulus N until the number of multiplications is complete. Lastly, exponential with modulus (a^x mod N) can be developed by repeated controlled multiplication [1]. With this high-level framework in mind, one can design much larger circuits where there could be hundreds or thousands of these core adder units within the circuit. Most importantly, the register only needs to be transformed to the QFT basis once and all subsequent additions can occur before finally converting back to the computation basis. The reason this is important is because the QFT and IQFT have Hadamard gates whereas the pure addition components of the adder do not; they only have controlled phase rotations. Therefore, the more additions that can be incorporated into the circuit, the more the workload of each individual thread or state in superposition can do.

This advanced implementation of Shor's algorithm is not yet complete, but the framework is ready to at least test a theoretical circuit. With the QFT, IQFT, and adder complete, the application can dynamically build circuits using these units of work. Below is a plot showing the performance of a quantum algorithm comprised of 200, 2000, and 20000 quantum adders from1 to 48 threads. As one can see, a small amount of performance gain can be seen for as few as 200 quantum adders but dramatic improvement that drops by ½ for with a doubling of threads isn't seen until a circuit approaches 20,000. Once the workload for a given thread amortizes the overhead cost of managing and spawning the threads, the time to solution can start to greatly improve.

Amanda Turney
ENGR-517
12/9/2022

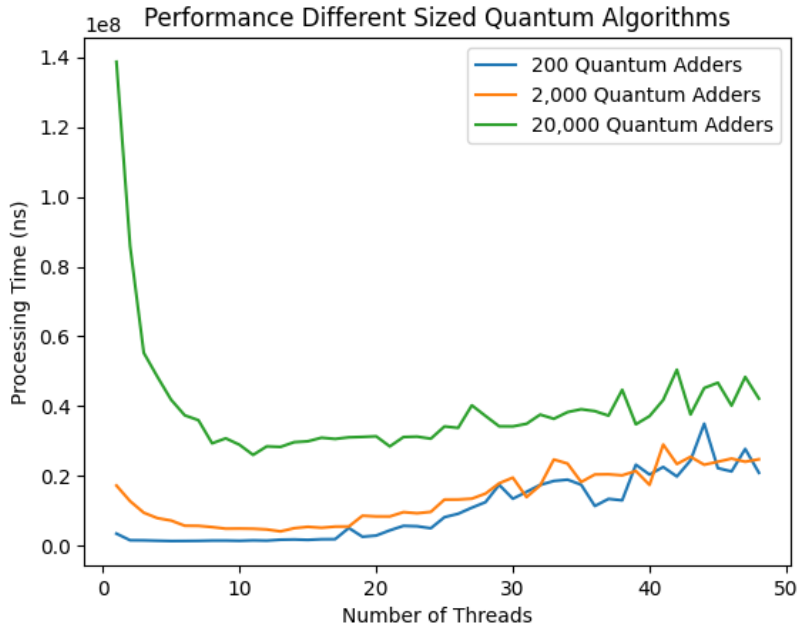

Figure 3: Performance of a general quantum algorithm of varying size for given number of threads.

Shor's algorithm is a cornerstone quantum algorithm that is a great example of how something so theoretical can solve practical problems. While the number of qubits that a real quantum system can actually manage is still very low, there's immense benefit in learning quantum algorithms either for the eventual application on quantum hardware or at the very least, to solve problems in a vastly different way than the classically inspired methods. The implementation of this quantum algorithm simulation utilizes OpenMP and a tasking parallelization strategy to simulate the superposition of states that a quantum algorithm generates. While strong scaling of the proof-of-concept implementation with N=15 did not display any strong scaling, a much larger, more generalized circuit was investigated for different number of threads. Overall, this circuit does benefit from both strong scaling and weak scaling as the larger number of threads did effectively half the processing time (up until a point). The link to the GitHub repo and directory where the code for the completed Shor's Algorithm can be found is here: https://github.iu.edu/engr-hpc-fa22/turneya/tree/main/final_project

The GitHub repo and directory where the code for the advanced, dynamic quantum algorithm builder can be found here:

https://github.iu.edu/engr-hpc-fa22/turneya/tree/main/quantum_ops

References:

1) https://tsmatz.wordpress.com/2019/05/22/quantum-computing-modulus-add-subtract-multiply-exponent/
2) https://qiskit.org/textbook/ch-algorithms/quantum-fourier-transform.html

Amanda Turney
ENGR-517
12/9/2022

3) https://qiskit.org/textbook/ch-algorithms/quantum-phase-estimation.html
4) https://learn.microsoft.com/en-us/azure/quantum/user-guide/libraries/standard/algorithms
5) https://medium.com/@sashwat.anagolum/qftaddition-ce0a0b2bc4f4
6) https://medium.com/@sashwat.anagolum/arithmetic-on-quantum-computers-multiplication-4482cdc2d83b
7) https://tsmatz.wordpress.com/2019/06/04/quantum-integer-factorization-by-shor-period-finding-algorithm/