Evaluating OpenMP and BoostCompute on Quantum Simulations

Achintya Gopal
Johns Hopkins University
agopal2@jhu.edu

Abstract

This project explored the effects of parallelization using OpenMP and BoostCompute on a quantum simulating library. OpenMP uses simple loop parallelism and should consistently do better than the serial version; however, OpenMP will do much better for algorithms with multiple quantum gate operations due to the startup costs of creating a vector and other serial operations. However, we have to consider the effects of skew on the loops. BoostCompute will do much better than OpenMP in the case of huge vectors and multiple operations since BoostCompute will have huge startup costs.

1 Code Base

The code base is a small library for simulating quantum circuits. Due to the nature of quantum systems, the mathematics behind quantum mechanics is linear algebra; hence, the representation of a quantum system is to use vectors and possibly matrices. Each value stored in the vector describing the quantum system is the probability amplitude, meaning the probability is found by taking the project of the probability amplitude by its complex conjugate. Because of this, the probability amplitude can be a complex number. However, due to testing with BoostCompute, I havent tried to incorporate complex numbers in this project.

The size of the vector grows exponentially with the number of bits in a quantum system as there is an associated probability amplitude with each configuration of the bits.

Most of the gates used in a quantum circuit are 1 bit operations, which simplifies the computational complexity as it only has to look at two values within the vector at a time. This introduces opportunities for parallelism.

The final step of most algorithms is the measurement step where the user measures a specific configuration of the quantum system and the probability of what they measure is dependent on the probability amplitude stored in the vector.

2 Tests

In this project, I compare a serial implementation, OpenMP implementation, and BoostCompute implementation of the quantum simulating library. The tests I run are: running the Hadamard gate on one bit, running the Hadamard gate on one bit and then measuring the system, running the Hadamard gate on each bit, running the Hadamard gate on each bit and then measuring the system, and Grovers algorithm. The Hadamard gate is the operator:

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

2.1 Grover's Algorithm

Grovers algorithm is a quantum search algorithm which runs in $O(\sqrt{n})$ where n is the number of configurations possible for the quantum system. This is a good test since Grovers algorithm is a stream of multiple gates being applied to the circuit as well as contains a few steps of measurements which cannot be parallelized, so affects the amount of the algorithm that can be run in parallel.

3 OpenMP

The simplest change I added using OpenMP was wrapping each loop with a pragma with default scheduling (Test 1). I tested this versus using a guided schedule (Test 2) as there is some skew. The code tries to update the vector in place, causing half the iterations to update the vector and the other half not to. The graphs show that the guided scheduling helps the partitioning of the work to the threads

which was predictable due to the clear evidence of skew.

Since there is a skew, I tested to see if it was possible to program away the skew while still updating the vector in place. I tested both default (Test 3) and guided (Test 4) scheduling. This did do better in the one Hadamard gate case since there were more outer loops than threads which helps balance the costs of using for loops within. However, this does not do as well for the multiple Hadamard gates since in one side of the edge cases, the inner loop is only 1 or 2 iterations and the other side, the outer loops is only 1 or 2 iterations.

To try to deal with this variance, I write code which deals with different cases in different ways as well as apply loop unrolling (Test 5). This decreased the readability of the code, however it is interesting to see how much the smaller algorithms speedup versus the longer algorithms. The code is very much geared to speeding up well for 4 threads; for more threads, I would add more if statements to balance the problems that show up from skew. In the one Hadamard gate case, it is faster than the Test 1 and Test 2 because of loop unrolling, however it is slower than Test 3 and Test 4 most likely due to the costs of the if statements. However in the case of Multiple Hadamard gates, the speed up from loop unrolling and dealing with the edge cases in special ways causes the code to run faster than all the other four tests.

I ran the one gate Hadamard test and multiple gate Hadamard test on all five configurations given above. I did not test the measurement step as the measurement step is serial is approximately the same amount for each configuration (approximate since there is a random number generator). I ran the Grovers algorithm test on the best performing readable code which was Test 2 and compared it to the complex code (Test 5). To make the results more comparable, I only let the code for Grovers algorithm run for one iteration (which is guaranteed to happen about 90% of the time). The complex code (Test 2 for Grovers algorithm) did approximately the same for less bits for the more intuitive code (Test 1 for Grovers algorithm) but did better as more bits were used.

Looking at the results of the complex code in comparison to the serial implementation, the speedup for one Hadamard gate using 26 bits was 1.847; the speedup for one Hadamard gate with measuring at the end is 1.811; the speedup for a Hadamard gate per bit was 2.617; the speedup for a Hadamard gate per bit and then measuring at the end is 2.595; the speedup for Grovers algorithm for 16 bits is 2.551. The measuring test always had less speed up which occurs since the measuring step is always serial. The multiple Hadamard gate example had more of a speedup than one Hadamard gate since the startup costs of creating a vector of size 2^n is more dominant in the one Hadamard gate case. Grovers algorithm had enough gates that the startup costs of creating the vector is hidden and so the speedup is similar to that of the test with multiple Hadamard gates and a measurement since Grovers algorithm had multiple gates and two measurements. Applying Amdahls law, the parallel portions for the five tests given 4 threads are 61.1%, 59.7%, 82.4%, 82.0%, and 81.6% respectively.

4 BoostCompute

For the BoostCompute code, I wrote all the gate code in OpenCL and then used BoostComputes stringify function to be able to call it from BoostCompute. I compared the results of BoostCompute versus the OpenMP implementation with multiple if statements on all five tests. I ran the tests with measurements since in BoostCompute, the measurement step must first copy the vector from the GPU to the CPU and then perform the algorithm.

Looking at the graph for the time for the test with One Hadamard Gate, the time complexity of the serial implementation is $O(2^n)$ where n is the number of bits. It can be seen that BoostCompute does worse than the serial implementation for less than about 20 bits. This happens due to the large startup costs of BoostCompute. Since the time does not increase until 24 bits, it can be inferred that the majority of the time for these cases are the startup costs. However, once it starts to reach 26 bits, which is close to amount of memory my GPU has, the code runs much slower than 24 bits, but much faster than the serial implementation.

For the graph with One Hadamard Gate and then Measured, the BoostCompute implementation runs

slower than the serial version up to 20 bits and barely faster than the OpenMP implementation at 26 bits. This happens because there is a large cost in copying the vector from the GPU's memory to the CPU's memory which adds to the startup costs.

For the case when the Hadamard Gate is run on every bit, the time complexity of the serial implementation is $O(n2^n)$. The time to run the operations outweigh the startup costs, so the Boost Compute code runs better than the serial and OpenMP implementation at about 14 bits.

Same as the case of measuring with one hadamard gate, the BoostCompute's time slows down more than the serial and OpenMP implementation. Due to this, BoostCompute does not do better until 20 bits as used.

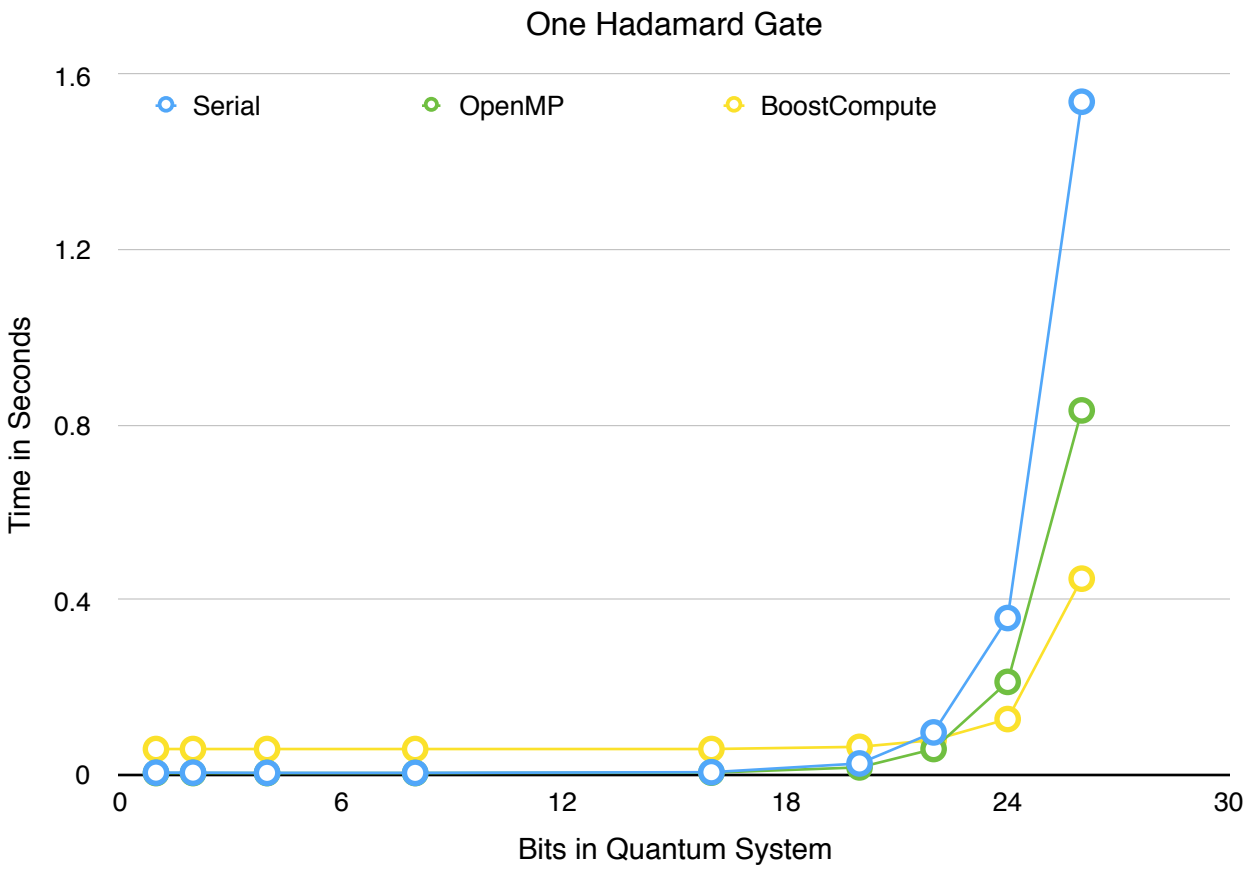
For Grover's algorithm, the time complexity of the serial implementation is about $O(3^n)$. BoostCompute does better at 14 bits. Whereas in the case of the serial implementation and the OpenMP case the code would slow down significantly enough to be "useless" after 16 bits, using BoostCompute gives the opportunity to do more computations with more bits.

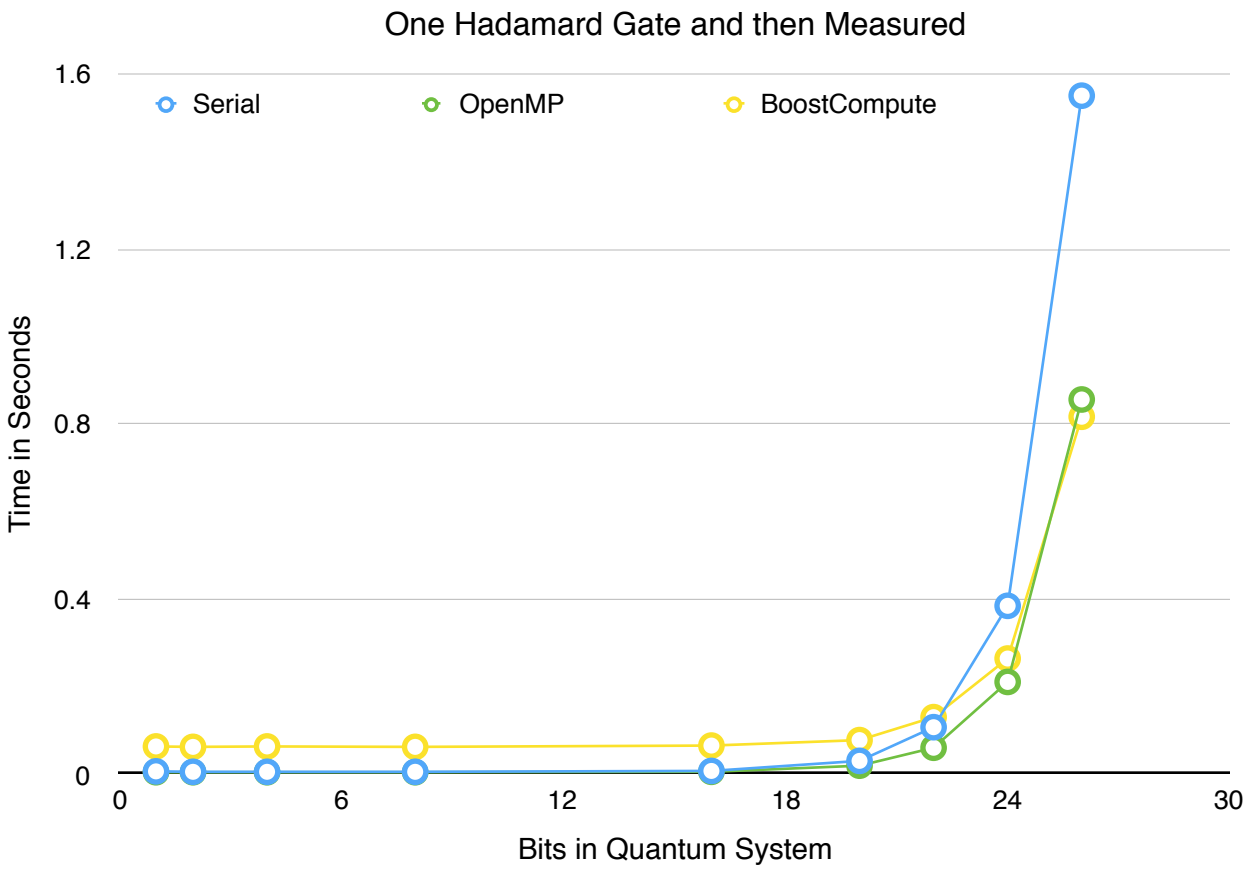
However, one limitation of BoostCompute is the GPU on my computer can only hold about 2^{27} floats. One solution to overcome this is to do some computation in the GPU, transfer it back to the CPU, and do some computation on another part of the vector. However, looking at the One Hadamard Gate that is measured example, it can be seen that the time in transferring would undo any positive effects of using the GPU.

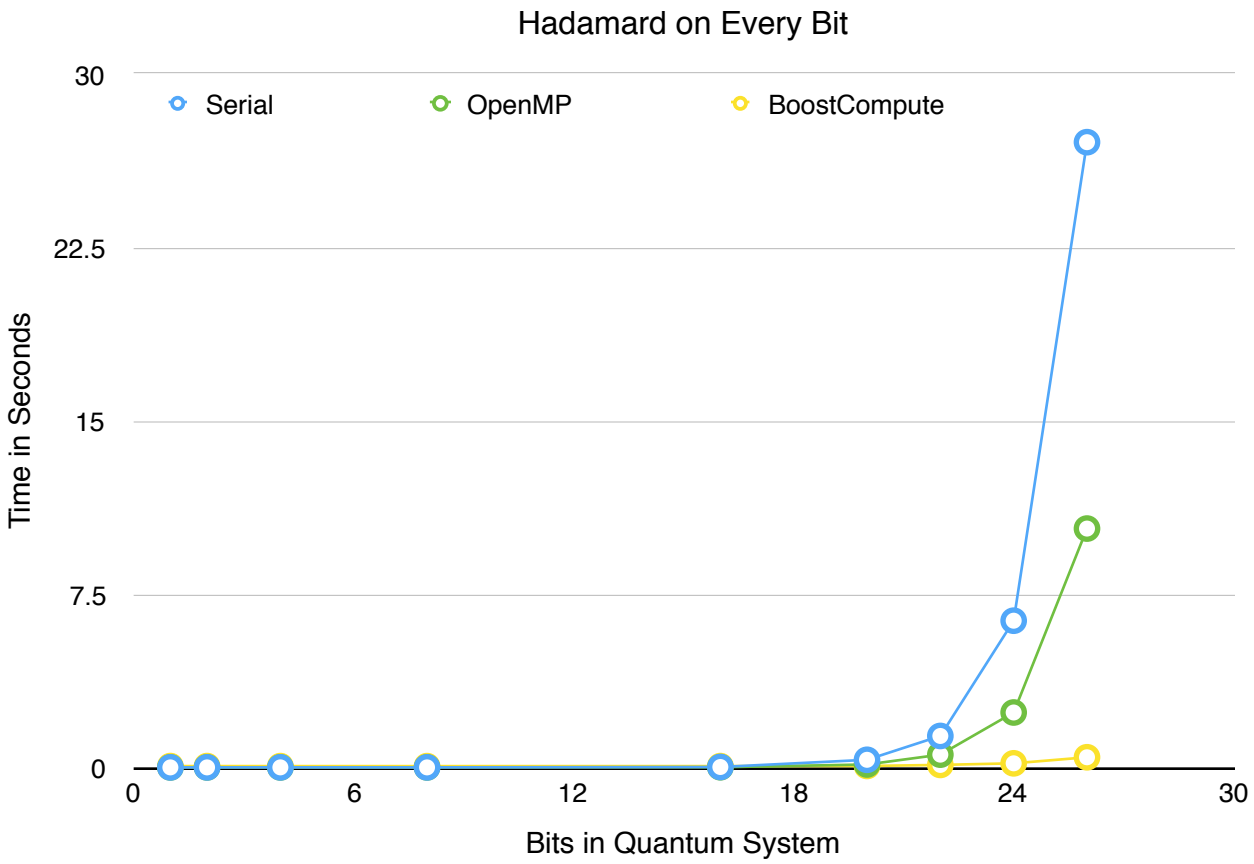
Calculating speedup is difficult in the case of BoostCompute as the startup costs is dependent to the size of the vector and the serial version is implemented quite differently than the BoostCompute version due to the step of transfering data.

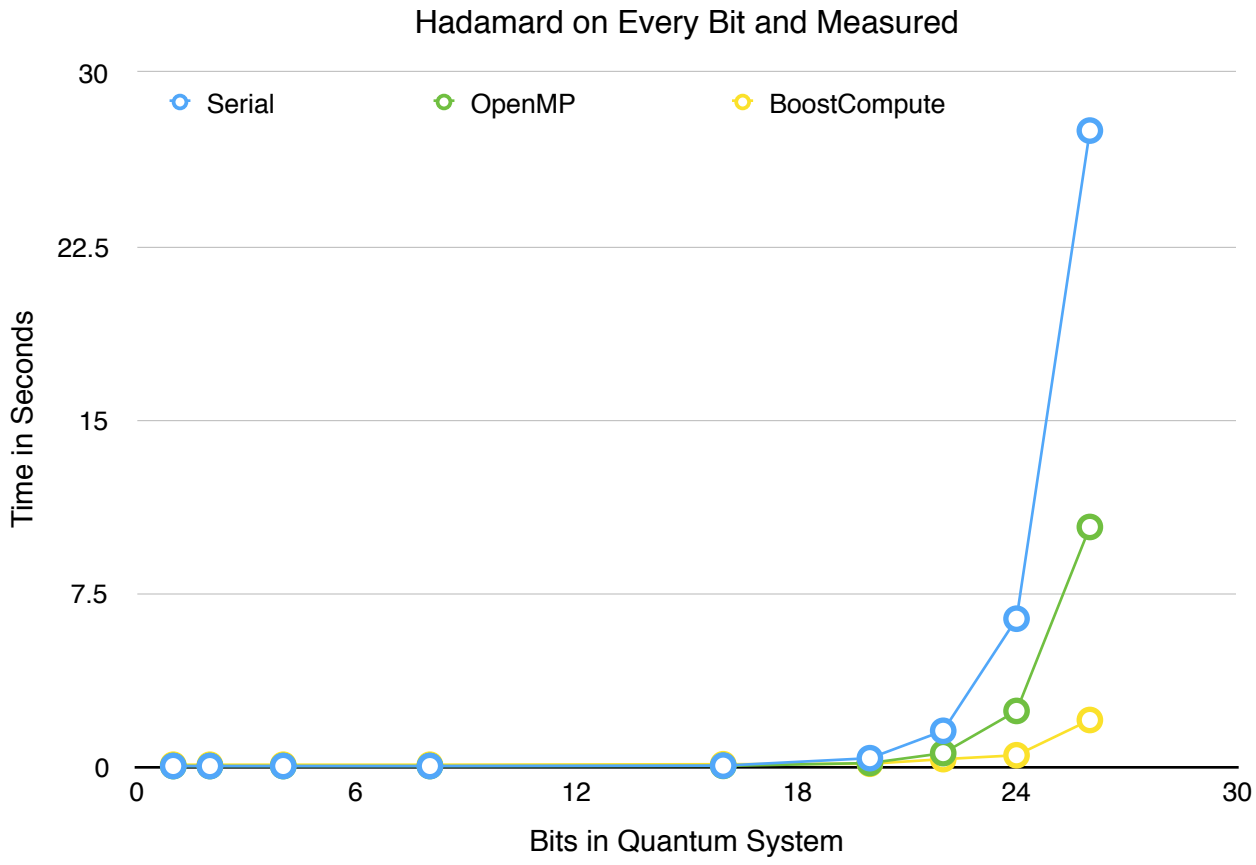
5 Conclusion

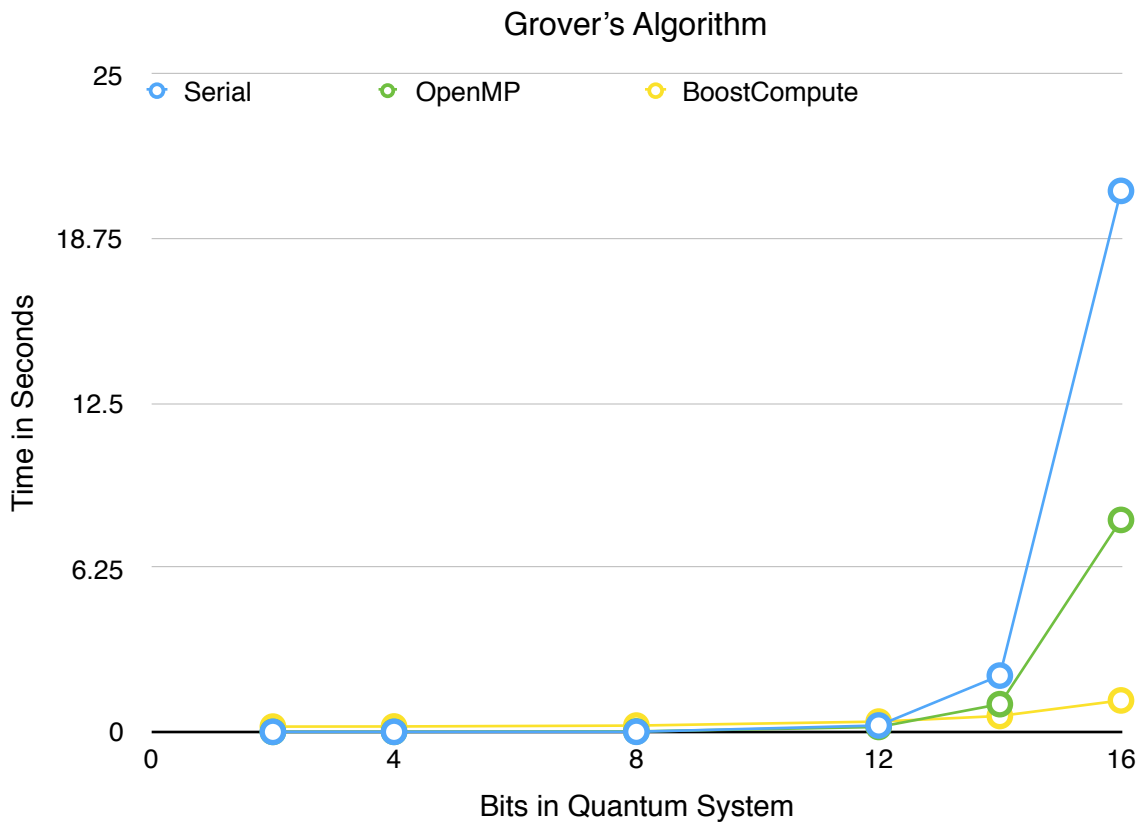
In conclusion, OpenMP does do better than BoostCompute in the case of smaller number of values in the vector since BoostCompute has massive startup costs. However, due to the number of threads in the GPU, BoostCompute outperforms OpenMP once the time to perform operations outweigh the startup costs. This is aligns closely to what I thought would come from this project.

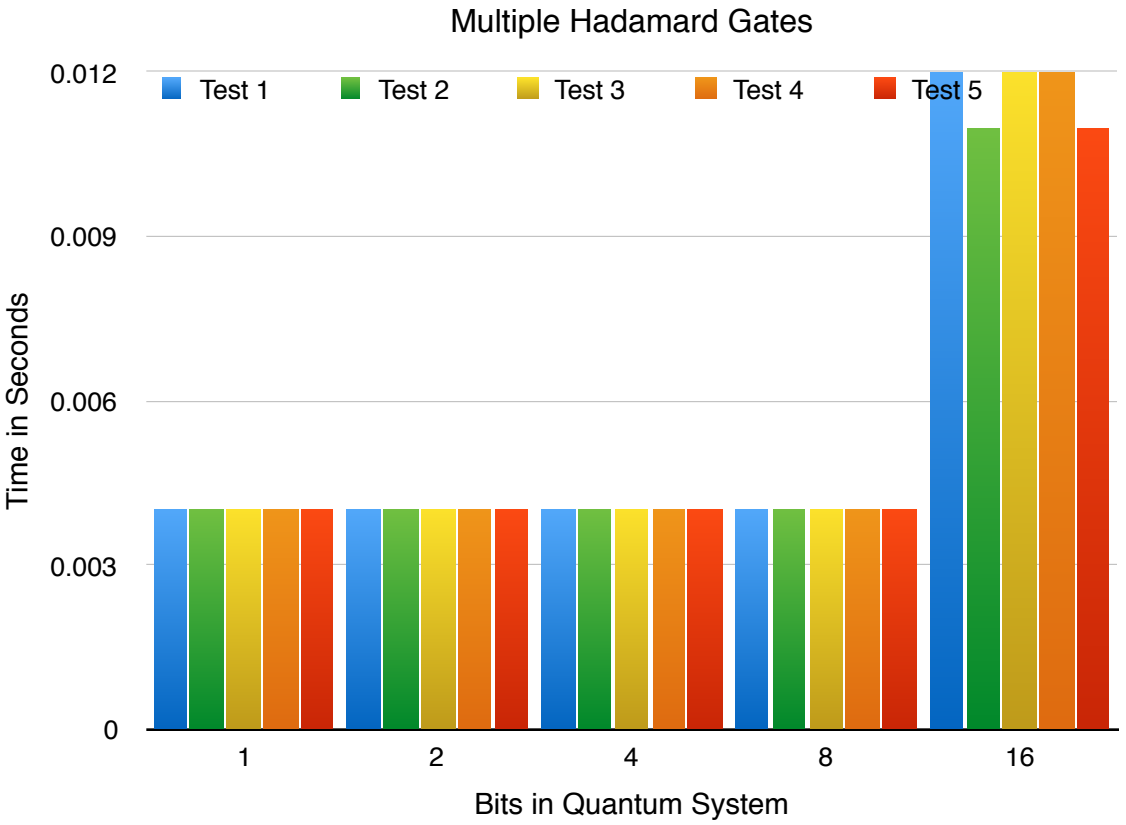

One Hadamard Gate
Serial
OpenMP
BoostCompute
1.6
1.2
0.8
0.4
0
Time in Seconds
0
6
12
18
24
30
Bits in Quantum System

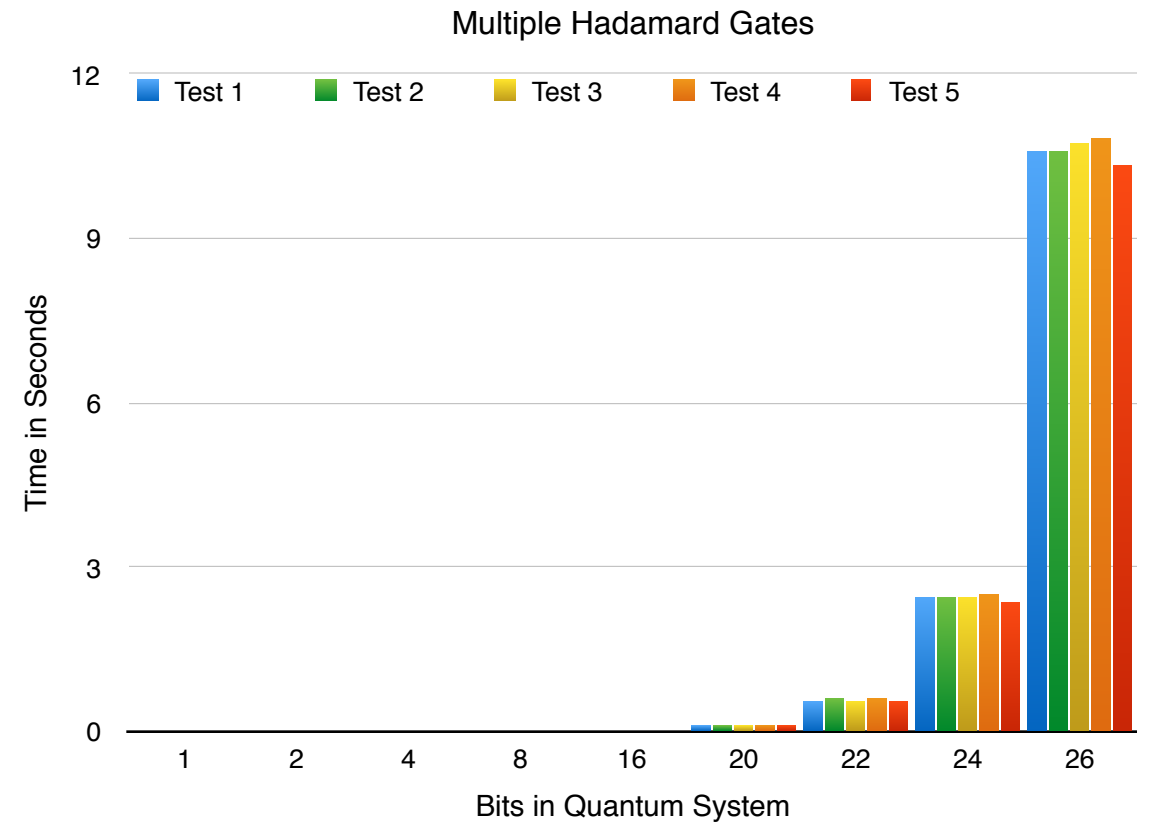

One Hadamard Gate and then Measured
Serial
OpenMP
BoostCompute
1.6
1.2
0.8
0.4
0
Time in Seconds
0
6
12
18
24
30
Bits in Quantum System

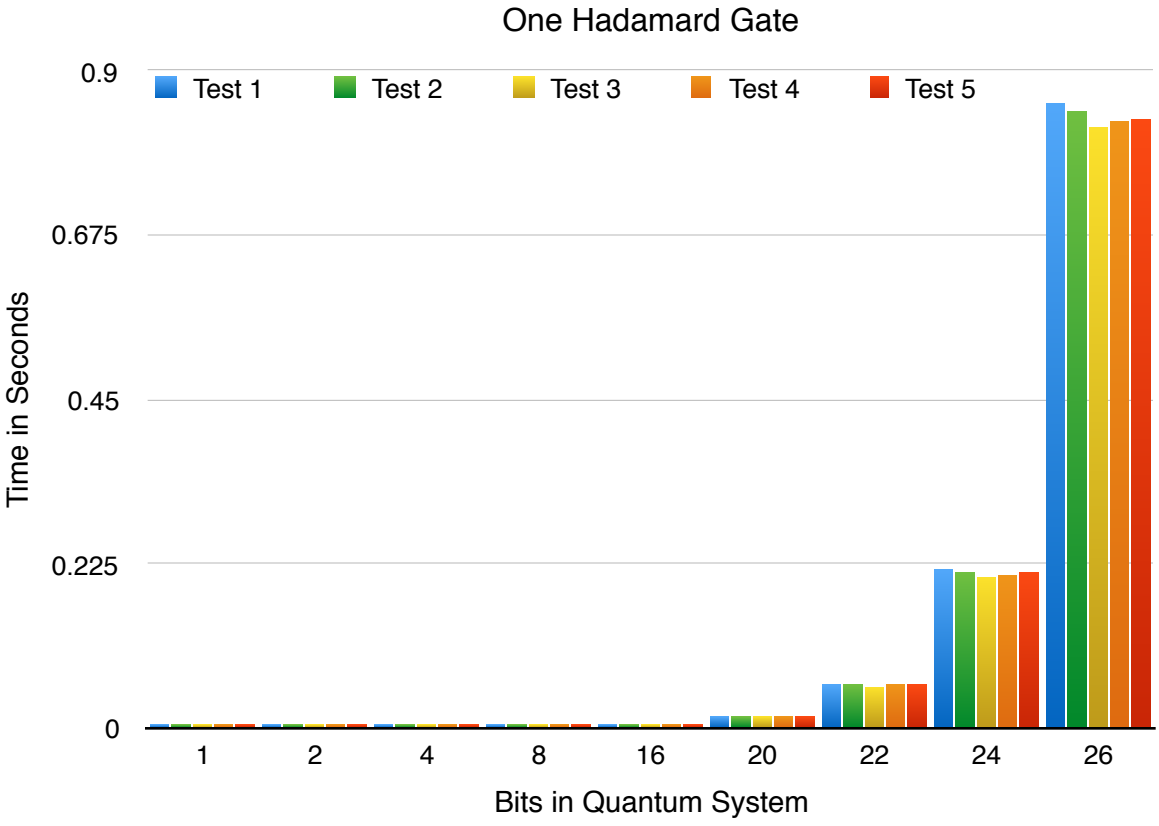

Hadamard on Every Bit
Serial
OpenMP
BoostCompute
30
22.5
15
7.5
0
Time in Seconds
0
6
12
18
24
30
Bits in Quantum System

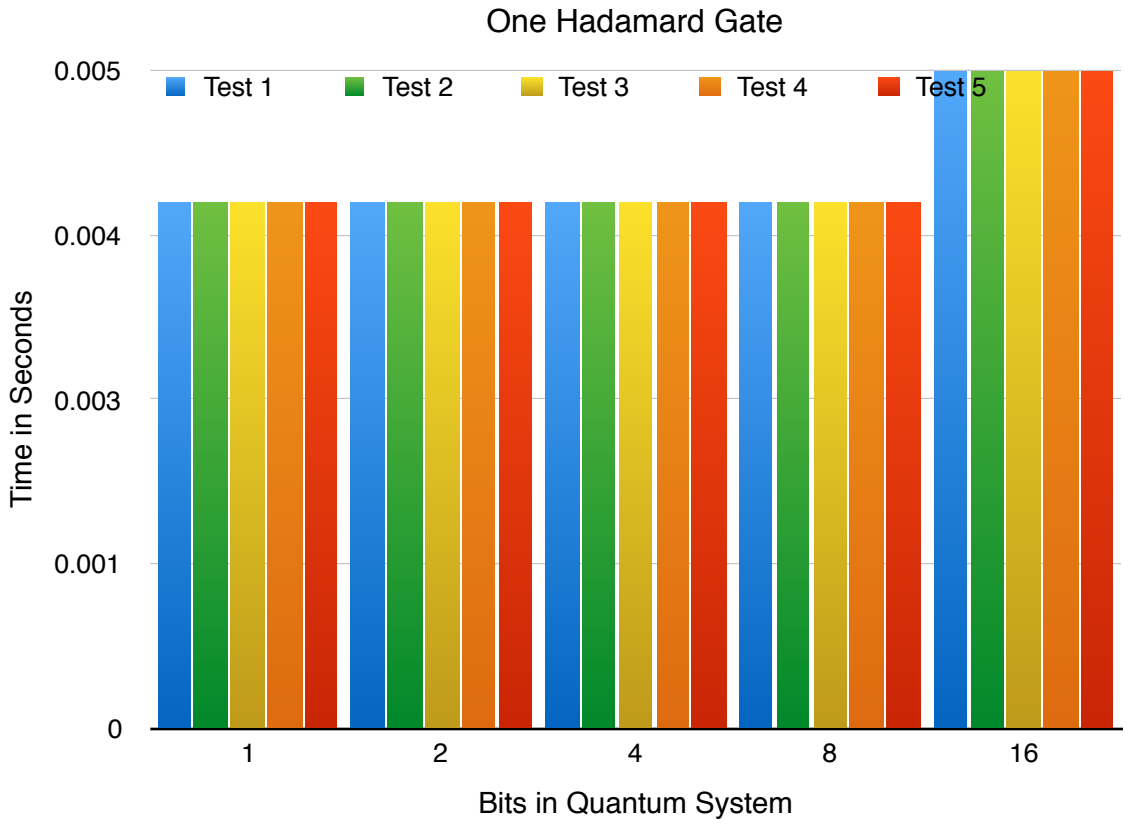

Hadamard on Every Bit and Measured
Serial
OpenMP
BoostCompute
30
22.5
15
7.5
0
Time in Seconds
0
6
12
18
24
30
Bits in Quantum System

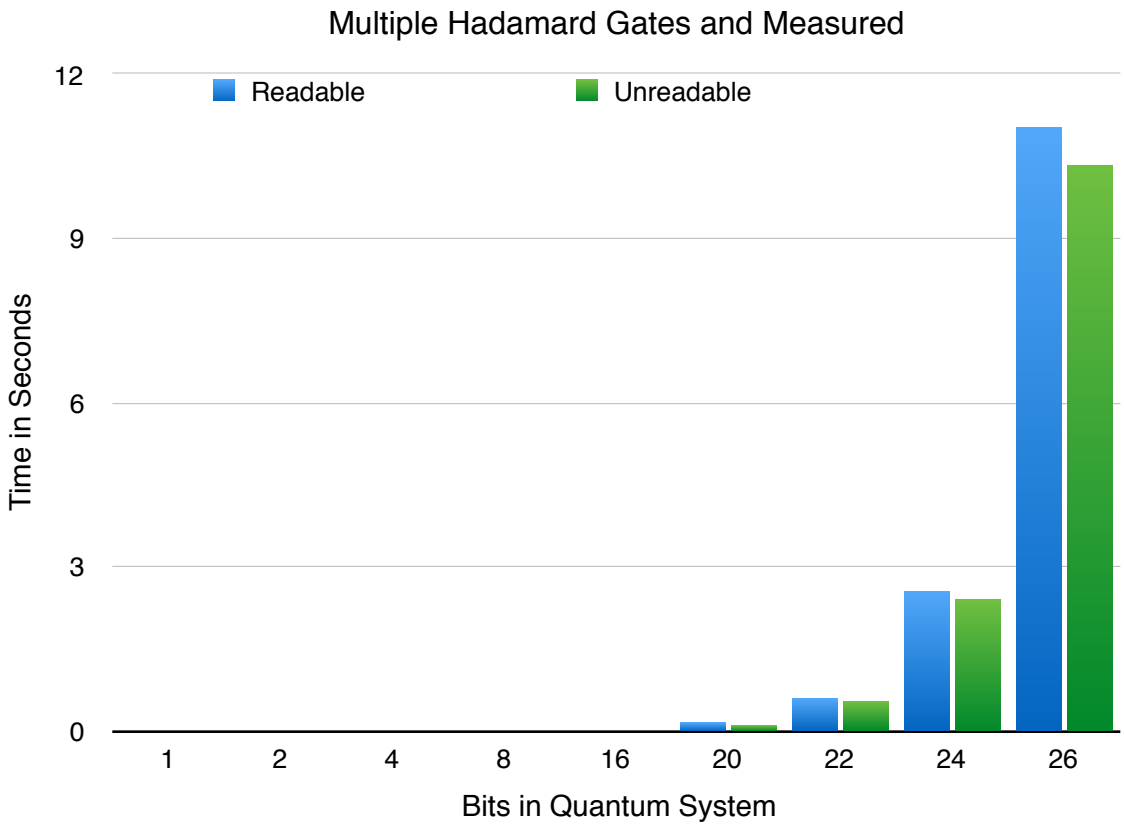

Grover's Algorithm
Serial
OpenMP
BoostCompute
25
18.75
12.5
6.25
0
Time in Seconds
0
4
8
12
16
Bits in Quantum System

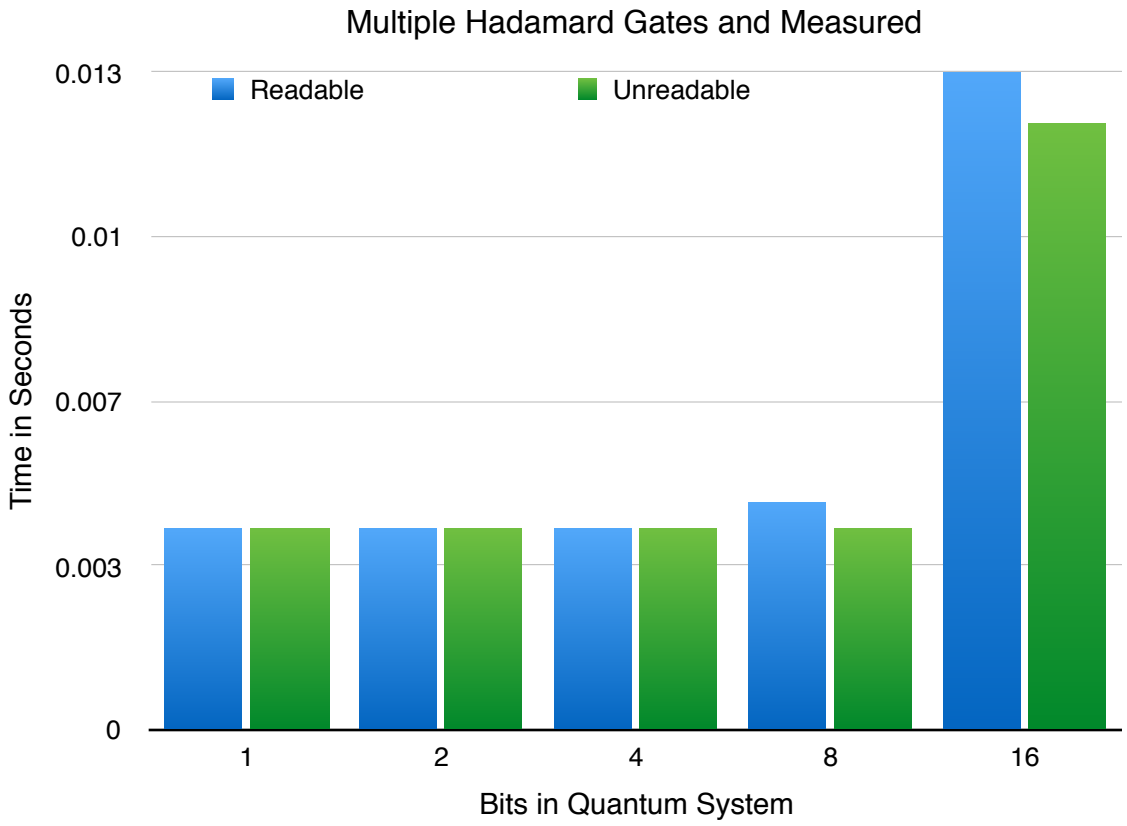

Multiple Hadamard Gates
Test 1
Test 2
Test 3
Test 4
Test 5
0.012
0.009
0.006
0.003
0
Time in Seconds
1
2
4
8
16
Bits in Quantum System

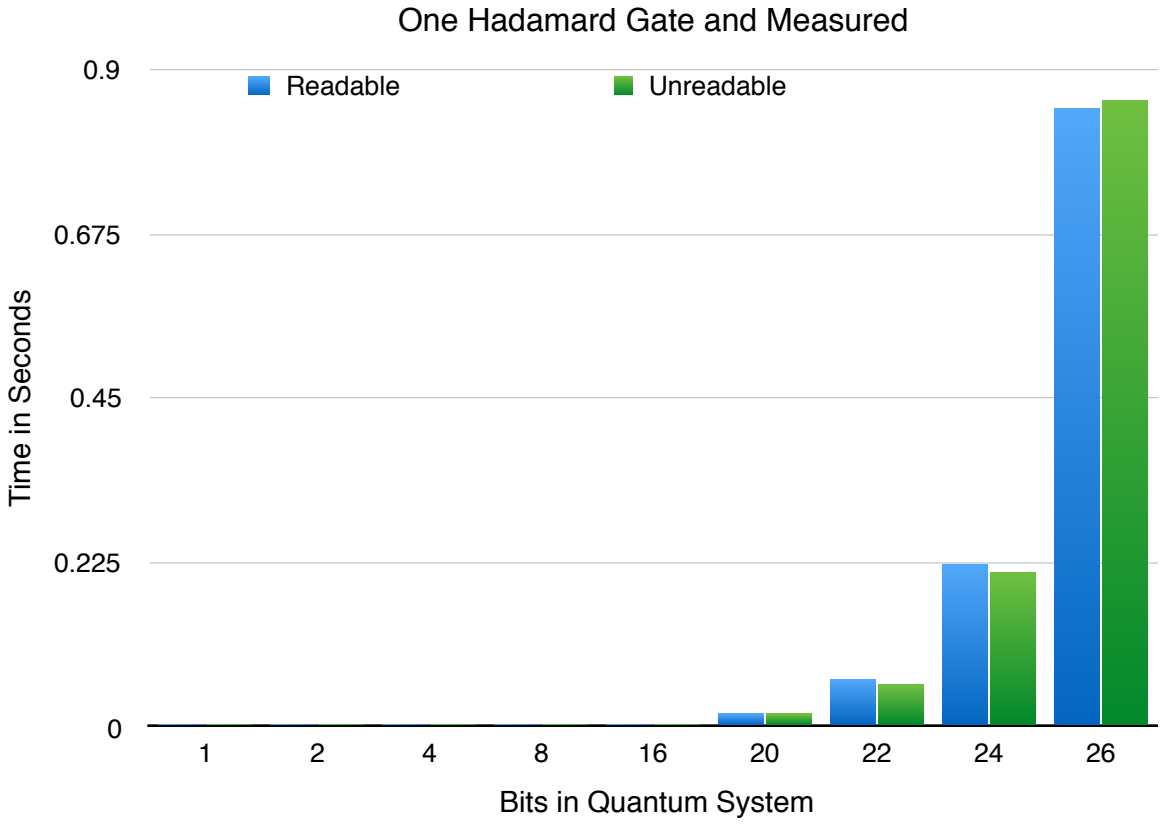

Multiple Hadamard Gates
Test 1
Test 2
Test 3
Test 4
Test 5
12
9
6
3
0
Time in Seconds
1
2
4
8
16
20
22
24
26
Bits in Quantum System

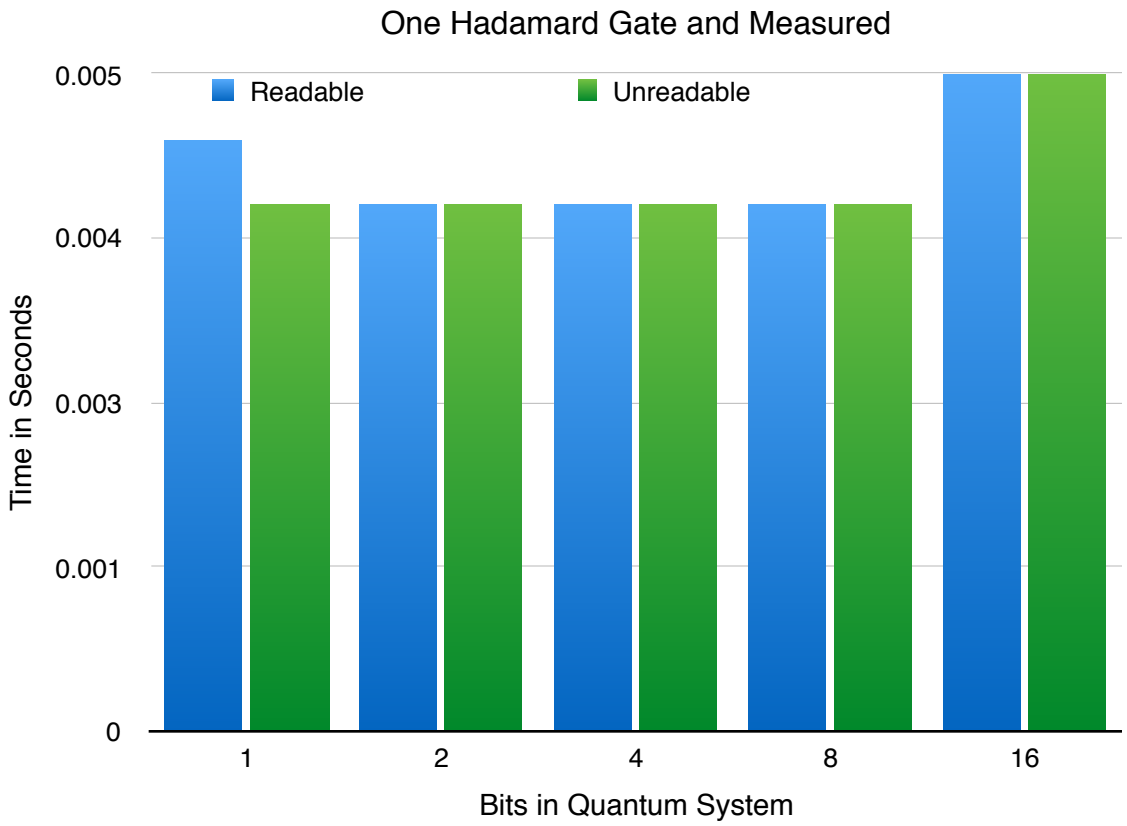

One Hadamard Gate
Test 1
Test 2
Test 3
Test 4
Test 5
0.9
0.675
0.45
0.225
0
Time in Seconds
1
2
4
8
16
20
22
24
26
Bits in Quantum System

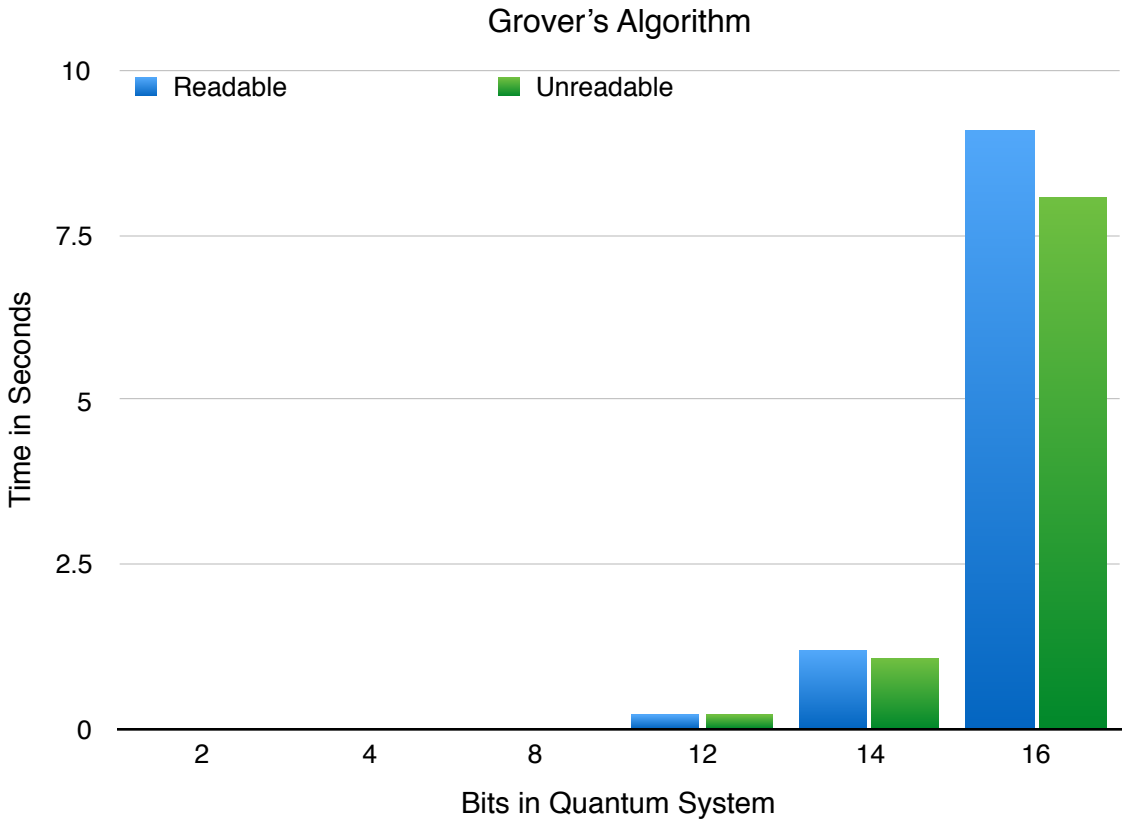

One Hadamard Gate
Test 1
Test 2
Test 3
Test 4
Test 5
0.005
0.004
0.003
0.001
0
Time in Seconds
1
2
4
8
16
Bits in Quantum System

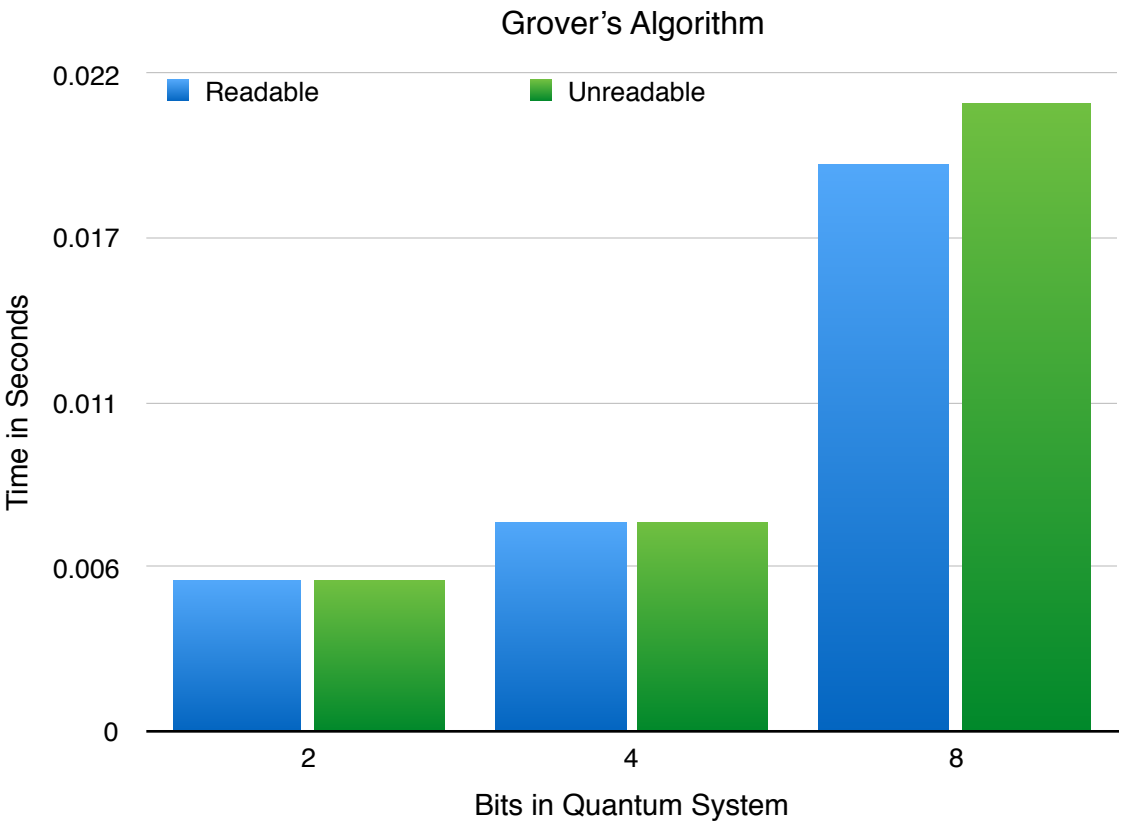

Multiple Hadamard Gates and Measured
Readable
Unreadable
12
9
6
3
0
Time in Seconds
1
2
4
8
16
20
22
24
26
Bits in Quantum System


Multiple Hadamard Gates and Measured
Readable
Unreadable
0.013
0.01
0.007
0.003
0
Time in Seconds
1
2
4
8
16
Bits in Quantum System


One Hadamard Gate and Measured
Readable
Unreadable
0.9
0.675
0.45
0.225
0
Time in Seconds
1
2
4
8
16
20
22
24
26
Bits in Quantum System


One Hadamard Gate and Measured
Readable
Unreadable
0.005
0.004
0.003
0.001
0
Time in Seconds
1
2
4
8
16
Bits in Quantum System


Grover's Algorithm
Readable
Unreadable
10
7.5
5
2.5
0
Time in Seconds
2
4
8
12
14
16
Bits in Quantum System


Grover's Algorithm
Readable
Unreadable
0.022
0.017
0.011
0.006
0
Time in Seconds
2
4
8
Bits in Quantum System