IMPLEMENT A QUANTUM CIRCUIT SIMULATOR

QUANTUM PROGRAMMING

Madhav Sankar Krishnakumar madhavsankar@ucla.edu

March 27, 2022

Contents

ABSTRACT

On a classical computer, a quantum circuit simulator is implemented in Python for a subset of QASM programs. We then discuss our code organization and testing results for the same.

1 Highlight

- **Comprehensive evaluation/testing:** Apart from the benchmarks provided, tested the code on QASM programs of other algorithms like Deutsch–Jozsa, Bernstein Vazirani and Grover's.
- **Vivid Graphical plots:** Apart from the testing, the effect of number of qubits, number of lines of code and the quantum volume on the simulation time is well studied and graphs along with observations are provided in this report. Range of simulation times and time taken per gate operations are also plotted.
- **Code well designed for improved usability and ease of understanding:** The code has been well divided into modules and their functionality and dependence are clearly mentioned through the low level design flowchart. Apart from this, each of the module is well divided into various methods which are well documented.
- **Understanding how the Simulation time/difficulty of simulation of a problem can be assessed as a function of the number of qubits and number of lines of code:** After plotting the graphs with regard to simulation time as a function of number of qubits and number of lines of code, it is evident that neither of the graphs are monotonic. This implies that neither of the factors alone could directly explain the simulation time/difficulty of a problem. Therefore, a new parameter p, which is defined as a function of the two factors (number of qubits and lines of code) is defined to explain the difficulty or easiness to simulate a QASM program.

2 Design and Evaluation

2.1 Low Level Design

The code has been divided into different components with their own independent responsibilities. This can be depicted as follow:

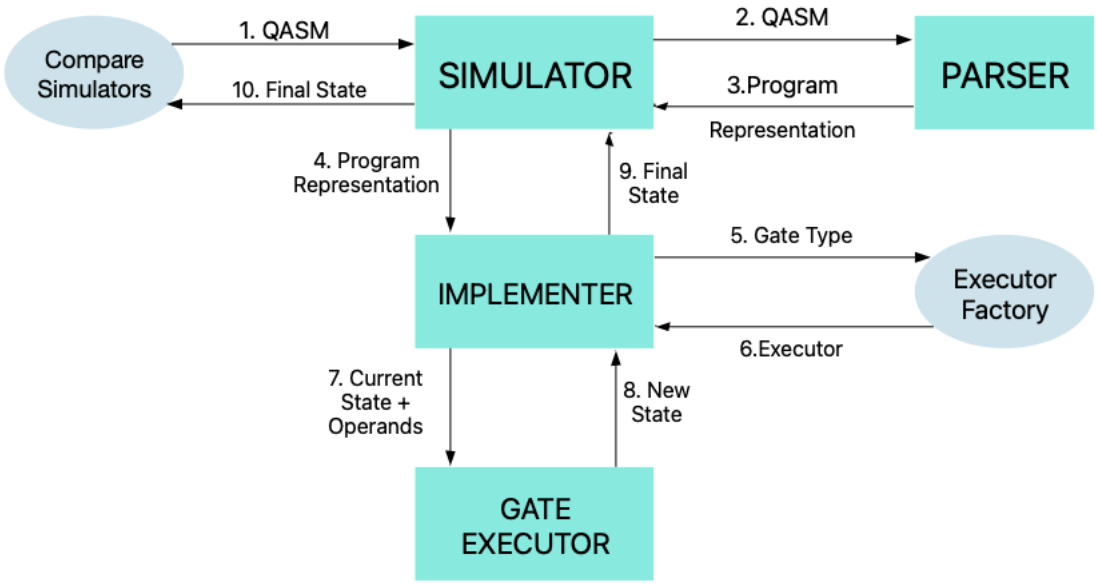

Figure 1: Low Level Design

Let us try to understand what each of the components do:

- `Compare Simulators`: This code was given to us. They help compare our simulators to that of cirq simulators. They read the various QASM files and pass them to the Simulators.
- `Parser`: The parser, as the name suggests handles the parsing of the QASM file and generates a program representation. It basically converts the given QASM file into a list of statements where each statement has an associated gate and list of operands.
- `Executor Factory`: The executor factory implements the Factory design principle. So basically each gate has its own execution logic and hence separate executor. So the execution factory simply returns the corresponding executor for the given gate type.
- `Gate Executor`: This module consists of the core logic of each quantum gate. So given the starting state and the list of operands, the gate applies the logic over the operands and converts the starting state into a new state. This new state is then returned.
- `Implementer`: The implementer executes the complete program. It initializes a starting state with all qubits being in state 0 and then applies each statement of the program representation. For a given statement, it calls the execution factory to find the corresponding gate's executor. Then it calls the executor to apply the gate over the current state. The implementer returns back the final state.
- `Simulator`: The simulator is the base code which handles the entire logic. It calls the parser to obtain the program representation and calls the implementer to run the program and get the final state. It returns this final state to the caller.

2.2 Code Readability

The whole code was divided into multiple modules as explained before. Each of these modules are further divided into functions. The following are the main functions and their purpose within each module:

2.2.1 Simulator

- `simulate(qasmString)`: This function parses the QASM string into a Program Representation and then uses the implementer to simulate the code and retrieve the final state.

2.2.2 Parser

- `_preProcess(qasmString)`: This function removes any trailing 0s and the first 4 lines.
- `_findNumberOfQubits(qasmString)`: This function find the number of qubits needed to run the code. We scan through the entire program and decide this.
- `_findOperand(gate)`: This function finds the gate enum given a string.
- `_createSentence(sent)`: This function converts the QASM sentence into a statement object. A statement object consists of a gate and list of operands.
- `parse(self)`: This function parses the QASM string into a Program Representation object, which consists of the number of qubits and the list of statements.

2.2.3 Implementer

- `implement(self)`: This method creates an initial state where all the qubits are 0 and then applies each of the program statement one by one. It calls the Executor Factory to find the corresponding gate's executor and then uses the executor to apply the gate on the current state.

2.2.4 Executor Factory

- `getExecutor(self, gate)`: Executor Factory maps the gates to the corresponding execution logic / executor.

```
if gate == Gates.X:
    return Xgate()
elif gate == Gates.H:
    return Hgate()
elif gate == Gates.CX:
    return CXgate()
elif gate == Gates.T:
    return Tgate()
elif gate == Gates.TDagger:
    return TDaggergate()
else:
    return ParentGate()
```

2.2.5 Gate Executor

Parent Gate

- `decimalToBinary(self, x, n)`: This function converts the integer, x into a binary string of the given length, n.
- `binaryToDecimal(self, x)`: This function converts a binary string x to integer.
- `setString(self, s, pos, value)`: This function sets s[pos] = value and returns the new string.
- `execute(self, state, operands)`: This function has the core execution logic of the Gates. All the children classes override this method.

HGate

- `execute(self, state, operands)`: This function has the core execution logic of the Hadamard Gate.

XGate

- `execute(self, state, operands)`: This function has the core execution logic of the X Gate.

CXGate

- `execute(self, state, operands)`: This function has the core execution logic of the CX Gate.

TGate

- `execute(self, state, operands)`: This function has the core execution logic of the T Gate.

TDaggerGate

- `execute(self, state, operands)`: This function has the core execution logic of the T Dagger Gate.

2.2.6 Test

- `preProcess(qasmString)`: This function removes any trailing 0s and the first 4 lines and returns the processed string along with the file length.
- `findNumberOfQubits(qasmString)`: This function find the number of qubits needed to run the code. We scan through the entire program and decide this.
- `cirq_simulate(qasm_string: str)` : This function runs the Cirq Simulator.

2.3 Parameterizing the solution in n

The Parser module reads through the QASM file and finds what the maximum number of qubits, n is and this information is added to the program representation. Therefore, the implementer knows n and can create a 2^n length array of complex numbers to store the state. Then we iterate through each gate and apply the operation one by one. Therefore, the code is extended to take into account any number of qubits and any number of oeprations.

2.4 Testing

The testing was done rigorously on the correctness of the simulation for various values of n. Initially, tests were done on some simple quantum circuits generated from the simple algorithms earlier. These QASM circuits are added to the folder QASM_2. The algorithms tested included Bernstein-Vazirani, Deutsch–Jozsa and the Grover's Algorithm.

```
(base) madhavsankar@Madhavs-MacBook-Air Simulator % python CompareSimulators.py 'QASM_2'

QASM_2/DJ_n2_Balanced2.qasm
[0j, 0j, (-0.707+0j), (0.707+0j), 0j, 0j, 0j, 0j]
True
QASM_2/BV.qasm
[0j, 0j, 0j, 0j, 0j, 0j, 0j, 0j, 0j, 0j, 0j, 0j, (0.707+0j), (-0.707+0j), 0j, 0j]
True
QASM_2/Grover.qasm
[0j, 0j, (-1+0j), 0j]
True
QASM_2/DJ_n2_Constant1.qasm
[(-0.707+0j), (0.707+0j), 0j, 0j, 0j, 0j, 0j, 0j]
True
QASM_2/DJ_n2_Constant0.qasm
[(0.707+0j), (-0.707+0j), 0j, 0j, 0j, 0j, 0j, 0j]
True
QASM_2/DJ_n2_Balanced1.qasm
[0j, 0j, 0j, 0j, (0.707+0j), (-0.707+0j), 0j, 0j]
True
```

Figure 2: Traditional Algorithms Test Results

Deutsch–Jozsa (n = 2 AND Constant 0):

```
Simulator Output: [(0.707+0j), (-0.707+0j), 0j, 0j, 0j, 0j, 0j, 0j]
```

Therefore there is an equal probability of observing 000 and 001. Considering that the last qubit is helper qubit, the result is always 00. As we know from Deutsche Jozsa algorithm, when the output is all 0, the function is constant.

Deutsch–Jozsa (n = 2 AND Constant 1):

```
Simulator Output: [(-0.707+0j), (0.707+0j), 0j, 0j, 0j, 0j, 0j, 0j]
```

Therefore there is an equal probability of observing 000 and 001. Considering that the last qubit is helper qubit, the result is always 00. As we know from Deutsche Jozsa algorithm, when the output is all 0, the function is constant.

Deutsch–Jozsa (n = 2 AND Balanced):

```
Balanced Type 1:
Simulator Output: [0j, 0j, 0j, 0j, (0.707+0j), (-0.707+0j), 0j, 0j]

Balanced Type 2:
Simulator Output: [0j, 0j, (-0.707+0j), (0.707+0j), 0j, 0j, 0j, 0j]
```

For type 1, there is an equal probability of observing 100 and 101. So the output is 10. For type 2, there is an equal probability of observing 010 and 011. So the output is 01. As we know from Deutsch–Jozsa algorithm, when the output is not all 0, the function is balanced.

Bernstein Vazirani (n = 3 AND a = 110):

```
Simulator Output: [0j, 0j, 0j, 0j, 0j, 0j, 0j, 0j, 0j, 0j, 0j,
                   0j, (0.707+0j), (-0.707+0j), 0j, 0j]
```

There is an equal probability of observing 1100 and 1101. As the last qubit is a helper qubit, the output is 110 as expected.

Grover's Algorithm (n = 2 AND x = 10):

```
Simulator Output: [0j, 0j, (-1+0j), 0j]
```

We can see that we will observe 10 with 100% probability.

After observing that the simulator works well for the traditional algorithms, the give Compare Simulator was used to test all the 14 benchmark programs.

```
(base) madhavsankar@Madhavs-MacBook-Air Simulator % python CompareSimulators.py 'QASM'

QASM/con1_216.qasm
True
QASM/mini_alu_305.qasm
True
QASM/sym6_316.qasm
True
QASM/miller_11.qasm
True
QASM/one-two-three-v3_101.qasm
True
QASM/hwb5_53.qasm
True
QASM/cm152a_212.qasm
True
QASM/squar5_261.qasm
True
QASM/rd84_142.qasm
True
QASM/f2_232.qasm
True
QASM/alu-bdd_288.qasm
True
QASM/decod24-v2_43.qasm
True
QASM/cnt3-5_179.qasm
True
QASM/wim_266.qasm
True
```

Figure 3: Test Results

As can be observed, all 14 benchmark programs succeeded.

2.5 Scalability

We have listed the number of qubits, number of lines of code and the simulation time of the 14 benchmark programs.

Program	Qubits	Lines	Simulation Time
miller_11.qasm	3	54	0.0005519390106201172
decod24-v2_43.qasm	4	56	0.0006067752838134766
one-two-three-v3_101.qasm	5	74	0.0008099079132080078
hwb5_53.qasm	6	1,340	0.06719827651977539
alu-bdd_288.qasm	7	88	0.0015270709991455078
f2_232.qasm	8	1,210	0.04066896438598633
con1_216.qasm	9	958	0.06692695617675781
mini_alu_305.qasm	10	177	0.015784263610839844
wim_266.qasm	11	990	0.15200591087341309
cm152a_212.qasm	12	1,225	0.36562585830688477
squar5_261.qasm	13	1,997	4.5015480518341064
sym6_316.qasm	14	274	0.33472585678100586
rd84_142.qasm	15	347	0.8084738254547119
cnt3-5_179.qasm	16	179	0.837928056716919

2.5.1 Benchmarks: Variation of Simulation Time with n:

The following is the plot of the amount of time taken to simulate the various benchmarks. Each qasm program was run 10 times and the average is taken. The plot highlights the difference in the Simulation time as a function of the number of qubits, n.

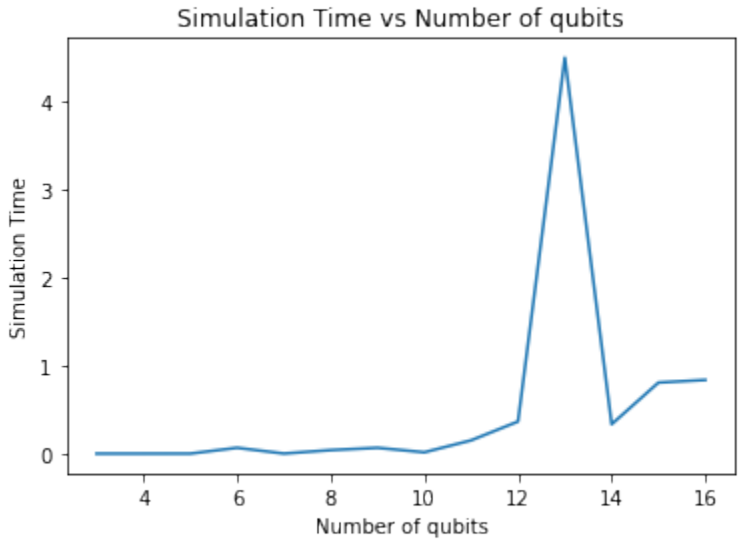

Figure 4: Simulation Time vs Number of qubits

- We observe that overall the simulation time increases with increasing number of qubits. This is expected as we are maintaining a state of length 2^n and we apply the various operations on them. Therefore it makes sense that the simulation time will increase with increasing number of qubits.
- The only exception to this is the case where number of qubits is 13 (squar5_261.qasm). This has a very large simulation time as the number of lines of code and hence the number of operations is much larger than any other program.
- To clearly understand the impact of number qubits, compare cm152a_212.qasm (will be mentioned here as A) and cnt3-5_179.qasm (B). The former has much larger number of operations, around 1221 and the latter has just 175 operations. But we can see that B takes more than double the time as A for simulation. This is mainly caused

due to the difference in the number of qubits (12 vs 16). These 4 qubit difference makes the state size differ by a factor of $2^{16}/2^{12}$, a staggering 16 times.

2.5.2 Variation of Simulation Time with n for a given number of operations:

The following is the plot of the amount of time taken to simulate a custom hand written code of 14 gate operations for various number of qubits. Each qasm program was run 10 times and the average is taken. The plot highlights the difference in the Simulation time as a function of the number of qubits, n given a constant number of operations.

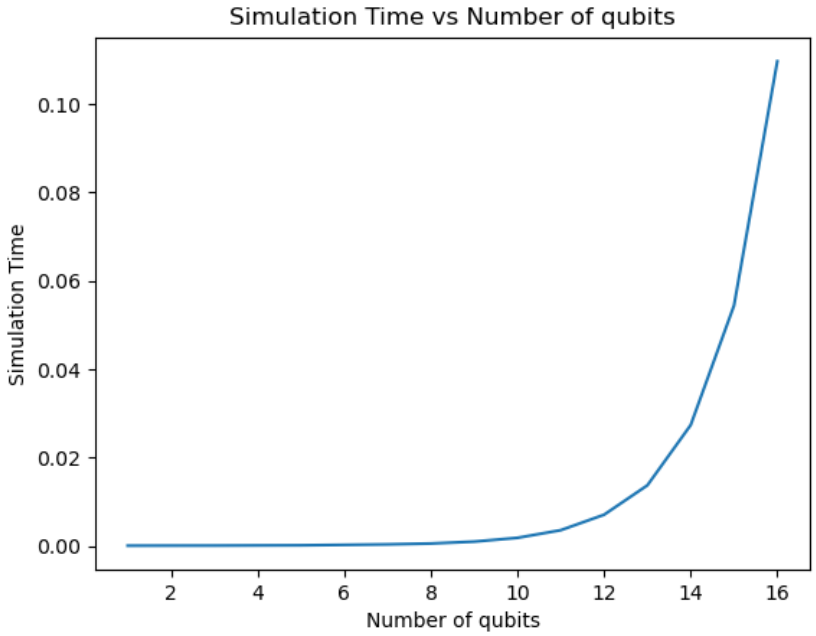

Figure 5: Simulation Time vs Number of qubits for 14 operations.

- We observe that overall the simulation time increases exponentially with increasing number of qubits. This is expected as we are maintaining a state of length 2^n and we apply the various operations on them. Therefore it makes sense that the simulation time will increase with increasing number of qubits.
- After setting the number of operations as constant, the only parameter the time depends on is the number of qubits and its dependence is exponential.

2.5.3 Benchmarks: Variation of Simulation Time with number of operations:

The following is the plot of the amount of time taken to simulate the various benchmarks. Each qasm program was run 10 times and the average is taken. The plot highlights the difference in the Simulation time as a function of the number of lines. The number of lines ineherently shows the number of quantum operations performed.

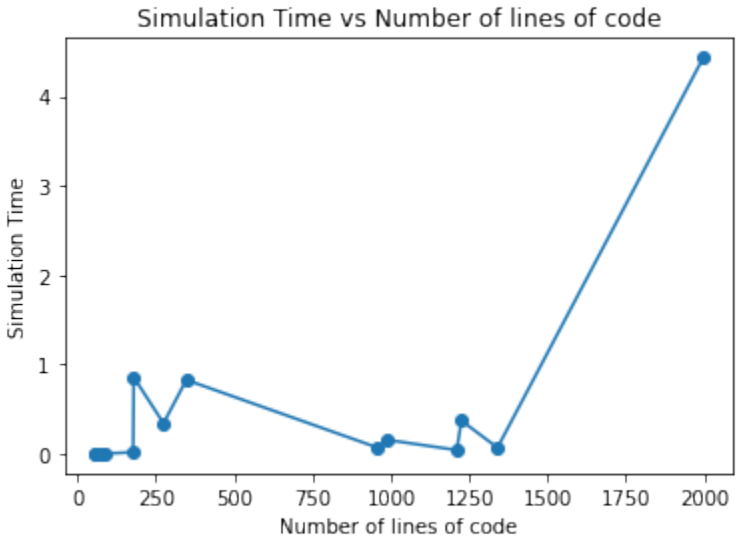

Figure 6: Simulation Time vs Number of lines

- We observe that overall the simulation time increases with increasing number of lines. This is expected as increasing the number of lines means that the number of operations also increase.
- Unlike number of qubits, we see that number of lines is not that major a contributer. This is evident from the numerous exceptions to the generic trend. For instance, there is the first jump at 179 as that is a 16 qubit program. Similarly 347 is a 15 qubit program.

2.5.4 Variation of Simulation Time with number of operations for a given number of qubits:

The following is the plot of the amount of time taken to simulate a custom hand written code of 12 qubits for various number of operations. Each qasm program was run 10 times and the average is taken. The plot highlights the difference in the Simulation time as a function of the number of operations given a constant number of qubits.

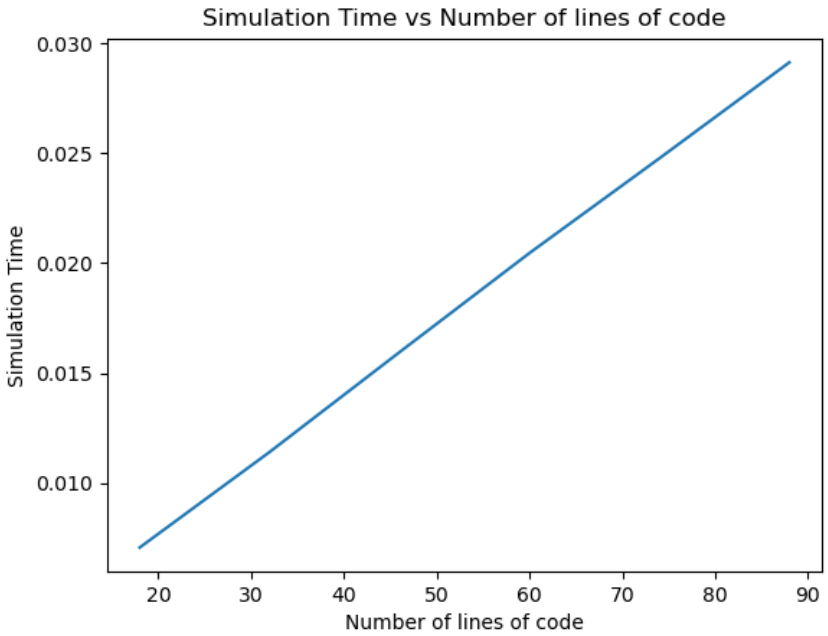

Figure 7: Simulation Time vs Number of operations for 12 qubits.

- We observe that overall the simulation time increases linearly with increasing number of operations. This is expected as we are maintaining a state of length 2^12 and we apply the various operations on them. Therefore it makes sense that the simulation time will increase with increasing number of operations.
- After setting the number of qubits as constant, the only parameter the time depends on is the number of operations and its dependence is linear.

2.5.5 Benchmarks: Variation of Simulation Time with Quantum Volume:

The following is the plot of the amount of time taken to simulate the benchmark programs and plot them as a function of quantum volume. Quantum Volume can be defined as the product of the number of qubits and the number of operations. Each qasm program was run 10 times and the average is taken.

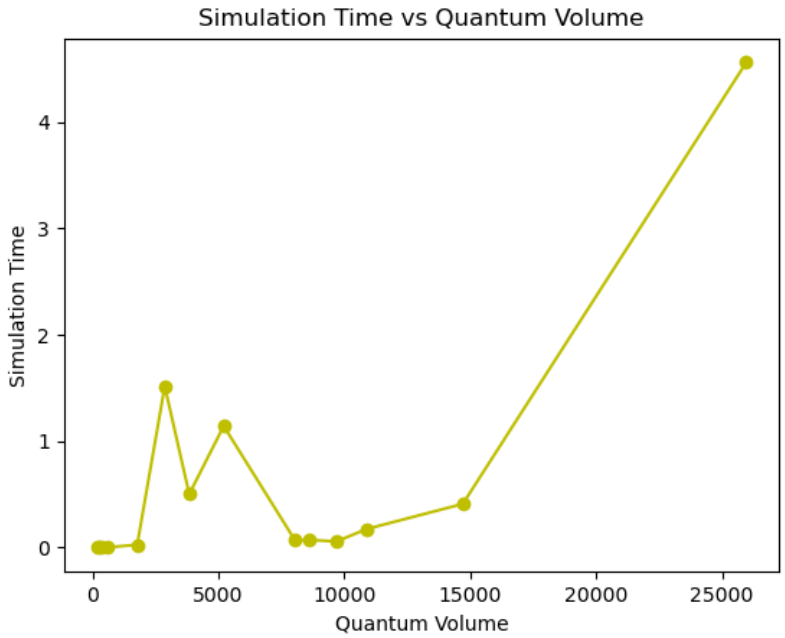

Figure 8: Simulation Time vs Quantum Volume.

- We do not observe a monotonic trend. Some programs with higher qubits but not that high a number of operations seems to take more time. Therefore number of qubits require more priority.
- Though Quantum Volume is a great real world measure of how good a quantum computer is, it cannot be taken as a measure of simulation difficulty. This is quite obvious considering the fact that executing an operation or storing the state is very different between quantum exection and simulation. For example, operation of a 2 qubit gate is a complicated process involving register allocation and swapping in a Quantum Computer while it is just an if statement in simulation.

2.5.6 Benchmarks: Variation of Simulation Time with number of qubits and lines of code:

We observed that the simulation time depends on both the number of qubits and the number of lines of code. We cannot plot a direct comparison to it as the number of lines are in 100s while the number of qubits are all between 3 and 16. So the two factors and normalized and a linear function of the two factors are plotted against the simulation time. The linear function obviously had to give more weightage to number of qubits as that turned out to be more determinant of the final simulation time (based on previous graphs). So we define the parameter p as:

$$p = 5\alpha + \beta$$

where α is the normalized number of qubits and β is the normalized number of lines of code.

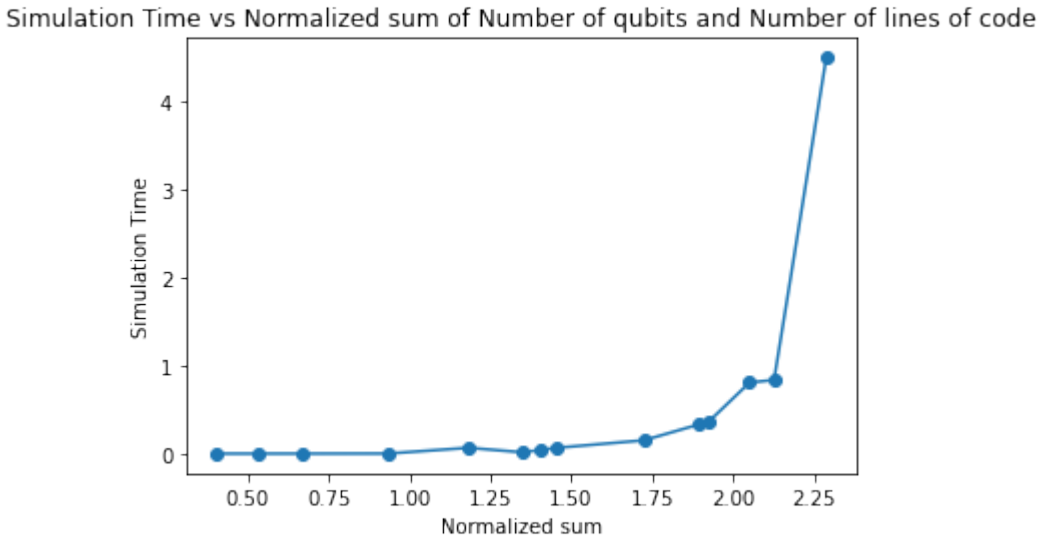

Figure 9: Simulation Time vs p

- We observe a monotonic relationship and this parameter p can be used as an estimate of the difficulty of a problem.
- The term p was derived at using the previous two graphs. As can be observed from 4, the number of qubits is the major factor that decides the final simulation time. This makes sense mathematically as the increase in number of qubits leads to an exponential increase in the state size. From 6, we observe that the number of lines do play a significant role but there are multiple exceptions. Therefore, number of qubits should have a higher importance in the parameter, p. After trying a few possible values, 5:1 ratio seemed to provide good results.
- The table below is sorted on the increasing simulation time, which can be viewed as the increasing difficulty. From the table we can see how well the p value computed above can be seen as an estimate for the difficulty/simulation time of the programs.

Program	Qubits	Lines	p	Simulation Time
miller_11.qasm	3	54	0.404	0.0005519390106201172
decod24-v2_43.qasm	4	56	0.534	0.0006067752838134766
one-two-three-v3_101.qasm	5	74	0.669	0.0008099079132080078
alu-bdd_288.qasm	7	88	0.933	0.0015270709991455078
mini_alu_305.qasm	10	177	1.348	0.015784263610839844
f2_232.qasm	8	1,210	1.401	0.04066896438598633
con1_216.qasm	9	958	1.455	0.06692695617675781
hwb5_53.qasm	6	1,340	1.182	0.06719827651977539
wim_266.qasm	11	990	1.723	0.15200591087341309
sym6_316.qasm	14	274	1.895	0.33472585678100586
cm152a_212.qasm	12	1,225	1.924	0.36562585830688477
rd84_142.qasm	15	347	2.050	0.8084738254547119
cnt3-5_179.qasm	16	179	2.125	0.837928056716919
squar5_261.qasm	13	1,997	2.287	4.5015480518341064

2.5.7 Variation of Simulation Time based on the gate type:

The following is the plot of the amount of time taken to simulate a custom hand written code of 10 qubits and 50 operations for various gate types. Each qasm program consists of only a single type of gate operation applied to various

qubits and their average execution time across 10 runs is taken. The plot highlights the difference in the Simulation time as a function of the type of operation.

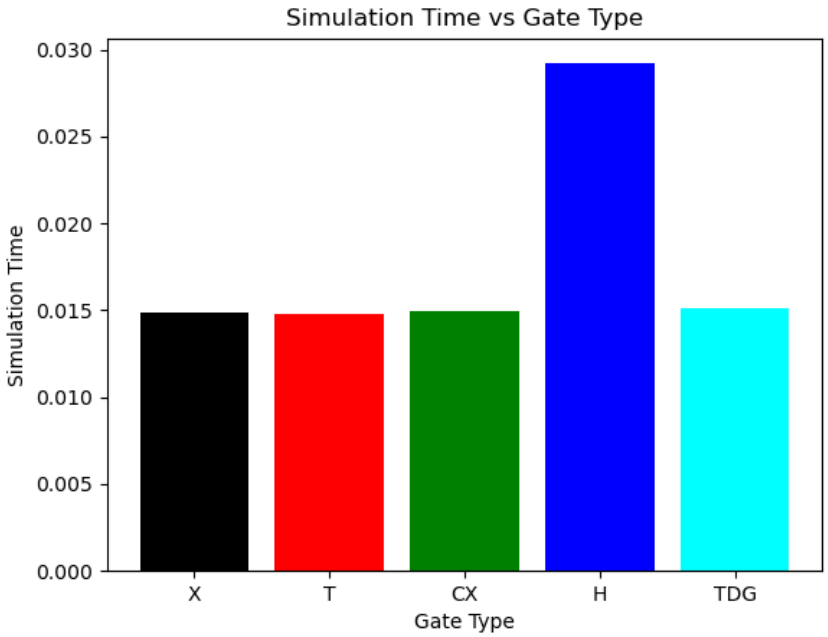

Figure 10: Simulation Time vs Type of gate operation.

- We observe that the simulation time of all the gates except the Hadamard are fairly consistent. This makes sense as all the other gate operations are mere multiplications while the Hadamard gate has to find the bit representation of the qubit and divide the amplitude between the 2 possible final states.
- It is interesting to note that though CX is a 2-Qubit operation, in the simulator it is merely and X operation with an added if condition and hence has almost the same execution time as a single qubit operation like an X gate.

2.5.8 Range Simulation Times:

The following is the plot of the range amount of time taken to simulate con1_216.qasm code. The qasm program is run 1000 times and the range is plotted.

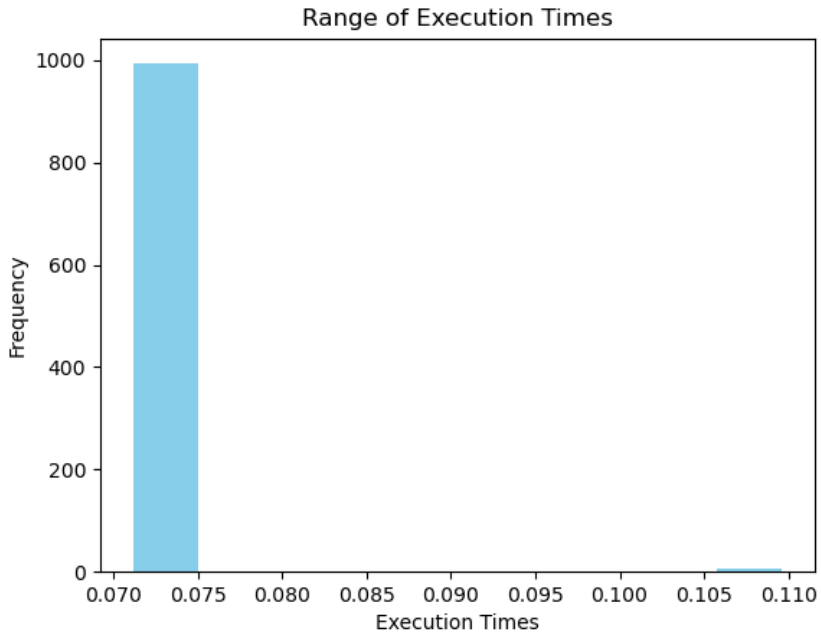

Figure 11: Range of Simulation Times.

- We observe that the simulation times are fairly consistent across runs. This is expected as we are running a local simulation and the environment is consistent across runs.
- We do observe some very rare edge cases but it is extremely rare.

2.5.9 Trying out edge cases:

- A large number of gates are tried and it can be observed that using the rounding off to 3 decimals can sometimes lead to inaccuracies. CompareSimulators.py is an exact copy of the compare_simulators.py given after removing the round off. This turns out to be more efficient in handling edge cases.
- Then the number of qubits is increased. For 24 qubits, the program seems to run endlessly but has not crashed the system. For 32 qubits, the program is instantly killed due to its high memory requirements.

3 Readme

The CompareSimulator.py is the given file to compare the results of my Simulator with that of Cirq. The QASM folder is given as well. Ensure cirq is installed. Then change directory to the Simulators folder.

Input and Output:

- Input: Pass the QASM files that satisfy the given grammar to a folder and pass the folder name as input to test.
- Output: The file name will be printed out along with the result: True/False. The result shows if the simulator generated a final state that is close enough to that generated by the cirq simulator.
- Output Graphs: Output graphs are saved into the Simulator Folder.

Run:

- To compare the simulators for the 14 Benchmark programs. Use the compare_simulators.py already added. A copy of the Simulator.py code called cs238.py is created for compatibility. Then run this:

```
python compare_simulators.py QASM
```

- To compare the simulators for the traditional algorithms:

```
python CompareSimulators.py QASM_2
```

- To run the simulation time tests on the given Benchmarks:

```
python Test.py QASM
```

- To run the simulation time tests for custom programs of same number of qubits:

```
python Test.py QASM_SameQubits
```

- To run the simulation time tests for custom programs of same number of operations:

```
python Test.py QASM_SameLines
```

- To run the simulation time tests for various gate types:

```
python Test_Gates.py QASM_GateTypes
```

- To find the range of simulation times:

```
python Test_Range_Times.py QASM
```