Quantum Circuit Simulator in a Parallel Environment

CSE 633
Spring 2021
Winton, Daniel

What is a Quantum Circuit?

A quantum circuit is a model for quantum computation in which a collection of qubits initialized to some state are run through a series of quantum gates.

They allow us to simulate quantum algorithms without dealing with the physical obstacles that currently limit quantum computers.

Computing a Quantum Circuit

The canonical way to compute a quantum circuit is through use of matrix operations. A single qubit can be represented by a 2 x 1 matrix, a state of n qubits can be given as a 2^n x 1 column matrix and a quantum gate for n qubits can be given by 2^n x 2^n unitary matrix.

The result of the quantum circuit is the product of our initial 2^n x 1 matrix representing the initial n-qubit system by each of the matrices representing the quantum gates it passes through.

Singular Qubit States

Each qubit is in a superposition between the states 0 and 1. The state 0 is denoted |0> and is given by the matrix $|0> = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and the state 1 is denoted by |1> and is given by the matrix $|1>= \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. An arbitrary qubit Ψ is given by a linear combination of these states with $|\Psi>=c_1|0>+c_2|1>$ where c_1, c_2 are complex numbers such that $|c_1|^2+|c_2|^2=1$. We have that $|c_1|^2$ and $|c_2|^2$ give the probabilities that when Ψ is measured it will end up in state |0> or |1> respectively. Notice Ψ can also be given by the matrix $\Psi= \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$.

Multiple Qubit States

We can also represent multiple qubits using the same notation. For example a two qubit system is in a superposition of basis states |00>,|10>,|01>, and |11>. So for quantum system $|\Psi\rangle = c_1|00\rangle + c_2|01\rangle + c_3|10\rangle + c_4|11\rangle$ we have $|c_1|^2+|c_2|^2+|c_3|^2+|c_4|^2=1$ and the magnitude of these scalars c_i squared gives the probability that the qubit when measured will collapse into the state it is in front of.

The matrix representation of each of these 4 basis states can be given the kronecker product of the matrix representation of its two individual qubits.

For example |01> is given by $\begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$.

Quantum gates

Quantum gates are given by square unitary matrices. A unitary matrix is a matrix whose conjugate transpose is its inverse.

A simple example of a quantum gate for a single qubit is the Pauli-X gate, which sends the state |0> to the state |1> and the state |1> to the state |0>. It is given by $X=\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. This can be easily verified as for example X |0> = $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix}=\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ =|1>.

Quantum gates continued

What if we had initial state |11> and we wanted to apply the Pauli-X gate to the first qubit and do nothing to the second qubit? The state |11> is given by a 4x1 matrix and the Pauli-X gate is given a 2 x 2 matrix, so we cannot multiply them. We have to take the Kronecker tensor operation on the Pauli-X gate with the control gate (given by the identity matrix) to get a 4 x 4 matrix representing the gates our two qubit system is going through. We can now multiply these two matrices together to get the resulting system of the two qubits.

Simple Quantum Circuit Example

Here we compute the quantum circuit detailed in the last slide. We initialize our two qubit system to the state |11> and run the first qubit through the Pauli-X gate and the second qubit through the control gate. We end up with the state with matrix representation

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

=|01>, which is what we'd expect.

Arbitrary Qubit States

We use the same idea of using the Kronecker tensor product to get our collection of qubits in our initial system, and gates acting on them into appropriate sized matrices.

A system of n qubits will be represented by a 2^n x 1 matrix and a gate acting on this system will be given by a 2^n x 2^n matrix. Taking the product of these two matrices gives a representation of the resulting qubit system.

Notice that each position in the resulting column matrix corresponds to a basis state of qubits and the square of the magnitude of the element in that position yields the probability that when measured the system will end up in that corresponding basis state.

Why Use a Parallel Environment for Quantum Circuits?

We can compute a quantum circuit by multiplying the 2^n x 2^n matrices representing the quantum gates by the 2^n x 1 matrix representing the qubit system in the order that the gates act on the system.

This can be done in $\Theta(2^{2n+1})$ time. We take the inner product of each row of the gate matrix with the column system. This is $2^{(n+1)}$ operations (2^n multiplications and additions) for each row and we have 2^n rows, so this is 2^{2n+1} computations.

Notice that this is assuming we have integer entries, but we could have complex entries instead. To multiply two complex numbers we have to take four integer products, two integer sums and an integer difference. To add two complex integers we have to do two integer sums.

Qubit Run Time Growth

Imagine we have an n qubit register and we run it through one quantum gate (where all of our entries are integers). This means we have the product of a 2^n x 2^n matrix by a 2^n x 1 matrix. As shown in the previous slide, we have 2^{2n+1} computations. The chart to the right shows just how quickly this value grows as we increase the value of n (i.e. the number of qubits in our system.) For 16 qubits we have about 8.59 billion computations and for 32 qubits we have about 3.69 quintillion computations.

n	2^{2n+1}
1	8
2	32
4	512
8	131072
16	8.59E9
32	3.69E19

Choosing an Algorithm for Matrix Multiplication

My quantum circuit simulator will use the Scalable Universal Matrix Multiplication Algorithm (SUMMA) to compute the matrix multiplication necessary in a parallel environment. There are many parallel algorithms for matrix multiplication, however SUMMA has less constraints than other algorithms -- such as not necessitating square matrices -- making it a good choice for this project.

Outer Product of Vectors

SUMMA makes use of the outer product operation of vectors.The outer product of vectors is actually the kronecker tensor product treating the first vector in the product as a column matrix and the second vector as a row matrix. Given two vectors **u**=($u_1,u_2,...,u_n$) and **v**=($v_1,v_2,...,v_m$) we have the outer product of **u** and **v** is denoted by **u ⊗ v** and is given by the matrix **u ⊗ v =**

$$\begin{pmatrix} u_1v_1 & u_1v_2 & \dots & u_1v_m \\ u_2v_1 & u_2v_2 & \dots & u_2v_m \\ . & & \dots & . \\ . & & & . \\ . & & & . \\ u_nv_1 & u_nv_2 & \dots & u_nv_m \end{pmatrix}.$$

SUMMA Algorithm

Let **A** be an m x n matrix and **B** be an n x p matrix. The product, which we will denote **C**, of **A** and **B** is an m x p matrix. The SUMMA Algorithm calculates **C** by computing n partial outer products. The algorithm is as follows

```
For k:= 0 to n-1
    C[:,:]+=A[:,k] ⊗ B[k,:].
```

The algorithm computes the outer product for each column of **A** with its corresponding row of **B** and takes the sum of each of these to obtain **C**.

SUMMA Example

Let **A**= $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ **B**= $\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$.Then **C**=**A** x **B** = $\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$. This can be easily verified using the standard method of matrix multiplication. Computing **C** using SUMMA we compute the outer product $\begin{pmatrix} 1 \\ 3 \end{pmatrix} \otimes \begin{pmatrix} 5 & 6 \end{pmatrix}$ = $\begin{pmatrix} 1\cdot5 & 1\cdot6 \\ 3\cdot5 & 3\cdot6 \end{pmatrix}$ = $\begin{pmatrix} 5 & 6 \\ 15 & 18 \end{pmatrix}$ and the outer product $\begin{pmatrix} 2 \\ 4 \end{pmatrix} \otimes \begin{pmatrix} 7 & 8 \end{pmatrix}$ = $\begin{pmatrix} 2\cdot7 & 2\cdot8 \\ 4\cdot7 & 4\cdot8 \end{pmatrix}$ = $\begin{pmatrix} 14 & 16 \\ 28 & 32 \end{pmatrix}$. We then take the sum of these two outer products and get the matrix $\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$, as desired.

Easier Parallel Algorithm/Implementation

In the case that our matrices can fit on a single processor we can run a parallel version of SUMMA as follows:

1. Partition the 2^n x 2^n square gate matrix column wise into sub matrices that are approximately 2^n x $2^n/p$ matrices, where p is the number of processors.
2. Scatter each of these sub-matrices to processors.
3. Partition the 2^n x 1 system matrix rows wise into sub matrices that are approximately $2^n/p$ x 1 matrices (the number of rows per sub matrix will correspond to the number of columns per sub matrix above.)
4. Scatter each of these sub matrices to processors.
5. Compute the outer product of the corresponding columns and rows in the processor and add them together.
6. Gather the resulting matrices back to the original processor.
7. Take the sum of these matrices, yielding our resulting qubit system matrix.

Easier Parallel Algorithm Example Part 1

Let’s say we wanted to multiply together the matrices $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ and $\begin{pmatrix} 10 \\ 11 \\ 12 \end{pmatrix}$

and and we were working on four processors. We would partition and scatter them (from processor 1) to processors 2, 3 and 4, like so:

P_2 $\qquad$ P_3 $\qquad$ P_4

$\begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix} \begin{pmatrix} 10 \end{pmatrix}$ $\qquad$ $\begin{pmatrix} 2 \\ 5 \\ 8 \end{pmatrix} \begin{pmatrix} 11 \end{pmatrix}$ $\qquad$ $\begin{pmatrix} 3 \\ 6 \\ 9 \end{pmatrix} \begin{pmatrix} 12 \end{pmatrix}$

Easier Parallel Example Part 2

Now we compute the outer product on each of our three processors.

On P_2 we find $\begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix} \otimes \begin{pmatrix} 10 \end{pmatrix} = \begin{pmatrix} 10 \\ 40 \\ 70 \end{pmatrix}$, on P_3 we get $\begin{pmatrix} 22 \\ 55 \\ 88 \end{pmatrix}$ and on P_4 we get $\begin{pmatrix} 36 \\ 72 \\ 108 \end{pmatrix}$.

We then gather these three column vectors back to processor one and add them together to get column vector $\begin{pmatrix} 68 \\ 167 \\ 266 \end{pmatrix}$.

This is the product of the the two original matrices, as desired.

Harder Parallel Idea

In the case that (some sub-collection) of our matrices does not fit on a single processor we need to change the way we implement creating the matrices representing the initial quantum system and the quantum gates. We construct a parallel kronecker tensor product so that the information is already scattered to the processors in the way we dictated on the previous slide. This will eliminate several steps of the previous algorithm as well as deal with the issue of too much information for a single processor to handle. In the easier parallel algorithm we could only compute circuits for up to 11 qubits. With the harder parallel algorithm we can compute circuits for 13 qubits on two processors and 14 qubits with at least 8 processors.

Harder Parallel Algorithm/Implementation

1. Find least m such that 2^m>p, where p is the number of processors, by computing $\log_2 p$, truncating it, and then adding 1. Let's call this number k.
2. Take the Kronecker product of the first k qubits and the Kronecker product of the quantum gates acting on them. We might need more than k as some gates act on multiple qubits. We can resolve by adding to k -- up until we get to the point where we get k=n where n is the number of qubits and then we will just run the easier algorithm.
3. Partition and then scatter the resulting initial system matrix row wise into p rows and the gate matrix into p columns.
4. Take and broadcast the Kronecker product of the final n-k qubits and the gates acting on them.
5. Take the Kronecker product on each processor of the portion of the first k qubit initial system scattered to it and the n-k qubit system broadcasted to it.
6. Take the Kronecker product on each processor of the portion of the quantum gate acting on the first k qubits scattered to it and the quantum gate acting on the final n-k qubits broadcasted to it.
7. Compute the outer product of the corresponding columns and rows of initial system matrix and the quantum gate matrix on each processor and add them together.
8. Gather the resulting matrices back to the original processor.
9. Take the sum of these matrices, yielding our resulting qubit system matrix.

Harder Parallel Algorithm Notes

For k less than (rounded down) n/2, we could let k equal (rounded down) n/2. This would help with the memory issue even more and we could possibly compute circuits with more qubits.

The harder parallel algorithm works for initial qubit system and the first matrix it goes through. After this we treat the qubit system as we do in the easier algorithm and the quantum gate as we do in the harder algorithm.

Outputting the Collapse State Probabilities

Typically a quantum circuit simulator gives as an output the possible outcome basis states of our system and the probability that when the system is measured we end up in each of these states.

To get this desired output, I created a function (based on the itertools package product function) that creates a list of all n-length permutations of 0’s and 1’s yielding the possible outcome basis states in the same order that a system matrix represents them.

We then iteratively take the magnitude squared of the system matrix entries and the corresponding state from the permutation list.

Scope of My Project

My project is actually two-fold.

Part 1: I am using a random initial system made up of qubits that are randomly assigned to be a zero qubit or a one qubit and a random quantum gate made up of quantum gates that are randomly assigned from a list of the most common quantum gates. I then use these two matrices to analyze the harder parallel algorithm for multiplication.

Part 2: I made a quantum circuit simulator that is fed user input. The user decides how many qubits they want (with a restriction of 13 qubits) and what gates they want to run the qubits through. They then receive as an output the probabilities of collapsing in each state when the system is measured.

What I have Done and What is Left to do?

The “part 1” of my project was completed and the results and discussion of the results follow this slide.

For “part 2,” I have a working model of a quantum circuit simulator.

I want to improve the UI to make it more user friendly and aesthetically pleasing.

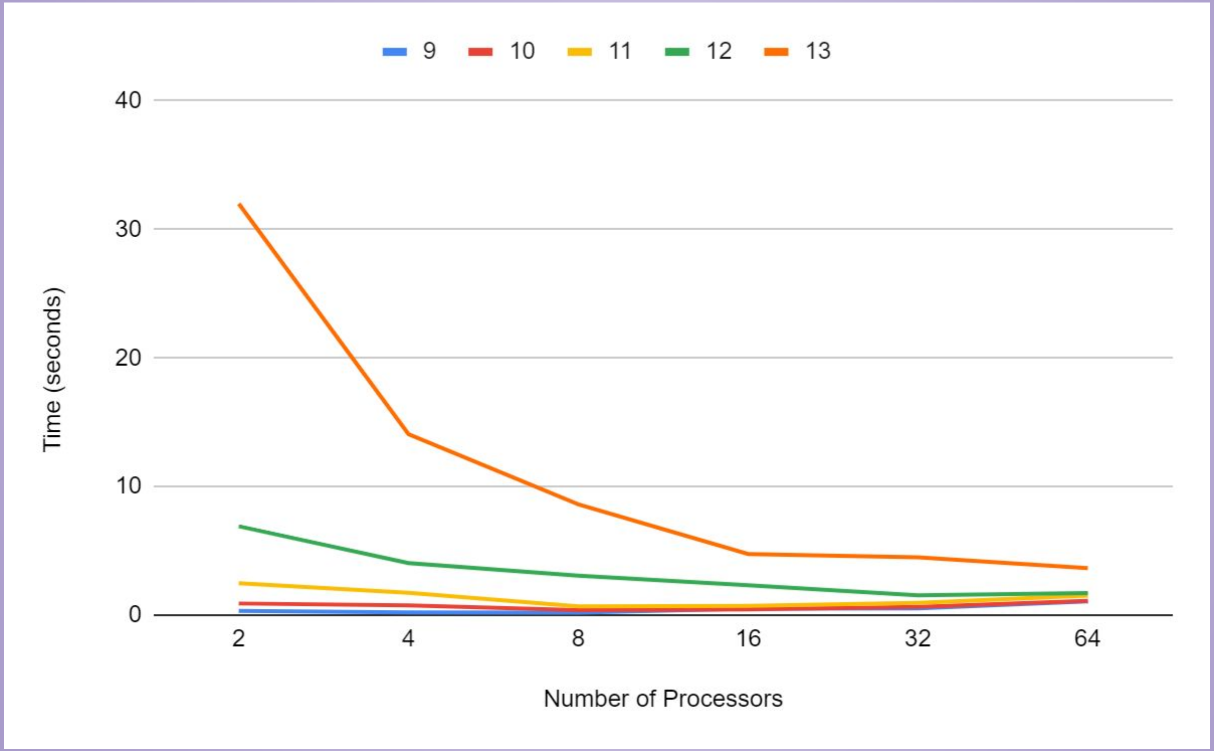
I want to improve the functionality of the quantum circuit simulator.

I will continue working on this project after this term ends!

Results

Processors by Qubits	9	10	11	12	13
2	0.3251357079	0.9020287275	2.4785748	6.91037333	32.00427692
4	0.2171532154	0.7471555948	1.728068304	4.038782382	14.07228718
8	0.1785531044	0.3852031469	0.6742284536	3.070298147	8.605862141
16	0.4911147833	0.4466632843	0.7243359804	2.324968433	4.743582511
32	0.5142184734	0.6466075659	0.9537539482	1.547226858	4.49328742
64	1.073498774	1.115497446	1.524611259	1.709969521	3.648782134

Results



9
10
11
12
13
40
30
20
10
0
Time (seconds)
2
4
8
16
32
64
Number of Processors

Discussion

We see parallel speed-up and then slow-down for 9-12 qubits. The parallel slow-down starts earlier when running on less qubits.

We would almost certainly see slow-down for 13 qubits if run on more processors. However, even for the 13 qubit case the amount of speed-up from 16 to 64 processors is so small that it would not be worth it in a real-world situation to pay for these extra processors.

For 14 qubits we start having issues with memory size and need to run on at least 8 processors. I did not add this data to my results because it would be impossible to compare to the other cases.

Huge Thank Yous!

I want to say huge thank yous to:

Professor Miller and Professor Regan at University at Buffalo

Ms. Cornelius at the Center for Computational Research- University at Buffalo

References

https://homepages.cwi.nl/~rdewolf/qcnotes.pdf

people.csail.mit.edu/virgi/matrixmult-f.pdf

cseweb.ucsd.edu/classes/fa12/cse260-b/Lec13.pptx (ucsd.edu)

http://www.mscs.mu.edu//~rge/mscs6060/assignments/Assignment3.pdf

Parallel Programming with MPI For Python - Research Computing in Earth Sciences (rabernat.github.io)

mpimatrixmult.pdf (cuny.edu)