Parallel Simulation of a Quantum Circuit

Benjamin Hall
Department of Computational Math, Science, and Engineering
Michigan State University
(Dated: December 7, 2018)

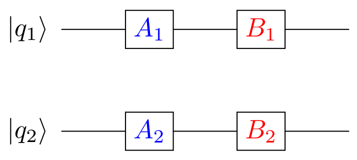
INTRODUCTION

Quantum computing algorithms are simulated on classical computers in order to check if they would execute properly on a quantum computer. The simulation algorithm can be parallelized by dividing up sequential steps among several processes and collapsing the outer loops of the algorithms for the required mathematical operations. This results in the general speed-up of run-time for the algorithm, the main goal of this project. The report will start with an explanation of quantum circuits, followed by the methodology of parallelization, the results of several scaling studies, and the discussion of the results.

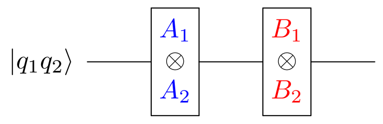
Simulation of Quantum Circuits

Quantum circuits are similar to classical logic circuits in their layout. They consist of a series of gates (A_i, B_i, etc.) that act on qubits $|q_i\rangle$. The qubits represent physical states, specifically, quantum binary systems (such as an electron being spin-up or spin-down, or a particle being in the ground state or first excited state of a system.) Mathematically, qubits are represented by 2×1 vectors. The gates represent physical events that happen to the qubits (such as the application of a microwave pulse or magnetic field.) Gates are mathematically represented by 2×2 unitary matrices. [1]

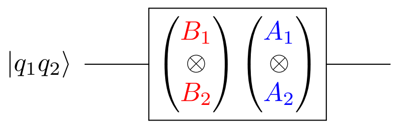
An example circuit with 2 qubits and a gate depth of 2 is given below:



Each column is collapsed by taking the Kronecker product of its gates



Then, the resulting row is collapsed by taking the matrix product of the gates in reverse order



Finally, this is executed as the following matrix vector multiplication

$$(B_1 \otimes B_2)\,(A_1 \otimes A_2)\,|q_1 q_2\rangle$$

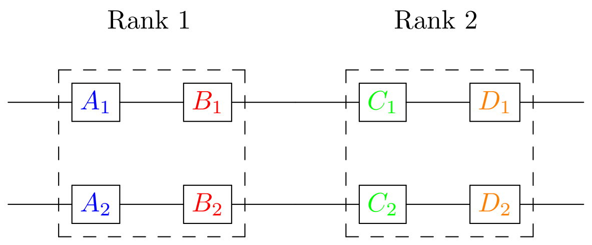
The complex squares of the elements of the resulting vector correspond to the probabilities of the qubits being measured in the corresponding states.

METHODS

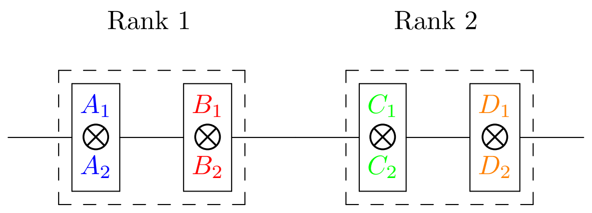
Dividing Work Using MPI

Through MPI, the columns are divided evenly among the processes. This results in each process being in charge of m/p gate columns, where m is the gate depth and p is the number of processes. (Note that p must divide m.) This is exemplified below with a gate that has 2 qubits and a gate depth of 4 being simulated by two processes.

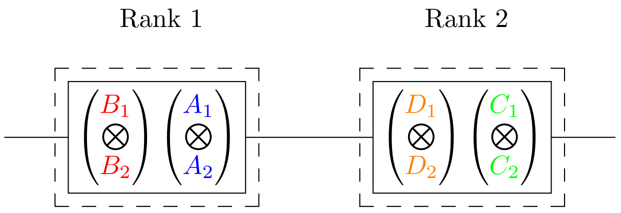
Step 1, the four columns are divided in half between the two processes.



Step 2, the columns are collapsed, with each rank only having to compute $m/p = 4/2$ Kronecker products.



Step 3, the resulting row is collapsed, with each rank only having to matrix-multiply together $m/p = 4/2$ matrices.



Step 4, the resulting p products are sent, in reverse order, to a master vector contained on all ranks via `MPI_Allgather`.

$$\left[\begin{pmatrix} D_1 \\ \otimes \\ D_2 \end{pmatrix}\begin{pmatrix} C_1 \\ \otimes \\ C_2 \end{pmatrix}, \begin{pmatrix} B_1 \\ \otimes \\ B_2 \end{pmatrix}\begin{pmatrix} A_1 \\ \otimes \\ A_2 \end{pmatrix}\right]$$

Note that, in order to gather the information using the predefined MPI data-type, `MPI_Type_vector`, I encoded all matrices as flattened 2d vectors (row-major).

Step 5, the master rank matrix-multiplies the products (contained in the master row) together in serial.

$$[(D_1 \otimes D_2)(C_1 \otimes C_2)]\,[(B_1 \otimes B_2)(A_1 \otimes A_2)]$$

This algorithm can be extended to an arbitrary number of qubits and gate depth. The MPI commands used to execute this method were learned from Parallel Computing in MPI and OpenMP [2].

Collapsing Loops Using OpenMP

Two mathematical operations are used in this simulation: matrix-multiplication and the Kronecker product.

Matrix-multiplication is a binary operation that takes in two matrices (size a_row × a_col and a_col × b_col) and outputs a matrix (of size a_row × b_col). Mathematically, the matrix-multiplication of $n_1 \times m_1$ and $n_2 \times m_1$ matrices A and B is given by

$$AB = \begin{pmatrix} \sum_k^{m_1} a_{0k}b_{k0} & \cdots & \sum_k a_{0k}b_{km_2} \\ \vdots & \ddots & \vdots \\ \sum_k^{m_1} a_{n_1k}b_{k0} & \cdots & \sum_k a_{n_1k}b_{km_2} \end{pmatrix}$$

The algorithm for computing it consists of 3 nested loops. Pseudo-code for the matrix-multiplication $AB = C$ is given below:

```
Matrix-Multiplication:
for 0 ≤ i < a_row
   for 0 ≤ j < a_col
      for 0 ≤ k < b_col
         C[i][j] += A[i][k] * B[k][j]
```

OpenMP is used to collapse the outer 2 loops. The best possible speed-up would be achieved if a_row × a_col threads are used as this would reduce the time complexity of the algorithm from $\mathcal{O}(n^3)$ to $\mathcal{O}(n)$ (for the matrix-multiplication of two $n \times n$ matrices.)

The Kronecker product ($\otimes$) is a binary operation that takes in two matrices (size a_row × a_col and b_row × b_col) and outputs a matrix (of size a_rowb_row × a_colb_col). Mathematically, the Kronecker product of an $n \times m$ matrix A and an arbitrary-size matrix B is given by

$$A \otimes B = \begin{pmatrix} a_{00}B & \cdots & a_{0m}B \\ \vdots & \ddots & \vdots \\ a_{n0}B & \cdots & a_{nm}B \end{pmatrix}$$

The algorithm for computing it consists of 4 nested loops. Pseudo-code for the Kronecker product $A \otimes B = C$ is given below:

```
Kronecker Product:
for 0 ≤ i < a_row
   for 0 ≤ j < a_col
      for 0 ≤ k < b_row
         for 0 ≤ l < b_col
            C[i * b_row + k][j * b_col + l]
            += A[i][j] * B[k][l]
```

OpenMP is used to collapse the outer 3 loops. The best possible speed-up would be achieved if a_row × a_col threads are used as this would reduce the time complexity of the algorithm from $\mathcal{O}(n^4)$ to $\mathcal{O}(n)$ (for the Kronecker product of two $n \times n$ matrices.)

Parallelization Strategies

I wrote two versions of the parallelized algorithm. The first, (`qcp.cpp` aka quantum computing parallel), works as described in the method section. This includes step 3, where each process collapses the part of the row that it was given. In the second method (`qcp2.cpp`) the matrix-multiplication of step 3 was replaced with concatenation. That is, the processes did not matrix-multiply together the matrices in their sections of the row; instead, they concatenated the vectors that represented these matrices. This concatenated vector was then gathered into one master vector by `MPI_Allgather`. The master rank then broke the the master vector back into the matrices (collapsed columns) that each rank had, and then matrix-multiplied them together in serial.

The first version was used in the series vs parallel test (from the results section) because it provides the best speed up by parallelizing the row collapse (step 3). However, the second method was used in the strong, weak, and thread scaling tests (results section) because the speed-up due to the parallelization of the collapsing of the rows (step 2) is more detectable. This is because, in method 2, step 2 is the only parallelized part of the algorithm.

Additional Parallelization Strategies

The division of the columns between ranks decreases the time to collapse the columns by a factor of the number of processors used (perfect speed-up). However, the division of columns between ranks does not provide perfect speed-up of the time to matrix-multiply together the matrices on the resulting row. If there are m columns and p processes, then each process matrix-multiplies together, m/p matrices in parallel. But then, the resulting p products are multiplied together in serial. This results in an overall run-time for the matrix-multiplication portion of the simulation (t_{mm}) that goes as

$$t_{mm} \propto \frac{m}{p} + p$$

which can be minimized by setting the derivative of t_{mm} with respect to p equal to zero

$$0 = \frac{\partial t_{mm}}{\partial p} = -\frac{m}{p_{\min}^2} + 1$$

which implies that the number of processes that minimizes the time to execute the matrix-multiplication portion of the algorithm is given by

$$p_{\min} = \sqrt{m}$$

Thus, while increasing the number of processes p decreases the run-time to collapse the columns, anything beyond $p = \sqrt{m}$ results in an increase in run-time to collapse the row. Further parallelization of the row collapse portion of the algorithm would increase overall speed-up and allow the number of processes to be increased, thus decreasing the run-time of the column collapsing portion of the simulation.

One possible method to do this is an algorithm analogous to the vector reduction method discussed in *Introduction to High Performance Scientific Computing* [3]. The method is optimal when the number of processes is half the number of matrices. Each process matrix-multiplies together its two matrices and then every other process passes its result to the process to its left. This continues recursively, with half the processes continuing on two do more work and half of them "dropping out"). Below is the pseudo-code for the method:

```
for (s = 2; s < 2*m; s *= 2)
    for (i = 0; i < m - s/2; i += s)
        if (rank == i/2)
        vec[i] = matmul(vec[i], vec[i + s/2])
```

where `vec` is the vector of matrices to be multiplied together and `matmul(A, B)` matrix-multiplies the matrices A and B together. A schematic for the example of this algorithm with 8 matrices is shown in Figure 1 below:

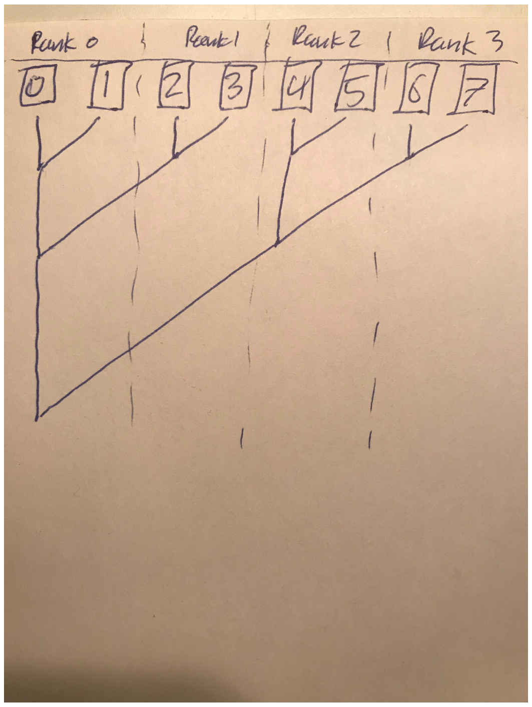


FIG. 1. Schematic of parallel algorithm to compute the matrix-multiplication of a string of 8 matrices.

This algorithm improves the run-time of the serial algorithm by a factor of $p/\log_2 n$. This sub-algorithm would replace the all-gather of the current simulation algorithm in step 4.

RESULTS

Verification

To verify the correctness of the serial and parallel code, I always had every gate be the Haddamard gate (H) in all implementations of the codes. The Haddamard gate puts a qubit into a superposition state and is represented by the following 2 × 2 unitary matrix:

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

It is also convenient that that H matrix-multiplied by itself, $H^2 = \mathbb{I}$, the identity matrix. Thus, if the gate depth is even, the initial state remains unchanged. And if the gate depth was odd, the final state is the (positive) equal superposition of all possible states. I always chose the initial state of all qubits equal to $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ (as most quantum computers do). Thus, the following should happened

$$\begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix} \to \begin{cases}\begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix}, & \text{if } m \text{ is even}\\ \frac{1}{2^{m/2}}\begin{pmatrix}1\\1\\\vdots\\1\end{pmatrix}, & \text{if } m \text{ is odd}\end{cases}$$

where m is the gate depth (number of columns of the circuit). Both programs (serial and parallel) always returned the correct vector, regardless of how many gates I used.

Scaling

Parallel vs. Serial

After a gate depth of 32, the parallel simulation (with 4 processes) becomes faster than the serial simulation (Figure 2). This partially meets the goal of reducing run-time of the algorithm. When it is faster, it is faster by a factor of 4 since I used it with 4 processes and the speed-up is perfect. I suspect that the parallel simulation is slower than the serial code for less then 32 processors because the extra communication time to `MPI_Allgather` from 4 processes outweighs the saved time due to parallelization with such few processes. The parallel simulation eventually achieves better speed-up because, while the communication time stays the same after a gate depth of 4 (since there are only 4 processes), the time saved by parallelization continues to increase with increasing gate depth. The communication time stays constant (for constant number of qubits) because the size of the matrix that results from each rank reducing its part of the row remains constant no matter how many matrices are multiplied together (step 3). This is because the resulting matrix of the matrix-multiplication of two $n \times n$ matrices is still an $n \times n$ matrix; that is, the size of the resulting matrix, and hence the amount of data, does not increase. Therefore, the size of the resulting matrix of a string of matrix-multiplications is the same regardless of the number of matrices in the string.

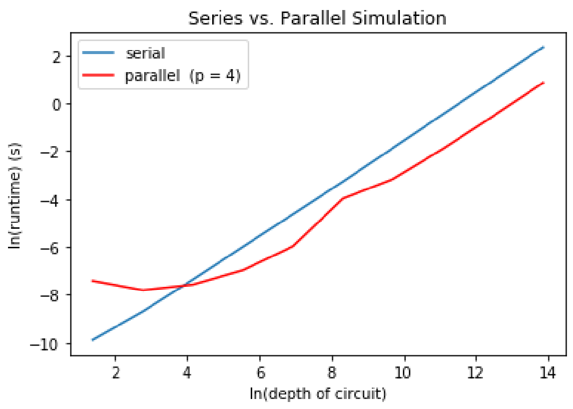


FIG. 2. Run-time is directly proportional to depth of circuit in the simulation of a 2×4^k circuit with $p = 4$ processes. $(k = 1, 2, ..., 10)$

Strong Scaling

In strong scaling, the gate depth is held constant as the number of processes increases. The strong scaling run-time starts off roughly inversely proportional to the number of processes (Figure 3). This meets the goal of reducing the run-time of the simulation. Then, it starts to flatten out. I suspect that this is because, while the collapsing of the columns is perfectly parallelizable, the collapsing of the row is not. In my algorithm, as mentioned in the Additional Parallelization Strategy sub-section, the number of matrices that must be multiplied together is actually equal to the number of processes. Thus, the run-tine increases as the number of processes increases. Additionally, as the number of processes increases, the communication time necessary to all-gather their data increases as well.

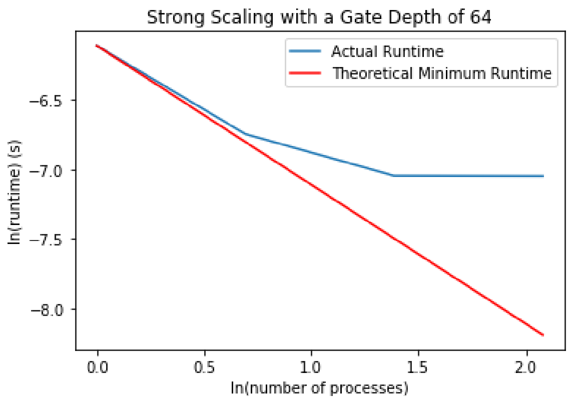


FIG. 3. Strong Scaling: Plot of run-time to compile a 2×64 circuit verses number of processes.

Weak Scaling

In weak scaling, as the number of processes increases, the gate depth increases proportionally so that the number of gate columns per process (8) stays constant. We actually see an increase in run-time that is directly proportional to the number of processes (Figure 4), not meeting the goal of reducing of the run-time of the simulation. This is because, even though the time to collapse the columns (step 2) is staying constant (every processes collapses 8 columns),the time to collapse the master column (step 5) continues to increase (since it goes as p).

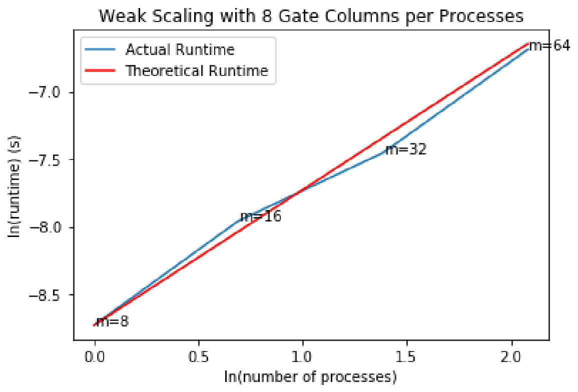


FIG. 4. Weak Scaling: Plot of run-time to compile a 2×64 circuit verses number of processes. Weak: Plot of run-time to compile a 2×8^m circuit vs. depth of circuit m.

Thread Scaling

The run-time is inversely proportional to the OpenMP thread count (Figure 5), fully meeting the goal of reducing the run-time of the algorithm. This is because, OpenMP was used to collapse the inner (perfectly nested) loops of the algorithms for the two mathematical simulations involved in the simulation (matrix-multiplication and the Kronecker product). Thus, the increase in threads results in perfect speed-up of run time because each process is only executing $1/p^{\text{th}}$ of the tasks required by the algorithms.

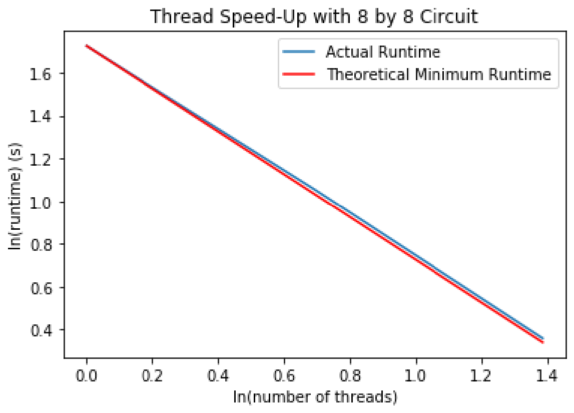


FIG. 5. Thread Scaling: Run-time is inversely proportional to thread count.

Discussion

Load Balancing

The same number of columns is given to each process in step 1. Thus for step 2 (collapsing of the row), the work is divided evenly among the processes. Thus, the number of matrices that each process matrix-multiplied together was also evenly distributed (step 3). This is guaranteed as I only ever tested the algorithm with a number of processes that divided the gate depth. Had the algorithm been built to handle cases when the number of processes did not divide the gate depth, the remainder of the division would have been sent to the master rank which would then have to do more work than the other ranks. Specifically, in step 2, the master rank would collapse $m/p + m \mod p$ columns while the rest would collapse only m/p. The same goes for the number of matrices that each rank would matrix-multiply together in step 3. I choose not to consider this case because then the size of the data that each rank gives `MPI_Allgather` would be different which is not handle-able by that MPI call. In either case, after the all-gather, only the master rank continues to do work (step 5), matrix-multiplying together the p matrices that it has been sent.

Now I will turn to the strategy mentioned in the Additional Parallelization Strategies sub-section. In each recursive step, each process has the same amount of work to do (multiply 2 matrices together). But, overall across all iterative steps, rank 0 has to do the most work (matrix-multiplication in all $\log_2(m)$ steps), followed by the ranks that are multiples of $m/4$ (matrix-multiplication for $\log_2(m) - 1$ steps), followed by the ranks that are multiples of $m/8$ (matrix-multiplication for $\log_2(m) - 2$ steps), and so on such that ranks that are multiples of $m/2^d$ have to do matrix-multiplication for $\log_2(m) - (d-1)$ steps. This continues until we reach the odd numbered ranks that only have to do matrix-multiplication on the very first recursive step.

Memory Usage

When the columns are divided up among the ranks using MPI, the resulting data that the separate ranks create (through collapsing the columns with the Kronecker product and collapsing their portion of the row with matrix-multiplication) is private to those individual ranks. Thus, it is an example of distributed (not shared) memory. Therefore, step 4 (the all-gather) needs to be implemented so that the master rank can access the resulting matrix of each of the ranks and matrix-multiply them together.

For the additional parallel strategy to collapse the row, each rank would have to send its resulting matrix to the rank to its left so that that rank may use it for the next recursion step.

As mentioned in the scaling studies, the increase in memory size and transfer required for the MPI_Allgather likely contributed to the non-perfect speed-up in the strong scaling study and the serial algorithm's out-performance of the parallel algorithm for less than 32 processes. Details are given in the Scaling section.

Implementation of HDF5

For my algorithm, the implementation of HDF5 was not necessary. I hard-coded in the initial state that each qubit would be and the gate that would fill the circuit matrix. Additionally, I wrote functions that then initialized the proper vector that represented n qubits of the same specified state and the $n \times m$ matrix that represented the quantum circuit filled with the specified gate. The only output required to do the scaling studies was the run-time which is just a single number (double). For verification, the resulting vectors were easily discernible (either a 1 followed by all 0s or all 1s as mentioned in the Verification sub-section

CONCLUSION

First, thee simulation of quantum circuits was explained step by step. Then, it was described how such a simulation could be (and was) parallelized. This in included breaking up the column collapsing evenly between processes that ran in parallel via MPI (step 1) and the collapsing of the outer loops of the algorithms for the two mathematical operations used in the simulation. Next, the two different versions of my code were explained and the pros and cons of both were contrasted. An additional parallelization strategy was explained that could have parallelized the serial second half of the simulation for future work. Next, several scaling studies were explained. The parallel-to-serial run-time ratio was shown to equal to the number of processes (when the number of processes was large enough). The run-time was shown to be inversely proportional to thread count. Strong scaling run-time was shown to be roughly inversely proportional to number of processes (initially) while weak scaling run-time was shown to be directly proportional. Finally, aspects of the algorithm such as the even load balancing of the algorithm and the distributed memory usage (and increase memory sharing) were discussed. To conclude, the parallelization of the simulation of a quantum circuit presented here does decrease run-time.

[1] Nielsen, Michael, and Isaac Chuang. *Quantum Computation and Quantum Information*

[2] Victor, Eijkhout. *Parallel Programming in MPI and OpenMP.* 2017.

[3] Victor, Eijkhout. *Introduction to High Performance Scientific Computing.* 2016.