Implementing and Evaluating the Efficiency of a Quantum Linear Systems Algorithm

Dhruv Yalamanchi

Abstract

We demonstrate the advantage quantum computers bring to solving computationally intensive tasks by presenting the results of determining the solutions to large systems of linear equations using both a quantum algorithm and a classical, deterministic algorithm. In the quantum approach, we implement the Harrow-Hassidim-Lloyd (HHL) algorithm for solving linear systems using a quantum programming framework called Qiskit. The HHL algorithm fundamentally operates using a method called eigenvalue inversion in which it produces and acts on estimates for the eigenvalues of the matrix that represents the coefficients in the system of linear equations. For the classical algorithm, we implement an algorithm in Python that computes the exact solution to the input system of equations using Gaussian elimination with partial pivoting, which is the most efficient classical matrix inversion algorithm. We then feed randomly generated square matrix equations representing N by N linear systems into these two algorithms and record how internal properties such as matrix condition number and sparsity impact metrics such as runtime and solution state fidelity. Scaling N enables us to experimentally determine the respective time complexities of the algorithms and perform a comparison of their efficiencies. We show that, unlike Gaussian elimination, our implementation of the HHL algorithm has a strong dependence on condition number and sparsity and scales subquadratically in N, achieving a polynomial speedup in complexity.

1 Rationale

The advent of quantum computing has introduced new ways of approaching computational problems. By leveraging quantum mechanical properties such as superposition and entanglement, this new paradigm of computing has been able to provide a polynomial or even exponential speedup over conventional methods for numerous computationally expensive tasks. The particular problem of solving linear systems of equations is not only of fundamental theoretical importance, but it also has a myriad of real-world applications such as in the solution of partial differential equations, machine learning, fitting polynomial curves, and even fluid simulation, often acting as an essential subroutine of more complex processes. However, as the amount of data used to describe the linear systems becomes extremely large, classical solvers, including both direct and iterative methods, often cannot provide solutions in a reasonable amount of time. The Harrow-Hassidim-Lloyd (HHL) algorithm [2, 5, 6] is a quantum algorithm designed to tackle the following problem: given a coefficient matrix $A \in \mathbb{C}^{N \times N}$ and a vector $\vec{b} \in \mathbb{C}^N$, produce a quantum state that is proportional to the solution vector $\vec{x} \in \mathbb{C}^N$ satisfying the equation $A\vec{x} = \vec{b}$. To approximate the threshold value of N at which the quantum algorithm surpasses classical algorithms in efficiency, we will experimentally determine and compare the time complexities for both the HHL algorithm and the Gaussian elimination (GE) with partial pivoting method. However, unlike direct numerical methods, the execution time for the HHL algorithm depends not only on the size of the input but also on internal characteristics of the system. Therefore, we will also investigate the quantum algorithm's dependence on inherent parameters like A's condition number κ (defined as the ratio of the matrix's largest and smallest eigenvalues) and sparsity s (defined as the percentage of elements that equal zero) to gain valuable insight into the circumstances under which the HHL algorithm yields solutions with optimal efficiency and accuracy. This information will reveal specific applications for which the quantum algorithm is more ideal to use than existing classical approaches.

2 Algorithm Outline and Implementation

We implement the HHL algorithm using the Qiskit quantum programming framework [1] and the GE algorithm using Python 3. To execute the quantum algorithm, we also employ the `statevector_simulator` backend provided by Qiskit Aer which allows us to simulate quantum circuits without noise. In Sections 2.1 and 2.2, we outline the steps of both algorithms and provide relevant implementation details.

2.1 Gaussian Elimination With Partial Pivoting

One of the fastest known direct classical methods for solving linear systems is Gaussian elimination. By performing elementary row operations such as swapping different rows, multiplying all of the elements in a row by a certain value, or adding a multiple of one

row to another, this algorithm transforms the coefficient matrix A into an upper triangular matrix, or one that is in row echelon form. However, the possibility of division by 0 during one of these manipulations is a problem that arises from this method. To make the algorithm more numerically stable for any general matrix and reduce round-off error, we also integrate partial pivoting into our implementation. For this algorithm, we represent the $N \times N$ system in matrix form:

$$\underbrace{\begin{bmatrix} a_{1,1}^{(0)} & a_{1,2}^{(0)} & \dots & a_{1,N}^{(0)} \\ a_{2,1}^{(0)} & a_{2,2}^{(0)} & \dots & a_{2,N}^{(0)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N,1}^{(0)} & a_{N,2}^{(0)} & \dots & a_{N,N}^{(0)} \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}}_{\vec{x}} = \underbrace{\begin{bmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_N^{(0)} \end{bmatrix}}_{\vec{b}} \tag{1}$$

where the superscript above each element indicates the number of transformations that have been applied to it.

2.1.1 Transforming A into Row Echelon Form

A *pivot* element is essentially one of form $a_{k,k}$ where $1 \leq k \leq N$ for a fixed row, called the pivot row. We first let row 1 be the pivot row. If the first pivot element $a_{1,1}$ is zero or is very small, we interchange the first row with another such that the new pivot element is more stable. Once the pivot has been corrected, we zero out all of the elements below the pivot in the same column by multiplying all of the elements in row i by $\frac{a_{1,1}}{a_{i,1}}$ for $2 \leq i \leq N$. After this transformation, all of the elements below the pivot have the value $a_{1,1}$. We can then replace row i with the difference between the pivot row and row i, which produces the equation:

$$\begin{bmatrix} a_{1,1}^{(0)} & a_{1,2}^{(0)} & \dots & a_{1,N}^{(0)} \\ 0 & a_{2,2}^{(1)} & \dots & a_{2,N}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{N,2}^{(1)} & \dots & a_{N,N}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1^{(0)} \\ b_2^{(1)} \\ \vdots \\ b_N^{(1)} \end{bmatrix} \tag{2}$$

Note that these row operations are applied to the elements of $\vec{b}$ as well. We then fix the second row to be the pivot row, letting $a_{2,2}$ become the new pivot. Following the same procedure outlined above, we zero out the elements below the pivot in its respective column. After performing the rest of the transformations, the resultant matrix equation is:

$$\begin{bmatrix} a_{1,1}^{(0)} & a_{1,2}^{(0)} & \dots & a_{1,N}^{(0)} \\ 0 & a_{2,2}^{(1)} & \dots & a_{2,N}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{N,N}^{(N-1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1^{(0)} \\ b_2^{(1)} \\ \vdots \\ b_N^{(N-1)} \end{bmatrix} \tag{3}$$

A is now in row echelon form.

2.1.2 Performing Back Substitution

Now that A is in row echelon form, we have sufficient information to determine the solutions to the system. From the Nth row of the system, we have:

$$(a_{N,N})\, x_N = b_N \implies x_N = \frac{b_N}{a_{N,N}} \tag{4}$$

Similarly, the $(N-1)$th row gives:

$$(a_{N-1,N-1})\, x_{N-1} + (a_{N-1,N})\, x_N = b_{N-1} \implies x_{N-1} = \frac{b_{N-1} - (a_{N-1,N})\, x_N}{a_{N-1,N-1}} \tag{5}$$

which we can evaluate by substituting the value of x_N. In general, any component x_j of $\vec{x}$ can be computed as follows:

$$x_j = \frac{b_j - \sum_{i=j+1}^{N} (a_{j,i})\, x_i}{a_{j,j}} \tag{6}$$

This process of computing x_N, then substituting that back into the previous equation to determine x_{N-1}, and repeating through x_1 is known as *back substitution*. By converting A to row echelon form and performing these back substitutions, we are able to produce the complete solution vector $\vec{x}$.

2.2 The HHL Algorithm

In the quantum approach, the matrices and vectors are treated as quantum mechanical operators and states, respectively. Thus, the problem $A\vec{x} = \vec{b}$ becomes:

$$A\,|x\rangle = |b\rangle \tag{7}$$

where $|x\rangle$ and $|b\rangle$ are quantum states that store the respective unit vector representations of $\vec{x}$ and $\vec{b}$. The numerical solution to (7) is $|x\rangle = A^{-1}\,|b\rangle$, which the HHL algorithm prepares using eigenvalue inversion. If A is Hermitian, meaning that it possesses real eigenvalues as well as an orthonormal eigenbasis, we can treat its eigenvectors as the orthonormal basis states of an N-dimensional quantum system. Moreover, an $N \times N$ Hermitian matrix with N distinct eigenvalues $\lambda_j \in \mathbb{R}$ and corresponding eigenvectors $|u_j\rangle \in \mathbb{R}^N$ has the spectral decomposition $A = \sum_{j=0}^{N-1} \lambda_j\,|u_j\rangle\,\langle u_j|$, which implies:

$$A^{-1} = \sum_{j=0}^{N-1} \lambda_j^{-1}\,|u_j\rangle\,\langle u_j| \tag{8}$$

We can also express $|b\rangle$ as a normalized quantum state in the eigenbasis of A:

$$|b\rangle = \sum_{j=0}^{N-1} b_j \, |u_j\rangle \tag{9}$$

where $b_j \in \mathbb{C}$ such that $\sum_{j=0}^{N-1} |b_j|^2 = 1$. Here, knowledge of the values of the coefficients b_j is not required. Using (8) and (9), we can also express the solution vector in this eigenbasis:

$$\begin{aligned} |x\rangle &= \left(\sum_{j=0}^{N-1} \lambda_j^{-1} \, |u_j\rangle \langle u_j| \right) \left(\sum_{k=0}^{N-1} b_k \, |u_k\rangle \right) \\ &= \sum_{j=0}^{N-1} \lambda_j^{-1} b_j \, |u_j\rangle \end{aligned} \tag{10}$$

We approach the problem of solving an $N \times N$ system of equations using the quantum circuit model, which enables us to carry out manipulations on a set of qubits using a sequence of quantum gates. The objective of the HHL algorithm is to operate on these qubits in a manner such that their final state provides an approximation to the desired solution given by (10).

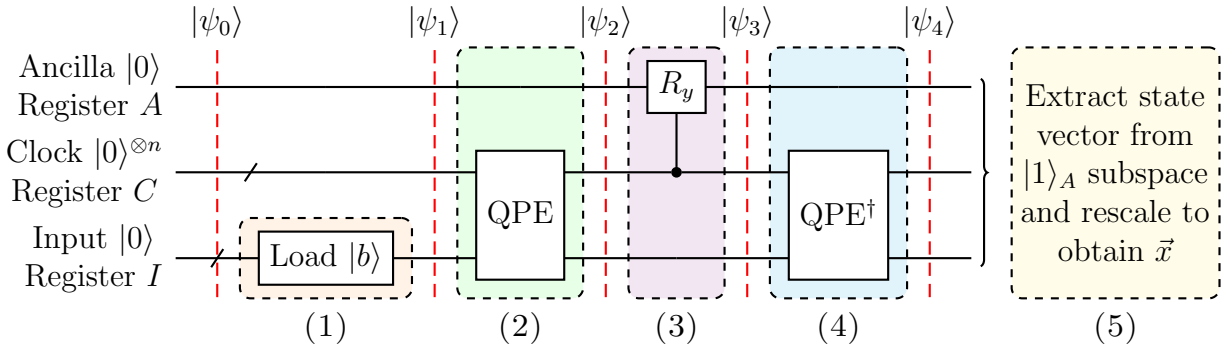

Figure 1: Circuit diagram for the HHL algorithm. The five time slices indicate the total state of the system $|\psi_i\rangle$ before each of the five shown steps. The ancilla register A contains the ancillary qubit upon which we will perform the conditional rotation R_y. The clock register C contains the n qubits we will use to estimate the eigenvalues of A. The input register I contains $m = \log_2 N$ qubits and will store $|x\rangle$.

Broadly, the HHL procedure consists of five primary steps, which are illustrated in Figure 1. We initialize all qubits to $|0\rangle$, so the system begins in the state:

$$|\psi_0\rangle = |0\rangle_A \, |0\rangle_C^{\otimes n} \, |0\rangle_I \tag{11}$$

where the subscripts below the kets indicate the registers which store the corresponding states. For our Qiskit implementation of HHL, we first declare a `QuantumRegister` and a

`QuantumCircuit` to which we will append the different subroutines of the algorithm.

2.2.1 Loading $|b\rangle$

As shown in Figure 1, Step (1) of the HHL algorithm is to encode $|b\rangle$ into the input register. In Qiskit, we achieve this by creating a `Custom` state object to initialize register I given both $\vec{b}$ and the number of necessary qubits m. We then assemble the initial state's circuit by calling `construct_circuit` and appending it to the `QuantumCircuit`. The state of the system thus becomes: $|\psi_1\rangle = |0\rangle_A |0\rangle_C^{\otimes n} |b\rangle_I$. Since any quantum state can be decomposed into an arbitrary, orthonormal basis, we can equivalently express the state in the eigenbasis of A:

$$|\psi_1\rangle = \sum_{j=0}^{N-1} b_j |0\rangle_A |0\rangle_C^{\otimes n} |u_j\rangle_I \tag{12}$$

2.2.2 Performing QPE

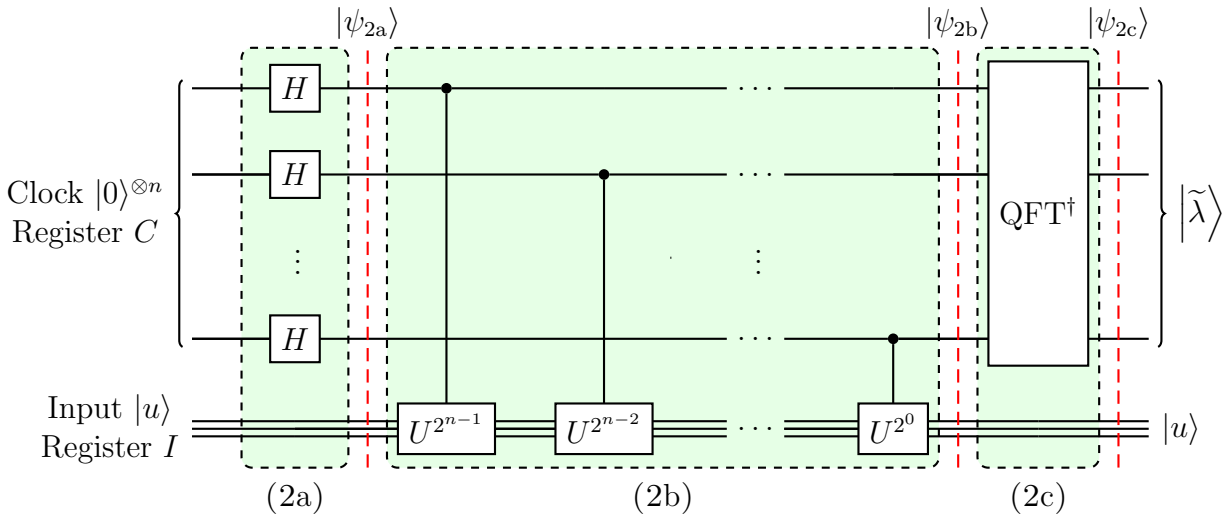

Figure 2: Circuit diagram for the quantum phase estimation subroutine. The top register contains the n clock qubits, all of which have been initialized to $|0\rangle$. The bottom register is the input register which stores the eigenvectors $|u_j\rangle$ and is unaffected throughout the entire QPE algorithm.

Step (2) of the algorithm is to apply a quantum phase estimation (QPE) routine. When a unitary operator U acts on one of its eigenvectors $|u\rangle$, it is essentially applying some phase φ that depends on the eigenvector we are using:

$$U |u\rangle = e^{2\pi i\varphi} |u\rangle \tag{13}$$

The goal of the phase estimation algorithm is to estimate φ. The precision of this estimate depends on n or the number of clock qubits. Letting Hermitian operator A have eigenvectors

$|u_j\rangle$ and corresponding eigenvalues λ_j, we have that $e^{2\pi iA}$ is unitary with eigenvectors $|u_j\rangle$ and eigenvalues $e^{2\pi i\lambda_j}$. Thus, it is clear from (13) that, by letting $U = e^{2\pi iA}$, we can use QPE to determine the eigenvalues λ_j of A. As can be seen in Figure 2, Step (2a) of QPE is to apply a Hadamard transform $H^{\otimes n}$ [3] to the clock qubits. This produces the state:

$$|\psi_{2a}\rangle = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{N-1} b_j \, |0\rangle_A \, (|0\rangle + |1\rangle)_C^{\otimes n} \, |u_j\rangle_I$$

Step (2b) is to perform the controlled-U^{2^k} operation for all non-negative integers $k < n$. This transforms the system's state into:

$$\begin{aligned} |\psi_{2b}\rangle &= \frac{1}{\sqrt{2^n}} \sum_{j=0}^{N-1} b_j \, |0\rangle_A \, [(|0\rangle + e^{2\pi i 2^{n-1}\lambda_j} \, |1\rangle) \\ &\quad (|0\rangle + e^{2\pi i 2^{n-2}\lambda_j} \, |1\rangle) \cdots (|0\rangle + e^{2\pi i 2^0 \lambda_j} \, |1\rangle)]_C \, |u_j\rangle_I \\ &= \frac{1}{\sqrt{2^n}} \sum_{j=0}^{N-1} \sum_{k=0}^{2^n-1} b_j \, |0\rangle_A \, e^{2\pi i \lambda_j k} \, |k\rangle_C \, |u_j\rangle_I \end{aligned}$$

where $|k\rangle$ is introduced to represent all 2^n Fourier basis states, namely the integers between 0 and $2^n - 1$ inclusive, as n-bit binary strings [2]. Finally, in Step (2c), we apply an inverse Quantum Fourier Transform or QFT†, which is a combination of unitary transformations that essentially maps the system's state to:

$$|\psi_2\rangle = |\psi_{2c}\rangle = \sum_{j=0}^{N-1} b_j \, |0\rangle_A \left|\tilde{\lambda}_j\right\rangle_C |u_j\rangle_I \tag{14}$$

where $\left|\tilde{\lambda}_j\right\rangle$ are the binary representations of $|2^n\lambda_j\rangle$ to n bits of precision [2,5]. For example, to store the eigenvalue $\lambda = \frac{1}{2}$ in the clock register using 4 qubits, we would encode the state $|1000\rangle_C$ because $1000_2 = 8_{10} = 2^4 \cdot \lambda \implies \lambda = \frac{1}{2}$. In Qiskit, we implement the phase estimation circuit by creating an `EigsQPE` object, specifying the number of clock qubits we wish to approximate the eigenvalues with, the Hermitian matrix A we will use to perform the Hamiltonian simulation, and the type of transform we want to apply (namely a `Standard IQFT`). We then call `construct_circuit` on this object to build the phase estimation circuit and append it to the existing HHL circuit.

2.2.3 Applying Controlled Rotation

After having stored the eigenvalue estimates of A in the clock qubits, we perform Step (3) which is to apply a rotation on the qubit in the ancilla register conditioned on the clock register. When we perform the controlled $R_y(\tilde{\lambda}_j^{-1})$ operation, we encode the inverse of the eigenvalues into the probability amplitudes of the ancilla qubit [2,6], as given by the system's

state:

$$|\psi_3\rangle = \sum_{j=0}^{N-1} b_j \left(\sqrt{1 - \frac{\gamma^2}{\tilde{\lambda}_j^2}} |0\rangle_A + \frac{\gamma}{\tilde{\lambda}_j} |1\rangle_A \right) \left|\tilde{\lambda}_j\right\rangle_C |u_j\rangle_I \tag{15}$$

where γ is a normalization constant. To program this, we initialize a `LookupRotation` object which allows us to perform the controlled rotation of the ancilla qubit by using a lookup table of rotation angles. We then call the `construct_circuit` method on this object and append the returned circuit to the HHL circuit.

2.2.4 Performing QPE†

It is now clear that the $|1\rangle_A$ subspace, or the $|1\rangle$-subspace of the ancilla register, is now proportional to $\tilde{\lambda}_j^{-1}$. However, the current qubits and registers are entangled. Since the eigenvalues stored are no longer needed, we uncompute the clock register by applying an inverse quantum phase estimation algorithm. By applying QPE† in Step (4) of the HHL algorithm, we are effectively reverting all of the clock qubits back to the state $|0\rangle$, leaving us with the system's overall state:

$$|\psi_4\rangle = \sum_{j=0}^{N-1} b_j \left(\sqrt{1 - \frac{\gamma^2}{\tilde{\lambda}_j^2}} |0\rangle_A + \frac{\gamma}{\tilde{\lambda}_j} |1\rangle_A \right) |0\rangle_C^{\otimes n} |u_j\rangle_I \tag{16}$$

To implement QPE† into our Qiskit circuit, we simply append the circuit we obtain from calling `construct_inverse` on the `EigsQPE` object we previously created.

2.2.5 Obtaining $\vec{x}$ From State Vector

The $|1\rangle_A$ subspace $\sum_{j=0}^{N-1} \gamma \frac{b_j}{\tilde{\lambda}_j} |1\rangle_A |0\rangle_C^{\otimes n} |u_j\rangle_I$ is now in a state that is proportional to our desired solution $\vec{x}$. In order to obtain the solution, we must extract the scaled solution from the system's quantum state vector and modify it accordingly. This is Step (5) of our implementation of the HHL algorithm. We are interested in the probability amplitudes of the basis states given by $\sum_{j=0}^{N-1} |1\rangle_A |0\rangle_C^{\otimes n} |u_j\rangle_I$. These happen to be the first N states in the second half of the statevector, which we can visualize by running the `statevector_simulator` on the HHL circuit. Extracting and normalizing these elements produces the normalized $\hat{x}$. The desired solution vector $\vec{x}$ is given by:

$$\vec{x} \coloneqq e^{-i\theta} \|\vec{x}\| \hat{x} \tag{17}$$

where $e^{-i\theta}$ is some global phase and $\|\vec{x}\|$ denotes the norm of the solution vector. We first introduce a new variable $\vec{b'} = A\hat{x}$. Then, we compute $\|\vec{x}\|$ as follows:

$$\|\vec{x}\| = \frac{\|\vec{b}\|}{\|\vec{b'}\|} \tag{18}$$

Now, let the elements of $\vec{b}$ be $\beta_1, \beta_2, \ldots, \beta_N$ and those of $\vec{b'}$ be $\beta'_1, \beta'_2, \ldots, \beta'_N$. The phase angle θ can be found by calculating:

$$\theta = \frac{1}{m} \sum_{i=1}^{N} \arg\left(\beta_i \circ (\beta'_i)^*\right) \tag{19}$$

where $\circ$ denotes the Hadamard product or element-wise multiplication and the * operator indicates the complex conjugate. Evaluating (17) using (18) and (19), we finally obtain $\vec{x}$, the solution to the $N \times N$ system of equations $A\vec{x} = \vec{b}$.

3 GEPP Algorithm Complexity Analysis

To perform a complexity analysis for the GEPP algorithm, we created a program that produces a random square full rank coefficient matrix A of specified size N and sparsity s (while computing and recording the condition number κ) and a randomly generated constant vector $\vec{b}$ of size N, as outlined in Figure 3.

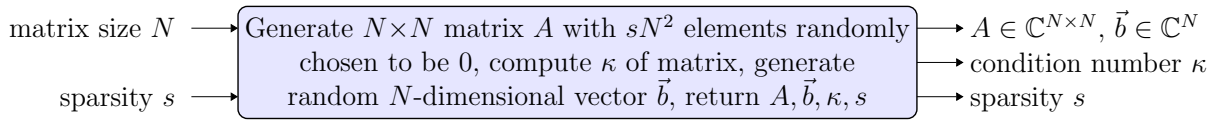

Figure 3: Input-output block diagram of GEPP algorithm data generation program

For each of five different sparsities $s \in \{0.75, 0.80, 0.85, 0.90, 0.95\}$ for $N \in \{32, 64, \ldots, 2048, 4096\}$, we called the data generation program 30 times and recorded the time it took to perform GEPP $\left(A, \vec{b}\right)$ for each data set. The results are displayed in Figure 4. When we plot both N and runtime on a logarithmic scale, the data exhibit a very strong linear relationship with a correlation coefficient of $r > 0.9998$ for all five sparsity sets. Moreover, the five shown lines of best fit are nearly identical with an average slope of 3.0799 and an intercept of -8.5719. A line on this graph with this slope and intercept would take on the form:

$$\begin{aligned} \log\left(T(N)\right) &= 3.0799 \log(N) - 8.5719 \implies \\ T(N) &= \left(1.6800 \times 10^{-9}\right) N^{3.0799} \end{aligned} \tag{20}$$

where $T(N)$ is the algorithm's runtime as a function of input matrix size. Thus, since $T(N) \approx O\left(N^3\right)$, the GEPP algorithm's runtime scales cubically in N. s and κ had no observable impact on the runtime data, indicating that the GEPP algorithm has no dependence on matrix sparsity or condition number.

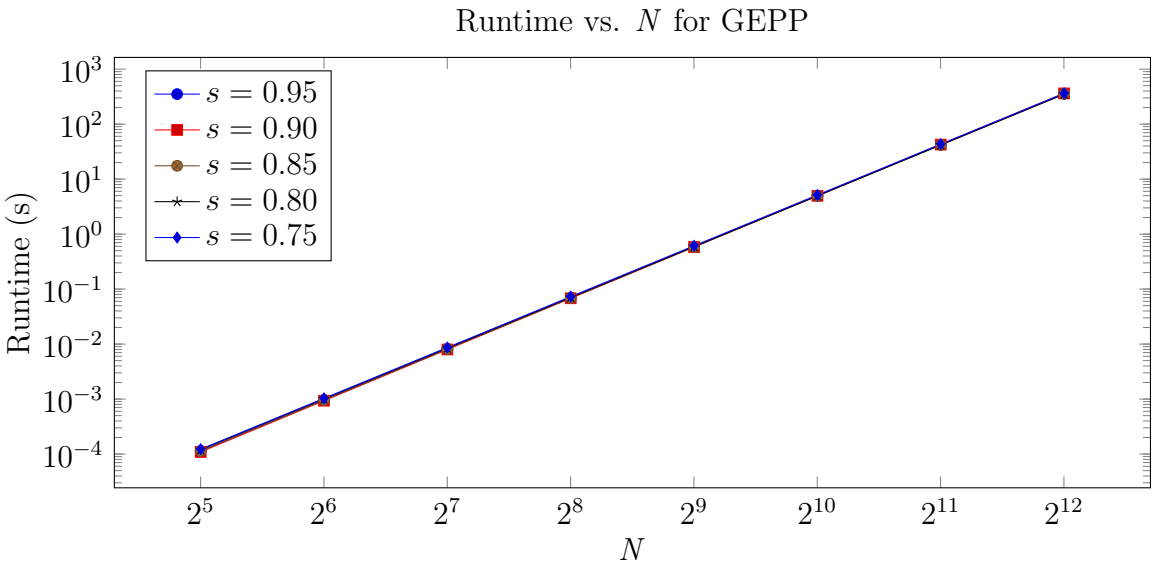

Figure 4: Scaling with N for the GEPP algorithm. Both runtime and N are plotted on a logarithmic scale. Lines of best fit are shown for $s = 0.75, 0.80, 0.85, 0.90, 0.95$. In each case, we averaged 30 runs of the GEPP algorithm.

4 HHL Algorithm Complexity Analysis

We performed a very similar analysis for the HHL algorithm. The HHL data generation program returns a random full rank Hermitian coefficient matrix A of specified size N and sparsity s, its condition number κ, and a random constant vector $\vec{b}$, as shown in Figure 5.

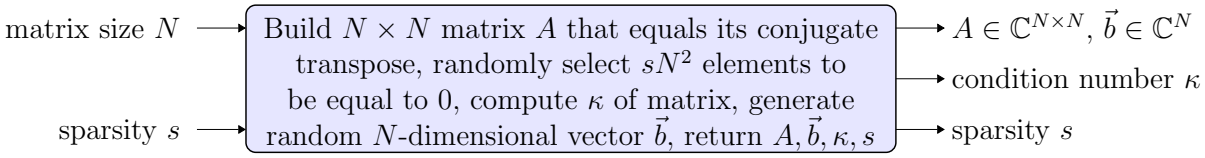

Figure 5: Input-output block diagram of HHL algorithm data generation program

For all trials, we performed phase estimation with $n = 3$ clock register qubits to reduce runtime while maintaining a reasonable degree of accuracy in our eigenvalue estimates. Since the runtime and accuracy of our implementation of the HHL algorithm are extremely sensitive to changes in sparsity, we did not hold sparsity constant as we scaled N, but instead we defined four different sparsity classes which yielded comparable results across different values of N. These sparsities are specified in Table 1. In order for an $N \times N$ system to have a unique solution, it must have full rank and at least N nonzero elements. As such, the maximum sparsity for any given N, represented by Class #1, is $1 - \frac{1}{N}$. For the higher sparsity classes, the sparsity at a given value of N progressively decreases (and by the same amount for each N). For each of the 16 N-s combinations shown in the table, we called the data generation program 15 times. We then recorded the time it took to perform HHL $\left(A, \vec{b}\right)$

	Class #1	Class #2	Class #3	Class #4
$N = 4$	12/16	11/16	10/16	9/16
$N = 8$	56/64	54/64	52/64	50/64
$N = 16$	240/256	232/256	224/256	216/256
$N = 32$	992/1024	960/1024	928/1024	896/1024

Table 1: Table of sparsity classes for the HHL algorithm

as well as the state fidelity between the HHL solution and the exact solution for each trial. The state fidelity F between two quantum states $|\psi_\rho\rangle, |\psi_\sigma\rangle$ is a measure of how close they are to each other. This metric, which could range from 1.0 (which indicates that the two input states are identical) to 0.0 (which indicates that the two input states are completely orthogonal or dissimilar) is given by

$$F\left(|\psi_\rho\rangle, |\psi_\sigma\rangle\right) = |\langle\psi_\rho|\psi_\sigma\rangle|^2 \tag{21}$$

if both $|\psi_\rho\rangle$ and $|\psi_\sigma\rangle$ are pure quantum states. Figure 6 displays the results of scaling with N for each of the four sparsity classes. Plotting the four graphs in Figure 6 on a logarithmic scale effectively linearizes the data. The correlation, slope, intercept, and associated runtime expression $T(N)$ for these graphs are included in Table 2. For all four sparsity classes, the

	Class #1	Class #2	Class #3	Class #4
Correlation	0.9954	0.9972	0.9953	0.9980
Slope	1.6428	1.7381	1.8477	1.9531
Intercept	-0.7411	-1.2061	-1.1404	-1.1852
$T(N)$	$0.1815N^{1.6428}$	$0.0622N^{1.7381}$	$0.0724N^{1.8477}$	$0.0653N^{1.9531}$

Table 2: Table summarizing the results from Figure 6(a-d)

HHL algorithm ran in $O\left(N^m\right)$ with $1.5 < m < 2.0$. Lowering s caused a noticeable increase in m, indicating that lower matrix sparsity resulted in more rapid scaling. In addition to this, there is an evident positive correlation between κ and runtime, as colors closer to red, which are indicative of higher condition numbers, are generally located above colors closer to blue on all graphs. Furthermore, plotting κ against s for each of the four values of N in Figure 7, we observe that κ and s have a clear impact on solution state fidelity. Specifically, since colors closer to red generally appear below those that are closer to blue for a given sparsity, the HHL algorithm yields more accurate solutions for systems with lower values of κ. Solution state fidelity is similarly improved for more sparse matrices. The HHL algorithm achieves optimal solution state fidelity and runtime for matrices that fall under Class #1 (where $s = 1 - \frac{1}{N}$).

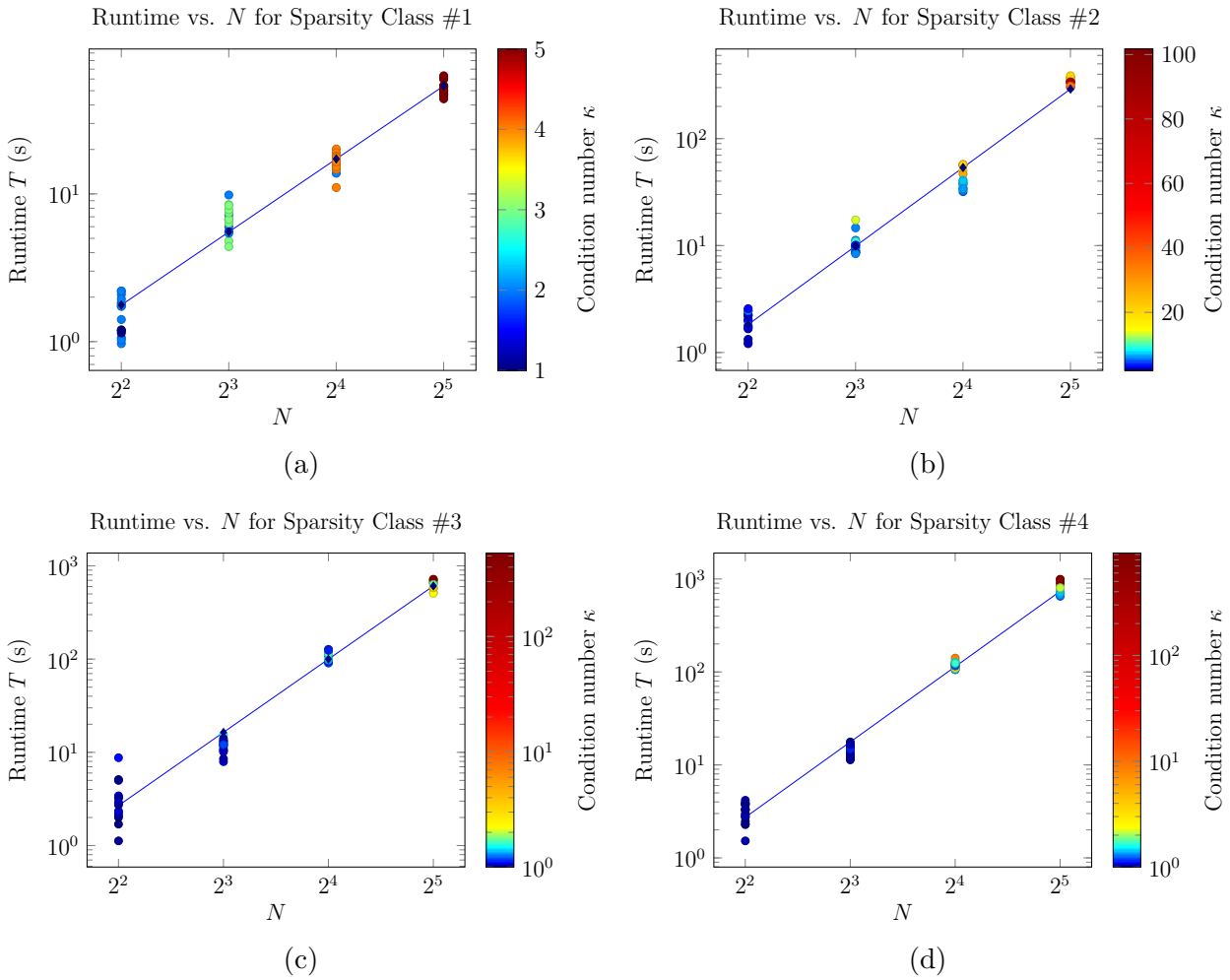

Figure 6: Scaling with N for the HHL algorithm for the four sparsity classes outlined in Table 1. For all graphs, both runtime and N are plotted on a logarithmic scale. Lines of best fit are shown for each graph. For each N and sparsity class, there are 15 data points.

5 Discussion and Future Steps

For all sparsity classes, the HHL algorithm noticeably outperformed the GEPP algorithm, achieving a polynomial speedup in complexity for all four cases. However, the runtime speedup cannot be observed for small values of N, such as the ones we evaluated in our analysis. Therefore, we can compare the experimental runtime equations for the HHL algorithm with that of the GEPP algorithm. Assuming that our observed trends hold for larger values of N, we can estimate the approximate sizes N at which the GEPP algorithm overtakes the HLL algorithm in computational runtime. These calculated crossover values are displayed in Table 3. For systems exceeding these values of N, we would expect our implementation of the HHL algorithm to be faster than the GEPP algorithm. Our complexity analysis of the GEPP algorithm demonstrates that neither κ nor s has any impact on performance, which makes sense as GEPP is a direct method for solving linear systems that depends only on

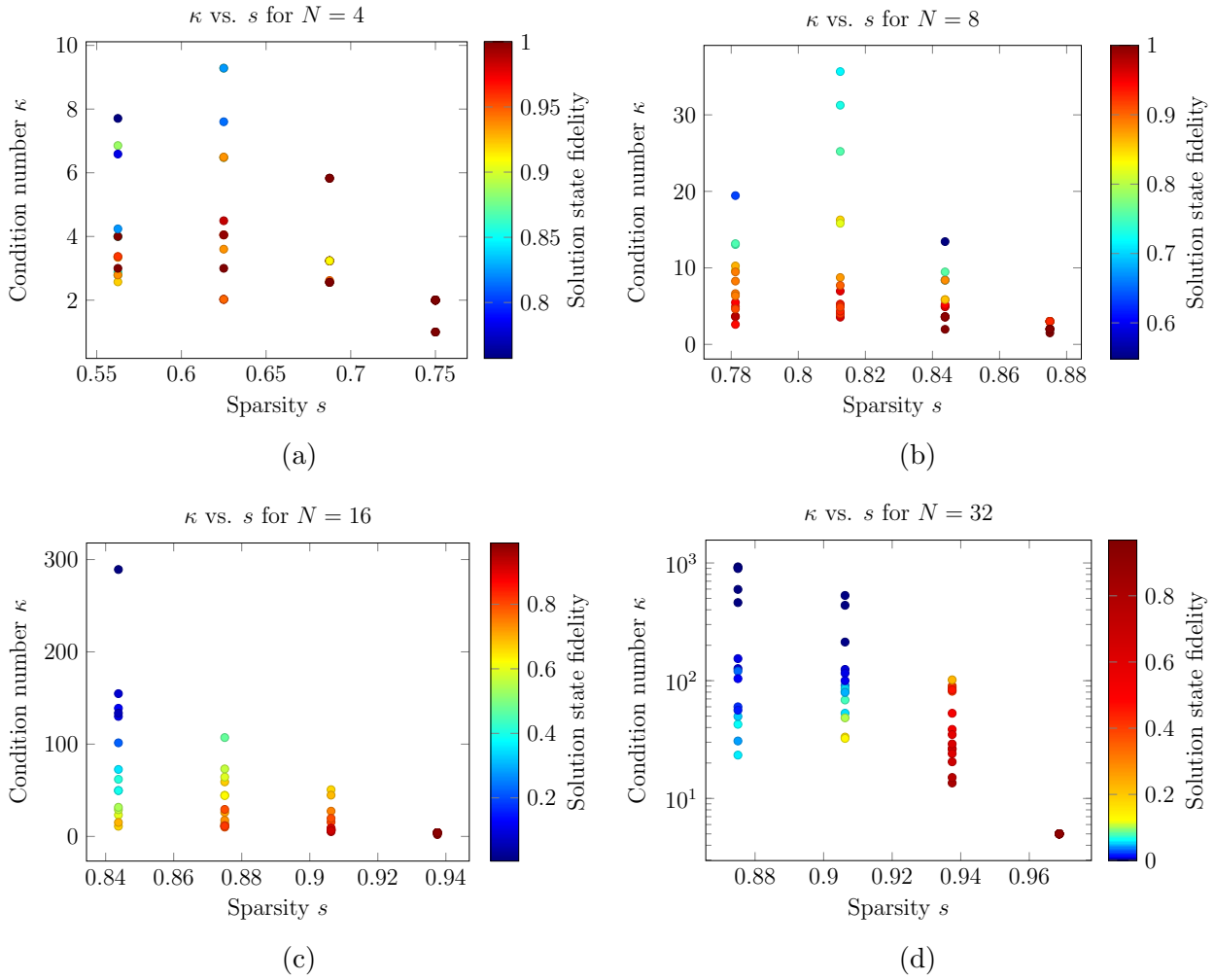

Figure 7: Plotting condition number against sparsity for $N = 4, 8, 16, 32$. Condition number is plotted on a linear scale for (a-c) and on a logarithmic scale for (d). Each graph has four distinct sparsities which correspond to the sparsity classes in Table 1.

matrix size and not the actual values of the matrix elements [4]. In contrast, we found that both κ and s are important factors in the performance of the HHL algorithm. Specifically, we found that higher values of κ and lower values of s resulted in slower run times and less accurate solutions. As κ grows, the coefficient matrix A tends more and more towards a non-invertible matrix, and as such, the solutions become less and less stable. This would make it more difficult for the algorithm to approximate the desired solution. Furthermore, we observed that matrices with lower sparsities typically had higher associated condition numbers.

In conclusion, the HHL algorithm for solving the quantum linear systems problem has practical applications in many areas, particularly the growing field of quantum machine learning. However, implementations of HHL are typically restricted to a particular problem or are subroutines of larger algorithms. The HHL algorithm is also typically only used to

	Class #1	Class #2	Class #3	Class #4
Crossover N	2.8×10^5	3.1×10^5	1.1×10^6	6.3×10^6

Table 3: Table of approximate N values at which the GEPP algorithm matches the HHL algorithm in runtime for the four sparsity classes

obtain samples or functions of the solution vector (such as an expectation value). In this project, we presented the results of using our modified, general-purpose implementation of the algorithm that produces the entire solution. Although this loses part of the quantum advantage by requiring at least $O(N)$ time, it enabled us to evaluate the performance of HHL as a standalone algorithm and qualitatively understand the impact of κ and s on runtime and solution accuracy. In the future, we aim to assess the HHL algorithm's efficiency and accuracy for larger matrix sizes such as $N = 64$ and $N = 128$ to investigate whether or not the trends we observed hold for larger system sizes. We can also model the accuracy and runtime trade-off in the HHL algorithm when we introduce additional clock register qubits. Despite using the algorithm in a somewhat unconventional manner, we have demonstrated that it still outperforms even the most efficient classical matrix inversion algorithm in complexity, and our work shows that we can still experience and appreciate the benefits of quantum computing even while running a general-purpose quantum algorithm on a local classical machine.

6 Acknowledgements

I would like to thank Dr. Jonathan Bennett, my mentor at the North Carolina School of Science and Mathematics, for his thoughtful guidance throughout the entire research process. I would also like to acknowledge Dr. Iman Marvian, an assistant professor in the Physics and Electrical and Computer Engineering departments at Duke University, for introducing me to several quantum algorithms including the HHL algorithm, phase estimation, and the quantum Fourier transform. I would finally like to thank the North Carolina School of Science and Mathematics and the North Carolina School of Science and Mathematics Foundation for providing me with this opportunity and the funding necessary to produce this work.

References

[1] Abraham H., et al. Qiskit: An open-source framework for quantum computing, 2019.

[2] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical Review Letters*, 103(15), 2009.

[3] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, USA, 10th edition, 2011.

[4] William H. Press, William T. Vetterling, Saul A. Teukolsky, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge Univ. Press, 3 edition, 2007.

[5] Philipp Schleich. *How to solve a linear system of equations using a quantum computer*. Jul 2019.

[6] The Qiskit Team. Solving linear systems of equations using hhl, Nov 2020.