Summary of Experiments

Adam Smith

May 30, 2024

Contents

1 Introduction

This document serves to present findings from using the Microsoft Azure Quantum Platform as well as some experimentation with the quantum minimum finding and HHL algorithms.

2 Microsoft Azure Quantum

2.1 Connecting to Backends and Submitting Jobs in Python

Microsoft Azure provides a few ways to connect and send circuits to the available backends. Documentation is available here. The first is to use the `Q#` programming language in VS Code, a language developed by Microsoft for expressing and running quantum algorithms. `Q#` programs can be submitted through a VS Code extension or through a Python API, however, limited resources for `Q#` currently exist and there is little compatibility with other quantum frameworks. For this reason, and to avoid the time commitments required to learn a new programming language, developing quantum circuits with Qiskit (or Cirq) and submitting them through the Azure Quantum Python API was preferred, and the `Q#` method was not explored.

To connect to Azure Quantum through Python, you first need to install the `azure-quantum` Python API. This can be installed using `pip` with the command `pip install azure-quantum[qiskit]` (or `pip install azure-quantum[cirq]` if you intend to develop using Cirq) on Windows or `pip install "azure-quantum[qiskit]"` on Mac OS.

Once installed, you will also need to create a (or use an existing) Quantum Workspace in your Microsoft Azure account to serve as a management interface for all processes run through Azure Quantum. You can then connect to the Azure Quantum provider in Python with the code

```
from azure.quantum import Workspace
from azure.quantum.qiskit import AzureQuantumProvider

workspace = Workspace(
    resource_id=RESOURCE_ID,
    location=WORKSPACE_LOCATION,
)
```

```
provider = AzureQuantumProvider(workspace)
```

where `RESOURCE_ID` and `WORKSPACE_LOCATION` should be replaced with your resource ID and location. Both of these are available on the Quantum Workspace page of the Azure Quantum dashboard.

The available quantum backends can be viewed with the command

```
for backend in provider.backends():
    print(backend.name())
```

and a specific backend (e.g. the IonQ QPU backend) can be selected using the command

```
backend = provider.get_backend("ionq.qpu")
```

Circuits can then be sent to the backend using the standard Qiskit method. For example, the following code sends a circuit creating and measuring a 2-qubit entangled state to the IonQ QPU backend and prints the returned counts:

```
from qiskit import QuantumCircuit, transpile

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.measure_all()

qc_transpiled = transpile(qc, backend, optimization_level=3)
job = backend.run(qc_transpiled, shots=1024)
result = job.result()  # This blocks until result are returned
counts = result.get_counts()
print(counts)
```

2.2 Jobs and Queue Times

When a job is submitted to a backend, it is placed into a queue behind other jobs waiting to run. All jobs submitted to Azure Quantum are available on the Azure Quantum dashboard. To view them, go to your Quantum Workspace and select `Operations -> Job Management`. There you can see data such as the job ID, status, target backend, and the cost estimate

(although I found this to be inaccurate as it always shows $0, even when running on a paid QPU).

The time a job spends in the queue depends on the number of jobs waiting for processing, and can be quite long for some backends. Average queue times can be viewed in the Azure Quantum dashboard by navigating to your Quantum Workspace and selecting `Operations -> Providers` and clicking the drop-down arrow on the left of each provider. As of the time of writing, the queue time for the `ionq.qpu` backend is 605 hours, or about 25 days, making experiments difficult to run on this backend. On the other hand, Rigetti's QPU backend `rigetti.qpu.ankaa-2` states an average queue time of only 1 minute.

I was able to successfully run jobs on the Rigetti and Quantinuum QPU backends with reasonable queue times. For IonQ's QPU, as of the time of writing, the jobs I submitted 10 days ago are still in the queue with a status of "waiting".

2.3 Timeouts and Job Types

The `result` method on the `AzureQuantumJob` object returned by `backend.run()` blocks until the results are computed and returned. However, this method times out after a few hours, and will raise an exception in Python. As such, it is often necessary to retrieve jobs separately at a later time. This can be done using the job ID and the `get_job` method from the `Workspace` class. When a job is submitted, the job ID can be retrieved with the `id` method:

```
job = backend.run(qc)
job_id = job.id()
```

The job itself can be retrieved at a later time through

```
job = workspace.get_job(job_id)
```

However, the job returned by `workspace.get_job` is of type `azure.quantum.job.job.Job`, whereas the job returned by `backend.run` has type `azure.quantum.qiskit.job.AzureQuantumJob`. `Job`'s are not compatible with Qiskit's job methods such as `get_counts` and have a format that is specific to the backend on which they were run, making the `Job` class tedious to work with, particularly if you are running on several backends. Fortunately, one can easily convert a `Job` to an `AzureQuantumJob` with

```
from azure.quantum.qiskit import AzureQuantumJob
```

```
azure_job = workspace.get_job(job_id)
job = AzureQuantumJob(backend=backend, azure_job=azure_job)
```

2.4 Cost Estimation

For the IonQ and Quantinuum backends, estimated costs can easily be obtained using the `backend.estimate_cost` method. For example,

```
from azure.quantum import Workspace
from azure.quantum.qiskit import AzureQuantumProvider
from qiskit import QuantumCircuit, transpile

# Connect to the Azure Quantum platform
workspace = Workspace(
    resource_id=RESOURCE_ID,
    location=WORKSPACE_LOCATION,
)
provider = AzureQuantumProvider(workspace)
backend = provider.get_backend("ionq.qpu")  # Or a Quantinuum backend

# Define a circuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.measure_all()

# Transpile the circuit for the backend
qc_transpiled = transpile(qc, backend, optimization_level=3)

# Get the estimated cost of running the circuit
cost = backend.estimate_cost(qc_transpiled, shots=1024)
print(f"Estimated␣cost:␣{cost.estimated_total}␣{cost.currency_code}")
```

For IonQ devices, the estimated cost is returned in USD. For Quantinuum devices, the estimated cost is returned in units of H-System Quantum Credits (HQC) (or eHQC for the

quantum emulator/simulator). There is no cost for running on IonQ's or Rigetti's quantum simulators.

Rigetti backends do not provide an `estimate_cost` method. Rigetti charges by job execution time on their quantum processor, and there is no added charge per shot, per qubit, or per gate. A price is not provided in the Microsoft Azure Quantum documentation for Rigetti systems, however, I found it to be relatively inexpensive compared to the other providers. For example, a 4-qubit 55-gate circuit with 1000 shots cost about 50 cents to run on the Rigetti system, whereas, according to the cost estimation tool, it would cost 75 HQC (worth more than $500 USD) on a Quantinuum QPU, and about $5 USD on an IonQ QPU.

2.5 Microsoft Resource Estimator

Microsoft provides a backend for resource estimation called the Resource Estimator. The resource estimator is accessible the same way you connect to any other Azure Quantum backend:

```
from azure.quantum import Workspace
from azure.quantum.qiskit import AzureQuantumProvider

# Connect to the Azure Quantum platform
workspace = Workspace(
    resource_id=RESOURCE_ID,
    location=WORKSPACE_LOCATION,
)
provider = AzureQuantumProvider(workspace)
resource_estimator = provider.get_backend("microsoft.estimator")
```

To run the resource estimator, call the run method on a quantum circuit:

```
job = resource_estimator.run(quantum_circuit, shots=1024)
resources = job.result()
```

The returned value is a dictionary containing information about physical and logical qubit counts, runtimes, gate counts, and many other circuit attributes.

The resource estimator provides lots of functionality, for example one can estimate resources of error-correction schemes applied to circuits, however, I did not get the chance

to explore more than its basic functionality. Refer to the documentation for more details.

3 Extending Dijkstra's Algorithm With Quantum Minimum Finding

The section details some exploration into extending Dijkstra's path finding algorithm using quantum minimum finding (QMF). This builds off of work done for the Quantum Launchpad Pilot project "Identifying Cost-Efficient Low-GHG Pathways for Bioenergy and Wood Products". Motivations and impacts of this project will not be discussed here — for this refer to the QLP project — but rather we focus on the some QMF and extended Dijkstra's algorithm implementations and the viability of running them on real quantum hardware.

3.1 Overview of Dijkstra's Algorithm and QMF

Dijkstra's algorithm is an algorithm for finding the shortest path between two nodes in a weighted graph which may represent, for example, road or trail networks. Implementing Dijsktra's algorithm involves repeatedly computing the neighbouring node with the minimum distance from the current node (i.e. finding the minimum of a list). Classically, this has linear time complexity in the size of the list.

Quantum minimum finding aims to apply a variant of Grover's search algorithm to speed up this minimum computation. Grover's algorithm is an algorithm for unstructured search that finds marked elements in a list with time complexity proportional to the square root of the length of the list. The adaptation of Grover's algorithm to search for finding the minimum of a list is called quantum minimum finding (QMF). Two variants of QMF were explored, each of which will be discussed in the following.

3.2 Oracle and Time Complexity Caveat

It is worth noting that Grover's search algorithm, and, hence, QMF, is an oracular algorithm. This means that it relies on a quantum oracle — an unexposed quantum operation that is used as input to the main algorithm. For example, Grover's algorithm relies on a quantum oracle to apply a relative phase to the states which are being searched for. QMF maintains a current guess at the minimum, and similarly uses a quantum oracle to apply a relative phase to states which are lower than the current guess at the minimum, if there are any.

Time complexity for oracular algorithms is usually given in terms of number of oracle calls. This is somewhat in-line with classical algorithms, where, for example, the "oracle" for a classical minimum finding algorithm is a comparison of two elements. However, in reality, quantum oracles may be difficult or computationally intensive to run or define. For the following QMF algorithms, the time complexity to define the oracle is superlinear, and I am unaware of a method to reduce this. As such, as they stand, true time complexity of QMF is worse than the classical case, and further development of the oracle implementation is necessary to beat the classical algorithm.

3.3 Dürr-Høyer QMF

The original QMF algorithm, which I refer to as Dürr-Høyer QMF, was proposed by Dürr and Høyer in 1996 in [1]. This algorithm finds an index i such that the i^{th} element of a list T is its minimum. First, a random index i is chosen as the guess at the index of the minimum, and a Grover-like search algorithm is applied to find an index j such that $T[j] < T[i]$. j then becomes the guess at the index of the minimum in the next iteration. Importantly, this algorithm does not require knowledge of the number of possible indices j each iteration, i.e. the number of indices j such that $T[j] < T[i]$, which is required in the standard Grover's algorithm.

Dürr-Høyer QMF tracks runtime (oracle calls) and terminates when the runtime reaches $22.5\sqrt{N} + 1.4\log_2^2 N$ where N is the length of T. With this runtime, a list of length at least 739 is required before the number of oracle calls in Dürr-Høyer QMF is less than the classical algorithm. For most networks in real-world applications, I expect each node to have fewer neighbours than this, so Dürr-Høyer QMF would likely not be of use in Dijkstra's algorithm.

3.3.1 Implementation

Dürr-Høyer QMF was implemented in Python to examine its applicability and success probability. Using an ideal simulator to run the quantum circuits, Dürr-Høyer QMF was found to return the minimum of a list of random integers of length 10 to 1000 with 100% success probability using only a single shot to simulate circuits. As such, this algorithm is sufficiently accurate to replace the classical minimum finding in Dijkstra's algorithm. Moreover, for lower length lists, the circuit depth remains low enough that reasonable results can be expected from runs on noisy hardware with a low number of shots. However,

due to the large number of circuits required to run Dürr-Høyer QMF and the costs and wait times associated with running these circuits, this algorithm was not run on real quantum hardware.

3.4 Makhanov QMF

A more recently developed QMF algorithm, which I refer to as Makhanov QMF, was developed by Makhanov *et al.* in [2]. This algorithm differs from Dürr-Høyer QMF in that it does not run for a fixed runtime, but rather returns when the guess at the index of the minimum does not improve. This decreases the required number of circuits to run the algorithm, but also reduces the success probability of the algorithm.

Makhanov QMF also requires knowledge of the number of list elements less than the current guess, which is calculated classically in linear time each iteration. Methods for implementing a Grover search when the number of marked elements is not known exist (as is used in Dürr-Høyer QMF), and I suspect these could be incorporated into Makhanov QMF to eliminate this requirement (at the cost of a few more oracle calls).

3.4.1 Implementation

Makhanov QMF was implemented in Python to examine its applicability and success probability. Again, an ideal simulator was used to simulate the quantum circuits. Makhanov QMF was found to have a success probability significantly lower than Dürr-Høyer QMF when using a single shot. To increase this success probability, one can increase the number of shots per circuit, or run the entire algorithm multiple times and accept the most commonly returned response. Figure 1 shows the success probability for a varying number of shots and algorithm repetitions. It is notable that increasing the number of algorithm repetitions significantly increases the success probability of the algorithm. However, increasing the number of shots past 16 has little affect on the success probability. Increasing the length of the lists of values to 100 also seemed to have little affect on the success probabilities, but a systematic decrease in success probability of about 3% was noted when the length was increased to 1000.

To achieve a success probability of at least 99% on 10-element lists, it is necessary to run the algorithm 100 times with 4 shots, or 50 times with 8 shots. Because of this, it is unlikely that this algorithm will provide any benefit over classical minimum finding when incorporated into Dijkstra's algorithm and run for real-world applications.

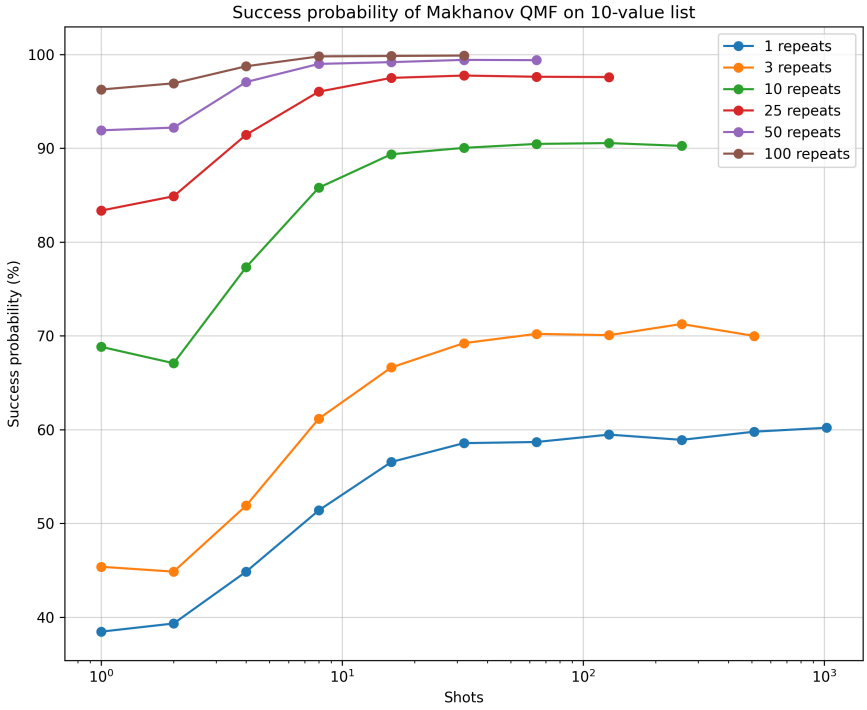

Figure 1: Success probability of Makhanov QMF on a 10-value list for a varying number of shots and algorithm repeats. Note the logarithmic scale on the horizontal axis.

3.4.2 Makhanov QMF on Real Quantum Hardware

To test Makhanov QMF on real quantum hardware, the algorithm was tasked with finding the minimum of the following 10-element list:

```
[78, 15, 46, 52, 5, 14, 66, 85, 3, 6]
```

To yield a high success probability while keeping the number of quantum circuits relatively low, the algorithm was repeated 50 times using 1 shot per circuit. According to figure 1, this gives a roughly 92% probability of success, assuming the circuits can be run with relatively little noise.

Due to the high costs associated with the Quantinuum QPU, this algorithm was submitted to the IonQ and Rigetti QPU backends only.

As of the time of writing, the circuits sent to the IonQ QPU backend are still waiting in the queue, so results are not yet available.

Results from circuits sent to the Rigetti QPU were returned within a couple minutes from submission. The 50 runs of Makhanov QMF sent a total of 74 circuits to the QPU, each with 4 qubits, about 50 gates, and 1 shot. Of these 74 circuits, 6 failed to execute properly and had to be rerun. From this run, the Makhanov QMF algorithm successfully found the minimum of the list (3) and its index (8).

To reduce the gate depth of the circuits submitted to the QPUs, the circuits were transpiled using the options `optimization_level=3` and `approximation_degree=0.95`. The former is the highest level of optimization available through the Qiskit transpiler. The latter further simplifies the circuit by returning a circuit that closely approximates the original circuit, but is simpler. Using this option reduced gate depth by a factor of about 2 to 5 times. In general, setting `approximation_degree` to a value less than 1 can drastically reduce gate depth while still producing circuits that approximate the original circuit to well enough to return correct results.

3.5 Extended Dijkstra's Algorithm

An implementation of Dijkstra's algorithm extended with QMF — which I will refer to as the extended Dijkstra's algorithm or EDA — was created in Python, as well as a purely classical implementation. Dijkstra's algorithm involves repeatedly computing the node with the minimum distance from the currently considered node. In EDA, this classical minimum finding is replaced with the QMF algorithm.

The 10-node graph of figure 2 was used to test EDA. This graph was created by randomly assigning edges to each pair of nodes with 50% probability, and assigning a random edge weight between 1 and 100 to each edge. Both EDA with Dürr-Høyer (using a single shot and repetition) and Makhanov QMF (using 8 shots with 50 repetitions) successfully computed the shortest paths from node 0 to all other nodes in the graph when run on a quantum circuit simulator.

To evaluate costs of running EDA on quantum hardware through Microsoft Azure, the algorithm was set up to estimate the cost of running each circuit submitted during EDA on real quantum hardware, but run the circuits using a simulator such as to not actually incur the costs. Estimated costs and the required number of circuits to run each algorithm on the IonQ and Quantinuum backends are shown in table 1. As is evident from the table,

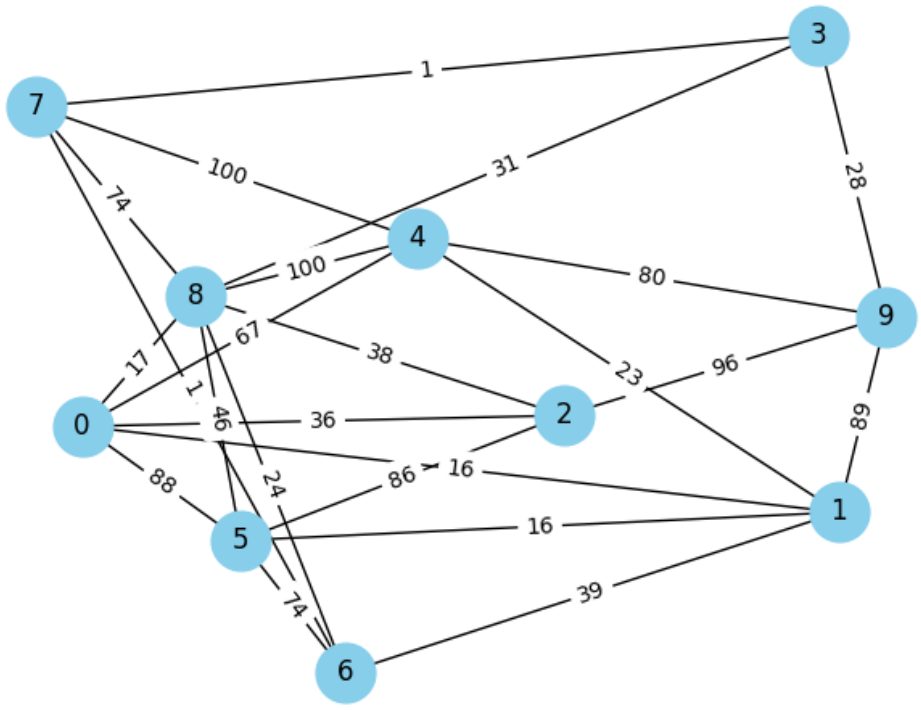

Figure 2: The weighted graph used to test the extended Dijkstra's algorithm.

Algorithm	# Circuits	IonQ Estimated Costs	Quantinuum Estimated Costs
EDA with Dürr-Høyer QMF	1099	$1101 USD	5515 HQC
EDA with Makhanov QMF	1155	$1178 USD	6370 HQC

Table 1: The estimated costs and number of circuits associated with running EDA on the graph of figure 2.

the costs for running EDA on real quantum hardware for are prohibitively high, so EDA was not run on any real quantum backend.

4 Experiments with HHL

The Harrow-Hassidim-Lloyd (HHL) algorithm is a quantum algorithm for solving a linear system of equations. Given a $2^n \times 2^n$ Hermitian matrix A and a corresponding vector

$\mathbf{b} \in \mathbb{R}^{2^n}$, the HHL algorithm solves the equation $A\mathbf{x} = \mathbf{b}$ by preparing the state $|\mathbf{x}\rangle$ whose amplitudes equal the elements of $\mathbf{x}$. Since the quantum state is not directly accessible, HHL limits to efficiently computing expectation values of the form $\langle \mathbf{x}|M|\mathbf{x}\rangle$ for some observable M.

The guide of [3] was used to set up the algorithm in the case where

$$A = \begin{bmatrix} 1 & -\frac{1}{3} \\ -\frac{1}{3} & 1 \end{bmatrix} \tag{1}$$

and

$$\mathbf{b} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2}$$

The intent is to use this example to compare outputs of various quantum hardware providers. The circuit implementing the HHL algorithm is shown in figure 3.

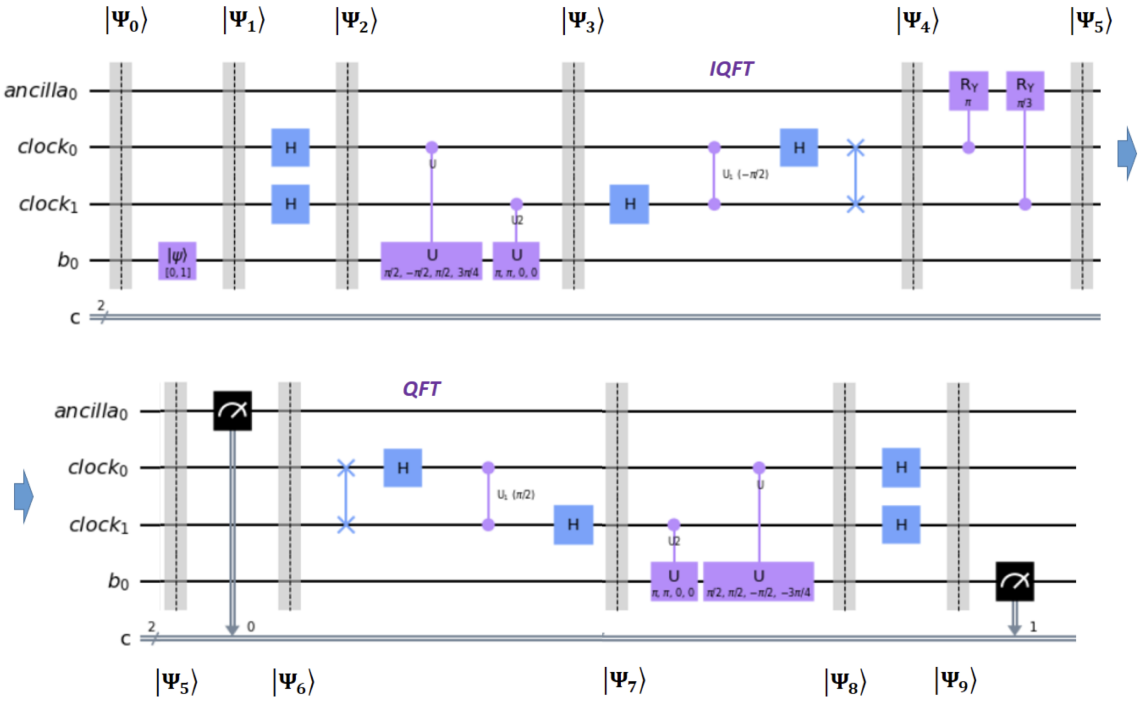

Figure 3: The HHL circuit corresponding to the numerical example [3]

4.1 Running HHL on QPUs

Running this circuit on an ideal simulator, we can obtain a reference distribution of measurement outcomes with which we can compare the distributions obtained from quantum hardware providers. The circuit was run through Azure Quantum on a Quantinuum H1-1 QPU and a Rigetti Ankaa-2 QPU. Results were not able to be obtained from an IonQ QPU due to the large queue times. For comparison purposes, the circuit was also run on the IBM Sherbrooke QPU from IBM — a 127-qubit processor made freely available by IBM. Measurement outcome distributions for runs with 1000 shots (500 for Quantinuum due to high costs) are shown in figure 4. All circuits were transpiled with the Qiskit transpiler with `optimization_level=3`. In an effort to reduce circuit depth and improve results for the IBM and Rigetti QPUs, the circuit was also transpiled and run with `approximation_degree=0.9`.

Results show that the Quantinuum QPU successfully produced results very similar to the ideal simulator, indicating that this processor can effectively run circuits of this size with little error. On the other hand, the Rigetti processor failed to produce results similar to the expected values, and the returned results would not be sufficient to solve the linear system. IBM Sherbrooke landed somewhere in between the Quantinuum and Rigetti processors — the distribution is shaped roughly correctly, but a large amount of noise is evident. Interestingly, reducing circuit complexity by transpiling with `approximation_degree=0.9` improved the results for IBM Sherbrooke, but no improvement is seen when the same is done for the Rigetti QPU.

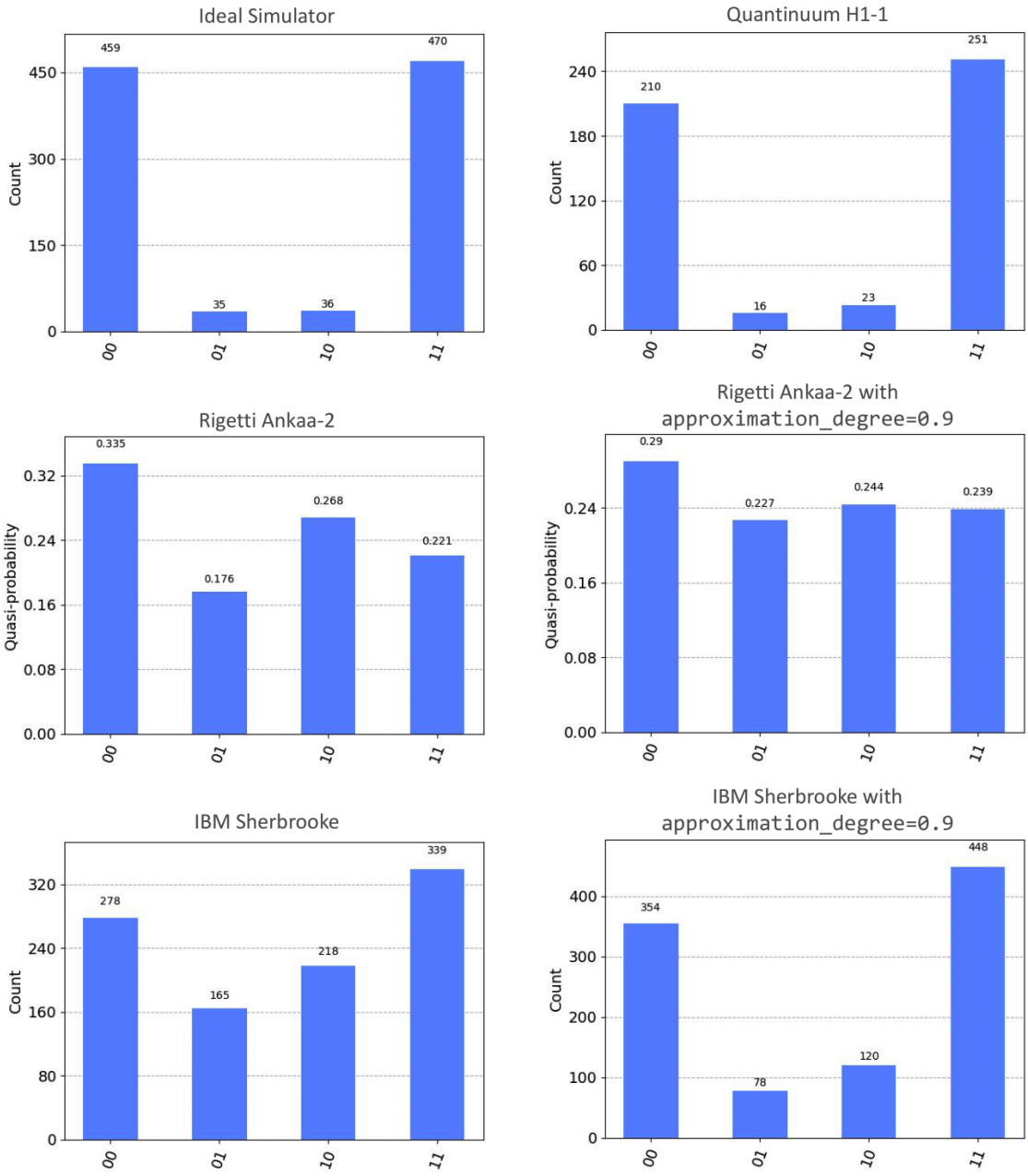

Figure 4: Distributions of measurement outcomes resulting from running the HHL example on various quantum providers.

References

[1] Christoph Durr and Peter Hoyer. *A Quantum Algorithm for Finding the Minimum*. 1999. arXiv: quant-ph/9607014 [quant-ph].

[2] Henry Makhanov et al. *Quantum Computing Applications for Flight Trajectory Optimization*. 2023. arXiv: 2304.14445 [quant-ph].

[3] Anika Zaman, Hector Jose Morrell, and Hiu Yung Wong. "A Step-by-Step HHL Algorithm Walkthrough to Enhance Understanding of Critical Quantum Computing Concepts". In: *IEEE Access* 11 (2023), pp. 77117–77131. ISSN: 2169-3536. DOI: 10.1109/access.2023.3297658. URL: http://dx.doi.org/10.1109/ACCESS.2023.3297658.