Classical-Quantum Systems

An exploration of the client-server model in the context of quantum computing

Justin Clark

ABSTRACT

Quantum computers have the potential solve problems that are currently infeasible with classical computers. However, there are drawbacks to quantum computers. In particular, quantum computers are probabilistic and struggle to perform simple deterministic operations like addition and multiplication which are trivially easy for classical computers. Time will tell the best method for quantum computers to perform such tasks, but in their current infancy stage the best way to interact with a quantum computer is through some sort of hybrid classical-quantum system. This paper will explore the process of interacting with a quantum computer via a client-server model.

QUICK BACKGROUND

This paper can be seen as an extension of previous paper of mine where I present details on quantum computing architectures [2], but it can also be standalone. The quantum computing space is still in its infancy and there are quite limited resources on the subject that are not highly technical and theoretical research papers. However, this is part of the fun of such a new space. The basic idea of quantum computing is as follows: classical information is encoded in bits, typically represented by voltages. It turns out that we can run many types of computations on bits. To extend this idea, it was proposed by Richard Feynman that it should be possible to run computations on information encoded by particles that behave according to the laws of quantum physics. Quantum programs can theoretically perform computations *much* better on problems such as combinatorial optimization and also problems that require the discovery of some sort of periodicity, such as prime factorization [2]. These particles that encode quantum information have come to be known as qubits. A quantum computation starts with some initial state of qubits and the process naturally evolves based on the probabilistic nature of the quantum system until a final result is reached.

INTRODUCTION

With the limits of Moore's Law quickly approaching [1], researchers are working on alternative methods of computation to keep up with the exponential improvement in computing power. There have been several methods suggested, but quantum computing has been one of the most promising technologies. The problem with quantum computing is that it is a foreign concept to a vast majority of computer scientists and only large organizations can afford the infrastructure, necessitating some way for everyday end users to program quantum algorithms.

Quantum programs are currently defined through some sort of high-level abstraction of the quantum world. One of these abstractions is the quantum circuit model [6], which define programs as a series of quantum gates. Similar to how classical gates perform some sort of operation on bits, quantum gates perform specific operations on *qubits*, the quantum counterpart to a bit. The nice thing about quantum gates is that they are really just matrix operations. So, a quantum circuit is simply a sequence of matrix operations on qubits. One constraint to quantum circuits is that all the matrix operations need to be unitary operations [3]. A quantum program is unitary if all the gates could be reversed and the program would still converge to some answer. Note that such a constraint is something that will need to be verified in compiling a quantum program. When we reframe quantum algorithms as series of matrix operations, we can see that understanding quantum physics is not necessary. Having only a deep understanding of linear algebra gives one an opportunity to understand quantum algorithms [4]. Besides the quantum circuit model, we have the QRAM model (discussed in the next section), which has not been widely adopted in these early days of quantum computing but promises to provide a more general framework for programming quantum computers. Whether using the quantum circuit model or the QRAM model, the ability to write quantum programs does not mean that one will have the infrastructure that is needed to interact with quantum computers. This motivates companies to create APIs for end users to learn quantum computing.

Companies such as IBM, Google and Microsoft are all working on developing their own quantum computing capabilities. Part of the development of these technologies includes educating the public about how to write quantum algorithms, which also requires some sort of high-level abstraction so that end users on a classical computer can interact with quantum hardware. As was the case in the early days of classical computing [5], there are many programming languages and libraries being developed in hopes of creating a more user-friendly interaction with a seemingly alien technology [6,7]. The dominant approach to creating an environment in which the public can interact with a quantum computer is through some sort of API. IBM [6] and D-Wave [8] are two of the leading companies in terms of quantum computer API development. While companies take quite different approaches to quantum computers [2], the model they use is fundamentally the same.

In this paper, we will discuss a client-server model that allows an end user (without access to a multi-million dollar quantum computer) to write quantum algorithms and receive results on a classical computer. There are three major steps in this model that will discussed. First, an abstraction such as a library or programming language for the *client* to write a quantum algorithm. Second, there will be some sort of verification and compilation process either at the client or server to ensure that each of the quantum system defines a unitary operation and that the quantum circuit is optimal. Lastly, after the program is sent to the server, the initial state of the specified algorithm must be prepared and scheduled on a quantum computer. It turns out that the initial state preparation is actually one of the bottlenecks in quantum computing since quantum algorithms themselves are quite fast, but energizing particles (what is needed in qubit state preparations) is currently a much slower process. With these three steps, the underpinnings of simple client-server model are complete and we will have a better idea of what such a system might look like. We will skip the step where results from the quantum computer measures the end result of the computation as classical bits and sends it back to the client since that is a trivial problem in networking.

CLIENT SIDE QUANTUM ABSTRACTIONS

As discussed in the previous section, there are two major types of abstractions that allow normal users to interact with a quantum computer. The first of which are external libraries or SDKs that are written for existing programming languages. We will focus primarily on IBMs Qiskit, a quantum computing SDK written for the Python programming language. As one might expect, not everyone has free access to quantum computing resources. Instead, users are required to register for the IBM Quantum Experience [12] and obtain an access token, which is one of the topics discussed in [11] in the context of the FreeBSD operating system. The tokens give users different access rights, depending on the level of authority. For example, a user in the “free tier” will have a token that only allows quantum programs to be simulated on a classical computer.

How does IBMs Qiskit work in the first place? It is mean to allow users to specify the circuit of a quantum algorithm. Figure 1 below shows what one of these circuits might look like (though, in a more visual way than with Qiskit). With Python, it is easy enough to add gate after gate in the quantum system, ultimately leading to a full circuit. If one is familiar with deep learning frameworks such as Tensorflow and Keras [14], the use of Qiskit should seem relatively familiar. An example of the “Hello World!” of quantum computing with Qiskit is shown in the Appendix of [2].

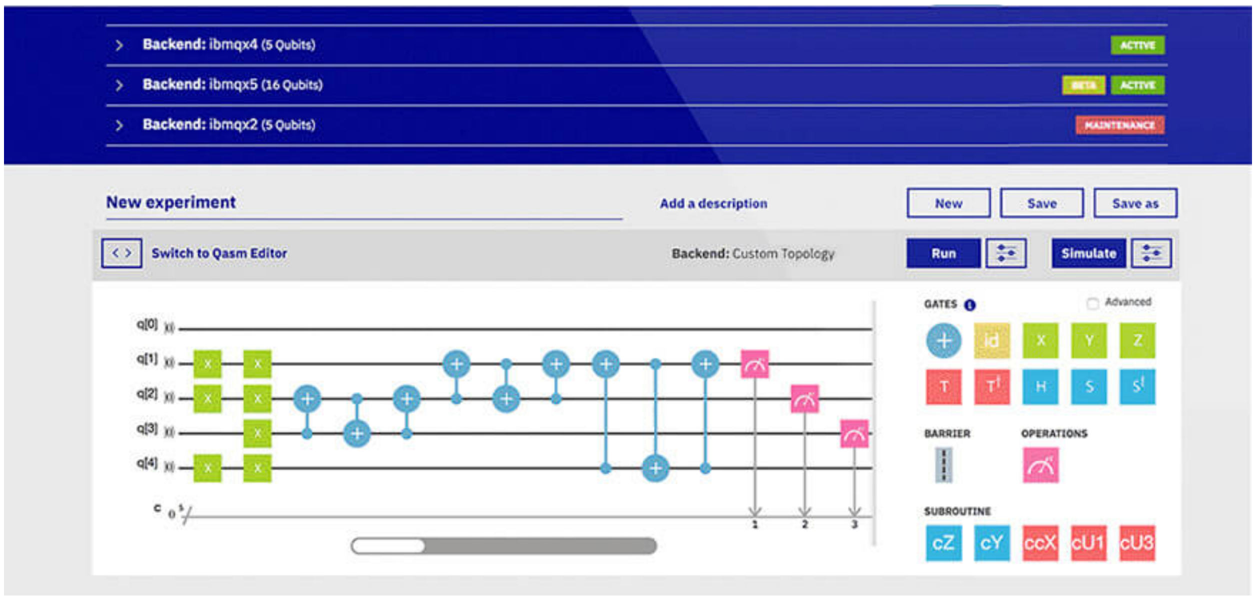


Figure 1: This drag and drop tool by IBM gives users experience creating quantum circuits before moving to actual quantum computations. One can learn how to use gates such as Hadamard gates (fundamental to many quantum algorithms) and readout gates (converting probabilistic qubits to classical bits).

While Qiskit has allowed many users to learn the basics of quantum computing, it is also limited to the IBM hardware. Additionally, designing quantum circuits is not the only paradigm in programming quantum algorithms. One such paradigm is presented in [6], "*Another virtual hardware model, and one which is perhaps even better suited for the interpretation of quantum programming languages, is the QRAM model of Knill [15]. Unlike the quantum circuit model, the QRAM models allows unitary transformations and measurements to be freely interleaved. In the QRAM model, a quantum device is controlled by a universal classical computer. The quantum device contains a large, but finite number of individually addressable quantum bits, much like a RAM memory chip contains a multitude of classical bits.*" With the QRAM model, the future of writing quantum algorithms might actually look more like classical computing where there are many general programming languages that allow programs to run on many types of quantum hardware and can be expressed in different programming paradigms.

There are several programming languages that work with the early versions of the QRAM programming paradigm (as opposed to the circuit model paradigm), some of which are imperative and some of which are functional. One of the most popular imperative languages is *Q Language*. Q Language is an extension of C++ and provides similar functionality to Qiskit (Hadamard gates, quantum Fourier Transforms, measurements, etc.). A popular functional language is cQPL, which is primarily used in research about quantum communication protocols over a network. This language is particularly useful in the context of quantum cryptography. Examples of each of these two programming languages are shown in Exhibit 1 [16].

Exhibit 1: Examples of two popular quantum programming languages that can be used by client-side users to express quantum algorithms.

Q Language

```
Qreg x1(); // 1-qubit quantum register with initial value 0
Qreg x2(2,0); // 2-qubit quantum register with initial value 0
```

cQPL

```
new qbit q := 0; // initialize qubit to the “zero” state
q *= H; // perform the Hadamard operation on the qubit
measure q then { print "Head!" } else { print “Tail :("; };
```

Time will tell what the dominant method of programming a quantum computer will be. These early innovations (Qiskit, Q Language and cQPL) are likely to evolve over the next few decades as research and development introduces more and more users to the quantum computing space. One of the things that will likely remain constant, however, is the need for high-level quantum programming languages to abstract away the tedious particle physics necessary to perform computations. We can think of current quantum computing languages as the quantum counterparts of assembly code. This, of course, is not the most efficient way to program. The same way programmers today use high-level programming languages and libraries to increase the capabilities of code, it is possible that we will see the future of quantum programming have similar characteristics.

In summary, SDKs and programming languages offer a way for users of classical computers to interact with quantum computers without the need for fully understanding the ins and outs of quantum computing architectures. Instead, APIs can be used to send code to a server, which will actually perform computations. However, just as in classical computing, validation of quantum programs are necessary before the computation can actually take place. This necessitates a discussion about quantum verification and compilation.

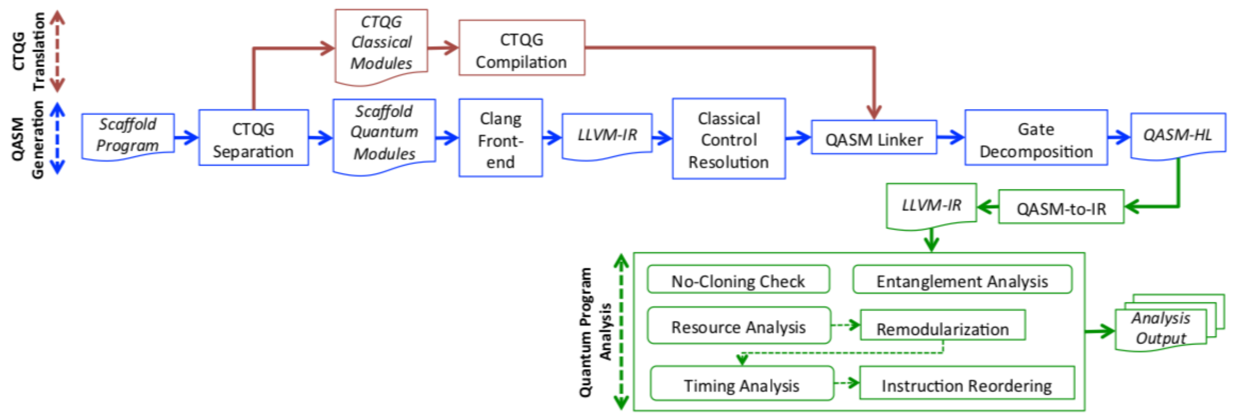
QUANTUM VERIFICATION AND COMPILATION

The difficultly in verifying the correctness of quantum programs is that they are probabilistic in nature and furthermore the programs must contain a series of gates (matrix operations) that are unitary. Determining whether or not a program is unitary is important because the physical world representation of unitary operations is related to the entanglement of the particles used in quantum computations and bad things can happen if a system is *not* unitary. Right now programs are so simple that we can quickly determine whether or not some system is unitary. However, when quantum programs become longer and more complicated, there is no guarantee that the current rules of verification will hold. In other words, there needs to be more formal rules in place for quantum program compilation and verification. It is worth noting that most (if not all) of the current methods to verify quantum programs are actually being done on classical computers.

In [17], the authors discuss a formal quantum verification protocol for programs written in the QPL language discussed previously. The mathematics discussed in [17] are beyond the scope of this paper, but the basic idea is that the authors used an extension of *Hoare Logic* (used in classical program verification) and can show whether or not the series of gates are unitary with some level of confidence. This method is not 100% free of error because the programs are probabilistic. Instead, we can get a degree of certainty that a program is correctly specified. This poses some very obvious issues.. if a compiler told us our C++ program is right with 90% certainty, the program would have a 10% chance of failing at run-time. This isn't *always* catastrophic, but there needs to be more certainty as we move to the future of quantum computing.

There have been several improvements over the extended Hoare Logic method of verifying quantum programs. For example, the authors of [18] have taken an approach using something called the Solovay-Kitaev theorem (again, beyond the scope of this paper) where the idea is to draw tight "nets" (a mathematical object) around the unity of series of gates. The smaller the net, the quicker we can verify the unity of programs. Once we have the nets around the set of possible gates, this method actually provides certainty about the unity of a quantum program, but can be quite computationally expensive.

The two papers discussed so far are solely about verifying the unity of quantum programs. However, this is only one step of the puzzle. When larger quantum computers are feasible (i.e. computers with more qubits), we will be able to run much larger and complex quantum computations. In order to make these large programs optimal, some sort of optimized compiling is necessary beyond the mathematical unity verification. Instead, there needs to be some way to take a high-level program (specified by someone on the client side of this system on a classical computer) and convert that to an optimal unitary sequence of quantum gates. In [19], researchers from Princeton, UCSB and IBM explore some possible methods for doing just this. One interesting proposal of this paper is to use a sort of quantum assembly language as one step in the compilation process, which is the same thing that is done in C and C++ compilers. QASM [20] is referenced as an assembly language that could be used in quantum compilation. In fact, QASM is the assembly language that is used by IBMs



[19] Figure 2: This flow chart shows the basic structure of how the IBM system would go through a compilation process when a quantum program is sent to the server. Note that this whole process takes place on a classical computer which will be fed into a quantum system after analyzing the input program.

Qiskit when programs are sent to the server via API calls. We can see more details about the compilation flow in Figure 2.

When unitary verification and quantum compilation are finished, the program developed by the client can actually be run on a quantum computer. The execution of a quantum algorithm is the last step in the funnel, but by far the most complicated part in the process. In the next section, we will talk about preparing the initial states of quantum systems as well as how the server might schedule quantum programs based on expected difficulty in state preparation.

QUANTUM STATE PREPARATION AND SCHEDULING

For all quantum computations, state preparation is necessary regardless of the hardware and will take place on the server after a quantum program has been verified and compiled. The basic idea behind a so-called quantum “state” is that the particles that encode quantum information (or qubits) will start in some state and evolve to some end state. The end state is the result of the quantum computation. Typically, the starting quantum state is a high-energy state and the ending state is a lower-energy state. So, how exactly do we define the starting state? The answer depends on the quantum computing architecture at hand, but the commonality of all state preparation is that the qubits being used in the quantum computer are energized in certain ways so their stating state of each qubit is either $1 \cdot |0\rangle + 0 \cdot |1\rangle$ or $0 \cdot |0\rangle + 1 \cdot |1\rangle$, which means the bits start in a perfectly known state but evolve to a probabilistic intermediate state, such as $0.424 \cdot |0\rangle + 0.576 \cdot |1\rangle$, and are ultimately measured to be either zero or one at the end of the computation. Unless we talk about specific architectures, this is a superficial level of understanding of quantum state preparation. So, let us briefly discuss Microsoft's approach: topological quantum computing [23]. In [21], it is shown that the order in which qubits are energized is key to a functioning topological quantum computation. Many details of the particle physics behind the argument are skipped here, but the basic idea is that the particles used in quantum information processing exhibit some form of topological structure in terms of spins, lattices, and wave patterns. The order in which qubits should be energized is defined topologically. When qubits are energized in the proper order, qubits can theoretically remain entangled and cohered for a longer period of time which is one of the major challenges facing quantum computing.

The final step we need to cover is scheduling of quantum processes on the server. The interesting part of this scheduling problem is that quantum programs do not run in loops nor are there multiple quantum processes running at the same time. Instead, particles/qubits that encode quantum information pass through some system of gates once and reach a final result before the next process can start. This makes it difficult to use similar scheduling programs as a Unix and Linux systems (ULE and CFS [11]) due to lack of multiprocessing. One major approach to quantum scheduling is discussed in [22] where the author quantifies the difficulty of state preparation. Not all quantum states are equally easy to prepare. For example, some algorithms require more quantum processing than others. In other words, more qubits are required to be entangled and cohered for more complicated problems, leading to much more difficult state preparation. Quantifying this difficulty can actually be used to schedule quantum jobs on the server. For example, programs with lower initialization complexity might have a higher priority and programs with higher initialization complexity might have lower priority (or vice versa). The reason this is a pretty good metric for determining priority is because quantum computations do not take long at all and the bottleneck is actually in the initial state preparation. As always, the goals of the system must be taken into consideration. Until multiprocessing is incorporated into quantum computers, we may continue to see simpler algorithms such as initialization complexity or even FIFO scheduling.

SUMMARY

We have discussed several key components to a possible client-server model for developing on quantum computers. Since we are currently in the early stages of quantum computing, it is likely that we may not see quantum devices at home for quite some time. In the meantime, it will be good for end users to learn tools, such as IBMs Qiskit SDK as well as some emerging quantum programming languages, such as cQPL and Q Language. These high-level abstractions of quantum computing will make it easier to get started in the space. However, it is also important for serious end users to understand how quantum programs are verified and compiled. In this paper, we discussed the key takeaways from a couple of mathematical models for verifying quantum programs as well as a paper that provides a deeper overview of quantum circuit optimization. Lastly, we discussed what happens when a user-defined program is actually sent to the server and the quantum computation starts. The initial state preparation depends on the quantum computing architecture, but is a key element in the process. In fact, the complexity of initializing a state can be an important metric in scheduling quantum programs, ultimately leading to more efficient server-side throughput.

REFERENCES

[1] Kumar, S. (2015). Fundamental Limits to Moore's Law.

[2] https://github.com/justinmclark/papers/blob/master/Final%20Paper%20Quantum%20Architectures.pdf

[3] Schwinger, J. (1960). Unitary operator bases. Proceedings of the national academy of sciences of the United States Of America, 46(4), 570.

[4] Lipton, R. (2014). Quantum algorithms via linear algebra : a primer. Cambridge, Massachusetts: The MIT Press.

[5] Wexelblat, R. L. (Ed.). (2014). *History of programming languages*. Academic Press.

[6] Selinger, P. (2004, April). A brief survey of quantum programming languages. In *International Symposium on Functional and Logic Programming* (pp. 1-6). Springer, Berlin, Heidelberg.

[7] Cross, A. (2018). The IBM Q experience and QISKit open-source quantum computing software. In *APS Meeting Abstracts*.

[8] https://cloud.dwavesys.com/qubist/

[9] Metodi, T. S., Faruque, A. I., & Chong, F. T. (2011). Quantum computing for computer architects. *Synthesis Lectures on Computer Architecture*, *6*(1), 1-203.

[10] http://www.cs.ucsb.edu/~chong/QC/

[11] McKusick, M. K., Neville-Neil, G. V., & Watson, R. N. (2014). *The design and implementation of the FreeBSD operating system*. Pearson Education.

[12] https://quantumexperience.ng.bluemix.net/qx

[13] https://developer.ibm.com/code/2017/05/17/developers-guide-to-quantum-qiskit-sdk/

[14] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (pp. 265-283).

[15] Shor, P. W. (1994, November). Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science* (pp. 124-134). IEEE.

[16] https://www.quantiki.org/wiki/quantum-programming-language

[17] Kakutani, Y. (2009, December). A logic for formal verification of quantum programs. In *Annual Asian Computing Science Conference* (pp. 79-93). Springer, Berlin, Heidelberg.

[18] Zhiyenbayev, Y., Akulin, V. M., & Mandilara, A. (2018). Quantum compiling with diffusive sets of gates. *Physical Review A*, *98*(1), 012325.

[19] Javadi Abhari, A., Patil, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F. T., & Martonosi, M. (2014, May). ScaffCC: a framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers* (p. 1). ACM.

[20] Cross, A. W., Bishop, L. S., Smolin, J. A., & Gambetta, J. M. (2017). Open quantum assembly language. *arXiv preprint arXiv:1707.03429*.

[21] Hamma, A., & Lidar, D. A. (2008). Adiabatic preparation of topological order. *Physical review letters*, *100*(3), 030502.

[22] Girolami, D. (2019). How difficult is it to prepare a quantum state?. *Physical Review Letters*, *122*(1), 010505.

[23] Nayak, C., Simon, S. H., Stern, A., Freedman, M., & Sarma, S. D. (2008). Non-Abelian anyons and topological quantum computation. *Reviews of Modern Physics*, *80*(3), 1083.