In ACM XRDS Crossroads 26:1, Fall 2019.

Towards a Universal Quantum Programming Language

Jens Palsberg

Jun 3, 2019

Quantum computing may be more powerful than classical computing but has a radically different programming model. Current languages are in their infancy; future languages are likely to be different. Now is a great time for language designers and implementers to try new ideas.

Introduction

Until recently, every quantum computer had its own programming language. Are we moving towards a universal quantum programming language? Will a winner emerge among the many languages that flourish at the moment? What are the challenges in making a universal language? The field of quantum programming is expanding rapidly and the time has come to look into the crystal ball. What I see there is that compilers will be a key enabler of a universal language. I am going to support that with three points. First, while quantum algorithms exist, few of them are ready for prime time. Second, today's quantum computers have different gate sets, which is a challenge. Third, optimizing quantum compilers face large-scale combinatorial explosion.

Quantum algorithms exist but few are ready for practical use

Let me begin with one of the drivers of language design, namely algorithms.

Shor's algorithm and Grover's algorithm require too many qubits. The most famous quantum algorithm is Shor's algorithm for factoring integers. This algorithm has the promise to break some forms of cryptography, and it has generated a lot of interest in quantum computing. However, it requires a quantum computer of a capacity that is not expected to be available until many years into the future. Specifically, people have estimated that Shor's algorithm requires more than 10,000 qubits before it can beat factorization on classical (Von Neumann) computers. This is a large number because today the largest quantum computer has fewer than 100 qubits.

Another example of a well-known quantum algorithm is Grover's algorithm for search. People have thought of many application areas for Grover's algorithm, yet those will have to wait because Grover's algorithm requires more than 100,000 qubits. Or, more accurately, Grover's algorithm requires a large number of qubits before it is faster than running a search on classical computers.

In the next five years, we should look to other algorithms than Shor's and Grover's to motivate language design.

The quantum zoo has dozens of algorithms but many are similar. Currently, the Quantum Algorithm Zoo, quantumalgorithmzoo.org, lists 60 algorithms. Most of them have better worst-case complexity than their classical counterparts. However, many of them are similar in nature and few are expected to provide an advantage on near-term quantum computers. The paucity of highly promising quantum algorithms is a conundrum for language designers. Specifically, how can we know what programming style to support before a wide variety of algorithms is available? So far language designers have moved forward with well-known ideas that have worked in other domains. For example, a PyQuil programmer can use Python to build a list that represents a quantum circuit and then give that list to an execution engine. The execution engine can be a quantum computer or a quantum simulator. A Cirq programmer can use Python and an execution engine in a similar manner. In contrast, Q# is a domain-specific language that mixes classical code and quantum code. In this case, the Q# compiler produces a quantum circuit that can be executed on an execution engine.
One could also ask the dual question: how difficult is algorithm design before good programming notation becomes available? So far, this has been a nonissue because algorithm designers can use the language of linear algebra to represent their algorithms. However, as algorithms grow larger and more complicated, a need for abstraction mechanisms is likely to become more pressing. Will those abstraction mechanisms be similar to ones we know already, or will unique aspects of quantum computing call for novel mechanisms?

Hope for quantum advantage lies in optimization, simulation, and machine learning. Researchers have high hopes for these three areas of quantum algorithms. For each one, researchers see the potential for a quantum advantage based on fewer than 10,000 qubits. The most promising area consists of algorithms for optimization. Many optimization problems are NP-complete, which means that optimal algorithms on a classical computer scale poorly. Classical computers can also solve such problems approximately, and so can a quantum computer. For example, the Quantum Approximate Optimization Algorithm (QAOA) is a promising approach to quantum optimization. Recent work suggests that quantum computers may be better than classical computers at solving optimization problems approximately. Such a quantum advantage will kick in at a considerable problem size that people have estimated require at least 3,000 qubits to represent. In 2019, DARPA plans to invest $33 million in finding out whether QAOA or related algorithms can give a quantum advantage on near-term quantum computers.

Two other promising areas are algorithms for simulation and for machine learning. In particular, simulation of quantum chemistry has attracted much attention and is at the heart of what motivated quantum computing in the first place. Similarly, quantum machine learning is an active area with much potential.

Today’s quantum computers have different gate sets

Now let us look at the computers that a universal language will run on.

The world has few quantum computers. Let us first distinguish between quantum annealers and gate-based quantum computers. Quantum annealers such as the ones produced by D-Wave are specialized computers that can run some -- but far from all -- algorithms. In contrast, today’s gate-based quantum computers are Turing-complete. So far, the world has seen few gate-based quantum computers. One estimate from 2018 suggested that four gate-based quantum-computers existed, though since then the number has grown.
A gate-based quantum computer works as follows. The state of the computer is *n* qubits, which we can think of as a vector of 2^n complex numbers. Each step of computation is the application of a matrix (usually called a gate) to some of the state. Each step takes 10–100 nanoseconds on today’s quantum computers. When the computation ends, we can measure the final state. Quantum computing is probabilistic in nature, so typically we will want to repeat the computation a large number of times, such as 10 million. This will give us a probability distribution over the observed final states.

No two quantum computers are alike at the gate level. The current quantum computers tend to be rather different at the hardware level. For example, they may use trapped ion qubits or superconducting qubits, which in turn can be realized in different ways. Additionally, the instruction-set architectures that they expose to software are rather different. They all support gates, but different sets of gates. This means that a machine-level program for one quantum computer usually needs a significant amount of work before it can run on a different quantum computer. This challenge is increased by another factor, which is the physical layout of the qubits. Some gates operate on two qubits, but on some quantum computers, those two qubits must be physically connected. In some cases, a quantum algorithm may want to apply a gate to two unconnected qubits, which means that some swapping of qubits has to happen first. Different quantum computers tend to have different layouts so the needed swapping of data is different for each one.

We should expect more diversity as new forms of qubits emerge. As new quantum computers and new qubits have emerged, the trend is towards more diversity in the instruction sets. This increases the challenge for compilers that both want to optimize code and to target multiple quantum computers. In classical computing, a compiler usually optimizes first and then

target a particular architecture second. However, this may be the wrong order for quantum computing. The reason is that optimized code that is fast on one quantum computer may well be slow on a different quantum computer. In this respect, quantum computers are rather different from classical computers. For a classical computer, the overhead associated with targeting a particular instruction is usually small. Often, the targeting step is mostly a matter of picking the right syntax for each instruction. In contrast, for a quantum computer, a gate that is native to one quantum computer may well take many gates to implement on a different computer.
One idea is to merge the optimization problem and the targeting problem, yet this may create a problem that is too difficult to solve efficiently. If we are going to run a single universal quantum programming language on every quantum computer, the compiler will be a key component.

Optimizing compilers face large-scale combinatorial explosion

Now let us take a closer look at the compilation problem.

Adding a qubit doubles the computational power. On a quantum computer, adding a qubit doubles the computational power, which makes it a precious resource. One of the objectives of a compiler is to do qubits management with the goal of getting as much work as possible out of every qubit. In contrast to classical computing, this management must be done entirely at compile time. One of the challenges is that all quantum computing is reversible, which entails that destructive update of the state of a qubit is impossible. Additionally, a program cannot clone a qubit in an unknown state so making a temporary copy and later restoring is also impossible. Thus, using a qubit for multiple purposes is nontrivial. This is in contrast to registers in classical computers where copying of a register with an unknown value is straightforward.

The main job of compilers is to fight decoherence. We are entering the era of Noisy Intermediate Scale Quantum computers, also known as the NISQ era. Quantum computers in the NISQ era will have up to 1,000 qubits and will be able to execute up to 100 computation steps. Each such step takes between 10 and 100 nanoseconds on current quantum computers so an entire computation take less than 10 microseconds. What happens after 100 computation steps? The answer is that the quantumness goes away, which is a phenomenon known as decoherence. At the hardware level, a quantum computer is a delicate and brittle device that operates at a temperature below 0.1 Kelvin to maintain quantumness. However, at some point, the quantumness will turn classical and the ability to do quantum computing will be lost. A quantum compiler can fight decoherence by optimizing the number of computation steps.

The space of optimizations is vast and hard to navigate. Many recent papers on quantum compilers demonstrate considerable interest in the area. Given that gate-based quantum computers came into existence fairly recently, the papers that report on experiments are particularly exciting. Papers tend to focus on minimizing circuit depth, minimizing swaps of qubits, and minimizing the count of particular gates. They use a wide variety of abstractions to model minimization problems.

A survey from 2018 found just three open-source compilers for quantum languages. Other compilers may be available as executables, but researchers are unable to enhance and extend them. Researchers have shown that the problem of minimizing circuit depth for a given qubit layout is NP-complete [1]. However, if the qubit layout is a planar graph -- a graph that can be drawn such that no edges cross each other -- the complexity of the minimization problem remains open. One approach to practical optimization of a quantum circuit is to phrase the problem as a constraint optimization problem and then apply a planning algorithm. This is likely to produce a good solution within a bound on compilation time [2].

An alternative to optimizing the entire circuit is to focus on minimizing the swaps that may be needed before we can apply a multiple-qubits operation. Researchers have presented various approaches that include formulating the problem as a satisfiability problem and using a SAT-solver [3], or formulating the problem via use of a dependency graph [4]. Swaps are typically implemented via the use of three so-called CNOT gates, which are two-qubits gates. Researchers have studied how to decrease the count of CNOT gates. Some of the approaches that have been considered are to formulate the problem as a satisfiability problem [5], and to apply matrix identities that help simplify a given circuit [6]. A recent approach had success with aggregating qubit operations into 10-qubits units and then scheduling and customizing circuits that implement such units [7].

Overall, an important task for a compiler is to minimize the number of computation steps, while preferably targeting multiple quantum computers.

Conclusion

Today a language designer has few algorithms for driving the language design, many different targets on which a language can run, and a huge combinatorial problem that a compiler must solve. If we are going to get a universal quantum language, most likely it has yet to be invented. Now is a great time for language designers and implementers to try new ideas.

Bio: *Jens Palsberg is a professor and a former department chair of Computer Science at University of California, Los Angeles (UCLA). He is the Chair of SIGPLAN, a former editor-in-chief of TOPLAS, and a former PC chair and a former general chair of POPL. In 2012 he received the SIGPLAN Distinguished Service Award.*

References

[1] Botea, A. et al. On the complexity of quantum circuit compilation. *In Proceedings of Symposium on Combinatorial Search (SOCS'18)*, 2018.
[2] Venturelli, D. et al. Quantum circuit compilation: An emerging application for automated reasoning. *In Proceedings of The Scheduling and Planning Applications woRKshop (SPARK'19*), 2019.
[3] Hattori, W. and Yamashita, S. Quantum circuit optimization by changing the gate order for 2d nearest neighbor architectures. *In International Conference on Reversible Computation (RC'18)*. Springer, Cham, 2018, 228-243.
[4] Itoko, T. et al. Quantum circuit compilers using gate commutation rules. I*n Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'19)*. ACM, New York, 2019, 191-196.
[5] Meuli, G. et al. SAT-based {cnot, t} quantum circuit synthesis. *In International Conference on Reversible Computation (RC'18)*. Springer, Cham, 2018, 175-188.
[6] Nam, Y. S. et al. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information* 4, 1 (2018), 23.
[7] Shi, Y. et al. Optimized compilation of aggregated instructions for realistic quantum computers. *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, New York, 2019, 1031-1044.