Introduction to Quantum Computing

A guide to solving intractable problems simply

Brad Huntting
David Mertz

In the next few decades, quantum computers are likely to move out of science fiction and research labs (largely at IBM) and into practical applications. A class of problems surrounding complex combinatorics that plague deterministic computers can be solved efficiently on Quantum Computers (QCs). This article, which builds on a basic knowledge of the mathematics of vectors, gives an introduction to quantum computing. For illustration, examples use qcl (quantum computing language), a free programming language for quantum computers distributed under the GNU General Public License. qcl allows developers to simulate and examine a "virtual" quantum computer. The authors, Brad Huntting and David Mertz, are experts in mathematics and veteran programmers. David Mertz is a frequent contributor to developerWorks.

Alan Turing invented the programmable computer in 1936 (see Related topics) as a thought experiment to show that certain mathematical problems were not computable. Implicit in his argument was the idea that a computer, armed with sufficient resources, is capable of realizing any reasonable algorithm.

Since that time, the computer industry has not only managed to build programmable computing machines, they’ve also outdone themselves by doubling the capabilities every eighteen months or so. Despite these frenetic advances in computer technology, modern computers are still unable to make significant dents in hard problems. Problems that require exponential resources (compared to the size of the problem itself), remain as intractable today as they were in 1936.

In 1982 Richard Feynman suggested that the venerable Turing machine might not be as powerful as people thought. Feynman was trying to simulate the interaction of N particles with quantum mechanics. Try as he might, he was unable to find a general solution without using exponential resources.

Yet somehow, nature is able to simulate this mathematical problem using only N particles. The conclusion was inescapable: nature is capable of building a fundamentally superior computing device, and that suggests that the Turing machine had not been the all-purpose computer people had thought.
Visualizing a quantum computing problem

QC algorithms involve thinking in terms of probabilistic factors, a conceptual change for current programmers. In some ways, this is like the conceptual shift involved in using OOP, or functional programming, or multi-threading, for the first time. In another sense the shift is more fundamental since it opens up a whole new class of (probably) non-equivalent problems.

Let's imagine the following problem. We need to find a path through a complicated maze. Every time we follow one path, we soon come across new branches. Even knowing there is some path out, it is easy to get lost. A well-known algorithm, so to speak, for walking a maze, is the right hand rule -- follow the right hand wall until you are out (including around dead-ends). This may not be a very short path, but at least you will not repeat the same corridors. In computer terms, this rule is also known as recursive tree descent.

Now let's imagine another solution. Stand at the entrance to the maze, and release a sufficient quantity of colored gas to fill every corridor of the maze simultaneously. Have a collaborator stand at the exit. When she sees a whiff of colored gas come out, she simply asks those gas particles what path they traveled. The first particle she queries will most likely have traveled the shortest possible path through the maze.

Naturally, gas particles are not entirely wont to tell us about their travels. But QCs act in a manner very similar to our scenario. Namely, they fill the whole problem space, and then only bother asking for the correct solution (leaving all the dead-ends out of the answer space).

The qcl quantum computer simulator

Simulating a quantum computer on a traditional classical computer is a hard problem. The resources required increase exponentially with the amount of quantum memory under simulation, to the point at which simulating a QC with even a few dozen quantum bits (qubits) is well beyond the capability of any computer made today.

qcl only simulates very small quantum computers, but fortunately it's just powerful enough to demonstrate the concept behind some useful QC algorithms.

Like the supercomputers of yesteryear, the first of tomorrow's QCs will probably consist of some exotic hardware at the core which stores and manipulates the quantum state machine. Surrounding this will be the life support hardware that sustains it and presents the user with a reasonable programming environment. qcl simulates such an environment by providing a classical program structure with quantum data types and special functions to perform operations on them.

Let's start with some familiar operations from classical computing using qcl. Since qcl is an interactive interpreter with a syntax vaguely similar to C, we can just fire it up and start entering commands into it. To make our examples more readable we'll restrict the number of quantum bits under simulation to 5.
Listed 1. Initializing qcl and dumping a qubit

```
$ qcl --bits=5
[0/8] 1 |00000>
qcl> qureg a[1];
qcl> dump a
: SPECTRUM a: |....0>
1 |0>
```

Here we have allocated a 1 qubit (Boolean) variable from the qcl quantum heap. The quantum state of the machine, \( |00000\rangle \), is initialized to all zeros. The \( |\rangle \) characters signify that this is a quantum state (sometimes called a ket), while the string of 5 0's (one for each bit in the quantum heap) form the label for the state. This is known as the Dirac notation (a.k.a. bra-ket) for quantum mechanics. Its main advantage over traditional mathematical notation for linear algebra is that it's much easier to type.

"qureg a[1]" allocates a one bit variable \( a \) from the quantum heap. The \( \text{dump a} \) command gives us some information about \( a \). The SPECTRUM line shows us where the qubits for \( a \) have been allocated in the quantum heap; in this case the 0-bit of \( a \) is the rightmost qubit in the heap. The next line tells us that, were we to measure \( a \), we would see "0" with probability "1".

Of course the ability to peek at quantum memory is only possible because qcl is a simulator. Real quantum bits can't be observed without irreversibly altering their values. More on this later.

Many of the primitive quantum operators provided by qcl are familiar from classical computing. For instance, the \( \text{Not()} \) function flips the value of a bit.

Listed 2. A Boolean function on a qubit

```
qcl> Not(a);
[2/8] 1 |00001>
```

\( \text{Not()} \) applied again to the same qubit will undo the effect of the first which is exactly what we would expect from classical computing.

The \( \text{CNot}(x,y) \) operator tests the value of \( y \) and if it is "1", it flips the value of \( x \). This is equivalent to the statement \( x^=y \) in C.

Listed 3. Some more qubit operators (CNot)

```
qcl> qureg b[2];
qcl> Not(b[1]);
[3/8] 1 |00100>
qcl> CNot(b[0], b[1]);
[3/8] 1 |00110>
qcl> dump b[0];
: SPECTRUM b[0]: |...0.>
1 |1>
qcl> dump b[1];
: SPECTRUM b[1]: |...0.>
1 |1>
```
The `CNot()` operator, like the `Not()` operator is its own inverse. Apply it a second time and it reverses the effect of the first, leaving you in the same state as when you started.

This idea of reversibility is critical to understanding quantum computing. Theoretical physics tells us that every operation on quantum bits (except for measurement) must be undoable. We must always keep enough information to work any operation backwards. This means that operations like assignment (`x=y`), AND (`x&=y`), and OR (`x|=y`)—which we take for granted in classical computing—have to be modified for use in QC. Fortunately, there's a straightforward formula for converting irreversible classical operations into quantum operations.

First we never overwrite a quantum bit except to initialize it to "0". So where we would have done an assignment (`x=y`), we instead initialize the target (`x=0`) and use exclusive or (`x^=y`) as in the example above.

**Listing 4. Reversible simulation of Boolean AND**

```
nomadic$ qcl --bits=5
[0/8] 1 |00000>
qucl> qureg c[3];
qucl> Not(c[1]);
[3/8] 1 |00010>
qucl> Not(c[2]);
[3/8] 1 |00100>
qucl> dump c :
    : SPECTRUM c: |..210>
    1 |110>
qucl> CNot(c[0], c[1] & c[2]);
[3/8] 1 |00101>
qucl> dump c :
    : SPECTRUM c: |..210>
    1 |111>
```

The `CNot(x, y & z)` will flip the value of `x` if `y` and `z` are "1". So if `x` is initialized to "0" before we start, this is effectively the same thing as calculating `y&z` and storing the value in `x`. It's a subtle distinction, but one that is critical in quantum computing.

Now let's look at some operations that have no classical analogues. The most striking, and at the same time one of the most useful, is the Hadamard function, appropriately labeled `Mix()` by qcl. `Mix()` takes a computational basis state like `|0>` or `|1>` and turns it into a quantum superposition. Here's a one qubit example:

**Listing 5. Superposing states with Mix()**

```
[0/8] 1 |00000>
qucl> qureg a[1];
qucl> dump a;
    : SPECTRUM a: |....0>
    1 |0>
qucl> Mix(a);
[1/8] 0.707107 |00000> + 0.707107 |00001>
qucl> dump a;
    : SPECTRUM a: |....0>
    0.5 |0> + 0.5 |1>
```
In this example we exploited the quantum mechanics principle of superposition. According to the principle, if we were to measure a, we would see "0" or "1" with an equal probability of 0.5 (0.707107).

If you've never been exposed to this concept of superposition before it can be a little confusing. Quantum mechanics tells us that small particles, such as electrons, can be in two places at once. Similarly a quantum bit can have two different values at the same time. The key to understanding this all is vector arithmetic.

Unlike a classical computer where the state of the machine is merely a single string of ones and zeros, the state of a QC is a vector with components for every possible string of ones and zeros. To put it another way, the strings of ones and zeros form the basis for a vector space where our machine state lives. We can write down the state of a QC by writing out a sum like so:

Listing 6. The vector state of a quantum computer

\[ a|X> + b|Y> + \ldots \]

Where \( X, Y, \) etc are strings of ones and zeros, and \( a, b, \) etc are the amplitudes for the respective components \( X, Y, \) etc. The \( |X> \) notation is just the way physicists denote "a vector (or state) called \( X\)."

The \texttt{Mix()} operator (Hadamard operator) when applied to a bit in the \( |0> \) state will transform the state into \( \text{sqrt}(0.5)(|0>+|1>) \) as in the example above. But if we apply \texttt{Mix()} to a bit that's in the \( |1> \) state we get \( \text{sqrt}(0.5)(|0>-|1>) \). So if we apply \texttt{Mix()} twice to any qubit (in any state) we get back to where we started. In other words, \texttt{Mix()} is it's own inverse.

If we have two qubits \( a \) and \( b \) (initialized to zero) and we perform a sequence of quantum operations on \( a \) and then do the same to \( b \), we would expect to wind up with \( a \) and \( b \) having the same value, and we do.

Listing 7. Independent superposed qubits

```
qcl> qureg a[1];
qcl> Not(a);
[1/8]  1 |00001>
qcl> Mix(a);
[1/8] 0.707107 |00000> + -0.707107 |00001>
qcl> qureg b[1];
qcl> Not(b);
[2/8] 0.707107 |00010> + -0.707107 |00011>
qcl> Mix(b);
[2/8] 0.5 |00000> + -0.5 |00010> + -0.5 |00001> + 0.5 |00011>
qcl> dump a
  : SPECTRUM a: |....0>
  0.5 |0> + 0.5 |1>
qcl> dump b
  : SPECTRUM b: |...0.>
  0.5 |0> + 0.5 |1>
```
In this example, a and b are completely independent. If we measure one it should have no effect on the other.

### Listing 8. Measurement-independent qubits

```qcl
qcl> measure a;
[2/8] -0.707107 |00001> + 0.707107 |00011>
qcl> dump b
: SPECTRUM b: |...0.>
0.5 |0> + 0.5 |1>
```

As expected, the spectrum of b was unchanged by measuring a.

If the operations were more complicated than a simple Not();Mix(), we might be tempted to perform them only once on a and then copy the value from a to b. OK, we can't really copy (because it's not a reversible operation), but we can initialize b to zero and CNot(b,a) which accomplishes the same goal.

Alas, this doesn't do what we would expect. But let's just try it and see what we do get:

### Listing 9. Attempting a qubit-copy operation

```qcl
qcl> qureg a[1];
qcl> Not(a);
[1/8] 1 |00001>
qcl> Mix(a);
[3/8] 0.707107 |00000> + -0.707107 |00001>
qcl> qureg b[1];
qcl> CNot(b,a);
[2/8] 0.707107 |00000> + -0.707107 |00011>
qcl> dump a;
: SPECTRUM a: |....0>
0.5 |0> + 0.5 |1>
qcl> dump b;
: SPECTRUM b: |...0.>
0.5 |0> + 0.5 |1>
```

The spectrum of a and b look correct. And indeed if we were to measure just a or b we would get the same result as above. The difference lies in what happens when we measure both a and b.

Keep in mind that the outcome of a measurement is random, so if you're repeating this experiment, your mileage may vary.

### Listing 10. Measurement collapsing qubit superposition

```qcl
qcl> measure a;
[2/8] -1 |00011>
qcl> dump b
: SPECTRUM b: |...0.>
1 |1>
```

By measuring a, we collapsed the superposition of b. This is because a and b were entangled in what physicists call an EPR pair, after Einstein, Podolsky, and Rosen, all of whom used this in
an attempt to show that quantum mechanics was an incomplete theory. John Bell, however, later demonstrated entanglement in real particles by experimental refutation of the Bell Inequality (which formalized the EPR thought experiment).

What happens when you try to copy one quantum variable onto another? You wind up with entanglement rather than a real copy.

Deutches problem
Suppose we are given a function that takes a one bit argument and returns one bit. And to keep things on the up and up, let's require that this be a pseudo-classical function. If we hand it a classical bit (0 or 1) as an argument, it will return a classical bit.

There are exactly 4 possible functions that fit this requirement.

Listing 11. All four possible Boolean unary functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f(x) \rightarrow 0)</td>
<td>constant zero result</td>
</tr>
<tr>
<td>(f(x) \rightarrow 1)</td>
<td>constant one result</td>
</tr>
<tr>
<td>(f(x) \rightarrow x)</td>
<td>identity function</td>
</tr>
<tr>
<td>(f(x) \rightarrow \neg x)</td>
<td>boolean negation</td>
</tr>
</tbody>
</table>

The first two are constant, meaning they output the same value regardless of input. The second two are balanced meaning the output is 0 half the time and 1 half the time. Classically there's no way to determine if \(f()\) is constant or balanced without evaluating the function twice.

Deutches problem asks us to determine whether \(f()\) is constant or balanced by evaluating \(f()\) only once. Here's how it works.

First, we have to construct a pseudo-classic operator in qcl that evaluates \(f(x)\). To do this we'll define a qfunct with arguments for input and output. For example:

Listing 12. Defining a quantum function in qcl

```qcl
qufunct F(qureg out, quconst in) {
    CNot(out, in);
    Not(out);
    print "f(x) = ~x";
}
```

If out is initialized to "0", calling this function will change out to \(f(x) = \neg x\). You can comment out either the \(CNot()\) or \(Not()\) lines to get one of the other three possible functions. After we put the above code snippet in a file called f_def.qcl we can test \(F()\) to make sure it does what we want:

Listing 13. Interactively importing and testing \(F()\)

```qcl
qcl> include "f_def.qcl"
qcl> qureg in[1];
```
Now let's reset the quantum memory, and run Deutche's algorithm. It works by first putting the in and out bits into a superposition of four basis states.

**Listing 14. Deutche's algorithm (line numbers added)**

```qcl
(01) qcl> reset;
(02) qcl> int result;
(03) qcl> Not(out);
(04) [2/8] 1 |00010>
(05) qcl> Mix(out);
(06) [2/8] 0.707107 |00000> + -0.707107 |00010>
(07) qcl> Mix(in);
(08) [2/8] 0.5 |00000> + 0.5 |00001> + -0.5 |00010> + -0.5 |00011>
(09) qcl> F(out,in);
(10) : f(x)= ~x
(11) [2/8] 0.5 |00010> + 0.5 |00001> + -0.5 |00000> + -0.5 |00011>
(12) qcl> Mix(out);
(13) [2/8] 0.707107 |00011> + -0.707107 |00001>
(14) qcl> Mix(out);
(15) [2/8] -1 |00011>
(16) qcl> measure in, result;
(17) [2/8] -1 |00011>
(18) qcl> if (result == 0) { print "constant"; } else { print "balanced"; }
(19) : balanced
```

With lines 1-7 we put the in/out bits into a superposition of four base states with positive amplitudes +0.5 for states where "out=0" and negative amplitudes -0.5 where "out=1." Note that even though we have four non-zero amplitudes, the sum of the squares of the absolute values of the amplitudes always adds up to one.

At line 9 we run the quantum function $F()$ which XORs the value of $f(x)$ into the out qubit. The function $F()$ is pseudo-classical, meaning it swaps basis vectors around without changing any amplitudes. So after applying $F()$ we still have two amplitudes with value "+0.5" and two with the value "-0.5."

By applying the $F()$ function to a superposition state, we have effectively applied $F()$ to all four basis states in one fell swoop. This is what's called "quantum parallelism" and it's a key element of
QC. Our simulator will, of course, have to apply \( F() \) to each of the basis states in turn, but a real QC would apply \( F() \) to the combined state as a single operation.

The \( \text{Mix()} \) functions at lines 14 and 16 flip the machine state out of a superposition and back into a computational base state (\( |00011> \)). If we had not run \( F() \) at line 9, this would have brought us back to the state we had at line 4 (this is because \( \text{Mix()} \) is its own inverse). But because we swapped amplitudes with \( F() \), undoing the superposition puts us into a different state than where we were at line 9. Specifically the in qubit is now set to "1" rather than "0".

It’s also instructive to note that the amplitude of -1 in line 15 is unimportant. A quantum state is a vector whose overall length is of no interest to us (as long as it's not zero). Only the direction of the vector, the ratios between the component amplitudes, is important. By keeping quantum states as unit vectors, the transformations are all unitary. Not only does this make the theoretical math a lot easier, it keeps the errors incurred doing numerical calculations on classical computers from snowballing out of control.

**Controlled phase transformation**

The original goal of quantum computing was to simulate the behavior of arbitrary quantum systems using a small set of basic components. So far we have discussed the \( \text{Not()} \), \( \text{CNot()} \) and \( \text{Mix()} \) functions. To round out the set and allow for universal quantum computation, we need the Controlled Phase function, \( \text{CPhase()} \).

\( \text{CPhase()} \) takes a (classical) floating point number as its first argument and a qubit as it's second argument. \( \text{CPhase}(a,x) \) alters the component amplitudes of the machine's base states as follows:

- the amplitudes for base states where \( x \) is \( |0> \) are unchanged, while
- the amplitudes for base states where \( x \) is \( |1> \) are multiplied by \( \exp(i*a)=\cos(a)+i\sin(a) \).

The coefficients for the machine states where \( x=1 \) are rotated in the complex plain by \( a \)-radians. For example:

**Listing 15. Demonstrating the CPhase() function**

```
$ qcl --bits=5
[0/5] 1 |00000>
qucl> qureg a[1];
qucl> Mix(a);
[1/5] 0.707107 |00000> + 0.707107 |00001>
qucl> CPhase(3.14159265, a);
[1/5] 0.707107 |00000> + -0.707107 |00001>
qucl> reset
[1/5] 1 |00000>
qucl> Mix(a);
[1/5] 0.707107 |00000> + 0.707107 |00001>
qucl> CPhase(0.01, a);
[1/5] 0.707107 |00000> + (0.707071,0.00707095) |00001>
qucl> dump a
: SPECTRUM a: |....0> 0.5 |0> + 0.5 |1>
```

Introduction to Quantum Computing
Since $\exp(i\pi)=-1$, $\text{CPhase}(\pi,x)$ will flip the sign of the $|1\rangle$ component. $\text{CPhase}(0.01,x)$ rotates the phase of the $|1\rangle$ component by one one-hundredth of a radian in the complex plane. The parenthesized tuple $(0.707071,0.00707095)$ is the qcl representation of the complex number $\exp(0.01i)=0.707071+i*0.00707095$.

**Bigger problems and solutions**

Deutches problem and its N-bit generalization, the Deutch-Jozsa problem, may be interesting, but they don't have much practical value. Fortunately, there are other quantum algorithms that promise bigger payoffs.

Shor's algorithm, for example, is able to find the period of a function of N bits in polynomial time. While this doesn't sound like a big deal, the difficulty of factoring and finding a discrete logarithm forms the basis of most if not all public-key cryptography systems.

Less spectacular, but much easier to implement is Grover's algorithm which searches an unordered list of N items in $O(\sqrt{N})$ time. The best classical algorithm takes, on average, $N/2$ iterations to search such a list.

**Conclusions**

One of the tasks of classical computers since their inception as been to simulate electrical circuits to help design faster computers. This feedback loop has helped fuel half a century of explosive growth in the computer industry. Quantum Computing has the potential to shift this explosive growth into an even higher gear as QC's are used in the creation of faster and more powerful quantum computing elements.

In Aug 2000, Isaac L. Chuang of the IBM Almaden Research Center announced that he and his collaborators had constructed a 5 qubit machine using a molecule with 5 FI atoms (see Related topics). Unfortunately this technology probably won't scale up to a usable size.

So when will the first scalable quantum computer be built? There are several candidate technologies for storing, manipulating and moving qubits. A complete list is beyond the scope of this article, but it's probably safe to say that the first useful QC is still one or two decades away.
Related topics

• Download qcl, the programming language for quantum computers discussed throughout this article.
• Introduction to Quantum Computation and Information is a good collection of articles. Prior to Nielsen and Chuang, it was probably the best available book on the subject.
• "Experimental realization of order-finding with a quantum computer", a discussion of the 5 qubit machine built by IBM Almaden Research Center, can also be found at: Vandersypen, L.M.K., et al. Preprint.
• Browse more Linux resources on developerWorks.
• Browse more Open source resources on developerWorks.