The Computer Science Side of Quantum Computation

Morgan Betts
98053250

April 4, 2003

Contents

1 Introduction

Classical computer scientists entering the field of quantum computing are forced to change their approach to many problems. Leaving behind the familiarity of programming languages and Turing Machines, they are presented with a confusing variety of formalisms (Dirac notation, matrices, gates, operators, etc)[6]. It has been thought that approaching quantum computation in a manner more analogous to classical computer science would not only ease the transition of the neophyte quantum computer scientist, but would assist in the movement of standard classical techniques to the quantum world.

This paper will attempt to introduce a more computer science-centric approach to quantum computation. Section 2 gives an introduction to Qauntum Turing Machines, and their relationship to classical Turing Machines. Followed by a discussion of proposed quantum programming languages in Section 3. Finally, some conclusions are made about these topics in Section 4. Examples of common quantum algorithms implemented in the discussed quantum programming languages are presented in Appendix A.

2 Quantum Turing Machines

Classical Turing Machines (TM) have long been used in the study of computer science. Proposed by Alan Turing in his 1936 paper "On Computable Numbers, With an Application to the Entscheidungsproblem", they were an attempt to give a mathematically precise definition of an "algorithm" or "mechanical procedure". Within the area of complexity theory, Turing Machines are used heavily, and even define the basic concepts, such as computability and polynomial-time computation.

Introduced by Deutsch in 1985, Quantum Turing Machines (QTM) are equally as useful in the quantum realm of computability and complexity theory. However, this paper will merely introduce the structure of what a QTM is. For more information regarding quantum complexity theory see [3] and [4].

2.1 Introduction to QTMs

A QTM $\mathcal{Q}$ consists of a current state q, which is from a set of possible states Q, a tape configuration T represented by an infinite string from a finite set Σ of symbols, and the discretized head position ξ, taking values in the set $\mathbf{Z}$. It is assumed that Q contains two specific states q_0 and q_f representing the initial configuration and the final configuration of the processor. Thus, any configuration C of $\mathcal{Q}$ is represented by a triple $C = (q, T, \xi)$ in the configuration space $Q \times \Sigma^{\#} \times \mathbf{Z}$ (see [2], [5]).

As per usual in quantum mechanics, the dynamics of $\mathcal{Q}$ are described by a unitary operator U which specifies the evolution of any state $|\psi(t)\rangle$ during a single computational step so that we have

$$|\psi(t)\rangle = U^t|\psi(0)\rangle$$

for all positive integer t. While this operator U describes the general evolution of the QTM, there are requirements that the QTM operate finitely [5], that is:

1. only a finite system is in motions during any one step
2. the motion depends only on the state of a finite subsystem
3. the rule that specifies the motion can be given finitely in the mathematical sense

To satisfy these conditions, the motion is dependent only on the current processor state q and the current tape symbol (denoted $T(\xi)$). This gives rise to the the local transition function

$$D(q, T(\xi), q', T'(\xi), d) = c$$

where q,$q' \in Q$, $T(\xi)$,$T'(\xi) \in \Sigma$, $d \in \{-1, 0, 1\}$, $c \in \mathbf{C}$ and $0 \leq \|c\| \leq 1$.

More simply, this function means: if the current state is q and the symbol under at the head position is $T(\xi)$, then with amplitude c the machine should write the new symbol $T'(\xi)$, move the head right by d positions and go to processor state q'. Allowing $\|c\|$ to be in the range $[0, 1]$ creates non-determinism, if $\|c\| \in \{0, 1\}$ this would be a deterministic QTM.

From the finite operation requirement on the QTM, and based upon the definition of the local transition function, the time evolution operator U can be determined [5]:

$$U|q, T, \xi\rangle = \sum_{q', T'(\xi), d} D(q, T(\xi), q', T'(\xi), d)|q', T_\xi^{T'}, \xi + d\rangle$$

for any configuration (q, T, ξ), where $T_\xi^{T'}$ is the tape string defined by

$$T_\xi^{T'}(m) = \begin{cases} T'(\xi) & \text{if } m = \xi \\ T(m) & \text{if } m \neq \xi \end{cases}$$

2.2 Halting a QTM

After a QTM has halted, the result of the computation is obtained by measuring the tape string. In a classical TM, the machine is checked after every step, and the entering of the q_f state signals that the program has halted. However, continual observation of an entire quantum mechanical system will collapse the wave function, thus leaving a purely classical system. To avoid this, Deutsch thus introduced an additional qubit to his QTM, called the halt qubit, together with an observable, called the halt flag, with the eigenstates $|0\rangle$ and $|1\rangle$. The processor configuration is then described by the state vector $|q\rangle|1\rangle$ if q is the final state, and by $|q\rangle|0\rangle$ otherwise. Obviously, the halt qubit is initialized to $|0\rangle$ before computation begins, and every valid quantum algorithm sets the halt qubit to $|1\rangle$ when it enters q_f but does not interact with the halt qubit other than that[5].

Deutsch claimed that the observable can then be periodically observed from the outside without affecting the operation of the QTM. However, in

1997 J.M.Myers argued that this halting procedure spoils the computation as the non-halt qubits become entangled with the halt qubit and measurement of the halt flag changes the state. Ozawa went on to more precisely reformulate Deutsch's halting procedure, and demonstrated that while the measurement of the halt qubit does change the state of the QTM, it does not change the probability distribution of the outcome – thus not spoiling the computation. In [5], Ozawa goes further to outline a new halting procedure in which a halt qubit is unnecessary and just requires a measuring apparatus to precisely measure the observable $\hat{n}_0 = |q_f\rangle\langle q_f|$. This procedure works as follows:

1. The halt flag $\hat{n}_0$ is measured instantaneously after every step.

2. Once the halt flag is set to $\hat{n}_0 = 1$, the QTM will not change the halt flag nor the result of the computation.

3. After the measurement of the halt flag $\hat{n}_0$ give the outcome 1, the tape string $\hat{T}(S)$ in the data slot is measure and the outcome of this measurement is defined to be the output of the computation.

It was then proved that use of this halting procedure does not affect the probability distribution of the output.

3 Quantum Programming Languages

As research in the classical computer science domain increased, both circuit diagrams and Turing Machines were found inadequate for the description of algorithms. While both were capable of universal computation, they provided neither straightforward data representation nor high-level control structures. To alleviate this problem programming languages were created.

Quantum programming research is currently in a similar state. While quantum circuits have been usable for the description of the simple algorithms that exist, they do not allow for any of the features of modern computer programming languages (such as supporting structures, abstracted data types, specification and rigorous program derivation[8]). Additionally a quantum programming language (QPL) could also reduce the learning curve for prospective computer scientists, thus opening up the field to a larger number of people.

3.1 Requirements of a QPL

While no general practical quantum computer has been created, there is some idea on how such a computer should be structured. The QRAM model is the structure that is currently favoured, and is composed of two parts: a "master" classical computer, and a "slave" device dealing with the quantum resources. The "master" stores and executes the program, and sends requests to the "slave" to perform operations upon the quantum resources (such as unitary operations, or measurements). While a quantum computer could perform all the actions that the classical computer is performing, the thought is to try and keep the quantum processing part as limited in time as possible in order to help prevent decoherence [10].

Even though there are numerous quantum programming languages proposed, they agree on the requirements of such a language:

Completeness:the language must powerful enough to express the entire quantum circuit model.

Simplicity:the language should be as simple as possible, while still meeting the completeness requirement.

Hardware Abstraction:since no standard quantum hardware exists, the language must be independent from the actual implementation. This will

allow the same program to be recompiled and run on different quantum resource devices.

Separability:the language must keep classical programming and quantum programming separated, in order to be able to move to a classical machine all those computations which do not need, or which do not enjoy any speedup in being executed on, a quantum device.

3.2 Proposed Quantum Programming Languages

This paper will look at three proposed quantum programming languages and asses the merits of each one. The first QPL examined is QCL (Quantum Computation Language) created by Bernhard Ömer [6]. Next is qGCL (quantum Guarded-Command Language) by J.W.Sanders and P.Zuliani [8], which is an extension of E.Dijsktra's Guarded-Command Language. Finally, the Q language, designed and implemented by S.Bettelli, T.Calarco and L.Serafini [10], is discussed.

3.2.1 QCL

QCL is an interpreted language whose syntax is similar to C or Pascal, which allows for the complete implementation and simulation of quantum algorithms (including classical components)[6]. Some of the main features of this language include: inverse execution (allowing for on-the-fly determination of the inverse operator), quantum memory management (allowing for local quantum variables) and transparent integration of Bennet-style scratch space management.

The first of these features is rather interesting as the interpreter automatically derives the inverse operator. While this is easy if the operator is a simple unitary operator (then $U^{-1} = U^{\dagger}$), it is more complex when the operator to invert is a function (in QCL a function call can be inverted by use of the adjungation flag '!'). A function is then inverted by use of the fact that

$$(\prod_{i=1}^{n} U_i)^{\dagger} = \prod_{i=n}^{1} U_i^{\dagger}$$

Since the sequence of applied suboperators is specified using a procedural classical language which cannot be executed in reverse, the inversion is

achieved by the delayed execution of operator calls. When the adjungation flag is set, the operator body is executed and all calls of suboperators are pushed on a stack which is then processed in reverse order with inverted adjungation flags [6]. This automatic derivation of the inverse operator is possible only on specific function types, ones in which their execution may in no way depend on the global program state (which includes global variables, option and the state of the internal random number generator).

In a similar manner, the quantum memory management allows the use of quantum variables that are local to a function. When a temporary scratch register is allocated, the memory management system tracks all operators applied to this scratch register until it goes out of scope. At this time the result registers are saved, the computation is run in reverse (to clear the scratch register), and the scratch register is released.

The Bennet-style scratch space management feature is a method to reduce the extra space required to compute non-reversible functions. Non-reversible functions (such as integer division: $DIV2'|i\rangle = |i/2\rangle$, since $DIV2'|0\rangle \neq DIV2'|1\rangle$) can be converted into reversible functions by storing additional information about the arguments (such as defining the integer division function to be: $DIV2'|x,0\rangle = |x, \lfloor x/2 \rfloor\rangle$, now $DIV2'|0,0\rangle \neq DIV2'|1,0\rangle$). While this allows the computation of non-reversible, it also require extra storage for intermediate results. In longer calculations, this would require an increasing number of "junk" bits which are not required in the final result. One simple solution to this problem was proposed by Bennet: if a composition of two non-reversible functions $f(x) = h(g(x))$ is to be computer, the scratch space for the intermediate result can be 'recycled" using the following procedure (where G and H are the quantum functions for g and h, and the indices indicate the registers operated on):

$$|x,0,0\rangle \xrightarrow{G_{12}} |x,g(x),0\rangle \xrightarrow{H_{23}} |x,g(x),h(g(x))\rangle \xrightarrow{G_{12}^{\dagger}} |x,0,f(x)\rangle$$

Without scratch-management, the evaluation of a composition of depth d needs d operations and requires $d-1$ junk registers. If all the intermediate results were to be uncomputed would require $2d-1$ operations and $d-1$ scratch registers. However, a combined use of scratch registers and junk registers can reduce the necessary space by $O(1/\sqrt{d})$ with a computation overhead of $O(2)$ [6].

Since there are few quantum computers available, the QCL interpreter comes with a quantum computer simulator. This nice addition allows algo-

rithms to be implemented and tested without the need of an actual quantum computer, thus giving computer scientists a head start on algorithm design. Similarly, since this is an interpreter (and simulator), it allows for easy viewing of the current state of the quantum computer while stepping through a quantum algorithm. This ability could assist newcomers grasp the concepts behind non-intuitive quantum algorithms.

3.2.2 qGCL

The Guarded-Command Language (GCL) was proposed by E.Dijkstra as a general, imperative language. The advantage of GCL over standard programming languages of the time, was that an algorithm in written in GCL could be proven correct mathematically. The probabilistic Guarded-Command Language (pGCL) is an extension of GCL that includes that adds nondeterminism and probabilism; the former arises in specifying some quantum algorithms and the latter is required in order to "observe" a quantum system [8]. pGCL is also extended to include allow procedure invocation (this is also a standard extension to the original GCL). Finally, the quantum Guarded-Command Language (qGCL) extends pGCL to add transformation (transforming a classical bit register to its quantum analog), initialization (prepares a quantum register for computation), evolution (the iteration of unitary operators on quantum registers) and finalization (the observation of a quantum register).

These extensions however, were added very carefully and qGCL still supports specification and rigorous program derivation. Both of these abilities are all-the-more important when dealing with quantum algorithms, which, being intricate and exploiting non-standard intuition, provide exactly the kinds of programs for which refinement calculi were developed.

Later work by Zuliani [9], presented a technique for transforming a generic pGCL program into an equivalent reversible pGCL program (thus allowing execution on a quantum computer).

While the actual language may appear atypical to a pure computer scientist, it becomes quite usable after a small amount of effort, and the ability to formally prove an algorithm correct is well worth the effort. However, while there exist quantum computer simulators, and compilers for GCL, a compiler for qGCL that made use of a quantum computer simulator was not found. Thus, while qGCL is able to transfer the specification and rigorous program derivation from classical computer science over to quantum computation, it

is not a language that is useful in terms of actually performing experiments with them.

3.2.3 The Q Language

The simply named Q language is a library of quantum objects for use with C++. It follows the typical QRAM model, but differs from both QCL and qGCL in that it stresses the use of quantum operator objects instead of functions for representing quantum circuits.

The main advantage of the operator object is cited as the potential for operator composition and simplification. In the functional approach, if a Hadamard were applied to a qubit twice, then the classical computer would communicate to the quantum resource device twice, telling it to perform a Hadamard transformation each time. However, in the operator object method the operator object, which would consist of two concatenated Hadamard gates, could perform the simplification of the two gates and realize that applying the identity unnecessary (thus increasing performance). More simply this allows for composition and simplification of operators even before allocating the quantum registers (by embedding the simplification routines within the operator composition primitives). It is noted, however, that some optimization routines may be too expensive for automatic embedding, and it is possible to leave to the programmer the freedom to call the simplification routine at their discretion [10].

Another feature of the Q language, is the automatic calculation of the inverse of an operator. This is done in the same manner as in the QCL language, however no mention is made of the restriction to special case operators whose execution does not depend on the global program state (as in QCL).

There has been some discussion over the use of an interpreted QPL (as in QCL) versus a compiled QPL (as in the Q language), and in [11] both Bettelli and Ömer state their opinions on this topic. Bettelli's effort has been to make as much of their quantum computing extension compiled as in interpreted languages there is no compiler to catch code that the computer cannot handle, which is especially true for unusual hardware like quantum computers. Ömer on the other hand mentions that it will be difficult to create a compiler that understands the requirements of quantum computers, and thus these restrictions will have to be enforced solely by the implementation of the class library and the discipline of the programmer. While both approaches have

distinct advantages and disadvantages, it is yet to be seen which one will prevail.

Not only is this language is built upon one of the most popular programming languages, but also comes with a built-in quantum simulator. This greatly reduces the learning curve and makes it very accessible to a wide array of computer scientists.

4 Conclusions

As Turing Machines are relegated to the area of complexity theory in classical computer science, Quantum Turing Machines will be of little use outside of quantum complexity theory in quantum computation. However, since quantum computation is still such a young science there is much work to be done in this area, with many exciting results yet to be proved.

Existing quantum programming languages are definitely an asset to research in quantum computation. Not only do they provide reliable mechanisms to implement and test quantum algorithms, but also a conceptually easier method for approaching these algorithms (at least from a computer science point-of-view). With the progress of mathematically verifiable languages (such as qGCL), the correctness of algorithms can be proven more easily, and programs can be derived directly from algorithms (see Section 7 in [8]).

At the time when quantum computing devices becomes physically realizable, the features of these languages will aid in the development of longer quantum algorithms (through use of such features as automatic derivation of operator inverses and Bennet-style scratch space management).

While this approach to quantum algorithms may not prove immediately effective in advancing quantum algorithms, the benefit is to widen this area of research and allow computer scientists to more easily apply their expertise in devising algorithms.

A Algorithms Implemented in QPLs

Below some common quantum algorithms are shown implemented in the QPLs discussed previously.

A.1 Discrete Fourier Transform in QCL

```
operator dft(qureg q) {  // main operator
    const n=#q;          // set n to length of input
   int i; int j;         // declare loop counters
   for i=0 to n-1 {
      for j=0 to i-1 {   // apply conditional phase gates
         CPhase(2*pi/2^(i-j+1),q[n-i-1] & q[n-j-1]);
      }
      Mix(q[n-i-1]);     // qubit rotation
   }
   flip(q)               // swap bit order of output
}
```

Extracted from [6].

A.2 Grover's Search in qGCL

var $\chi : q(\mathcal{B}^n), i : \mathcal{B}^n, j : 0..2^n \bullet$
$In(\chi);$
do N *times* $\rightarrow$
$\chi := T_f(\chi);$
od;
$Fin[\Delta](i);$
$j := num(i)$

where the transform T_f is defined by

$$T_f : q(\mathcal{B}^n) \rightarrow q(\mathcal{B}^n)$$
$$(T_f\chi)(x) \hat{=} (-1)^{f(x)}\chi(x)$$

and M inverts χ (point-wise) about its average

$$M : q(\mathcal{B}^n) \rightarrow q(\mathcal{B}^n)$$
$$(M\chi)(x) \hat{=} 2[2^{-n}\Sigma_{y:\mathcal{B}^n}\chi(y)] - \chi(x)$$

Extracted from [8].

A.3 Order Finding in Q

```
Qbitset order_finding(int x, int N, int n, float epsilon)
{
    int t = n + ceil(log(1+1/(2*epsilon))/log(2));
    int m = ceil(log(N)/log(2));
    int q = x;
    Qop controlled_multiply[t];
    for (int i=0; i<t; ++i, q=((q*q)%N))
        controlled_multiply[i] << Qop(generate_multiply(q, N), 1);
    Qop mixer = QHadamard(t);
    Qreg phase(t);
    Qreg eigen(m, 1);
    mixer(phase);
    for (int i=0; i<t; i++ )
        controlled_multiply[i](phase[i] & eigen);
    return phase.measure();
}
```

Extracted from [10].

References

[1] M.A.Nielsen, I.L.Chuang, "Quantum Computation and Quantum Information", Cambridge University Press, New York, NY, 2000

[2] Andrés Sicard and Mario Vélez, Some relations between quantum Turing machines and Turing machines, quant-ph/9912012, 2001

[3] Ethan Bernstein and Umesh Vazirani, Quantum complexity theory, http://epubs.siam.org/sam-bin/dbq/article/30092, 1997

[4] Tomoyuki Yamakami, A Foundation of Programming a Multi-Tape Quantum Turing Machine (Preliminary Version), quant-ph/9906084, 1999

[5] Masanao Ozawa, Quantum Turing Machines: Local Transition, Preparation, Measurement, and Halting, quant-ph/9809038, 1998

[6] Bernhard Ömer, "A Procedural Formalism for Quantum Computing", Master Thesis (theoretical physics), http://tph.tuwien.ac.at/~oemer/qcl.html, 1998

[7] Bernhard Ömer, Classical Concepts in Quantum Programming, quant-ph/0211100, 2002

[8] J.W.Sanders, P.Zuliani, Quantum Programming, http://web.comlab.ox.ac.uk/oucl/publications/tr/index.html, 2000

[9] P.Zuliani, Logical reversibility, http://www.research.ibm.com/journal/rd/456/zuliani.html, 2001

[10] S.Bettelli, T.Calarco, L.Serafini, Toward an architecture for quantum programming, quant-ph/0103009, 2001

[11] Eric Smalley, Programming Goes Quantum, http://www.trnmag.com/Stories/032801/Programming_goes_quantum_032801.html, 2001