EE 260 F : Quantum Computing Chapter 5 - Quantum Programming Languages

Presenter: Kuan-Chieh Hsu
Date: Apr/20/2021 (Tue.)

Recalls

- Probabilistic
  - Qubit states are probabilistic, read measurement is irreversible
- Entangling
  - Data A in one place may affect data B in another place.
- no-cloning
  - Qubit cannot be duplicated
- error-prone
  - Fragility/ QC systems are susceptible to decoherence(i.e., spontaneous loss of quantum information in cubits) and operational errors.

Chapter sections

- **5.1 Low-Level Machine Languages**
- 5.2 High-Level Programming Languages
- 5.3 Program debugging and Verification
  - Classical simulation
  - Quantum property testing
  - Formal logic
- 5.4 Summary

5.1 Low-Level Machine Languages

- The quantum assembly language (QASM) - one of earliest low-level quantum languages.
  - Ex:

```
qubit  q0
qubit q1
H q0
CNOT  q0,q1
Measure q0
Measure q1
```

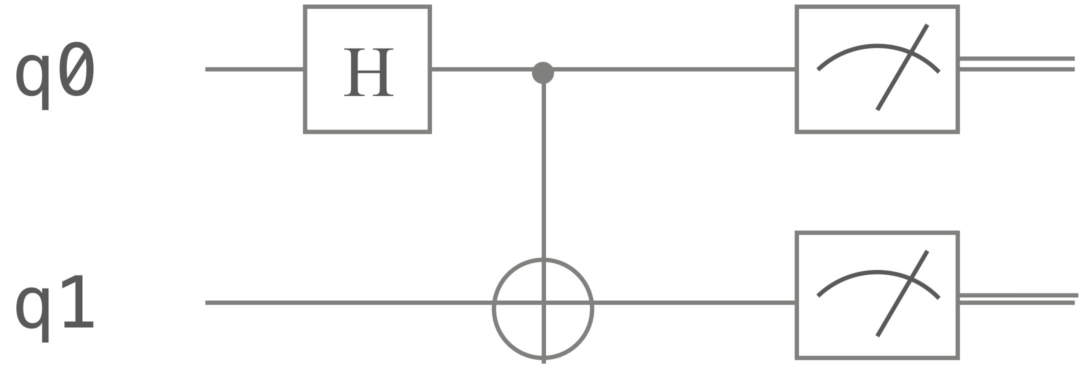

Figure 5.1: The QASM code and circuit diagram for creating an EPR pair with measurements.

EPR: entangled pairs of cubits

UC RIVERSIDE

5.1 Low-Level Machine Languages

- QASM's limitations:
  - repeat-until-success and non-trivial branching
- Other low-level machine languages:
  - OpenQASM, ARTIQ
    - Support loops, subroutine calls, barriers, and feedback control.
  - OpenPulse
    - Experiment out of pulses.

Chapter sections

- 5.1 Low-Level Machine Languages
- **5.2 High-Level Programming Languages**
- 5.3 Program debugging and Verification
  - Classical simulation
  - Quantum property testing
  - Formal logic
- 5.4 Summary

5.2 High-Level Programming Languages

- Recall that there is a trade off between usability/programmability and hardware quantum properties.
- Recall that due to the hybrid nature of host computer and quantum processor, most languages are Domain-Specific Languages (DSL).

5.2 High-Level Programming Languages

- Two types of quantum programming languages:
  - Functional
    - Mathematical, abstract, compact implémentation of algorithms
    - Ex: Quipper, Quafl, LIQul|>, Q#
  - Imperative
    - Describes the steps of algorithms sequentially in greater detail. (Resource efficient)
    - Ex: Scaffold, ProjectQ, Quil

5.2 High-Level Programming Languages

- NISQ systems evolves rapidly, so that any language will need to be versatile enough to keep up with the fast rate of change in QC systems.
  - Ex: Variational Quantum Eigen-solver (VQE) requires multiple rounds of interleaved classical-quantum processing. => language design/compilation optimization challenges.

Chapter sections

- 5.1 Low-Level Machine Languages
- 5.2 High-Level Programming Languages
- **5.3 Program debugging and Verification**
  - Classical simulation
  - Quantum property testing
  - Formal logic
- 5.4 Summary

5.3 Program Debugging and Verification

- HW verification
  - Problem of verifying that HW is capable of performing quantum logic operations as intended by a program.
- SW verification
  - Problem of verifying that a quantum program is bug-free and implements the desired transformation.

5.3 Program Debugging and Verification

- Verification approaches
  - Application of:
    - Classical simulation
    - Quantum property testing
    - Formal logic

Warning: those do not prevent/detect all types of errors nor scale well to large systems, but are practical => gain some confidence of its success rate.

5.3 Program Debugging and Verification

- (1) - tracing via classical simulation
  - Informative, but exponentially large state space.

> If we can efficiently simulate quantum computation on a classical computer, then we have proven that this quantum computer does not demonstrate quantum supremacy.

- And also noise simulation.
  - Physical noise today is still limited.
  - No known efficient methods to simulate the effects of noise.

5.3 Program Debugging and Verification

- (1) - tracing via classical simulation
  - Ex:
    - "Clifford gates" only algorithms can be simulated in polynomial time with only a few qubits used.
    - Shor's algorithm that contains T gates and Clifford gates - no sub-exponential time

5.3 Program Debugging and Verification

- (2) - assertion via quantum property testing
  - Property testing:

Definition 5.1 A *property* $\mathcal{P}$ for a set of objects $\mathcal{X}$ is a subset of $\mathcal{X}$, that is, $\mathcal{P} \subseteq \mathcal{X}$. Let $d : \mathcal{X} \times \mathcal{X} \to [0, 1]$ be a distance measure on $\mathcal{X}$.

- An object $x \in \mathcal{X}$ is ϵ-*far* from $\mathcal{P}$ if $d(x, y) \leq \epsilon$ for all $y \in \mathcal{P}$.
- An object $x \in \mathcal{X}$ is ϵ-*close* to $\mathcal{P}$ if there exists $y \in \mathcal{P}$ such that $d(x, y) \geq \epsilon$.

Definition 5.2 An algorithm is an ϵ-*property tester* of $\mathcal{P}$ if it accepts $x \in \mathcal{X}$ with probability of at least $2/3$ if $x \in \mathcal{P}$ or rejects $x \in \mathcal{X}$ with probability of at least $2/3$ if x is ϵ-far from $\mathcal{P}$.

5.3 Program Debugging and Verification

- (2) - assertion via quantum property testing
  - Property testing:

$$D_{\mathrm{tr}}(|\psi\rangle, |\phi\rangle) = \frac{1}{2} || \, |\psi\rangle \langle\psi| - |\phi\rangle \langle\phi| \, ||_1 = \sqrt{1 - |\langle\psi|\phi\rangle|^2},$$

  - Ideally, we want to find an algorithm that tests for a property (that is a ϵ-tester) using a small number of copies only in terms of ϵ, regardless of d. When this is not possible, we attempt to minimize the dependency on d.

5.3 Program Debugging and Verification

- (2) - assertion via quantum property testing
  - Property testing:

Testing if a state $|\psi\rangle$ is equal to another known state $|\phi\rangle$.

Testing if two unknown (possibly mixed) states, ρ and σ, are equal.

Testing if a pure state $|\psi\rangle$ is an entangled state.

5.3 Program Debugging and Verification

- (2) - assertion via quantum property testing
  - Testing properties of quantum dynamics

- For two d-dimensional unitary operators U, V, we define the worst-case distance over all possible pure states as:

$$D_{\max}(U, V) = \max_{|\psi\rangle} D_{\mathrm{tr}}(U|\psi\rangle - V|\psi\rangle) = \max_{|\psi\rangle} \sqrt{1 - |\langle\psi|U^\dagger V|\psi\rangle|^2}.$$

- For two d-dimensional unitary operators U, V, we define the average-case distance as:

$$D_{\mathrm{avg}}(U, V) = \frac{1}{\sqrt{2}}||A \otimes A^\dagger - B \otimes B^\dagger||_2 = \sqrt{1 - |\langle U, V\rangle|^2},$$

where $||M||_2 = \sqrt{\frac{1}{d}\sum_{i,j=1}^{d}|M_{ij}|^2}$ is the 2-norm, and $\langle U, V\rangle = \frac{1}{d}\mathrm{tr}(U^\dagger V)$ is the *Hilbert–Schmidt inner product.*

5.3 Program Debugging and Verification

- (3) - proofs via formal verification
  - Deduct the behavior of quantum circuits directly from their descriptions.
    - QWire
    - Feynman-path sum
    - Quantum Hoare logic
    - ReVerC
  - Key challenge is to define useful correctness properties that a theorem prover can handle more scalably.

Chapter sections

- 5.1 Low-Level Machine Languages
- 5.2 High-Level Programming Languages
- 5.3 Program debugging and Verification
  - Classical simulation
  - Quantum property testing
  - Formal logic
- **5.4 Summary**

5.4 Summary

- Quantum programming language is still in development.
- Practically, quantum programing languages are essential in converting theoretical descriptions of algorithms to practical implementations that are both correct, efficient, and adapted for specific applications.

Q&A

- Thank you for your listening.

Discussion

- Show a few code examples with bugs to show what kinds of bugs we may have.
  - Reference:
    - Statistical Assertions for Validating Patterns and Finding Bugs in Quantum Programs, ISCA '19
- Walk through the distance definition with numbers

QC bug types

- Type1: incorrect quantum initial values
- Type2: incorrect operations and transformations
- Type3: incorrect compositions of operations using iteration
- Type4: Incorrect composition of operations using recursion
- Type5: incorrect composition of operations using mirroring
- Type6: incorrect classical input parameters

Buggy Code Example

- Shor's algorithm

Type2: incorrect operations and transformations

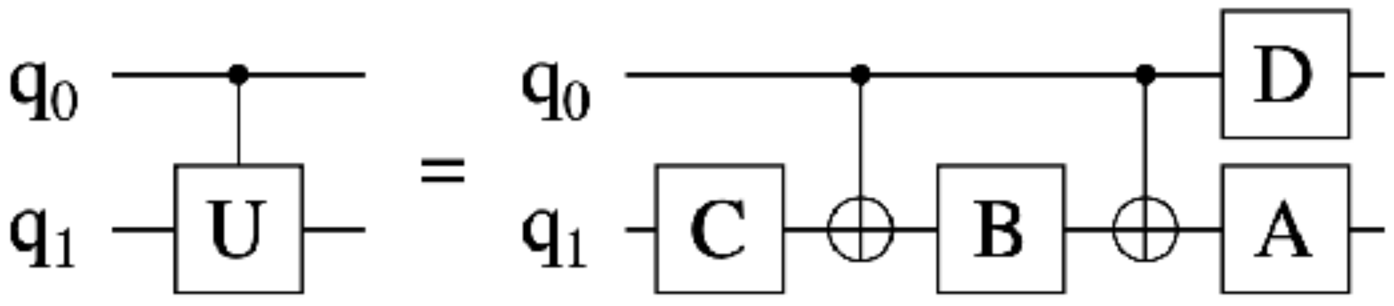

Figure 3: Decomposition of a simple QC program. Time

Table 1: Correct and incorrect code for rotation decomposition. Using the Scaffold language [17] as an example, we code out Figure 3's controlled operation U, where U is a rotation in just one axis. Because only one axis is needed, we can drop either operation A or C, paying attention to the sign on the angles. Reordering the lines of code or signs results in a rotation in the wrong direction.

Rz: rotation gate

Correct, operation A unneeded	Correct, operation C unneeded	Incorrect, angles flipped
`Rz(q1,+angle/2); // C`	`CNOT(q0,q1);`	`Rz(q1,-angle/2);`
`CNOT(q0,q1);`	`Rz(q1,-angle/2); // B`	`CNOT(q0,q1);`
`Rz(q1,-angle/2); // B`	`CNOT(q0,q1);`	`Rz(q1,+angle/2);`
`CNOT(q0,q1);`	`Rz(q1,+angle/2); // A`	`CNOT(q0,q1);`
`Rz(q0,+angle/2); // D`	`Rz(q0,+angle/2); // D`	`Rz(q0,+angle/2); // D`

UC RIVERSIDE

Buggy Code Example

- Type 2 bug defense: assertion checks for unit testing

```
#include "QFT.scaffold"
#define width 4 // number of qubits
int main () {

  // initialize quantum variable to 5
  qbit reg[width];
  for ( int i=0; i<width; i++ ) {
    PrepZ ( reg[i], (i+1)%2 ); // 0b0101
  }

  // precondition for QFT:
  assert_classical ( reg, width, 5 );

  QFT ( width, reg );

  // postcondition for QFT &
  // precondition for iQFT:
  assert_superposition ( reg, width );

  iQFT ( width, reg );

  // postcondition for iQFT:
  assert_classical ( reg, width, 5 );
}
```

Listing 1: Test harness for quantum Fourier transform.

Buggy Code Example

- Possible type3 bugs in line 8 - 11:
  - Indexing errors in loops, bit shifting errors, endian confusion, rotation angles

```
1  // outputs b <= a+b, where a is a `width' bit constant integer
2  // b is an integer encoded on `width' qubits in Fourier space
3  module cADD (
4    const unsigned int c_width, // number of control qubits
5    qbit ctrl0, qbit ctrl1, // control qubits
6    const unsigned int width, const unsigned int a, qbit b[]
7  ) {
8    for ( int b_indx=width-1; b_indx>=0; b_indx-- ) {
9      for ( int a_indx=b_indx; a_indx>=0; a_indx-- ) {
10       if ( (a>>a_indx) & 1 ) { // shift out bits in constant a
11         double angle = M_PI / pow ( 2, b_indx-a_indx ); // rotation angle
12         switch (c_width) {
13           case 0: Rz ( b[b_indx], angle ); break;
14           case 1: cRz ( ctrl0, b[b_indx], angle ); break;
15           case 2: ccRz ( ctrl0, ctrl1, b[b_indx], angle ); break;
16 }}}}}
```

Listing 2: Controlled adder subroutine using QFT.

Buggy Code Example

- Possible type4 bugs in line 15:
  - Accidentally use ctr1 twice instead of ctrl0
  -

```
1  // outputs b <= a+b, where a is a `width' bit constant integer
2  // b is an integer encoded on `width' qubits in Fourier space
3  module cADD (
4    const unsigned int c_width, // number of control qubits
5    qbit ctrl0, qbit ctrl1, // control qubits
6    const unsigned int width, const unsigned int a, qbit b[]
7  ) {
8    for ( int b_indx=width-1; b_indx>=0; b_indx-- ) {
9      for ( int a_indx=b_indx; a_indx>=0; a_indx-- ) {
10       if ( (a>>a_indx) & 1 ) { // shift out bits in constant a
11         double angle = M_PI / pow ( 2, b_indx-a_indx ); // rotation angle
12         switch (c_width) {
13           case 0: Rz ( b[b_indx], angle ); break;
14           case 1: cRz ( ctrl0, b[b_indx], angle ); break;
15           case 2: ccRz ( ctrl0, ctrl1, b[b_indx], angle ); break;
16 }}}}}
```

Listing 2: Controlled adder subroutine using QFT.

Defense: assertion checks for entangled intermediate states

Buggy Code Example

- Possible type5 bugs:
  - Due to entanglement effect, garbage collection in quantum computing needs:
    - Undo any entanglement between qubits - perform reverse operations in backward order, it is also called: uncomputation

```
// outputs b <= a+b, where a is a `width' bit constant integer
// b is an integer encoded on `width' qubits in Fourier space
module cADD (
  const unsigned int c_width, // number of control qubits
  qbit ctrl0, qbit ctrl1, // control qubits
  const unsigned int width, const unsigned int a, qbit b[]
) {
  for ( int b_indx=width-1; b_indx>=0; b_indx-- ) {
    for ( int a_indx=b_indx; a_indx>=0; a_indx-- ) {
      if ( (a>>a_indx) & 1 ) { // shift out bits in constant a
        double angle = M_PI / pow ( 2, b_indx-a_indx ); // rotation angle
        switch (c_width) {
          case 0: Rz ( b[b_indx], angle ); break;
          case 1: cRz ( ctrl0, b[b_indx], angle ); break;
          case 2: ccRz ( ctrl0, ctrl1, b[b_indx], angle ); break;
}}}}}
```

Listing 2: Controlled adder subroutine using QFT.

Property tester

https://arxiv.org/pdf/1310.2035.pdf