Quest Journals
Journal of Software Engineering and Simulation
Volume 9 ~ Issue 7 (2023) pp: 30-35
ISSN(Online) :2321-3795 ISSN (Print):2321-3809
www.questjournals.org

Research Paper

Quantum vs Classical in Non-Determinism Programming

Stephanie Wang
Stanford University Online High School

Abstract
*A non-deterministic (**ND**) programming language is a type of programming language that allows multiple outcomes for a given input or set of inputs. In non-deterministic programming, there are a set of possible outcomes for a given input, and the program will choose one of those outcomes at runtime.*
In classical computing, non-deterministic programming is commonly implemented using randomization, which can be time-consuming, inefficient, and limited to explore all possible outcomes.
Traditional computers use bits in computation representing data as either 1 or 0, whereas quantum computers use quantum bits (aka, qubits) representing both 1 and 0 at the same time. By creating a superposition of all possible states of a given problem, a quantum computer can explore all possible solutions to the problem at once rather than sequentially.

Keywords
Non-determinism, Quantum computing, Programming, Python

***R**eceived 08 July, 2023; **R**evised 18 July, 2023; **A**ccepted 20 July, 2023 © The author(s) 2023.*
P**ublished with open access at **www.questjournals.org

I. Introduction

In this section, we will introduce the ***While*** programming language with random assignment, representing infinite non-determinism, and study it by comparing classic programming and quantum programming in this paper.

1.1. *While*^{RA} programming language

The language of while programming is a fundamental model for imperative programming on any data type. Imperative programming is a programming paradigm that describes a sequence of steps, executed in order to accomplish a particular task.

Firstly, ***While*** programming language [1] supports concurrent assignments,

$$x_1, \ldots, x_m := v_1, \ldots, v_m$$

where $x_1, \ldots, x_m$ are input variables and $v_1, \ldots, v_m$ are given values with $1 \leq i \leq m$.

The flow control and sequencing of statements contain the following three constructs for the given statements ***Stmt***$_1$, ***Stmt***$_2$ and ***Stmt***, and the boolean test *b*:

(i) *sequential composition*: ***Stmt***$_1$; ***Stmt***$_2$,
(ii) *conditional*: **if** *b* **then** ***Stmt***$_1$ **else** ***Stmt***$_2$ **fi**,
(iii) *iteration*: **while** *b* **do** ***Stmt*** **end**.

Now we add the random assignment x := ? to this language, which we call ***While***^{RA}, our non-deterministic programming language, for variables x over any algebra.

To summarize, the ***ND*** programming language based on ***While***^{RA} programming language combines its constructs as follows [2],

(i) *sequential composition*
(ii) *conditional*
(iii) *iteration*

(iv) *concurrent assignment*
(v) *random assignment*.

In the following chapters, we will look at different quantum computing technologies, compare classical computing and quantum computing, and their differences in handling non-determinism.

II. Quantum and Classic Computing

Quantum computing is a type of computing that uses quantum-mechanical phenomena to perform operations on data. These phenomena include superposition and entanglement that describe the particles' behavior at the atomic and subatomic levels based on the principles of quantum mechanics [3].

There are several top quantum computing services available for researchers, developers, and businesses who want to experiment with quantum computing or incorporate it into their applications. Here are some of the top quantum computing services,

2.1 IBM Quantum Experience

IBM is one of the leading companies in developing quantum computing technologies. They offer a cloud-based quantum computing platform called IBM Quantum Experience, which provides access to various quantum computing hardware and simulators, along with tools and resources for programming and developing quantum applications. Users can write quantum programs using popular programming languages such as Python and Qiskit and run these programs on IBM's quantum hardware or simulators [4].

2.2 Microsoft Azure Quantum

Microsoft Azure Quantum is a cloud-based quantum computing service that provides access to quantum simulators and hardware from different vendors, as well as development tools and integration with other Azure services. Azure Quantum allows users to run quantum algorithms and experiments using quantum hardware and simulators from different vendors, such as Honeywell, IonQ, and Quantum Motion.

2.3 Classical Computing

Classical computing refers to the use of digital computers that process information using classical bits, which can be in a state of either 1 or 0; and the bits are manipulated using logical operators such as AND, OR, and NOT, and are stored and transmitted using physical devices such as transistors.

Programming is the process of creating software applications using programming languages such as Java, Python, and C++. These classical programming languages are designed to work with classical computers with a set of rules and syntax for writing instructions that computers can execute. Some of the most popular classical programming languages include,

1. Java: A general-purpose language that is designed to be platform-independent and can be used to create a wide range of applications, from desktop and mobile apps to enterprise systems and web applications.
2. Python: A high-level, interpreted language that is widely used for scientific computing, data analysis, artificial intelligence, and web development.
3. C++: A high-performance language that is used for system programming, and applications that require efficient memory management and low-level access to the hardware layer.

In the next chapter, we will explore and compare Quantum Computing and Classical Computing in non-deterministic Programming.

III. Quantum vs. Classical in *ND* programming

In classical computing, non-deterministic programming is usually implemented using randomization, which can be time-consuming, inefficient, and limited in its ability to explore all possible outcomes.

Traditional computers use bits representing information as either 1 or 0, whereas quantum computers use quantum bits (aka, qubits) representing both 1 and 0 at the same time. By creating a superposition of all possible states of

a given problem, a quantum computer can explore all possible solutions to the problem at once rather than sequentially [5].

We want to implement ***ND*** programming in Chapter 1 using IBM Quantum Computing and a classical programming language like Python and compare their differences.

In the IBM Quantum Experience platform, quantum circuits are typically defined using the Qiskit Python package. Qiskit provides a QuantumCircuit class that allows you to define quantum circuits and apply quantum gates to qubits.

3.1 Concurrent assignment

In the IBM Quantum Experience, concurrent assignment of qubits in a quantum circuit can be done as below,

```
from qiskit import QuantumCircuit

# Create a 2-qubit quantum circuit
circ = QuantumCircuit(2)

# Define the concurrent assignments
circ.cx(0, 1)
circ.cx(1, 0)
```

In classical programming with Python [6], a concurrent assignment can be done as below,

$$x_1, ..., x_m = v_1, ..., v_m$$

where $x_1, ..., x_m$ are program input arguments and $v_1, ..., v_m$ are assigned values.

You can also write a function that does concurrent assignments with arbitrary numbers of arguments and values,

```
def concurrent_assign(**kwargs):
    return kwargs
```

3.2 Random assignment

Random assignment in IBM Quantum Experience can be achieved using qubits that represent both 1 and 0 at the same time.

```
from qiskit import QuantumCircuit, execute

# Define the number of qubits and create a quantum circuit
n_qubits = 4
circuit = QuantumCircuit(n_qubits, n_qubits)

# Choose a backend to run the circuit
backend = least_busy(provider.backends(
    filters = lambda x: x.configuration().n_qubits >= n_quibits and
        not x.configuration().simulator))
job = execute(circuit, backend=backend, shots=1)
```

You can use **random** module to build random assignments in classical programming with Python.

```
import random

x = bool(random.getrandbits(1))
```

3.3 Sequential composition

Here is an example of the sequential composition of two quantum circuits in IBM Quantum Experience,

1. we created two quantum circuits **circuit1** and **circuit2** that operate on different quantum registers **qreg1** and **qreg2**, respectively. We add gates to **circuit1** and **circuit2** using the **circuit.h** and **circuit.cx** methods.
2. We combine the two circuits into one.
3. Finally, we measure the qubits in **qreg1** and store the measurement results in **creg1**.

```
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister

# Create two quantum registers and two classical registers
num_qubits = 2
qreg1 = QuantumRegister(num_qubits, name='qreg1')
qreg2 = QuantumRegister(num_qubits, name='qreg2')
creg1 = ClassicalRegister(num_qubits, name='creg1')
creg2 = ClassicalRegister(num_qubits, name='creg2')

# Create two quantum circuits
circuit1 = QuantumCircuit(qreg1, creg1, name='circuit1')
circuit2 = QuantumCircuit(qreg2, creg2, name='circuit2')

# Add gates to circuit1 and circuit2
circuit1.h(qreg1[0])
circuit1.cx(qreg1[0], qreg1[1])
circuit2.cx(qreg2[0], qreg2[1])

# Combine the circuits into one and measure quibits
circuit_new = circuit1 + circuit2
circuit_new.measure(qreg1, creg1)
```

In classical programming with Python, we defined two simple functions **add_one** and **multiply_by_two** that perform basic arithmetic operations, respectively. We then define a new function add_one_then_multiply_by_two that composes them sequentially.

```
def add_one(x):
    return x + 1

def multiply_by_two(x):
    return x * 2

# Sequential composition of the two functions
def add_one_then_multiply_by_two(x):
    y = add_one(x)
    z = multiply_by_two(y)
    return z

# Example usage
result = add_one_then_multiply_by_two(3)
```

3.4 Conditional

In the context of quantum computing, guarded command conditionals can be used to control the flow of execution based on the outcome of quantum measurements. In the following example, we created a quantum circuit with two qubits and two classical bits, added a conditional gate that applies to the second qubit if the measurement outcome on the first qubit is 1, and then executed the circuit job.

```
from qiskit import QuantumCircuit, Aer, execute

# Create a quantum circuit with 2 qubits and 2 classical bits
qc = QuantumCircuit(2, 2)

# Apply a Hadamard gate to the first qubit
qc.h(0)

# Measure the first qubit and store the result in the first classical bit
qc.measure(0, 0)

# Add a conditional gate based on the measurement outcome
qc.x(1).c_if(0, 1)

# Measure the second qubit and store the result in the second classical bit
qc.measure(1, 1)

# Execute the circuit on a local simulator
backend = Aer.get_backend('qasm_simulator')
job = execute(qc, backend, shots=1024)

# Get the results and print the counts
counts = job.result().get_counts(qc)
print(counts)
```

In classical programming with Python, you can use the if, elif, and else statements (or equivalent) to implement a guarded command conditional. Here's an example,

```
x = 10
y = 20

if x < y:
    print("x is less than y")
elif x == y:
    print("x is equal to y")
else:
    print("x is greater than y")
```

3.5 Iteration

In quantum computing, a guarded command iteration is a loop that executes a code block until a specific condition is met. Here's an example of how to implement a guarded command iteration in IBM Quantum Experience,

```
from qiskit import QuantumCircuit, Aer, execute

qc = QuantumCircuit(2, 2)
qc.h(0)

# Set the loop counter to 0 and loop until the counter is equal to 10
i = 0
while i < 10:
    # Measure the first qubit and store the result in the first classical bit
    # Add a conditional gate based on the measurement outcome
    qc.measure(0, 0)
    qc.x(1).c_if(0, 1)

    # Increment the loop counter
    i += 1

# Measure the second qubit and store the result in the second classical bit
qc.measure(1, 1)

# Execute the circuit on a local simulator
backend = Aer.get_backend('qasm_simulator')
job = execute(qc, backend, shots=1024)

# Get the results and print the counts
counts = job.result().get_counts(qc)
```

In classical programming with Python, you can use while loop or for loop to execute a code block (or commands) until the counter ends. Here's an example,

```
# Initialize the loop counter and loop until counter is 5
i = 0
while i < 5:
    # Print the current value of the loop counter
    print("Loop counter:", i)

    # Increment the loop counter by 1
    i += 1
```

IV. Conclusion

We explored and compared Quantum Computing and Classical Computing in non-deterministic Programming, including assignment, control, and sequencing of basic computations.

In quantum computing, non-determinism is an inherent property of quantum mechanics. Measuring a qubit in a superposition state collapses the state to a single value, but the outcome of the measurement cannot be predicted with certainty. Therefore, quantum algorithms can be thought of as inherently non-deterministic, as they involve manipulating qubits in a way that exploits the probabilistic nature of quantum mechanics.

The use of non-determinism in quantum computing can improve the performance of the solution for certain types of problems, particularly in areas such as optimization and machine learning. However, designing and implementing non-deterministic quantum algorithms is a significant challenge because it requires careful consideration of the probabilistic nature of quantum mechanics and the constraints imposed by the hardware.

While classical computing and programming are still the dominant paradigm in computing, quantum computing and programming are emerging fields that are expected to have a significant impact in the coming years.

References

[1]. Tucker, J.V. & Zucker, J.I. (1999). Computation by 'while' programs on topological partial algebras. *Theoretical Computer Science*, 219 (1-2), 379 – 420. https://doi.org/10.1016/S0304-3975(98)00297-7

[2]. Jiang, W., Wang. Y. & Zucker, J.I. (2007). Universality and semicomputability for nondeterministic programming languages over abstract algebras. *The Journal of Logic and Algebraic Programming*, 71(1), 44 – 78. https://doi.org/10.1016/j.jlap.2006.09.001

[3]. Rieffel, E.G. & Polak, W.H. (2014). Quantum computing: A gentle introduction. The MIT Press.

[4]. Loredo, R. (2020). Learn quantum computing with Python and IBM quantum experience: A hands-on introduction to quantum computing and writing your own quantum programs with Python. Packt Publishing.

[5]. Nielsen, M.A. & Chuang, I. (2011). Quantum computation and quantum information. Cambridge University Press.

[6]. Lott, S.F. & Baffy, R. (2022). Functional Python programming. Packt Publishing.

*Corresponding Author: Stephanie Wang