CSBB 311 : QUANTUM COMPUTING

LAB ASSIGNMENT 5 : Scalable Shor's Algorithm

Submitted By:

Name: KARTIK MITTAL

Roll No: 221210056

Branch: CSE

Semester: 5th Sem

Group: 2

Submitted To: Dr. VS Pandey

Department of Computer Science and Engineering

NATIONAL INSTITUTE OF TECHNOLOGY DELHI

________________2024________________

Theory -

1. Introduction to Shor's Algorithm

- Shor's Algorithm is a quantum algorithm for integer factorization, which efficiently finds the prime factors of a given integer N. Its efficiency comes from its ability to solve problems that are considered intractable on classical computers, making it a foundational quantum algorithm with implications for cryptography.

2. Period Finding in Shor's Algorithm

- A central component of Shor's Algorithm is its reliance on quantum period finding, which utilizes Quantum Phase Estimation (QPE) to determine the period of a specific modular exponentiation function.

3. Workflow of Scalable Shor's Algorithm

- **Quantum Circuit Preparation**: Initialize a circuit with minimal qubits for quantum period finding and apply controlled operations.
- **Modular Exponentiation**: Compute the modular exponentiation values on the quantum circuit, which represent the periodic function for factorization.
- **Iterative Phase Estimation**: Use IQPE or a reduced-qubit phase estimation approach to find the period accurately while minimizing qubit resources.
- **Classical Computation**: After retrieving the measurement outcomes, perform classical post-processing to identify factors based on the computed period.

4. Importance of Scalability in Shor's Algorithm

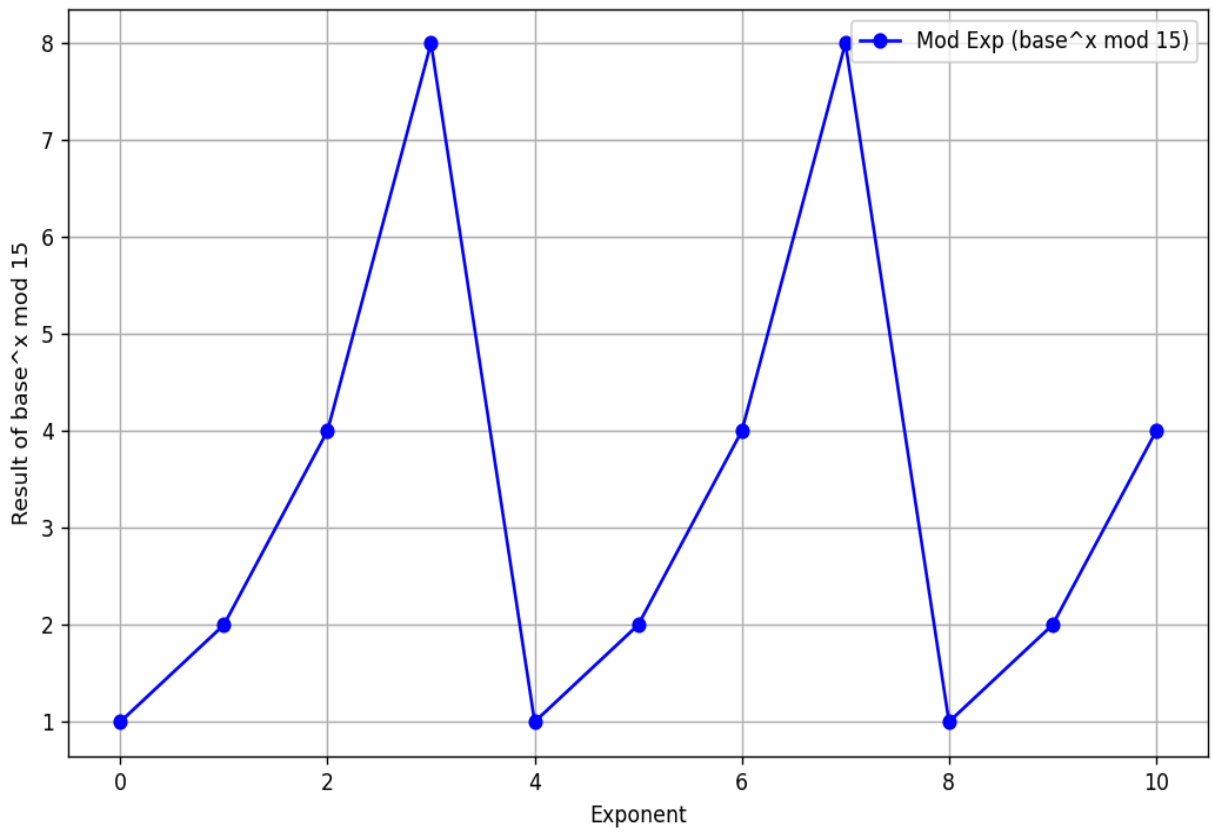
- The scalability of Shor's Algorithm is essential for making it applicable on practical quantum hardware with limited qubit resources.
- Scalable implementations, such as those utilizing IQPE, enable the algorithm to factor larger numbers with fewer qubits, preserving accuracy while reducing hardware demands.

Code (Modular Exponentiation) -

```
from qiskit import QuantumCircuit
import numpy as np
import matplotlib.pyplot as plt

# Function to perform modular exponentiation
def modular_exponentiation(base, exponent, modulus):
    return pow(base, exponent, modulus)

# Function to demonstrate modular exponentiation and plot the results
def visualize_modular_exponentiation(base, max_exponent, modulus):
    # Initialize a sample quantum circuit (for illustration only, no quantum operations)
    quantum_circuit = QuantumCircuit(4)

    # Compute modular exponentiation for each exponent value from 0 to max_exponent
    results = [modular_exponentiation(base, exp, modulus) for exp in range(max_exponent + 1)]
    print(f"Results of modular exponentiation (base={base}, modulus={modulus}): {results}")

    # Visualize results with a plot
    plt.figure(figsize=(10, 6))
    plt.plot(range(max_exponent + 1), results, marker='o', linestyle='-', color='blue',
             label=f"Mod Exp (base^x mod {modulus})")
    plt.xlabel("Exponent")
    plt.ylabel(f"Result of base^x mod {modulus}")
    plt.legend()
    plt.grid()

    # Show the plot
    plt.show()

    # Adding title below the plot
    plt.figtext(0.5, -0.05, f"Modular Exponentiation for base = {base}, modulus = {modulus}",
                wrap=True, horizontalalignment='center', fontsize=12)

# Example usage
visualize_modular_exponentiation(base=2, max_exponent=10, modulus=15)
```

Output -

```
Results of modular exponentiation (base=2, modulus=15): [1, 2, 4, 8, 1, 2, 4, 8, 1, 2, 4]
```

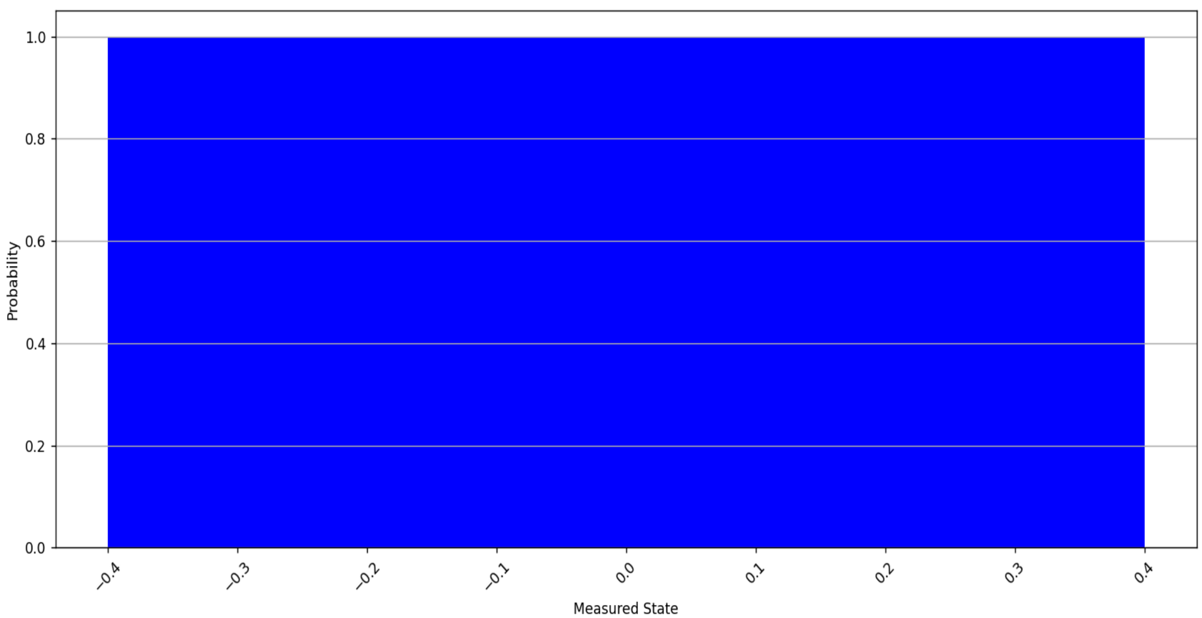


Code (Period Finding) -

```
from qiskit import QuantumCircuit
from qiskit.primitives import Sampler
import numpy as np
import matplotlib.pyplot as plt

# Define the Quantum Fourier Transform (QFT) function
def apply_qft(circuit, num_qubits):
    for qubit in range(num_qubits):
        circuit.h(qubit)
        for target in range(qubit + 1, num_qubits):
            circuit.cp(np.pi / 2 ** (target - qubit), target, qubit)
    circuit.barrier()

# Construct the quantum circuit for period finding
def perform_quantum_period_finding():
    num_qubits = 3
    circuit = QuantumCircuit(num_qubits)

    # Apply Hadamard gates to all qubits
    circuit.h(range(num_qubits))

    # Apply Quantum Fourier Transform
    apply_qft(circuit, num_qubits)

    # Measure all qubits
    circuit.measure_all()

    # Run the circuit using the Sampler to get the results
    sampler = Sampler()
    job = sampler.run(circuit)
    result = job.result()
    measurement_counts = result.quasi_dists[0]  # Retrieve measurement probabilities

    return measurement_counts

# Execute the period finding circuit
measurement_counts = perform_quantum_period_finding()
print("Measurement results for period finding:", measurement_counts)

# Plot the measurement outcomes
plt.figure(figsize=(8, 5))
plt.bar(measurement_counts.keys(), measurement_counts.values(), color='blue')
plt.xlabel("Measured State")
plt.ylabel("Probability")
plt.title("Measurement Results for Quantum Period Finding Circuit")
plt.xticks(rotation=45)
plt.grid(axis='y')
plt.show()
```

Output -

```
Measurement counts from period finding: {0: 0.999999999999999}
```



Measurement Results for Quantum Period Finding

Conclusion -

- **Efficiency-Scalability Trade-off**: Scalable Shor's Algorithm optimizes quantum resources, making it feasible to factorize larger integers by balancing quantum gate depth with the number of qubits. This scalability is essential for real-world applications, where efficient use of limited quantum resources is crucial.
- **Algorithmic Significance**:By enhancing factorization capabilities on quantum devices, Scalable Shor's Algorithm serves as a significant step toward practical applications in cryptography and computational number theory, offering a tangible impact on industries relying on large-scale factorization and secure encryption.