



ECSE 420: Parallel Computing

Final Project : Implementation of Shor's Algorithm on Qiskit and Cirq

Group 18

Venkata Satyanarayana Chivatvam - 260893291

Shaun Soobagrah - 260919063

Rania Ouassif - 260861621

Rakshitha Ravi - 260849587

December 10th 2022

Table of content

| | |
|--|-----------|
| Introduction to Quantum Computing | 3 |
| Introduction to Shor's Algorithm | 3 |
| Shor's algorithm protocol | 4 |
| Modular exponentiation | 4 |
| Quantum Fourier transform | 5 |
| IBM's Qiskit Implementation | 6 |
| Google's Cirq Implementation | 7 |
| Comparison and summary | 11 |
| Other providers | 11 |

Introduction to Quantum Computing

A quantum computer is a type of computer that uses the principles of quantum mechanics to store and process information. Unlike classical computers, which use bits to represent information as either 0s or 1s, quantum computers use quantum bits, or qubits, to represent information as a combination of 0s, 1s, and both 0s and 1s at the same time. This allows quantum computers to perform calculations much faster than classical computers, and to solve problems that are intractable on classical computers. Quantum computers use a variety of physical systems to store and manipulate qubits, such as atoms, photons, and electrons. Manipulation of qubits to achieve the desired results can be done by applying certain gates in an order.

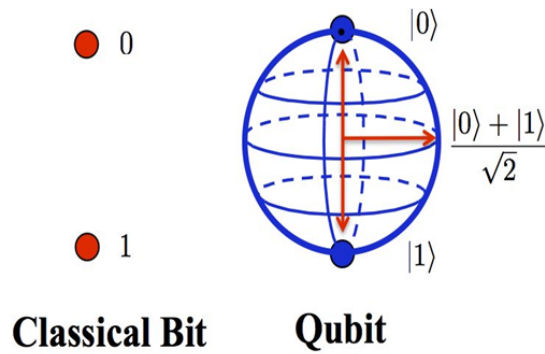


Fig1: Classical bit vs Qubit

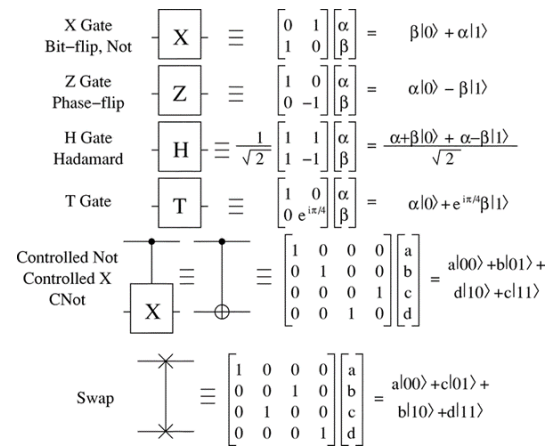


Fig2: Quantum Gates

Introduction to Shor's Algorithm

Shor's algorithm is a quantum algorithm for efficiently factoring large numbers, a problem that is believed to be difficult for classical computers. The algorithm was developed by mathematician Peter Shor in 1994. Shor's algorithm works by using the properties of quantum mechanics to simultaneously examine multiple potential factors of a number, allowing it to quickly find the correct factors with a high probability. This can be done much faster than with classical algorithms, which must test each potential factor individually.

Current fastest classical algorithm to factor a number is fast fourier transform with time complexity of

$$O\left(e^{1.9(\log N)^{1/3}(\log \log N)^{2/3}}\right)$$

Whereas Shor's algorithm's theoretical time complexity is of polynomial time portraying exponential speedup:

$$O((\log N)^2 (\log \log N) (\log \log \log N))$$

Shor's algorithm has important implications for the field of cryptography, as many of the commonly used methods for encrypting and protecting sensitive information such as RSA, The Finite Field, and The Elliptic Curve Diffie-Hellman key exchanges rely on the difficulty of factoring large numbers. The ability to quickly factor these numbers could potentially make these encryption methods less secure.

However, the full potential of Shor's algorithm has not yet been realized, as it requires a large and complex quantum computer to run effectively. As the technology for building such computers continues to advance, it is possible that Shor's algorithm will become more practical and widely used in the future.

Shor's algorithm protocol

We are trying to find the prime factors(P and Q) of a number N ($N = P \cdot Q$):

- Pick a number “a” that is coprime with N
- Find the order “r” of a function $a^r \pmod{N}$, Smallest r such that $a^r = 1 \pmod{N}$
- If r is even:
- $X = a^{r/2} \pmod{N}$
- If $X + 1 \not\equiv 0 \pmod{N}$:
- $\{P, Q\} = \{\gcd(x+1, N), \gcd(x-1, N)\}$ (Great!)
- Else choose another “a” and repeat the above steps

Modular exponentiation

In Shor's algorithm, modular exponentiation is used as part of the process for efficiently factoring large numbers. The algorithm uses quantum computers to simultaneously examine multiple potential factors of a number, and modular exponentiation is used to quickly test each of these potential factors to determine if they are correct.

For example, classically, in the equation " $5^7 \pmod{11}$ ", the base is 5, the exponent is 7, and the modulus is 11. To perform the modular exponentiation, we would first calculate 5^7 (which equals 78125), and then divide that result by 11 and take the remainder. In this case, the remainder would be 9, so the final result of the modular exponentiation would be 9.

To perform the modular exponentiation in Shor's algorithm, the quantum computer first calculates the value of the base raised to the exponent modulo the modulus. This value is then used to determine if the potential factor is correct or not. If the result of the modular exponentiation is 1, then the potential factor

is not correct and the algorithm moves on to the next potential factor. If the result is not 1, then the potential factor is likely to be correct and the algorithm proceeds to the next step to confirm it. Modular exponentiation is an essential part of Shor's algorithm because it allows the quantum computer to quickly test multiple potential factors at the same time, making the algorithm much more efficient than classical methods for factoring large numbers.

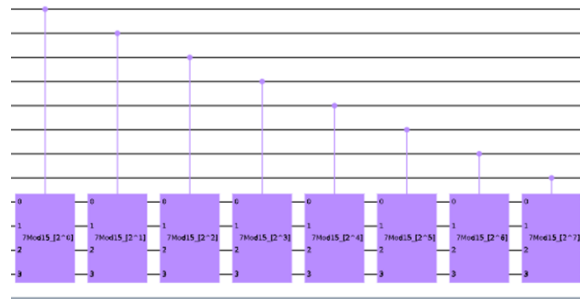


Fig3: quantum circuit

| Input Register | $7^a \text{ Mod } 15$ | Output Register |
|----------------|-----------------------|-----------------|
| $ 0\rangle$ | $7^0 \text{ Mod } 15$ | 1 |
| $ 1\rangle$ | $7^1 \text{ Mod } 15$ | 7 |
| $ 2\rangle$ | $7^2 \text{ Mod } 15$ | 4 |
| $ 3\rangle$ | $7^3 \text{ Mod } 15$ | 13 |
| $ 4\rangle$ | $7^4 \text{ Mod } 15$ | 1 |
| $ 5\rangle$ | $7^5 \text{ Mod } 15$ | 7 |
| $ 6\rangle$ | $7^6 \text{ Mod } 15$ | 4 |
| $ 7\rangle$ | $7^7 \text{ Mod } 15$ | 13 |

Fig4: output of quantum circuit

Quantum Fourier transform

The quantum Fourier transform (QFT) is a mathematical operation used in quantum computing and quantum algorithms, such as Shor's algorithm for factoring large numbers. To implement the QFT, a quantum computer must be able to perform the following operations:

1. Initialize qubits in the computational basis state
2. Apply a Hadamard gate to each qubit
3. Apply controlled phase shift gates to each qubit
4. Measure the qubits in the computational basis

To initialize the qubits, the quantum computer must prepare them in the computational basis state, which is a specific state of the qubits that is required for the QFT to work properly. This can be done by setting the qubits to 0 or 1, depending on the specific implementation of the QFT. Next, the quantum computer applies a Hadamard gate to each qubit. The Hadamard gate is a quantum gate that acts on a single qubit and changes its state in a specific way that is necessary for the QFT. After the Hadamard gates have been applied, the quantum computer applies controlled phase shift gates to each qubit. These gates are similar to the Hadamard gates, but they have a slightly different effect on the qubits. Finally, the quantum computer measures the qubits in the computational basis. This allows it to determine the result of the QFT, which is used in the next steps of the algorithm. Overall, implementing the QFT on a quantum computer requires a combination of precise control over the qubits and the ability to apply the necessary quantum gates. As the technology for building quantum computers continues to advance, it is likely that the QFT will become easier to implement and more widely used in quantum algorithms.

Below is the aggregation of quantum gates that perform QFT. An inverse QFT can be achieved by just inverting gates applied on qubits from bottom to top.

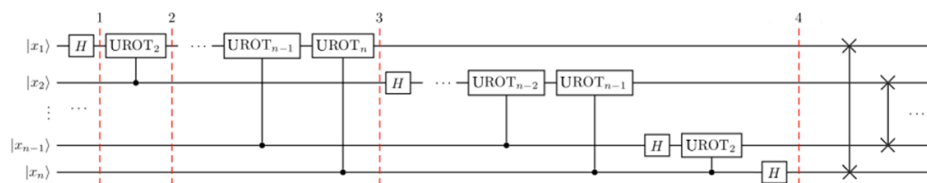


Fig5: Quantum Fourier Transform

IBM's Qiskit Implementation

This is the implementation by IBM of Shor's algorithm. We start with our number N for which we are trying to find the prime factors of. We take a random number a and we input it into the method `qpe_amod`. This method simulates the quantum circuit. Different method calls in this method simulate different parts of the circuit. The method is shown below:

```

def qpe_amod15(a):
    n_count = 8
    qc = QuantumCircuit(4+n_count, n_count)
    for q in range(n_count):
        qc.h(q) # Initialize counting qubits in state |+>
    qc.x(3+n_count) # And auxiliary register in state |1>
    for q in range(n_count): # Do controlled-U operations
        qc.append(c_amod15(a, 2**q),
                  [q] + [i+n_count for i in range(4)])
    qc.append(qft_dagger(n_count), range(n_count)) # Do inverse-QFT
    qc.measure(range(n_count), range(n_count))
    # Simulate Results
    aer_sim = Aer.get_backend('aer_simulator')
    # Setting memory=True below allows us to see a list of each sequential reading
    t_qc = transpile(qc, aer_sim)
    qobj = assemble(t_qc, shots=1)
    result = aer_sim.run(qobj, memory=True).result()
    readings = result.get_memory()
    print("Register Reading: " + readings[0])
    phase = int(readings[0],2)/(2**n_count)
    print("Corresponding Phase: %f" % phase)
    return phase

```

Fig6: qpe_amod method

The qpe_amod15 function creates a circuit with all the required gates to find out the periodicity. C_amod15 is the function which creates the unitary operator gates that simulate the modulo arithmetic. Upon that, qft_dagger is applied to get the eigenvalue with respect to the qubits from which we derive the eigenphase. Those methods are present in the source code

To get the periodicity from the eigenphase, we use the continued fraction method to get the periodicity. Upon that, we use the classical computation as defined in the algorithm before to get the factors of the given number. The implementation is shown below:

```

N = 15
np.random.seed(1) # This is to make sure we get reproduceable results
a = randint(2, 15)
factor_found = False
attempt = 0
while not factor_found:
    attempt += 1
    print("\nAttempt %i:" % attempt)
    phase = qpe_amod15(a) # Phase = s/r
    frac = Fraction(phase).limit_denominator(N) # Denominator should (hopefully!) tell us r
    r = frac.denominator
    print("Result: r = %i" % r)
    if phase != 0:
        # Guesses for factors are gcd(x^{r/2} ± 1, 15)
        guesses = [gcd(a**(r//2)-1, N), gcd(a**(r//2)+1, N)]
        print("Guessed Factors: %i and %i" % (guesses[0], guesses[1]))
        for guess in guesses:
            if guess not in [1,N] and (N % guess) == 0: # Check to see if guess is a factor
                print("factor found: %i" % guess)
                factor_found = True

```

Fig7: IBM implementation of shor' algorithm

Google's Cirq Implementation

Cirq is an open source library which is used for writing, optimizing and manipulating quantum circuits and has a lot of useful abstractions that can be used like the print function we see below. We are using their Shors algorithm implementation .

```
▶ """Example of the quantum circuit for period finding."""  
n = 15  
x = 7  
circuit = make_order_finding_circuit(x, n)  
print(circuit)
```

Fig8.

Quantum order finding is essentially quantum phase estimation with unitary operation U that computes the modular exponential function. We have our main method of `find_factor` as follows


```

def find_factor(
    n: int,
    order_finder: Callable[[int, int], Optional[int]] = quantum_order_finder,
    max_attempts: int = 30
) -> Optional[int]:

    # If the number is prime, there are no non-trivial factors.
    if sympy.isprime(n):
        print("n is prime!")
        return None

    # If the number is even, two is a non-trivial factor.
    if n % 2 == 0:
        return 2
    c = find_factor_of_prime_power(n)
    if c is not None:
        return c

    for _ in range(max_attempts):
        # Choose a random number between 2 and n - 1.
        x = random.randint(2, n - 1)

        # Most likely x and n will be relatively prime.
        c = math.gcd(x, n)
        if 1 < c < n:
            return c
        r = order_finder(x, n)

        if r is None:
            continue

        if r % 2 != 0:
            continue
        y = x**(r // 2) % n
        assert 1 < y < n
        c = math.gcd(y - 1, n)
        if 1 < c < n:
            return c

    print(f"Failed to find a non-trivial factor in {max_attempts} attempts.")
    return None

```

Fig9.

And the supporting methods called here can be found below :

The quantum part of Shor's algorithm is just phase estimation with the unitary corresponding to the modular exponential gate. After this we run the print circuit and the circuit can be visualised as seen in the next step.

```

def make_order_finding_circuit(x: int, n: int) -> cirq.Circuit:

    L = n.bit_length()
    target = cirq.LineQubit.range(L)
    exponent = cirq.LineQubit.range(L, 3 * L + 3)

    # Create a ModularExp gate sized for these registers.
    mod_exp = ModularExp([2] * L, [2] * (2 * L + 3), x, n)

    return cirq.Circuit(
        cirq.X(target[L - 1]),
        cirq.H.on_each(*exponent),
        mod_exp.on(*target, *exponent),
        cirq.qft(*exponent, inverse=True),
        cirq.measure(*exponent, key='exponent'),
    )

```

Fig10.

In the figure below we see the circuit which was printed using the print function (shown above) which is

basically a diagram that shows us how the computing is being performed and also the number of Qubits being used. Each input (marked 0 to 14) in our circuit represents 1 Qubit, which shows that a total of 15 qubits were used to perform our calculation. We can also see that the gates used are shown, for example the H represents the Hadamard gate.

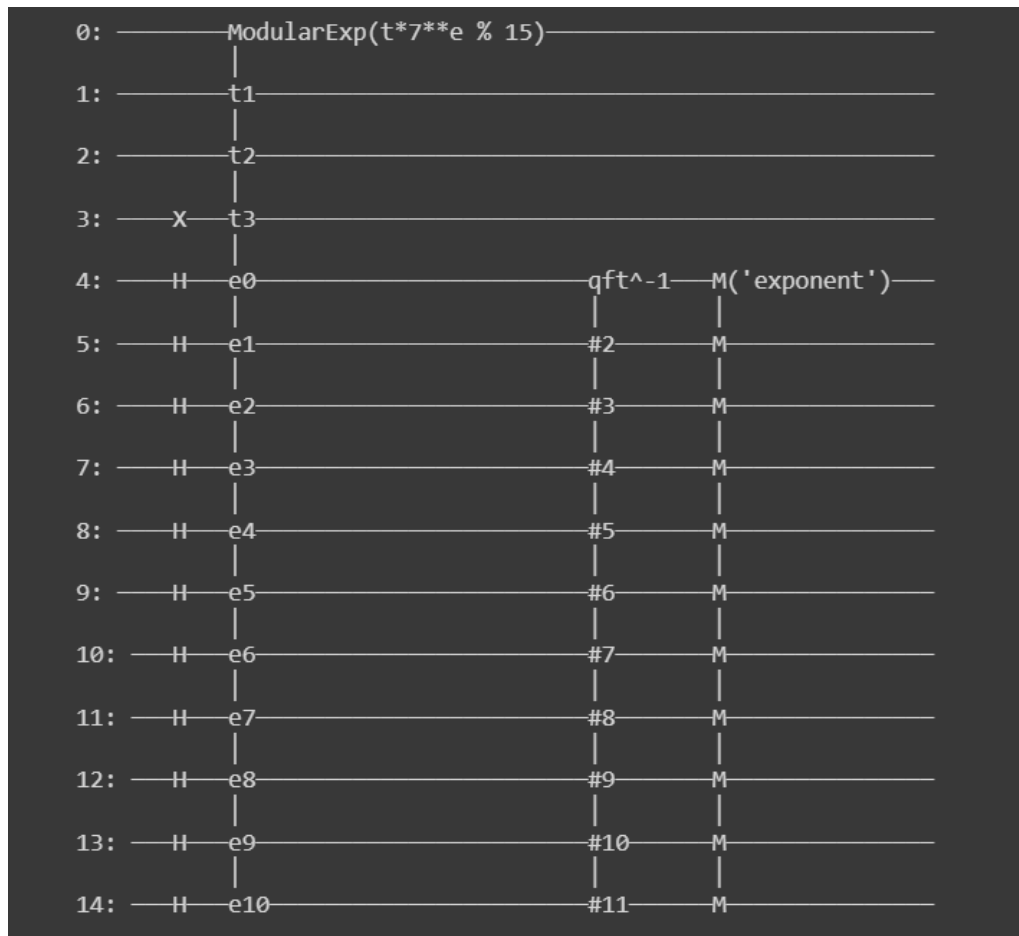


Fig11: Cirq Implementation circuit diagram

We then print the measurements

```
[ ] """Measuring Shor's period finding circuit."""
circuit = make_order_finding_circuit(x=5, n=6)
res = cirq.sample(circuit, repetitions=8)

print("Raw measurements:")
print(res)

print("\nInteger in exponent register:")
print(res.data)
```

Raw measurements:
exponent=00111001, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000

Integer in exponent register:
exponent
0 0
1 0
2 256
3 256
4 256
5 0
6 0
7 256

Fig12: results from google's implementation

A streamlined function called order finder below creates the circuit, executes it, and processes the measurement result and sends the result to the main factor_finder method.

```
def quantum_order_finder(x: int, n: int) -> Optional[int]:
    # Check that the integer x is a valid element of the multiplicative group
    # modulo n.
    if x < 2 or n <= x or math.gcd(x, n) > 1:
        raise ValueError(f'Invalid x={x} for modulus n={n}.')

    # Create the order finding circuit.
    circuit = make_order_finding_circuit(x, n)

    # Sample from the order finding circuit.
    measurement = cirq.sample(circuit)

    # Return the processed measurement result.
    return process_measurement(measurement, x, n)
```

Fig13.

We then use a classic post processing method with the continued fractions algorithm to return r

```
def process_measurement(result: cirq.Result, x: int, n: int) -> Optional[int]:

    # Read the output integer of the exponent register.
    exponent_as_integer = result.data["exponent"][0]
    exponent_num_bits = result.measurements["exponent"].shape[1]
    eigenphase = float(exponent_as_integer / 2**exponent_num_bits)

    # Run the continued fractions algorithm to determine f = s / r.
    f = fractions.Fraction.from_float(eigenphase).limit_denominator(n)

    # If the numerator is zero, the order finder failed.
    if f.numerator == 0:
        return None

    # Else, return the denominator if it is valid.
    r = f.denominator
    if x**r % n != 1:
        return None
    return r
```

Fig14.

We ran this cell to call the `find_factor` method and run our algorithm for 2 numbers 15 and 35 and the factors were printed as shown below

```
# Number to factor
n = 35

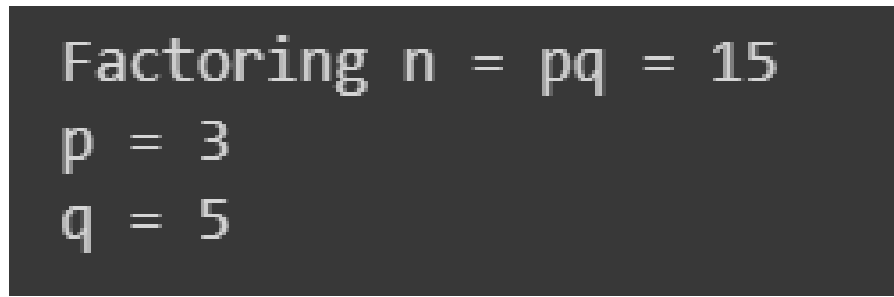
# Attempt to find a factor
p = find_factor(n, order_finder=quantum_order_finder)
q = n // p

print("Factoring n = pq =", n)
print("p =", p)
print("q =", q)
```

Fig15.

```
Factoring n = pq = 35
p = 5
q = 7
```

Fig16.



```
Factoring n = pq = 15
p = 3
q = 5
```

Fig17.

Comparison and summary

When comparing Qiskit and Cirq, we find that Cirq has more abstractions than Qiskit. This is due to the fact that Cirq has more built in functions whereas when using Qiskit, we have to write the code for those functions, an example is the previously mentioned IQT.

Additionally, Qiskit simulators are faster and more robust compared to Cirq simulators. One possible reason for this is the fact that Qiskit has been in the Quantum field longer than Google.

To conclude, for us to achieve any speedup using quantum computers, we would need more qubits. For example, Shor's algorithm requires $n^2 + 4$ qubits to give an ideal output, with n being the number of bits of the given number. So, as of now, there is no scenario where QC performed better than classical computers. However, there is more innovation now, such as IBM who recently released its 433 qubit based quantum computer, called Osprey

Other providers

There are other providers currently available in the market. First, there is Amazon Braket which provides a platform for some quantum computing companies like D-wave to host their services but they do not provide a SDK.

Then, Xanadu provides SDK along with quantum computers. Their key research is related to Quantum Machine learning called PennyLane and quantum photonics (strawberry field).

Finally, Q-ctrl and qBraid are some of the other prominent quantum computing providers.