qpu

April 28, 2023

1 Welcome to the Quantum Parallel Universe

1.1 Initial Setup

1.1.1 Imports

[1]:
```
import math
from IPython.display import Latex
from qiskit import Aer, execute, QuantumCircuit
from qiskit.circuit import Qubit
from qiskit.visualization import plot_histogram
```

1.1.2 Globals

Manually Managed Variables

[2]:
```
# number of qubits: int
N = 16

# shots: int
shots = 2**16

# Quantum Simulator Object
simulator = Aer.get_backend("aer_simulator")
```

Automatically Managed Variables

[3]:
```
# linear GHZ container
linear = {
  'circuit': None,
  'job': None,
  'result': None,
  'time': None,
  'error': { '0': None, '1': None }
}

# logarithmic GHZ container
log = {
  'circuit': None,
```

```
    'job': None,
    'result': None,
    'time': None,
    'error': { '0': None, '1': None }
}

# ideal shots per state
isps = shots / 2
```

1.2 Generate $|\mathrm{GHZ}_N\rangle$ Circuits1

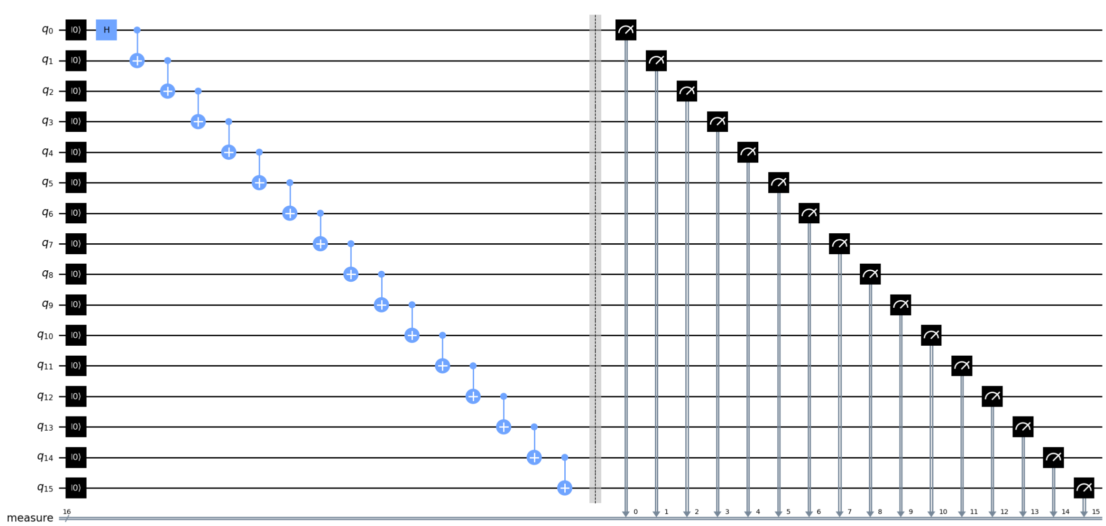
1.2.1 Generate Linear Time Complexity Circuits for $|\mathrm{GHZ}_N\rangle$

[4]:
```
def linear_complexity_GHZ(N: int) -> QuantumCircuit:
    if not isinstance(N, int):
        raise TypeError("Only integer arguments accepted.")
    if N < 1:
        raise ValueError("There must be one or more qubits.")

    c = QuantumCircuit(N)
    for i in range(N):
        c.reset(i)
    c.h(0)
    for i in range(1, N):
        c.cx(i-1, i)
    c.measure_active()
    return c
```

[5]:
```
linear['circuit'] = linear_complexity_GHZ(N)
linear['circuit'].draw(output='mpl', fold=-1)
```

[5]:



1.2.2 Generate Logaritmic Complexity Circuits for $|\mathrm{GHZ}_{2^m}\rangle$

[6]:
```
def _log_complexity_GHZ(m: int) -> QuantumCircuit:
  if not isinstance(m, int):
    raise TypeError("Only integer arguments accepted.")
  if m < 0:
    raise ValueError("`m` must be at least 0 (evaluated 2^m).")

  if m == 0:
    c = QuantumCircuit([Qubit()])
    c.reset(0)
    c.h(0)
  else:
    c = _log_complexity_GHZ(m - 1)
    for i in range(c.num_qubits):
      c.add_bits([Qubit()])
      new_qubit_index = c.num_qubits - 1
      c.reset(new_qubit_index)
      c.cx(i, new_qubit_index)
  return c
```

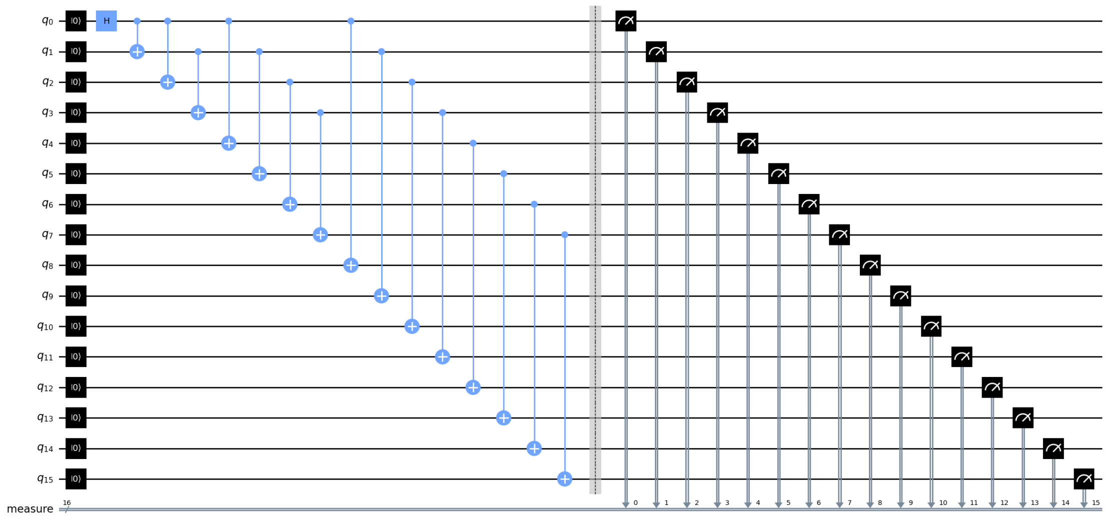
1.2.3 Generate Logaritmic Complexity Circuits for $|\mathrm{GHZ}_N\rangle$

[7]:
```
def log_complexity_GHZ(N: int) -> QuantumCircuit:
  if not isinstance(N, int):
    raise TypeError("Only an integer argument is accepted.")
  if N < 1:
    raise ValueError("There must be one or more qubits.")

  m = math.ceil(math.log2(N))
  num_qubits_to_erase = 2**m - N
  old_circuit = _log_complexity_GHZ(m=m)
  new_num_qubits = old_circuit.num_qubits - num_qubits_to_erase
  new_circuit = QuantumCircuit(new_num_qubits)
  for gate in old_circuit.data:
    qubits_affected = gate.qubits
    if all(old_circuit.find_bit(qubit).index < new_num_qubits for qubit in␣
↪qubits_affected):
      new_circuit.append(gate[0], [old_circuit.find_bit(qubit).index for qubit␣
↪in qubits_affected])
  new_circuit.measure_active()
  return new_circuit
```

[8]:
```
log['circuit'] = log_complexity_GHZ(N)
log['circuit'].draw(output='mpl', fold=-1)
```

[8]:



1.3 Quantum Simulation & Results

1.3.1 Create Simulator Jobs

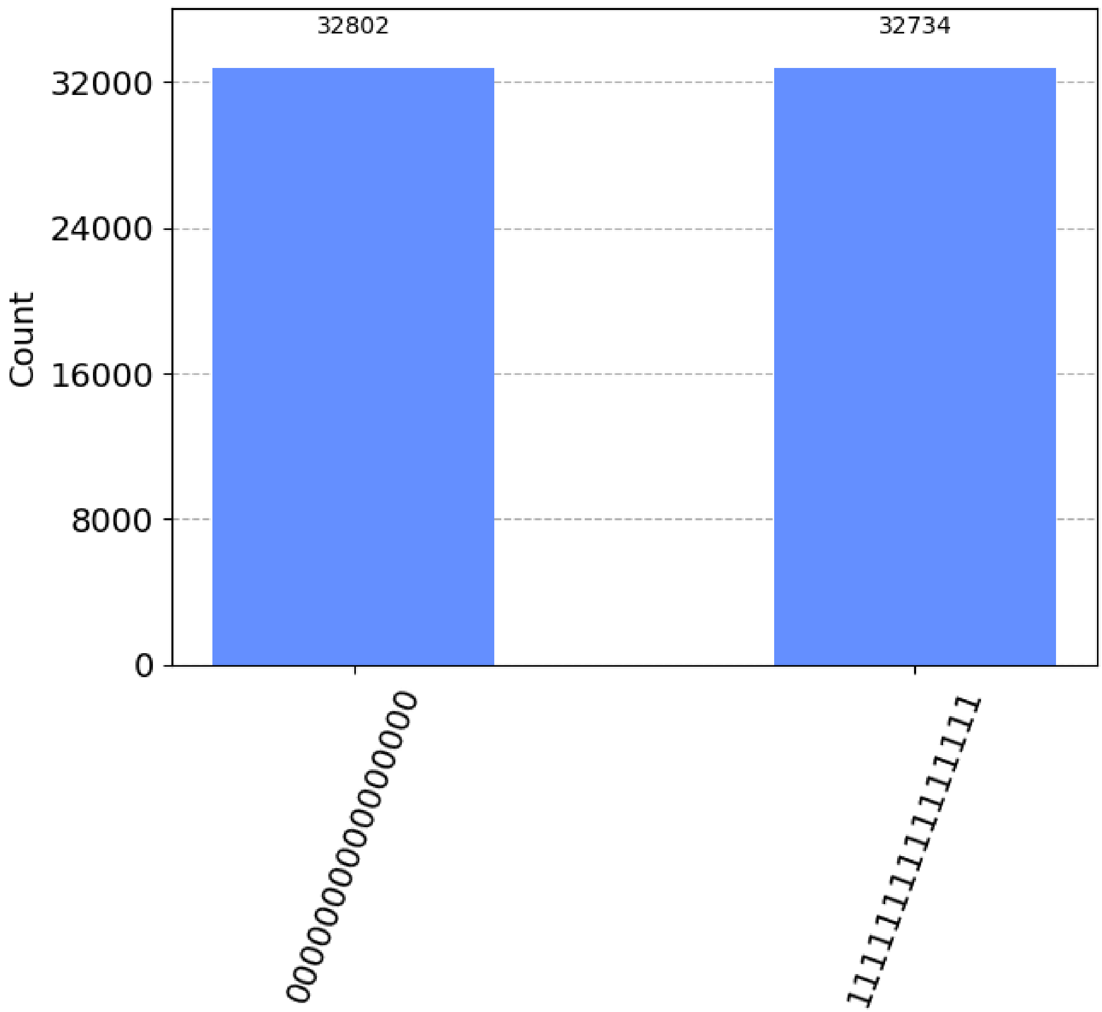
[9]:
```
linear['job'] = execute(linear['circuit'], simulator, shots=shots)
log['job'] = execute(log['circuit'], simulator, shots=shots)
```

1.3.2 Execution Histograms

Linear

[10]:
```
linear['result'] = linear['job'].result()
plot_histogram(linear['result'].get_counts())
```

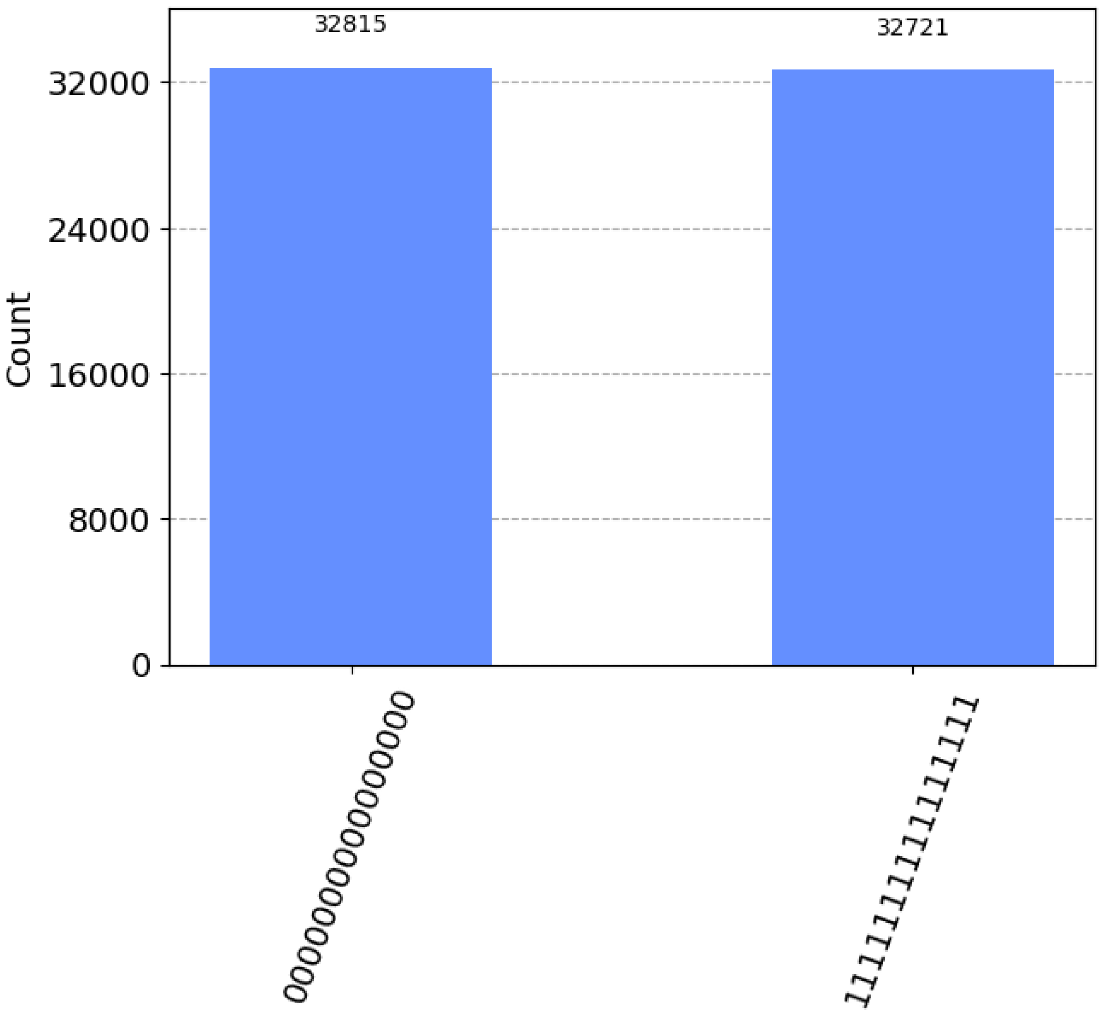
[10]:



Logaritmic

[11]:
```
log['result'] = log['job'].result()
plot_histogram(log['result'].get_counts())
```

[11]:



1.4 Error Analysis

1.4.1 Linear Error Percentage

State $|0\rangle$

[12]:
```
linear['error']['0'] = abs((linear['result'].get_counts()['0' * N] - isps) /␣
↪isps)
Latex(f"""\\begin{{equation*}}{linear['error']['0'] *␣
↪100}\%\\end{{equation*}}""")
```

[12]:

$$0.103759765625\%$$

State $|1\rangle$

[13]:
```
linear['error']['1'] = abs((linear['result'].get_counts()['1' * N] - isps) /␣
 ↪isps)
Latex(f"""\\begin{{equation*}}{linear['error']['1'] *␣
 ↪100}\%\\end{{equation*}}""")
```

[13]:

$$0.103759765625\%$$

1.4.2 Logarithmic Error Percentage

State $|0\rangle$

[14]:
```
log['error']['0'] = abs((log['result'].get_counts()['0' * N] - isps) / isps)
Latex(f"""\\begin{{equation*}}{log['error']['0'] * 100}\%\\end{{equation*}}""")
```

[14]:

$$0.1434326171875\%$$

State $|1\rangle$

[15]:
```
log['error']['1'] = abs((log['result'].get_counts()['1' * N] - isps) / isps)
Latex(f"""\\begin{{equation*}}{log['error']['1'] * 100}\%\\end{{equation*}}""")
```

[15]:

$$0.1434326171875\%$$

1.5 Speed-Up Analysis

1.5.1 Run-Times

Linear

[16]:
```
linear['time'] = linear['result'].time_taken
Latex(f"""\\begin{{equation*}}{linear['time']}\\space\\text{{seconds}}\\end{{equation*}}""")
```

[16]:

$$1.291309118270874\text{seconds}$$

Log

[17]:
```
log['time'] = log['result'].time_taken
Latex(f"""\\begin{{equation*}}{log['time']}\\space\\text{{seconds}}\\end{{equation*}}""")
```

[17]:

$$0.635613203048706\text{seconds}$$

1.5.2 Amdahl's Law

Parallel Portion

[18]:

```
S_latency = linear['time'] / log['time']
P = (N * (1 - (1 / S_latency))) / (N - 1)
Latex(f"""\\begin{{equation*}}
    P = \\dfrac{{N\\left(1 - \\dfrac{{1}}{{S_\\text{{latency}}}}\\right)}}{{N -
 ↪1}} = \\dfrac{{{N}\\left(1 - \\dfrac{{1}}{{{S_latency}}}\\right)}}{{{N -
 ↪1}}} = {P * 100}\%
    \\end{{equation*}}
  """)
```

[18]:

$$P = \dfrac{N\left(1 - \dfrac{1}{S_\text{latency}}\right)}{N - 1} = \dfrac{16\left(1 - \dfrac{1}{2.031595807131028}\right)}{15} = 54.162784599052614\%$$

Sequential Portion

[19]:

```
S_EQ = 1 - P
Latex(f"""\\begin{{equation*}}S_\\text{{EQ}} = 1 - P = {S_EQ *
 ↪100}\%\\end{{equation*}}""")
```

[19]:

$$S_\text{EQ} = 1 - P = 45.8372154009473 86\%$$

1.6 References

1. arXiv:1807.05572