qpu

May 1, 2023

1 Welcome to the Quantum Parallel Universe

1.1 Initial Setup

1.1.1 Immutable Imports

[1]:
```
import math
import re
from IPython.display import Latex
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.circuit import Qubit
```

1.1.2 Globals

Manually Managed Variables & Imports

[2]:
```
# number of qubits: int
N = 15

# IBMQ Mock Backend (https://qiskit.org/documentation/stable/0.42/apidoc/
 ↪providers_fake_provider.html#fake-v1-backends)
from qiskit.providers.fake_provider import FakeCairo
backend = { 'device': FakeCairo() }
```

Automatically Managed Variables

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

# logarithmic GHZ container
log = {
```

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

# ideal shots per state
isps = 512

# parallel sections
init = [ 0, 1, 2 ]
i = len(init)
k = 1
while len(init) <= N:
  init += [i] * 2**k
  i += 1
  k += 1
s = init[N]

# IBMQ Mock Backend
if N > 0:
  backend['name'] = re.sub(r'(_|fake|v\d)', ' ', backend['device'].backend_name.
 ↪lower()).title()
  backend['num_qubits'] = backend['device'].configuration().num_qubits
  backend['simulator'] = AerSimulator.from_backend(backend['device'])
else:
  raise RuntimeError(msg=f"Invalid N={N}, must be 0 < N <␣
 ↪{backend['num_qubits']}")
```

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

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

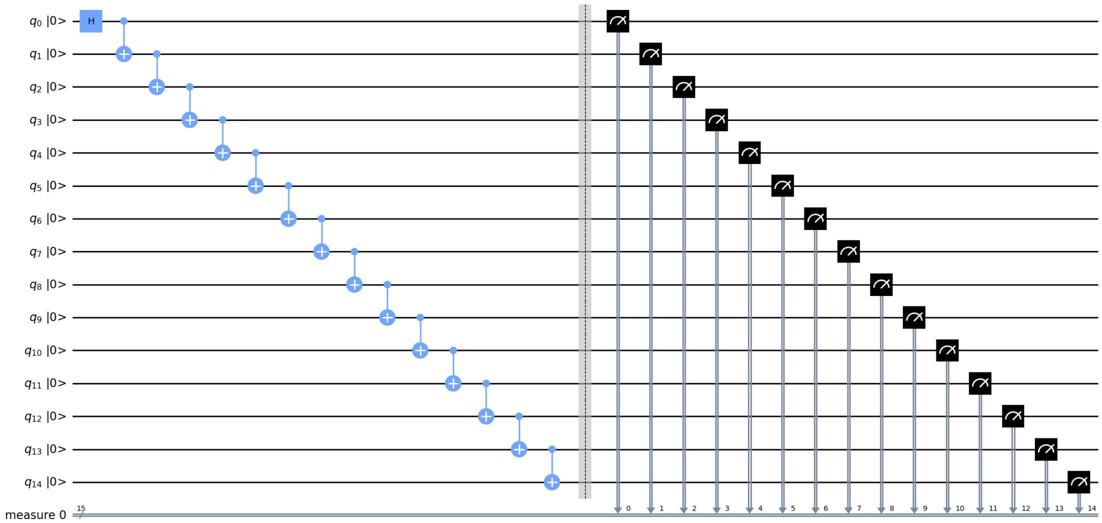
[4]:
```
def linear_complexity_GHZ(N: int) -> QuantumCircuit:
  if not isinstance(N, int):
    raise TypeError("Only integer arguments accepted.")
  if N < 1:
    raise ValueError("There must be one or more qubits.")

  c = QuantumCircuit(N)
  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, initial_state=True)
```

[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.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.cx(i, new_qubit_index)
  return c
```

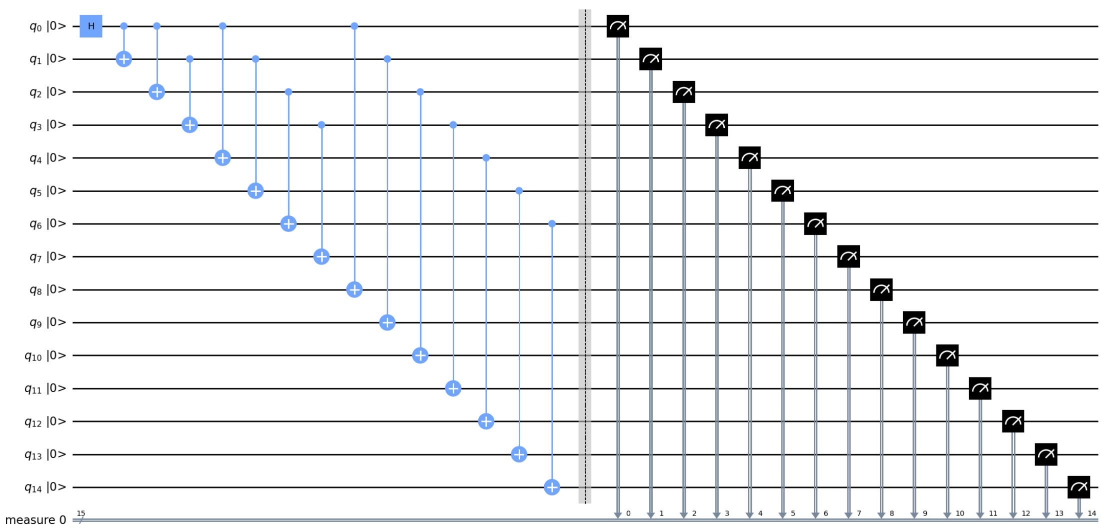
1.2.3 Generate Logaritmic Complexity Circuits for $|\mathrm{GHZ}_N\rangle$

[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, initial_state=True)
```

[8]:



1.3 Quantum Simulation

1.3.1 Device

[9]:
```
Latex(f"""\\begin{{equation*}}
          \\text{{{backend['name']} ({backend['num_qubits']} qubits)}}
          \\end{{equation*}}""")
```

[9]:

$$\text{Cairo (27 qubits)}$$

1.3.2 Transpile Circuits

[10]:
```
linear['transpiled'] = transpile(linear['circuit'], backend['simulator'],␣
↪scheduling_method="alap", optimization_level=0)
```

[11]:
```
log['transpiled'] = transpile(log['circuit'], backend['simulator'],␣
↪scheduling_method="alap", optimization_level=0)
```

1.3.3 Run Simulations

[12]:
```
linear['job'] = backend['device'].run(linear['transpiled'])
```

[13]:
```
log['job'] = backend['device'].run(log['transpiled'])
```

1.3.4 Block for Results

[14]:
```
linear['result'] = linear['job'].result()
```

[15]:
```
log['result'] = log['job'].result()
```

1.4 Error Analysis

1.4.1 Linear Error Percentage

State $|0\rangle$

[16]:
```
try:
  linear['error']['0'] = abs((linear['result'].get_counts()['0' * N] - isps) /␣
↪isps)
except KeyError:
  linear['error']['0'] = 1
Latex(f"""\\begin{{equation*}}{linear['error']['0'] *␣
↪100}\%\\end{{equation*}}""")
```

[16]:

$$62.890625\%$$

State $|1\rangle$

[17]:
```
try:
  linear['error']['1'] = abs((linear['result'].get_counts()['1' * N] - isps) /␣
 ↪isps)
except KeyError:
  linear['error']['1'] = 1
Latex(f"""\\begin{{equation*}}{linear['error']['1'] *␣
 ↪100}\%\\end{{equation*}}""")
```

[17]:

$$91.796875\%$$

1.4.2 Logarithmic Error Percentage

State $|0\rangle$

[18]:
```
try:
  log['error']['0'] = abs((log['result'].get_counts()['0' * N] - isps) / isps)
except KeyError:
  log['error']['0'] = 1
Latex(f"""\\begin{{equation*}}{log['error']['0'] * 100}\%\\end{{equation*}}""")
```

[18]:

$$60.546875\%$$

State $|1\rangle$

[19]:
```
try:
  log['error']['1'] = abs((log['result'].get_counts()['1' * N] - isps) / isps)
except KeyError:
  log['error']['1'] = 1
Latex(f"""\\begin{{equation*}}{log['error']['1'] * 100}\%\\end{{equation*}}""")
```

[19]:

$$89.6484375\%$$

1.5 Speed-Up Analysis

1.5.1 Run-Times

Linear

[20]:
```
linear['time'] = linear['result'].time_taken
Latex(f"""\\begin{{equation*}}{linear['time']}\\space\\text{{seconds}}\\end{{equation*}}""")
```

[20]:

$$63.719013929367065 \text{seconds}$$

Log

[21]:
```python
log['time'] = log['result'].time_taken
Latex(f"""\\begin{{equation*}}{log['time']}\\space\\text{{seconds}}\\end{{equation*}}""")
```

[21]:

$$8.857570171356201\text{seconds}$$

1.5.2 Amdahl's Law

Parallel Portion

[22]:
```python
S_latency = linear['time'] / log['time']
P = (N * (1 - (1 / S_latency))) / (N - 1)
Latex(f"""\\begin{{equation*}}
    P = \\dfrac{{s\\left(1 - \\dfrac{{1}}{{S_\\text{{latency}}}}\\right)}}{{s -␣
↪1}} = \\dfrac{{{s}\\left(1 - \\dfrac{{1}}{{{S_latency}}}\\right)}}{{{s -␣
↪1}}} = {P * 100}\%
    \\end{{equation*}}
  """)
```

[22]:

$$P = \dfrac{s\left(1 - \dfrac{1}{S_\text{latency}}\right)}{s - 1} = \dfrac{5\left(1 - \dfrac{1}{7.193735154977713}\right)}{4} = 92.24894531059856\%$$

Sequential Portion

[23]:
```python
S_EQ = 1 - P
Latex(f"""\\begin{{equation*}}S_\\text{{EQ}} = 1 - P = {S_EQ *␣
↪100}\%\\end{{equation*}}""")
```

[23]:

$$S_\text{EQ} = 1 - P = 7.751054689401449\%$$

1.6 References

1. arXiv:1807.05572