量子計算入門ハンズオン

第3章 本物の量子計算機を使ってみよう

このノートでは、QURI Partsから実際の量子計算機を使用する方法を解説します。最後には、実際に量子計算機にジョブを投げてみることにしましょう。

無料で使用できる実機は、ジョブの実行までの待ち時間もあり、結果が返ってくるまでに数時間かかる可能性があります。そこでこのノートは他の講義ノートとは分割し、実行したまましばらく放置できるようにしました。

(お知らせの通り、IBM Quantumの無料アカウントを作成し、トークンを取得できるようにしておいてください。)

環境構築

繰り返しになってしまいますが、こちらのノートでもQURI Partsをインストールします。
(Shift + Enterで実行できます。)

In [ ]:
```
!python -m pip install quri-parts[qulacs,qiskit]
```

```
   Successfully installed ...
```

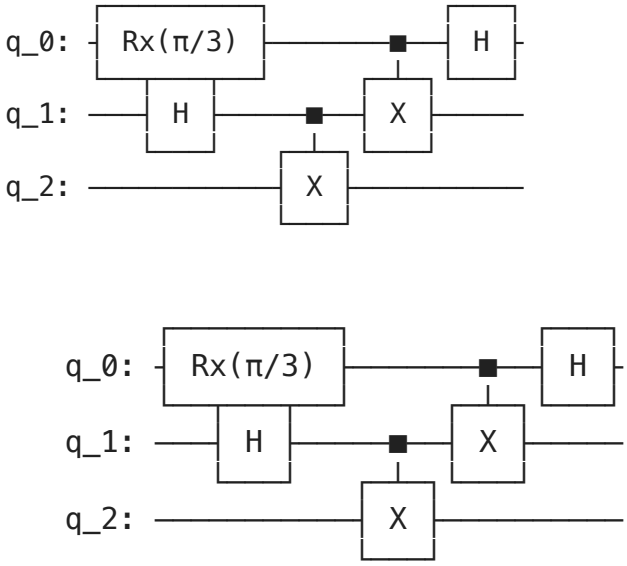
インストールが終わって上のようなメッセージが表示されたら、確認のために量子回路を作ってみましょう。

In [ ]:
```
from math import pi
from quri_parts.circuit import X, RX, CNOT, QuantumCircuit
from quri_parts.qiskit.circuit import convert_circuit

from quri_parts.circuit import QuantumCircuit

circuit = QuantumCircuit(3)
circuit.add_RX_gate(0, pi/3)
circuit.add_H_gate(1)
circuit.add_CNOT_gate(1, 2)
circuit.add_CNOT_gate(0, 1)
circuit.add_H_gate(0)

print(convert_circuit(circuit))
```



上のような回路が表示されたことを確認してください。

実機でのサンプリングの方法

QURI Partsはプラットフォーム独立になるように設計されているため、シミュレータに対しても現実の量子計算機に対しても、ほとんど同じコードで処理を記述できます。

前提条件

ここでは実際の量子計算機を利用できるプラットフォームとして、IBM Qiskitを使用します。IBM Quantumで提供されているデバイスを使用するには、アカウントを用意しトークンを取得する必要があります。詳細はIBM Quantumのウェブサイトを参照してください。ここではまずは、IBM Quantumのアカウントが必要ない、Qiskitで提供されているシミュレータを使用して説明を行います。実機とシミュレータはどちらも同じインタフェースを持っているため、そのまま置き換え可能です。

回路の準備

まずは準備としてサンプリングを行うための回路を作成します。

In [ ]:
```
from math import pi
from quri_parts.circuit import QuantumCircuit
# 4量子ビットのサンプル回路を作成
circuit = QuantumCircuit(4)
circuit.add_X_gate(0)
circuit.add_H_gate(1)
circuit.add_Y_gate(2)
circuit.add_CNOT_gate(1, 2)
circuit.add_RX_gate(3, pi/4)
```

SamplingBackendとSampler

現実のデバイスを使用するには、まず `SamplingBackend` を作成し、それを使って `Sampler` を作成する必要があります。 `SamplingBackend` は様々なバックエンドのデバイスや計算ジョブ、計算結果を扱うための、統一的なインタフェースを提供しています。

`SamplingBackend` オブジェクトの作成方法は、バックエンドの種類によります。Qiskitのシミュレータを使用する場合は、 `qiskit_are.AerSimulator` オブジェクトを渡して、 `QiskitSamplingBackend` を作成します。

In [ ]:
```
from qiskit_aer import AerSimulator

device = AerSimulator()
```

In [ ]:
```
from quri_parts.qiskit.backend import QiskitSamplingBackend

backend = QiskitSamplingBackend(device)
```

このバックエンドオブジェクトを直接使用することもできますが、通常はその必要はありません。 `SamplingBackend` は `sample()` メソッドを持ち、 `SamplingJob` オブジェクトを返します。サンプリングジョブからは結果を取り出すことができます。

In [ ]:
```
job = backend.sample(circuit, n_shots=1000)
result = job.result()
print(result.counts)
```

```
{13: 71, 5: 414, 11: 74, 3: 441}
```

バックエンドを直接使うかわりに、バックエンドから `Sampler` を作成することができます。

In [ ]:
```
from quri_parts.core.sampling import create_sampler_from_sampling_backend

sampler = create_sampler_from_sampling_backend(backend)
sampling_result = sampler(circuit, 1000)
print(sampling_result)

# 並列実行されるSamplerの作成
from quri_parts.core.sampling import create_concurrent_sampler_from_sampl

concurrent_sampler = create_concurrent_sampler_from_sampling_backend(back
```

```
{13: 83, 5: 434, 11: 71, 3: 412}
```

`Sampler` を使用する場合、量子状態に対する演算子のサンプリング推定は、シミュレータに対する場合とまったく同じ方法で実行できます。

加算器 (addr) の作成

量子計算では少なくとも古典計算でできることはすべて実現できます。(ただし、それが古典計算より効率的かどうかは、ハードウェアやアルゴリズムの進歩によります。) ここでは、2つの入力値を足し算する論理回路を作ってみましょう。こうした論理回路は加算器と呼ばれ、古典計算機の最も基本的な演算回路のひとつです。

まずは使用するモジュール等をまとめてimportしておきます。

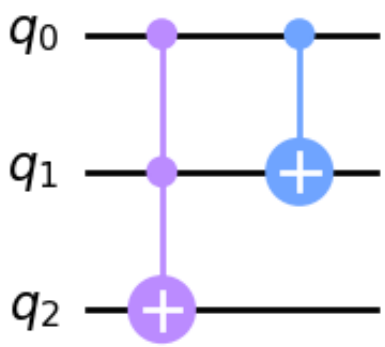
In [ ]:
```
from quri_parts.circuit import QuantumCircuit
from quri_parts.circuit.transpile import RZSetTranspiler
from quri_parts.core.sampling import Sampler, create_sampler_from_samplin
from quri_parts.core.state import ComputationalBasisState
from quri_parts.qiskit.backend import QiskitSamplingBackend

from collections import defaultdict

import qiskit
from qiskit.providers.fake_provider import FakeVigo
from qiskit_aer import AerSimulator
from qiskit import IBMQ, transpile
from qiskit.providers.ibmq import least_busy
```

半加算器の作成

半加算器は下図のような回路で実現できます。q0とq1の入力ビットの足し算の結果がq1に出力され、繰り上がりがq2に出力されます。(q0の結果は捨てられます。) 繰り上がりビットはキャリーと呼ばれます。このような加算器は下の桁からの繰り上がりを考慮していないという意味で、半加算器と呼ばれます。



1つめのゲートはToffoliゲート (CCNOTゲート) で、2つのコントロールビット (図では黒丸) がどちらも$|1\rangle$の時、ターゲットビット (図では+) を反転させます。2つめのゲートはCNOTゲートで、コントロールビットが$|1\rangle$の時、ターゲットビットを反転させます。さまざまな入力に対して、加算と繰り上がりが出力されることを確認してみてください。

これをQURI Partsで記述すると、以下のようになります。入力を状態準備回路に埋め込み、演算回路とつなげて、最後にSamplerで測定を行っています。

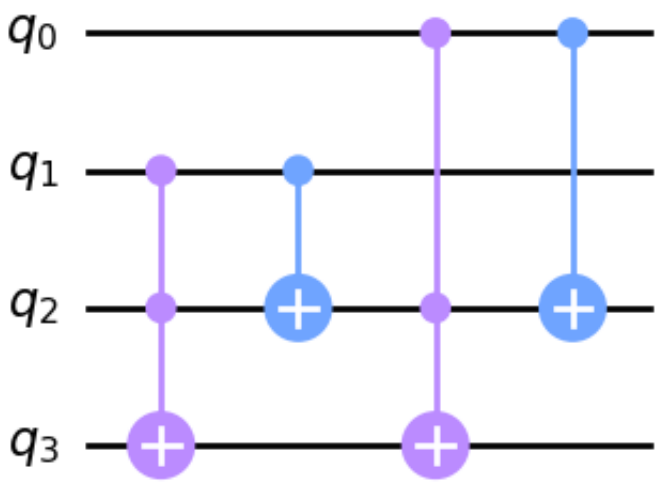
In [ ]:
```python
def half_addr(q0: int, q1: int, sampler: Sampler) -> None:
    circuit = QuantumCircuit(3)
    circuit.add_TOFFOLI_gate(0, 1, 2)
    circuit.add_CNOT_gate(0, 1)

    # bits引数に指定した値の下位ビット側から順に、各量子ビットに割り当てられます
    state = ComputationalBasisState(3, bits=int(f"0{q1}{q0}", 2))
    circuit = state.circuit + circuit

    counts = sampler(circuit, 1000)
    result = defaultdict(int)
    for k, v in counts.items():
        result[k >> 1] += v  # 下位1ビットを捨てて、同じ結果を集計します
    print(f"{q0} + {q1} = {dict(result)}")
```

全加算器の作成

下の桁からの繰り上がりを考慮した加算器は全加算器と呼ばれます。下図では、q0に下の桁からの繰り上がりが入力され、q0 + q1 + q2の結果がq2に出力されます。q3は上の桁への繰り上がりを意味します。この回路も同様にQURI Partsで記述してみましょう。



In [ ]:
```python
def full_addr(q0: int, q1: int, q2: int, sampler: Sampler) -> None:
    circuit = QuantumCircuit(4)
    circuit.add_TOFFOLI_gate(1, 2, 3)
    circuit.add_CNOT_gate(1, 2)
    circuit.add_TOFFOLI_gate(0, 2, 3)
    circuit.add_CNOT_gate(0, 2)

    # bits引数に指定した値の下位ビット側から順に、各量子ビットに割り当てられます
    state = ComputationalBasisState(4, bits=int(f"0{q2}{q1}{q0}", 2))
    circuit = state.circuit + circuit

    counts = sampler(circuit, 1000)
    result = defaultdict(int)
    for k, v in counts.items():
        result[k >> 2] += v  # 下位2ビットを捨てて、同じ結果を集計します
    print(f"{q0} + {q1} + {q2} = {dict(result)}")
```

半加算器と全加算器を繋げることで、任意の桁の足し算が表現できます。興味がある方は実装してみてください。

最後に、シミュレータや実機などから作成したSamplerを切り替えて入力の組み合わせを試すために、ユーティリティ関数を用意してみます。

In [ ]:
```
def test_table_haddr(s: Sampler) -> None:
    half_addr(0, 0, s)
    half_addr(0, 1, s)
    half_addr(1, 0, s)
    half_addr(1, 1, s)


def test_table_faddr(s: Sampler) -> None:
    full_addr(0, 0, 0, s)
    full_addr(0, 0, 1, s)
    full_addr(0, 1, 0, s)
    full_addr(0, 1, 1, s)
    full_addr(1, 0, 0, s)
    full_addr(1, 0, 1, s)
    full_addr(1, 1, 0, s)
    full_addr(1, 1, 1, s)
```

シミュレータでの実行

準備ができたので、まずはシミュレータで実験してみましょう。

In [ ]:
```
sim = AerSimulator()  # シミュレータ
backend = QiskitSamplingBackend(sim)
sampler = create_sampler_from_sampling_backend(backend)
test_table_haddr(sampler)
test_table_faddr(sampler)
```

```
0 + 0 = {0: 1000}
0 + 1 = {1: 1000}
1 + 0 = {1: 1000}
1 + 1 = {2: 1000}
0 + 0 + 0 = {0: 1000}
0 + 0 + 1 = {1: 1000}
0 + 1 + 0 = {1: 1000}
0 + 1 + 1 = {2: 1000}
1 + 0 + 0 = {1: 1000}
1 + 0 + 1 = {2: 1000}
1 + 1 + 0 = {2: 1000}
1 + 1 + 1 = {3: 1000}
```

ノイズのない理想的な量子計算機では、全てのサンプリング結果が正しい答えを返しています。古典計算機と同じように1回だけ実行すれば正しい答えが得られそうです。

ノイズありシミュレータでの実行

次に、現在の実際の量子計算機に近いノイズを人工的に加えたシミュレータで実験してみましょう。

In [ ]:
```
noise_sim = AerSimulator.from_backend(FakeVigo())  # 実機のノイズを再現したシ
backend = QiskitSamplingBackend(noise_sim)
sampler = create_sampler_from_sampling_backend(backend)
test_table_haddr(sampler)
test_table_faddr(sampler)
```

```
0 + 0 = {2: 5, 0: 988, 1: 7}
0 + 1 = {0: 28, 3: 5, 1: 967}
1 + 0 = {0: 45, 1: 949, 3: 6}
1 + 1 = {0: 32, 1: 1, 3: 6, 2: 961}
0 + 0 + 0 = {1: 5, 0: 982, 2: 13}
0 + 0 + 1 = {1: 957, 3: 18, 2: 1, 0: 24}
0 + 1 + 0 = {2: 1, 3: 11, 0: 24, 1: 964}
0 + 1 + 1 = {3: 8, 1: 3, 0: 25, 2: 964}
1 + 0 + 0 = {0: 32, 2: 4, 3: 17, 1: 947}
1 + 0 + 1 = {2: 969, 1: 4, 0: 20, 3: 7}
1 + 1 + 0 = {1: 5, 3: 4, 0: 35, 2: 956}
1 + 1 + 1 = {2: 23, 1: 28, 3: 946, 0: 3}
```

1 + 1のような単純な計算にも関わらず、何回か間違えてしまっています。それでも、何度もサンプリングを繰り返すことで、正しい答えが最も多くの回数得られています。

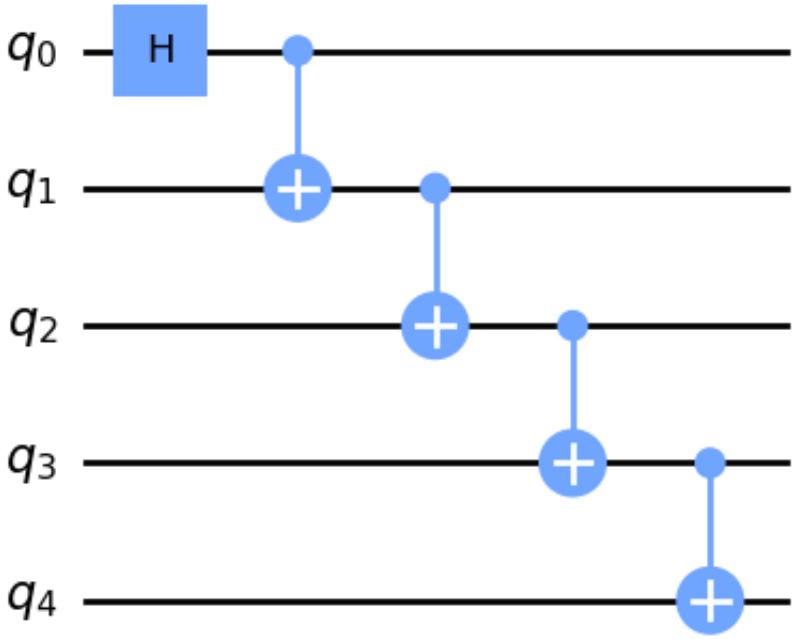
この場合の結果のばらつきは、量子計算自体の特性ではなく、現時点での量子計算機のハードウェア的な制約によるものです。近い未来には、ノイズによるエラーを低減した誤り耐性量子計算機が実現することが期待されています。

シミュレータとの処理時間の比較

実機で計算すると言っても、SDKのAPIを使って計算を投げているだけでは、シミュレータでなく、本当に量子計算機が動いているのか気になる人もいるかもしれません。計算が本当に量子計算機で実行されているのかを検証するには、Fitzsimons-Kashefiのプロトコルなどが存在しますが、ここでは簡単のため、問題の大きさに対する処理時間の変化が、シミュレータと実機で異なった傾向を示すことを観察する方法を紹介しておきましょう。

GHZ回路

GHZ回路と呼ばれる量子回路は、任意の数の量子ビットに対して、$|00\ldots0\rangle + |11\ldots1\rangle$という状態を作り出します。例えば以下は、5量子ビットの回路例です。



量子ビット数を変えてGHZ回路を作成し、実行時間を計測すると、シミュレータと実機で異なる傾向が観察できるはずです。

ただし、IBM Quantumの無料アカウントでは、ジョブの実行までの待ち時間があるので、量子回路の純粋な実行時間をそのままでは計測できません。IBM Quantumのウェブサイトで処理時間が確認できるので、その値を使っても良いかもしれません。

興味がある方は講義後にも実験を継続してみてください。

Exercise 3

5量子ビットのGHZ回路を作成し、シミュレータでサンプリングを行ってみてください。

In [ ]:
```
circuit = QuantumCircuit(5)
circuit.add_H_gate(0)
circuit.add_CNOT_gate(0, 1)
circuit.add_CNOT_gate(1, 2)
circuit.add_CNOT_gate(2, 3)
circuit.add_CNOT_gate(3, 4)

sim = AerSimulator()  # シミュレータ
backend = QiskitSamplingBackend(sim)
sampler = create_sampler_from_sampling_backend(backend)

counts = sampler(circuit, 1000)
print({bin(k): v for k, v in counts.items()})
```

```
{'0b0': 515, '0b11111': 485}
```

実機での計算

それでは最後に、実際の量子計算機を使ってみましょう。IBM Quantumのアカウントは事前に登録されていると思います。ご自身のアカウントのトークンをコピーし、以下のコード内に埋め込んで、実行してみましょう。

(トークンは自分以外に公開しないように注意してください。サイトで再発行し、現在のトークンを無効化することもできます。)

In [ ]:
```
def prepare_real_device():
    IBMQ.save_account(
        "",  # IBM Quantumのトークンを入れます
        overwrite=True,
    )
    provider = IBMQ.load_account()
    backends = provider.backends()
    print(backends)
    backend_lb = least_busy(provider.backends(simulator=False, operationa
    print(backend_lb)
    return backend_lb
```

以下のコードを実行して、実際の量子計算機を使って、半加算器で1 + 1を計算してみましょう。このコードでは最も空いているサーバを自動的に探してきますが、ジョブが実行されるまでには、しばらく順番待ちをしなければいけないかもしれません。実行状態はIBM Quantumのウェブサイトの各アカウントから確認できます。

このノートブックはしばらく走らせたままにして、続きの講義に移りましょう。

In [ ]:
```
device = prepare_real_device()  # 実機
backend = QiskitSamplingBackend(device)
sampler = create_sampler_from_sampling_backend(backend)
half_addr(1, 1, sampler)  # 1 + 1を計算します
```

In [ ]: