Vol.2021-QS-4 No.8
2021/10/15

HPCクラスタを利用した量子計算シミュレータの実装

高橋 ひとみ[1,a)]　井床 利生[1,b)]　土井 淳[1,c)]　堀井　洋[1,d)]

概要：量子コンピュータが実用化されつつある現在、古典コンピュータによる量子計算のシミュレーションはアルゴリズムの動作検証を行う上で有用である。しかし、計算が困難な処理を可能とする量子コンピュータを古典コンピュータでシミュレーションするには、小規模の量子回路を実行する際にも大量の計算時間が必要となる。特にノイズ付きのシミュレーションではショット毎にシミュレーションの実行が必要となるため、ノイズなしの量子回路と比較すると実行時間が大きくなる。そこで本論文では、複数の量子回路やノイズ付きのシミュレーションの計算時間を短縮するため、オープンソースの量子計算シミュレータである Qiskit Aer を HPC クラスタ上で実行可能にした。本実装を用いた評価では、通常の Qiskit Aer と比較し最大 86%のシミュレーション時間の削減が実現できた。

An Implementation of Quantum Computing Simulator for HPC Cluster

1. はじめに

量子コンピューティングの研究が進み、今まで論理的なものとして扱われていた量子アルゴリズムが、実際のゲート型量子コンピュータ上で検証可能となってきている。量子コンピュータ上で実行できる量子ビット数は増加しており、2023 年には 1000 量子ビットのシステムの実現されることが発表されている [1]。一方、多くの量子ビット数を利用する量子アルゴリズムの実行は、既存のコンピュータではシミュレーションが困難である。量子計算シミュレーションでは多次元の複素ベクトルを扱うため、それらの状態を保持するためには莫大なメモリと計算コストが必要となるからである。例えば、倍精度の複素数を用いた状態ベクトルを求める場合、n 量子ビットのシミュレーションに必要なメモリ量は 2^{n+4} バイトとなる。50 量子ビットのシミュレーションであっても、16 ペタバイトのメモリ量が必要となるため、既存の計算資源でのシミュレーションは困難である。

量子計算のシミュレーションは小さい量子ビット数のアルゴリズムしか実行できないが、簡単に何度も実行できるため、新しいアルゴリズムの動作検証、パラメータによる性能の影響などのテストを行うには非常に有用なツールとなる。しかし、小さい量子ビット数のシミュレーションでも、大量の量子回路をシミュレーションする場合や、量子計算機のノイズを状態ベクトルを用いてシミュレーションする場合では、シミュレーション時間が非常にかかってしまう。

そこで本論文では、オープンソースのシミュレータである Qiskit Aer [2] と、並列実行フレームワークの Dask [3] を用いて、複数の回路やノイズ付きシミュレーションを、分散並列環境で並列実行する手法を提案する。Dask は、クラウド、コンテナ、HPC クラスタと、様々なプラットフォームで並列実行の実現を可能とするフレームワークで、機械学習やデータフレームの分散処理に多く利用されている。本手法では、Qiskit Aer 内で、要求されるシミュレーションが並列実行可能であれば複数のシミュレーションに分割し、Dask クラスタ上で並列実行することで、高速にシミュレーションを実現する。並列にシミュレーションした結果は集約され、通常のシミュレーションと同等の結果が返されるため、Qiskit Aer に Dask のスケジューラを指定する以外は、既存のアプリケーションを変更する必要はない。本論文の実験では、通常の Qiskit Aer と比較し最大 86%のシミュレーション時間の高速化が、HPC クラスタを用いて実現可能なことを確認している。

この後の本論文の構成として、まず第 2 節で HPC クラ

[1] 日本 IBM 株式会社　東京基礎研究所
IBN Japan, Ltd., 19-21, Nihonbashi, Hakozaki-cho, Chuo-ku, Tokyo 103–8501
[a)] hitomi@jp.ibm.com
[b)] itoko@jp.ibm.com
[c)] doichan@jp.ibm.com
[d)] horii@jp.ibm.com

スタや複数の計算機を利用して量子計算シミュレーションを行う関連研究を紹介し、第 3 節で本論文が取り扱う量子計算シミュレーションの種類や特徴を説明する。次に第 4 節で通常の Aer を用いたノイズ付きシミュレーションの特徴を説明し、既存のシミュレーションの動作およびその問題点について示す。その後、第 5 節でクラスタを利用したシミュレーションの実装について説明し、それらの評価を第 6 節にて示す。最後に第 7 節にてまとめについて述べる。

2. 関連研究

まず、クラウドなどのコンピュータクラスタやマルチノードを用いる、量子計算シミュレータについて関連研究を述べる。マイクロソフト社は自社のクラウド環境である Azure を用いた Azure Quantum [4] を提供している。ここで動作する量子回路は、同社が提供する量子コンピュータ向け開発キット [5] による Q#というプログラミング言語を用い記述される。この Q#で書かれたプログラムは、Azure Quantum のサービスを通じクラウド上でのシミュレーションもしくは IonQ、Honeywell が提供する実機での実行も可能である。もし Azure Quantum 上でシミュレーションを実行する場合、ユーザはワークスペースと呼ばれるリソースへジョブとして Q#のプログラムを送信できる。ワークスペースへのリソースはユーザが自由に設定でき、実行する量子回路により柔軟な設定可能であるため、シミュレーションはリソースが許す限り同時に送信および並列実行が可能である。ただしノイズ付き量子計算のシミュレーションはサポートしていない。

Amazon 社は自社の AWS を用いた量子コンピューティングサービスである、Amazon Braket [6] を提供している。Amazon Braket においても、クラウドを用いたシミュレーションもしくは Rigetti、IonQ や D-Wave が提供している実機上でも量子回路を実行できる。ユーザは Amazon が提供する Python ライブラリである Amazon Braket Python SDK や D-Wave が提供する Ocean SDK を用い、量子アルゴリズムを記述できる。シミュレータのバックエンドは 3 種類から選択でき、34 量子ビットまでの状態ベクトルシミュレータの実行が可能であり 35 件まで並列実行可能な SV1、17 量子ビットまでの密度行列シミュレーターが実行可能な DM1、50 量子ビットまでのテンソルネットワークシミュレータが実行可能な TN1 が存在する。ノイズ付き量子計算のシミュレーションは DM1 により可能であるが、密度行列によるシミュレーションのみで状態ベクトルを用いたシミュレーションはサポートしていない。

Google 社は Google Cloud をバックエンドとした量子コンピューティングサービスである Google Quantum AI [7] を提供している。ユーザは同社より提供される Python SDK である Cirq を用い量子アルゴリズムを記述できる。量子アルゴリズムをシミュレーションする場合は qsim と呼ばれるシミュレータ上で実行可能であり、この qsim はユーザ自身のローカルマシン上で動作可能もしくは Google Cloud 上で Docker としても実行できる。qsim は Aer と同様にノイズ付きシミュレーションも可能であり、密度行列および状態ベクトルを用いたシミュレーションを提供している。ユーザは記述した量子回路の規模に合わせ、qsim が動作する Google Cloud の VM リソースを柔軟に割当できる。また、Google Quantum AI もシミュレーションだけではなく、自身が提供する実機や AQT、IonQ、Pasqal、Rigetti が提供する実機上でも Cirq によって記述したアプリケーションが実行可能である。

つぎにオープンソースの量子計算シミュレータにおいて、複数ノードで実行可能なシミュレータを説明する。ただし、シミュレータの同時実行数を増やすことが目的ではなく、分散メモリを使用し量子ビット数が大きい量子回路のシミュレーションが目的であるため、本論文とは目的が異なる。Intel 社 Parallel ComputingLab で開発された Intel-QS (qHiPSTER (The Quantum High Performance Software Testing Environment)) [8], [9] は、マルチノード、マルチコアをサポートし、大規模な分散環境で量子計算シミュレーションが可能なシミュレータである。コードは C 言語で記述されており、ソースコードも Git 上に公開されている [10]。論文 [8] では 2048 ノード、196TB RAM の実験環境を用い、42 量子ビットまでのシミュレーションを実現している。

QuEST (Quantum Exact Simulation Toolkit) [11] は Jones らにより提案されたシミュレーションである。C 言語で記述されており、ソースコードはオープンソースとして Git 上に公開されている [12]。OpenMP を用いた並列化、MPI による複数ノードへの対応もしており、かつシングル GPU のサポートもしている。またバッチジョブを使用するクラスタコンピューティング環境でもシミュレーション可能であり、シミュレーションを動作させるホストとして、通常のラップトップからスーパーコンピュータまで同一のプログラムで実行できるよう設計されている。量子アルゴリズムを記述するために必要なゲートの関数群は全て、C のライブラリとして提供されており、QuEST を用いて量子アルゴリズムを作成する場合は、それらの関数が使用された C プログラムとなる。

ProjectQ [13] は Thomas らが開発した Python ベースのシミュレーションであり、オープンソフトウェアとして公開されている [14]。ProjectQ を用いたアプリケーションは ProjectQ だけではなく、IBM、AQT が提供する実機や、AWS Braket、IonQ のサービス上でも実行できる。一部のコアとなる処理は C++のモジュールになっており、ゲートフュージョン、SIMD などの最適化を用い高速化を実現している。また、シングルノードだけではなく MPI を使用したマルチノードでも動作可能なよう実装されてお

り、論文では 8192 ノード、0.5PB のメモリを用い 45 量子ビットまでのシミュレーションを実現している。量子プログラム用に SDK も公開しており、Python コードに埋め込む形で DSL として記述できる。

3. 量子計算シミュレーション

本節ではノイズ付き量子計算シミュレーションの基礎について説明し、その後、密度行列および状態ベクトルを用いたシミュレーション方法とその特徴について説明する。

3.1 ノイズ付き量子計算シミュレーション

本論文では、量子計算のモデルとして、量子回路、即ち、量子演算とその演算を作用させる量子ビットの組の系列を考える。量子演算としては、初期化・ゲート（ユニタリ演算）・測定を考える。ノイズを考慮しない場合の量子計算シミュレーションでは、これら三種類の量子演算を古典計算機上で実行できればよい。一方で、ノイズを考慮した量子計算シミュレーションでは、それらに加えて、ノイズを表現する演算である Kraus 演算（定義は後述）も実行できる必要がある。ここで、Kraus 演算は、ある混合状態を別の混合状態に写す演算であるという点で、純粋状態を純粋状態に写す他の三種の演算とは異なる。つまり、ノイズを考慮しない場合のシミュレーションでは、純粋状態上の計算のみを扱えば十分であるが、ノイズを考慮する場合には、本質的に混合状態上の計算を扱う必要がある。以下では、この「混合状態上の計算」をどのように扱うかの違いに応じた二通りのシミュレーション方法を説明する。

3.2 密度行列によるシミュレーション

まず、混合状態を表現する行列（密度行列）をそのままメモリ上に保持する方法が考えられる。この場合、n 量子ビットの系の混合状態を、$2^n \times 2^n$ の複素行列として保持することになる。全ての量子演算はこの行列に対する計算として定義することができる。例えば、任意のゲートはユニタリ行列 U の形で与えられ、密度行列 ρ に対してゲートを適用した後の密度行列は $U^\dagger \rho U$ として計算できる。同様に、任意の Kraus 演算は $\sum_i K_i^\dagger K_i = I$ を満たす行列集合 $\{K_i\}$ の形で与えられ、密度行列 ρ に対して Kraus 演算を適用した後の密度行列は $\sum_i K_i^\dagger \rho K_i$ として計算できる。

密度行列を用いたシミュレーションは、混合状態を陽に保持するため、多量のメモリ（4^n のオーダ）を必要とする。その代わり、同一量子回路の複数回の実行をシミュレーションする場合に、測定直前の状態を一度だけ計算し、その状態に対する測定結果のみを複数回算出する、という効率化が可能である。以降、量子回路の一回の実行のことを Qiskit に倣い**ショット**と呼ぶ。

3.3 状態ベクトルによるシミュレーション

実は、Kraus 演算による状態変化（$\rho \mapsto \sum_i K_i^\dagger \rho K_i$）を、「現在の状態 ρ による確率 $\mathrm{tr}(K_i^\dagger \rho K_i)$ での、状態 $\frac{K_i^\dagger \rho K_i}{\mathrm{tr}(K_i^\dagger \rho K_i)}$ への遷移」と捉えることが可能である。この解釈に基づくことで、純粋状態 $|\psi\rangle$ に対する Kraus 演算を、「確率 $||K_i|\psi\rangle||$ での純粋状態 $\frac{K_i|\psi\rangle}{||K_i|\psi\rangle||}$ への遷移」と捉えることができる。つまり、上記の確率的な計算に対して、サンプリングを行いながら計算を進めることで、Kraus 演算を含む全ての量子演算を純粋状態上で行えることが分かる。

このように、純粋状態の行列表現（状態ベクトル）をメモリ上に保持するタイプのシミュレータでもノイズ付きシミュレーションが可能であり、モンテカルロ波動関数法とも呼ばれる。コード 1 に状態ベクトルを用いて Kraus 演算をシミュレーションする際のサンプルコードを示す。n 量子ビットの系の状態ベクトルは、要素数 2^n の複素ベクトルであるから、密度行列を保持するよりも大幅に少ないメモリでシミュレーションを行うことができる。一方で、Kraus 演算を含む量子回路　(ノイズ付き量子回路)　に対しては、サンプリングを行いながら計算を進めるため、基本的に途中状態を再利用した効率化は行えず、ショット毎に回路の先頭からシミュレーションを繰り返す必要がある。そのため、図 1 にあるように、状態ベクトルを用いたシミュレーションの方が、密度行列を用いたシミュレーションよりも、より多い量子ビット数のノイズ付き量子回路を扱える反面、ショット数が多い場合には、より多くの実行時間を必要とする（図 1 の詳細な説明は次節）。

```
// kmats: 演算を定義する行列集合 Kraus
// state: 状態ベクトル
// r: [0, の乱数 1]
accum = 0
for j in range(len(kmats)-1):
        p = state.apply(kmats[j]).norm()
        accum += p
        if accum > r:
                return state.apply(kmats[j]) / p
return state.apply(kmats[-1]) / (1 - accum)
```

コード 1　状態ベクトル上の Kraus 演算のサンプルコード

4. Aer におけるノイズ付き量子計算シミュレーション

本節では Qiskit Aer によるノイズ付きシミュレーションの特徴を解説する。Aer では、量子回路のシミュレーション手法が複数実装されており、量子回路により最適なシミュレーション手法が自動で選択される。本論文が対象とするノイズ付きシミュレーションでは密度行列もしくは状態ベクトルを用いたシミュレーションを選択することが一般的である。第 3 節で述べたとおり、密度行列のノイズ付きのシミュレーションはショット数に関係がなく、実行時

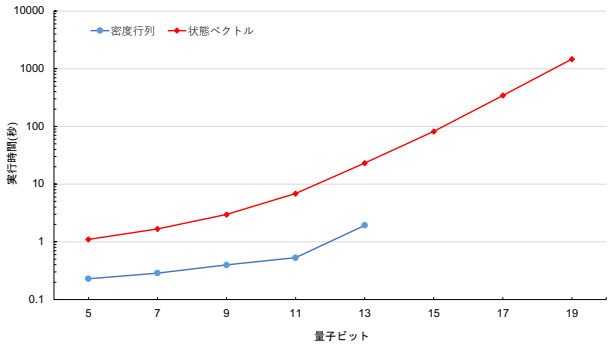

図 1 量子ビット数とシミュレーション時間 (ショット:100000)

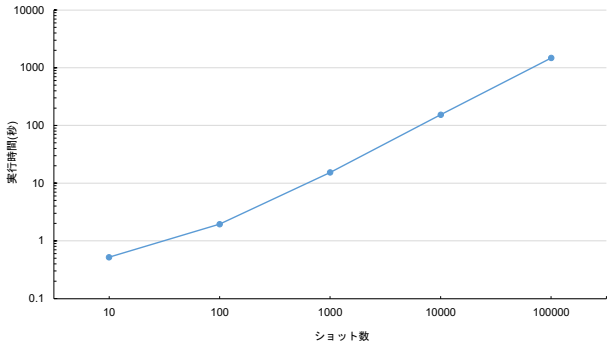

図 2 ショット数とシミュレーション時間 (量子ビット:19)

間は短いがメモリの使用量が大きい。一方、状態ベクトルではショット数の回数だけシミュレーションする必要があり実行時間は長いが、メモリの使用量は、n 量子ビットのシミュレーションで密度行列の 2^n 分の 1 となる。

Aer の密度行列、および、状態ベクトルを用いたノイズ付きシミュレーションの特徴を、予備実験により確かめる。具体的には通常の Qiskit Aer を用い、ノイズ付き量子計算のシミュレーション時間を測定した。Aer のバージョンは 0.8 を使用し、IBM Cloud 上に 1VM を作成した。VM の性能は OS: CentOS8.0、メモリ：64 GB、CPU: 32 x 2.0 GHz となっている。

まず、Quantum Volume (QV) の回路を 5 から 19 量子ビットまで 2 量子ビット毎、Depth を 1 とし 100000 ショットを実行するコードを作成し、シミュレーション時間を測定した。結果を図 1 に示す。横軸は量子ビット数、縦軸はシミュレーション時間 (秒) を示し、密度行列、状態ベクトルのそれぞれの実行時間を表している。量子ビット数が上がるとシミュレーション時間が指数的に増え、特に状態ベクトルは密度行列のシミュレータと比較しシミュレーション時間が長い。密度行列はメモリ量の制約上 13 量子ビットまでのシミュレーションのみで、15 量子ビット以上の測定は出来なかった。本実験では各シミュレーションの実行時間を測定するために、Aer のシミュレーション方法を明示的に指定し実行したが、シミュレーション方法を指定しない場合、Aer がシミュレーション方法を自動で選択するため、5 から 13 量子ビットまでは密度行列を使用し、15 量子ビット以降は状態ベクトルのシミュレータを選択する。

次に、量子ビット 19 の QV を用い、ショット数を 10、100、1000、10000、100000 と変化させたノイズ付きシミュレーションを実行する。図 2 に結果を示す。19 量子ビットでは Aer は状態ベクトルによるシミュレーションを選択するため、ショット回数だけシミュレーションの実行が増える。そのため、ショット数が増加するとシミュレーション時間の増加が見られた。これらの実験結果より、ノイズ付きの量子回路を状態ベクトルでシミュレーションする場合、ショット数や量子ビットによってはシミュレーション時間が多くかかってしまう。本論文はこの問題に着目し、ノイズ付きのシミュレーション時間を縮小させるため、シミュレーションの並列実行が可能となる機能を Aer へ追加する。

5. クラスタ環境が利用可能となる量子計算シミュレーションの機能拡張

本研究ではオープンソースのシミュレータである Qiskit Aer [2] の機能を拡張し、複数台のマシン上に Aer を並列実行できる機能を追加した。具体的には (1) 複数の量子回路をシミュレーションする場合、各量子回路にデータを分割、コピーすることでそれぞれのスレッドもしくマルチノード上で並列実行する機能、(2) ノイズ付きの量子回路を状態ベクトルによってシミュレーションする場合、量子回路のショット数を分割しコピーをすることで、スレッドもしくはマルチノード上で並列実行する機能を実装した。本節ではまず、機能拡張を行った Aer の動作手順について述べ、次に各機能の詳細な実装について説明する。

5.1 Qiskit Aer の動作手順

ユーザが Qiskit SDK を用い記述した量子回路を Aer がシミュレーション実行する場合、Aer は量子回路より選択したシミュレーション手法をバックエンドとして設定し、そのインスタンスを作成する。その後、Qiskit SDK により記述された量子回路は、以下のメンバが存在する `QasmQobj(qobj)` というクラスに変換される

QasmQobj

- `qobj_id`：Qobj の識別子
- `config`：シミュレーション実行時の設定情報。ノイズ情報やショット数なども含まれる
- `header`：バージョンなどの情報
- `experiments`：量子回路の情報。リスト構造で複数の量子回路を格納できる

Qobj の `config` にはユーザから渡されるノイズ情報や

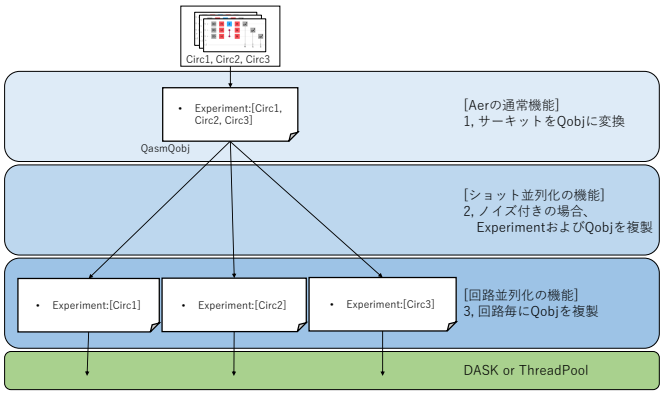

図 3 複数の量子回路におけるシミュレーションの並列実行

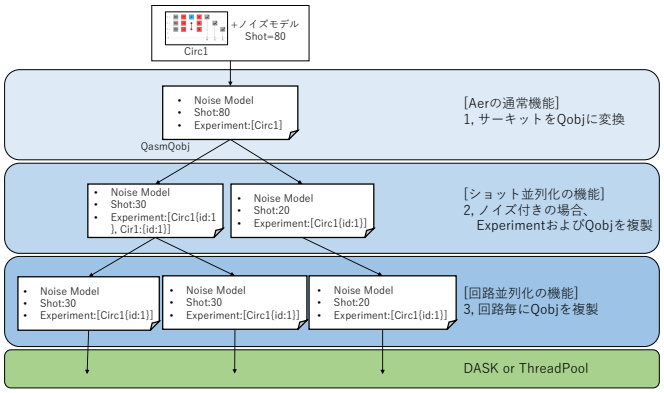

図 4 ノイズ付き量子回路におけるシミュレーションの並列実行

ショットの回数など、シミュレーションもしくは実機の実行に必要な設定情報が格納される。また、量子回路の情報は `experiments` と呼ばれるメンバに格納される。このメンバはリスト構造であるため、複数の量子回路を一つの Qobj に格納できる。Qobj に変換された量子回路はジョブと紐付けされ、Python の `concurrent.futures.Threadpool` が格納された `Executor` により、`experiments` のリスト内の量子回路を一つずつシミュレーションしていく。

本実装はこのジョブ実行機能を拡張し、複数の量子回路およびノイズ付き量子回路の並列実行を実現した。また、ジョブ実行は `Threadpool` に加え Dask [3] も使用できるよう修正した。`Threadpool` と Dask の `Client` はジョブを実行し結果を取得する関数の名前、引数、返り値などがほぼ同一であり、既存のコードを大きく修正せずにそのまま使用できる。ただし、Aer 内部では Dask を使用したクラスタの作成自体はできないため、ユーザがクラスタを作成した後、それらの情報を格納した `Client` を Aer へ渡せるよう、Aer へのオプションを新たに追加した。これにより Dask がサポートするクラスタ環境、例えばマルチノードや Kubernetes、Hadoop、HPC のバッチジョブシステム上でも Aer の実行が可能となる。

5.2 複数の量子回路における並列実行

Qobj の `experiments` に複数の量子回路が存在する場合、それらの回路は独立に実行できるため、それぞれのシミュレーション結果を集約すれば、Aer とのシミュレーション結果は同じとなる。

本機能の導入にあたり、`max_job_size` というオプションを追加した。このオプションは `experiments` のリスト内に存在する量子回路の上限値を設定し、それを超えるようなら Qobj を複製し `experiments` のリストを分割する。例えば `experiments` 内に 100 量子回路が存在し `max_job_size` が 10 に設定さていた場合、本機能は `experiments` 内の量子回路を 10 個ずつ分割し、コピーされた Qobj へそれぞれ格納される。

本機能の動作手順として、まず Qobj 内の `experiments` 内に複数量子回路が存在するかチェックする。もし複数量子回路が存在する場合、`max_job_set` をチェックする。この値を超える個数の量子回路が存在する場合、値が収まるよう量子回路のリストを分割し、それを格納する分割個数分の Qobj をコピーする。もしサンプリングで使用するランダムシードがユーザから α と設定されていた場合、0 番目の Qobj のシードを α、n 番目の Qobj のシードを $\alpha + n$ と設定する。これによりサンプリングの際に各ノードが同一の値を算出することを防ぐ。コピーされた Qobj はユーザからの `Executor` に渡され、それぞれジョブとして実行される。図 3 では `max_job_set` を 1 に設定し、ユーザの量子回路から生成された Qobj の `experiments` 内には 3 つの量子回路が存在する。リスト内の量子回路は `max_job_set` を超えるため、1 つずつ分割されコピーされた 3 つの Qobj に格納される。その後、それぞれの Qobj は Dask もしくは Threadpool に渡され、シミュレーションが並列実行される。

シミュレーションの終了後、Aer では以下の `Result` に実行結果を格納しユーザに返す。

Result

- `backend_name`：バックエンド名
- `backend_version`：バックエンドのバージョン
- `qobj_id`：シミュレーションを実行した Qobj の ID
- `job_id`：バックエンドのジョブに対する ID
- `success`：シミュレーションの実行が成功したかどうか
- `results`：Qobj の `experiments` 内のリスト順に対応するシミュレーション結果。リスト構造

本機能により分割されたリストを含む Qobj は、それぞれの計算結果が `Result` に格納され返信される。そのため本ジョブが終了しそれぞれの結果が返った際、コピーされた Qobj を一つに統合する必要がある。そこで、本機能は `experiments` が分割されたそれぞれの Qobj と実行時のジョブ ID を紐付け、どのジョブの計算結果を一つに統合

するかという情報をジョブ送信時に保存する。動作手順として、まず、返信された最初の Result に統合される他のジョブが存在する場合、それら全てのジョブ ID の計算結果が返るまで待つ。全ての計算結果が返って来た場合、ジョブ ID が一番若い results リストへ、ジョブ ID の若い順に各 results 内の結果を追加していく。これらの動作手順により分割前の Qobj に対する、統合された Result が生成できる。

```
1 exc = Client('dask1:8786')
2 qbackend = AerSimulator()
3 qbackend.set_options(executor=exc,
      max_job_size=1)
4 result = qbackend.run(circ_list).result()
```

コード 2　シミュレーションの並列実行を行うサンプルコード

コード 2 に本機能を使用する際のサンプルコードを示す。本機能を使用するためには、通常のユーザプログラムに 1 行目の Executor の作成、3 行目の set_options による Executor のセットおよび max_job_set の設定の 2 行を追加するだけである。もし Threadpool を使用したい場合、1 行目の Dask Client を Threadpool の宣言に変更するだけである。

5.3　ノイズ付き量子回路におけるショットの並列実行

次にノイズ付き量子回路におけるシミュレーションの並列実行機能について述べる。ノイズ付きシミュレーションを状態ベクトルを用いてシミュレーションする場合、ショット回数分だけ量子回路のシミュレーション実行が必要である。このショット回数のシミュレーションもすべて独立に並列実行できる。そこで、本機能ではノイズ付き量子回路のシミュレーションであり、ショット回数が 1 以上である場合は Qobj 内の experiment および Qobj 自体をコピーすることで、並列実行を可能にした。

図 4 に大まかな本機能の概要を示す。本機能の導入にあたり、max_shot_size というオプションを追加した。このオプションはショット数の上限を設定し、それを超えるようなら experiments 内の量子回路をコピーする。ショット数の設定は Qobj の config 内に存在し、experiments 内の量子回路全てにこの設定が反映される。そのため、max_shot_size で設定されたショット数が割り切れない場合、余りのショット数を設定するため、Qobj をコピーする必要がある。例えば図 4 のように max_shot_size を 30 に設定し、ユーザからショット数 80 のノイズ付き量子回路が渡されたとする。ショット数は $30 \times 2 + 20 = 80$ となり、ショット数を 30 に設定した Qobj 内の experiments には、同じ量子回路が 2 つとなるようコピーされる。また、ショット数を 20 に設定し、experiments 内の量子回路は 1 つにした Qobj がコピーされる。さらに、シミュレーション結果が返って来た際、experiments 内のどの量子回路の結果を集約すれば良いかを示す id が追加される。同じ id ならば results 内のサンプリングカウントを集約し一つの結果としてユーザに返す。ここで複製された Qobj は次の回路並列化機能により複数の量子回路が存在する場合、max_job_set の値に従い experiments を分割、複製した Qobj に格納され Dask などの Executor に渡される。並列化機能が受信したシミュレーション結果は、統合された最大 2 つの Qobj がショット並列化機能に渡される。ここでは各 Qobj のシミュレーション結果に、送信時に追加された id が存在するかチェックする。もし id が存在すれば results 内のカウントを集約し、リスト内の一番初めの結果にまとめ、他のデータを削除することで、ユーザが受け取る Result を一つに集約できる。最後にノイズ付き量子回路における並列実行のサンプルコードを示す。コード 2 に示したサンプルコードへ、max_shot_size のオプションを設定する、qbackend.set_options(max_shot_size=100) を追加するだけで、ノイズ付き量子回路のシミュレーションが並列に実行できる。

6.　評価

本節では Qiskit Aer に追加拡張した並列実行化機能についての評価を説明する。まず複数ノードによるシミュレーション時間の変化の評価を行い、次にジョブ管理に LSF を用いた、バッチジョブシステム上での測定を行った。

6.1　複数ノードによるノイズ付き量子計算シミュレーション

本小節では本論文で実装した機能である、ノイズ付き量子計算シミュレーションの実行時間を測定する。実験環境として第 4 節で使用した同じスペックの VM を 1-5 台作成し、VM 数を変化させシミュレーション時間を測定した。使用したアプリケーションは第 4 節と同様にノイズ付きの QV を用い、5 から 19 量子ビットまで 2 量子ビット毎、Depth を 1 とし 100000 ショットに設定した。また、max_shot_size は 1000、max_job_size は 1 としている。結果を図 5 に示す。横軸が量子ビット数、縦軸が実行時間 (秒) を示す。それぞれ通常の Qiskit Aer、本機能を用い Executor に Dask を使用した Aer の実行時間を 1-5VM で示している。通常の Aer では 13 量子ビットまで密度行列を使用するため、シミュレーション速度が速いが、それ以降は状態ベクトルを使用するため急激にシミュレーション時間が長くなる。本機能のシミュレーションではすべて状態ベクトルを用いシミュレーションを行っているため、通常の Aer と比較し 13 量子ビットまでは実行時間が長くなっている。15 量子ビット以降は通常の Aer と比較し、1VM ではほぼ性能は変わらず本機能の追加によるオーバヘッド

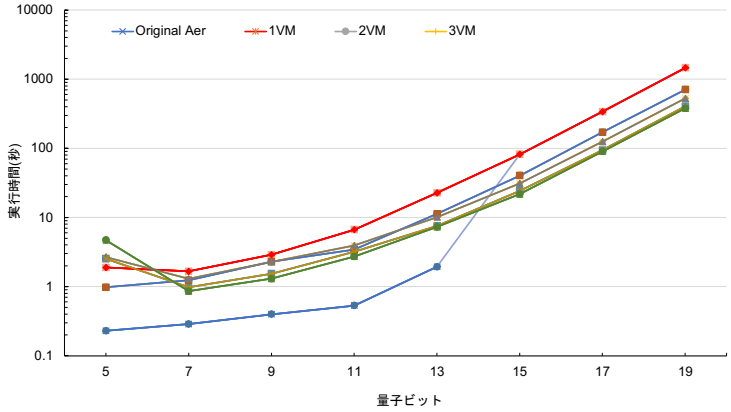

図 5 1-5VM における量子ビット数とシミュレーション時間

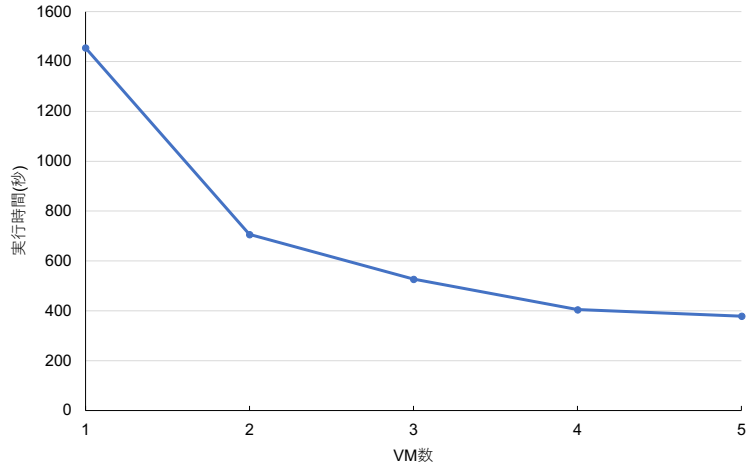

図 6 VM 数とシミュレーション時間 (量子ビット:19、ショット:100000)

は少ないと考えられる。また VM の台数を増やす毎にシミュレーション時間が減少し、2VM 以上であれば 15 量子ビット以降は通常の Aer と比較し実行時間が短くなった。

次に台数効果を示すため量子ビット数を 19、ショット数を 100000 に設定し、さらに VM の台数を 1-5VM と変化させ、シミュレーション時間を測定した。結果を図 6 に示す。横軸は VM の台数、縦軸は実行時間 (秒) となっている。VM の台数が増加する毎にシミュレーション時間が減少している。特に 1VM から 2VM への減少が大きく、ほぼ理論値である 51%に削減できた。

```
1 cluster = LSFCluster(
2     cores=4,
3     memory="30␣GB"
4 )
5 exc = Client(cluster)
```

コード 3 LSF クラスタのサンプルコード

6.2 バッチジョブシステムにおけるノイズ付き量子計算シミュレーション

本節は Executor に Dask を用い、バッチジョブシステム上でノイズ付き量子計算シミュレーション時間を計測した。本評価で使用したバッチジョブシステムは、スケジューラとして LSF が使用されており、各コンピュータノードは CPU: Intel Xeon E5-2600 v4 (28 Core)、メモリ:512GB、OS:RHEL7.6 が搭載されている。

Dask でバッチジョブシステムを使用するためには、コード 2 の 1 行目の代わりに、コード 3 のような Cluster を定義し exc にセットするだけである。

コード 3 は LSF クラスタを使用し、1 ジョブ辺り Core を 4、メモリを 30GB 割り当てるというリソース定義をしている。本評価も同じリソース定義を用い、ジョブを 3 から 8 に変化させシミュレーション時間を測定した。使用した量子回路は第 6.1 節と同様にノイズ付き QV、量子ビット数は 19、ショット数は 100000 と設定した。また、`max_shot_size` は 1000、`max_job_size` は 1 としている。図 7 に結果を示す。横軸はジョブ数、縦軸は実行時

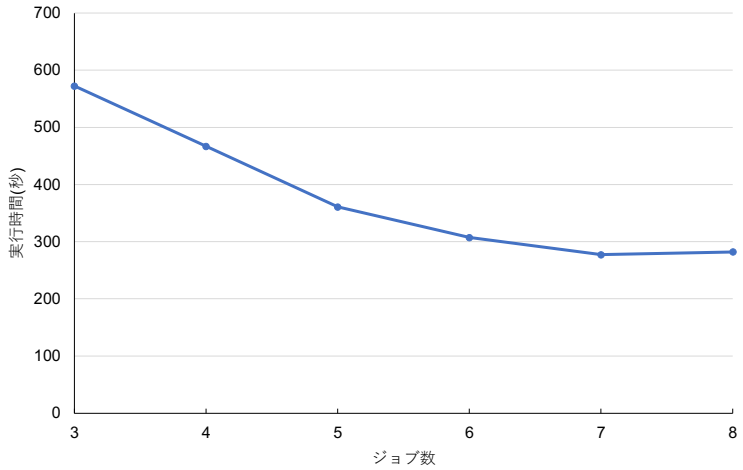

図 7 クラスタ環境におけるシミュレーション時間 (量子ビット:19)

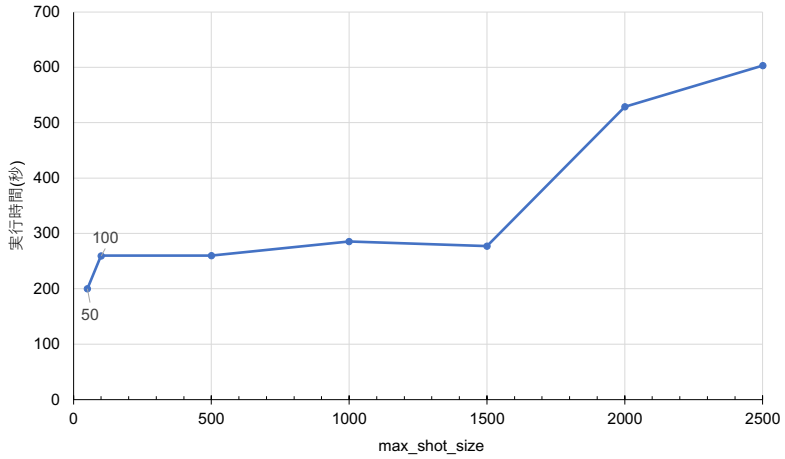

図 8 クラスタ環境における max_shot_size とシミュレーション時間 (量子ビット:19, ショット:100000, ジョブ数:7)

間 (秒) となっている。VM と同様にジョブ数が増加するとともにシミュレーション時間が減少した。7 ジョブでシミュレーション時間が 277 秒、8 ジョブでシミュレーション時間は 282 秒となった。次に、`max_shot_size` を変化させシミュレーション時間を測定した。使用したシナリオはノイズ付き QV、量子ビット数は 19、ショット数は 100000 とし、ジョブのサイズは前の評価において一番シミュレーション時間が短い 7 ジョブに設定した。また、`max_shot_size` は 50、100、1000、1500、2000、2500 と変化させた。`max_shot_size` を 10 にした場合、Dask にエラーが発生し測定ができなかった。図 8 に結果を示す。横軸は `max_shot_size`、縦軸は実行時間 (秒) となっている。`max_shot_size` が一番小さい 50 の場合、シミュレーション時間が 200 秒と一番短くなり、ローカルのみで動作する Aer とのシミュレーション時間と比較し、86%シミュレーション時間が減少できた。Dask のスケジューラはタスクサイズが小さいジョブに適した仕様になっているため、`max_job_size` が小さい方が CPU を効率良く使用できる。

7. まとめ

本論文は既存の Qiskit Aer に複数回路およびノイズ付き量子回路のシミュレーションを並列実行する機能を追加した。ユーザから複数の量子回路を渡された場合、それらを分割し、それぞれのジョブとしてシミュレーションを実行することで、並列実行が可能となった。またノイズ付き量子回路では量子回路をコピーし、複数の量子回路にすることで、同様に並列実行が可能となった。本機能を評価するため、シミュレーションを行うジョブの Executor として Dask を用い、複数 VM やバッチジョブシステムでシミュレーション時間を測定した。VM およびバッチジョブが増える毎にシミュレーション時間が減少し、LSF クラスタによるバッチジョブシステムにおいて 7 ジョブでシミュレーションを行った場合、通常のローカルノードのみの Aer と比較し 86%のシミュレーション時間が減少した。

参考文献

[1] Gambetta, J.: Ibm’s roadmap for scaling quantum technology, *IBM Research Blog https://www. ibm.*

com/blogs/research/2020/09/ibm-quantum-roadmap (2020).
[2] qiskit-aer: `https://github.com/Qiskit/qiskit-aer`.
[3] DASK: `https://dask.org/`.
[4] Microsoft: Azure Quantum, `https://azure.microsoft.com/en-us/services/quantum/`.
[5] Microsoft: Microsoft Quantum Development Kit, `https://www.microsoft.com/en-us/quantum/development-kit`.
[6] Amazon: Amazon Braket, `https://aws.amazon.com/braket`.
[7] Google: Google Quantum AI, `https://quantumai.google/`.
[8] Guerreschi, G. G., Hogaboam, J., Baruffa, F. and Sawaya, N. P. D.: Intel Quantum Simulator: a cloud-ready high-performance simulator of quantum circuits, *Quantum Science and Technology*, (online), DOI: 10.1088/2058-9565/ab8505 (2020).
[9] Smelyanskiy, M., Sawaya, N. P. D. and Aspuru-Guzik, A.: qHiPSTER: The Quantum High Performance Software Testing Environment, *arXiv:1601.07195* (2016).
[10] qHiPSTER (Source Code): `https://github.com/intel/Intel-QS`.
[11] Jones, T., Brown, A., Bush, I. and Benjamin, S. C.: QuEST and High Performance Simulation of Quantum Computers, *Scientific Reports*, Vol. 9, No. 1, p. 10736 (online), available from ⟨https://doi.org/10.1038/s41598-019-47174-9⟩ (2019).
[12] QuEST (Source Code): `https://github.com/aniabrown/QuEST`.
[13] Häner, T. and Steiger, D. S.: 0.5 Petabyte Simulation of a 45-qubit Quantum Circuit, *Proceedings of ACM SC 2017* (2017).
[14] ProjectQ (Source Code): `https://github.com/ProjectQ-Framework/ProjectQ`.