Vol.2020-QS-1 No.4
2020/10/16

キャッシュブロッキングを利用した効率的な量子計算機シミュレーションの並列化

土井 淳[1,a)]　堀井 洋[1,b)]

概要：量子計算機のハードウェアやソフトウェアの開発において実機を用いる以外に古典計算機におけるシミュレーションを併用することで開発効率を高めることができる．シミュレーションを利用することで，ノイズや不安定さの無い理想的な環境でアルゴリズムの開発を行ったり，実機では扱えないような大きな問題を実験したりといった活用方法がある他，限られた実機の計算機資源を共有するのに比べてシミュレーションは待ち時間なしに利用できる利点がある．しかしながら古典計算機において大きな量子ビット数を扱おうとすると必要なメモリサイズが指数関数的に増大するため，計算機クラスター等を用いて分散メモリ並列化を行い大きなメモリ空間を利用できるようにする必要がある．シミュレーションの並列化において，性能のボトルネックとなるのは分散メモリ間のデータ交換である．効率よく並列化を行うためにはデータ交換の回数を減らす工夫が必要となる．そこで，我々はキャッシュブロッキングを用いてデータ交換を減らす手法を提案する．この手法では使用する量子ビットをローカルメモリに収まる数に制限し，量子回路上でスワップゲートを挿入することですべての量子ゲートをローカルメモリ内に移動することで少ないデータ交換で実行できるようにする．本発表ではオープンソースの量子計算機シミュレーターである Qiskit Aer に分散メモリ並列化を実装しその効果を検証する．

1. はじめに

量子計算機は組み合わせ最適化問題のように計算量が爆発的に増大するような，従来の古典計算機では扱うことが困難であったような問題を解くことができる可能性を持った次世代の計算機として期待されている．近年，扱える問題や規模に制限があるものの実際に動作する汎用ゲート型の量子計算機が登場しつつあり，研究開発が活発になってきている．しかしながら，現状では誰でも自由に利用できるというような計算機リソースが十分に用意されているという状態ではなく，またノイズ等によって安定した結果を得るために工夫が必要であり，実用的にはまだ従来の計算機に代わるようなレベルには達していない．そこで，量子計算機のためのアルゴリズムやソフトウェアの開発は効率よく行うために，古典計算機上で量子計算機シミュレーターを用いる方法が広く用いられる．量子計算機シミュレーターは，古典計算機のメモリ上に量子状態を記録し，量子ゲート演算による量子状態の変化をシミュレートする．このときノイズのない状態でもシミュレーションが可能なため，理想的環境における量子アルゴリズムを開発/テストすることも可能である．

古典計算機において量子状態は倍精度の複素数の確率振幅として記録し，n-qubit の量子計算をシミュレートするには長さ 2^n の配列が必要であり，すなわち，16×2^n バイトのメモリが必要である．同時に量子ゲート演算の計算量も $O(2^n)$ であり，必要なメモリサイズと計算量ともに，量子ビット数に対して指数関数的に増大する．このとき，1 台の古典計算機のメモリでは 20〜30 qubit 程度の量子状態を記録するのが精一杯であり，これ以上大きな qubit を持つ量子回路のシミュレーションを行うには，複数台の古典計算機を接続したクラスターを用いる必要がある．そのため，量子計算機シミュレーションの分散メモリ並列化を行う必要がある．[1]

また，量子計算機シミュレーションを高速に実行するための一つの方法に GPU を利用する方法がある．GPU は消費電力量あたりの計算量の観点から近年多くの計算機に採用されており，多くの計算アルゴリズムを高速化するのに広く用いられている．量子計算機シミュレーションでも，指数関数的に増大する計算量を扱うために GPU による高速化が適していると考えられるが，GPU に搭載されているメモリ容量は一般的な計算機のメモリ容量よりも比較的小さく，大きな量子ビット数の問題を扱うにはやや不利で

1 IBM Quantum, IBM Research - Tokyo
19-21, Nihonbashi Hakozaki-cho, Chuo-ku, Tokyo 103–8510, Japan
a) doichan@jp.ibm.com
b) horii@jp.ibm.com

ⓒ 2020 Information Processing Society of Japan

ある．例えば NVIDIA の Tesla V100 ?の場合 16GB または 32GB のメモリが使用できるが，16GB の場合最大で容量の半分までを量子状態を保存するために利用できるとして 29-qubit の量子計算まで実行可能である．これ以上の規模の問題を扱う場合，やはり複数枚の GPU が必要となる．あるいは，CPU のメモリに一部の量子状態を保存する必要がある．[1]

このように，複数枚の GPU や複数の計算機ノードに分散されて保存された量子状態を用いてシミュレーションを行うには，互いにデータを参照するために分散メモリ間でデータの交換を行う必要が生じる．このようなデータ交換の通信速度は一般的にメモリアクセス速度に比べると非常に遅く，シミュレーションの実行時間の増大やスケーラビリティの低下の原因となりうる．そこで，本論文では，通信量を削減する方法として，入力された量子ゲート回路について，本来実行するにはデータの交換が必要となるようなゲートを事前にスワップゲートを挿入することでデータ交換無しで実行できるようにする方法を提案する．この手法では，スワップゲートを用いることで，ゲートを実行するときにデータ通信の必要の無い領域に必要なデータを移動し，実行に関係ないものと入れ替えて退避するという，古典計算機におけるキャッシュブロッキングに似た操作を行う．

本論文で提案する手法は，オープンソースソフトウェアである Qiskit Aer [2] に複数 GPU および複数計算ノードを用いた分散メモリ並列化行った．GPU プログラミングには，オープンソースソフトウェアの保守性や生産性を考え，Thrust ライブラリ [3] を用いた．また MPI を用いてクラスター上での複数ノード間の分散メモリ並列化を行った．最後に，実際の HPC クラスターを用いて，キャッシュブロッキングを用いた場合と用いない場合でのシミュレーション時間の比較を行った．

2. 量子計算機シミュレーション

2.1 量子計算機シミュレーション概要

量子回路は量子ゲートの組み合わせでアルゴリズムを作り量子状態を操作していく．量子状態は確率振幅の重ね合わせとして表現され，最終的にこれらを観測することで，それぞれの量子ビットが 0 か 1 として確立する．量子状態は n-qubit のテンソル積の形で表し，従来の計算機上では，図 1 のように 2^n の長さの複素数の配列，量子状態ベクトルとして保存する．

量子ゲートは量子状態を変化させるような演算で，シミュレーションでは配列内の複素数の確率振幅の組に作用させる．代表的な量子ゲートに次の式に示す u3 ゲートがあり，このゲートによってある量子ビットを回転させる効果がある．

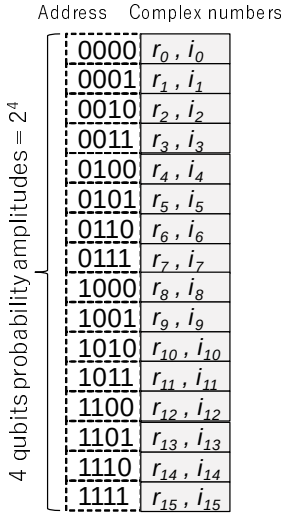

図 1 4 qubit の場合の量子状態を表す配列の例．長さ 16 の複素数配列として記憶する．

$$u3(\theta,\psi,\lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\psi}\sin(\theta/2) & e^{i(\psi+\lambda)}\cos(\theta/2) \end{pmatrix}. \quad (1)$$

シミュレーション上では，図 2 に示すように，2x2 の複素数行列をベクトルに乗じる処理になる．ここで k-qubit について u3 ゲートを適用する場合，ベクトル成分となる 2 つの確率振幅の組は，配列のアドレスに 2^k の XOR をとることで求めることができる．

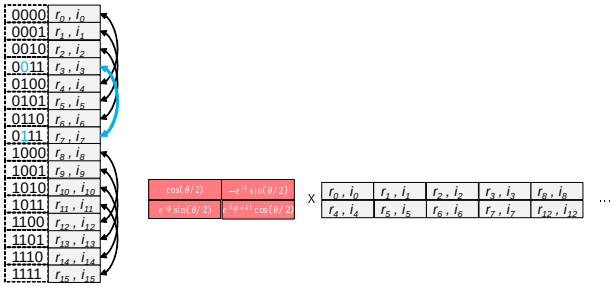

図 2 k=2 についての u3 ゲートの計算の例．2^k 離れた 2 つの確率振幅の組をベクトルとし 2x2 の行列を乗じる．

このようなゲートをシミュレーションするには 2^n の長さの配列のすべての要素を更新するため，1 つの量子ゲートをシミュレーションするためにはすべての配列要素についてアクセスする必要がある．このとき，確率振幅が分散メモリ空間に保存されていると，k が小さい場合はそれぞれの分散メモリ上で独立に並列的に計算することが可能であるが，k が大きい場合は，異なるメモリ空間に相手となる確率振幅が保存される場合があり，その場合は分散メモリ間でデータ交換を行う必要が生じる．

2.2 Qiskit Aer

Qiskit(Quantum Information Science Kit) [4] は，オープンソースな量子計算のための統合環境であり，量子計算

を行うためのプログラミング環境，ライブラリ，実機およびシミュレーションを実行するためのツール群から構成され，誰でも簡単に量子計算を始めることができるようになっている．

Qiskit Aer [2] は古典計算機の上で量子計算機シミュレーションを行うためのフレームワークで，量子計算機実機で実行するのと同じプログラムを Qiskit Aer 上でも実行することが可能である．Qiskit Aer にはいくつかのシミュレーターの実装があるが，本研究では，ここまで説明してきた量子状態ベクトル型 (statevector) のシミュレーターについて並列化を行う．

3. 量子計算機シミュレーションの GPU による高速化

3.1 Thrust ライブラリ

Qiskit Aer は C++で実装されており，すべての処理はクラス内に定義される．通常の CUDA のプログラミングでは，GPU カーネルはグローバルな関数として書かれるため，Qiskit Aer の開発ポリシーには合わない．そこで，CUDA によって直接カーネルプログラムを書く代わりに，Thrust ライブラリ [3] を利用する．Thrust は Standard Template Library(STL) 互換の機能を持つクラスライブラリで，配列の管理や，GPU に最適化された配列に対する基本的な操作などの機能を持っている．比較的直感的にプログラミングが可能で，一般的な CUDA プログラムよりも可読性が高く，高い保守性が得られると思われる．筆者の感覚的には，CUDA FORTRAN に非常に近いプログラミングが可能で，とても簡単に GPU を利用できると感じた．

3.2 Thrust による GPU による高速化

Thrust ではラムダ式を用いることで比較的簡単に GPU カーネルを実装することができる．単純な逐次アクセスな関数を作るだけであれば，配列を渡して式を書くだけで実装できるが，量子計算機シミュレーションでは与えられたインデックスから確率振幅の組を取り出して計算しなくてはならないため，逐次に与えられるインデックスから，アドレスに変換する部分を自分で書く必要がある．

逐次に計算を行うための `thrust::for_each` に渡す関数を自分で定義するためにはクラスを定義し，operator に実装を書く．`operator` の引数に逐次インデックスを受け取るようにし，インデックスからアドレスを計算する．operator 内部からは継承したクラスのメンバ変数に設定した値のみ参照でき (CUDA におけるカーネル関数の引数にあたえられるイメージ)，ここに実際の量子状態ベクトルのポインタを持たせることで，逐次アクセスではなくても任意の位置のデータにアクセスができるようになる．u3 ゲートで用いる 2x2 の行列は，ここではそれぞれ 4 つの複素数の定数として渡す．これらはコンストラクタで設定することが可能である．

次のコードは，u3 ゲートの実装である．

```
struct u3_lambda : public unary_function<int,void>
{
 complex<double>* pVec;
 complex<double> m0,m1,m2,m3;
 int qubit;
 int add;
 int mask;
 u3_lambda(complex<double>* pV,
           complex<double>* pM,int q)
 {
  pVec = pV;
  qubit = q;
  m0 = pM[0];
  m1 = pM[1];
  m2 = pM[2];
  m3 = pM[3];
  add = 1ull << qubit;
  mask = add - 1;
 }
 __host__ __device__
 void operator()(const int &i) const
 {
  int i0,i1;
  complex<double> q0,q1;
  i1 = i & mask;
  i0 = (i - i1) << 1;
  i0 += i1;
  i1 = i0 + add;
  q0 = pVec[i0];
  q1 = pVec[i1];
  pVec[i0] = m0 * q0 + m2 * q1;
  pVec[i1] = m1 * q0 + m3 * q1;
 }
};
```

また，以下のコードは，u3 ゲートを GPU で実行する場合の実行方法を示す．

```
auto ci = thrust::counting_iterator<int>(0);
int n = 1 << (nqubits-1);
thrust::for_each(thrust::device,ci,ci+n,
                 u3_lambda(vec,mat,qubit));
```

他の量子ゲートについても同様に実装を行った．

4. 量子計算機シミュレーションの並列化と最適化

4.1 チャンクを利用した分散メモリ並列化

量子状態を保存するのに 1 つのメモリ空間では十分に足

りない場合，確率振幅の配列を複数のメモリ空間に分散して保持する必要がある．元の配列を分割して，各メモリ空間に分配すればいいのだが，中途半端な位置で区切ってしまうと参照したい確率振幅の値が別のメモリ空間に配置されてしまったり不都合を生じるため，分割する際は必ず，2 のべき乗の位置で区切るようにする．しかしながら，図 3 のように，2 のべき乗に限定してしまうと，メモリ空間の数も 2 のべき乗である必要が生じて自由度が低くなってしまうのと，その単位で扱うと，データ交換を行う際に必要なバッファの分倍のメモリサイズが必要になってしまうので工夫が必要である．そこで，図 4 のようにチャンクと呼ばれる，2 のべき乗のサイズのさらに小さい単位で処理するようにする．すなわち，各メモリ空間は，2 のべき乗の倍数のサイズを持つことになり，メモリ空間の数は任意で，かつ各メモリ空間のサイズはバラバラでも良くなり柔軟性が増す．

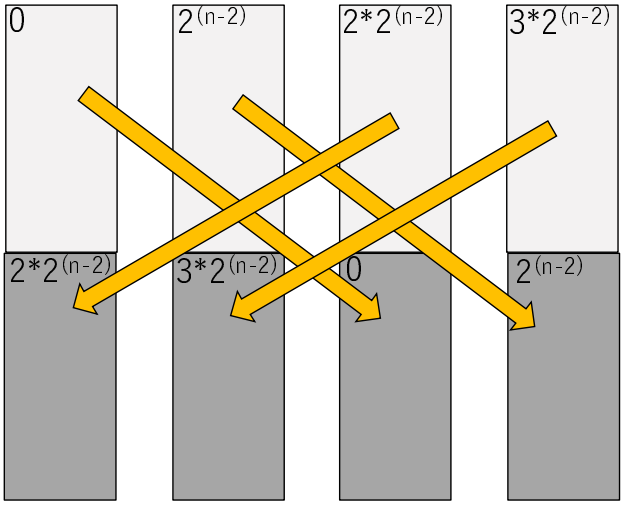

図 3 分散メモリ空間に 2 のべき乗境界で確率振幅を分配した例．この例では $k = n - 1$ qubit のゲートを適用しデータ交換を行っているが倍のメモリ量を必要とする．

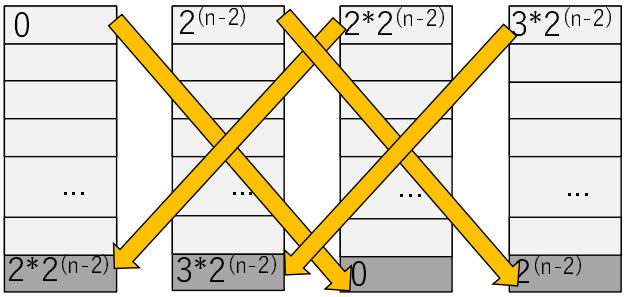

図 4 さらに小さいチャンクの単位で分割し，複数チャンクを分散メモリに分配した例．データ交換に使用するバッファはチャンクと同じサイズで済む．

チャンクの qubit 数を nc とすると，各チャンクは 2^{nc} 個の確率振幅を持つ．すなわち，全体の qubit 数が n のとき，全体のチャンク数は 2^{n-nc} 個である．このとき，ゲートの qubit 数が k とすると，$k < nc$ のとき，データの参照はチャンク内にとどまるため，データ交換は必要ない．それ以外の場合，チャンク間でデータを参照しあう必要があるが，同じメモリ空間内に相手のチャンクがあれば直接データを参照できる．

4.2 キャッシュブロッキングによる最適化

チャンクの qubit 数 nc よりも，ゲートの qubit 数 k が小さいときは，チャンク内で処理が完結するため，それぞれのチャンクは独立した量子状態とみなして，並列に処理を行うことができる．つまり，すべてのゲートの qubit 数が $k < nc$ となるようが量子回路が用意できれば，チャンク間のデータ交換無しで，シミュレーションを行うことが可能となる．Qiskit Aer では，シミュレーションを実行する前に，入力された量子回路を最適化する前処理を行うことができる．この仕組みを利用して，入力された量子回路について，すべてのゲートが $k < nc$ を満たすようにゲートを移動するような最適化を実装した．図 5 のように $nc + 1$ にゲートがある場合このままだと，このゲートを実行するには，チャンク間でデータ交換を行う必要がある．そこで，nc よりも小さな qubit にゲートを移動させたい．ゲートを移動するには，図 6 のように，スワップゲートを挿入することでチャンク内部で処理できるようにすることができる．もちろん，このとき qubit 2 に他の量子ゲートが存在すれば，qubit $nc + 1$ に移動されてしまうので，再度スワップゲートを挿入してチャンク内に戻してあげる必要がある．なお，このとき挿入されるスワップゲートは，一般的な量子スワップゲートとは区別する必要があり，シミュレーターのためのノイズを生じない特別なゲートとして扱うこととする．

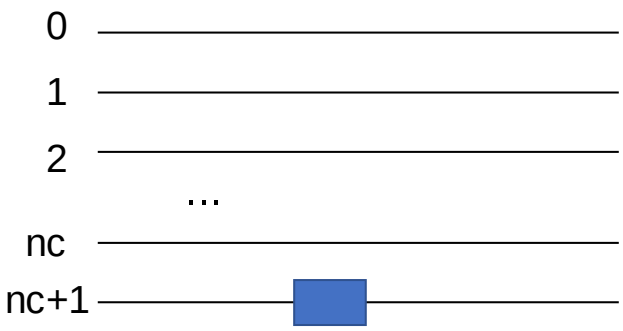

図 5 量子回路にチャンクの qubit nc 以上の qubit に量子ゲートがある場合の例．

すべての量子ゲートについて同様にスワップゲートを挿入することで，データ交換無しで量子ゲートをシミュレートできるようになるが，当然，スワップゲートの処理は，

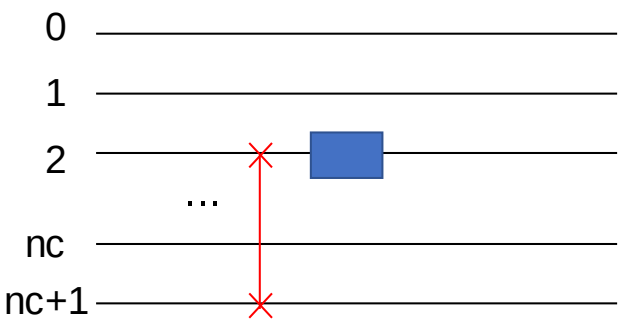

図 6　qubit $nc+1$ と 2 の間にスワップゲートを挿入することで，量子ゲートを qubit 2 に移動する．

チャンク間でデータを交換する処理そのものであるので，挿入されるスワップゲートの数が多いと逆に処理時間が増大してしまう．したがって可能な限り少ないスワップゲートの挿入ですべての量子ゲートをチャンク内に移動することが望まれる．つまり，可能な限り連続するゲートをまとめて移動し，チャンク内で一気に実行してしまうような操作が望ましい．このように，チャンク外にある処理の遅いゲートの集合を，チャンク内に移動して一気に速く処理する，という様子が古典計算機におけるキャッシュブロッキングに似ていることから，我々はこの操作をゲートのキャッシュブロッキングと呼ぶ．

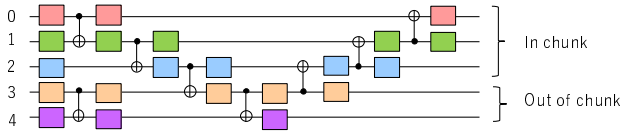

図 7　5 qubit の量子回路の例．ここではチャンクの qubit を $nc=3$ とすると，qubit 3 と 4 上にある量子ゲートを実行するにはチャンク間でデータ交換が必要である．

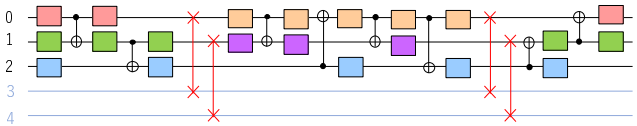

図 8　スワップゲートを入れて，キャッシュブロッキングを行った例．

図 7 のような量子回路が入力されると，図 8 のようにすべての量子ゲートがチャンク内に移動された量子回路が出力されるような前処理を行う．これによりすべての量子ゲートはチャンク間でデータ交換を行うことなくシミュレートできるようになる．この例のように，スワップゲートとスワップゲートの間に連続する量子ゲートは，あるチャンクについて一度に処理を適用する．こうすることで，CPU のメモリのように比較的遅いメモリ空間に保存されているチャンクを GPU のような比較的速いメモリ空間に，文字通りキャッシュさせて高速に処理することが可能となる．連続する量子ゲートの数が多いほど，キャッシュに移動するコストの割合を減らせるので，効率が良くなる．

このような前処理は，時間を書ければ最適解を求めることが可能であるが，あくまでもシミュレーションを速くするための前処理であるのであまり時間をかけるのは望ましくない．そこで，次を満たすようにヒューリスティックに処理する．(i) 可能な限り少ないスワップゲートの挿入にする (ii) スワップゲート間にできるだけ多くの量子ゲートを入れる (iii) スワップゲートはまとめて実行する．

Qiskit Aer では，量子回路は，量子ゲートの列として与えられる．量子ゲートは与えられた必ずしも順番通りに実行する必要はなく，異なる qubit 上の量子ゲートは実行する順番を変えても問題無い．ただし，CNOT ゲートのように複数 qubit にまたがる量子ゲートがあった場合，以降の関連する qubit 上にある量子ゲートは，複数 qubit を持つ量子ゲートを実行してからでないと実行できない．この特徴を利用して，複数 qubit を持つ量子ゲートをトリガーとして，スワップゲートを挿入していく．

まず，量子ゲートの列を，キュー 1 に入れておく．キュー 1 からゲートを取り出し，チャンクの内側であればそのまま出力し，外側であれば，キュー 2 に退避しておく．複数 qubit を持つ量子ゲートが現れた場合，複数 qubit がチャンクの内外をまたぐようなものだった場合，このゲートはキュー 2 に追加し，このゲートに関連する qubit を次回チャンク内に入れる qubit の候補として記録する．以降，これらの qubit 上のゲートは実行できないので，それらの qubit はブロックする．ブロックされた qubit 上のゲートが現れたらキュー 2 に退避させる．また，複数 qubit のゲートがブロックされた qubit 上にあった場合，残りの qubit も波及されてブロックされる．次の qubit の候補の数がチャンクの大きさ nc に達するまで処理を進め，候補になる qubit がチャンク内に入るようにスワップゲートを出力し，キュー 2 内のゲートの qubit をそれに応じて変換する．この処理の疑似コードを以下にまとめる．

```
量子ゲートの列を queue1 に入れる
queue1 が空になるまで繰り返す
  queue1 が空になるまで繰り返す
    gate = pop queue1
    if gate が複数 qubit
      if gate の qubit がブロックされている
        push gate to queue2
        関連する qubit をブロックする
      else if チャンクの外側にまたがる
        push gate to queue2
        関連する qubit を次回のチャンクの候補に保存
        関連する qubit をブロックする
      else
        gate を出力
```

```
    else
      if gate の qubit がブロックされている
        push gate to queue2
      else
        if qubit はチャンク内
          gate を出力
        else
          push gate to queue2
  スワップゲートを出力する
  queue2 内の qubit をスワップ後の qubit に変換する
  queue1 = queue2
```

5. 性能評価

5.1 計算機環境

本論文で提案するキャッシュブロッキングによる並列化の最適化を用いた量子計算機シミュレーションについて，表 1 に示す，GPU を搭載した並列計算機を用いて，従来手法 (baseline)[1] と比較を行った．計算機ノードは，IBM Power System AC922 [5] を用いた．各ノードあたり，6 つの NVIDIA Tesla V100 [6] が搭載されている．GPU あたり 16GB のメモリが搭載されているため，最大で GPU あたり 29 qubit までの量子回路をシミュレートできる．また CPU のメモリは 512GB であり，CPU を用いた場合 34 qubit までシミュレート可能であるが，図**??**のように，CPU と GPU の両方のメモリを利用すると 1 qubit 追加して，35 qubit までシミュレート可能となる．

表 1 性能評価に使用する計算機環境

Cluster node	IBM Power System AC922
CPU	POWER9
Number of sockets per node	2
Number of cores per socket	21
CPU memory size	512GB
GPU	NVIDIA Tesla V100
Number of GPUs per node	6
GPU memory size	16GB
CPU - GPU interconnect	NVLink2
Inter connect	Infiniband EDR
OS	Red Hat Enterprise Linux Server 7.6
Compiler	GCC 8.3.0
CUDA Toolkit	CUDA 10.1
MPI	IBM Spectrum MPI 10.3.1

ここでは，ベンチマークとして 2 つの量子回路を用いた．1 つ目は Quantum Volume [7] と呼ばれるランダムに生成された回路で，アルゴリズムとして意味のある回路ではないが，すべての qubit 間で CNOT ゲートによる量子もつれを生じさるような回路であり，量子計算機の性能や安定性等を評価するのに役立つ回路である．2 つ目は Quantum Fourier Transform (QFT) で，多くのアプリケーションで使用されるフーリエ変換のアルゴリズムである．

5.2 シングルノードの性能評価

図 9 および図 10 は，Quantum Volume および QFT の回路を 1 ノードの IBM Power System AC922 を用いてシミュレーションした場合のシミュレーションにかかる実行時間をまとめたものである．CPU のみを用いて実行した場合と，GPU を利用した場合では従来手法と提案手法の処理時間を比較する．

グラフを見ると，三段階に処理時間のパターンが変化しているのが分かる．最初の 29 qubit までの区間は GPU1 枚にデータが収まるため，メモリ空間をまたぐデータ交換が生じない．そのため，従来手法と提案手法で性能の差は無い．ただし，QFT においては比較的計算時間が律速するため状況が異なり，1 GPU に収まる問題サイズであっても，提案手法を用いて複数 GPU を用いて処理を分散した方が高速になる．図 10 で，28〜29 qubit について，提案手法では 1 GPU ではなく，6 GPU を利用した結果をプロットしている．

次に，30〜32 qubit のときは，6 GPU を使用しており，GPU のメモリ上にデータが収まった状態である．提案手法では従来手法に比べてデータ交換の回数が少なくなっているため，29 qubit までの処理時間の傾向とほぼ同じように増加していっているのが分かる．一方従来手法では Quantum Volume において GPU 間のデータ交換によって性能が大きく低下しているのが分かる．しかしながら QFT ではデータ交換の処理時間の割合が比較的小さいので提案手法より若干遅い程度となった．

最後に，33 qubit 以上のときは，6 つの GPU を用いてもデータを保持できないので，CPU のメモリも併用して処理を行っている．従来手法においては，CPU にチャンクがある場合，そのチャンクは CPU で処理していた．提案手法では，ある程度まとまった数のゲートを連続して同じチャンクに適用できるようになったため，CPU にあるチャンクも GPU へコピーして GPU で処理するようにしている．CPU と GPU のデータ転送時間に比べてゲートをシミュレーションするのにかかる時間が十分に大きければ，CPU で処理するよりも速くなるので，提案手法ではまとめてゲートを適用して処理時間を長くすることで良い結果を得られている．古典的な意味のキャッシュブロッキングが非常に有効に効いていると言える．特に図 10 の QFT では，計算時間律速であるので効果が大きい．

5.3 複数ノードによるスケーリングの評価

図 11 および，図 12 は，Quantum Volume を IBM Power System AC922 を 1 ノードから 16 ノードまで使用してシミュレートした場合の処理時間の比較をまとめたものである．ノードあたり 1 MPI タスクを使用している．これら

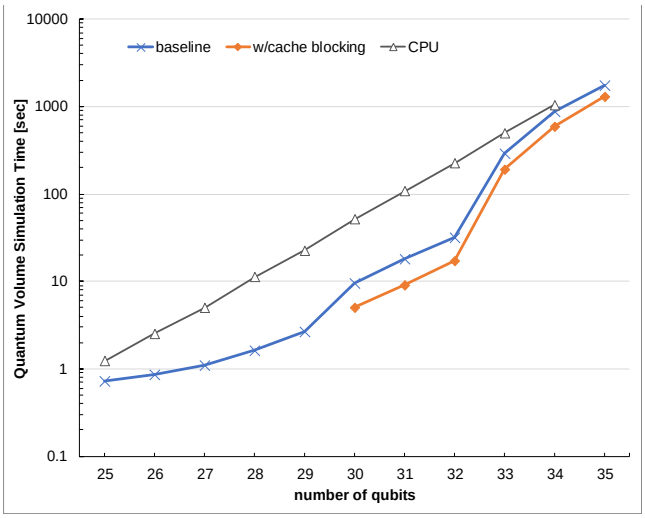

図 9　1 ノードの IBM Power System AC922（6 GPU）における Quantum Volume (depth=10) ベンチマークの実行時間の比較.

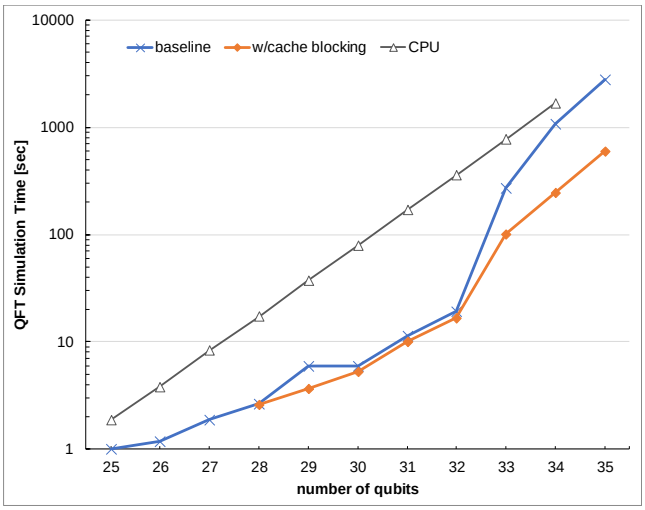

図 10　1 ノードの IBM Power System AC922（6GPU）における QFT の実行時間の比較.

は，問題サイズを固定した強スケーリングを評価している．図 11 は，30 qubit の Quantum Volume を使用した場合の結果であるが，30 qubit は 1 ノードのときですでに GPU 上にデータが乗り切るため比較的高速に処理ができている．そのため 2 ノードを用いた場合，ノード間のデータ交換のオーバーヘッドによってかえって処理時間が増大してしまっている．その後ノード数を増やしていくと処理時間は減少していくが 1 ノードを使用したときに比べてそれほど速くはならない．従来手法では特にデータ交換のオーバーヘッドが大きくスケーラビリティが悪いが，提案手法によって改善がみられる．

図 12 では，34 qubit の Quantum Volume を用いている．1 ノードでは CPU も併用しないと実行できない問題なので，この場合ノード数を増やすことによるノードあたりのデータサイズの減少と高速化が期待できる．しかしながら従来手法では期待した高速化が得られていない．提案手法では期待通りにノード数を増やすことで処理時間を減らすことができている．4 ノード以上を使用すると GPU 上にすべてのデータが保存できるのでスケーリングの傾きが変わっている．（データ交換のオーバーヘッドが比較的大きくなるため）

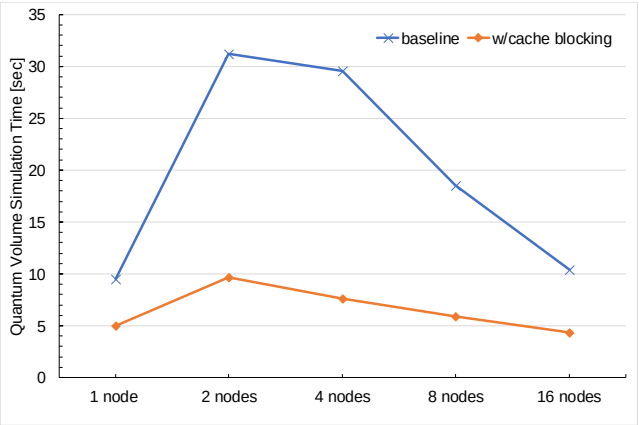

図 11　IBM Power System AC922（6 GPU）における 30 qubit の Quantum Volume (depth=10) ベンチマークの実行時間の比較．1 ノードでは 6 GPU をそれ以上では 1 GPU を使用している．

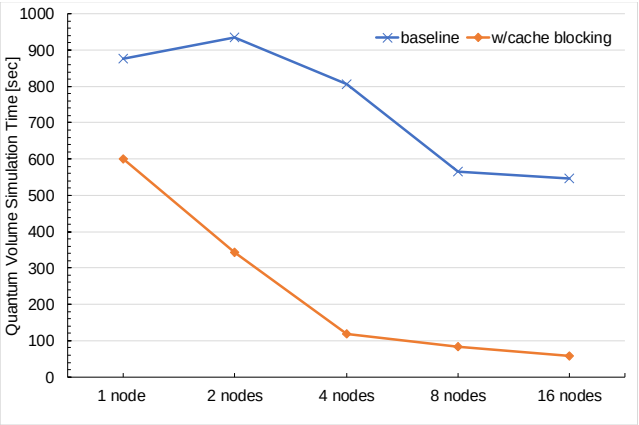

図 12　IBM Power System AC922（6 GPU）における 34 qubit の Quantum Volume (depth=10) ベンチマークの実行時間の比較．1 ノードと 2 ノードでは CPU を併用，それ以上では GPU のみを使用している．

図 13 および，図 14 は，Quantum Volume をノードあたりの問題サイズが一定になるように回路の qubit 数を調整して，IBM Power System AC922 を 1 ノードから 16 ノードまで使用してシミュレートした場合の処理時間の比較をまとめたものである．つまり，弱スケーリングを評価している．図 13 はノードあたり 2^{30} 個の確率振幅を扱うように問題を設定している．すなわち，1 ノードの時 30 qubit，4 ノードの時 32 qubit の回路についてシミュレーションしている．同じように，図 14 では，ノードあたり 2^{33} 個の確

率振幅を扱うように問題を設定している．弱スケーリングでは実行時間がノードを増やしても変わらないのが理想的であるが，一般的に通信のオーバーヘッドや並列化できない処理等が原因でノードが増えるごとに処理時間が増えていく傾向がある．従来手法ではどちらのサイズでもデータ交換のオーバーヘッドによりノードを増やした時の性能低下が大きいが，提案手法では性能低下を小さく抑えることができている．

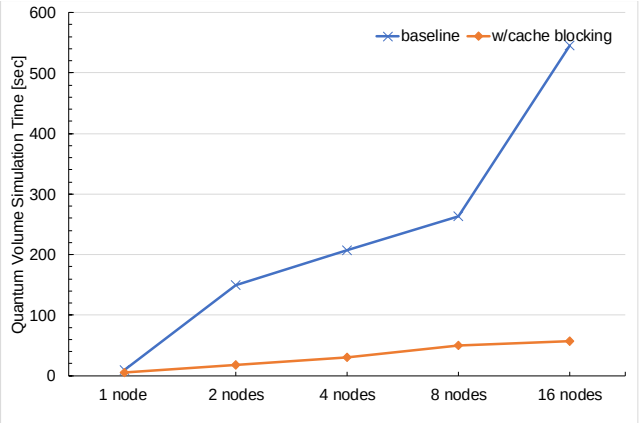

図 13 IBM Power System AC922（6 GPU）における Quantum Volume (depth=10) ベンチマークについて 1 ノードあたり 2^{30} 個の確率振幅を扱うように qubit 数を設定した場合の実行時間の弱スケーリングの比較．

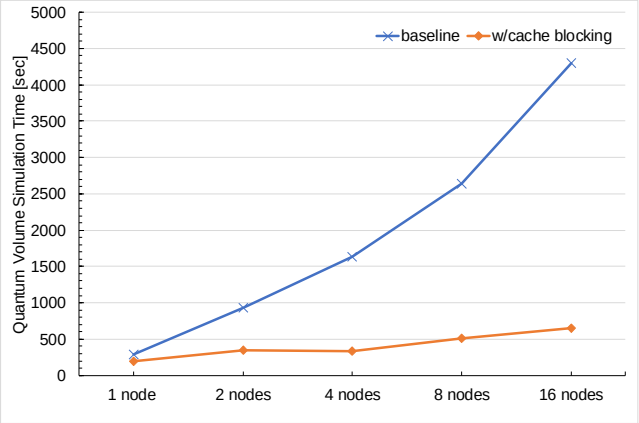

図 14 IBM Power System AC922（6 GPU）における Quantum Volume (depth=10) ベンチマークについて 1 ノードあたり 2^{33} 個の確率振幅を扱うように qubit 数を設定した場合の実行時間の弱スケーリングの比較．

6. まとめ

量子計算機に関するハードウェアやソフトウェアの開発において，量子計算機シミュレーションの需要は高まっているが，大きな問題を解こうとすると指数関数的にデータサイズや計算量が爆発するため，大規模な並列計算機を用いてシミュレーションを行うことが望まれている．しかしながら，並列計算を行うためには，分散メモリ間でデータを交換する必要があり，データ通信のオーバーヘッドにより，思ったように並列処理による効果が得られないことがある．本論文では，チャンクと呼ばれる 2 のべき乗のサイズの配列単位で処理を行い，チャンク単位でのキャッシュブロッキングを行い，通信によるオーバーヘッドを軽減する手法を提案した．チャンクの qubit 数より小さい qubit に適用するゲートはチャンク内で処理を完結できることを利用して，シミュレーションの前処理において入力された量子回路の量子ゲートを全てチャンク内で処理できるようにスワップゲートを挿入することで，チャンク間のデータ交換の回数を減らすことができた．

実際の並列計算機において提案手法と従来手法についてシミュレーションの処理時間を比較したところ，提案手法を用いることによって，複数 GPU を用いた場合，複数計算ノードを用いた場合ともに，大きな性能の向上が確認できた．特に大きな qubit の回路を扱う場合，GPU と CPU のメモリ共にチャンクを保存する場合，従来手法では CPU で処理していたものを，提案手法ではチャンク単位で GPU にキャッシュブロッキングできるようになったことで大きく性能が向上された．また，強スケーリング，弱スケーリング共に，提案手法によって向上できることが確認できた．

今後の展望としては，提案手法であるキャッシュブロッキングは，ファイル上にチャンクを保存する場合にも有効であると考えられる．よって，並列計算機とネットワークストレージを組み合わせることで，より大きな qubit を持つ回路をシミュレーションできると考えられる．また，本論文で扱ったシミュレーション方法は確率振幅を直接アップデートしていく手法であるが，これ以外のシミュレーション手法についても，同様にキャッシュブロッキングによる並列化が有効であるか検討したい．

参考文献

[1] Doi, J., Takahashi, H., Raymond, R., Imamichi, T. and Horii, H.: Quantum Computing Simulator on a Heterogenous HPC System, *Proceedings of the 16th ACM International Conference on Computing Frontiers*, New York, NY, USA, Association for Computing Machinery, pp. 85–93 (online), DOI: 10.1145/3310273.3323053 (2019).
[2] Qiskit: Qiskit Aer: A High Performance Simulator Framework for Quantum Circuits (2019).
[3] : Thrust - Parallel Algorithms Library (2019).
[4] Qiskit: Qiskit: An Open-source Framework for Quantum Computing (2017).
[5] Vetter, S., Nohria, R. and Santos, G.: IBM Power System AC922 Technical Overview and Introduction, *An IBM Redpaper publication* (2018).
[6] Corporation, N.: NVIDIA Tesla V100 (2018).
[7] Cross, A. W., Bishop, L. S., Sheldon, S., Nation, P. D. and Gambetta, J. M.: Validating quantum computers using randomized model circuits, *Phys. Rev. A*, Vol. 100, p. 032328 (online), DOI: 10.1103/PhysRevA.100.032328 (2019).

ⓒ 2020 Information Processing Society of Japan