Oral Presentations | Track 2: Big Data Infrastructure Technologies, Security & Privacy

Thu. Feb 29, 2024 4:00 PM - 6:10 PM JST | Thu. Feb 29, 2024 7:00 AM - 9:10 AM UTC | T2-A オンライン（Zoom Events)

Emerging hardware / performance evaluation

座長:藤原 靖宏(日本電信電話株式会社)
コメンテータ:油井 誠(トレジャーデータ株式会社)

4:00 PM - 4:05 PM JST | 7:00 AM - 7:05 AM UTC
Introduction

4:05 PM - 4:30 PM JST | 7:05 AM - 7:30 AM UTC
[T2-A-6-01] サーバシステムの性能データ転送における効率化手法の評価と考察

*飯山 知香[1]、平井 聡[2]、山岡 茉莉[2]、福本 尚人[2]、小口 正人[1] (1. お茶の水女子大学、2. 富士通株式会社)

4:30 PM - 4:55 PM JST | 7:30 AM - 7:55 AM UTC
[T2-A-6-02] 量子ランダムアクセスメモリ上での関係データベースの構成法の一検討

*御手洗 陽紀[1]、合田 和生[2] (1. 東京大学 大学院情報理工学系研究科、2. 東京大学 生産技術研究所)

4:55 PM - 5:20 PM JST | 7:55 AM - 8:20 AM UTC
[T2-A-6-03] 永続メモリ向けロックフリー索引Bz木に関する研究

*中山 宗[1]、杉浦 健人[1]、石川 佳治[1]、陸 可鏡[1] (1. 名古屋大学大学院情報学研究科)

5:20 PM - 5:45 PM JST | 8:20 AM - 8:45 AM UTC
[T2-A-6-04] ラベルの出現頻度に着目したFPGAを用いた正規パス問合せの提案

*溝谷 祐大[1]、小林 諒平[2]、藤田 典久[2]、朴 泰祐[2]、天笠 俊之[2] (1. 筑波大学大学院理工情報生命学術院システム情報工学研究群情報学位プログラム　知識・データ工学研究室(KDE)、2. 筑波大学 計算科学研究センター)

5:45 PM - 6:10 PM JST | 8:45 AM - 9:10 AM UTC
[T2-A-6-05] 量子回路シミュレータの実行環境変化に伴う性能評価と特性比較

*青木 望美[1]、山崎 雅文[2]、平井 聡[2]、山岡 茉莉[2]、福本 尚人[2]、小口 正人[3] (1. お茶の水女子大学人間文化創成科学研究科博士前期課程理学専攻情報科学コース小口研究室、2. 富士通（株）、3. お茶の水女子大学理学部情報科学科)

サーバシステムの性能データ転送における効率化手法の評価と考察

飯山　知香† 　平井　聡†† 　山岡　茉莉†† 　福本　尚人†† 　小口　正人†

† お茶の水女子大学　〒 112–8610 東京都文京区大塚２丁目１－１
†† 富士通株式会社　〒 211–8588 神奈川県川崎市中原区上小田中４丁目１－１
E-mail: †{chika,oguchi}@ogl.is.ocha.ac.jp, ††{ahirai,yamaoka.mari,fukumoto.naoto}@fujitsu.com

あらまし　近年，多数台サーバの共有利用などに関する需要が増加している．これらのサーバの負荷分散などをリアルタイムで行うためには，各サーバの性能データ分析を低オーバヘッドで行う必要がある．そこで本研究では，ターゲットサーバからデータ解析用サーバへのデータ転送時のオーバヘッドを考慮した効率的な転送手法の開発を目的としている．本報告では，転送効率化手法として，データ解析用サーバ性能の改善，転送タイミングの制御，および転送側で CPU 負荷の低いコアでの転送を行う．また，各手法に関して，データ転送時間や CPU 負荷率の計測を行い，その効果を評価する．さらに，CPU に負荷をかけるベンチマークと転送処理を同時実行し，それぞれに与える影響を分析する．

キーワード　時系列データ，データ転送，CPU 負荷率，転送タイミング制御，転送コア制御

Evaluation and Consideration of Improvement Plans to Increase the Efficiency of Performance Data Transfer for Server Systems

Chika IIYAMA†, Akira HIRAI††, Mari YAMAOKA††, Naoto FUKUMOTO††, and Masato OGUCHI†

† Ochanomizu University
2–1–1 Otsuka, Bunkyou-ku, Tokyo 112–8610, Japan
†† Fujitsu Limited
4–1–1 Kamikodanaka, Nakahara-ku, Kawasaki-shi, Kanagawa 211–8588, Japan
E-mail: †{chika,oguchi}@ogl.is.ocha.ac.jp, ††{ahirai,yamaoka.mari,fukumoto.naoto}@fujitsu.com

Key words　Time-series data, data transfer, CPU load factor, transfer timing control, transfer core control

1. はじめに

近年，多数台サーバの共有利用や分散処理利用に関する需要が増えてきている．その際，各サーバのハードウェアレベルを含む詳細な性能データをリアルタイムで収集・分析・提示することで，より効果的な負荷分散やシステムのチューニングが可能になる．また，多数台サーバのデータを一元的にリアルタイムで分析する際，分析対象のサーバとその分析を行うサーバは分割され，サーバ間での性能データ転送が必要になる場合が多くある．さらに，時系列化された性能データはデータサイズが大きく，扱う際のオーバヘッドも大きくなる可能性がある．本研究では，この性能データとして，システム情報の中でも比較的詳細かつデータサイズの大きい，サーバの CPU コアごとに収集するデータを想定している．以上のことから，現在サーバ間でより効率的に性能データを転送する手法の実現が求められている．

2. 性能データ転送における課題と改善案

性能データ転送の効率化手法を検討するにあたり，[1] にてデータ収集を行うサーバと解析を行うサーバが分割された環境を想定して，データ収集・転送時のリソース利用量やオーバヘッドを計測した．その結果，複数コアの情報を 1 つのコアで転送する手法ではコア数に伴い転送処理の遅延が増大することが分かった．そこで [2] にて，1 つの CPU コアで行っていた転送処理を各 CPU コアで行うように並列化し，性能を評価・比較することで遅延の改善方法を検討した．その結果，解析を行うサーバの CPU コア数が少ないことが原因でデータ転送処理に遅延が発生している可能性が推察された．そのため，[3] にて，CPU コア数等の構成が異なる 2 台のサーバを使用して比較を行い，解析を行うサーバの性能差がデータ転送を行うサー

バでの処理に与える影響を分析した結果，解析を行うサーバの増強だけでは改善できないボトルネックがあることが推察された．また，解析を行うサーバを増強させた際の転送処理部分のCPU 負荷率が高く，転送処理と同一 CPU コア上で CPU 負荷ベンチマークを同時に実行させると少なからず互いに性能劣化が発生することが分かった．そこで本報告では，コアごとの転送タイミングをずらす転送タイミング制御を行い，転送効率が改善されるか分析した．加えて，転送側で CPU 負荷の低いコアでの転送を行う転送コア制御を行い，同時に動作しているベンチマークおよび転送処理において性能劣化が抑えられるか確認した．

3. 関 連 研 究

本研究と収集対象とする性能データが類似しているツールとして，Score-P と Vampir が挙げられる．Score-P(Scalable Performance Measurement Infrastructure for Parallel Codes) [4] というツールでは，主に並列 HPC アプリケーションの性能分析を行うことができる．Vampir(Visualization and Analysis of MPI Resources) [5] というツールでは，多数台の計算ノードのプロファイルデータや相互通信データなどを統合解析し可視化することができる．これらのツールでは，アプリケーションの実行後に，各ノードで収集した性能データをファイルベースで収集/分析/可視化するため，本研究が目指す実行時の性能データを時系列データとしてリアルタイムで収集・分析する手法とは異なる．既存研究 [6] では，スーパーコンピュータ京などの独自の CPU 性能データを汎用的なデータ形式に変換し，上記の Score-P や Vampir で扱えるようにするための手法が提案されている．分析しようとしている内容は本研究と類似しているが，データ収集と転送・分析を同時に行うことは困難である．

本研究とデータ収集時のオーバヘッド削減手法などの関連性が高い研究として，分散トレーシングという手法がある．クラウド環境で広く利用されているマイクロサービス・アーキテクチャに基づいたソフトウェアの詳細動作を解析することができる．分散トレーシングでは主にアプリケーションを対象領域として性能データを収集・分析しているのに対し，本研究では低レイヤを対象としている点が異なる．Google 社が 2010 年に発表した論文 [7] を発端に，Twitter 社が Zipkin [8] を，Uber 社が Jeager [9] を開発した．それぞれ OSS として公開され，多くのクラウド基盤で利用されている．

プロファイルデータをデータベースに取り込み分析する手法が本研究と類似している既存研究 [10] では，Google 社のデータセンターで行われているプロファイルデータの収集・分析手法が解説されている．既存研究 [11] では，上記手法を用いてデータセンターのワークロードが分析されている．数千台のサーバからランダムでプロファイルデータ収集を行い，その情報を元にアプリケーションチューニングを行うと共に，パフォーマンス監視やアプリケーション配備 (アフィニティ) 最適化にも利用している．本研究では CPU や OS 等の低レイヤの性能データをオープンなソフトウェア (OSS) を使用/改良して収集/転送/分析することを目指している．

4. 実　　験

4.1 評 価 環 境

環境構築として，データ収集用サーバ (以下収集サーバ) とデータ解析用サーバ (以下解析サーバ) を用意した．評価環境概要を図 1 に示す．収集サーバには，Linux Kernel の標準的なイベントデータ収集/トレース機能のフレームワークである perf [12] を使用した．また，CPU 負荷をかける性能ベンチマークとして，UnixBench [13] の Dhrystone，io 負荷をかける性能ベンチマークとして stress io を使用した．解析サーバには，収集した時系列データを管理する時系列 DB として InfluxDB [14] を使用した．InfluxDB については [15] で解説されている．事前に収集した stress [16] 実行時の perf record のデータをデータ転送プログラムを用いて転送し，解析サーバ上の InfluxDB に格納した．各使用技術のバージョンは，perf が 3.10.0，stress が 1.0.4，UnixBench が 5.1.3，InfluxDB が 1.8.3 である．

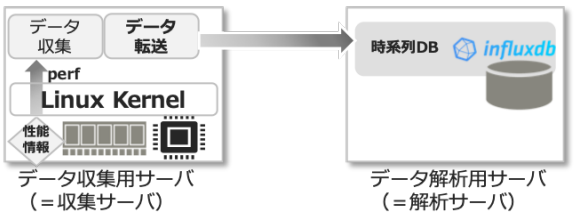


図 1　評価環境概要

解析サーバには，構成が異なる 2 台のサーバを用いて分析を行った．以降，CPU コア数，メモリ容量が多いサーバを高性能解析サーバ，少ないサーバを低性能解析サーバと記載する．収集サーバと高性能解析サーバの構成を表 1，低性能解析サーバの構成を表 2 に示す．各サーバのネットワーク構成図を図 2 に示す．

表 1　収集サーバ・高性能解析サーバ環境

機種名	Fujitsu PRIMERGY CX2550 M1
CPU	Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz (× 2CPU)
コア数	14
メモリ	128GB
OS	CentOS 7

表 2　低性能解析サーバ環境

機種名	Dell PowerEdge R620
CPU	Intel(R) Xeon(R) CPU E5-2603 v2 @ 1.80GHz (× 1CPU)
コア数	4
メモリ	16GB
OS	CentOS 7

収集した CPU 性能データを，InfluxDB に転送可能な形式に変換し，転送するまでの流れを以下に示す．まず収集データか

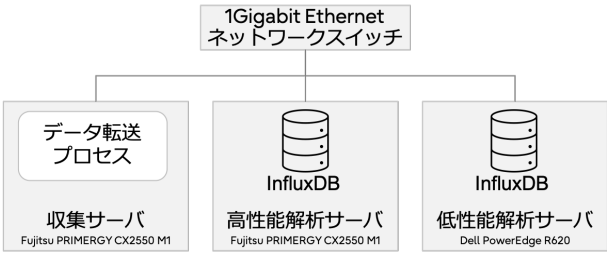


図 2　ネットワーク構成図

ら，必要なデータ (時刻，プロセス ID，スレッド ID，実行アドレス) のみを抽出する．1CPU コアの 1 秒分の収集データからは，100 マイクロ秒単位で収集した場合は 10000 レコードのデータが抽出される．この時，それぞれ 64bit のデータが 4 項目あるため，1CPU コアに関する転送データサイズは約 320KB となる．次に，抽出データを Python Pandas [17] を利用して DataFrame 形式に格納する．DataFrame 形式に格納したデータのサンプルを図 3 に示す．最後にこのデータを InfluxDB の Python Client モジュールを利用して解析サーバに転送する．上記モジュールには Pandas の DataFrame 形式データを InfluxDB サーバに書き込む API(Influxdb.DataFrameClient) [18] があり，今回はこれを利用している．

時刻(ns単位)	Process ID	Thread ID	EIP(実行アドレス)
2021-12-20T13:18:36.159557443	698085	698085	0x55e3e9dd1151
2021-12-20T13:18:36.160568219	698085	698085	0x7faa24bd7e02

図 3　DataFrame 形式データのサンプル

4.2　転送タイミング制御

4.2.1　実 験 内 容

転送タイミング制御の効果を検証するため，以下の実験 1〜3 の計測を行なった．実験 1 では，CPU コア数を変更しながらデータ転送時間を計測した．実験 1 で得られた結果の裏付けを取るため，CPU コア数 n=14 で実験 2〜3 を行なった．CPU コア数 n=14 で計測を行ったのは，コア数が多いほどオーバヘッドは分かりやすく，また表 1 に示すように収集サーバの 1CPU あたりのコア数は 14 であるためである．実験 2 では，データ転送処理部分の CPU 負荷率を計測した．実験 3 では，InfluxDB の CPU 負荷率を計測した．転送タイミング制御は，CPU コア数 n での転送において各コア i で転送前に i/n 秒 sleep し，コアごとの転送タイミングを 1/n 秒ずつずらすことで実装した．

4.2.2　実 験 結 果

実験 1 のデータ転送時間計測の結果を図 4 に示す．図 4 の各 CPU コア数における平均転送時間を表 3 に示す．図中の各点がコアを表している．表 3 ではスペースの都合上，低性能解析サーバはパターン 1，高性能解析サーバはパターン 2，低性能解析サーバ＋転送タイミング制御ありはパターン 3，高性能解析サーバ＋転送タイミング制御ありはパターン 4 と表記している．6 回計測を行い，初回転送のオーバヘッドによる影響を避けるため，2〜6 回目の結果のうち，全コアの平均のデータ転送時間が中間のものを用いた．低性能解析サーバおよび高性能解析サーバ使用時ともに，転送を実行するコア間で転送時間にばらつきがあり，CPU コア数の増加に伴いばらつきも増加していたが，転送タイミング制御を加えたことでそれらが減少し，CPU コア数の増加に伴う転送時間の増加が抑えられたことが読み取れる．また，転送タイミング制御を行うことで，低性能解析サーバ使用時においても，高性能解析サーバ使用時と同等の転送時間で転送できることが分かる．以上の結果から，データ転送時間の短縮において，転送タイミング制御は有用な手法であると推察される．

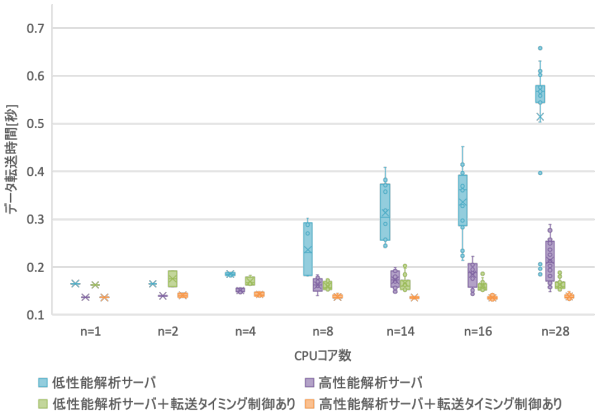


図 4　データ転送時間 (転送タイミング制御)

表 3　図 4 各 CPU コア数における平均転送時間

CPU コア数	パターン 1	パターン 2	パターン 3	パターン 4
n=1	0.17 秒	0.14 秒	0.16 秒	0.14 秒
n=2	0.17 秒	0.14 秒	0.18 秒	0.14 秒
n=4	0.19 秒	0.15 秒	0.17 秒	0.14 秒
n=8	0.24 秒	0.16 秒	0.16 秒	0.14 秒
n=14	0.31 秒	0.17 秒	0.17 秒	0.14 秒
n=16	0.34 秒	0.18 秒	0.16 秒	0.14 秒
n=28	0.51 秒	0.21 秒	0.16 秒	0.14 秒

実験 2 のデータ転送処理部分の CPU 負荷率計測の結果を図 5 と表 4 に示す．図 5 では各コアの CPU 負荷率とデータ転送時間を抽出した結果を示している．図中の各点がコアを表している．表 4 では表 3 と同様に，スペースの都合上，低性能解析サーバはパターン 1，高性能解析サーバはパターン 2，低性能解析サーバ＋転送タイミング制御ありはパターン 3，高性能解析サーバ＋転送タイミング制御ありはパターン 4 と表記している．6 回計測を行い，初回転送のオーバヘッドによる影響を避けるため，2〜6 回目の結果のうち，全コアの平均のデータ転送時間が中間のものを用いた．図 5 より，全てのパターンで共通して，転送時間が長くなるほど CPU 負荷率は低くなっていることから，コアごとの転送時間のばらつきは，転送処理中に発生した Idle 時間によるものであると推測できる．また，図 5 より，転送タイミング制御を加えることで，低性能解析サーバおよび高性能解析サーバ使用時ともに，転送時間および CPU 負荷率のばらつきが減少していることが読み取れる．以上の結果

から，転送タイミング制御による転送時間の短縮は，転送中の Idle 時間が減少したためであると推察される．表 4 より，最大 CPU 負荷率は 67.4%程度と高くなっていることが読み取れる．これによる同一 CPU コア上で動作する処理への影響が大きければ，転送用のコアを分けるなどの対応が必要になると考えられる．

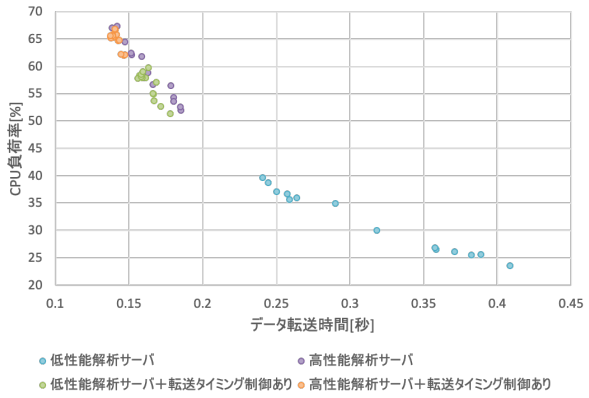


図 5　データ転送処理部分の CPU 負荷率 (n=14，並列転送版)

表 4　データ転送処理部分の CPU 負荷率 (n=14，並列転送版)

	パターン 1	パターン 2	パターン 3	パターン 4
平均	31.6 %	58.7 %	56.6 %	64.9 %
最大	40.0 %	67.4 %	59.8 %	66.8 %
最小	23.6 %	52.0 %	51.4 %	62.1 %

実験 3 の InfluxDB の CPU 負荷率計測について，低性能解析サーバでの結果 (パターン 1) を図 6，高性能解析サーバでの結果 (パターン 2) を図 7，低性能解析サーバ＋転送タイミング制御ありでの結果 (パターン 3) を図 8，高性能解析サーバ＋転送タイミング制御ありでの結果 (パターン 4) を図 9 に示す．図には 1 回計測の値を用いた．転送タイミング制御を加えたことで，低性能解析サーバおよび高性能解析サーバともに，CPU 負荷がかかる時間の範囲が広がり，各コアの CPU 負荷率の値は多少上下するが，CPU 負荷の総量はほとんど変化していないことが読み取れる．これは，転送元のコアにおいて転送タイミングが最大 1 秒程度ずれることで，転送先の InfluxDB でも負荷がかかるタイミングにずれが生じるが，転送されるデータ量に変化はないためであると推察される．よって，転送タイミング制御による転送時間の短縮は，InfluxDB 側の CPU 負荷率によるものではないと推察できる．

4.3　転送コア制御

4.3.1　実 験 内 容

転送コア制御の効果を検証するため，以下の実験 1～5 の計測を行なった．実験 1 では，CPU コア数を変更しながらデータ転送時間を計測した．実験 2～5 では，データ転送プログラムの CPU コア数は n=14 で計測を行なった．CPU コア数 n=14 で計測を行ったのは，転送タイミング制御の実験と同様，コア数が多いほどオーバヘッドは分かりやすく，また表 1 に示す

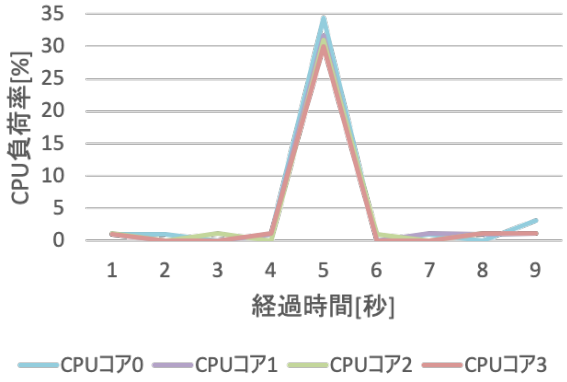


図 6　InfluxDB の CPU 負荷率 (n=14，並列転送版，低性能解析サーバ)

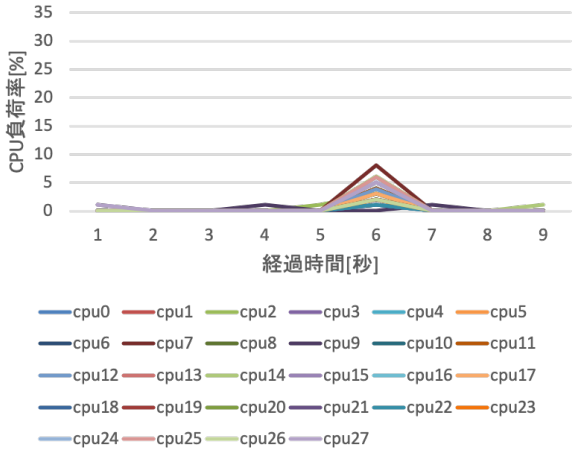


図 7　InfluxDB の CPU 負荷率 (n=14，並列転送版，高性能解析サーバ)

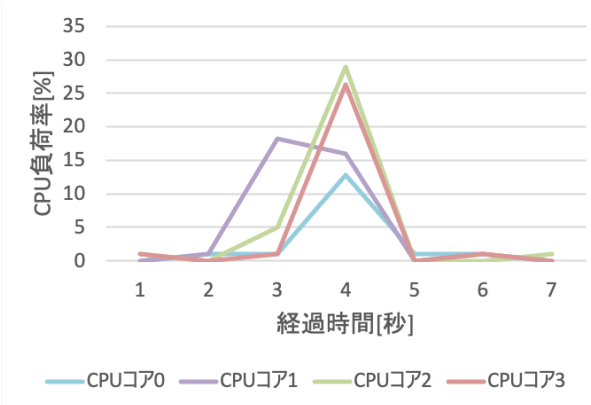


図 8　InfluxDB の CPU 負荷率 (n=14，並列転送版，低性能解析サーバ＋転送タイミング制御あり)

ように収集サーバの 1CPU あたりのコア数は 14 であるためである．実験 2 では，データ転送プログラムと CPU 負荷ベンチマークを同時に動作させた際の相互影響を計測した．実験 3 では，データ転送プログラムと CPU 負荷ベンチマークに加えて io 負荷ベンチマークも同時に動作させた際の相互影響を計測した．実験 4 では，実験 3 の計測時の CPU 負荷率を計測した．

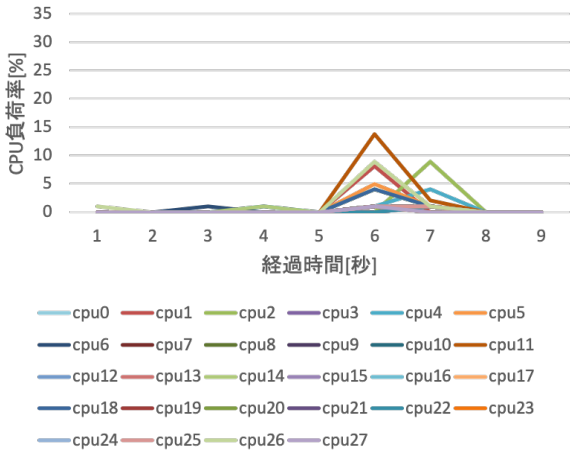


図 9　InfluxDB の CPU 負荷率 (n=14，並列転送版，高性能解析サーバ＋転送タイミング制御あり)

実験 5 では，実験 3 の計測時の各プロセスのコア遷移を計測した．転送コア制御は，転送側で毎秒 Idle 率の高いコアを検出し，該当コアで 4 コア分 (または端数分) のデータの転送を行わせることで実装した．そのため，転送は全コアではなく”n/4 の切り上げ”個のコアで実行される．該当コアで転送するデータ量は，1 コア転送版にて転送に 1 秒以上かからないコア数分とした．

4.3.2　実験結果

実験 1 のデータ転送時間の結果を図 10 に示す．図中の各点がコアを表している．CPU コア数 n=14 にて，3 つのコアでは 4 コア分，1 つのコアでは端数の 2 コア分のデータを転送しているため，一部転送時間が短くなっている．n が 4 以下の場合は 1 コア転送版と同等の転送時間になっていることが読み取れる．また，n が 4 より大きい場合はおおよそ n=4 の時と同等の転送時間になっているが, CPU コア数の増大に伴い転送時間および転送時間のばらつきが増大していることが読み取れる．これは，複数の Idle コアから同時に転送を行うようになることで，InfluxDB 側の負荷等により転送時の Idle 時間が増加するオーバヘッドが発生したためであると推察される．

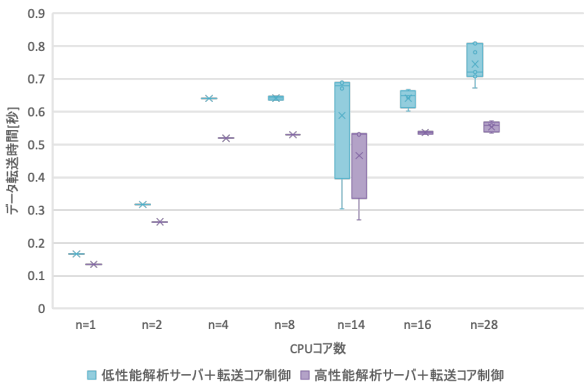


図 10　データ転送時間 (転送コア制御)

データ転送プログラムと CPU 負荷ベンチマークを同時に動作させた際の相互影響を計測した実験 2 の結果を表 5 に示す．データ転送プログラムの CPU コア数は n=14，つまり転送を実行するコアは 4 つで計測した．CPU 負荷ベンチマークとして，UnixBench のテストケースの一つで整数演算処理を行う dhry2reg を使用し，10 個のコアで実行した．データ転送プログラムとベンチマークプログラムをそれぞれ単独で動作させた場合と同時に動作させた場合を比較し，データ転送時間伸び率，ベンチマーク値劣化率，ベンチマーク実行時間伸び率を算出した．表には 3 回計測の平均の値を用いた．表 5 より，伸び率および劣化率が全て 1.00 であることから，同時動作においても互いにほとんど影響を与えていないことが分かる．これは，転送コア制御により CPU 負荷率の低いコア (ベンチマークの動作していないコア) が検出され，転送処理が実行されたためであると推察される．よって，転送コア制御を行うことで，転送処理とベンチマークの動作するコアを分割することが可能であり，その場合互いに性能劣化を起こさずに実行できることが分かる．

表 5　dhry2reg(10 コア) の動作時 (コア指定あり)

	データ転送時間	ベンチマーク値	ベンチマーク実行時間
単独動作	0.61 秒	26247.8	13.3 秒
同時動作	0.61 秒	26167.4	13.2 秒
伸び率/劣化率	1.00	1.00	1.00

データ転送プログラムと CPU 負荷ベンチマークに加えて io 負荷ベンチマークも同時に動作させた際の相互影響を計測した実験 3 の結果を表 6 に示す．CPU コア数 n=14 でのデータ転送プログラムと同時に，CPU 負荷ベンチマークである dhry2reg を 10 コア，io 負荷ベンチマークである stress io を 4 コアで実行させた場合の，dhry2reg とデータ転送プログラムの性能劣化を分析した．表には 3 回計測の平均の値を用いた．なお，ベンチマークが動作するコアは 14 コアの範囲内では指定せずに計測を行なった．表 6 より，転送処理は多少劣化しているが，dhry2reg ベンチマークではほとんど性能劣化が発生していないことが読み取れる．これは，CPU 負荷の低いコアを検出して転送を行わせる転送コア制御により，同一 CPU コア上で転送処理と stress io ベンチマークが同時に動作し，それらと異なるコア上で dhry2reg ベンチマークが動作したためであると推察できる．

表 6　dhry2reg(10 コア) と stress io(4 コア) の動作時 (コア指定なし)

	データ転送時間	ベンチマーク値	ベンチマーク実行時間
単独動作	0.61 秒	26247.8	13.3 秒
同時動作	0.62 秒	26187.5	13.2 秒
伸び率/劣化率	1.03	1.00	1.00

実験 4 では，実験 3 の計測時の各コアの CPU 負荷率を計測し，ベンチマーク特性を確認した．なお，ベンチマークが動作

するコアは 14 コアの範囲内では指定せずに計測を行なった．結果を図 11〜14 に示す．図 11 より，15〜24 秒でユーザレベルの CPU 負荷率が 100%近くなっていることが読み取れる．これは，dhry2reg ベンチマークは整数演算の繰り返し処理を行うためであると考えられる．また，図 12，13，14 より，システムレベルの CPU 負荷率やディスク I/O リクエスト待ち，idle 時間が多く発生していることが読み取れる．これは，stress io ベンチマークは sync システムコールによるディスクキャッシュ同期の繰り返し処理を行うためであると考えられる．

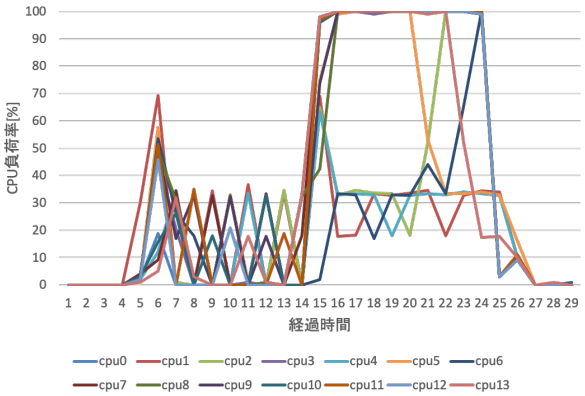


図 11　CPU 負荷率 (ユーザレベル)

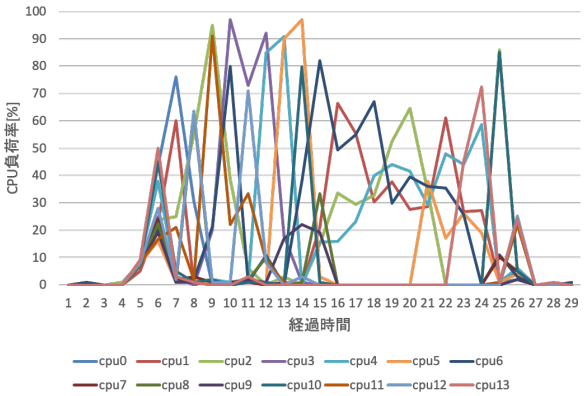


図 12　CPU 負荷率 (システムレベル)

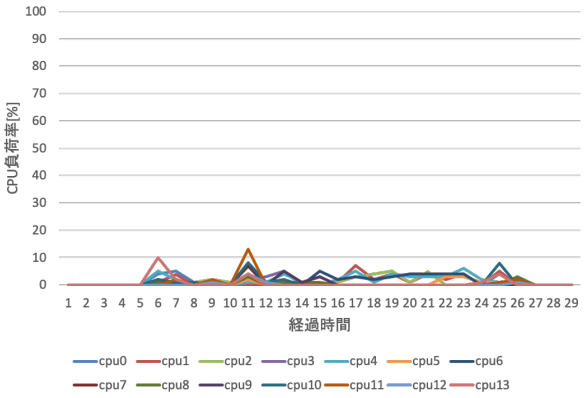


図 13　CPU 負荷率 (ディスク I/O リクエスト待ち)

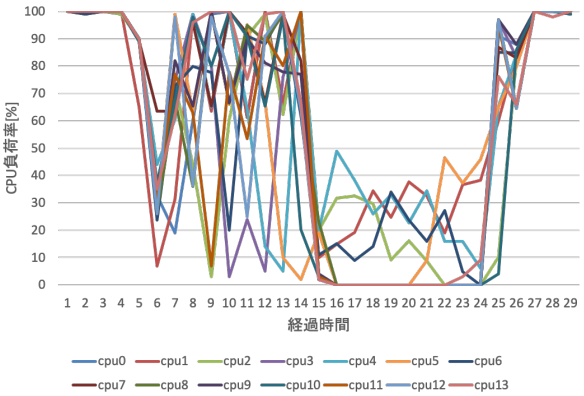


図 14　CPU 負荷率 (idle 時間)

実験 5 では，実験 3 の計測時の各プロセスのコア遷移を計測した．結果を表 7 に示す．dhry2reg ベンチマークはほとんどコア遷移していないことが読み取れる．これは，dhry2reg ベンチマークはユーザレベル 100%で動作し続ける (OS が遷移させない) ためであると考えられる．また，データ転送プログラムと stress io ベンチマークは比較的コア遷移の割合が高いことが読み取れる．データ転送プログラムは，実行した約 20 秒間のうち，dhry2reg ベンチマークが動作している 10 秒ほどの間は dhry2reg ベンチマークが動作していないコアを使用し，それ以外ではほとんど毎回使用するコアが遷移していたため，コア遷移の割合は平均 0.50 ほどになった．stress io ベンチマークも同様に，dhry2reg ベンチマークが動作している間は dhry2reg ベンチマークが動作していないコアを主に使用していた．また，全体を通して処理の繰り返し毎に 4 コア中 1 コアが遷移することが多く，コア遷移の割合は平均 0.25 ほどになった．これは，stress io ベンチマークはディスク I/O リクエスト待ちやアイドル時間が多く，コア遷移しやすいためであると考えられる．

表 7　コア遷移の割合

データ転送プログラム	0.50
dhry2reg	0.01
stress io	0.25

5.　まとめと今後の予定

本報告では，データ転送の効率化手法として，転送タイミング制御と転送コア制御を行い，その効果を評価した．その結果，低性能解析サーバ使用時においても，転送タイミング制御を行うことで，高性能解析サーバ使用時と同等の転送時間で転送できることが分かった．また，転送コア制御を行うことで，転送処理による同一 CPU 上で動作する処理の性能劣化を抑えられることが分かった．今後は，コアごとの転送タイミングをずらす転送タイミング制御において，より効率の良いずらし方の検討を行う．また，転送側で CPU 負荷の低いコアを検出して転送を行わせる転送コア制御にて，CPU 負荷の低いコアのより適切な予測・抽出方法についても検討していく．

謝　　辞

本研究の一部はお茶の水女子大学と富士通株式会社との共同研究契約に基づくものであり，JST CREST JPMJCR22M2 の支援を受けたものである．

文　　献

[1] 飯山知香, 平井聡, 山岡茉莉, 福本尚人, and 小口正人. サーバシステムの性能データ収集および転送における効率化手法の考察. In マルチメディア、分散、協調とモバイル *(DICOMO2022)* シンポジウム, 2022.
[2] 飯山知香, 平井聡, 山岡茉莉, 福本尚人, and 小口正人. サーバシステムの性能データ収集および転送効率化に向けた改善案の評価. In 第 *15* 回データ工学と情報マネジメントに関するフォーラム *(DEIM2023)*, 2023.
[3] 飯山知香, 平井聡, 山岡茉莉, 福本尚人, and 小口正人. サーバシステムの性能データ転送効率化に向けた改善案の評価. In マルチメディア、分散、協調とモバイル *(DICOMO2023)* シンポジウム, 2023.
[4] score-p. `https://www.vi-hps.org/projects/score-p`.
[5] vampir. `https://vampir.eu`.
[6] 阿部文武, 中村朋健, and 志田直之. プロファイラにおける汎用的な cpu 性能情報と表示機能. Technical Report 18, 富士通株式会社 次世代テクニカルコンピューティング開発本部, 2 2016.
[7] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. *Google Technical Report dapper-2010-1*, 2010.
[8] zipkin. `https://zipkin.io`.
[9] jeager. `https://www.jaegertracing.io/`.
[10] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010.
[11] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 158–169, 2015.
[12] perf. `https://perf.wiki.kernel.org/index.php/Main_Page`.
[13] Unixbench. `https://code.google.com/archive/p/byte-unixbench/`.
[14] influxdb. `https://www.influxdata.com/products/influxdb/`.
[15] Syeda Noor Zehra Naqvi and Sofia Yfantidou. Time series databases and influxdb, université libre de bruxelles. In *Advanced Databases Winter Semester 2017-2018*, 2017.
[16] stress. `https://linux.die.net/man/1/stress`.
[17] pandas. `https://pandas.pydata.org`.
[18] Dataframeclient. `https://influxdb-python.readthedocs.io/en/latest/api-documentation.html#dataframeclient`.

量子ランダムアクセスメモリ上での関係データベースの構成法の一検討

御手洗洸陽紀† 合田 和生††

† 東京大学大学院情報理工学系研究科電子情報学専攻 〒 116–8656 東京都文京区本郷 7–3–1
†† 東京大学生産技術研究所 〒 153–8505 東京都目黒区駒場 4–6–1
E-mail: †{hmitarai,kgoda}@tkl.iis.u-tokyo.ac.jp

あらまし 量子コンピュータでは，量子的な重ね合わせ状態にあるデータに対して同時並列的に演算を行うことが可能であり，この性質は多くの量子アルゴリズムにより利用されている．しかし，量子的な重ね合わせ状態を準備する過程が効率的に行えなければ，量子アルゴリズムの古典アルゴリズムに対する優位性が失われる可能性がある．そのため，量子計算で用いるデータを量子的な重ね合わせ状態として効率的に準備する技術は非常に重要であり，近年盛んに研究されている．本研究では，ゲート型量子コンピューターにおける量子状態準備において関係データベースの概念を導入し，その基本的な実装や計算量の見積もりについて検討を行う．本研究で提案する量子版のデータベースシステムは，今後量子計算の基盤技術として重要な役割を果たすことが期待される．

キーワード データベース技術，量子コンピューティング，量子情報技術，QRAM，問い合わせ処理

1 はじめに

量子コンピュータでは，量子的な重ね合わせ状態にあるデータに対して同時並列的に演算を行うことが可能である．1992 年に提案された Deutsch–Jozsa のアルゴリズム [1] は，そのような量子コンピュータの性質を利用し，古典計算に対する量子計算の優位性を示した初めてのアルゴリズムである．Deutsch らは量子コンピュータに有利で実用性に乏しい問題設定ではあるものの，古典コンピュータで指数回数オラクル [1] に質問する必要があるのに対し，量子コンピュータでは 1 回の質問で問題を解くことができると示した．その後，1990 年代から 2000 年代にかけて，現在でも活発に議論される Grover のアルゴリズムや Shor のアルゴリズムといった量子アルゴリズムが多く提案された [2–4]．2010 年代以降は量子機械学習などのより応用的なアルゴリズムが提唱され，量子コンピュータの産業応用への期待が高まっている [5–12]．また，ハードウェアに目を向けてみると 1999 年に NEC の中村らが超伝導量子ビット (qubit) の基礎となるクーパー対箱を実証して以降，目覚ましい勢いで研究開発が進められている [13–19]．IBM は 2023 年に 1121 qubit の量子プロセッサーを発表し，さらに 2030 年頃には誤り訂正機能を持ったプロセッサを開発することを目標としている [2]．

ゲート型量子コンピュータでの計算は大きく分ければ 3 つの手順に分けられる．1 つ目は量子状態の準備，2 つ目は量子アルゴリズムを用いた計算，3 つ目は量子状態からの結果の読出しである．これら 3 つの手順のうち，ボトルネックになることが特に懸念されているのは，1 つ目の量子状態の準備と 3 つ目の量子状態からの結果の読出しである．例えば，線形方程式を解く HHL アルゴリズム [4] では，N 次元のベクトル $\boldsymbol{x}, \boldsymbol{b}$ と，$N \times N$ 次元の行列 A に対して，連立 1 次方程式 $A\boldsymbol{x} = \boldsymbol{b}$ の解 $\boldsymbol{x} = A^{-1}\boldsymbol{b}$ を $O(\mathrm{poly}(N))$ の計算量で量子状態に埋め込むアルゴリズムである．しかし，このアルゴリズムではベクトル $\boldsymbol{b}$ が振幅符号化により量子状態埋め込まれていることを前提しているうえ，結果の読み出しについても，愚直に行った場合 $O(N)$ 回アルゴリズムを繰り返さなければならない．量子状態からの結果の読出しにおけるボトルネックは，結果を格納した量子状態の測定時に重ね合わせ状態が壊れることに由来しており，アプリケーションの工夫などで対応される [20, 21]．一方，量子状態の準備に関しては，効率的な量子ランダムアクセスメモリ (QRAM) [22–27] の存在が仮定されることが多い．しかしながら，QRAM の存在を仮定しただけでは実世界の複雑で互いに関係を持ったデータを，量子状態にエンコードするには非効率的である可能性がある．古典コンピューティングにおいてもデータが膨大になると，データベースでのデータの検索やそのデータのメモリ上への配置は大きなボトルネックとなる場合が多い．これらの課題を解決するうえで強力なツールとなるのはデータベース管理システム (DBMS) である [28]．我々は，古典的な関係データベースシステムの手法を量子状態準備に導入することで複雑で互いに関係を持ったデータを効率的に量子的な重ね合わせ状態として準備することができると考えた．

本研究では，ゲート型量子コンピューターにおける量子状態準備において，関係データベースの概念を導入する．DBMS の中でも，最も広く使われている関係データベース管理システム (RDBMS) での問い合わせ処理において重要な演算は，結合演算 (join) である [28, 29]．古典コンピュータにおける結合演算には，nested-loop join，hash join，sort-merge join など方法が用いられているが，量子コンピューター上での結合演算ではこれらの手法をそのまま流用することはできず，量子コンピューターで実現可能な方法を検討する必要がある．本稿では特に，結合演算に重きを置いて，ゲート型量子コンピューターにおける関係データベースの構成法について検討を行う．今回

1：本稿においてオラクルとは，正解を出力する何らかの量子回路を意味する．
2：https://www.ibm.com/quantum/blog/quantum-roadmap-2033, [Online; accessed 7-Jan-2024]

提案する構成法は大きく 2 つに分けられる．1 つ目は，QRAM 上での関係データベースの構成である．我々は，参照先のリレーションのタプルの QRAM 上でのアドレスを foreign key と一緒に保存しておくことにより，高速な結合演算が実行可能であることを示す．2 つ目は，近似的な振幅符号化 (Approximate Amplitude Encoding; AAE) を用いた関係データベースの構成である．AAE は QRAM と比べてゲート数が少なく済み，より near-term な技術であると考えられる．我々は，AAE に適した結合演算手法の候補として，Grover のアルゴリズムを用いた結合演算を提案する．また，QRAM，AAE いずれを用いた量子版関係データベースにおいても実行可能な汎用的な結合演算も提案する．この手法は，リレーションのタプルの数が非常に小さいケースでは，高速に結合演算が可能であり，事前に用意する QRAM などを削減できるといった利点がある．複雑で互いに関係を持ったデータを効率的に量子状態にエンコードすることができる量子版のデータベースシステムは，今後量子計算の基盤技術として重要になる可能性がある．

本稿の構成は以下の通りである．まず，第 2 節ではデータを量子的な重ね合わせ状態としてエンコードする手法である，QRAM と AAE を紹介し，第 3 節ではデータ検索を行う量子アルゴリズムである Grover のアルゴリズムを紹介する．そして，第 4 節において，本稿で提案する量子版のデータベースシステムの構成法について述べ，第 5 節では，その構成法について考察を行う．最後に，第 6 節で本稿のまとめを述べる．

2 データの量子状態へのエンコーディング

データのアドレスを量子状態 $|i\rangle$ で与えて，そのアドレスに対応する n ビットのデータ $D_i = \{0,1\}^n$ を量子状態 $|D_i\rangle$ として取り出す機能を量子ランダムアクセスメモリ (QRAM) と呼ぶ．QRAM は，具体的には以下で表される機構である [22].

$$\sum_i |i\rangle |0^n\rangle \xrightarrow{\text{QRAM}} \sum_i |i\rangle |D_i\rangle \tag{1}$$

古典的なランダムアクセスメモリとの違いは，アドレスは重ね合わせ状態で与えられ，取り出されるデータも重ね合わせ状態になっている点である．QRAM の機能はユニタリ変換として実現されると仮定されることが多く，その具体的な実装方法は現在盛んに研究されている．

しかしながら，厳密な QRAM では多くの量子ゲートが必要となるため，近い将来での実現は難しいと考えられる．そこで，データの量子的な重ね合わせ状態へのエンコーディング手法として，Parameterized Quantum Circuit (PQC) を用いた近似的な振幅符号化 (Approximate Amplitude Encoding; AAE) が提案されている [20]．この手法は，エンコードされた量子状態に誤差が生じることと引き換えに，少ないゲート数で実装可能であるため，QRAM と比べて near-term な技術であると考えられる．振幅符号化は，$N = 2^n$ 次元の実ベクトル $\boldsymbol{d} = (d_1, d_2, \cdots, d_N)^\top$ を以下のように n 量子ビット状態 $|D_{\text{AE}}\rangle$ の振幅として埋め込む符号化方法である．

$$|D_{\text{AE}}\rangle = \sum_{j=1}^{N} d_j |j\rangle \tag{2}$$

AAE では，ユニタリ変換 $U_{\text{AAE}}(\boldsymbol{\theta})$ で表される PQC を用いることで，式 (2) で表される状態を近似する．ただし，$\boldsymbol{\theta}$ は PQC のパラメータのベクトルである．

$$U_{\text{AAE}}(\boldsymbol{\theta}) |0^n\rangle = \sum_{j=1}^{N} a_j |j\rangle \tag{3}$$

ここで，AAE の目標は $a_j = d_j \; \forall j$ を $O(ln)$ のゲート数で達成することである．l は回路の深さであり，$O(\text{poly}(n))$ 程度の大きさであると想定されている．つまり，AAE を用いることで，$O(\text{poly}(n))$ 程度のゲート数で，データを効率的に量子状態にエンコードすることができると考えられている．ただし，事前に $U_{\text{AAE}}(\boldsymbol{\theta})$ のパラメータ $\boldsymbol{\theta}$ をデータ $\boldsymbol{d}$ に合わせて学習させる必要がある．

以上のように，データを量子的な重ね合わせ状態にエンコードする技術として QRAM と AAE を取り上げた．しかしながら，これらの技術だけでは実世界の複雑で互いに関係を持ったデータを，量子状態にエンコードするには非効率的である可能性がある．例えば図 1 に示すリレーション STUDENT，LAB，そして問い合わせ STUDENT $\bowtie_{\text{STUDENT.Lname=LAB.Lname}}$ LAB を考える。これらの内ある 1 つのリレーションや問い合わせ結果を選び，そのタプルすべてを計算基底符号化により量子的な重ね合わせ状態としてエンコードする場合，ナイーブな実装では，すべてのリレーションや問い合わせに対して 1 つずつ QRAM あるいは PQC を用意する必要がある．

3 Grover のアルゴリズム

Grover のアルゴリズムはオラクル $f : \{0,1\}^n \to \{0,1\}$ が与えられたときに，$f(x_0) = 1$ を満たす x_0 を高確率で効率的に得ることができるアルゴリズムである [2]．ここでは簡単のため，$f(x_0) = 1$ を満たす x_0 はただ一つだけ存在するものとする．以下にアルゴリズムの概要を示す．

（1） まず，n 量子ビットの状態 $|0^n\rangle$ を準備する．そして，$|0^n\rangle$ に対しての各量子ビットに対して Hadamard 変換を作用させ，以下の量子状態 $|\psi\rangle$ を得る．

$$|\psi\rangle = H^{\otimes n} |0^n\rangle \tag{4}$$

（2） 続いて，オラクルが以下のようなユニタリ変換 U_f で実装されているものとして，この U_f を $|\psi\rangle$ 作用させる．

$$U_f |x\rangle = \begin{cases} -|x\rangle & \text{for } x = x_0 \\ |x\rangle & \text{otherwise} \end{cases} \tag{5}$$

（3） さらに，$|\psi\rangle$ に対する反転を行う以下のユニタリ変換 U_ψ を $U_f |\psi\rangle$ 作用させる．

$$U_\psi \equiv I - 2 |\psi\rangle \langle\psi| \tag{6}$$

ただし，I は単位行列である．なお，U_ψ はユニタリ変換

STUDENT

Sname	Address	Lname
Akari Hongo	…, Tokyo	Obara Lab
Yayoi Asano	…, Tokyo	Asuke Lab
Hajime Komaba	…, Tokyo	Obara Lab
Sora Kashiwa	…, Chiba	Obara Lab
Rin Tanashi	…, Tokyo	Asuke Lab

LAB

Lname	Professor
Obara Lab	Sakura Obara
Asuke Lab	Gohei Asuke

STUDENT $\bowtie_{\text{STUDENT.Lname=LAB.Lname}}$ LAB

Sname	Address	Lname	Lname	Professor
Akari Hongo	…, Tokyo	Obara Lab	Obara Lab	Sakura Obara
Yayoi Asano	…, Tokyo	Asuke Lab	Asuke Lab	Gohei Asuke
Hajime Komaba	…, Tokyo	Obara Lab	Obara Lab	Sakura Obara
Sora Kashiwa	…, Chiba	Obara Lab	Obara Lab	Sakura Obara
Rin Tanashi	…, Tokyo	Asuke Lab	Asuke Lab	Gohei Asuke

図 1　リレーションの例．リレーション STUDENT の属性 Lab はリレーション LAB を参照する foreign key である．リレーション LAB の属性 Name はリレーション LAB の primary key である．何らかのアプリケーションで問い合わせ STUDENT $\bowtie_{\text{STUDENT.Lname=LAB.Lname}}$ LAB の結果を量子的な重ね合わせ状態にエンコードする必要が生じた場合，ナイーブな実装では，STUDENT $\bowtie_{\text{STUDENT.Lname=LAB.Lname}}$ LAB に対して 1 つの QRAM あるいは PQC を用意する必要がる．量子版の関係データベースシステムでは，リレーション STUDENT と LAB を保存した QRAM を用いて STUDENT $\bowtie_{\text{STUDENT.Lname=LAB.Lname}}$ LAB の結果をを高速に得ることを目指す．

$U_{\mathrm{r}} \equiv |\psi\rangle\langle 0^n|$ を用いると，以下のように表すことができる．

$$U_\psi = U_{\mathrm{r}} \left(I - 2\,|0^n\rangle\langle 0^n|\right) U_{\mathrm{r}}^\dagger \tag{7}$$

本節の場合，$U_{\mathrm{r}} = H^{\otimes n}$ である．

(2), (3) を $\sqrt{N}$ 回繰り返したのちに状態を測定すると，$1-1/N$ 以上の確率で x_0 を得ることができる．

Grover のアルゴリズムではオラクルへの質問回数が $O(\sqrt{N})$ であり，古典的な全探索のアルゴリズムに対して 2 乗加速が実現されている．Grover のアルゴリズムを現実世界の問題に適用するには，まず，オラクルをユニタリ変換 U_f の形で量子回路で実装する必要がある．また，上述のアルゴリズムでは，$U_{\mathrm{r}} = H^{\otimes n}$ となっており，$|\psi\rangle$ は，n ビットで表現可能な $N = 2^n$ 通りの値すべてが等しく重ねあわせられた状態となっている．つまり，現実世界のデータベースに，Grover のアルゴリズムを適用するには，所望のデータのみを重ね合わせ状態としてエンコードするユニタリ変換 U_{r} が必要である．

4　提 案 手 法

量子デバイスにおいても古典コンピュータにおける関係データベースシステムと同様に，STUDENT と LAB の 2 つのリレーションのみを保存しておく．そのうえで，問い合わせ STUDENT $\bowtie_{\text{STUDENT.Lname=LAB.Lname}}$ LAB に対して，効率的にその結果を返すことを目指す．

QRAM は 2 節で述べたように，非常に多くのゲート操作が必要になるため，近い将来での実現は難しいと考えられる．一方，AAE を用いたエンコーディングは QRAM と比べて near-term な技術であると考えられる．ただし，あくまで近似的なエンコーディングしか行うことができず，エンコーディング結果に必ずエラーが生じるため，応用先に制約が生じる．本節では，QRAM 用いたときと，AAE を用いたときの量子版関係データベースシステムの構成法に関して，主に結合演算に焦点を当てて検討を行う．

4.1　QRAM を用いた関係データベースの構成

本項では図 2 に示すリレーションの QRAM 上での配置を例に，QRAM 上での関係データベースの構成法を議論する．図 2 では，4 つの QRAM を使用し，それぞれの QRAM に STUDENT と LAB のインデックスファイルとデータファイルが保持されている．各インデックスファイルにはデータファイルの QRAM 上のアドレスを指し示すポインタを保存する．なおインデックスファイルのレコードは，QRAM_1，QRAM_3 上でアドレスの 0 から隙間なく敷き詰められているものとする．QRAM_2，QRAM_4 にはデータファイルが保存されている．データファイルのレコードは，インデックスファイルのレコードのように QRAM 上でアドレス 0 から順に敷き詰めていく必要はない．ただし QRAM_2 では，STUDENT のレコードに加

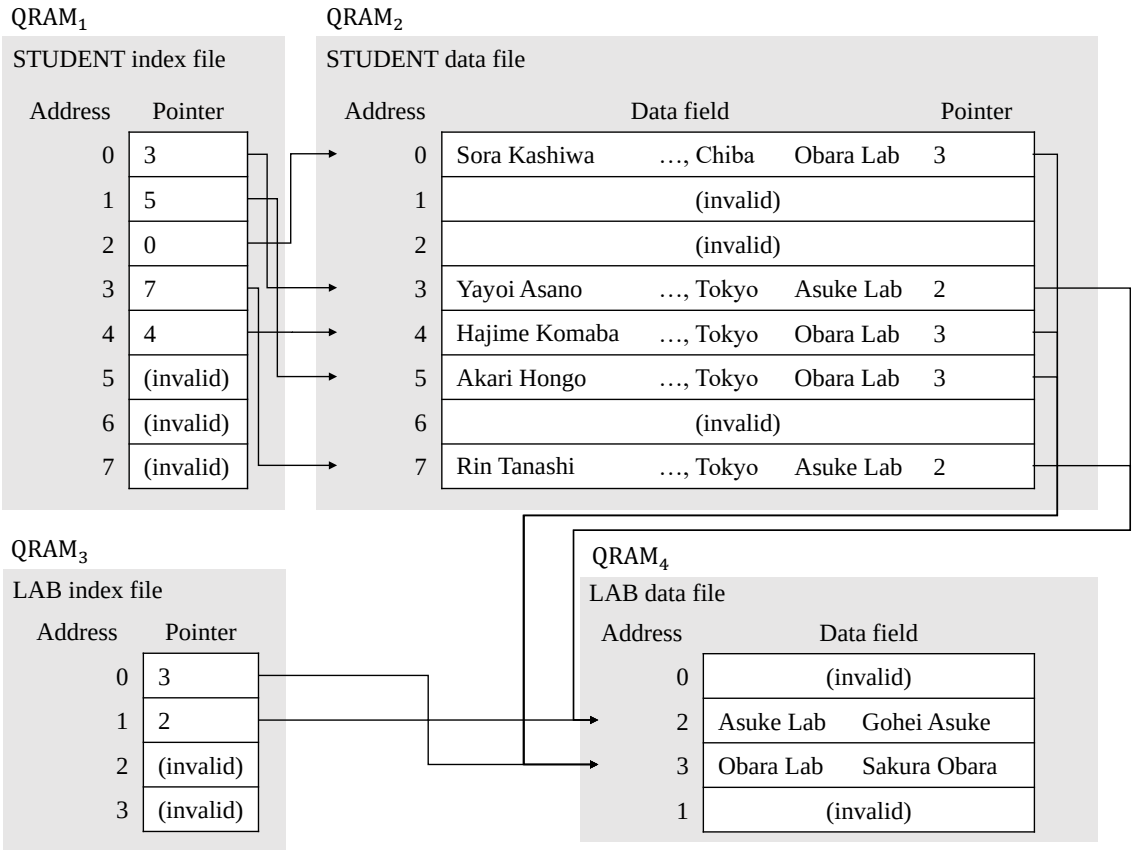


図 2　リレーション STUDENT と LAB の QRAM 上での配置の例．4 つの QRAM それぞれに，STUDENT と LAB のインデックスファイルとデータファイルが保持されている．各インデックスファイルにはデータファイルのレコードの QRAM 上のアドレスを指し示すポインタを保存する．なおインデックスファイルのレコードは，QRAM_1，QRAM_3 上でアドレスの 0 から隙間なく敷き詰める．QRAM_2，QRAM_4 にはデータファイルが保存されており，QRAM_2 では，STUDENT のタプルに加えて LAB のレコードを参照するためのポインタも保存する．データファイルのレコードはインデックスファイルのように QRAM 上でアドレス 0 から順に敷き詰めていく必要はない．なお，本稿では STUDENT と LAB を別々の QRAM 上に保存するものとしたが，同一の QRAM 上に保存することも可能であり，どちらがより効率的かは議論の余地がある．

えて LAB のレコードを参照するためのポインタも保存する．

a） データベースの更新

リレーション STUDENT(LAB) に対してタプルの挿入が生じた場合は，QRAM_2(QRAM_4) の空いているアドレスに新たなレコードを書き込む．またそれと同時に，QRAM_1(QRAM_3) の末尾に新たなレコードを書き込んだアドレスの書き込む．一方，STUDENT(LAB) のタプルの削除が生じた場合は，QRAM_1(QRAM_3) の対応するポインタを削除し，代わりに末尾のアドレス上に保存するポインタをその位置に書き込む．ただし，LAB のタプルを削除しそれを伝搬させる場合は，削除される LAB のタプルのレコードを参照する STUDENT のタプルのレコードを指し示すインデックスファイルのポインタも削除する．磁気ディスク等の古典記憶装置上でデータファイルを保存する場合，レコードの削除に際して効率的なデータの読み出しなどのためにデータファイルを物理的に再配置する必要が生じるケースがある．しかしながら，QRAM はあらゆるアドレスのデータを同時並列的に読み出す機能が備わっていると仮定されるため，我々はデータファイルのレコードを QRAM 上でアドレス 0 から順に敷き詰めるように再配置する必要がないと考えている．このようにレコードの QRAM 上のアドレスが変更されることがないため，QRAM_2 で LAB のレコードのアドレス (ポインタ) を保持していたとしても，それを高頻度で書き換えるといった不利益は生じない．

b） データベースからのデータの取得

まず，STUDENT の全タプルを計算基底符号化による重ね合わせとして量子状態にエンコードする手順を説明する．最初に，インデックスファイルにアクセスするため，QRAM_1 で有効なアドレスの重ね合わせ状態準備する．

$$|0\rangle|0\rangle|0\rangle \to \sum_{i=0}^{4} |i\rangle|0\rangle|0\rangle \tag{8}$$

この操作のナイーブな実装の一例として，以下の実装が考えられる．

（1） QRAM_1 に有効・無効を表す 1 量子ビットのフラグを用意する．

（2） QRAM_1 でアドレス 0 からアドレス α までが有効な

場合は，0 から $2^\beta - 1 \leqq \alpha$ を満たす最大の $\beta \in \mathbb{Z}$ について，0 から $2^\beta - 1 \leqq \alpha$ までの重ね合わせ状態 $\sum_{i=0}^{2^\beta-1} |i\rangle$ をアダマールゲートを用いて準備する．

（3） QRAM_1 に (2) で準備したアドレスを渡し，各アドレスに格納されたポインタと有効・無効フラグを得る．

（4） 有効・無効フラグを測定する．無効が測定された場合は，(2)–(3) を繰り返す．繰り返し回数の期待値は，最悪のケースでも 2 回未満である．

しかしながら，上述の手順はユニタリでない欠点があり，より優れたアルゴリズムは今後の研究課題である．次に，QRAM_1 を用いて，STUDENT データファイルへのポインタをエンコードする．

$$\sum_{i=0}^{4} |i\rangle |0\rangle |0\rangle \xrightarrow{\mathrm{QRAM}_1} \sum_{i=0}^{4} |i\rangle \left|P_i^{\mathrm{STUDENT}}\right\rangle |0\rangle \tag{9}$$

ただし，$P_i^{\mathrm{STUDENT}} \in \{0, 3, 4, 5, 7\}$ は STUDENT のインデックスファイルに含まれるポインタを表す．最後に QRAM_1 を用いて，STUDENT データファイルを取得する．

$$\sum_{i=0}^{4} |i\rangle \left|P_i^{\mathrm{STUDENT}}\right\rangle |0\rangle \xrightarrow{\mathrm{QRAM}_2} \sum_{i=0}^{4} |i\rangle \left|P_i^{\mathrm{STUDENT}}\right\rangle \left|t_i^{\mathrm{STUDENT}}\right\rangle \tag{10}$$

ただし，t_i^{STUDENT} は STUDENT リレーションのうちの 1 つのタプルを表す．なお，t_i^{STUDENT} には図 2 に示すように，LAB のタプルを参照するためのポインタも含まれている．式 (8) から式 (10) までの操作により，STUDENT の読み出しは完了する．LAB のリレーションの取得も，STUDENT でのケースとまったく同様の手順により行うことができる．さらに，STUDENT $\bowtie_{\mathrm{STUDENT.Lname=LAB.Lname}}$ LAB の結果は，STUDENT の取得の手順に 1 つステップを加えるだけで可能となる．式 (10) の状態を準備したうえで，$|t_i^{\mathrm{STUDENT}}\rangle$ の LAB のタプルを参照するためのポインタ部分を QRAM_4 に渡す．それにより，STUDENT と LAB の結合演算が自然と実行され，STUDENT $\bowtie_{\mathrm{STUDENT.Lname=LAB.Lname}}$ LAB の結果が得られる．

$$\sum_{i=0}^{4} |i\rangle \left|P_i^{\mathrm{STUDENT}}\right\rangle \left|t_i^{\mathrm{STUDENT}}\right\rangle |0\rangle \xrightarrow{\mathrm{QRAM}_2} \sum_{i=0}^{4} |i\rangle \left|P_i^{\mathrm{STUDENT}}\right\rangle \left|t_i^{\mathrm{STUDENT}}\right\rangle \left|t_i^{\mathrm{LAB}}\right\rangle \tag{11}$$

以上のように，は結合演算は単に QRAM の読み出し操作だけによって完結する．そのため，式 (8) の操作と QRAM の読み出しにかかる時間が量子ビット数 n に対して，いずれも $O(\mathrm{poly}(n))$ であれば，STUDENT $\bowtie_{\mathrm{STUDENT.Lname=LAB.Lname}}$ LAB の取得にかかる時間のオーダーも $O(\mathrm{poly}(n))$ となる．つまり，QRAM 上においても関係データベースを構成することで，結合されたリレーションも高速に取得することが可能となる．

4.2 AAE を用いた関係データベースの構成

本節では，図 1 に示すリレーションの具体的な例は用いず，より一般に議論を行う．まず，リレーション $\mathrm{R}(A_1, A_2, \cdots, A_{m_\mathrm{R}}), \mathrm{S}(B_1, B_2, \cdots, B_{m_\mathrm{S}})$ に対応した量子状態として $|\mathrm{R}\rangle$，$|\mathrm{S}\rangle$ を以下の式 (12)–(13) のように定義する．

$$|\mathrm{R}\rangle = \sum_{j=1}^{K_\mathrm{R}} \left|t_j^\mathrm{R}\right\rangle = \frac{1}{\sqrt{K_\mathrm{R}}} \sum_{j=1}^{K_\mathrm{R}} \left|v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j\right\rangle \tag{12}$$

$$|\mathrm{S}\rangle = \sum_{j=1}^{K_\mathrm{S}} \left|t_j^\mathrm{S}\right\rangle = \frac{1}{\sqrt{K_\mathrm{S}}} \sum_{j=1}^{K_\mathrm{S}} \left|w_1^j, w_2^j, \cdots, w_{m_\mathrm{S}}^j\right\rangle \tag{13}$$

ここで，A_i, B_i はそれぞれ R,S の属性，$t_j^\mathrm{R} =< v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j >, t_j^\mathrm{S} =< w_1^j, w_2^j, \cdots, w_{m_\mathrm{S}}^j >$ は R,S のタプル，v_i^j, w_i^j はタプル $t_j^\mathrm{R}, t_j^\mathrm{S}$ の A_i, B_i に対応する値である．また，$K_\mathrm{R}, K_\mathrm{S}$ はそれぞれ R,S のタプルの数であり，$m_\mathrm{R}, m_\mathrm{S}$ は R,S の属性の数である．これらの $|\mathrm{R}\rangle$ と $|\mathrm{S}\rangle$ は以下の関係式を満たす AAE のユニタリ変換 $U_{r_\mathrm{R}}, U_{r_\mathrm{S}}$ で近似的に準備されるものとする．

$$U_{r_\mathrm{R}} |0^{n_\mathrm{R}}\rangle \sim |\mathrm{R}\rangle \tag{14}$$

$$U_{r_\mathrm{S}} |0^{n_\mathrm{S}}\rangle \sim |\mathrm{S}\rangle \tag{15}$$

ただし，n_R と n_S は R と S のタプルのビット長である．

a） データベースの更新

リレーション R,S に対してタプルの挿入・更新・削除などが生じた際には，PQC の再学習が必要となる．そのため，保存されたデータがわずかに更新された場合などに，効率的に $\boldsymbol{\theta}$ を学習するための手法を研究することが今後の課題になると考えられる．

b） データベースからのデータの取得

データベースから R と S を取得するには，基底状態に対して $U_{r_\mathrm{R}}, U_{r_\mathrm{S}}$ を作用させるだけでよい．一方，R $\bowtie$ S を取得するにはより複雑な操作が必要となる．まず，R と S の直積 $\mathrm{R} \times \mathrm{S}$ を量子状態で表す．これは単純に $|\mathrm{R}\rangle$ と $|\mathrm{S}\rangle$ のテンソル積 $|\mathrm{R} \times \mathrm{S}\rangle = |\mathrm{R}\rangle \otimes |\mathrm{S}\rangle$ で実現される．そして，結合条件を表すユニタリ変換 U_f を準備し，結合条件を満たした $\mathrm{R} \times \mathrm{S}$ のタプルに対応する量子状態の振幅を Grover のアルゴリズムにより大きくすることで，結合演算に相当する操作が実現される．なお，式 (7) 中の U_r は本節の例では $U_{r_\mathrm{R}} \otimes U_{r_\mathrm{S}}$ となり，$U_\psi = U_{r_\mathrm{R}} \otimes U_{r_\mathrm{S}} \left(I - 2 |0^{n_R+n_S}\rangle \langle 0^{n_R+n_S}|\right) U_{r_\mathrm{R}}^\dagger \otimes U_{r_\mathrm{S}}^\dagger$ となる．また，オラクルへの質問回数は $O(\sqrt{K_\mathrm{R} K_\mathrm{S} / \min(K_\mathrm{R}, K_\mathrm{S})})$ となる．したがって，全体としてのゲート数は，U_ψ と U_f で必要となるゲート数 G_ψ, G_f を用いると，$O\left((G_\psi + G_f)\sqrt{K_\mathrm{R} K_\mathrm{S} / \min(K_\mathrm{R}, K_\mathrm{S})}\right)$ となる．図 3 に，Grover のアルゴリズムを用いて結合演算に相当する操作を行う際の量子回路を示す．ただし，図 3 では $m_\mathrm{R} = m_\mathrm{S} = 2$ とし，A_1, A_2, B_1, B_2 の値は，いずれも 4 ビットであるとした．

4.3 汎用的な結合演算

本項では，QRAM と AAE，いずれを用いた関係データベースでも使用可能な汎用的な結合演算を提案する．前節と同様に，

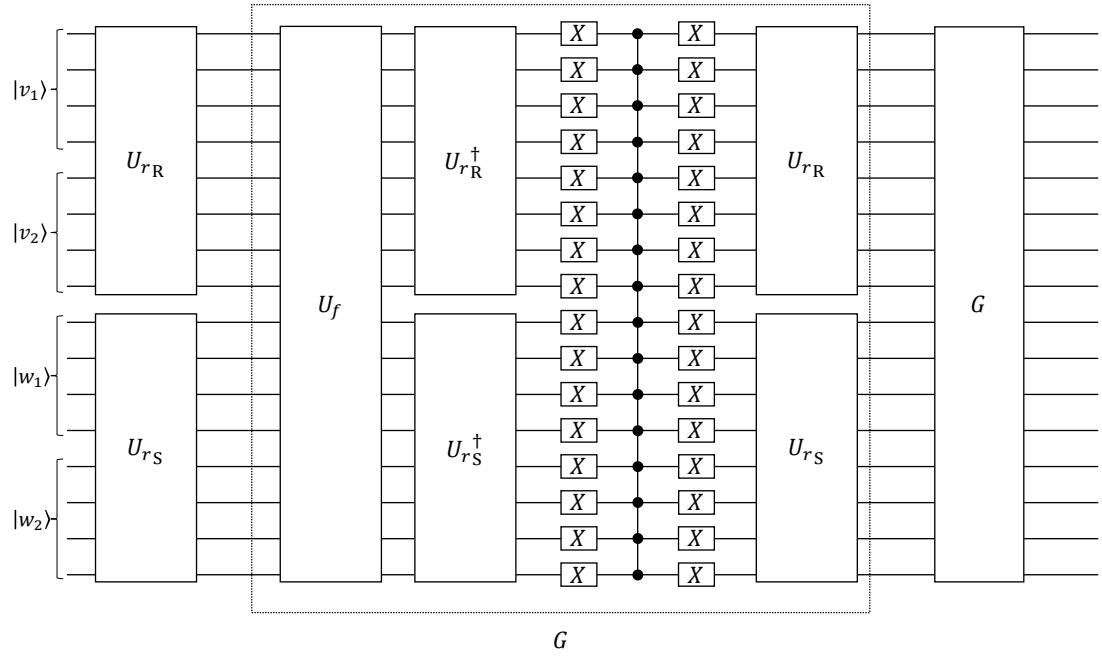


図 3　Grover のアルゴリズムを用いた結合演算を行うための量子回路．リレーション R, S は AAE のユニタリ $U_{r_\mathrm{R}}, U_{r_\mathrm{S}}$ で近似的にエンコードされる．図中の G は $G \equiv \left(U_{r_\mathrm{R}} \otimes U_{r_\mathrm{S}} \left(I - 2\left|0^{n_\mathrm{R}+n_\mathrm{S}}\right\rangle\left\langle 0^{n_\mathrm{R}+n_\mathrm{S}}\right|\right) U_{r_\mathrm{R}}^\dagger \otimes U_{r_\mathrm{S}}^\dagger\right) U_f$ として定義したユニタリ演算子であり，Grover のアルゴリズムの各繰り返しで 1 回ずつ作用させる．

2 つのリレーション $\mathrm{R}(A_1, A_2, \cdots, A_{m_\mathrm{R}}), \mathrm{S}(B_1, B_2, \cdots, B_{m_\mathrm{S}})$ を仮定する．ただし，本節では R の属性 A_{m_R} は，S の属性 B_{m_S} を参照する foreign key であるとし，$\mathrm{R} \bowtie_{A_{m_\mathrm{R}}=B_{m_\mathrm{S}}} \mathrm{S}$ に対応する量子状態を得ることを考える．

ここから，帰納法的に結合演算の手法を示す．なお，本項では状態の規格化定数は無視して表記する．まず，タプル t_j^R を保持する量子ビット $|v_1, v_2, \cdots, v_{m_\mathrm{R}}\rangle$，$t_j^\mathrm{S}$ を保持する量子ビット $|w_1, w_2, \cdots, w_{m_\mathrm{R}}\rangle$，そして，2 量子ビットの補助量子ビット (ancilla) $|a_0, a_1\rangle$ からなる量子システム $|v_1, v_2, \cdots, v_{m_\mathrm{R}}\rangle |w_1, w_2, \cdots, w_{m_\mathrm{R}}\rangle |a_0, a_1\rangle$ を考える．ここで，a_0 は結合演算の途中であることを表すフラグ，a_1 は結合演算のが完了していることを表すフラグとして用いる．このシステムにおいて，以下のように t_0^S から t_{h-1}^S までが t_j^R と結合されているものとする．

$$\sum_{j \notin \pi_1(\mathcal{J})} \left|v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j\right\rangle |0, 0, \cdots, 0\rangle |00\rangle + \sum_{(j,k)\in\mathcal{J}} \left|v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j\right\rangle \left|w_1^k, w_2^k, \cdots, w_{m_\mathrm{R}}^k\right\rangle |01\rangle \quad (16)$$

ただし，

$$\mathcal{J} = \left\{(j,k) \in \mathbb{Z}^2 \mid 0 < k < h \wedge j > 0 \wedge v_{m_\mathrm{R}}^j = w_1^k\right\}$$

であり，$\pi_1(\cdot)$ は (j,k) の第一成分への射影である．また，集合 $\mathcal{J}'$ を以下のように定義する．

$$\mathcal{J}' = \left\{(j,k) \in \mathbb{Z}^2 \mid 0 < k < h+1 \wedge j > 0 \wedge v_{m_\mathrm{R}}^j = w_1^k\right\}$$

式 (16) に示す状態の $|w_1, w_2, \cdots, w_{m_\mathrm{R}}\rangle$ に対して，S の h 番目のタプル t_h^S を書き込んでいく．最初に，$v_{m_\mathrm{R}}^j = w_1^h$ であるときに a_0 を反転させ，以下の状態を得る．

$$\sum_{j\in\pi_1(\mathcal{J}')\setminus\pi_1(\mathcal{J})} \left|v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j\right\rangle |0, 0, \cdots, 0\rangle |10\rangle + \sum_{j\notin\pi_1(\mathcal{J}')} \left|v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j\right\rangle |0, 0, \cdots, 0\rangle |00\rangle + \sum_{(j,k)\in\mathcal{J}} \left|v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j\right\rangle \left|w_1^k, w_2^k, \cdots, w_{m_\mathrm{R}}^k\right\rangle |01\rangle \quad (17)$$

次に，$a_0 = 1$ の場合，$|w_1, w_2, \cdots, w_{m_\mathrm{R}}\rangle$ に対して S の h 番目のタプル t_h^S を書き込み，以下の状態を得る．

$$\sum_{j\in\pi_1(\mathcal{J}')\setminus\pi_1(\mathcal{J})} \left|v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j\right\rangle \left|w_1^h, w_2^h, \cdots, w_{m_\mathrm{R}}^h\right\rangle |10\rangle + \sum_{j\notin\pi_1(\mathcal{J}')} \left|v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j\right\rangle |0, 0, \cdots, 0\rangle |00\rangle + \sum_{(j,k)\in\mathcal{J}} \left|v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j\right\rangle \left|w_1^k, w_2^k, \cdots, w_{m_\mathrm{R}}^k\right\rangle |01\rangle \quad (18)$$

さらに，$a_0 = 1$ の場合 a_1 を反転し，以下の状態を得る..

$$\sum_{j\in\pi_1(\mathcal{J}')\setminus\pi_1(\mathcal{J})} \left|v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j\right\rangle \left|w_1^h, w_2^h, \cdots, w_{m_\mathrm{R}}^h\right\rangle |11\rangle + \sum_{j\notin\pi_1(\mathcal{J}')} \left|v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j\right\rangle |0, 0, \cdots, 0\rangle |00\rangle + \sum_{(j,k)\in\mathcal{J}} \left|v_1^j, v_2^j, \cdots, v_{m_\mathrm{R}}^j\right\rangle \left|w_1^k, w_2^k, \cdots, w_{m_\mathrm{R}}^k\right\rangle |01\rangle \quad (19)$$

最後に，$v_{m_\mathrm{R}}^j = w_1^h$ であるときに a_0 を反転させ，以下の状態を得る．

$$
\begin{aligned}
&\sum_{j\in\pi_1(\mathcal{J}')\setminus\pi_1(\mathcal{J})}\left|v_1^j,v_2^j,\cdots,v_{m_\mathrm{R}}^j\right\rangle\left|w_1^h,w_2^h,\cdots,w_{m_\mathrm{R}}^h\right\rangle|01\rangle\\
&+\sum_{j\notin\pi_1(\mathcal{J}')}\left|v_1^j,v_2^j,\cdots,v_{m_\mathrm{R}}^j\right\rangle|0,0,\cdots,0\rangle\,|00\rangle\\
&+\sum_{(j,k)\in\mathcal{J}}\left|v_1^j,v_2^j,\cdots,v_{m_\mathrm{R}}^j\right\rangle\left|w_1^k,w_2^k,\cdots,w_{m_\mathrm{R}}^k\right\rangle|01\rangle\\
&=\sum_{j\notin\pi_1(\mathcal{J}')}\left|v_1^j,v_2^j,\cdots,v_{m_\mathrm{R}}^j\right\rangle|0,0,\cdots,0\rangle\,|00\rangle\\
&+\sum_{(j,k)\in\mathcal{J}'}\left|v_1^j,v_2^j,\cdots,v_{m_\mathrm{R}}^j\right\rangle\left|w_1^k,w_2^k,\cdots,w_{m_\mathrm{R}}^k\right\rangle|01\rangle
\end{aligned}
\tag{20}
$$

以上のように，t_0^S から t_{h-1}^S までが t_j^R と結合されていれば，t_h^S を t_j^R と結合させることが可能である．また，t_k^S が 1 つも結合されていない状態に対して，t_0^S を結合する場合も，全く同じ手順において結合演算が可能である．したがって，上で述べた手順を繰り返すことで R と S の結合演算が実行可能である．この手法による時間計算量は，多入力 CNOT ゲートを ancilla を用いて分解することで，システム全体の量子ビット数 n，S のタプル数 K_S を用いると，$O\left(K_\mathrm{S}\mathrm{poly}\left(n\right)\right)$ となる．したがって，K_S が非常に小さいケースでは，高速に結合演算が可能である．また，S をエンコードする QRAM などを準備する必要がないといった利点がある．

5 考　　察

QRAM 上での関係データベースにおける結合演算は，4.1 節で述べたようにインデクシングによって QRAM 上で自然と実行でき，実行時間はリレーションのタプルのビット数を n とすると，$O(\mathrm{poly}(n))$ となる（ただし，QRAM の読出し時間が $O(\mathrm{poly}(n))$ であると仮定した）．古典の関係データベースシステムにおける結合では，少なくともリレーションのタプル数に比例する項がコストとして発生するため，本手法は極めて高速な結合演算である．しかしながら，この実装は（少なくとも現時点では）foreign key と primary key が等しいときに結合する EQUIJOIN でのみ適用できると考えられる．より一般の結合，さらには選択を行うには，測定を用いる必要になる可能性がある．このとき，Grover のアルゴリズムを用いることも考えられるが，QRAM のエラーの観点からも Grover のアルゴリズムの使用は近い将来では難しいと考えられる [24]．

一方，得られるデータにエラーが生じることが許容できる場合，4.2 節で述べた AAE を用いた関係データベースの構成も候補に挙がる．AAE では QRAM のようにアドレスに対応したデータを返す仕組みがないため，QRAM のような $O(\mathrm{poly}(n)$ 程度の時間での結合演算は不可能である．ただし，QRAM と比べてケート数が少なく，また誤差も許容されるケースを想定しているため，結合にグローバーのアルゴリズムが適用可能であると考えられる [6]．U_f は multi-controlled X ゲートで定数個の ancilla を用いるなどの工夫により，$O(\mathrm{poly}(n_\mathrm{R}+n_\mathrm{S}))$ のゲート数で実装できる．また，AAE に必要なゲート数は $O(\mathrm{poly}(n_\mathrm{R})+\mathrm{poly}(n_\mathrm{S}))$ で十分であると考えられている [20]．したがって，結合演算で必要なゲート数は $O((\mathrm{poly}(n_\mathcal{R}+n_\mathcal{S})+\mathrm{poly}(n_\mathcal{R})+\mathrm{poly}(n_\mathcal{S}))\sqrt{K_\mathrm{R}K_\mathrm{S}/\min(K_\mathrm{R},K_\mathrm{S})})$ となることが見込まれる．また選択に関しても，U_f に選択の条件を埋め込むことで Grover のアルゴリズムを適用することが可能となる．なお，Grover のアルゴリズムでは通常アルゴリズムの最後に測定を行うが，AAE を用いた実装ではもとから量子状態にエラーが含まれており，測定を行ったとしてもデータベースに含まれないデータの振幅が 0 にはならない．そのため，我々はアプリケーションによって，Grover のアルゴリズムでは通常アルゴリズムの最後に測定を行うかどうか，考慮する余地があると考えている．

選択と結合にかかわるクエリ最適化に関しては，今後の研究課題である．古典の関係データベースでは結合よりも選択を先に行うことで，I/O コストや CPU コストの削減につながるが，今回提案した量子版の関係データベースにおいても同様な最適化が可能であるかは，注意して考えなければならない．特に，Grover のアルゴリズムを用いた結合演算では，ユニタリ U_r を繰り返し作用させなければならず，選択を Grover のアルゴリズムを用いて先に行ったからといって，ゲート数や回路の深さの削減につながるとは限らない．また，選択に測定を用いると，その後 Grover のアルゴリズムを用いた結合演算を行うことができない点にも注意が必要である．結合演算の種類の選択も最適化の余地があり，例えば 4.3 節で提案した結合演算が他の手法よりも有利になるケースもあると考えられる．

そして，今回提案した量子版の関係データベースシステムは，量子デバイスだけで動作するものではなく，背後には古典コンピュータや古典ストレージの存在を前提としている．データベースのリレーションも，QRAM や PQC 上だけに保持されているのではなく，古典のストレージにも保存されていなければならない．そのため，我々は古典システムと量子システムの両方を合わせたシステム全体の最適化の余地も多く残されていると考えている．さらに，本研究ではデータベースとして関係データベースを用いたが，量子コンピューティングで実データを扱う際に最適なモデルが関係モデルであるかは不明である．関係モデルよりも量子計算に適したデータモデルも今後検討する必要があると考えられる．

6 おわりに

本稿では，量子回路上での関係データベースの実装方法について，特に結合に焦点を当てて検討を行った．我々は，主に 2 つの実装方法を提案した．1 つ目は，QRAM 上での関係データベースの構成である．我々は，参照先のリレーションのタプルの QRAM 上でのアドレスを foreign key と一緒に保存しておくことで，高速な結合演算が実行可能であることを示した．2 つ目は，AAE を用いた関係データベースの構成である．AAE は QRAM と比べてゲート数が少なく済み，より near-term な技術であると考えられる．我々は，AAE に適した結合演算手法の候補として，Grover のアルゴリズムを用いた結合演算を提案した．

また，QRAM，AAE いずれを用いた関係データベースにおいても実行可能な汎用的な結合演算も提案した．この手法は，リレーションのタプルの数が非常に小さいケースでは，高速に結合演算が可能であり，S をエンコードする QRAM などを準備する必要がないといった利点がある．

我々が知る限り，量子コンピューティングの分野に関係データベースの考え方を導入したのは本研究が初めてだと考えられる．そのため，この分野における研究の余地は大きく残されている．今後はクエリ最適化や，古典システムと量子システムを合わせたデータベースシステム全体の最適化に関する研究が必要となる．

文　献

[1] D. Deutsch and R. Jozsa, “Rapid solution of problems by quantum computation,” Proc. R. Soc. London, Ser. A, vol.439, no.1907, pp.553–558, 1992.
[2] L.K. Grover, “A fast quantum mechanical algorithm for database search,” Proc. STOC, p.212–219, 1996.
[3] P.W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” SIAM Review, vol.41, no.2, pp.303–332, 1999.
[4] A.W. Harrow, A. Hassidim, and S. Lloyd, “Quantum algorithm for linear systems of equations,” Phys. Rev. Lett., vol.103, p.150502, 2009.
[5] K. Mitarai, M. Negoro, M. Kitagawa, and K. Fujii, “Quantum circuit learning,” Phys. Rev. A, vol.98, p.032309, 2018.
[6] H. Tezuka, K. Nakaji, T. Satoh, and N. Yamamoto, “Grover search revisited: Application to image pattern matching,” Phys. Rev. A, vol.105, p.032440, 2022.
[7] Y. Sano, M. Norimoto, and N. Ishikawa, “Qubit reduction and quantum speedup for wireless channel assignment problem,” IEEE Trans. Quantum Eng., vol.4, pp.1–12, 2023.
[8] P. Niroula and Y. Nam, “A quantum algorithm for string matching,” npj Quantum Inf., vol.7, no.1, p.37, 2021.
[9] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P.J. Love, A. Aspuru-Guzik, Jeremy LO’brien, “A variational eigenvalue solver on a photonic quantum processor,” Nat. Commun., vol.5, no.1, p.4213, 2014.
[10] Y. Sato, R. Kondo, S. Koide, and S. Kajita, “Quantum topology optimization of ground structures using noisy intermediate-scale quantum devices,”arXiv:2207.09181, 2022.
[11] I. Kerenidis and A. Prakash, “Quantum recommendation systems,”arXiv:1603.08675, 2016.
[12] S. Gao, F. Hayes, S. Croke, C. Messenger, and J. Veitch, “Quantum algorithm for gravitational-wave matched filtering,” Phys. Rev. Res., vol.4, p.023006, 2022.
[13] Y. Nakamura, Y.A. Pashkin, and J. Tsai, “Coherent control of macroscopic quantum states in a single-Cooper-pair box,” Nature, vol.398, no.6730, pp.786–788, 1999.
[14] A. Blais, R.-S. Huang, A. Wallraff, S.M. Girvin, and R.J. Schoelkopf, “Cavity quantum electrodynamics for superconducting electrical circuits: An architecture for quantum computation,” Phys. Rev. A, vol.69, p.062320, 2004.
[15] J. Clarke and F.K. Wilhelm, “Superconducting quantum bits,” Nature, vol.453, no.7198, pp.1031–1042, 2008.
[16] L. DiCarlo, J.M. Chow, J.M. Gambetta, L.S. Bishop, B.R. Johnson, D.I. Schuster, J. Majer, A. Blais, L. Frunzio, S.M. Girvin, and R.J. Schoelkopf, “Demonstration of two-qubit algorithms with a superconducting quantum processor,” Nature, vol.460, no.7252, pp.240–244, 2009.
[17] J. Koch, T.M. Yu, J. Gambetta, A.A. Houck, D.I. Schuster, J. Majer, A. Blais, M.H. Devoret, S.M. Girvin, and R.J. Schoelkopf, “Charge-insensitive qubit design derived from the Cooper pair box,” Phys. Rev. A, vol.76, p.042319, 2007.
[18] M. Steffen, D.P. DiVincenzo, J.M. Chow, T.N. Theis, and M.B. Ketchen, “Quantum computing: An IBM perspective,” IBM J. Res. Dev., vol.55, no.5, pp.13:1–13:11, 2011.
[19] F. Arute, K. Arya, R. Babbush, D. Bacon, J.C. Bardin, R. Barends, R. Biswas, S. Boixo, F.G. Brandao, D.A. Buell, et al., “Quantum supremacy using a programmable superconducting processor,” Nature, vol.574, no.7779, pp.505–510, 2019.
[20] K. Nakaji, S. Uno, Y. Suzuki, R. Raymond, T. Onodera, T. Tanaka, H. Tezuka, N. Mitsuda, and N. Yamamoto, “Approximate amplitude encoding in shallow parameterized quantum circuits and its application to financial market indicators,” Phys. Rev. Res., vol.4, p.023136, 2022.
[21] P. Rebentrost and S. Lloyd, “Quantum computational finance: quantum algorithm for portfolio optimization,”arXiv:1811.03975, 2018.
[22] V. Giovannetti, S. Lloyd, and L. Maccone, “Quantum random access memory,” Phys. Rev. Lett., vol.100, p.160501, 2008.
[23] V. Giovannetti, S. Lloyd, and L. Maccone, “Architectures for a quantum random access memory,” Phys. Rev. A, vol.78, p.052310, 2008.
[24] S. Jaques and A.G. Rattew, “QRAM: A survey and critique,”arXiv:2305.10310, 2023.
[25] C.T. Hann, G. Lee, S.M. Girvin, and L. Jiang, “Resilience of quantum random access memory to generic noise,” PRX Quantum, vol.2, p.020311, 2021.
[26] R. Asaka, K. Sakai, and R. Yahagi, “Two-level quantum walkers on directed graphs. ii. application to quantum random access memory,” Phys. Rev. A, vol.107, p.022416, 2023.
[27] C.T. Hann, C.-L. Zou, Y. Zhang, Y. Chu, R.J. Schoelkopf, S.M. Girvin, and L. Jiang, “Hardware-efficient quantum random access memory with hybrid quantum acoustic systems,” Phys. Rev. Lett., vol.123, p.250501, 2019.
[28] R. Elmasri and S.B. Navathe, Fundamentals of database systems, Addison-Wesley, 2016.
[29] P. Mishra and M.H. Eich, “Join processing in relational databases,” ACM Comput. Surv., vol.24, no.1, p.63–113, 1992.

DEIM Forum 2024 T2-A-6-03

永続メモリ向けロックフリー索引Bz木に関する研究

中山　宗†　杉浦　健人†　石川　佳治†　陸　可鏡†

† 東海国立大学機構名古屋大学大学院情報学研究科　〒 464–8601 愛知県名古屋市千種区不老町
Email: {nakayama, lu}@db.is.i.nagoya-u.ac.jp, {sugiura, ishikawa}@i.nagoya-u.ac.jp

あらまし　近年，主記憶に不揮発性メモリを用いた永続メモリ環境に注目が集まっている．永続メモリ環境では，障害時においても主記憶のデータが消失せず，残ったデータを用いた復帰が可能となる．永続メモリの扱いは難しく，利用を容易にするためのミドルウェアとして，永続メモリ向け索引構造の研究開発が行われている．永続メモリ向け索引の 1 つである Bz 木は，任意のデータ型を扱えるという大きな長所を持つが，既存研究にてその性能の低さが指摘されている．しかし，Bz 木の使用していたライブラリの性能が低いこと，アルゴリズムに誤りが存在していたことから，既存研究においては Bz 木の性能が正しく測定されていない可能性がある．そこで本研究では，修正と改善を加えた Bz 木の性能を改めて評価することで，既存研究において Bz 木の性能が過小評価されていたことを示す．

キーワード　データ構造・索引，並列・分散処理，永続メモリ．

1 はじめに

近年，主記憶に不揮発性メモリを用いた永続メモリ環境に注目が集まっている．大規模な容量を備えつつ，従来の揮発性メモリに迫る性能を実現した，Intel Optane DC Persistent Memory Modules（DCPMM）が 2019 年ごろに登場した．これにより，DCPMM を用いた永続メモリ環境が実現可能となった．

永続メモリの長所の 1 つは，主記憶に残ったデータを用いた復帰が可能なことであるが，このような復帰を行うためには常に復帰可能な状態を維持する必要がある．永続メモリ環境では障害発生時においても，不揮発性によりデータが消失せず，消失せずに残ったデータを用いた復帰が可能である．しかし，残ったデータを用いた復帰を行うためには，主記憶に残るデータを常に復帰可能な状態に維持する様々な工夫が必要となる．例えば障害によってロックを取ったスレッドの情報が失われると，どのスレッドがロックを解除すべきか不明となり，ロックをかけられたアイテムが残り続ける．このように，復帰可能な状態を維持するために注意すべき点が多く存在するため，永続メモリ上でのプログラムは非常に困難なものとなる．そこで，永続メモリの利用を容易にするためのミドルウェアとして，永続メモリ向け索引構造の研究開発が行われている．

永続メモリ向け索引の 1 つである Bz 木 [1] は，任意のデータ型を扱えるという大きな長所を持つが，既存研究では性能が低いと報告されている．FPTree [2] や LB$^+$Tree [3] は，他の索引と比較しより良い性能を示すことが報告されている．しかし，扱うデータに制約を課すことで性能を上げているため，扱えるキーの種類が固定長 8 バイトのみに限られる．また，PACTree [4] や ROART [5] ，DPTree [6] などのトライ木ベースの索引は，バイナリデータに特化している．一方，B$^+$ 木を拡張した Bz 木は任意のデータ型を扱うことができ，文字列型やユーザ定義型を扱える汎用性を持つ．しかし，Lersch らによる実験 [7] において，Bz 木の性能が他の永続メモリ向け索引 [2,8,9] と比較し劣ることが報告されている．

だが，既存研究においては，Bz 木本来の性能が検証されていない可能性がある．本研究グループでは，Bz 木が使用するライブラリの 1 つである Persistent Multi-Word Compare-and-Swap（PMwCAS）命令 [10] を改善した．結果として，最大約 12 倍の性能改善が見られた．また，提案手法における Bz 木のアルゴリズムには誤りが存在し，不整合が生じ得る．ゆえに，Lersch らによる実験では，Bz 木の性能が正しく評価されていないと考えられる．

そこで本研究では，Bz 木の元論文の手法に修正および改善を加え，改めて Bz 木の性能を評価する．そして，Bz 木の性能が既存研究において過小評価されていたことを示す．3 章にて，Bz 木の元論文での手法の詳細を述べる．4 章では，Bz 木の元論文での手法の問題点，改善の余地を述べ，改善方法を提案する．最後に 6 章において，改善した Bz 木の性能を検証する．

2 関連研究

本章では，関連研究として索引構造における同時実行制御の発展と，永続メモリ環境，Bz 木が使用する Persistent Multi-Word Compare-and-Swap（PMwCAS）命令について詳細を述べる．

2.1 索引構造における同時実行制御の発展

データベースにおける索引構造にも，マルチスレッド環境にてより良い性能を出すための工夫が凝らされてきた．B$^{\text{link}}$ 木 [11] は，ロックの専有期間を短くするため，葉ノード間だけでなく中間ノード間にも兄弟リンクを付与した．OLFIT [12] は，楽観的な排他処理を用いることで，読み取りのためのロックの必要性を排除した．Bw 木 [13] は，論理ページと物理ページを分離し，木の変更を差分レコードと呼ばれる追記型のログを用いて表すことでロックフリー化を実現した．本研究で扱う Bz 木もまた差分レコードによる更新を採用しており，PMwCAS 命令を用いることで永続メモリ環境におけるロックフリー化を簡

略化している．

2.2 永続メモリ環境

永続メモリ環境 [14] とは，主記憶に不揮発性メモリを用いた環境である．高性能な不揮発性メモリである，Intel Optane DC Persistent Memory Modules（DCPMM）の登場により，永続メモリ環境は実現可能となった．DCPMM の性能は複数の先行研究 [7, 15–18] によって報告されている．Lersch らによる実験 [7] では，従来の揮発性メモリと比較し，読み込み帯域幅が約 3–8 倍，書き込み帯域幅が約 11–14 倍低いことを観測した．

また，Compute Express Link（CXL） [19] を用いた永続メモリ環境の実現も注目されている．CXL はオープンインターコネクト規格の 1 つであり，様々なデバイスへの高速な CPU 接続のために提案された．CXL はコヒーレントかつ帯域幅の広い接続を実現するため，CXL で接続したメモリにバッテリーで不揮発性を付与する [20] ことで，永続メモリとしても利用できる可能性が示されている．Fridman らは DCPMM を模倣して CXL メモリにアクセスした際の性能を検証し，DCPMM と比較して CXL メモリが高い帯域幅を持つこと示した [21]．一方で，Persistent Memory Development Kit（PMDK）などを利用した永続メモリ特有のプログラミングは CXL メモリであっても必須であり，データ構造や手続きの改善は依然として求められている．

2.3 Persistent Multi-Word Compare-and-Swap

Persistent Multi-Word Compare-and-Swap（PMwCAS）命令 [10] は，Wang らによって提案された，永続メモリ上の複数ワードをアトミックかつロックフリーに更新する命令である．対象となるメモリのデータを書き換える直前に，書き換え前のメモリの持つデータが想定通りであることを確認することで，ロックフリーなメモリの書き換えを実現する．Wang らの PMwCAS 命令のアルゴリズムは Multi-Word Compare-and-Swap（MwCAS）命令の 1 つである CASN アルゴリズム [22] を拡張したものである．

CASN アルゴリズムおよび Wang らによる PMwCAS 命令のアルゴリズムは，確保したメモリ領域のガベージコレクションを必要とする欠点を持つ．本研究グループでは，既にガベージコレクションを必要としない PMwCAS 命令のアルゴリズム [23] を提案し，性能の改善を達成している．本研究では PMwCAS 命令の改善が，PMwCAS 命令を使用する Bz 木の性能に及ぼす影響についても評価する．

3 Bz 木 概 要

Bz 木 [1] は PMwCAS 命令を用いて B^+ 木を拡張したロックフリーの索引構造である．本章では，Bz 木の元論文での構造と操作の詳細を述べる．Bz 木の構造を述べた後，Bz 木がサポートするノード操作の一部，構造変更操作の詳細を述べる．

3.1 構　　造

Bz 木は，Bz 木の根を指すルートポインタと，複数の中間ノードと葉ノードからなる．全体像を図 1 に示す．ルートポインタが根となる中間ノードへのポインタ，中間ノードが子となる中間ノードまたは葉ノードへのポインタを持つ．これにより，全体として木構造を構築する．

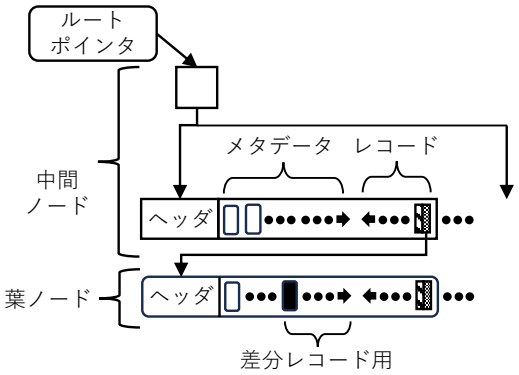


図 1　Bz 木の構造

図 1 に示すように，各ノードはヘッダと，複数のメタデータからなるメタデータ列，複数のレコードからなるレコード列を持つ．ヘッダは，ノードの大きさやレコード数，後述するステータスなど，ノード自身の情報が格納される．ヘッダが持つステータスは，表 1 に示す情報を持つ．メタデータは 1 つのレコードに対して 1 つ存在し，対応するレコードのメモリ上での位置など，対応するレコードの情報を格納する．レコードはキーとペイロードの組からなる．中間ノードのペイロードは葉ノードへのポインタである．

Bz 木は Bw 木 [13] と同様に，木への変更を差分レコードによって表す．差分レコードは対象ノードへの変更（レコードの追加・更新・削除）を表すレコードであり，葉ノードの空き領域に順に格納される．つまり，差分レコードの順序はノードに対して起きた変更の順序を表し，キーの順序とは一致しない．なお，Bw 木とは異なり，Bz 木において中間ノードは差分レコードを持たない．下の階層で構造変更が起きた際は後述する構造変更操作の手続きに従い，追加ないし削除されたものを含む全レコードが常にソートされた状態で格納される．

3.2 操　　作

本節では，Bz 木がサポートする操作の詳細を述べる．まず，Bz 木における操作の PMwCAS 命令による線形化方法を示すため，挿入操作の詳細を述べる．挿入操作は Bz 木内に存在しないキーを持つレコードを，特定のノードに挿入する操作である．その後，本研究の改善により変更を加える更新操作と削除操作，3 つの構造変更操作について説明する．更新操作は指定されたキーを持つレコードのペイロードを更新する操作，削除操作は指定されたキーを持つレコードを削除する操作である．

3.2.1 挿 入 操 作

挿入操作の挙動を図 2 に示す．手順としては，まずレコードの挿入対象となるノードを探索する．そして，挿入するレコードと同じキーを持つレコードがノード内に存在しないことを確認する．その後は 2 つの工程に大別される．

最初の工程では，挿入するレコードのための領域を確保する．

表 1　ステータスが持つ情報

名称	ビット数［bit］	詳細
record count	16	レコードの総数.
block size	22	レコードブロックが占有しているスペースのバイト数.
deleted size	22	レコードの削除により空いた状態となったスペースのバイト数.
frozen	1	ノードが凍結状態であることを示すフラグ.
control region	3	PMwCAS 命令のために使用される領域.

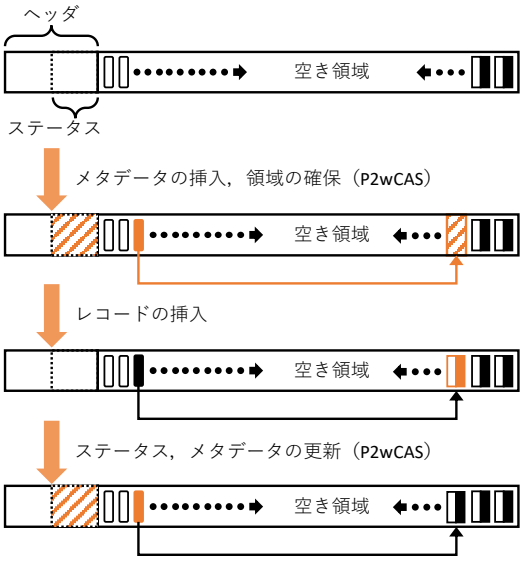


図 2　挿入操作におけるノード内の状態

指定されたキーを持つノードのステータスを読み取り，挿入するレコードに対応するメタデータと，新たなステータスを用意する．新たなステータスとは，変更前のステータスの record count と block size を挿入するレコードに合わせて変更したものである．用意するメタデータには，対応するレコードが書き込み途中である旨の情報が含まれる．そして，用意したメタデータとステータスを PMwCAS 命令を用いてノードに挿入することで，挿入するレコードのための領域を確保する．

そして最後の工程で，実際にレコードを挿入する．既に前の工程で領域を確保しているため，競合する操作を気にすることなくレコードを書き込む．レコードの書き込みが完了したら，ステータスと書き込み状態を示しているメタデータを PMwCAS 命令を用いて更新し，挿入操作を完了する．

3.2.2　更 新 操 作

更新操作は 2 つの場合によって挙動が異なる．1 つは，ペイロードが 8 バイトの固定長であり対象レコードが差分レコードでない場合である．この場合は，キーに手を加えずペイロードのみの変更によりレコードの更新を実現でき，置換更新と呼ぶ．もう 1 つは，ペイロードが可変長であるか，ペイロードが 8 バイトの固定長であるが対象レコードが差分レコードの場合である．この場合の更新操作の挙動は挿入操作と同じであり，追記更新と呼ぶ．

置換更新は，1 度の PMwCAS 命令にて実現する．手順としては，指定されたキーを持つレコードを含むノードを探索した後，ステータスと対象レコードのメタデータとペイロードを PMwCAS 命令を用いて更新する．メタデータを PMwCAS 命令の対象とすることで，対象ノードに対して同時に行われる削除操作を検知する．同様に，ペイロードを PMwCAS 命令の対象とすることで同時に行われる更新操作を，ステータスを対象とすることで同時に走る構造変更操作を検知する．

3.2.3　削 除 操 作

削除操作は，指定されたキーを持つレコードを削除する操作である．実際には，削除対象のレコードをノードから取り除くのでは無く，削除対象レコードを削除済み状態とする操作である．

具体的な手順としては，指定されたキーを持つノードを探索した後，PMwCAS 命令を用いて操作を実現する．PMwCAS 命令の対象は 2 つあり，1 つは削除対象となるレコードに対応するメタデータの，削除された状態への変更である．もう 1 つは，ノードのステータスの deleted size の，削除対象となるレコードとメタデータ分の増加である．PMwCAS 命令を用いることで更新操作と同様に，競合する書き込み操作や構造変更操作が行われていないことを確認しつつ，操作を完了する．

3.3　構造変更操作

Bz 木はノード再編成，ノード分割，ノードマージの 3 つの構造変更操作を持つ．ノード再編成は葉ノードが持つ全てのメタデータをソートする，葉ノード特有の操作である．ノード再編成は，葉ノードが持つ差分レコードのサイズが閾値を超えた場合に行う．ノード分割は保持するメタデータ列とレコード列の合計サイズが閾値を超えたノードを，2 つのノードに分割する操作である．ノード分割を行うことで，ノードの保持するデータがノードサイズを超えることを防ぐ．ノードマージは，2 つのノードを 1 つのノードへと統合する操作である．削除操作により，ノードが保持するデータのサイズが閾値を下回った場合に行う．データサイズが閾値を下回ったノードと，そのノードの左右に存在する兄弟ノードの 1 つを統合することで実現する．ノードマージを行うことで，保持するレコード数が少ないノードを減らし，索引構造全体におけるレコードの探索性能の低下を防ぐ．3.3.1 句にて構造変更操作に必要なメモリ管理について述べた後，3.3.2 句にて構造変更操作のアルゴリズムを紹介する．

3.3.1　構造変更操作におけるメモリ管理

構造変更操作は，索引外のノードを管理するコンポーネント（メモリマネージャと呼ぶ，図 3 参照）を必要とする．構造変更操作によるノードへの変更は，既存ノード内のデータを直接書き換えない．代わりに，変更結果となるノードを索引外に用意

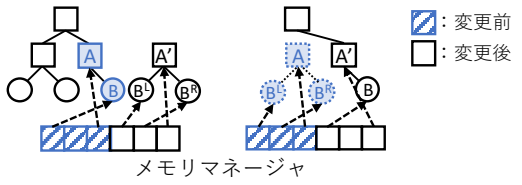


図 3　一時領域を用いた構造変更ノードの記憶

し，既存ノードと入れ替えることで実現する．索引内のノードは根ノードから辿ることが可能だが，索引外のノードは根ノードから辿ることができない．そのため，構造変更操作中の電源障害により，索引外に作られたノードはメモリリークの原因となり得る．ゆえに Bz 木は，永続メモリ上の一時的な使用を目的とする領域を要する．メモリマネージャは，永続メモリ上の一時領域とその領域を使用する索引外ノードを管理することで，メモリリークの発生を防ぐ．

メモリマネージャは，各スレッドに 6 つのノードを格納できる領域を提供する．構造変更操作に用いられるノードは，変更対象となるものと変更結果として作られるものに分けられ，それぞれ 3 種類存在する．1 つ目は構造変更操作の起点となるノード（以降，トリガノード）であり，構造変更前のこのノードを確認することで対象の構造変更操作が特定できる．分割・マージ操作であればトリガノードに加えて，その兄弟ノードおよび親ノードが構造変更の対象となり，一時領域に記憶される．一時領域となる配列ではこれら 6 種類のノードを固定の順番で格納し，リカバリ時は構造変更前のトリガノードを確認することで必要なリカバリ処理を決定する．

3.3.2　構造変更操作のアルゴリズム

3 つの構造変更操作の大まかな流れは共通しており，3 つの工程に大別される．各構造変更操作の流れを図 4 に示す．

1 つ目の工程では，対象ノードを凍結する．書き込み操作の対象となったトリガノードが，各構造変更操作の条件を満たす場合，構造変更操作を開始する．まず，構造変更操作を行うため，トリガノードを凍結（不変，immutable）状態にする．凍結状態とは，書き込み操作や削除操作，他の種類の構造変更操作を受け付けない状態である．トリガノードを凍結することで，トリガノードへのレコードの変更を防ぎつつ構造変更操作を行う．ノードの凍結は，ノードのステータスを対象とした PMwCAS

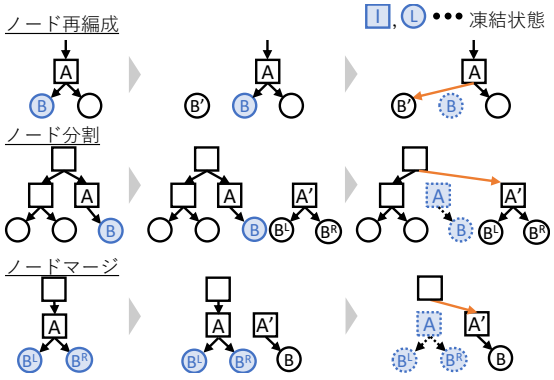


図 4　元論文における構造変更操作

命令によってアトミックに実現する．ステータス内の frozen フラグを立てることで，ノードは凍結状態になる．

2 つ目の工程では，トリガノードに構造変更操作を施した結果となるノードを索引外に用意する．ノード再編成では，トリガノードが持つレコードを全てソートしたノードを索引外に用意する．ノード分割やノードマージのようにノード数が変わる操作では，トリガノードだけでなくその親ノードも用意する．次の工程にて，交換対象となるポインタ数を減らすためである．ノード分割については，トリガノードを等分した 2 つのノードと，その親ノードを作製する．親ノードが持つトリガノードへのポインタは，新たに作製した 2 つのノードへのポインタへと変更する．親ノードに新たなポインタのためのスペースが無ければ，親ノードを対象としてノード分割を再帰的に開始する．ノードマージについても，トリガノードとその兄弟ノードを統合したノードに加え，その親ノードを作製する．ノード分割同様，親ノードが持つトリガノードと兄弟ノードへのポインタを，新たに作製したノードへのポインタへと変更する．親ノードが持つポインタの減少により，親ノードの持つデータサイズが閾値を下回った場合は，親ノードを対象としてノードマージを再帰的に開始する．

最後となる 3 つ目の工程にて，作製したノード（群）を索引に挿入することで，構造変更操作を完了する．作製したノード（群）の挿入は，対象ノード（群）との交換で実現する．ノード分割とノードマージの場合は，交換対象となるトリガノードの親ノードも凍結した後，ノードを交換する．ノードの交換は PMwCAS 命令を用いてアトミックに行う．

3 つ目の工程にて，ノードを交換する際の PMwCAS 命令の対象は 3 つである．1 つは，親ノードが持つトリガノードへのポインタであり，作製したノードへのポインタに変更することでノードを交換する．ノード分割とノードマージの場合は，親ノードへの索引内におけるポインタとなる．トリガノードへのポインタではなく，その親ノードへのポインタを交換対象とすることで，ノード数が変わる操作における PMwCAS 命令の対象の増加を防いでいる．2 つ目は，メモリマネージャが持つ作製したノードへのポインタであり，トリガノードへのポインタと交換する．交換済みの対象ノードへのポインタをメモリマネージャが持つことで，対象ノードをガベージ対象とする前に生じる電源障害にも対処可能である．3 つ目は，交換するポインタを持つノードのステータスである．交換するポインタを持つノードは，ノード編成の場合はトリガノードの親ノード，ノード分割とノードマージの場合は親ノードへのポインタを持つノードである．ステータスから，交換対象となるポインタを持つノードが凍結状態でないことを確認する．

4　修正による一貫性の保証

本研究では，Bz 木に 2 つの修正を加える．1 つは削除操作の修正，もう 1 つは構造変更操作の修正である．順に詳細を述べる．

4.1 削除操作の修正

元論文の手法では，追記更新と削除操作を同時に行った場合に，削除したはずのキー（レコード）がノードに残る不整合が生じ得る．不整合が生じ得る原因は，同一のキーを対象とする追記更新と削除操作が，お互いを検知できないことにある．置換更新と削除操作のお互いを検知する方法を示した後，追記更新と削除操作を同時に行うことで不整合が生じる例を示す．

置換更新と削除操作は，競合するお互いを検知できる．図 5 の左図は，置換更新と削除操作を同時に行った状態である．左下の状態では，削除操作によりメタデータやステータスが準備されてる際に，置換更新によりメタデータが更新されている．この後，削除操作を行うスレッドは，既存のメタデータが想定と異なることから競合する置換更新を検知する．そして，削除操作を中止することで不整合の発生を防ぐ．

置換更新と異なり，追記更新と削除操作は競合するお互いを検知できず，不整合が生じ得る．図 5 の右図は，追記更新と削除操作を同時に行った状態である．右下の状態では，削除操作によりメタデータやステータスが準備されてる際に，追記更新により領域の確保が行われている．対象メタデータに変更がないため，削除操作を行うスレッドは競合する追記更新を検知できず，操作を完了する．結果として，先行する削除操作によって削除されたはずのキー（レコード）が，ノード内に残り不整合が生じる．

元論文の手法にて追記更新と削除操作を同時に行った場合に不整合が生じる原因は，追記更新と削除操作の変更方法が異なる点にある．追記更新が対象キーの存在を確認した後に新規レコードを追加するのに対し，削除操作は対象キーの存在を確認した後，メタデータのみを更新する．ゆえに，追記更新と削除操作とで線形化の方針が異なるため，同時に行われるお互いの操作を検知できない．

追記更新と削除操作を同時に行った場合に生じる不整合を防ぐため，本研究では削除操作も差分レコードによって表す．つまり，削除操作をペイロードの空値への更新と捉え，追記更新と同様の手続きで空値を表す差分レコードを挿入することでレコードを削除する．なお，ペイロードが 8 バイトの固定長であり対象のレコードが差分レコードでない場合は，元論文と同様にメタデータへの削除フラグ埋め込みによりレコードの削除を実現する．

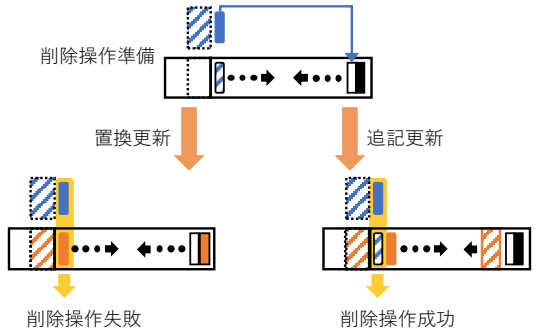


図 5　同時に削除操作と更新操作を行うことにより不整合が生じる例

4.2 構造変更操作の修正

元論文の手法では，複数の構造変更操作を同時に行うと操作結果が消失しうる．操作結果が消失しうるのは，大別して 2 つの場合である．1 つは，同階層にてノード分割（ないしマージ）とノード再編成を同時に行った場合である．もう 1 つは，ある階層でノード分割（ないしマージ）と同時に，1 つ上もしくは下の階層でノード分割やマージを行った場合である．

図 6 はノード分割と再編成を同時に行った結果，再編成を行ったスレッドによる書き込み操作の結果が消失する例である．図 6 の状態 1 ではあるスレッドがノード B を対象に再編成を行い，ノード B と新たに作成したノード B’ の交換を試みている．状態 2 はノード B の兄弟ノード C に対し他スレッドが同時に分割操作を行った状態であり，その後状態 3 のように再編成が先行したとする．このとき，状態 4 に示すように，再編成を完了したスレッドは当初の目的であるレコード書き込みを再編成後のノード B’ に対して行う．しかしノード分割を行うスレッドはノード B’ への書き込み操作を検知する術がないため，分割操作の完了後にはノード B’ に書き込まれたレコードが索引内から消失してしまう．

構造変更操作を同時に行う影響によるレコードの消失を防ぐため，本研究では構造変更操作におけるノードの凍結順序を変更する．図 7 のように，操作対象ノードの親ノードを交換するノード分割とノードマージについて，操作対象ノードの凍結と同時にその親ノードの凍結も行うよう変更する．これにより，操作対象ノードの親ノードが持つ子ノードへのポインタを対象とした PMwCAS 命令を防ぐ．すなわち，ノード分割やノードマージの対象ノードの兄弟ノードに対する構造変更操作は，PMwCAS 命令が成功せずに同時に走りえない．結果として，他の構造変更操作による操作対象ノードの兄弟ノードの差し替えを防ぎ，問題を解消する．

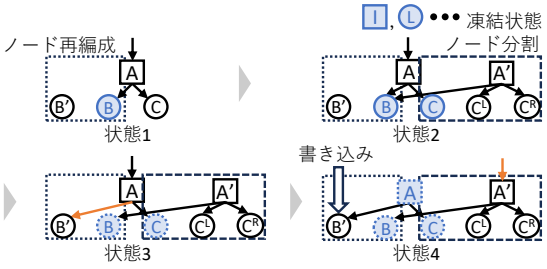


図 6　同時に行う構造変更操作により不整合が生じる例

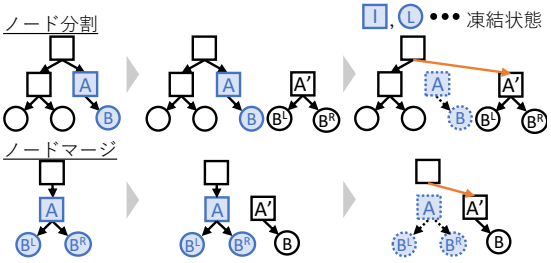


図 7　ノード分割とノードマージの改善手法

5 改　　善

本研究では，構造変更操作における追従処理を排除することで，Bz 木の改善を図る．改善内容を述べた後，追従処理を排除することに必要となるリカバリ処理について述べる．

5.1 構造変更操作における追従処理の排除

元論文の手法では，書き込み操作や削除操作の対象ノードが凍結状態であったスレッドは一旦操作を中断し，凍結状態の原因となった構造変更操作を追従する．操作対象ノードが凍結状態であった場合，まず，操作対象ノードの親ノードが他スレッドによる構造変更操作の対象となっていないか確認する．操作対象ノードの構造変更操作を追従した後に，削除する予定のノードにポインタを挿入することを防ぐためである．同様の理由から，操作対象ノードがその左右のノードに対して行われているノードマージの対象でないかも確認する．親ノードが他スレッドによる構造変更操作の対象になっている，もしくは操作対象ノードが左右のノードにおけるマージ操作の対象になっていた場合は，それらの操作を追従する．操作の対象になっていない場合は，操作対象ノードが持つレコード数やソート済みレコード数から，そのノードに行われている構造変更操作を特定する．その後，操作対象ノードに対して特定した構造変更操作を開始する．ゆえに，元論文に準ずる場合，複数のスレッドが同一の構造変更操作を行うこともあり得る．複数のスレッドが同一の構造変更操作を行った場合，最も速く操作を終了したスレッドの操作のみを反映し，その他のスレッドの操作は反映されない．

本研究グループでは既に揮発性メモリを対象とした Bz 木における性能測定 [24] にて，複数のスレッドが同一の構造変更操作を行う元論文の手法と比較し，構造変更操作の競合を検知したスレッドが対象ノードの再探索に戻る手法がより良い性能を示すことを観測している．不揮発性メモリ向け Bz 木を扱う本研究においても，元論文の手法ではなく，構造変更操作の競合を検知したスレッドが対象ノードの再探索に戻る手法を採用する．

5.2 リカバリ処理の追加

構造変更操作の追従処理を排除すると，リカバリ処理が必要となる．Bz 木の元論文にて一見非効率的な追従処理が採用されていた理由は，電源障害発生時のリカバリ処理を補うためである．構造変更操作中に電源障害が発生すると，凍結状態のノードが索引内に残り得る．電源障害により残された凍結状態のノードに対する構造変更操作は中止される．凍結状態ノードに対して書き込み操作は開始しないため，電源障害により残された凍結状態ノードは，特別な処置を施さない限り凍結状態であり続ける．この問題を解決するために元論文にて採用されていたのが，構造変更操作の追従処理である．書き込み操作の対象ノードが凍結状態である場合に追従処理が開始される．これにより，リカバリ後に対象ノードが凍結状態であり続けるために，操作が開始されない書き込み操作の発生を防ぐ．追従処理を排除した改善案では，代わりのリカバリ処理が必要となる．

リカバリ処理は，構造項変更操作で使用するメモリマネージャを活用する．全てのノードは，索引への挿入・削除時に必ず，メモリマネージャが管理する一時領域を経由する．そのため，一時領域上に残存するノード情報をログとして用いれば，追従処理は不要となる．

改善案におけるリカバリ処理の役割は 2 つに大別される．1 つは凍結状態ノードの除去である．電源障害により構造変更操作が中止された凍結状態ノードに対して，構造変更操作を再開することで，索引の再運用開始前に索引内の全ノードの凍結状態を解除する．もう 1 つは残留ノードによるメモリリークの防止である．構造変更操作にてメモリマネージャに管理される索引外のノードを，復帰時に解放することでメモリリークの発生を予防する．

リカバリ処理は 3 つの工程から成る．1 つ目の工程では，再開すべき構造変更操作を選定する．構造変更操作の最終工程では，トリガノードが新たに作製されたノードと交換される．ゆえに，メモリマネージャの管理する各トリガノードが，根ノードから辿れる場合に操作が終了していないと判断する．2 つ目の工程では，再開する構造変更操作に不必要なノードを解放する．解放するノードは，各トリガノード以外の全ノードである．新たに作製していたノードも，整合性の取れた状態であるか不明なため，解放した後構造変更操作を再開する．最後に，残ったトリガノードを対象に構造変更操作を再開する．構造変更操作の種類は，トリガノード内の状態から決定する．

6 性 能 評 価

改善を施した Bz 木の性能を，不揮発性メモリ向け B^+ 木の 1 つである FAST&FAIR [25] や，不揮発性メモリ向けスキップリストと比較することで評価する．Bz 木については，置換更新を採用した bz-inplace と，追記更新を採用した bz-append を用いる．それぞれの Bz 木は，本研究グループが改善を加えた PMwCAS 命令を使用する．FAST&FAIR については，読み取り操作の性能のみを比較する．

6.1 実 験 設 定

表 2 に本研究で用いるサーバの構成を示す．用いる構成は NUMA に対応しており，2 つの CPU ノードを持つ．本実験では，2 つの CPU の内 1 つのみを用いる．永続メモリを管理するメモリプールが，各ノードに分けて生成され，CPU ソケット間のリモートアクセスが生じることを避けるためである．

表 2　実験用サーバの構成

Item	Value
CPU	Intel(R) Xeon(R) Gold 6258R (two sockets)
DRAM	DIMM DDR4 (Registered) 2933 MHz (16GB × 12)
PMEM	DIMM Synchronous Non-volatile 2666 MHz (126GB × 8)
OS	Ubuntu 20.04.5 LTS
Compiler	GNU C++ ver. 9.4.0

表 3　ワークロード

Workload	Read/Write	Initial Keys	Access Pattern	Key Size
YCSB-A	0.5/0.5	1e7	random	8 byte
YCSB-C	1.0/0.0	1e7	random	8 byte

表 3 に本研究で扱うワークロードを示す．YCSB-A および YCSB-C は，YCSB（Yahoo! Cloud Serving Benchmark）[26] のワークロード A，C をそれぞれ表す．YCSB ワークロードは，一括挿入を用いて索引を構築した後に，読み取り操作と更新操作を行う．挿入済みのキーのみを扱うため，YCSB ワークロードにおける書き込み操作は，全て更新操作となる．ゆえに，YCSB ワークロードでは，構造変更操作の内，ノード再編成のみが発生し得る．各ワークロードでは各スレッドが最大 10^7 回操作し，総実行時間を総実行回数で割ってスループットを計算する．

各ワークロードにおけるキー分布は，式 (1) に示す Zipf の法則に従う．α は skew パラメータであり，N は要素の数を表す．α が 0 のとき，Zipf 分布は一様分布となる．α が 1 のとき，出現頻度が k 番目のキーは 1 番目のキーの出現頻度と比較して $\frac{1}{k}$ となる．すなわち，値 α が 0 から 1 に近づくほどキーに偏りが生じ，より競合の起きやすい設定となる．

$$f(k;\alpha,N) = \frac{1/k^{\alpha}}{\sum_{n=1}^{N} 1/n^{\alpha}} \qquad (1)$$

6.2　結　　果

YCSB-A および YCSB-C ワークロードにおいて，キー長を 8byte で固定しつつ，競合の生じやすい設定でスレッド数を変化させた時のスループットを図 8 および図 9 に示す．また図 10 は，YCSB-A においてスレッド数を 56 に固定した際の，各パーセンタイルに対応するレイテンシを示している．

YCSB-A の結果図 8 において，スキップリストが Bz 木よりも良い性能を示している理由は，スキップリストが 1 つのレコードのみを 1 つのノードに持つためであると考えられる．Bz 木は，1 つのノードに複数のレコードを持ち、かつ更新時に必ずステータスを PMwCAS の対象に含める．このため，Bz 木の競合の発生確率はスキップリストよりも高くなる．スキップリストも置換更新を採用しており，更新時には 1 ワードのみを PMEM へフラッシュする．これに対し、bz-inplace が合計 3 ワードのフラッシュが必要な点も，スキップリストと Bz 木の性能差の違いに影響していると考えられる．

読み取り操作のみからなる YCSB-C の結果図 9 から，bz-inplace の読み取り操作が，bz-append のものよりも高性能であることが伺える．この理由は，更新操作の違いによって生じる各読み取り操作の違いにあると考えられる．bz-inplace の読み取り操作は，置換更新を前提としてソート済みレコードの後に差分レコードを探索する．一方，bz-append の読み取り操作は，追記更新を考慮して差分レコードから探索する．このため，更新操作を含まない YCSB0-C においても，bz-inplace が bz-append より良い性能を示していると考えられる．また，YCSB-C におけるスキップリストの性能が，Bz 木や FAST&FAIR と比較して低

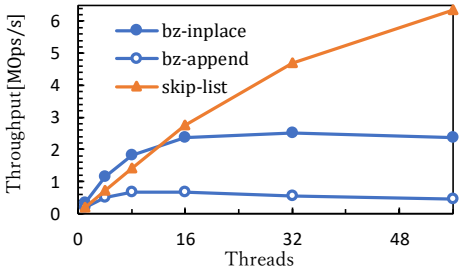


図 8　YCSB-A におけるスレッド数毎のスループット変化
（skew=1.0, key size=8 byte）

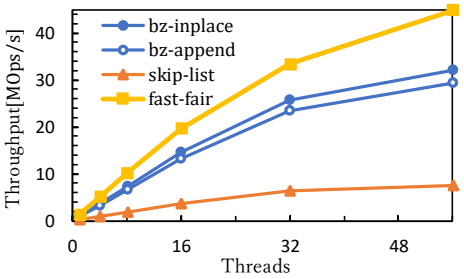


図 9　YCSB-C におけるスレッド数毎のスループット変化
（skew=1.0, key size=8 byte）

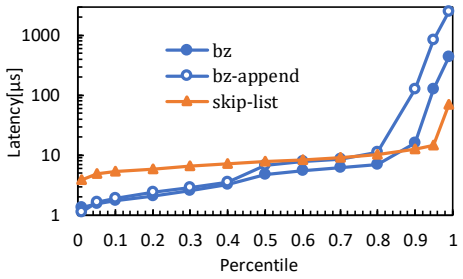


図 10　YCSB-A におけるテールレイテンシ
（skew=1.0, key size=8byte, threads=56）

い理由は，目的とするレコードにたどり着くまでに多くのリンクを経由する必要があるためであると推測される．FAST&FAIR は，扱うキーとペイロードが 8byte のみであることを前提に最適化を行っているため，Bz 木を上回る性能を出している．

2 つのワークロードにおいて，bz-inplace が bz-append と比較しより良い性能を示している理由は，読み取り操作と書き込み操作の違いにあると考えられる．読み取り操作の違いについては，先述した通りである．bz-inplace の扱う置換更新が必要とする PMwCAS 命令が 1 度のみであるのに対し，bz-append の扱う追記更新は 1 度に PMwCAS 命令を 2 回行う．更に，YCSB ワークロードは更新操作のみを行い，挿入操作を行わない．ゆえに，bz-append のみに，ノード再編成とそれによるノードの凍結が生じ得る．これらの要因により，bz-inplace と bz-append に性能差が生じていると考えられる．

更新操作自体の性能のみでなく，更新操作の再試行が bz-append の性能低下を招いていることが，図 10 から伺える．図 10 から，書き込み操作を含むワークロードにおいて，bz-

inplace や bz-append の性能が低下する理由は一部の高レイテンシな操作にあると考えられる．そして，レイテンシが高くなる理由は，競合による書き込み操作の中止や再試行であると推測される．この結果から，Bz 木の性能低下を防ぐためには，書き込み操作の中止や再試行を削減する工夫が必要となると考えられる．

6.3 考　　察

実験の結果から，Bz 木は読み取り操作の性能が高く，競合の高い状況での書き込み操作を苦手とすることが伺えた．ゆえに，Bz 木は読み取り操作主体のワークロードにて，より良い性能を発揮すると考えられる．そして，スキップリストとの比較により，Bz 木は実応用でも十分利用可能である可能性が示された．

Bz 木における書き込み操作の性能向上を図る方法の 1 つとして，PMwCAS 命令の排除が考えられる．Bz 木が使用する PMwCAS 命令は，複雑な永続メモリ上でのプロミングをより簡単に実現するための手法の 1 つであり，Bz 木に最適化された命令ではない．そのため，現状 PMwCAS 命令を用いている Bz 木の書き込み操作や構造変更操作に，PMwCAS 命令の代わりに独自のアルゴリズムを採用することが考えられる．PMwCAS 命令を使用せず，より各操作に最適化されたアルゴリズムへと変更することで，永続メモリに書き込む必要のあるデータ量を削減し得ると考える．

7 おわりに

本稿では，任意のデータ型を扱える永続メモリ向け索引である Bz 木に，修正と改善を施した．結果として，Bz 木本来の性能は既存研究で示されていたほど極端に悪いわけではなく，実応用でも十分利用可能であることが確認された．また，任意のデータ型を扱える Bz 木本来の性能を示すことにより，永続メモリ向け索引の性能基準の 1 つを示しすことができたと考える．今後の展望として，他の永続メモリ索引との性能比較が挙げられる．

謝　　辞

本研究の一部は JSPS 科研費 JP20K19804，JP21H03555，JP22H03594 の助成，日本電信電話株式会社との共同研究，および国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務（JPNP16007）の結果得られたものである．

文　　献

[1] J. Arulraj, J. Levandoski, U. Minhas, and P. Larson, “BzTree: A high-performance latch-free range index for non-volatile memory,” *PVLDB*, vol. 11, no. 5, pp. 553–565, 2018.

[2] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, “Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory,” 2016, pp. 371–386.

[3] J. Liu, S. Chen, and L. Wang, “Lb+trees: Optimizing persistent index performance on 3dxpoint memory,” *PVLDB*, vol. 13, no. 7, pp. 1078–1090, 2020.

[4] W.-H. Kim, M. K. Ramanathan, X. Fu, S. Kashyap, and C. Min, “Pactree: A high performance persistent range index using pac guidelines,” *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[5] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu, “ROART: Range-query optimized persistent ART,” in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 1–16.

[6] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen, “Dptree: differential indexing for persistent memory,” *PVLDB*, vol. 13, no. 4, pp. 421–434, 2019.

[7] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, “Evaluating persistent memory range indexes,” *PVLDB*, vol. 13, no. 4, pp. 574–587, 2019.

[8] S. Chen and Q. Jin, “Persistent b+-trees in non-volatile main memory,” *PVLDB*, vol. 8, no. 7, pp. 786–797, 2015.

[9] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, “Nv-tree: Reducing consistency cost for nvm-based single level systems,” 2015, pp. 167–181.

[10] T. Wang, J. Levandoski, and P.-A. Larson, “Easy lock-free indexing in non-volatile memory,” in *Proc. ICDE*, 2018, pp. 461–472.

[11] P. L. Lehman and S. B. Yao, “Efficient locking for concurrent operations on B-trees,” *ACM TODS*, vol. 6, no. 4, pp. 650–670, 1981.

[12] S. K. Cha, S. Hwang, K. Kim, and K. Kwon, “Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems,” in *PVLDB*, 2001, pp. 181–190.

[13] J. Levandoski, D. Lomet, and S. Sengupta, “The Bw-tree: a B-tree for new hardware platforms,” in *Proc. ICDE*, 2013, pp. 302–313.

[14] S. Scargall, *Programming persistent memory*. New York: Apress, 2020.

[15] Y. He, L. Duo, K. Huang, and T. Wang, “Evaluating persistent memory range indexes; part two,” *PVLDB*, vol. 15, no. 11, pp. 2477–2490, 2022.

[16] A. Van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, “Persistent memory i/o primitives,” in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, 2019, pp. 1–7.

[17] S. Gugnani, A. Kashyap, and X. Lu, “Understanding the idiosyncrasies of real persistent memory,” *PVLDB*, vol. 14, no. 4, pp. 626–639, 2020.

[18] L. Benson, L. Papke, and T. Rabl, “Perma-bench: benchmarking persistent memory access,” *PVLDB*, vol. 15, no. 11, pp. 2463–2476, 2022.

[19] D. D. Sharma, R. Blankenship, and D. S. Berger, “An introduction to the compute express link (cxl) interconnect,” 2023.

[20] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger, “Viyojit: Decoupling battery and dram capacities for battery-backed dram,” in *International Symposium on Computer Architecture (ISCA’17)*, 2017.

[21] Y. Fridman, S. Mutalik Desai, N. Singh, T. Willhalm, and G. Oren, “Cxl memory as persistent memory for disaggregated hpc: A practical approach,” in *Proceedings of the SC ’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W ’23, 2023, p. 983–994.

[22] T. L. Harris, K. Fraser, and I. A. Pratt, “A practical multi-word compare-and-swap operation,” in *International Symposium on Distributed Computing*, 2002, pp. 265–279.

[23] 西村 学，杉浦 健人，石川 佳治，“永続メモリ向け multi-word compare-and-swap 命令の改善”，第 15 回データ工学と情報マネジメントに関するフォーラム（DEIM2023），2B–5，2023．

[24] 中山 宗，杉浦 健人，石川 佳治，“Bz 木におけるマルチスレッドでの構造変更操作に関する性能評価”，第 85 回全国大会講演論文集，vol. 2023，no. 1，2023，pp. 453–454．

[25] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, “Endurable transient inconsistency in Byte-Addressable persistent B+-Tree,” in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 187–200.

[26] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proc. SoCC*. ACM, 2010, pp. 143–154.

ラベルの出現頻度に着目したFPGAを用いた正規パス問合せの提案

溝谷　祐大†　小林　諒平††　藤田　典久††　朴　泰祐††　天笠　俊之††

† 筑波大学大学院理工情報生命学術院　〒 305–8577 茨城県つくば市天王台 1 丁目 1-1
†† 筑波大学 計算科学研究センター　〒 305–8577 茨城県つくば市天王台 1 丁目 1-1
E-mail: †mizotani@kde.cs.tsukuba.ac.jp, ††{kobayashi,amagasa}@cs.tsukuba.ac.jp,
††{fujita,taisuke}@ccs.tsukuba.ac.jp

あらまし　近年，グラフ分析は盛んに行われており，グラフから様々な情報が取得されている．グラフ分析の中でも，ユーザが望むデータを取得するための手法として，正規パス問合せ (RPQ) が存在する．RPQ とはエッジにラベルが貼られたグラフデータを対象とした問合せであり，指定されたラベルの並びを持つパスがグラフ中に存在するかどうかを探索し，存在する場合そのパスの始点・終点ノードを結果としてユーザに返す処理である．ここで課題となるのが，RPQ 評価の計算時間である．近年，データ分析において対象データの大規模化を受けてから，RPQ の対象となるグラフも大規模化が予想されており，現実世界に存在するような多種多様かつ大規模なグラフに対しては，実行に多大な時間を要することが想定される．そのような大規模なデータを処理するために FPGA (Field Programmable Gate Array) などのハードウェアアクセラレータの利用が注目されている．FPGA とは任意の回路をプログラミングによって繰り返し実装可能なハードウェアチップである．FPGA を用いた RPQ の高速化の既存研究では，FPGA の回路規模をすべて有効に利用できない場合が存在することや，複数 FPGA への拡張が困難といった課題点が存在する．そこで本研究では複数カーネルを利用して並列に RPQ 処理を行う手法を提案する．複数カーネルを用いることで，各カーネルが FPGA 内部で独立した回路として実装され並列動作が可能なため，FPGA の回路をより有効に活用できることや，今後複数 FPGA への手法の拡張が容易になることが利点として挙げられる．提案手法では，複数カーネルを用いた手法を実装するためにラベルの出現頻度に着目した．出現頻度が低いラベルをレアラベルを定義し，グラフとクエリをレアラベルを用いて分割することで，複数カーネルを用いた RPQ 処理が可能となる．評価実験では，レアラベルと定義するラベルの個数，クエリ中に出現するレアラベルの個数が多いときに RPQ 評価に要する時間が短くなることを確認した．また，一定の条件のもとで比較手法である，三浦らの手法よりも高速に RPQ 評価を行えることも確認した．

キーワード　FPGA, OpenCL，正規パス問合せ, ハードウェアアクセラレータ, グラフデータベース

1 はじめに

グラフ構造は，様々なデータをノードとエッジ，ラベルで表したデータ構造のことであり，我々の身の回りの多種多様なデータの関係性を表すのに有用である．実世界では，道路ネットワークや，分子構造，デジタル回路等様々なものがグラフとして変換され，保持されている．グラフの分析は盛んに行われており，グラフから様々な情報が取得されている [1].

このようなグラフデータからユーザが望むデータを取得するための手法として，幅優先探索や深さ優先探索，正規パス問合せ (RPQ : Regular Path Query) [2] などが存在する．中でも，RPQ とはエッジにラベルが貼られたグラフデータを対象とした問合せであり，指定されたラベルの並びを持つパスがグラフ中に存在するかどうかを探索し，存在する場合そのパスの始点・終点ノードを結果としてユーザに返すものである．

ここで課題となるのが，RPQ 評価の計算時間である．近年の対象データの大規模化を受け，RPQ の対象となるグラフも大規模化により，現実世界に存在する様な多種多様かつ大規模なグラフに対しては実行に多大な時間を要することが想定される．近年では，そのような大規模なデータを処理するために GPU (Graphics Processing Unit) や FPGA (Field Programmable Gate Array) といったハードウェアアクセラレータの利用が注目されている．FPGA とは任意の回路をプログラミングによって繰り返し実装可能なハードウェアチップである．FPGA は実装された各回路の並列性を利用した，並列度の高い処理が可能である．また，FPGA は GPU と比べて動作周波数が低く，消費電力を抑えることができる．そのため，単位電力あたりのスループットは GPU を上回ることもある．さらに，FPGA は CPU や GPU と比べると低レイテンシとなっている．このような利点から，近年，FPGA は機械学習や金融市場，ビッグデータ処理と様々な分野で利用され始めている [3]．しかし，FPGA は Verilog や VHDL のようなハードウェア記述言語によってプログラムされることが一般的であり，大規模なプログラムの開発には多大なコストを要していた．しかし近年，C/C++や OpenCL 等の高水準言語を利用して FPGA のプログラムを可能にした，高位合成と呼ばれる技術が確立し，開発コストを抑えることが可能となっている．

FPGA を用いた RPQ の高速化の既存研究では，FPGA の回路規模をすべて有効に利用できない場合が存在することや，複

数 FPGA への拡張が困難といった課題点が存在する．そこで本研究では複数カーネルを利用して並列に RPQ 処理を行う手法を提案する．複数カーネルを用いることで，各カーネルが FPGA 内部で独立した回路として実装され並列動作が可能なため，FPGA の回路をより有効に活用できることや，今後複数 FPGA への手法の拡張が容易になることが利点として挙げられる．提案手法では，複数カーネルを用いた手法を実装するためにラベルの出現頻度に着目した．提案手法では，複数カーネルを用いた手法を実装するためにラベルの出現頻度に着目した．出現頻度が低いラベルをレアラベルを定義し，グラフとクエリをレアラベルを用いて分割することで，複数カーネルを用いた RPQ 処理が可能となる．

評価実験では，レアラベルの定義数を増やすことで処理時間が短くなること，クエリ中に出現するレアラベルの個数が増えることで RPQ 評価に要する処理時間が短くなることを確認した．また，特定の条件のもとで比較手法である三浦らの手法よりも高速に RPQ 評価を行えることも確認した．

本稿の構成を以下に示す．第 2 節で前提知識，第 3 節で関連研究を紹介する．第 4 節で提案手法について述べ，第 5 節で実験結果を示す．最後に第 6 節で本研究のまとめを述べる．

2 前提知識

2.1 正規パス問合せ (RPQ)

正規パス問合せ (Regular Path Query) とは，ラベル付きグラフ中のユーザが指定したパスを任意の 2 ノード間に存在するかを調べる問合せである．処理対象となるラベル付きグラフを $G = (V, E, \mathcal{L})$ とすると，以下のような RPQ 評価のセマンティクス R が定められる．

$$R := \varepsilon \mid l \mid l^{-} \mid R \circ R \mid R \cup R \mid R^{i,j} \qquad (1)$$

ここで ε は空列，l はラベルを順方向に，l^{-} はラベルを逆方向に進むことを表し，$\circ$ は結合，$\cup$ は和，$R^{i,j}$ は R の $k\,(i \leqq k \leqq j)$ 回の繰り返しを表している．RPQ の例として，図 1 で示すグラフ G_s を探索対象とした場合，

$$RPQ\ :\ R\ =\ (a \circ b) \cup (a^{-} \circ c^{1,2}) \qquad (2)$$

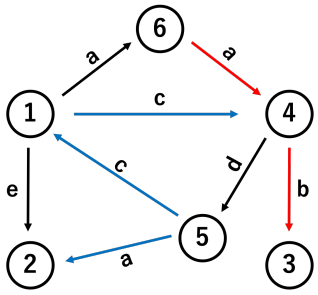
を適応すると

$$R(G_s)\ =\ \{(6,3), (2,1), (2,4)\} \qquad (3)$$

となる．前半のクエリ $(a \circ b)$ でパス $6 \to 4 \to 3$ が得られ，後半のクエリ $(a^{-} \circ c^{1,2})$ でパス $2 \to 5 \to 1, 2 \to 5 \to 1 \to 4$ が得られる．そのため，RPQ の結果として $\{(6,3), (2,1), (2,4)\}$ が得られる．

2.2 FPGA

Field Programmable Gate Array (FPGA) とは任意の回路をプログラミングによって繰り返し実装可能なハードウェアチップである．FPGA は実装された各回路の並列性を利用した，並列度の高い処理が可能である．



グラフG_s

図 1: RPQ の例 (式 (2) を適応)

また，FPGA は Graphics Processing Unit (GPU) と比べて低レイテンシであり，動作周波数が低く，消費電力を抑えることができる．このような利点から，近年，FPGA は機械学習やビッグデータ処理など様々な分野で利用され始めている．

従来，FPGA は Verilog や VHDL のようなハードウェア記述言語によってプログラムされることが一般的であり，大規模なプログラムの開発には多大なコストを要していた．しかし，近年 C/C++や OpenCL 等の高水準言語を利用して FPGA のプログラムを可能にした，高位合成 (HLS) と呼ばれる技術が確立し，開発コストを抑えることが可能となっている．

本研究では，OpenCL を用いて FPGA 上への回路の実装を行った．

3 関連研究

3.1 FPGA を利用した様々な分野での高速化

FPGA を利用した処理の高速化に関する研究は，様々な分野で行われている．グラフ分析の分野で Zhou ら [4] は，PageRank の高速化手法を提案した．また，機械学習の分野では，Elnawawy ら [5] がインターネット・トラフィックのリアルタイムでの分類について，ランダムフォレストを FPGA 上に実装した手法を提案している．さらに，複数 FPGA を用いたグラフ探索アルゴリズムの高速化事例として Dai ら [6] の研究がある．

3.2 RPQ の応用事例

RPQ の応用分野として Resource Description Framework (RDF) XML (Extensible Markup Language) やグラフデータベースなどが挙げられる．SPARQL1.1 [7] は RDF に対して単純な RPQ に対応している．近年，より複雑な RPQ に対応した SPARQL クエリ言語が研究されている [8].

3.3 RPQ 評価の高速化に関する研究

RPQ 評価を高速化する手法としてオートマトンベースの手法とインデックスベースの手法が挙げられる．

インデックスベースの手法として Path Index [9] が挙げられる．Path Index ではグラフ中の任意のパスを設定した長 k まで事前にインデクシングを行う．図 1 のグラフ G_s を例とすると $k = 2$ とした場合，作成されるインデックスは表 1 の様になる．このように事前にインデクシングを行うことで長さが k 以下の

パスの探索の場合，インデックスを参照するだけで即座に結果を取得することが可能となる．また，長さが k より大きいパスにおいてもインデックス同士を結合する事によって効率的に結果を取得することが可能となる．

表 1: Path Index の例 ($k = 2$ としてグラフ G_s (図 1) に適応)

(a) $k = 1$

パス	始点ノード	終点ノード
a	1	6
c	1	4
e	1	2
b	4	3
d	4	5
a	5	2
c	5	1
c	6	4

(b) $k = 2$

パス	始点ノード	終点ノード
$c \circ b$	1	3
$a \circ a$	1	4
$c \circ b$	1	5
$d \circ c$	4	1
$d \circ a$	4	2
$c \circ c$	5	4
$c \circ a$	5	6
$a \circ b$	6	3
$a \circ d$	6	5

また，オートマトンベースの高速化手法として Koschmieder らの手法 [10] も挙げられる．オートマトンベースの手法では与えられたクエリをもとにオートマトンを生成し，処理対象のグラフに適応していく．例として $R = (a \circ b) \cup (b^{1,2} \circ c)$ が与えられた場合，生成されるオートマトンは図 2 の様になる．この生成されたオートマトンをベースにグラフ中のすべてのノードを始点として探索が行われる．

Koschmieder らは，グラフ中に出現するラベルの頻度に着目した高速化を行った．グラフ中での出現頻度が低いラベルをレアラベルとして定義し，与えられたクエリをそのレアラベルで分割する．クエリの分割後，各クエリを満たすパスの探索を双方向探索アルゴリズムを用いて行う．それぞれのクエリを満たすパスとレアラベルを結合していき，結合が行われなかったパスは全体のクエリを満たすことがないためグラフ中から削除される．この処理を行うことによって，探索するクエリ長が短くなりクエリを満たすパスが探索を行うごとに少なくなっていくため，高速にパスの探索が可能となっている．結果として，従来のオートマトンベースの手法と比べ 2 倍から 6 倍の高速化を達成している．

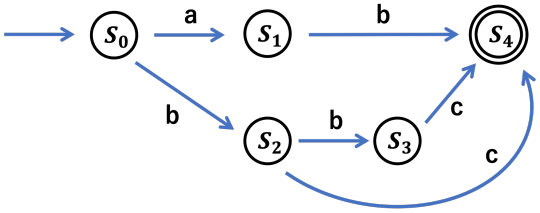


図 2: $R = (a \circ b) \cup (b^{1,2} \circ c)$ から生成されるオートマトン

4 FPGA を用いた RPQ 評価の高速化

FPGA を利用した RPQ 評価の高速化事例として，三浦らのオートマトンベースの手法 [11] がある．この研究では，NFA を FPGA 上に実装を行い RPQ 評価を行っている．Sidler らの研究 [12] において，FPGA 上に任意の NFA を実装するための実装方法が提案されている．三浦らは，その手法を参考に FPGA 上への NFA の実装を行っている．

三浦らが実装した NFA の構造を図 3 に示す．この NFA は Tokens，Trigger，State Transitions の 3 つの設定データを用いることで任意の NFA を実現している．Tokens は RPQ に現れるラベルの種類，Trigger はどのラベルがどの State へ状態遷移を起こすか，State Transitions は State 間の遷移をそれぞれ指定する．この手法の利点として，ユーザから与えられた RPQ に対してこれらの設定データを指定することで，FPGA 上の回路を変更する事なく任意の RPQ 評価を行うことができる点である．これは，一般的に FPGA の回路合成に数時間程度の時間を要することを考えると大きな利点であることがわかる．実際のパスの探索では，この NFA と Frontier を用いた幅優先探索を組み合わせた手法で行っている．

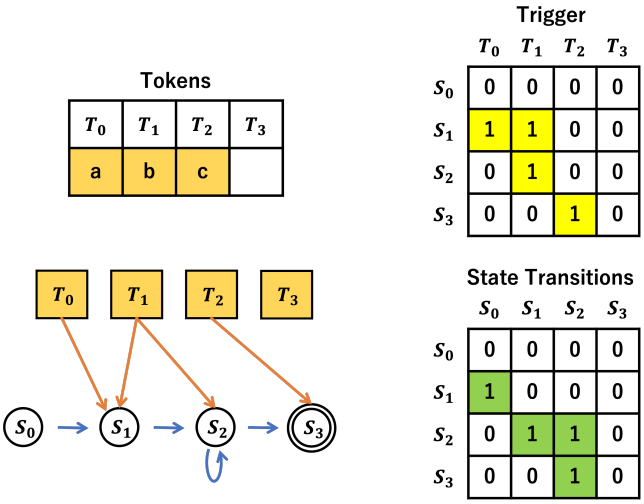


図 3: NFA の構造 ($R = (a \cup b) \circ b^{1,2} \circ c$)

5 提案手法

提案手法では，グラフデータを取得したあとでレアラベルを取得し，そのレアラベルのみを持つグラフデータとそれ以外のグラフデータに分割するステップ，与えられたクエリをレアラベルで分割するステップと FPGA 上で RPQ 評価を行うステップ，最後に得られたすべてのパスを結合し，元のクエリを満たすパスに変換するステップに分けて考える．

5.1 提案手法の処理の概要

図 4 に提案手法の処理の概要を示す．全体の処理は Host 側 (FPGA と通信を行う PC) で行われる処理と FPGA 側で行われる処理に分かれる．まず，処理対象のグラフデータを解析しどのラベルがレアラベルと成りうるかを決定する．その後，5.2 節で後述する方法でグラフデータを分割する．

また，ユーザから RPQ が与えられると，Host 側はそのクエリをレアラベルで分割し，すべてのクエリに対して 4 節で前述した NFA を生成する．その後，生成した NFA とすべてのグラフデータを隣接リスト形式で FPGA に送信する．NFA は FPGA に内蔵されている Block RAM に，グラフデータは FPGA ボー

ド上のグローバルメモリにそれぞれ配置される．FPGA は対象グラフ中のパスの探索と探索されたパスに対しての NFA を用いた RPQ 評価を行い，それぞれのクエリに対してそのクエリを満たすパスが FPGA から Host へ送信される．最後に，Host 側で受け取ったそれぞれのクエリを満たすパスに対して結合処理を行い，最終的にユーザから与えられた RPQ を満たすパスを結果とする．

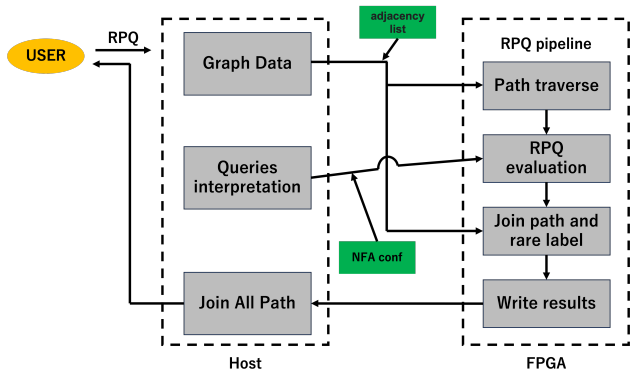


図 4: 全体の処理の概要

5.2 レアラベルの検出，グラフの分割

提案手法では，与えられたグラフデータからすべてのラベルが何回出現するかを確認する．その後，与えられたパラメータ (α) に基づき，レアラベルを決定する．α はユーザが指定するパラメータであり，グラフ中のラベルの中で出現率が低い α 個のラベルをレアラベルとして定義する．

図 5 の場合，$\alpha = 1$ のとき，グラフ中に現れるラベルの中で最も出現頻度の低いラベルがレアラベルとして定義される．図より，その条件を満たすラベルはラベル d であるため，グラフは図のようにラベル d を除いたグラフとラベル d のみのグラフの 2 つのグラフに分割される．

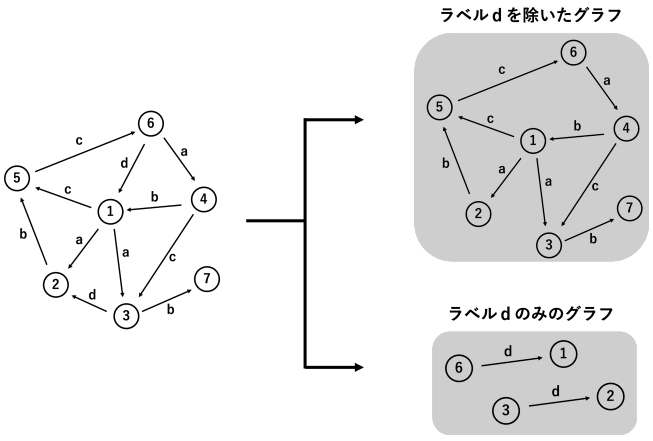


図 5: グラフ分割の例

5.3 クエリ分割

提案手法では，レアラベルの決定後，ユーザから与えられたクエリをレアラベルで分割し，それぞれのクエリに対する NFA を生成する．クエリ分割では，ユーザから与えられたクエリを前から探索していきレアラベルを発見すると，その前後でクエリを分割する実装とした．

ここで例として，ユーザから与えられた RPQ が $R = (a \cup c) \circ d \circ b$ であり，レアラベルが d の場合，レアラベルであるラベル d でクエリを分割するため，R は $R' = a \cup c$, $R'' = b$ の 2 つのクエリに分割される．その後，R', R'' に対応した NFA が生成される．

この実装方法では，クエリ中のレアラベルに和演算 ($\cup$) 等が含まれる場合について RPQ 評価の対象外としてる．つまり，クエリ中にレアラベルが出現したとき，その前後には単純な結合 ($\circ$) のみが現れることを前提とする．

しかし，レアラベルの前後に和演算 ($\cup$) 等が含まれる場合でも，それらのパスを事前に探索し，レアラベルのみのグラフに追加することで対応が可能となる．

5.4 レアラベルを用いた FPGA による正規パス問合せの提案

図 6 に FPGA 内の実装の概要を示す．提案手法では FPGA 上にカーネルを 4 つ定義して RPQ 処理を行う．複数のカーネルを利用することで，各カーネルが FPGA 内部で独立した回路として実装され並列動作が可能となり，FPGA の回路をより有効に活用できる．

5.4.1 FPGA の処理の概要

それぞれのカーネルの概要を以下に示す．

カーネル (1) : すべてのクエリに対するパスの探索，RPQ 評価を行い，RPQ 評価の結果に関わらず見つけたすべてのパスをカーネル 1 に送信

カーネル (2) : カーネル 1 から受け取ったすべてのパスから，RPQ 評価を満たすパスのみをカーネル 3 に送信

カーネル (3) : カーネル 2 から受け取ったパスに対してレアラベルとそのパスを結合し，その結果をカーネル 4 に送信

カーネル (4) : カーネル 3 から受け取ったパスをグローバルメモリにセット

各カーネル間のパスの送受信にはインテル FPGA SDK for OpenCL の拡張機能であるチャネルを利用する．チャネルを利用することで，外部メモリを通さずに FPGA 内部のみでデータの送受信が可能となり，カーネル間でのデータの受け渡しを高効率かつ低レイテンシに行うことが可能となる．

また，カーネル 1 で行うパスの探索，RPQ 評価は 4 節で前述した三浦らの手法をもとに実装を行った．

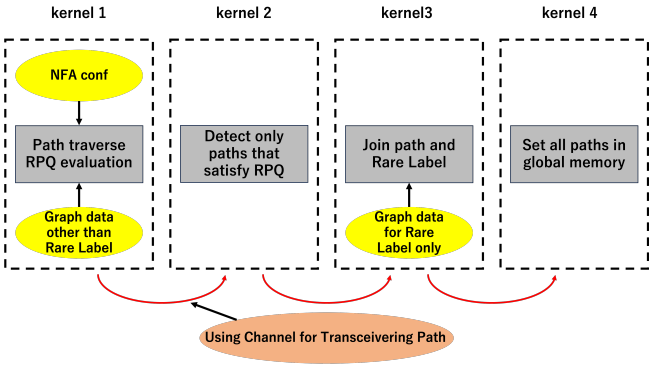


図 6: FPGA 内の実装の概要

5.4.2 RPQ 評価を満たすパスとレアラベルとの結合

提案手法では，ユーザから与えられた RPQ をレアラベルで分割するため，それぞれの RPQ を満たすパスとレアラベルを持つパスとを結合する必要がある．そこで，提案手法では前述した通りカーネル 3 で得られたパスとレアラベルを持つパスとの結合処理を行う．

まず，カーネル 3 ではグラフデータのうち，レアラベルのみで構成されるグラフデータで隣接リストに変換されたものをグローバルメモリに配置する．次に，カーネル 2 から受け取ったパスに対してレアラベルを持つパスとの結合ができるか判定する．例として，R' を満たすパスを (α, β) とするとレアラベルを持つパスとの結合は，R' を満たすパスの終点ノードである β が任意のノードとレアラベルが貼られたエッジを持つときに行う．したがって，パス (β, γ) に貼られているラベルがレアラベルのときパス (α, γ) をカーネル 4 に送信すればよい．

図 5 を例に取ると，パス $(1, 3)$ が R' を満たすパスであるため，カーネル 2 からカーネル 3 に送信される．そのパスをカーネル 3 が受け取ると，パスの終点ノードである 3 について，パス $(3, 2)$ がレアラベルが貼られたエッジであることがわかり，最終的にパス $(1, 2)$ をカーネル 4 に送信する．

また，R'' についてはレアラベルと結合する必要がないことがわかる．これは，与えられた RPQ の最後がレアラベルでないときにレアラベルと結合する必要がないためである．したがって，RPQ の最後がレアラベルでないときはカーネル 2 から受け取ったパスを，そのままカーネル 4 に送信する．

5.4.3 得られたパスの結合

提案手法ではクエリをレアラベルで分割しそれぞれのクエリに対して探索を行うため，それぞれの探索で得られるパスは元のクエリの一部を満たすパスとなる．そのため，最終的にクエリを満たすパスを結果として得るために，それぞれで得られたパスを結合する必要がある．提案手法では，得られたパスの結合は FPGA でなく CPU 上で行う実装とした．

図 5 の場合，R' を満たすパスと R'' を満たすパスのそれぞれを表 2 に示す．これらのパスを結合した結果得られるパスが最終的な RPQ を満たすパスとなる．

パス同士の結合は R' を満たすパスの終点ノードが R'' を満たすパスの始点ノードと一致している場合にそれらのパスが結合可能となる．$R' \circ d$ を満たすパス $(1, 2)$ と R'' を満たすパス $(2, 5)$ はそれぞれ上記の成約を満たしているため，これらのパスを結合することで R を満たすパス $(1, 5)$ が得られる．

表 2: $R' \circ d$，R'' をそれぞれ満たすパス

(a) $R' \circ d$ を満たすパス

始点ノード	終点ノード
1	2
5	1

(b) R'' を満たすパス

始点ノード	終点ノード
2	5
3	7
4	1

6 実　　験

本実験で使用した実験環境を以下の表 3 に示す．また，Host 側のコンパイルには g++ (GCC) 4.8.5 を，コンパイルオプションは-O3 を使用した．

表 3: 実験環境

(a) Host

CPU	Intel Xeon CPU E5-2690 v4 @ 2.60GHz (× 2)
OS	CentOS Linux (version 7)
Memory	DDR4 64GB (32GB / CPU)
Intel FPGA SDK for OpenCL	Version 19.4.0 Build 64 Pro Edition

(b) FPGA

Board	BittWare 520N
FPGA	Intel Stratix 10 GX 2800
Memory	Four banks of DDR4 SDRAM x 72 bits 8GB per bank (Total 32GB)
I/O	PCIe Gen3 x16
BRAM	229Mbits
MLAB	15Mbits

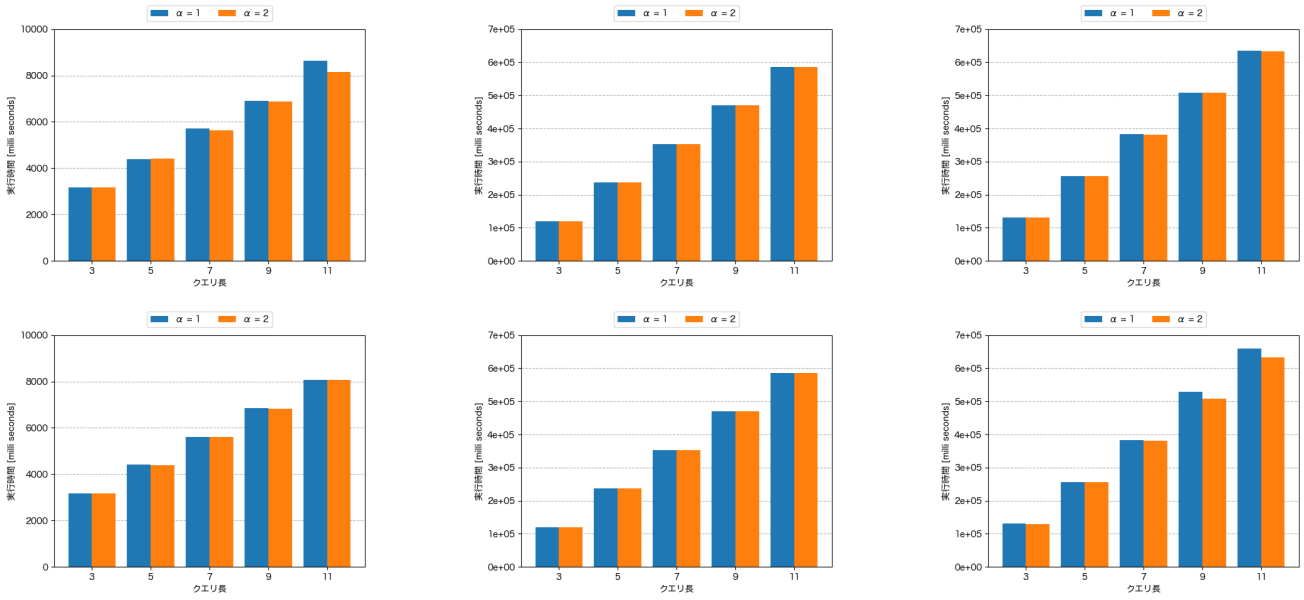
6.1 データセット

本実験では，Advogato [13]，DBLP Citation Network [14]，Reddit Hyperlink Network [15] の 3 種類のデータセットを用いて実験を行う．それぞれのデータセットの詳細は後述する．

また実験のために，それぞれのデータセットについて総ノード数，総エッジ数を変えずにラベルの情報のみを変更することとする．これは，レアラベルの出現率が提案手法にどのような影響を与えるかを調査するためである．ユーザが指定するパラメータ α について， α 個の任意のラベルの出現率が全体の RARE_PARAM%以下になるようにラベルの情報を変更する．実験では，$\alpha = 1, 2$， RARE_PARAM $= 5, 10$ の計 4 種類のラベルの変更を各グラフに行い，実験を行った．

6.1.1 Advogato

Advogato [13] は，フリーソフトウェアの開発者向けのオンラインコミュニティプラットフォームから作成された有向グラフで，ユーザ同士の信頼関係をラベル付きエッジで表したものである．advogato の情報を表 4 に示す．



(a) Advogato (b) DBLP (c) Reddit Hyperlink Network

図 7: パラメータによる影響

表 4: Advogato の詳細

ノード数	エッジ数	ラベルの種類
6541	51127	5

6.1.2 DBLP Citation Network

DBLP Citation Network [16] はコンピュータサイエンスに関する書誌サイト DBLP から収集した約 300 万件の論文データを集めたデータセットである．本実験では，この論文データから論文 ID，著者，引用論文 ID，開催地を抽出し，論文 (P)，著者 (A)，開催地 (V) をノードとする異種グラフデータを作成した．ラベルの種類は 6 種類とし，それぞれ $A \xrightarrow{writing} P$，$P \xrightarrow{writing\ by} A$，$V \xrightarrow{publishing} P$，$P \xrightarrow{published\ in} V$，$P \xrightarrow{citing} P$，$P \xrightarrow{cited\ by} P$ である．今回本実験が DBLP Citation Network から生成したグラフの詳細を表 5 に示す．

表 5: DBLP Citation Network の詳細

ノード数	エッジ数	ラベルの種類
65536	195039	6

6.1.3 Reddit Hyperlink Network

Reddit Hyperlink Network は英語圏のウェブサイトであり多種多様なウェブサイトへのリンクを収集し公開している Reddit [17] から作成されたグラフデータセットである．ここで，グラフ中の各ノードは subreddit と呼ばれる Reddit 内で特定のドメインに焦点を当てたコミュニティを表しており，各エッジは subreddit 間に貼られたハイパーリンクを表す．様々な属性がこのエッジに付与されており，その中の 1 つにエッジの始点コミュニティから終点コミュニティに対して何時にリンクが作成されたかを表す属性が存在する．この属性は 24 時間表記で表されており，本実験ではこの属性を 2 時間刻みで 12 分割し，12 個のエッジラベルに変換した．今回本実験が Reddit Hyperlink Network から生成したグラフの詳細を表 6 に示す．

表 6: Reddit Hyperlink Network の詳細

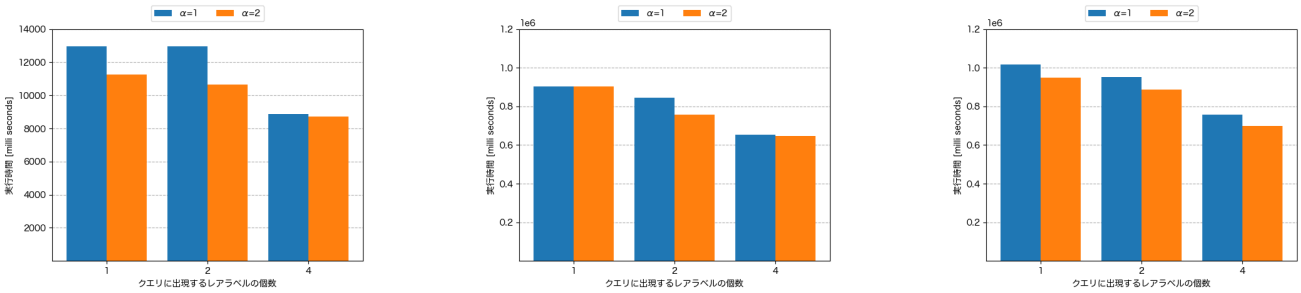
ノード数	エッジ数	ラベルの種類
67179	858488	12

6.2 パラメータによる性能比較

本実験では，パラメータ α，RARE_PARAM を変化させたときの性能比較を行う．図 7 に実験の結果の全体を示す．それぞれ上段の図が RARE_PARAM = 5，下段の図が RARE_PARAM = 10 に対する結果を示している．

α による影響：図より，クエリ長が 3，5，7，9 の場合では，α の値が変化しても，処理時間に大きな差は現れなかった．しかし，図 7 (a) の上段で，クエリ長が 11 の場合を見ると，$\alpha = 2$ のほうが処理時間が顕著に短くなっている．これは，クエリ長が長い場合，RPQ 評価のコストが大きくなるが，レアラベルと定義するラベルの個数が増えることで RPQ 評価の対象となるグラフデータの規模が小さくなるからである．また，レアラベルと定義するラベル数が多くなる場合でも，提案手法ではレアラベルとパスとの結合処理にかかるコストは変わらないことも理由である．

RARE_PARAM による影響：図より，ほとんどの場合で RARE_PARAM の違いによる影響は確認できなかった．これは，RARE_PARAM が小さいほどカーネル 3 で行うパスとレアラベルの結合に要するコストが減少する．しかし，グラフデータを分割する際，レアラベルを除いたグラフが元のグラフデータのサイズと大きく変わらない．一方で，RARE_PARAM が大きいとカーネル 3 の処理コストが増加するが，レアラベルを除いたグラフのサイズが小さくなるため，RPQ 評価に要する時間が短くなる．このような原因から RARE_PARAM の違いによる



(a) Advogato (b) DBLP (c) Reddit Hyperlink Network

図 8: クエリ中に出現するレアラベルの個数による影響

影響が確認できなかったと考える．

6.3 レアラベルの出現回数による実行時間の比較

本実験では前述した 3 つのデータセットを用いて，RPQ で指定するクエリにレアラベルの出現回数を変化させた際の提案手法の性能調査を行った．本実験の対象となるクエリは，和演算等を含まない結合演算のみの単純なクエリである．

この実験では，評価対象となる元のクエリ長を 16 に固定し，その中にレアラベルが 1 つ，2 つ，4 つの場合で実験を行う．また，実験で使用するクエリは，レアラベルの個数のみ指定し，それ以外の部分はランダムに生成したものを使用する．実験結果に示す処理時間はランダムに生成したクエリに対して RPQ 評価を 10 回行ったときの平均値を示す．

実験結果を図 8 に示す．図より，クエリ中に出現するレアラベルの個数が多くなると，RPQ 評価に要する時間が短くなっている．また，Advogato では α の変化による処理時間の変化は見られなかった．しかし，DBLP，Reddit Hyperlink Network では，すべての場合で $\alpha = 2$ のほうが高速になっている．

また，図 9 に，Advogato におけるカーネル 1 からカーネル 3 のそれぞれの処理時間を示した．この図からわかる通り，FPGA での処理時間はクエリに出現するレアラベルの個数が多くなるにつれて短くなっている．これは，より多くのレアラベルが出現するクエリでは，RPQ 評価の対象となるクエリの数が多くなるがその分，一つ一つのクエリ長が短くなり，1 つのクエリに対する処理時間が短くなるからだと考えられる．

また，図からカーネル 2 の処理時間が最も長いことがわかる．これは，カーネル 3 の RPQ 評価を満たすパスとレアラベルとの結合処理を行う．このとき，レアラベルのグラフ情報にアクセスするためにグローバルメモリへのアクセスが生じていることが原因と考えられる．通常 FPGA ではグローバルメモリへのアクセスは低速であるため，カーネル 2 ではカーネル 3 がパスを読み取ってから処理が終了するまで，チャネルへの書き込みを待つ必要があるため処理時間が長くなっている．

6.4 比較手法との性能評価

本実験では，比較手法である Path Index [9]，三浦らの手法 [11] との実行時間の比較を行う．今回使用したデータセットは Advogato と Reddit Hyperlink Network の 2 種類である．また，実行に 2,000 秒以上かかる場合はタイムアウトとした．

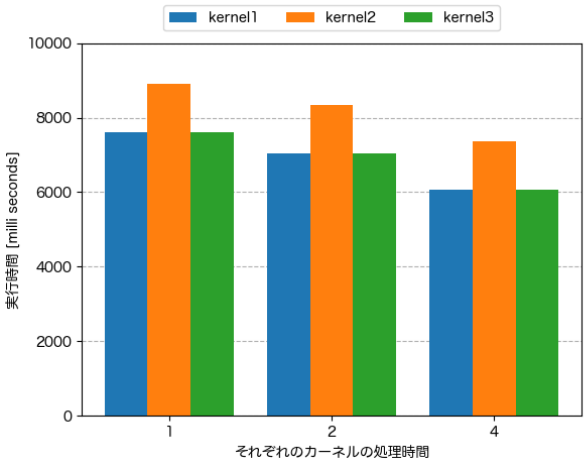


図 9: 各カーネルの処理時間 (Advogato)

図 11 に実験の結果を示す．図からわかる通り，Path Index がパスの長さが大きくなるに連れて処理時間も指数関数的に増加している．三浦らの手法ではパスの長さに比例した形で処理時間が増加している．本手法においてもパスの長さによる実行時間の増加具合は比例的になっている．しかし，提案手法は三浦らの手法と比べ低速になっている．これは三浦らの手法の特徴である，処理時間がクエリ長に比例することが原因である．提案手法では，クエリをレアラベルで分割し，それぞれのクエリに対して RPQ 評価を行っていく．そのため，クエリ長が短い場合，RPQ 評価を行う回数が増えることがボトルネックとなり低速になっている．

また，図 10 にクエリ長を 31 とした場合の三浦らの手法と提案手法の処理時間を示す．本実験では，クエリに出現するレアラベルの個数を 4 個とした．図より，この実験では，Advogato, Reddit Hyperlink Network に対しての RPQ 評価で，提案手法が三浦らの手法よりも高速に行えていることがわかる．

7 ま と め

本研究ではラベルの出現頻度に着目した FPGA を用いた正規パス問合せの手法を提案した．探索対象のグラフデータの中で出現頻度が低いラベルをレアラベルと定義して，そのレアラベルでクエリを分割する．FPGA 上では，カーネルを 4 つに分けることで各処理を並列的に処理できる実装とした．結果として，

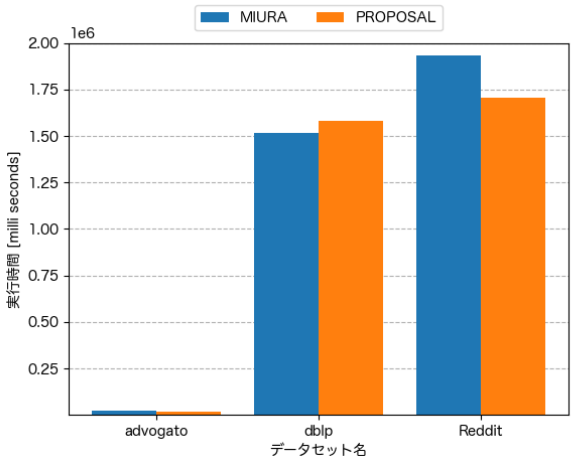
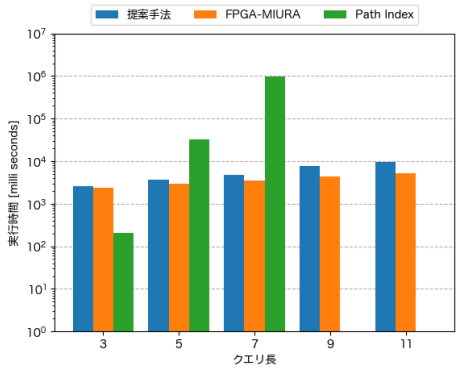
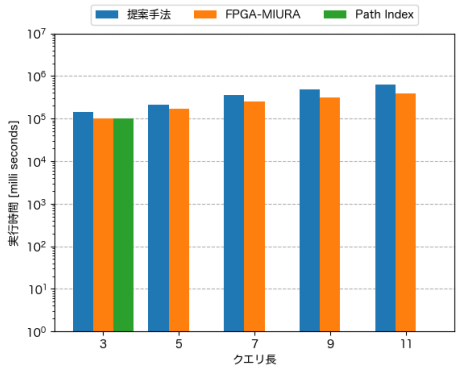


図 10: クエリ長が長い場合の処理時間の比較



(a) Advogato



(b) Reddit Hyperlink Network

図 11: 比較手法との処理時間の比較

レアラベルと定義するラベル数を増やすことでグラフを分割したときに，RPQ 評価の対象となるグラフデータのサイズを小さくすることができる，提案手法ではレアラベルとパスとの結合処理に要するコストはレアラベルの定義数に影響しないという点から元のクエリ長が長い場合に，レアラベルの定義数を増やすことで処理時間が短くなるという特徴が見られた．また，クエリ中に出現するレアラベルの個数が多くなることで，探索対象となるクエリ長が短くなり，一つ一つのクエリに対する RPQ 評価に要する時間が短くなる．これによって，クエリ中に出現するレアラベルの個数が増えることで RPQ 評価に要する処理時間が短くなるという特徴が見られた．評価実験では，クエリ長が長い場合に，三浦らの手法と比べ提案手法が高速に RPQ 評価を行えることもわかった．しかし，多くの点で提案手法は三浦らの手法と比べると低速となっているため，高速化が必要である．今後の課題として，複数 FPGA を用いた手法などが挙げられる．

謝　　辞

この成果は，国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の「ポスト 5G 情報通信システム基盤強化研究開発事業」（JPNP20017）の委託事業，JST CREST（JPMJCR22M2），科学研究費補助金（JP22H03694，JP22K17895）の支援によるものです．

文　　献

[1] Daniel E O'Leary. Artificial intelligence and big data. *IEEE intelligent systems*, Vol. 28, No. 2, pp. 96–99, 2013.
[2] Isabel F Cruz et al. A graphical query language supporting recursion. *ACM SIGMOD Record*, Vol. 16, No. 3, pp. 323–330, 1987.
[3] Shijie Zhou et al. Hitgraph: High-throughput graph processing framework on fpga. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 30, No. 10, pp. 2249–2264, 2019.
[4] Shijie Zhou et al. Optimizing memory performance for fpga implementation of pagerank. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6. IEEE, 2015.
[5] Mohammed Elnawawy et al. Fpga-based network traffic classification using machine learning. *IEEE Access*, Vol. 8, pp. 175637–175650, 2020.
[6] Guohao Dai et al. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 217–226, 2017.
[7] Sparql 1.1 query language, December 4, 2020.
[8] Faisal Alkhateeb et al. Extending sparql with regular expression patterns (for querying rdf). *Journal of web semantics*, Vol. 7, No. 2, pp. 57–73, 2009.
[9] George HL Fletcher et al. Efficient regular path query evaluation using path indexes. 2016.
[10] André Koschmieder et al. Regular path queries on large graphs. In *Scientific and Statistical Database Management: 24th International Conference, SSDBM 2012, Chania, Crete, Greece, June 25-27, 2012. Proceedings 24*, pp. 177–194. Springer, 2012.
[11] Kento Miura et al. An fpga-based accelerator for regular path queries over edge-labeled graphs. In *2022 IEEE International Conference on Big Data (Big Data)*, pp. 415–422. IEEE, 2022.
[12] Sidler David et al. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 403–415, 2017.
[13] Paolo Massa et al. Bowling alone and trust decline in social network sites. In *2009 Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing*, pp. 658–663. IEEE, 2009.
[14] Jie Tang et al. Arnetminer: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 990–998, 2008.
[15] Srijan Kumar et al. Community interaction and conflict on the web. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pp. 933–943. International World Wide Web Conferences Steering Committee, 2018.
[16] dblp: computer scinece bibliography. `https://dblp.org/`.
[17] reddit: the front page of the internet, December, 2020.

量子回路シミュレータの実行環境変化に伴う性能評価と特性比較

青木　望美† 山崎　雅文†† 平井　聡†† 山岡　茉莉†† 福本　尚人††
小口　正人†

† お茶の水女子大学　〒 112–8610 東京都文京区大塚 2 丁目 1 － 1
†† 富士通株式会社　〒 211–8588 神奈川県川崎市中原区上小田中 4 丁目 1 － 1
E-mail: †nozomi-a@ogl.is.ocha.ac.jp, ††{m.yamazaki,ahirai,yamaoka.mari,fukumoto.naoto}@fujitsu.com,
†††oguchi@is.ocha.ac.jp

あらまし　量子コンピュータは量子の性質を利用することで一部の問題に対して非常に高速な計算を実現可能として，現在も研究開発が盛んに行われている．しかし正確に動作する量子コンピュータの実現が期待されている中，量子コンピュータ上で動作するアプリケーションの開発などを同時進行で行うことが理想的である．量子回路シミュレータは従来型コンピュータ上で量子コンピュータの挙動を表現しているツールであり，量子コンピュータ実機上で動作する量子アプリの開発等にも貢献性が高い．本論文では数ある量子回路シミュレータの中でも Qiskit Aer シミュレータと Qulacs シミュレータについて，複数種の単一サーバ上における性能分析を行うとともに両者の計算機環境における量子回路シミュレータの性能比較を行う．

キーワード　性能分析，アプリ分析，プロファイラ，量子回路シミュレータ，HPC

1　はじめに

近年注目が集まっている量子コンピュータは一部の問題に対して従来型コンピュータと比べて非常に高速な計算を実現することができる可能性があるとされており，様々な機関で盛んに研究・開発がなされている．従来型コンピュータ上では情報の最小単位であるビットを 0 または 1 のどちらかの状態として表すが，量子コンピュータでは量子の持つ性質を利用して 0 と 1 が重なり合った状態を保持することができる量子ビットを扱うことが大きな特徴であり，この量子ビットを用い，量子ゲートなどを適切に組み合わせることで計算の解が出力される確率を高めて答えを導き出す．また，このとき複数通りの計算を同時に行うことと同様の挙動をするため高速な計算を実現することができるとされる（図 1 参照）．量子コンピュータにより高速化が期待されている計算問題の一例として，例えば量子化学計算などが挙げられる．しかし，現時点でもすでに複数の量子コンピュータが存在するものの，いずれもノイズの影響で計算結果にエラーが含まれるなど完全な挙動をするものとは言い難い．実際に理想的な挙動をする量子コンピュータが実現するまで，数年から数十年程度必要と言われている．

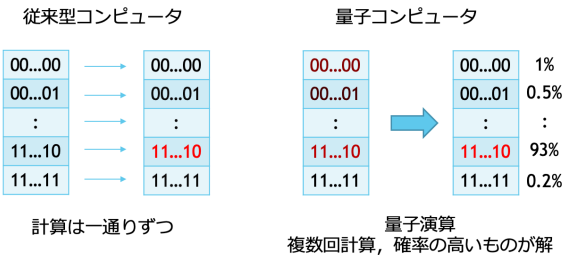


図 1　量子コンピュータの挙動

そして現在，従来型コンピュータ上で量子コンピュータの挙動を表現してシミュレートすることが可能な量子回路シミュレータが存在する．量子回路シミュレータを用いると計算結果は常に正確なものを得ることが可能である．量子コンピュータの実現まで数十年かかる可能性がある中で，量子コンピュータの開発と並行して量子回路シミュレータを使用して量子アプリ開発などを行うことは有用であり，このことから量子回路シミュレータに関する研究を行うことは有用性が高い．現在も IBM が開発した Qiskit Aer [1] や国立大学法人大阪大学と株式会社 Qunasys が共同開発した Qulacs [2]，Google が開発した Cirq [3] などの量子回路シミュレータが存在し，各シミュレータのユーザも多く有用であるが，それぞれの性能特性の違いは明らかになっていない．

そこで本研究では，複数の量子回路シミュレータを使用した複数の量子回路について性能分析を行い，性能特性の違いを明らかにする．前報 [4] では一種類のサーバ上で量子回路シミュレータの性能分析を行なった．本稿では性能が異なる複数種の実験用サーバ上でそれぞれ量子回路シミュレータを動作させ，それと同時にシステム稼働状況を収集し，実験環境の違いによる特性の違いを明らかにする．さらに収集したデータを用いて分析対象の量子回路シミュレータの動作改善方法について検討する．

2　関連研究

量子回路シミュレータには複数の方式が存在し，各方式の様々な量子回路シミュレータが存在する．その中でも State Vector 方式は，すべての量子状態をメモリに保持するため状態ベクトルをまとめて取得可能であり，計算量が量子回路の形状の影響を受けにくい．また，このような構造から動作途中の量子状態

を得ることも可能であるため，例えば量子アプリのデバッグ作業に有用である．Tensor Network 方式では量子回路をテンソルネットワークとして表す．また，シミュレートにあたって必要な量子状態のみを計算する．Density Matrix 方式では，量子状態を行列として表現しつつシミュレートを行う．しかしこのとき，必要なメモリ量は非常に多くなる．

例えば State Vector 方式の量子回路シミュレータに，Intel-QS がある [5]．Intel-QS は CPU ベースの分散型 State Vector 方式シミュレータであり，マルチコア CPU での使用のために OpenMP でマルチスレッド化されている．また，ひとつの大規模な回路だけでなく，複数の小/中規模回路のシミュレートも可能である．さらに Tensor Network 方式の量子回路シミュレータの例として，例えば TeNeS [6] が挙げられる．TeNeS は多体量子状態計算を行うためのオープンソースプログラムパッケージであり，OpenMP と MPI ハイブリッド並列に対応している．

続いて量子回路シミュレータの性能向上に関する関連研究を示す．今村らによって開発された mpiQulacs [7] は，State Vector 方式の量子回路シミュレータのひとつである Qulacs [2] を，A64FX CPU に対して最適化された量子回路シミュレータである．MPI を使用することで並列分散処理に対応している．演算を行う際は，CPU に搭載されている複数の演算を一度にまとめて実行することができる命令を使用することでメモリバンド幅を引き上げている．また，状態ベクトルを複数に分割し，各プロセスで均等に保持してローカルで計算を行い，他プロセスと状態ベクトルを共有する時のみ MPI 通信を使用することで並列処理を実現している．この他にも高速化のための工夫を複数行うことで，数十ノード単位の大規模シミュレーションに関して，他の量子回路シミュレータと比較して性能が高くなったことを確認している．

3 量子回路シミュレータ

前章でも述べたように，量子回路シミュレータには State Vector 方式と Tensor Network 方式など複数の方式が存在する．その中でも State Vector 方式は前述のとおり途中の量子状態を取得することが可能なため量子アプリ等の開発における有用性が高いという点から，今回の実験では State Vector 方式のシミュレータについて分析を行う．以下で分析に用いる量子回路シミュレータについて紹介する．

3.1 Qiskit Aer

Qiskit Aer [1] は量子コンピュータ用のフレームワークである Qiskit [8] に含まれる量子回路シミュレータである．Qiskit では量子回路を構築して実際の量子デバイス上で実行することが可能であり，モジュール構造のため最適化などの処理も行いやすく設計されている．さらに従来型コンピュータ上でも量子回路をシミュレートすることが可能となっており，量子回路シミュレータの方式は State Vector 方式と Tensor Network 方式の両方に対応している．

3.2 Qulacs

Qulacs [2] は国立大学法人大阪大学と株式会社 Qunasys によって開発された State Vector 方式の量子回路シミュレータである．主要部分は性能を重視しながら C++で実装されている．また使用時は Python インタフェースを介して使用することができるため，高速なシミュレーションと高い利便性を実現している．

さらに本稿では前章で紹介した mpiQulacs も実験対象シミュレータとして使用する．

4 性能分析ツール

本稿で使用した性能分析ツールについて説明する．

4.1 mpstat

mpstat は，CPU 使用率やハードウェア割り込みなどの情報を取得することが可能なコマンドである．通常各カテゴリについて，すべての CPU の平均値と各 CPU ごとの統計情報を得ることができる．

mpstat を用いることで各コアの挙動を詳細に確認することができるため，I/O 競合発生の可能性を発見することができるなどボトルネックの要因特定に有用である．本稿では特にユーザ時間割合を抜粋して紹介する．

4.2 Linux Perf

Linux Perf [9] は，主に Linux Kernel のイベントデータ収集，トレース機能を提供しているツールである．Linux Perf にはいくつかのサブコマンドが存在し，Linux Perf を使用する際は通常これらサブコマンドを併用する．

今回の実験では stat というサブコマンドを使用する．perf stat は，あるプログラムの実行中に指定したイベント値を出力することなどが可能である．取得することができるデータに，例えばページフォールトの回数やキャッシュヒット/ミス数などがある．

4.3 PCM Tools

PCM Tools [10] は CPU のリソース利用に関する情報を取得することができるツールである．PCM には複数のコマンドが含まれており，例えば pcm-memory を用いるとメモリバンド幅のモニタリングが可能である．実際に計測する際は必要に応じてこのようなコマンドを使用する．

なお，本稿では Primergy RX2540 M1 サーバにおけるデータ収集にてこの pcm-memory を用いている．

5 実　　験

5.1 実験概要

本稿では，Qiskit Aer と Qulacs それぞれのライブラリを使用した量子回路シミュレータの Python プログラムを実行し，実行中のシステム稼働状況を複数の性能分析ツールを用いて収集する．

使用する量子回路シミュレータのバージョンを表 1 に示す．

表 1　量子回路シミュレータのバージョン

qiskit	0.39.4
qiskit aer	0.11.2
qulacs	0.6.3
mpiQulacs	1.3.1

量子体積 (QV: Quantum Volume) は，量子コンピュータのシステムを評価する目的で，どの程度複雑な処理が可能かを示す指標として提案 [11] された．量子体積モデル回路はこの量子体積を評価するためのものであり，ランダムな組み合わせの量子ビットのペアに 2 量子ゲートを作用させる．以降この回路を QV 回路と記述する．

今回の実験には，Qiskit Aer，Qulacs，mpiQulacs それぞれを使用した各 Python プログラムに，この QV 回路を用いた．

測定時の量子ビット数は 30，回路の深さは 10 に設定した．

プログラム実行中の性能データ収集にあたり，bash スクリプトを使用した．動作は性能分析ツールの実行を開始した後シミュレーションプログラムの実行を開始し，シミュレーション実行が終わると性能分析ツールによるデータ収集も終了する形にした．また性能分析ツールの設定について，mpstat と vmstat は fflush() を用いることでデータ収集の間もバッファのフラッシュを行う．

また，Primergy RX2540 M1 サーバにおいて perf stat を使用して収集するイベントデータは以下に示すイベントに関して収集を行う．

- コアサイクル数，実行命令数，L1D キャッシュミス数，L1D キャッシュ参照数
- コアサイクル数，実行命令数，L2 キャッシュミス数，L2 キャッシュ参照数
- コアサイクル数，実行命令数，L3 キャッシュミス数，L3 キャッシュ参照数

さらに，同じく実験に使用する FX700 サーバにおいては以下に示すイベントに関してデータ収集を行う．

- コアサイクル数，実行命令数，L1D キャッシュミス数，L1D キャッシュ参照数，L2 キャッシュ参照数，L2 キャッシュミス数

そして Primergy RX2540 M1 サーバにおけるメモリバンド幅データ収集については PCM Tools の pcm-memory を用いる．また，今回使用する性能分析ツールは，すべて 1 秒間隔でデータの収集を行う．

5.2 実験環境

2 つの実験環境をそれぞれ表 2 と表 3 に示す．

以降，Primergy RX2540 M1 を RX サーバと記述する．また，mpiQulacs は A64FX CPU に対して最適化されたシミュレータであり，そのために FX700 サーバ上でのみ性能分析を行う．

表 2　実験環境 1

実験用サーバ	Primergy RX2540 M1
OS	Rocky Linux 8.6
CPU	Intel Xeon プロセッサー E5-2697v3(2.60GHz)
CPU コア数	14
L1d/L2/L3 キャッシュ	32 KB(core)/256 KB(core)/35 MB
メモリ	DDR4(2133) 128 GB

表 3　実験環境 2

実験用サーバ	FX700
OS	Rocky Linux 8.5
CPU	Fujitsu A64FX (2.0GHz)
CPU コア数	48
L1d/L2 キャッシュ	64KB(core)/8MBx4(CMG)
メモリ	HBM2 32 GB

5.3 実験結果

5.3.1 実行時間

はじめに量子回路の実行時間について表 4 に結果を示す．この表から，Qiskit Aer シミュレータと Qulacs シミュレータの比較において，今回の実験環境どちらも Qiskit Aer シミュレータと比べて Qulacs シミュレータの方が十数秒から数十秒程度量子回路の実行時間が短いことがわかる．そして FX700 サーバ環境において，mpiQulacs は Qulacs シミュレータよりさらに 30 秒程度回路実行時間が短いことがわかる．

表 4　量子回路実行時間（秒）

	RX	FX700
Qiskit Aer	129.1	77.5
Qulacs	119.0	49.5
mpiQulacs	-	19.5

さらに今回使用したシミュレーションプログラム内における処理の実行時間内訳を以下の図 2，図 3，図 4，図 5，図 6 にそれぞれ示す．なお，各グラフの右に処理内容を記述しており，プログラム中では記述されている処理内容のうち上に書かれた処理から順に実行している．

これらの図から Qiskit Aer シミュレータでは RX サーバ，FX700 サーバ共にサンプリング処理を含めた回路実行にかかる時間が最も長いことがわかる．同時に FX700 サーバ上では Qiskit ライブラリのインポートに 8 %程度の時間がかかっており，比較的目を引く結果であった．また，Qulacs シミュレータについては RX サーバ上では回路実行時間が最も長く，FX700 サーバ上ではサンプリング処理の時間が最も長いことがわかる．さらに mpiQulacs シミュレータについては，回路実行時間とサンプリング処理が実行時間のほとんどを占めており，各処理の時間割合の差は 2 %程度であった．同時に，Qulacs ライブラリ以外のライブラリのインポートに 5 %程度の時間を使用しており，こちらも比較的目を引く結果となった．

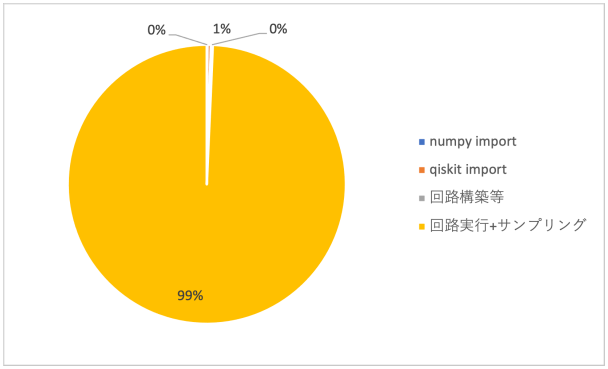


図 2　RX サーバ Qiskit Aer 実行時間内訳

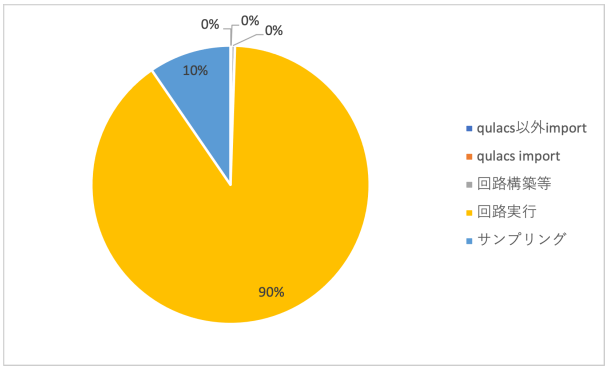


図 3　RX サーバ Qulacs 実行時間内訳

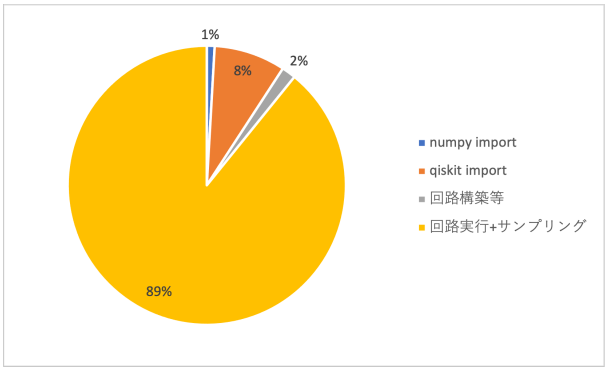


図 4　FX700 サーバ Qiskit Aer 実行時間内訳

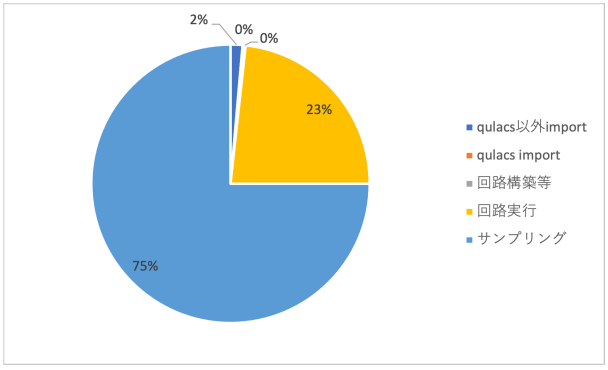


図 5　FX700 サーバ Qulacs 実行時間内訳

5.3.2　mpstat

続いて mpstat の結果について記述する．

mpstat で取得したプログラム実行中のユーザ時間割合の全コア平均値のグラフを図 7 に示す．

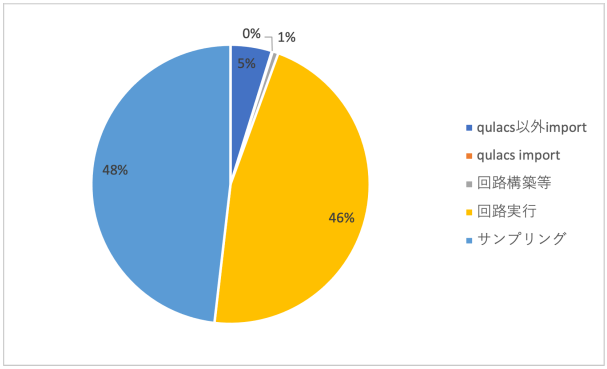


図 6　FX700 サーバ mpiQulacs 実行時間内訳

このグラフから，Qiskit の QV 回路は実行中ユーザ時間割合を約 100 %で保っていることがわかる．また，Qulacs QV 回路は，RX サーバ上では基本的に 100 %程度の値を保っているものの，終盤に数秒間ユーザ時間割合が低下した．また FX700 サーバ上では実行開始から 10 秒程度 100 %となっているが，その後著しく割合が低下した．Qulacs QV 回路のこれら現象については，図 3 と図 5 の結果からサンプリング処理が関係していると考えられる．さらに mpiQulacs は最大ユーザ時間割合が 90 %程度であり，図 6 よりこの部分は量子回路実行に対応，その後著しく値が低下する部分はサンプリング処理に対応していると考えられる．

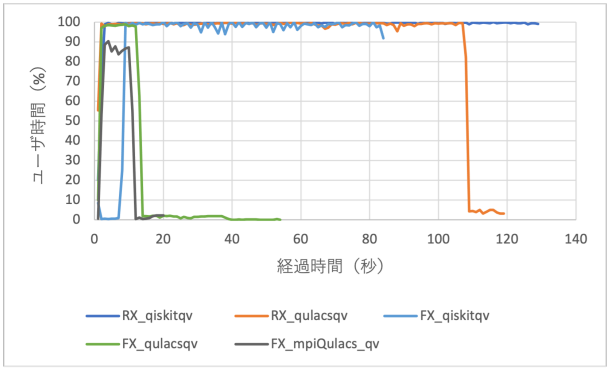


図 7　ユーザ時間割合（%）

5.3.3　Linux Perf

続いて Linux Perf における分析結果について記す．今回，サブコマンドとして stat を使用してキャッシュミス率等を取得した．

各環境におけるキャッシュミス率は以下の通りである．表 5 に RX サーバにおける L1D，L2，L3 キャッシュミス率を示す．さらに表 6 に FX700 サーバ上における L1D，L2 キャッシュミス率を示す．

また，図 8 に RX サーバにおける L1D キャッシュミス率，図 9 に L2 キャッシュミス率，図 10 に L3 キャッシュミス率を示す．さらに，図 11 に FX700 サーバにおける L1D キャッシュミス率，図 12 に FX700 サーバにおける L2 キャッシュミス率をそれぞれ示す．

表 5　RX サーバ キャッシュミス率（%）

	L1D	L2	L3
Qiskit Aer	1.5	45.9	67.9
Qulacs	2.7	54.3	38.4
mpiQulacs	-	-	-

表 6　FX700 サーバ キャッシュミス率（%）

	L1D	L2
Qiskit Aer	1.5	29.6
Qulacs	2.8	34.2
mpiQulacs	3.8	56.1

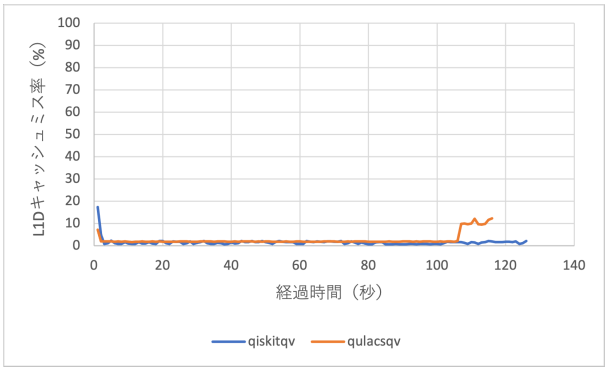


図 8　RX サーバ L1D キャッシュミス率（%）

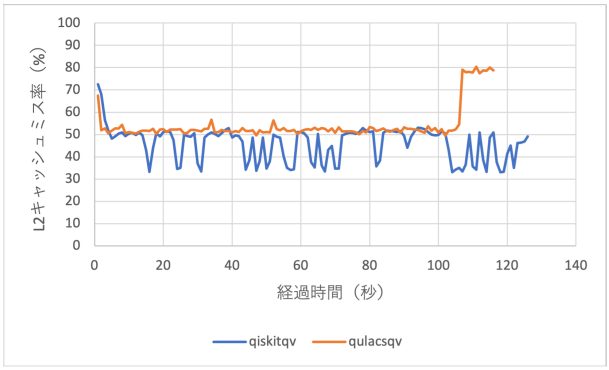


図 9　RX サーバ L2 キャッシュミス率（%）

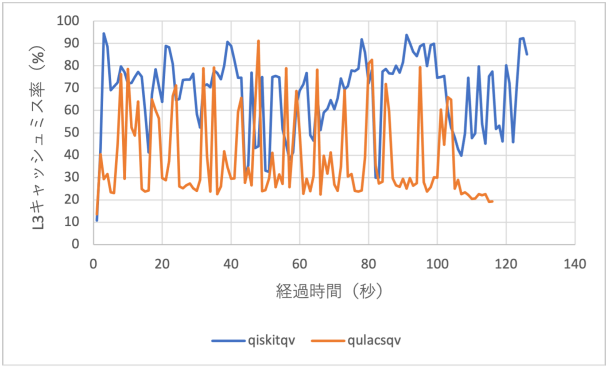


図 10　RX サーバ L3 キャッシュミス率（%）

表 5 から，L1D，L2 キャッシュミス率は Qulacs の方が高いが，L3 キャッシュミス率は Qiskit Aer より Qulacs の方が 43.4 %程度低い値となっていることがわかる．

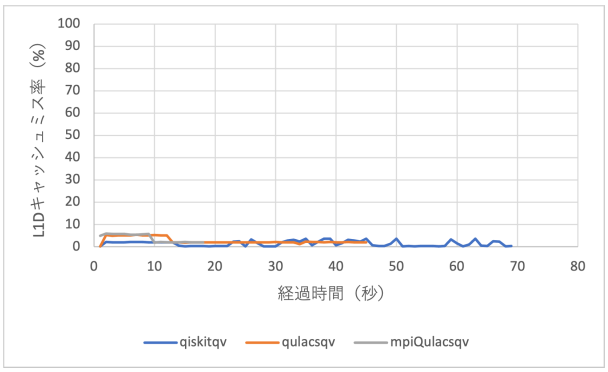


図 11　FX700 サーバ L1D キャッシュミス率（%）

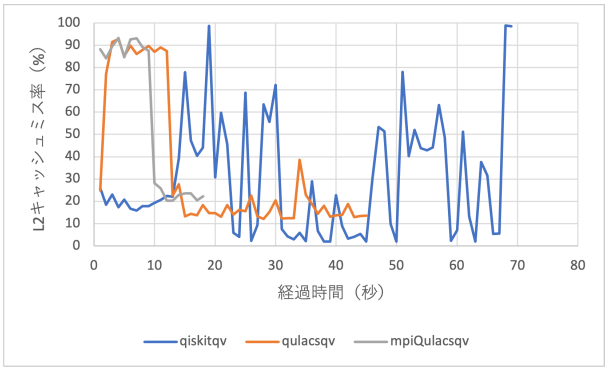


図 12　FX700 サーバ L2 キャッシュミス率（%）

さらに表 6 から，L1D キャッシュミス率，L2 キャッシュミス率の各項目についてすべて Qiskit Aer シミュレータよりも Qulacs シミュレータの方がキャッシュミス率が高く，そして Qulacs シミュレータよりも mpiQulacs の方がさらにキャッシュミス率が高い結果となった．

5.3.4　メモリバンド幅

続いて各サーバ上におけるメモリバンド幅計測結果について記述する．RX サーバのメモリバンド幅理論値は 68 GB/s，FX700 サーバにおけるメモリバンド幅理論値は 1024 GB/s である．

はじめに RX サーバについて，前述の通りメモリバンド幅の計測は PCM Tools のサブコマンドである pcm-memory を使用した．図 13 に結果のグラフを示す．このグラフから，プログラム実行中 Qulacs シミュレータは Qiskit Aer シミュレータと比べてメモリバンド幅を引き出していることがわかる．

続いて FX700 サーバ上におけるメモリバンド幅を図 14 に示す．このグラフから，Qulacs シミュレータは実行開始から 10 秒程度に渡りメモリバンド幅を 500 GB/s 近くまで引き出していることがわかる．また mpiQulacs も同様に実行開始から 10 秒程度 600 GB/s 近いメモリバンド幅を引き出していた．Qulacs シミュレータと mpiQulacs について，メモリバンド幅を高水準で引き出しているときの処理については，図 5 と図 6 から量子回路実行処理中であると考察される．

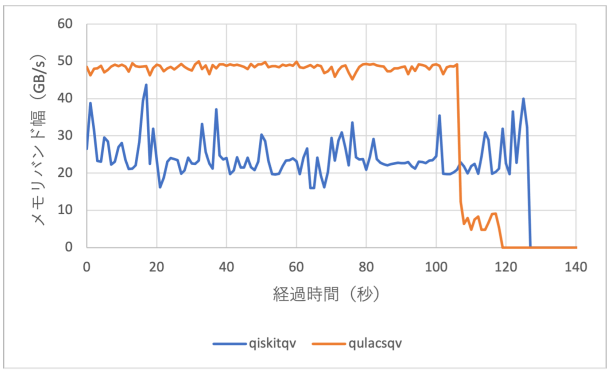


図 13　RX サーバ メモリバンド幅（GB/s）

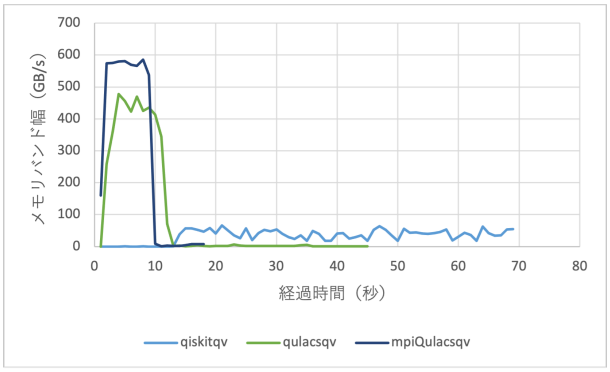


図 14　FX700 サーバ メモリバンド幅（GB/s）

6　まとめと今後の課題

本稿では異なる環境下で複数の量子回路シミュレータの性能データ収集を行い，性能の分析を行った．

その結果，量子回路の実行時間は両環境ともに Qiskit Aer シミュレータよりも Qulacs シミュレータの方が短い結果となった．さらに FX700 環境下において，mpiQulacs はこれら 2 つのシミュレータよりもさらに短い実行時間を記録した．そしてプログラム中の関数の実行時間割合の分析から，Qiskit Aer シミュレータは両環境において量子回路実行時間がその大半を占めていることがわかった．Qulacs シミュレータについては RX サーバ上では量子回路実行に消費する時間が比較的多く，FX700 サーバ環境下ではサンプリング処理に消費する時間が多いことがわかった．さらに mpiQulacs は量子回路実行とサンプリング処理にかかる時間割合が同程度であるとわかった．

また，mpstat の結果から Qiskit Aer シミュレータ実行時はプログラム実行中約 100 %のユーザ時間割合を保っていたことがわかるが，Qulacs シミュレータと mpiQulacs についてはサンプリング処理実行中と思われる瞬間著しく割合の低下が見られた．また，mpiQulacs の最大ユーザ時間割合は他 2 種類のシミュレータよりも 10 %程度低いことがわかった．

さらに perf stat を使用したキャッシュミス率の分析結果から，RX サーバでは L1D，L2 キャッシュミス率について Qiskit Aer シミュレータと比べて Qulacs シミュレータの方が高く，L3 キャッシュミス率は Qiskit Aer シミュレータの方が高い結果になった．また，FX700 サーバは L1D，L2 キャッシュミス率共に Qulacs シミュレータの方が高く，さらに mpiQulacs は Qulacs シミュレータよりも両キャッシュミス率が高い結果となった．

またメモリバンド幅の分析から，両環境共に Qulacs シミュレータは Qiskit Aer シミュレータと比べて明確にメモリバンド幅を引き出しており，FX700 サーバ環境下では mpiQulacs が最もメモリバンド幅を引き出していることがわかった．

これらの結果から，キャッシュミス率を抑えたとしても実行時間に与える影響は比較的小さく，メモリバンド幅を引き出すことが実行時間の短縮につながる可能性があるという考察が前報 [4] と比べさらに強固になった．さらに FX700 サーバにおいて，Qulacs シミュレータと mpiQulacs の処理のうちユーザ時間割合が低下しているサンプリング処理についても，より効率的な CPU 使用をするよう改善することで性能が高まる可能性も考えられる．

今後もさらに大規模システムを使用するなどし，量子回路シミュレータについて詳細な分析を進める．そしてその結果を基にシミュレータの動作改善方法を模索し，実装して検証を重ねていく．

謝　　辞

本研究の一部はお茶の水女子大学と富士通株式会社との共同研究契約に基づくものであり，JST CREST JPMJCR22M2 の支援を受けたものである．

文　　献

[1] Qiskit aer simulator. https://qiskit.org/documentation/tutorials/simulators/1_aer_provider.html.
[2] Qulacs ドキュメンテーション — qulacs ドキュメント. http://docs.qulacs.org/ja/latest/index.html.
[3] Cirq. https://quantumai.google/cirq.
[4] 青木 望美, 山崎 雅文, 平井 聡, 山岡 茉莉, 福本 尚人, and 小口 正人. 量子回路シミュレータの性能向上のための分析と考察. マルチメディア、分散、協調とモバイル *(DICOMO2023)* シンポジウム, 2023.
[5] Intel quantum simulator — intel qs documentation. https://intel-qs.readthedocs.io/en/docs/getting-started.html#.
[6] Tenes - pasums - 東京大学. https://www.pasums.issp.u-tokyo.ac.jp/tenes.
[7] Imamura Satoshi, Yamazaki Masafumi, Honda Takumi, Kasagi Akihiko, Tabuchi Akihiro, Nakao Hiroshi, Fukumoto Naoto, and Nakashima Kohta. mpiqulacs: A distributed quantum computer simulator for a64fx-based cluster systems. *Distributed, Parallel, and Cluster Computing*, 2022.
[8] Qiskit. https://qiskit.org.
[9] Perf wiki. https://perf.wiki.kernel.org/index.php/Main_Page.
[10] Intel® performance counter monitor - a better way to measure cpu utilization. https://www.intel.com/content/www/us/en/developer/articles/technical/performance-counter-monitor.html.
[11] A. W. Cross, L. S. Bishop, S. Sheldon, P. D. Nation, and J. M. Gambetta. Validating quantum computers using randomized model circuits. *Phys. Rev. A*, Vol. 100:p.32328(online), 2019. DOI: 10.1103/PhysRevA.100.032328.