DEIM2021 B25-4

外部整合的な地理分散トランザクションへの並列ロギングの適用

小倉　拓人† 　秋田　佳紀† 　宮澤　勇貴† 　川島　英之††

† 三菱 UFJ インフォメーションテクノロジー株式会社　〒 104-0053 東京都中央区晴海 2 丁目 1-40
†† 慶應義塾大学環境情報学部　〒 252-0882 神奈川県藤沢市遠藤 5322
E-mail: †{takuto_ogura,yoshiki_akita,yuki_miyazawa}@mufg.jp, ††river@sfc.keio.ac.jp

あらまし　決済用途に使われるトランザクションには外部整合性と信頼性が求められるが，これらの性質を保証するためには性能が犠牲となる．本研究ではシングルマスタ方式に基づき，これらの性質を保証しながら高性能を達成する機構を提案する．我々の提案は複製管理に集約ログ転送法で高性能化された Raft を用い，ロギングに並列ロギング手法 P-WAL を用いる．P-WAL では, 複数のログライタがそれぞれの専用領域に並列でログ書き込むことで, ロギング処理の高速化を行う．評価実験として，一括転送するログの集約数やロギングの並列数を変化させた場合の処理性能を測定した．その結果，実験ワークロードにおいて 10M tps を超える性能が観察された．

キーワード　地理分散データベース, 分散トランザクション, 分散一貫性, 障害回復

1 はじめに

1.1 動　　機

決済取引を扱う金融システムで用いられるデータベースには，高い信頼性と一貫性，そして高性能であることが求められると我々は考えている．これまでに多くの金融システムでは，その基幹をメインフレーム上に構築することで高信頼性を保証してきた．メインフレームはこれまで金融システムの安定稼働を支えてきた反面，維持管理のコストやシステム変更の柔軟性が低いといった問題が挙げられ，将来的には金融システムにおいてもメインフレームから Linux 等のオープン系システムに移行すると予測される．また近年では，電子決済の普及等に伴い，金融サービスのあり方にも変化が求められている．これまでのように中央集権型のシステムではなく，決済サービスにおいてもマイクロサービスのような分散型のアーキテクチャの採用を検討する必要があると考えられる．マイクロサービスの場合，トランザクションの整合性はアプリケーション側の実装を工夫することで保証する必要があり，整合性を保証した高速な処理を実現することはアプリケーション側の開発に多大なコストをかけることとなる．整合性を保証し，かつトランザクションを高速で処理することができる分散データベースを利用することができれば，分散システムにおける開発コストを削減し，多様なサービスの提供を実現することが期待される．金融システムがこうした急速な変化に対応するためにも，オープン系システム上で，高い信頼性と一貫性を保つことができる高性能なデータベースを開発することが求められる．

本研究では，オープン系システム上に構築することを前提とし，高信頼性を保証するために複数のノードにデータを複製し保有する分散データベースの仕組みを採用する．ノードを複製するバックアップノードは，災害時の安定稼働のため，日本全国に地理分散配置されていることを想定する．一貫性を保証するために，トランザクション一貫性レベルで最も高い外部整合的であることを条件とする．外部整合的であるとは，複製一貫性の指標で最も高い線形化可能一貫性レベルと，トランザクション分離性で最も高い逐次化可能分離レベルの双方を満たすことを表す．筆者らの目標性能を以下に示す．

（1） 1 秒間あたりのトランザクションの処理性能 (TPS; Throughput) が 1M tps 以上

（2） 1 件あたりのトランザクションの処理にかかる平均遅延時間 (RTT; Round Trip Time) が 400 ミリ秒以内

分散データベースの既存プロダクトとして，Spanner [1] や CockroachDB [2] が挙げられるが，いずれも上記の性能要件は満たしていない．外部整合性を保証し，かつこの性能を満たすシステムは我々の知る限り存在しない．

1.2 研 究 課 題

地理分散トランザクションを対象にして外部整合性を保証するアプローチにはシングルマスタ方式 [1] [2] [3] とマルチマスタ方式 [4] [5] [6] がある．前者の一部 [1] [2] は既にプロダクトになっており安定稼働実績を有するが，性能は TPC-C で 28,000tps 程度に留まる [1]．後者は multi-partition 方式において，各ノードの保持するデータに偏りがない場合に高性能を示すが，依然として研究段階であり安定稼働を有するプロダクトは存在しない．本研究の目的を達成するには，前者を高速化するか，あるいは後者を安定化させるかのいずれかとなる．本研究構想で念頭に置いている実用化に際して，困難なのは障害回復や進行性保証などのエンジニアリング詳細である．これらの理解には長い時間が必要となるため，研究段階システムの安定化にはリスクが高い．そのため本研究では前者の高性能化というアプローチを採用する．

シングルマスタ方式を高性能化する研究には，Spanner [1], CockroachDB [2], Carousel [3] がある．Spanner は TrueTime を用いることで分散トランザクション処理の並行性制御を高性

能化した．CockroachDB は Spanner の設計を踏襲したシステムであり分散合意アルゴリズム Raft [7] を用いて合意形成を行う．Carousel はトランザクションの中でも 2-round Fixed-set Interactive (2FI) を対象にし，Fast Paxos に基づき 2-phase commit を高速化し，TAPIR [8] に対する優位性を示した．これらの研究は並行性制御法の改善に努め，大きな性能向上をもたらした．

トランザクション処理の構成要素には並行性制御法 [9] に加えてロギングがある [10]．筆者らの研究 [11] では，既存研究と視点を変えロギングの効率化に注目した．これまでに筆者らは，Raft [7] と Strict 2-Phase Locking(S2PL) [10] から成る方式に，複数ログをまとめて転送する集約ログ転送法 [12] を適用した分散データベースを構築し，その性能を評価した [11]．集約転送により TPS の大幅な向上を実現することができた一方で, 日本全国にノードを地理分散配置することをエミュレートした実験においては，TPS が最大で 0.8M tps 程度と，我々の目標とする 1M tps の性能を達成できなかった．ロギングには更なる性能改善の余地があるのだろうか？

1.3 提　　案

集約ログ転送法は複数のトランザクションを一括処理することで性能改善を実現した．同方式ではログデータの永続化処理は一つのスレッド（Raft システムにおける `Timer` スレッド）により行われる．従って複数の Worker スレッドは単一のログバッファにアクセスせざるを得ず，アクセス衝突に伴う待機時間が発生する．コア数が増大する昨今，この方式では先進的アーキテクチャの恩恵に浴することができない．また，昨今のストレージデバイスは並列的な書き込みにより高性能を示すが, 我々の知る限りこの特性は地理分散トランザクション処理研究で活用されたことはなかった．

本論文は，ストレージアクセスに着眼し，ロギング並列化による外部整合的地理分散トランザクション処理の高性能化を提案する．非分散システムにおいてはロギングが並列化可能なことは明らかにされてきた [13] [14] [15] [16]．その中でも `epoch` を利用しない方式である Passive Group Commit [14] と P-WAL(early lock release 方式) [15] では，ロック解放後に複数の Worker スレッドが並列的にログを永続的デバイスに転送し，それらのログシーケンス番号 (`LSN`) の `minmax` 値を求めることでリカバリ可能下限を算出する．他方，P-WAL(conservative lock release 方式) [16] ではロギング完了までロックを解放しないためにリカバリ可能下限の算出は不要である．Passive Group Commit は DRAM を一切利用しない方式であるのに対して，P-WAL は DRAM を有するマシンを前提としたプロトコルである．

我々は P-WAL(conservative lock release 方式) [16] を文献 [11] に示した Raft と S2PL の組合せ方式に統合する．各トランザクションは S2PL に基づき必要なロックを全て獲得した時点でプリコミットとする．この時点ではデータベースの修正は行わない．Leader は Follower にログを転送して永続化に関する合意を形成する．それからデータベースの修正を行い，最後にロックを解放する．データベース修正を合意形成後に行う理由はアボート時のコストを抑制するためである．データベースの修正が完了した後，そのトランザクションの完了通知をクライアントへ返す．提案手法を実装して AWS EC2 にエミュレートした地理分散環境で性能を評価した結果，提案手法が 10M tps を超える性能を示すことが観察され，提案手法により要求性能が達成されることが明らかとなった．

1.4 論 文 構 成

本論文の構成は以下の通りである．まず準備として，これまでに開発した Raft と S2PL による分散データベース, 集約ログ転送法, および並列ロギングについて 2 章で述べる．3 章では, Raft に対して集約ログ転送法と並列ロギングを適用した提案手法の概要を述べる．4 章では, 提案手法の評価実験を行い, 得られた結果をもとに手法の有効性を評価する．5 章では関連研究を述べ，最後に 6 章で結論を述べる．

2 準　　備

2.1 Raft と S2PL の統合によるトランザクション処理

我々の開発する分散データベースでは，外部整合性を保証するために，Raft と S2PL を組み合わせた手法を採用している．Raft は分散合意アルゴリズムのひとつであり，線形化可能一貫性レベルを保証する．分散合意手法においては Paxos [17] や Viewstamped Replication [18] などがあるが，それらと比較してシンプルで理解しやすいという特徴がある．

Raft では，Leader と呼ばれるマスタとなるノードが最大 1 つと，それ以外の Follower と呼ばれるノードによって構成される．Leader ノードは，Client からのコマンドを受信し，自身のログへの書き込みや，Follower への複製を行う．Follower ノードは Leader からのログを受信し，それを自身に複製することで，バックアップとする．ここで，コマンドとは最終的にステートマシンに適用される一連の更新操作を指す．本研究ではコマンドを決済トランザクションと想定する．ログとは，コマンドの適用履歴であり，ログに書き込まれるコマンドをエントリと呼ぶ．

また，S2PL は並行性制御手法のひとつである．データアクセス時にロックを順次取得し，トランザクション内の全ての処理が完了してから全てのロックを解除，コミットする．これにより，逐次化可能分離レベルを保証する．

Raft と S2PL の組み合わせによるトランザクション処理をアルゴリズム 1 に示す．Leader ノードでは，Client からのコマンドを受信した後，S2PL によりコマンドに含まれるデータアイテムのロックをアルファベット順に取得する．次に，コマンドを実行し，データアイテムの更新を行う．コマンドの実行履歴はエントリとしてログに書き込む．Leader ノードはこのログを各 Follower に対して転送し，Follower からの応答を待ち受ける．Follower ノードは Leader からのログの複製が完了した場合は，RPC 応答を Leader に対して送信する．Leader では，過半数以上の Follower からの応答の受信が完了したら，

ロックしていたデータアイテムをアンロックし，Client に対してコミット通知を返却する．

Algorithm 1 Raft と S2PL における Leader ノードの処理

```
1: Recieve client command.
2: Lock data item in client command.
3: Update data item.
4: Locks own log.
5: Add entries to log.
6: Unlock log.
7: Broadcasts the entry to all follower.
8: if Received RPC responses from the majority of followers. then
9:     Unlock data item.
10:    Return the commit response to the client.
11: end if
```

2.2 集約ログ転送法

Raft と S2PL を組み合わせることにより，外部整合性を保証した分散データベースを実現することができる．一方で，この方式ではコマンドを 1 件ずつ実行し，ログを Follower に複製することになるため，処理性能が犠牲となる．そこで，複数のログを集約して転送することで，ノード間の通信回数の削減し，処理効率を向上させる集約ログ転送法を用いる．集約ログ転送法では Client から Leader へ送付するトランザクションと，Leader から Follower へ転送するログをそれぞれ集約する．ここで，Client から Leader へ一度に送付するコマンドを TxGrp とし，1 つの TxGrp に含まれるコマンドの数を TxGrpSize とする．また，Leader から Follower へ一度に転送するログエントリを LogGrp とし，1 つの LogGrp に含まれるエントリ数を LogGrpSize とする．筆者らは TxGrpSize と LogGrpSize を変化させた場合にトランザクションの処理性能がどのように変化するかを評価し，集約ログ転送法により，性能が大幅に向上することを明らかにした [11]．各ノード 5 都市に地理分散したと想定した実験において 0.8M tps 以上の性能を記録している．一方で，アルゴリズム 1 における Leader がログにエントリを追加する処理については，ログの集約数によらず逐次する処理する必要がある．

2.3 並列ロギング手法 P-WAL

ロギング処理を並列化する手法 P-WAL の概要について示す．P-WAL では，並列で動作する複数の Worker スレッドを用意し，それぞれに専用の WAL バッファと WAL ファイルを割り当てる．各 Worker スレッドは WAL バッファへのログレコードの挿入と WAL ファイルへの書き込みを独立して行うことで，ロギング処理を並列化する．Worker スレッドが書き込む WAL ファイルは独立しており，スレッド間の排他制御や競合については発生しない．一方で，このままでは Worker 間のログの順序性については保証されておらず，障害発生時に正常にリカバリすることができない．そこで P-WAL では，ログレコードに対して順序を一意に同定する番号 `LSN` を，共有カウンタを用いて割り当てることで，この問題を解決している．リカバリを行う際には，Worker スレッド間のログをマージし，`LSN` に基づいて順序を同定することで，正しい順序でデータを復元することが可能である．本研究では，この P-WAL を分散合意アルゴリズムである Raft に適用する．Raft におけるロギング処理を並列化することで，処理性能の高速化を図る．

3 提 案 手 法

Raft に対して P-WAL を適用した提案システムの概要について示す．システムはマルチスレッドで構成され，各スレッドが非同期で動作することで，並列ロギングを実現する．これにより，従来の Raft におけるロギングが逐次処理されるという課題を解決し，マルチコアのマシンリソースを有効に活用することが期待できる．また，並列ロギングを分散トランザクションに適用することで，Follower へのログの転送についても並列化される点についても提案手法の特徴である．各スレッドの役割やコミット時の処理について本章で述べる．

3.1 概　　要

提案システムのアーキテクチャを図 1 に示す．

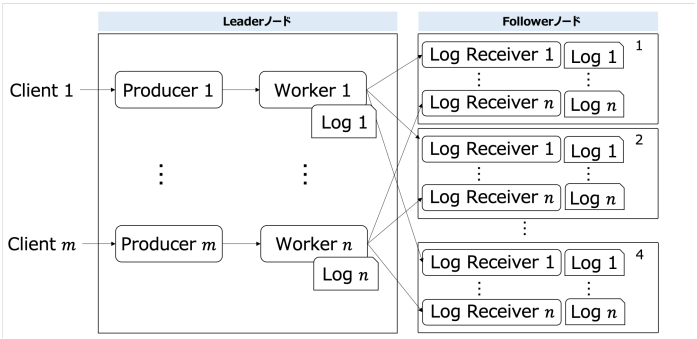

図 1　提案システムのアーキテクチャ

ロギング処理を並列化するために，Leader ノード内に複数の Worker スレッドを起動させる．また各 Follower では，Leader ノードの Worker スレッドに対応する形でログエントリを受信する Log Receiver スレッドを起動させる．Leader ノードは，Client から転送されるコマンドを Producer スレッドで受信する．Producer スレッドは，受信した TxGrp を 1 コマンドごとに分割し，各 Worker スレッドに割り当てられたキューに格納することで Worker スレッドにコマンドを渡す．各 Worker スレッドがこのコマンドをそれぞれ独立に処理し，ログの書き込みや Follower への複製を行うことで，Raft での並列ロギングを実現する．

3.2 Worker スレッド

Worker スレッドで行われる処理をアルゴリズム 2 に示す．Worker スレッドは，まず，自身に割り当てられたキューからコマンドを 1 件ずつ取り出す．取り出したコマンドに含まれるデータアイテムのロックを行なった後，自身の専用領域であるログバッファにコマンドを格納する．このログバッファに含ま

れるコマンドの数が，予め指定した LogGrpSize に達するか，またはタイムアウトになった場合，ログバッファの内容をログエントリとしてファイルに永続化し，各 Follower に転送する．各 Worker スレッドがログを書き込むログファイルは Worker スレッドごとに独立している．この際，Worker スレッド間のログの順序性を維持するために，LSN を LogGrp と対応する形で採番し，ログエントリと同時にログに書き込む．LSN はアトミック命令である fetch and add を用いて単調増加する整数として Leader ノード全体でユニークに採番される．

各 Worker スレッド内におけるロギング処理については，通常の Raft と同様に処理することができる．クラッシュリカバリの際に問題となるのは，Worker スレッドを跨いだログの順序性であるが，これは LSN によって保証される．LSN の採番からコミットまでの処理は並列に非同期で行われるため，合意形成の際に後から採番された LSN を持つエントリが先に採番された LSN を持つエントリを追い抜く可能性があるが，各エントリは S2PL によってロックを取得しており，エントリ間で競合しないことが保証されているため，リカバリ時に状態を復元する際に問題とならない．これにより，通常の Raft と S2PL を組み合わせた場合と同様に外部整合性を保証した分散トランザクション処理が可能となっている．

Follower への合意形成を行う前に，コマンドはタスクストアという領域に一時保存される．過半数の Follower からの RPC 応答により合意形成が完了した後に，タスクストアから取り出され，コマンドが実行されることで，ステートマシンが更新される．

Algorithm 2 Worker スレッドの処理

```
1: loop
2:    Dequeue the client command from Queue.
3:    Lock data item in client command.
4:    Append client command to logBuffer.
5:    if logBuffer.size > LogGrpSize or Timeout then
6:       Get log sequence number lsn.
7:       Add entries to log.
8:       Broadcasts the entry to all follower.
9:       Append commands in logBuffer to TaskStore.
10:   end if
11: end loop
```

3.3 コミット処理

コミット時の処理をアルゴリズム 3 に示す．コミット処理は Worker スレッドとは異なるスレッドで行う．Follower ノードは，Leader ノードの Worker スレッドから転送されたログエントリを LogReceiver スレッドで受信し，整合性に問題がなければ自身のログファイルに永続化した後，Leader ノードに対して ACK を返却する．ACK には，LSN が含まれる．Leader ノードは各 Follower から ACK を受信し，LSN に基づいてどのログに対する ACK であるかを識別する．過半数以上の Follower からの ACK を受信した場合，その LSN を持つログをコミットする．コミット時には，タスクストアから LSN に対応するコマンドを取得し，ステートマシンの更新を行う．コマンドの実行が完了次第，データアイテムのロックを解除し，Client に対してコミット完了の通知を送付する．

Algorithm 3 トランザクションのコミット処理

```
1: loop
2:    Receive the response from the follower node
3:    if Received RPC responses from the majority of followers.
      then
4:       pop the commands from TaskStore.
5:       for command in commands do
6:          Execute command.
7:          Unlock data item.
8:          Return the commit response to the client.
9:       end for
10:   end if
11: end loop
```

4 評　　価

ロギング処理の並列度に伴って性能がどのようにスケールするか，また，マルチコアのマシンリソースを有効に活用できているかを評価するために，評価実験を行なった．また，日本全国にノードを分散した場合をエミュレートした場合の性能についても測定を行なった．

4.1 実 験 環 境

実験は，Amazon Web Service(AWS) 上の Elastic Compute Cloud (EC2) 上に提案システムを構築することで行なった．実験に用いた EC2 の設定を表 1 に示す．

表 1　実験で用いた EC2 の設定

リージョン	東京
Availability Zone	全インスタンス同一
インスタンスタイプ	m5d.8xlarge
OS	RedHat Enterprise Linux 8
vCPU	32core
メモリ	128GiB
ストレージ	NVMe SSD 600GB

実験では，EC2 インスタンス 5 台に対してノードを起動する．5 台のノードの内 1 台が Leader ノードであり，残りの 4 台は Follower ノードである．Leader ノードに対し，複数の Client プロセスからトランザクションを多重に送付する．送付するトランザクションは送金を想定した決済トランザクションとする．なお，各トランザクションでは競合が起こらないようにキーの値を設定する．各インスタンスは同一のデータセンタに位置しており, 各ノード間のネットワーク遅延は 0 と仮定する．

4.2 実験方法

実験 1．Worker スレッドの増加に伴う性能評価

実験 1 では，ロギング処理の並列度を増やすことによる性能の変化を評価する．

複数の Client プロセスからトランザクションを多重に送付し，TPS と RTT を測定する．また，ロギング処理の並列化により，EC2 のマシンリソースを有効に使用できているかを評価するために，経過時間に伴う CPU 使用率も測定する．

TxGrpSize と LogGrpSize は実験ごとに同一の値を設定するものとし，これを単に GrpSize とする．GrpSize が 5,000, 10,000, 20,000 の場合のそれぞれについて，並列動作する Worker スレッド数を変化させて測定を行い，性能がどのように変化するかを評価する．また，集約ログ転送がない場合においても同様の実験を行い，性能の変化を比較する．

実験 2．距離遅延をエミュレートした場合の性能評価

実験 2 では，実際に日本全国にノードを分散配置した場合を想定し，性能を測定する．ノードの台数や実験環境は実験 1 のものと同一とする．各ノードは同一データセンタに配置されているため，距離に伴うネットワーク遅延をエミュレートするために Follower ノードで Leader ノードに対して応答を返す処理の中にノード間の距離に応じた遅延を sleep 関数により挿入する．ネットワーク遅延による遅延時間は，Broadband Acceleration Project [19] を参考に設定した．東京からの距離遅延を表 2 に示す．

表 2　東京からのネットワーク遅延仮定

都市	東京からの遅延時間
大阪	15ms
福岡	20ms
札幌	22ms
名古屋	8ms

4.3 結　　果

実験 1．Worker スレッドを増加させた場合の性能評価

集約ログ転送を行わない場合，GrpSize が 5,000 件，10,000 件，20,000 件の場合それぞれについて，並列起動する Worker スレッドの数を変化させた場合の TPS の変化を図 2 に示す．

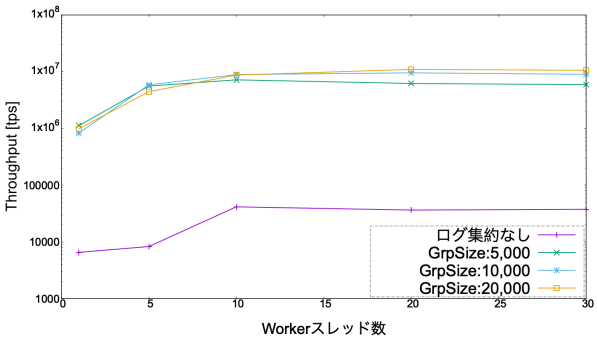

図 2　Worker スレッドの増加に伴う TPS の変化

GrpSize の数がいずれの場合においても，Worker スレッドの数を増やすことにより，処理性能がスケールしていることがわかる．Worker スレッド数が 10 の場合では，GrpSize が 10,000 の時が他の GrpSize の場合と比較して TPS が最も高く，8.8M tps 程度の TPS を記録した．

一方で，Worker スレッドが 30 の場合では GrpSize が 5,000 のものは性能がスケールせず，GrpSize が 20,000 の場合におよそ 10M tps と最も性能が高くなった．

次に，Worker スレッド数を変化させた場合の平均 RTT の変化を図 3 に示す．

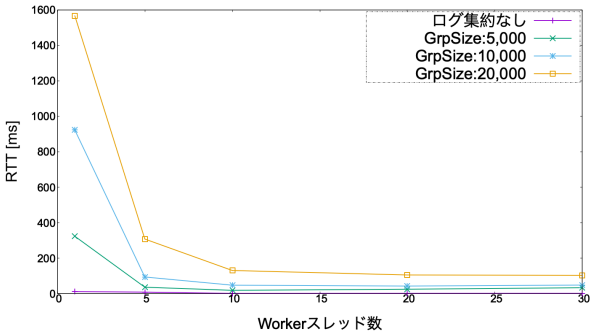

図 3　Worker スレッドの増加に伴う平均 RTT の変化

Worker スレッド数が 1 の時には，GrpSize が 20,000 の場合に平均 RTT が約 1,600 ミリ秒と，我々の目標数値を満たさない値となっている．Worker スレッドを 30 に増加させることで，RTT は約 100 ミリ秒となり，目標とする 400 ミリ秒を下回る結果となった．

次に，GrpSize が 20,000 の場合における Leader ノードの経過時間における CPU 使用率の変化を図 4 に示す．なお，CPU 使用率は 1 秒ごとに取得した 32 コアの平均値であり，グラフはトランザクションの処理開始地点を 0 秒としている．

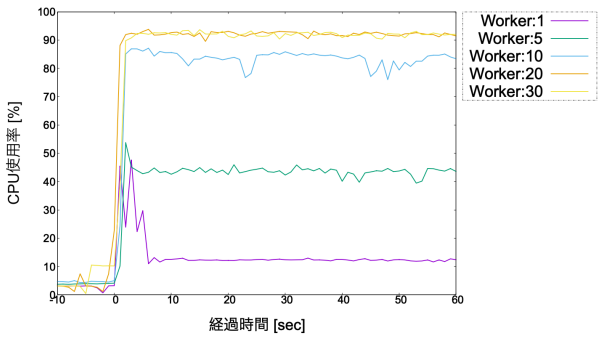

図 4　Worker スレッド数ごとの Leader ノードの CPU 使用率の変化

Worker スレッドの数が 1 の場合には 32 コアの平均使用率が 15%に満たない数値であったが，Worker スレッドを 5 つに増やすことによって，CPU 使用率は 40%以上まで使用されるようになった．Worker スレッドが 10 の場合では，CPU 使用率が 90%に近くなり，Worker スレッドが 20 以降は CPU 使用率が 90%以上まで達している．Worker スレッドが増えることで 32 コアの CPU リソースを有効に使えるようになっていること

がわかる．

実験 2．距離遅延をエミュレートした場合の性能評価

各 Follower ノードの RPC 応答の処理に sleep 関数を挿入し，実際の地理分散をエミュレートして検証を行なった結果を示す．距離遅延を挿入した場合の TPS の変化は図 5 にのようになった．

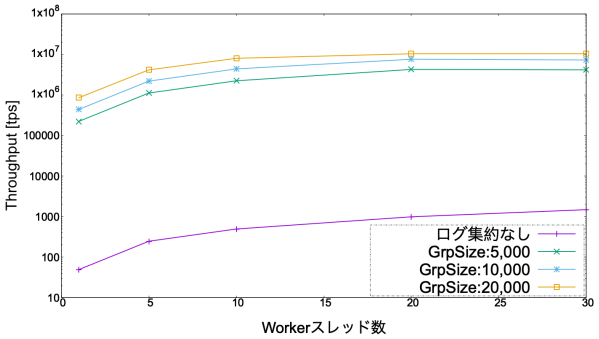

図 5　距離遅延を想定した場合の Worker スレッドによる TPS の変化

距離遅延挿入を行なった場合では，GrpSize が 20,000 で TPS が最大 10M tps 以上という結果になった．最大 TPS については距離遅延がない場合と比較して大きな変化はなかった一方で，ログ集約なしの場合では最大 TPS が 1,000tps 程度にまで大きく劣化した．GrpSize が 5,000 の場合では，Worker スレッド数が 1 の時の TPS が遅延なしと比べて 1/5 程度にまで劣化しており，距離遅延の影響を大きく受けている．しかし，Worker スレッド数を 30 まで増やした場合では，距離遅延なしと比較した性能劣化は 70%程度まで抑えられており，Worker スレッドを増やすことで距離遅延による影響が抑えられていることが分かる．

距離遅延を挿入した場合のトランザクション 1 件あたりの平均 RTT の変化を図 5 に示す．Worker スレッドが 1 の時には距離遅延の影響を大きく受けており，GrpSize がいずれの場合でも平均 RTT が 1,600 ミリ秒を超える結果となった．特にログ集約なしの場合は距離遅延なしの場合の平均 RTT が数ミリから数十ミリであったのに対して大幅に RTT が長くなっている．Worker スレッドを増やすことにより処理時間が短縮化され，Worker スレッド 30 の場合では，最も RTT の長い GrpSize が 20,000 の場合でも約 110 ミリ秒と，目標性能を達成することができた．

4.4　考　　察

実験 1 の結果では，Worker スレッド数を増やしロギング処理を並列化することにより，Leader ノード全体でみたコマンドの処理能力が上がり，TPS と平均 RTT が大きく向上することが示された．CPU 使用率についても Worker スレッド数を増やすことで 32 コアある CPU を 90%まで使うことができるようになっており，先行研究で課題とした，マシンリソースが有効に使えていない問題を改善できている．GrpSize が 5,000 の場合には，GrpSize が 10,000，20,000 のものと比較して Worker スレッドの増加によるスケールが鈍化する傾向が見

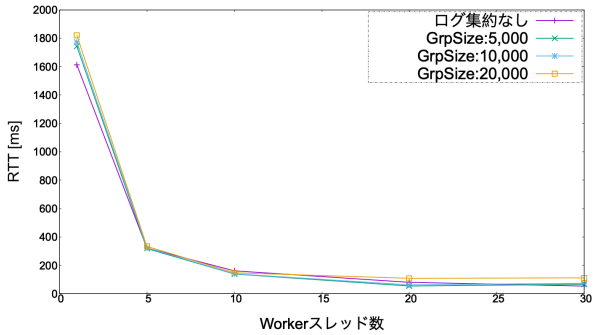

図 6　距離遅延を想定した場合の Worker スレッドによる平均 RTT の変化

られた．GrpSize が増加することで，Client から 1 度に送付されるコマンドの総数が大きくなるため，Leader ノードにかかる負荷が大きくなる．GrpSize が 5,000 の場合での負荷では，Worker スレッドの性能を最大まで活用できず，TPS がスケールしなかったことが考えられる．一方で GrpSize を増やすことでコマンド 1 件あたりの RTT は長くなるため，想定されるリクエスト数やマシンリソース等によって，Worker スレッド数や GrpSize の値を設定することが必要であると考えられる．

距離遅延をエミュレートした実験 2 では，Worker スレッドを増加させることで，距離遅延なしと同程度の性能を記録することができた．Worker スレッドが 1 の場合には，ロギング処理や Follower との通信が逐次処理され，一回の通信に，平均 16 ミリ秒程度の遅延が入るため，平均 RTT に大きな影響を受ける．Worker スレッド数を増やすことで，Follower との通信についても並列化されるため，距離遅延による影響が小さくなっていると考えられる．Worker スレッドを 30 まで増やすことで，最も RTT のかかる GrpSize が 20,000 の場合でも，約 110 ミリ秒と距離遅延がない場合と遜色ない処理速度となった．

一方で，今回のプロトタイプでは送金処理のみを想定してトランザクション処理を実装し，キーの競合しない入力データを用いて理想的な環境下でピーク性能を測定した．実システムとして稼働させることを想定して性能を測定した場合では，実装やワークロードによっては性能が劣化することも予測されるため，今後更なる検証が必要となる．

5　関 連 研 究

地理分散トランザクションを対象にして外部整合性を保証するアプローチにはシングルマスタ方式 [1] [2] [3] とマルチマスタ方式 [4] [5] [6] がある．シングルマスタ方式は Client との接続を許すマスタがただ一つであり，マルチマスタ方式はそれが複数である．前者の一部 [1] [2] は既にプロダクトになっており安定稼働実績を有するが，性能は TPC-C で 28,000tps 程度に留まる [1]．後者は multi-partition 方式で有用であり，skew が低い場合に高性能を示すが，依然として研究段階であり安定稼働を有するプロダクトは存在しない．前者に属する Spanner は，Raft と類似の合意形成アルゴリズムである Paxos をベー

スとしたプロトコルを採用しており，高可用性と水平スケーラビリティを特徴としている．CockroachDB は Spanner の設計を踏襲したシステムであり分散合意プロトコル Raft [7] を用いて分散合意を形成する．Carousel はトランザクションの中でも 2-round Fixed-set Interactive (2FI) を対象にし，Fast Paxos に基づき 2 phase commit を高速化し，TAPIR [8] に対する優位性を示した．本研究はこれらの従来研究とは異なりロギング処理に着眼し，複数ログの集約やロギングの並列化を行なっている．

Paxos ベースの分散合意プロトコルとして APUS [20] が挙げられる．APUS は Remote Direct Memory Access(RDMA) を用いた通信と複数命令の一括処理により高速化を行なっている．複数命令を一括で処理するというアプローチは本研究と同じである．本研究では RDMA ではなくマルチスレッドでロギングを並列化することにより高速化を図っている．

Chiller [21] では，データ競合を回避するためのパーティショニングによる高速な分散トランザクション手法を提案している．高速 RDMA 環境下では，ネットワーク遅延はボトルネックにならず，データ競合こそが処理を妨げるため，競合を回避するようデータの持ち方とコマンドの実行方法を工夫するというアプローチを取っている．

6 結　　論

本研究では，分散合意アルゴリズム Raft をベースとした地理分散データベースに対し，並列ロギング手法 P-WAL を適用することで効率的な分散トランザクション処理を実現する手法について提案した．複数の Worker スレッドを用い，それぞれが独立してロギングを行うことで，TPS とコマンド 1 件あたりの平均 RTT の双方において，大幅にスケールすることを示した．並列ロギングにより，これまで逐次処理されていた Raft のロギング処理が並列化され，CPU のリソースを有効活用できるようになり，また，Follower ノードとの通信についても並列化されることで，距離遅延による影響を低減できることがわかった．集約ログ転送法と並列ロギングを組み合わせた実験では，日本全国 5 都市への地理分散を想定した場合において，TPS が約 10M tps，平均 RTT が 110 ミリ秒と，目標数値を大きく満たす処理性能を記録することができた．しかし，検証で用いたコマンドは競合しないことを前提としたトランザクションであり，実環境ではデータの競合による性能劣化があると考えられる．今後は，データアイテムの競合を想定した場合の性能検証や，ノード故障時などを想定した場合の性能検証等を行なっていく必要がある．

文　　献

[1] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle adn Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *OSDI*, pages 251–264, 2012.

[2] Cockroachdb. `https://www.cockroachlabs.com/docs/stable/performance.html`.

[3] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In *SIGMOD*, pages 231–243, 2018.

[4] Kun Ren, Dennis Li, and Daniel J. Abadi. Slog: Serializable, low-latency, geo-replicated transactions. *PVLDB*, 12(11):1747–1761, 2019.

[5] Samuel Madden Yi Lu, Xiangyao Yu. Star: Scaling transactions through asymmetric replication. *PVLDB*, 12(11):1316–1329, 2019.

[6] Hua Fan and Wojciech Golab. Ocean vista: Gossip-based visibility control for speedy geo-distributed transactions. *PVLDB*, 12(11):1471–1484, 2019.

[7] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–319, 2014.

[8] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *SOSP*, pages 263–278, 2015.

[9] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. An analysis of concurrency control protocols for in-memory database with ccbench. *PVLDB*, 13(13):3531–3544, 2020.

[10] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[11] 秋田佳紀, 小倉拓人, 宮澤勇貴, and 川島 英之. 集約ログ転送法を用いた外部整合的トランザクション機構の評価. In **情報処理学会研究報告** *2020-OS-150*, 2020.

[12] 堀江悠樹, 梶原顕伍, 川島英之, and 建部修見. ログ転送グループ化による外部整合的トランザクション処理の高性能化. In **情報処理学会研究報告** *2020-OS-148*, 2020.

[13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

[14] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, 2014.

[15] 神谷孝明, 川島英之, and 建部 修見. 集約ログ転送法を用いた外部整合的トランザクション機構の評価. **情報処理学会論文誌データベース** *(TOD)*, 2017.

[16] Yasuhiro Nakamura, Hideyuki Kawashima, and Osamu Tatebe. Integrating tictoc with parallel logging. In *CANDAR CSA Workshop*, pages 1 – 7, 2017.

[17] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst*, 2(16):133–169, 1998.

[18] Brian M. Oki and Barbara Liskov. Viewstamped replication, a general primary copy. In *PODC*, pages 8–17, 1988.

[19] Broadband acceleration project. `https://bbzero.jp/background`.

[20] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *SoCC*, pages 94–107, 2017.

[21] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *SIGMOD*, pages 511—-526, 2018.