Vol.2022-ARC-248 No.35
Vol.2022-SLDM-198 No.35
Vol.2022-EMB-59 No.35
2022/3/10

並行性制御法による ROS TF の高品質化

荻原 湧志[1,a)]　萬 礼応[2]　大矢 晃久[2]　川島 英之[1]

概要：Robot Operating System(ROS) はロボットソフトウェア用のミドルウェアソフトプラットフォームであり，近年多くの研究用ロボットで用いられている．TF ライブラリは ROS で頻繁に使用されるパッケージであり，各座標系間の変換を有向木構造として管理し，効率的な座標変換情報の登録，座標変換の計算を可能にした．この有向木構造には非効率な並行性制御によりアクセスが完全に逐次化され，アクセスするスレッドが増えるに従ってパフォーマンスが低下する問題，及び座標変換の計算時にその仕様によって最新のデータを参照しないという問題があることがわかった．そこで，我々はデータベースの並行性制御法における 細粒度ロッキング法，及び 2PL を応用することにより，これら問題を解決した．提案手法では既存手法と比べ最大 257 倍のスループット，最大 282 倍高速化したレイテンシ，最大 132 倍のデータ鮮度となることを示した．

Make ROS TF high quality in concurrency control method

1. はじめに

1.1 TF ライブラリ

TF ライブラリ [1] は ROS[2] 上で動作し，各座標系間の変換を有向木構造として一元管理し，効率的な座標系間の変換情報の登録，座標系間の変換の計算を可能にした [1]．**図 1** 左は部屋の中にロボットと，ロボットから観測できる二つの物体がある様子を表す．図中にてロボットは円形，物体は星形で表現され，ロボットが向いている方向は円の中心から円の弧へつながる直線の方向で表される．直角に交わる二つの矢印は座標系を表し，交点が座標系の原点，二つの矢印が反時計回りにそれぞれ X・Y 軸を表す．ここでは地図座標系，ロボット座標系，そして二つの物体の座標系が表記されている．

図 1 左の各座標系間の位置関係を表す木構造は図 1 右で表現できる．ノードが各座標系を表し，エッジは子ノードから親ノードへの変換データが存在することを表す．

ノードは TF ではフレームと呼ばれ，フレーム中の文字列は各座標系に対応するフレーム名である．図 1 右では地図座標系のフレーム名は map，ロボット座標系のフレーム名は robot，物体 1 の座標系のフレーム名は object1 となる．親フレームへ張られたエッジは子フレームから親フレームへポインタが貼られていることを表し，子フレームから親フレームを辿ることができる．しかしながら，親フレームから子フレームを辿ることはできない．このため，map から object1 への座標変換を計算するには object1 から map への座標変換の計算をし，その逆変換を取る必要がある．

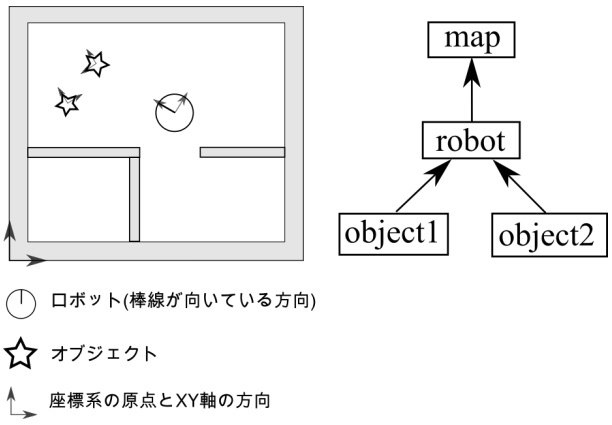

図 1　部屋の中のロボット，それに対応する TF の木構造

各フレーム間の座標変換情報はそれぞれ異なるタイミングで登録される．これに対処するため，TF では各フレーム間の座標変換情報を 10 秒間保存する．図 1 において各フレーム間の座標変換情報が登録されたタイミングを表す

[1] 慶應義塾大学環境情報学部
Faculty of Environment and Information Studies, Keio University
[2] 筑波大学システム情報系情報工学域
Department of Computer Science, University of Tsukuba
[a)] yushiogiwara@keio.jp

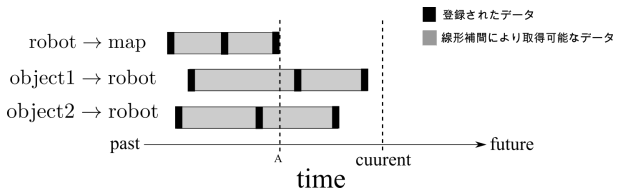

図 2 図 1 における位置関係登録のタイムライン

のが**図 2** である．横軸は時間軸を表し，左側が過去，右側が最新の時刻を表す．黒色のセルはデータがその時刻に登録されたことを表す．時刻 A では robot から map への座標変換の情報が得られるが，object1 から robot への座標変換の情報は時刻 A には存在しない．そこで，TF では前後のデータから線形補間を行うことにより該当する時刻の座標変換データを計算する．つまり，TF はある時刻の座標変換データが保存されているか線形補間で取得できる時に，その時刻の座標変換データを提供できる，とみなす．灰色の領域は線形補間により座標変換データが提供可能な時間領域を表す．

図 1 における位置関係登録のタイムラインが図 2 のようになっているとき，TF では object1 から map への最新の位置関係は次のように計算する．

まず，object1 から map への各エッジを確認する．ここでは object1 から map への各エッジは object1→robot, robot→map であることがわかる．

次に，どのエッジにおいてもなるべく最新の座標変換を提供できる時刻を確認する．図 2 を確認すると，object1→robot，robot→map において最新の座標変換情報が登録された時刻が最も古いのは robot→map である．このため，時刻 A がここでは要件を満たす．

最後に，時刻 A での各エッジのデータを取得し，それらを掛け合わせる．robot→map については登録されたデータを使い，object1→robot については線形補間によってデータを取得する．

1.2 研究課題

上述したように TF はロボットシステム内部の座標系間の位置関係を一元管理する．しかしながら，これには以下のような問題点が挙げられる．

問題 1：ジャイアント・ロック

TF の木構造には複数のスレッドが同時にアクセスするため並行性制御が必要となるが，既存の TF では一つのスレッドが木構造にアクセスしている際は他のスレッドは木構造にアクセスできないアルゴリズムとなっている．複数スレッドが木構造の別々の部分にアクセスするケース，及び複数スレッドが木構造の同じ部分のアクセスしているが全て読み込みアクセスのケースなど，排他制御が必要ではないケースにおいてもアクセスが完全に逐次化されている．これにより，マルチコアが常識となっている現代ではスループットやレイテンシに問題が生じる可能性がある．

問題 2：データの鮮度

上述のように，TF のフレーム間の座標変換計算インターフェースはある時刻の座標変換データが保存されているか線形補間で取得できる時に，その時刻の座標変換データを提供できるという仕様のため，最新のデータを使わない可能性がある．同時刻のデータを元に座標変換を行うためデータの時刻同期性はあるが，最新の座標変換データを使わないためデータの鮮度は失われる．これにより，ロボットの制御や自己位置推定に問題が生じる可能性がある．現在，TF ライブラリには最新の座標変換データをもとにフレーム間の座標変換計算をするインターフェイスは無い．

また，この仕様により座標系間の位置関係があまり変わらない場合についても頻繁にデータを登録する必要があり，無駄な処理が発生する．

問題 3：データの一貫性

問題 2 の解決策として，最新の座標変換データをもとにフレーム間の座標変換計算をするインターフェイスを提供するだけでは不十分である．これは，複数の座標変換データを登録している途中に最新の座標変換データをもとにフレーム間の座標変換計算をすると，ユーザーが期待するデータの一貫性がなくなる可能性があるからである．

1.3 貢献

問題 1 については，並行性制御法における細粒度ロッキング法を適用することによって解決した．細粒度ロッキング法は，並行性制御においてロックするデータの単位をなるべく小さくし，並行性を向上させる手法である．これにより，既存手法ではジャイアントロックによって TF へのアクセスは完全に逐次化されていたが，提案手法では細粒度ロックによってフレーム単位でのロックとなり，複数のフレームに並行にアクセスが可能になった．細粒度ロックを実装した場合のスループットは最大で 11,574,200tps，レイテンシは高々 0.7ms となった．また既存手法と比べ細粒度ロックを実装した場合はスループットは最大 243 倍，レイテンシは最大 172 倍高速化した．

問題 2，3 については，データベースの並行性制御法における 2PL を適用することにより，複数の座標変換の最新のデータを atomic に取得するインターフェース（lookupLatestTransformXact），及び複数の座標変換の最新のデータを atomic に更新するインターフェース（setTransformsXact）を提供することによって解決した．既存手法で提供されているインターフェイスではジャイアントロックにより TF へのアクセスが逐次化され，また§ 1.1 にて説明したように時刻の同期をとるという仕様のために過去の

Vol.2022-ARC-248 No.35
Vol.2022-SLDM-198 No.35
Vol.2022-EMB-59 No.35
2022/3/10

データを参照しデータの鮮度が落ちるという問題があったが，lookupLatestTransformXact と setTransformsXact を使うことによりこれは解決できた．既存手法と比べ，lookupLatestTransformXact と setTransformsXact を使った場合にはデータの鮮度は最大で 132 倍となった．また，既存手法と比べ最大 257 倍のスループット，最大 282 倍高速化したレイテンシとなり，このインターフェイスはスループットとレイテンシにおいても既存手法より優れていることを示した．既存の TF ライブラリ [17] に 1236 行分の変更を加え，提案手法を実装したコードを [19] に公開した．

1.4 構成

本論文の構成は次の通りである．第二節では関連研究について述べる．第三節では既存の TF の木構造とその問題点について述べる．第四節では提案手法である木構造への再粒度ロックの導入とデータの鮮度，データの一貫性のためのインターフェイスの提供について述べる．第五節では提案手法の評価結果を述べる．第六節では本研究の結論と今後の課題について述べる．

2. 関連研究

2.1 データベース分野におけるロボット研究

筆者の知る限り，データベース分野ではロボットを対象にした研究論文はトップ会議，トップジャーナルでは発表されたことがない．他方，産業界における珍しい例としては GAIA platform [11] が挙げられる．GAIA platform はリレーショナルデータベースと，そのデータベースに変更が加えられた時の処理を C++で宣言的に記述できる仕組みを組み合わせることにより，イベントドリブンなフレームワークでロボットや自動運転システムを構築するものである．このデータベースアクセスはトランザクションを用いて実行される．GAIA チームには snapshot isolation 提案者も含まれており，そのトランザクションアーキテクチャには一定の頑健性があることが期待される．データベース分野における高速並行性研究におけるトランザクション処理システムとして 2PL [4]，Silo [6]，MOCC [5]，Cicada [7] が挙げられる．

2.2 ロボット分野におけるデータベース研究

TF ライブラリのようにデータを時系列的に管理するライブラリとして SSM [3] が挙げられる．SSM では各種センサデータを共有メモリ上のリングバッファで管理することにより，時刻の同期を取れたデータを高速に取得することができる．産業用途にも利用可能にするため，ROS の次世代バージョンである ROS2 [12] の開発が進んでいるが，並行性制御アルゴリズムは ROS から変わっておらず，本研究のようなアプローチはない．

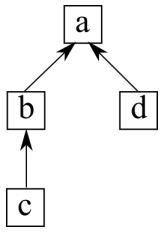

図 3 TF の木構造

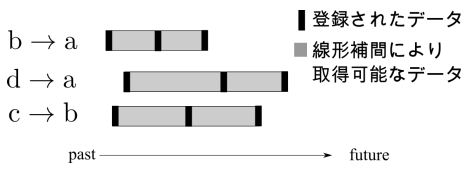

図 4 タイムライン

3. 既存の TF の木構造とその問題点

3.1 構造

TF ライブラリでは**図 3** のように各座標系間の位置関係を木構造で管理する．ノードが各座標系を表し，エッジは子ノードから親ノードへの座標変換データが存在し，また親ノードへポインタが貼られていることを表す．このため子ノードから親ノードへ辿ることはできるが，親ノードから子ノードを辿ることはできない．ノードはフレームと呼ばれ，フレーム内の文字列は各座標系に対応するフレーム名である．

各フレーム間の座標変換情報は，フレーム間のエッジに 10 秒間保存される．このため，各フレーム間の座標変換情報が登録された時刻を**図 4** のようなタイムラインで表現できる．黒のセルは登録されたデータを表し，灰色のセルは線形補間により座標変換データが取得可能な時刻を表す．横軸が時間軸を表し，左側が過去，右側が最新の時刻を表す．

3.2 TF の木構造のインターフェイス

TF ライブラリの木構造のインターフェイスは [9] に公開されており，この中でも頻繁に使用されるのが *lookupTransform* と *setTransform* である．TF の木構造へのアクセスは主に，高頻度で *lookupTransform* のみ呼び出す読み込み専用スレッドと，高頻度で *setTransform* のみ呼び出す書き込み専用スレッドで構成される．本研究ではこの二つのインターフェイスの改善を行う．

3.3 lookupTransform

二つのフレーム間の座標変換情報を取得するには *lookupTransform* を使う．この実装は [10] に公開されている．

lookupTransform は source フレームから target フレームへの最新の座標変換を計算し，これは source フレームから target フレームへのパスを辿ることにより計算される．次に説明するように，*lookupTransform* ではこの source フレームから target フレームへのパスを 2 度辿る．1 度目のアクセスでは，source フレームから target フレームへのパス上の，全てのフレームにおいて座標変換を提供できる最新の時刻を検索する．各フレーム間の座標変換情報は 10 秒間保存され，これは両端キューによって実装される．両

端キューの先頭にはそのフレームとその親フレームの最新の座標変換情報と時刻が記録されている．このため，次の手続きで source フレームから target フレームへのパス上の，全てのフレームにおいて座標変換を提供できる最新の時刻を検索できる．

(1) source フレームからルートフレームへのパス上の，全てのフレームにおいて座標変換を提供できる最新の時刻を検索する．各フレーム内の両端キューの先頭のデータの時刻を取得し，これらの時刻の中で最小のものが該当する時刻となる．
(2) target フレームからルートフレームへのパス上の，全てのフレームにおいて座標変換を提供できる最新の時刻を検索する．上述と同じ方法で取得できる．
(3) 得られた二つの時刻のうち小さい方が，source フレームから target フレームへのパス上の，全てのフレームにおいて座標変換を提供できる最新の時刻となる．

この 1 度目のアクセスは，実際には *getLatestCommonTime* インターフェイス内で行われ，*lookupTransform* では *getLatestCommonTime* をサブルーチンとして呼び出す．

2 度目のアクセスでは，1 度目のアクセスで取得した時刻 *time* をもとに source フレームから target フレームへの座標変換を計算する．これは次の手続きとなる．

(1) source フレームからルートフレームへの時刻 *time* における座標変換を計算する．これは，source フレームからルートフレームへのパス上の，全てのフレームにおける時刻 *time* での座標変換を掛け合わせることにより計算できる．各フレーム内の両端キュー内のデータのうち時刻 *time* の前後のデータを取得し，それらから線形補完を行い時刻 *time* での座標変換を取得できる．
(2) target フレームからルートフレームへの時刻 *time* における座標変換を，上述と同じようにして計算する．
(3) source フレームからルートフレームへの座標変換とルートフレームから target フレームへの座標変換を掛け合わせて，source フレームから target フレームへの最新の座標変換を計算できる．ルートフレームから target フレームへの座標変換は，target フレームからルートフレームへの座標変換の逆変換を取れば良い．

3.4 setTransform

二つのフレーム間の座標変換情報を更新するには *setTransform* を使う．このインターフェイスを呼び出すことにより新しい座標変換情報がタイムラインに追加される．上述したように，各フレーム内の両端キューでタイムラインを表現し，追加された座標変化情報はこの両端キーの先頭に追加される．座標変換情報は両端キューに 10 秒間保存され，10 秒以前の座標変換情報は両端キューの末尾から順にポップされる．

3.5 問題点

問題 1: ジャイアント・ロック

上述のように，TF ライブラリの木構造で主に使われるインターフェイスは *lookupTransform* と *setTransform* である．これらは複数のスレッドからアクセスされるので並行性制御を行う必要があり，TF ライブラリでは mutex オブジェクトを用いて木構造全体を保護している．このため，一つのスレッドが木構造にアクセスしている際は他のスレッドは木構造にアクセスできない．複数スレッドが木構造の別々の部分にアクセスするケース，及び複数スレッドが木構造の同じ部分のアクセスしているが全て読み込みアクセスのケースなど，排他制御が必要ではないケースにおいてもアクセスが逐次化される．

問題 2: データの鮮度

lookupTransform は二つのフレーム間の座標変換の計算において，フレーム間のエッジの全てにおいて座標変換データを提供できる時刻についての座標変換を計算するという仕様のため，最新の座標変換データが使われなくなるという問題がある．同時刻のデータを元に座標変換を計算するためデータの同期性はあるが，最新の座標変換データを使わないためデータの鮮度は失われる．これにより，ロボットの制御や自己位置推定に問題が生じる可能性がある．現在，TF ライブラリには最新の座標変換データをもとにフレーム間の座標変換計算をするインターフェイスは無い．

また，次に説明するようにこの仕様によって座標系間の位置関係があまり変わらない場合についても頻繁にデータを登録する必要があり，無駄な処理が発生する．

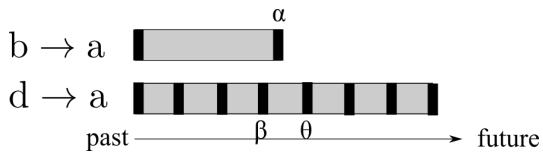

図 5 不必要な更新が必要な例

図 5 において、b→a はあまり座標系間の位置関係が変わらないために座標変換はあまり更新されないが，d→a の座標変換は頻繁に更新されるケースについて考える．座標変換の計算において b→a と d→a のデータを用いる場合，既存の TF では過去の鮮度の低い β と θ から座標変換の計算をしなくてはならない，これを避けるため，既存の TF ではあまり座標系間の位置関係が変わらない b→a においても，一定周期で同じ座標変換情報を登録する必要がある．このように，既存の TF では座標変換情報が変わらないにもかかわらず一定周期で同じ座標変換情報を登録する必要があり，余計な負荷がかかっている．

Vol.2022-ARC-248 No.35
Vol.2022-SLDM-198 No.35
Vol.2022-EMB-59 No.35
2022/3/10

問題 3: データの一貫性

問題 2 の解決策として，最新の座標変換データをもとにフレーム間の座標変換計算をするインターフェイスを提供するだけでは不十分である．次に説明するように，複数の座標変換データを登録している途中に最新の座標変換データをもとにフレーム間の座標変換計算をすると，ユーザーが期待するデータの一貫性がなくなる可能性があるからである．

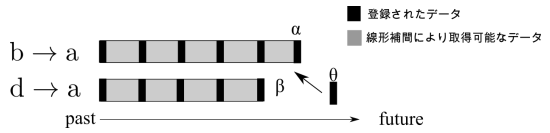

図 6 同時に座標変換が登録されるケース

図 6 は木構造が図 3 の時にフレーム b からフレーム a への座標変換と，フレーム d からフレーム a への座標変換が同時刻に登録されるケースでのタイムラインを表す．b→a と d→a のデータを用いてフレーム b からフレーム c への座標変換を計算する際，ユーザーは b→a と d→a のデータについては同時刻のものを使うことを期待する．しかしながら，最新の座標変換データをもとにフレーム間の座標変換計算をするインターフェイスを使うとユーザーの期待に反して図 6 のように θ がまだ登録されていない中間状態のタイムラインを観測し，α と β を元に座標変換してしまうことがある．従来の *lookupTransform* ではフレーム間のエッジの全てにおいて座標変換データを提供できる時刻についての座標変換を計算するという仕様のため，このような問題は発生しない．

4. 提案手法

本研究では，データベースの並行性制御法における細粒度ロッキング法及び 2PL を適用し，これらの問題を解決する．

4.1 細粒度ロックの導入

本研究ではデータベースの並行性制御法における細粒度ロッキング法を適用する．細粒度ロッキング法ではアクセスするデータにのみロックを確保し，さらにロックの種類を読み込みロックと書き込みロックに分ける．アルゴリズム 1 は細粒度ロックを実装した *lookupTransform* の疑似コードを表しており， 6〜8 行目のように，フレーム単位での読み込みロックの確保・解放により細粒度ロックを実装している．*getLatestCommonTime* についても，同じようにして細粒度ロックを実装した．アルゴリズム 2 は細粒度ロックを実装した *setTransform* の疑似コードを表しており，細粒度ロックを実装した *lookupTransform* と同じようにして細粒度ロックを実装している．

Algorithm 1 細粒度ロックを実装した lookupTransform

```
 1: function LOOKUPTRANSFORM(target, source)   ▷ フレーム source からフレーム target への最新の座標変換を計算する
 2:     time = getLatestCommonTime(target, source)
 3:     source_trans = I                        ▷ I は座標変換の単位元
 4:     frame = source   ▷ frame はエッジを辿る時に見るフレーム
 5:     while frame ≠ root do                   ▷ source から root まで辿る
 6:         frame.rLock()                       ▷ 読み込みロックを確保
 7:         (trans, parent) = frame.getTransAndParent(time)
 8:         frame.rUnlock()                     ▷ 読み込みロックを解放
 9:         source_trans *= trans               ▷ 座標変換の掛け合わせ
10:         frame = parent
11:     end while
12:     frame = target
13:     target_trans = I
14:     while frame ≠ root do                   ▷ target から root まで辿る
15:         frame.rLock()
16:         (tarns, parent) = frame.getTransAndParent(time)
17:         frame.rUnlock()
18:         target_trans *= trans
19:         frame = parent
20:     end while
21:     return source_trans * (target_trans)^-1
22: end function
```

Algorithm 2 細粒度ロックを実装した setTransform

```
1: procedure SETTRANSFORM(transform)
2:     frame = getFrame(transform.child_frame_id)
3:     frame.wLock()                           ▷ 書き込みロックを確保
4:     frame.insertData(transform)
5:     frame.wUnlock()                         ▷ 書き込みロックを解放
6: end procedure
```

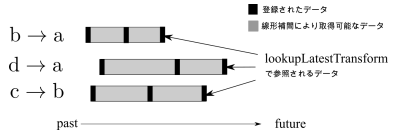

図 7 lookupLatestTransform で取得するデータ

4.2 並行性制御法の導入

上述した問題 2，3 については，複数の座標変換のデータを atomic に取得するインターフェース（*lookupLatestTransformXact*），及び複数の座標変換の最新のデータを atomic に更新するインターフェース（*setTransformsXact*）を提供して解決する．

4.2.1 データの鮮度の向上

問題 2 を解決するために，二つのフレーム間の座標変換に線形補間を行わずにフレーム間のエッジの最新の座標変換データを使うインターフェイスとして *lookupLatestTransform* を提案する．

lookupLatestTransform にて取得する座標変換データを，**図 7** に示す．

lookupLatestTransform により，§ 3.5 で説明した問題も解決できる．これは，*lookupLatestTransform* は *lookupTransform* とは異なり，最新の座標変換のみを見るため必

Vol.2022-ARC-248 No.35
Vol.2022-SLDM-198 No.35
Vol.2022-EMB-59 No.35
2022/3/10

Algorithm 3 lookupLatestTransform

```
 1: function lookupLatestTransform(target, source)
 2:     source_trans = I
 3:     frame = source
 4:     while frame ≠ root do
 5:         frame.rLock()
 6:         (trans, parent) = frame.getLatestTransAndParent()
 7:         frame.rUnlock()
 8:         source_trans *= trans
 9:         frame = parent
10:     end while
11:     frame = target
12:     target_trans = I
13:     while frame ≠ root do
14:         frame.rLock()
15:         (tarns, parent) = frame.getTransAndParent(time)
16:         frame.rUnlock()
17:         target_trans *= trans
18:         frame = parent
19:     end while
20:     return source_trans * (target_trans)^-1
21: end function
```

要な時にのみ座標変換の更新をすればよく，§ 3.5 で説明した余計な負荷がかかることはなくなるからである．

lookupLatestTransform をアルゴリズム 3 にて示す．

4.2.2 データの一貫性の確保

lookupLatestTransform により，最新の座標変換データをもとにしたフレーム間の座標変換が計算できる．しかし，§ 3.5 で説明したように複数のデータを読むとき，それらの間の一貫性が失われる事態が生じ得る．

そこで, 複数の座標変換を 2PL によって atomic に木構造に登録する *setTransformsXact* と，*lookupLatestTransform* を 2PL を使うように変更した *lookupLatestTransformXact* を提案する．*lookupLatestTransformXact*，*setTransformsXact* の擬似コードをそれぞれアルゴリズム 4，5 に示す．*lookupLatestTransformXact* では読み込みロックを取った後は 7 行目のように読み込みロックのリストに追加し、21 行目のように座標変換を取得した後にリスト内の全てのフレームのロックを解放することにより，2PL を実装している．*setTransformsXact* でも同じようにして 2PL を実装し，また Deadlock が発生しないように 9〜15 行目のようにして NoWait[16] を実装する．

5. 評価

以下，既存手法は old，細粒度ロックを実装した *lookupTransform*・*setTransforms* を snapshot, 2PL を用いて複数のデータを atomic に取得・更新できる *lookupLatestTransformXact*・*setTransformsXact* を latest と表記する．

5.1 実装

TF ライブラリの実装は Github のリポジトリ [17] で公開され，ブランチ毎に各 ROS のディストリビューション向けの実装がされている．このリポジトリのデフォルトブランチは melodic-devel であるが，TF の木構造の並行性

Algorithm 4 lookupLatestTransformXact

```
 1: function lookupLatestTransformXact(target, source)
 2:     rlock_list = [ ]
 3:     source_trans = I
 4:     frame = source
 5:     while frame ≠ root do
 6:         frame.rLock()
 7:         rlock_list.push_back(frame) ▷ 読み込みロックのリストに追加
 8:         (trans, parent) = frame.getLatestTransAndParent()
 9:         source_trans *= trans
10:         frame = parent
11:     end while
12:     frame = target
13:     target_trans = I
14:     while frame ≠ root do
15:         frame.rLock()
16:         rlock_list.push_back(frame)    ▷ 読み込みロックのリストに追加
17:         (tarns, parent) = frame.getTransAndParent(time)
18:         target_trans *= trans
19:         frame = parent
20:     end while
21:     for f in rlock_list do    ▷ リスト内の全ての要素のロックを解放
22:         f.rUnlock()
23:     end for
24:     return source_trans * (target_trans)^-1
25: end function
```

Algorithm 5 setTransformsXact

```
 1: procedure setTransformsXact(transforms)
 2:     wlock_list = [ ]
 3:     for trans in transforms do
 4:         frame = getFrame(trans.chzild_frame_id)
 5:         lock_success = frame.tryWLock() ▷ 書き込みロックの確保を試みる
 6:         if lock_success then
 7:             wlock_list.push_back(frame)
 8:         else
 9:             for f in wlock_list do    ▷ リスト内の全ての要素のロックを解放
10:                 f.wUnlock()
11:             end for
12:             wait for backoff before retry
13:             goto 2                              ▷ 処理をやり直す
14:         end if
15:     end for
16:     for trans in transforms do    ▷ Dirty read を避けるため，growing phase が終わってから書き込み
17:         frame = getFrame(trans.child_frame_id)
18:         frame.insertData(trans)
19:     end for
20:     for f in wlock_list do    ▷ リスト内の全ての要素のロックを解放
21:         f.wUnlock()
22:     end for
23: end procedure
```

制御アルゴリズムはどのブランチでも変わらない．また，ROS2 向けの TF ライブラリの実装は [18] で公開されているが，こちらも木構造の並行性制御アルゴリズムは ROS 向けのものと変わらない．

このため，本研究では Ubuntu18.04 を搭載したマシンに ROS Melodic Morenia をインストールし，TF ライブラリの Github のリポジトリ [17] の melodic-devel ブランチの実装を変更して実験を行った．C++言語で実装をし，1236 行分の変更を行った．この実装は [19] で公開されている．

5.2 実験環境

実験には Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz を 4 つ搭載したサーバを利用する．それぞれのコアは 32KB private L1d キャッシュ，1024KB private L2 キャッシュを持つ．単一プロセッサの 28 コアは 39MB L3 キャッシュを共有し，ハイパー・スレッディングを有効化している．トータルキャッシュサイズはおよそ 160MB である．メモリは DDR4-2666 が 48 個接続されており，一つあたりのサイズは 32GB，全体のサイズは 1.5 TB である．全ての実験において，実行時間は 60 秒という安定的な結果が得られる時間を選択した．

5.3 ワークロード

実験のワークロードは, 関節数が多いヘビ型ロボットの関節情報を TF に登録することを想定し, フレーム間の座標変換情報が一直線に与えられた構造に複数のスレッドからアクセスし計測を行う．*lookupTransform* のみを複数回呼び出すスレッド（読み込み専用スレッド），及び *setTransform* のみを複数回呼び出すスレッド（書き込み専用スレッド）をそれぞれ複数立ち上げ計測を行う．

実験においては以下のパラメータが存在する．

(1) joint: フレーム数．すなわち蛇型ロボットにおいては関節数を表す．
(2) read_ratio: 合計スレッド数のうち，読み込み専用スレッドの割合．
(3) read_len: 読み込み専用スレッドにて一回の *lookupTransform* 呼び出しで読みこむフレームの数．joint 個のフレームのうち一様分布を元に i 番目のフレームが選択され，そこから i+read_len 番目のフレームまでの座標変換が計算される．
(4) write_len: 書き込み専用スレッドにて一回の操作で座標変換情報を更新するフレームの数．joint 個のフレームのうち一様分布を元に i 番目のフレームが選択され，そこから i+read_len 番目のフレームまでの座標変換が更新される．

setTransform では write_len 個のデータの書き込みにおいて，write_len 回 *setTransform* を呼び出すので write_len タスク実行できたとみなすが，*setTransformsXact* では write_len 個のデータの書き込みでは，一回の呼び出しで write_len 個のデータの書き込みができるので，1 タスク実行できたとみなす．つまり，*setTransformsXact* の方がスループット・レイテンシの評価において不利になる．各インターフェイスの呼び出しに対応するタスク数を**表 1** に示す．

各実験は YCSB-A/C [15] ワークロードについて行った．YCSB-A/C はそれぞれ，読み込み操作と書き込み操作の割合が 50:50，100:0 のワークロードを指す．ここでは読み込み専用スレッドの数と書き込み専用スレッドの数の比でそれぞれのワークロードを再現するため，read_ratio をそれぞれ 0.5，1 に設定した．

特に記載がない場合は joint=1000000，read_len=16，write_len=16 で実験が行われている．また，Skew は 0 に設定して実験を行った．

5.4 YCSB-C

スループットについては**図 8** のように，old に比べて snapshot は最大 243 倍，latest は最大 257 倍のスループットとなった．old と比べ，論理コア数である 224 倍以上の性能差が出たのは，次に説明するように TF の木構造の C++の実装において各フレームを管理する方法を既存手法から変更したからだと考えられる．

各フレームは C++の実装において TimeCache クラスで表現され，TF の木構造中のフレーム群は std::vector<std::shared_ptr<TimeCache>> で管理される．std::shared_ptr 型は自身を参照しているスコープの数をカウンタで管理するため, 複数スレッドから std::shared_ptr 型のデータにアクセスする際にこのカウンタへの読み込み・書き込みが複数スレッドから行われる．これによりキャッシュミス率が増加し性能劣化につながる．これを避けるため，提案手法の実装ではフレーム群は std::shared_ptr 型を使わずに std::vector<TimeCache*> で管理される．このような実装の違いが論理コア数である 224 倍以上の性能差につながったと考えられる．

レイテンシについては**図 9**，**図 10** のように，どの手法においてもレイテンシとスレッド数が線形比例しているが，old に比べ snapshot，latest は小さいレイテンシとなった．

スループット，レイテンシのどちらにおいても提案手法が既存手法より優れているのは，既存手法ではジャイアントロックにより操作を並行に行えないが，提案手法では細粒度ロックと 2PL によって操作を並行に行えるからだと考えられる．また，latest の方が snapshot より優れた性能をしてしているのは，§ 3.3 で説明したように snapshot の *lookupTransform* では，*getLatestCommonTime* の呼び出しにより木構造を 2 度読み込む必要があるからだと考えられる．

書き込みが発生しないため，YCSB-C における書き込み

Vol.2022-ARC-248 No.35
Vol.2022-SLDM-198 No.35
Vol.2022-EMB-59 No.35
2022/3/10

表 1 各インターフェイスとタスクの数え方

インターフェイス	一度の呼び出しでアクセスするフレーム数	タスク数の数え方
old の lookupTransform	read_len	read_len 個のフレームを読むのが 1 タスク
old の setTransform	1	1 個のフレームに書き込むのが 1 タスク
snapshot の lookupTransform	read_len	read_len 個のフレームを読むのが 1 タスク
snapshot の setTransform	1	1 個のフレームに書き込むのが 1 タスク
latest の lookupLatestTransformXact	read_len	read_len 個のフレームを読むのが 1 タスク
latest の setTransformsXact	write_len	write_len 個のフレームに書き込むの　が 1 タスク

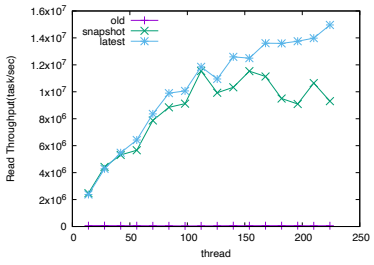

図 8 スループット（YCSB-C）

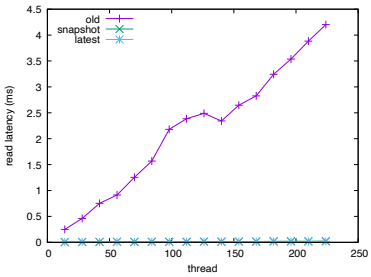

図 9 レイテンシ（YCSB-C）

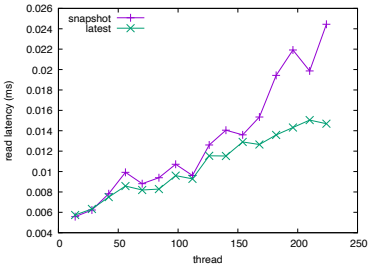

図 10 レイテンシ（YCSB-C）
snapshot と latest のみ

スループット，書き込みレイテンシ，データの鮮度，データの同期性，abort 率については記述しない．

5.5 YCSB-A

YCSB-A ではスループットについては**図 11** のように old に比べて snapshot は最大 61 倍，latest は最大 143 倍のスループットとなった．ここで，snapshot についてはスレッド数の増加とともにスループットが低下していることがわかる．

このようになる原因を調べるため，スレッド数とキャッシュミス率の関係について調べ，**図 12** に示した．ここからわかるように，latest ではキャッシュミス率がほとんど変化しないのに対し，snapshot ではキャッシュミス率が上がっていることがわかる．§ 4.1 で説明したように snapshot では *lookupTransform* にて同じ要素に対して二回の読み込みがある．この一回目の読み込みと二回目の読み込みの間にて *setTransform* による同じ要素への書き込みが発生するとキャッシュが汚染され，二回目の読み込み時にキャッシュミスとなる．スレッド数の増加とともにこの現象が増えることが原因となり，スループットの性能低下につながったと考えられる．

また，§ 3.3 にて説明したように二回目の読み込み時には両端キュー内の複数の要素への走査が発生する．このキュー内の要素へのアクセスもスループットの性能低下につながったと考えられる．

読み込みレイテンシについて**図 13**〜**図 14** に表示した．

スレッド数と abort 率の関係について**図 15** に示した．ここからわかるように，スレッド数と abort 率が線形比例していることがわかる．この理由はスレッド数の増加に伴い conflict が生じる可能性が高まったことだと考えられる．提案手法は NoWait であるため，deadlock の有無に関わらず，conflict 検知時に即時 abort する．

スレッド数とデータの鮮度の関係について**図 16** に示した．アクセスした各座標変換データのタイムスタンプの平均とアクセス時の時刻の差を delay とし，delay が少ない方がデータの鮮度が高いとみなす．ここからわかるように，snapshot ではスループットの低下によりスレッド数が増えるとデータの鮮度が落ちていくが，latest ではスレッド数に関係なく安定して高い鮮度のデータが取得できることがわかる．

5.6 制御周期

各スレッドにて操作を呼び出す周期各スレッドが一定の周期にて操作を呼び出すというのは，ROS において一般的なワークロードである．そこで，パラメータ frequency を設定し，操作の呼び出しが完了したのち，1 / frequency 秒待機してから再び操作を呼びだすワークロードにおける実験を行った．

図 17 は，スレッド数 200 の状態で frequency を 100 から 100000 まで変化させた時の読み込みレイテンシを表している．frequency を上げるとレイテンシも増えることがわかる．これは，制御周期を増やすことにより並行に実行される操作が増え，スレッド間の競合が増加するからだと考えられる．

5.7 議論

本研究のアプリケーションとして次の三つが挙げられる．

- 関節数が多く，高精度な制御が求められるヘビ型ロ

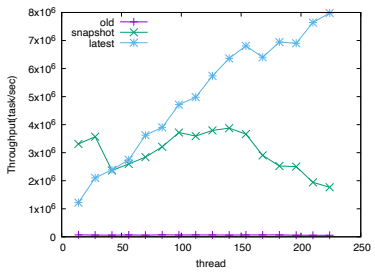

図 11 スループット（YCSB-A）

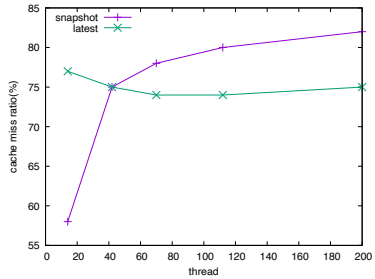

図 12 キャッシュミス率（YCSB-A）

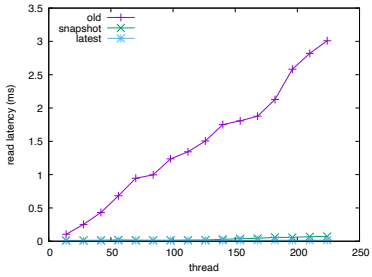

図 13 読み込みレイテンシ（YCSB-A）

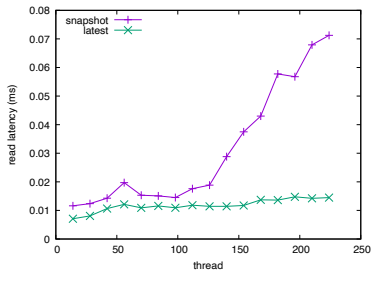

図 14 読み込みレイテンシ（YCSB-A）
snapshot と latest のみ

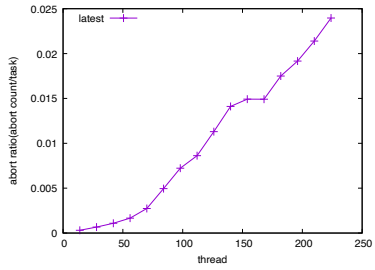

図 15 abort 率（YCSB-A）

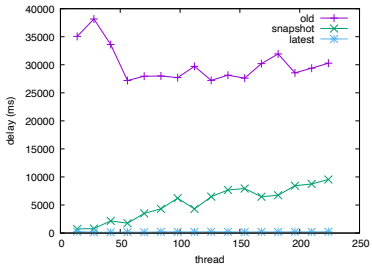

図 16 データの鮮度（YCSB-A）

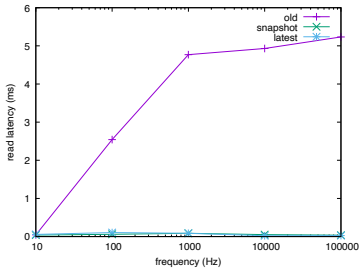

図 17 制御周期とレイテンシ

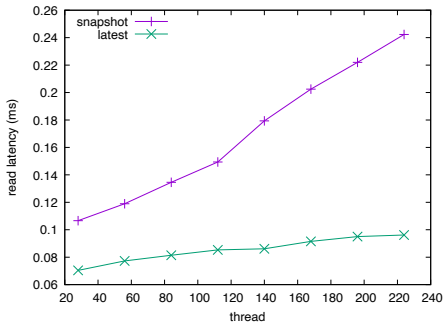

図 18 エッジクラウドのワークロードのエミュレーション

ボット

- 自動運転におけるエッジクラウド
- 大量のドローンの群の制御

自動運転におけるエッジクラウドでは，一部のローカルな区域における自動運転車両や発生した障害物の位置関係を管理することによって，渋滞の緩和や障害物の回避を行う事ができる．リアルタイム性が求められ，大量の自動運転車両や障害物の位置関係を TF ライブラリで管理するには，本研究で提案した手法を用いる事が有用である．

[21] には各都道府県における各自動車道（高速自動車国道，一般国道の自動車専用道路，一般国道など）の 12 時間あたりの交通量が記載されており，どの道路でも 12 時間あたりの交通量は高々 80,000 である．[22] には全国の高速道路の一日の交通量が記載されており，東北自動車道が 277,584 で最大である．これらのデータを元に，高速道路の入り口（IC）から出口（別の IC）までのローカルな区域の情報を一つのエッジサーバーが管理すると想定すると，次のようなワークロードが想定できる．

- エッジクラウドにて管理する自動車の台数は高々 1000 台．
- 各自動車は，自分の周辺の自動車や障害物，標識など，高々 20 の情報を書き込む．
- 各自動車は，渋滞回避や障害物などの回避のために高々 100 の周辺の物体の情報を読み込む．

本研究の評価において行った実験と同じように，joint=1000, read_ratio=0.5, read_len=100, write_len=20, frequency=120 に設定してこのワークロードを再現すると，読み込みレイテンシについて既存手法はスレッド数が 224 の状況において 16ms 程度となり，提案手法については **図 18** のようになった．frequency=120 で設定してあるので読み込みレイテンシは約 8.3ms 以内にする必要があるが，既存手法ではこのデッドラインに間に合っていないことがわかる．これに対し，提案手法の読み込みレイテンシは高々 0.24ms 程度となり，デッドラインに間に合っていることがわかる．

6. 結論

既存の TF ライブラリはジャイアンロックにより，アクセスが完全に逐次化されアクセスするスレッドが増えるに従ってパフォーマンスが低下する問題があった．

本研究では TF ライブラリにデータベースの並行性制御法における細粒度ロッキング法を用いてこの問題を解決した．その結果，既存手法と比べて最大 243 倍のスループット，最大 172 倍高速なレイテンシとなった．

また，既存の TF ライブラリはその仕様によって座標変換の計算時に最新のデータを参照しないためデータの鮮度が落ちるという問題があった．本研究では 2PL を適用することにより，複数の座標変換の最新のデータを atomic に取得できるインターフェイス（*lookupLatestTransformXact*），及び複数の座標変換の最新のデータを atomic に更新するインターフェイス（*setTransformsXact*）を提供することによって解決した．その結果，既存手法と比べて最大 132 倍のデータ鮮度となった．また，既存手法と比べ最大 257 倍のスループット，最大 282 倍高速化したレイテンシとなった．本研究の提案手法を適用した TF ライブラリは [19] にて公開している．

本研究では，トランザクションによって ROS の高精度化，高品質化を実現した．これは筆者の知る限り，これまでにないアプローチである．これはデータベースとロボットを融合した研究領域の第一歩であり，この研究領域の開拓を進める．

謝辞 この成果は，科研費 JP19H04117 ならびに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す．

参考文献

[1] T. Foote, ”tf: The transform library,” 2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA), 2013, pp. 1-6, doi: 10.1109/TePRA.2013.6556373.
[2] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in ICRA Workshop on Open Source Software, 2009.
[3] 竹内栄二朗（2008）. 移動ロボットの基本機能のモジュール化と環境地図生成に関する研究, 筑波大学博士（工学）学位論文.
[4] Philip A. Bernstein and Nathan Goodman, ”Concurrency Control in Distributed Database Systems” in ACM Computing Surveys, 1981, pp. 185-221
[5] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for high- lycontended dynamic workloads on a thousand cores. Proceedings of the VLDB Endowment, Vol. 10, No. 2, p. p. 49âĂŞ60, 2016.
[6] Stephen Tu, Wenting Zheng, Eddie Kohler †, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In SOSP, pages 18–32. ACM, 2013
[7] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. InProceedings of the 2017 ACM International Conference onManagement of Data, p. p. 21âĂŞ35, 2017.
[8] 南祐衣, 鈴木汰一, 山倉拓海, 構勇海, 潘鴻遠, 及川裕介（筑波大）, 荻原湧志, 川島英之（慶應大), 伊達央, 萬礼応（筑波大）: つくばチャレンジ 2021 における 筑波大学知能ロボット研究室チーム Aqua の取り組み. 第 22 回 計測自動制御学会.
[9] ros/geometry2：BufferCore.h(online)，入手先 ⟨https://github.com/ros/geometry2/blob/melodic-devel/tf2/include/tf2/buffer_core.h⟩ (2022.1.17).
[10] ros/geometry2：BufferCore.cpp(online)，入手先 ⟨https://github.com/ros/geometry2/blob/melodic-devel/tf2/src/buffer_core.cpp⟩ (2022.1.17).
[11] GAIA platform(online)，入手先 ⟨https://www.gaiaplatform.io⟩ (2022.1.16).
[12] ROS2(online)，入手先 ⟨https://docs.ros.org/en/rolling/⟩ (2022.1.18).
[13] Autoware，入手先 ⟨https://tier4.jp/en/autoware/⟩ (2022.1.18).
[14] Autoware-AI/core_perception：ndt_matching(online)，入手先 ⟨https://github.com/Autoware-AI/core_perception/tree/master/lidar_localizer/nodes/ndt_matching⟩ (2022.1.18).
[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. InProceedings of the 1st ACM symposiumon Cloud computing, p. p. 143âĂŞ154, 2010.
[16] Guna Prasaad, Alvin Cheung and Dan Suciu. Improving High Contention OLTP Performance via Transaction Scheduling.
[17] ros/geometry2(online)，入手先 ⟨https://github.com/ros/geometry2⟩ (2022.1.17).
[18] ros2/geometry2(online)，入手先 ⟨https://github.com/ros2/geometry2⟩ (2022.1.17).
[19] Ogiwara-CostlierRain464/geometry2(online)，入手先 ⟨https://github.com/Ogiwara-CostlierRain464/geometry2⟩ (2022.1.17).
[20] Ogiwara-CostlierRain464/geometry2：cache.cpp (online)，入手先 ⟨https://github.com/Ogiwara-CostlierRain464/geometry2/blob/68651682b124cb9d3403cf729e8f9e0c1c1319f5/tf2/src/cache.cpp#L166⟩ (2022.1.17).
[21] NEXCO 東日本：高速道路の通行台数と料金収入（令和 2 年度）(online)，入手先 ⟨https://www.e-nexco.co.jp/activity/word_data/data/r02.html⟩ (2022.1.20).
[22] 平成 27 年度 全国道路・街路交通情勢調査：一般交通量調査 集計表 交通量整理表（都道府県別道路種別別）(online)”，入手先 ⟨https://www.mlit.go.jp/road/census/h27/data/pdf/syuukei04.pdf⟩ (2022.1.20).