Vol.2019-ARC-240 No.7
Vol.2019-SLDM-191 No.7
Vol.2019-EMB-53 No.7
2019/2/27

グラフ処理を題材とした 最適なトランザクショナルメモリプログラミングの検討

山下 淳[1]　浅井 優太[1]　小林 龍之介[1]　二間瀬 悠希[1]　塩谷 亮太[2]　五島 正裕[3]　津邑 公暁[1]

概要：並列プログラミング環境では，共有変数の一貫性を保証するために一般的にロックが用いられる．しかし，ロックを適切な粒度で使用することが困難であることや，デッドロック状態に陥る可能性があることから，ロックを補完・代替する並行性制御機構としてトランザクショナルメモリ（Transactional Memory: TM）が提案されている．TM では，ロックで保護していたクリティカルセクションを含む一連の命令列をトランザクションとして定義する．そして，複数のトランザクション間で同一アドレスに対するアクセスが競合しない限りトランザクション同士を並列に実行することができる．しかし，TM の活用は未だ普及しておらず，その原因の一つに，TM の適切な利用方法および TM を効果的に使用可能なプログラムが認知されていないことが挙げられる．したがって，TM に適したプログラミング技法の体系化が必要であると考えた．そこでまず，TM を使用して性能が向上するプログラムと向上しにくいプログラムについて調査した．その結果，グラフ処理に分類される一部のアプリケーションでは，TM の利用により速度が向上しづらく，場合によっては低下することを確認した．そこで，グラフ処理をどのように記述すれば TM によって性能を引き出せるのかについて検討し，グラフ処理プログラムの適切な改変方法について検討した．

1. はじめに

これまでプログラムの高速化は，スーパスカラに代表されるような命令レベル並列性（Instruction-Level Parallelism: ILP）に基づく様々な手法や，集積回路の微細化による高クロック化の実現によって支えられてきた．しかしながら，プログラム中で抽出できる ILP には限界があり，消費電力や配線遅延の相対的な増大から，単一プロセッサ・コアの動作クロック周波数の向上も困難となりつつある．このような流れから，単一チップ上に複数のプロセッサ・コアを搭載したマルチコア・プロセッサが広く普及しつつある．マルチコア・プロセッサでは，これまで単一プロセッサ・コアで実行していたタスクを複数のプロセッサ・コアに分担し，並列処理することで，高いスループットを実現することができる．

このようなマルチコア環境では，複数のプロセッサ・コア間で単一アドレス空間を共有する共有メモリ型の並列プログラミングモデルが広く利用されている．このプログラミングモデルに基づいたプログラムでは，共有変数に対するアクセスを調停する必要があり，その調停を行う仕組みとして，これまで一般的にロックが用いられてきた．しかし，ロックを用いる場合，ロック操作のオーバヘッドによる速度性能の低下や，デッドロックの発生などの問題が生じる可能性がある．さらに，ロックを適切な粒度で使用することは困難であるため，ロックはプログラマにとって必ずしも利用しやすい仕組みではない．

そこで，ロックを補完・代替する並行性制御機構として**トランザクショナルメモリ（Transactional Memory: TM）**[1] が提案されている．TM は，データベースの更新・検索操作に用いられるトランザクションの概念をメモリアクセスに適用したものである．TM を使用する場合は，従来ロックで保護していたクリティカルセクションを含む一連の命令列を，トランザクションとして定義する．そして，共有変数に対するアクセス競合が発生しない限りトランザクション同士を投機的に並列実行することで，ロックを用いる場合よりも高い並列性を実現することができる．

なお，TM ではトランザクションの実行が投機的であるため，トランザクションを実行するスレッド間において，同一アドレスに対するアクセスが競合しているか否かを監視する必要がある（競合検出）．TM のハードウェア実装であるハードウェアトランザクショナルメモリ（**Hardware Transactional Memory: HTM**）[2], [3], [4] では，競合

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 東京大学
The University of Tokyo
[3] 国立情報学研究所
National Institute of Informatics

Vol.2019-ARC-240 No.7
Vol.2019-SLDM-191 No.7
Vol.2019-EMB-53 No.7
2019/2/27

検出の機構をハードウェアで実現することで，トランザクション操作によるオーバヘッドを抑制している．このような利点から，HTM は現在大きな注目を集めており，Intel 社の Haswell[5], [6] 以降のマイクロアーキテクチャや IBM 社のスーパーコンピュータ BlueGene/Q[7]，IBM 社のサーバ向けプロセッサ POWER8[8] 以降のプロセッサなどに搭載されている．また，実装の調査および評価が広く行われており，今後もさらなる性能の向上が期待されている．

このように，TM は商用プロセッサへの搭載が始まっており，TM の性能を改善するための研究も盛んに行われている．一方で，TM を使用するプログラムの記述に関する知見が不足しており，プログラミングの指針が確立されていない．したがって，TM に適したプログラミング技法の体系化が必要であると考えられる．そこで，本論文ではまず，TM を使用して性能が向上するプログラムと向上しにくいプログラムについて調査する．次に，調査の結果，性能が向上しにくい傾向にあると判明したグラフ処理に着目し，TM を使用した場合に起こりうることを考察する．そして，グラフ処理をどのように記述すれば TM で高速化できるかの検討を通じて，TM に適したプログラミング技法に関する知見の獲得を目指す．

2. トランザクショナルメモリ

2.1 TM の概要

マルチコア・プロセッサを活用する並列プログラミングでは，複数のプロセッサ・コア間で単一アドレス空間を共有する共有メモリ型のプログラミングモデルが広く利用されている．このプログラミングモデルでは，複数のプロセッサ・コアが共有メモリに対して行うメモリアクセスを調停する必要があり，その方法としてこれまで一般的にロックが用いられてきた．ロックを用いる場合，クリティカルセクション同士を排他的に実行することで，メモリアドレスの値の一貫性を保証する．しかし，適切な粒度でロックを適用することは困難である．例えばロックを粗粒度に定義する場合，プログラムの設計は容易であるが，逐次実行される区間が長くなるため，実行時に抽出できる並列性は低くなる．一方，細粒度にロックを定義する場合，粗粒度に定義した場合と比較して並列度が向上するが，プログラマはデッドロックの発生に細心の注意を払う必要があり，プログラミングが複雑になりやすい．さらに，頻繁なロック獲得・解放操作は膨大なオーバヘッドを発生させる可能性がある．以上のように，ロックは生産性と性能とがトレードオフの関係にあり，プログラマにとって必ずしも利用しやすい仕組みではない．

そこで，ロックに代わる並行性制御機構として，TM が提案されている．TM はデータベースの更新・検索操作に用いられる概念であるトランザクション処理をメモリアクセスに対して応用した機構である．TM を使用する場合，

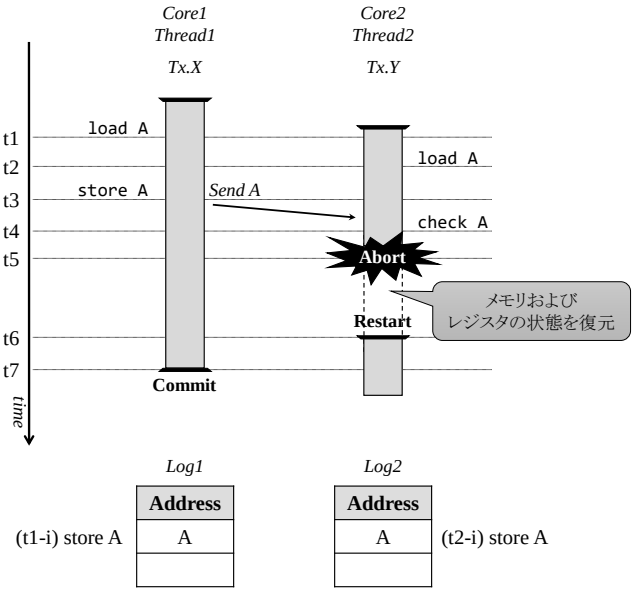


図 1 競合検出の動作

従来ロックで保護されていたクリティカルセクションを含む一連の命令列をトランザクションとして定義し，これらを投機的に並列実行することで，ロックを使用する場合よりも高い並列度が得られる．また，プログラマは，トランザクションを定義する際に，トランザクションの開始と終了を記述するだけでよく，デッドロックなどを意識する必要がない．したがって，TM は粗粒度ロックと同等以上の高いプログラマビリティを持ちつつ，細粒度ロックと同等以上の高い性能を発揮するポテンシャルを有しており，生産性と性能を両立しうる機構として期待されている．

2.2 競合検出

TM では，投機的にトランザクションを実行するために，共有変数へのアクセスを監視する．そして，トランザクションを実行している複数のスレッドが同一共有変数へアクセスを試みた場合，**競合（Conflict）**を検出する．この操作を**競合検出（Conflict Detection）**と呼ぶ．

ここで，トランザクションの並列実行および競合検出の動作を**図 1** に示す．図におけるバーはそれぞれトランザクション処理を示しており，下向きに時間軸をとっている．図中の A，B は共有変数のアドレスを表し，`load A`，`load B` はそれぞれアドレス A およびアドレス B に対するロード命令を，`store A`，`store B` はそれぞれアドレス A およびアドレス B に対するストア命令を，当該時刻において発行したことを表している．また，図の下部に示す Log は変数のアドレスを記憶するために各スレッドに関連付けられている専用領域を表す．

まず，*Thread*1 が `load A` を実行したとする（t1）．このとき，*Thread*1 はアドレス A を Log1 に記憶する（t1-i）．その後，*Thread*2 が `load A` を実行すると，アドレス A を Log2 に記憶する（t2 および t2-i）．次に，*Thread*1 が `store`

A を試みたとする（t3）．このとき，*Thread*1 は *Thread*2 に対して書き込み対象のアドレスであるアドレス A を送信する．*Thread*2 は，受信したアドレスと自身の専用領域に記憶されたアドレスとを比較する（t4）．この例では，*Thread*2 がトランザクション内で `load` したアドレスに対する `store` であるため，WaR アクセスに該当することを検知し，このアクセスを競合として検出する．アクセス競合を検出した場合，一貫性を維持するため，いずれかのトランザクションが実行を中止する．これを**アボート（Abort）**という．どちらのトランザクションがアボートするかは実装により異なるが，ここでは *Thread*2 の *Tx.Y* がアボートし，*Thread*1 の *Tx.X* が処理を続行したとする（t5）．トランザクションをアボートしたスレッドは，トランザクションの開始時点におけるメモリおよびレジスタの状態を復元する．この処理を**ロールバック（Rollback）**という．ロールバックを終えると，トランザクションの開始時点から処理を再実行する．この例では，*Thread*2 が *Tx.Y* 開始時点の状態を復元し，*Tx.Y* を再実行する（t6）．一方，処理を続行する *Thread*1 は，*Tx.X* の終了までに他の競合が検出されず，トランザクションの処理を最後まで完了した場合，トランザクション内で行ったデータの更新を確定できる（t7）．これを**コミット（Commit）**という．

2.3 競合以外のアボート原因

前節では，競合によりトランザクションがアボートすることについて述べたが，トランザクションがアボートする原因は，競合だけではない．競合以外の原因でアボートするケースは大きく分けて 3 つある．

1 つ目のケースは，トランザクション内でアクセスするアドレス数が，専用ハードウェアで記憶可能なアドレス数の上限を超えるケースである．競合検出のためのアクセス対象アドレスの記憶は，専用領域を用いて行うが，この領域の容量は当然ながら有限である．この上限を超える数のアクセスが発生する場合，一貫性を保証できなくなるため，トランザクションをアボートする．以降，本論文では，このようなアボートを**キャパシティアボート**と呼ぶ．

TM が搭載されたプロセッサで，トランザクションをキャパシティアボートする状況について説明する．Intel 社のプロセッサでは，キャッシュメカニズムを利用して競合検出を実現している．図 1 における Log の役割を L1d キャッシュが担っており，各スレッドはトランザクション内でアクセスしたアドレスを各キャッシュラインに記憶する．また，キャッシュラインにアクセスすると，キャッシュコヒーレンスプロトコルにしたがい，当該ラインへアクセスした通知を他のスレッドへ送信する．トランザクションをコミットする前に，トランザクション内でアクセスしたアドレスが記憶されたキャッシュラインに対する通知を受信した場合，他のスレッドによるこのアクセスが競合として検出される．そのため，トランザクションをコミットする前に，トランザクション内でアクセスしたキャッシュラインが追い出されると，当該ラインで記憶していたアドレスに対する競合を検出できなくなる．したがって，トランザクション内でアクセスしたキャッシュラインの追い出された場合にトランザクションをキャパシティアボートする．

IBM 社の POWER8 および POWER9 でも，Intel 社のプロセッサと同様にキャッシュメカニズムを利用しているが，それに加えて，TMCAM と呼ばれる TM 専用のハードウェアを利用して競合検出を実現している．この場合でも，TMCAM が記憶できるエントリ数の上限を超えると，競合を検出でいなくなるため，トランザクションをキャパシティアボートする．

2 つ目のケースは，トランザクション処理中に割込みが発生する場合である．Intel 社のプロセッサでは，トランザクションを実行中に割込みが発生した場合，一貫性を保証できなくなるため，トランザクションをアボートする．一方，IBM 社の POWER プロセッサでは，suspend および resume と呼ばれる機能が実装されており，割込みが発生しても，トランザクションをアボートしないことが可能になっている．suspend 機能を使用すると，実行中のトランザクションを一時的に中断することができる．suspend によってトランザクションを中断した状態で resume 機能を使用すると，中断した状態から復帰し，トランザクションの処理を再開することができる．ハードウェアによって自動でこれらの機能が使用され，トランザクション内で発生する割込みに対処することができる．なお，トランザクションを開始してからトランザクションを中断する間にトランザクション内でアクセスしたアドレスは記憶されたままであるため，トランザクションを中断している間にも，他のトランザクションとの競合検出は行われている．そのため，競合検出によって，中断しているトランザクションをアボートする必要がある場合，resume 機能によってトランザクションの処理を再開した直後にトランザクションをアボートする．

3 つ目のケースは，トランザクションとして定義した区間内に TM でサポートされていない処理が含まれる場合である．例えば，入出力処理およびキャッシュラインの状態を変更する処理が挙げられる．トランザクション内で入出力処理が発生した場合，その後の処理が原因でトランザクションをアボートしても，入出力した結果を破棄し，ロールバックすることができない．したがって，TM では printf 関数のような入出力を行う関数がトランザクション内に含まれる場合，トランザクションをアボートする．また，トランザクション内でキャッシュラインの状態を変更する命令が使用されると，当該ラインで記憶していたアドレスに対する競合を検出できなくなるため，トランザクションをアボートする．

Vol.2019-ARC-240 No.7
Vol.2019-SLDM-191 No.7
Vol.2019-EMB-53 No.7
2019/2/27

これら 3 つのケースでトランザクションをアボートした場合，ロールバック後にトランザクションを再実行しても，再び同様の原因でアボートし，処理を進行できなくなる．そのため，TM を使用してトランザクションを実行することが不可能となり，別の方法を用いてトランザクションとして定義した区間の処理を進行する必要がある．このように，別の方法を用いた実行に切り替えることを**フォールバック（Fallback）**という．フォールバック先として，一般にロックによる排他実行が用いられる．この場合，トランザクションとして定義した区間をクリティカルセクションとして排他実行するため，フォールバックする際に他のトランザクションを全てアボートする必要がある．したがって，フォールバックもトランザクションがアボートするケースとなる．

3. TM を使用して性能が向上しにくいプログラムの調査

2.2 節および 2.3 節で述べたように，トランザクションがアボートする原因は複数存在し，これらのアボートが TM を使用するプログラムの性能に影響を与えると考えられる．そこで，どのようなプログラムで性能が向上しにくいのか，また，向上しにくいプログラムに共通点があるのかを調査した．

調査環境を**表 1** に示す．調査には，Intel 社および IBM 社のプロセッサに搭載されている TM を使用した．ベンチマークプログラムには，TM の研究で広く用いられる STAMP ベンチマークスイート [9] から，7 種類のプログラムを用いた．評価に用いたプログラムとそれらの実行時引数は**表 2** に示す通りである．なお，2.2 節で述べたように，アクセス競合でトランザクションをアボートした場合，トランザクションを再実行するが，再実行してもアボートを繰り返し，処理を進行できなくなる可能性がある．そのため，トランザクションを再実行する回数の上限を設け，これを超える場合はトランザクションとして実行することを断念し，ロックにフォールバックするようにした．本調査では，Intel 社および IBM 社のプロセッサで再実行回数について調査した仲池らの研究 [10] に基づき，両プロセッサで再実行回数を 16 回とした．

調査結果を**図 2** に示す．図では，各プログラムの調査結果をそれぞれ 2 本の折れ線で示しており，それぞれ，Intel Xeon，POWER9 の結果に対応している．各グラフの縦軸は，TM を使用せずに 1 スレッドで逐次実行したときの実行時間に対する，TM を使用して実行した場合の速度向上率を表し，横軸はスレッド数を表す．なお，各プログラムは 10 回ずつ実行し，それらの実行時間の平均から速度向上率を算出している．また，1 つのスレッドが 1 つのコアを専有できるよう，taskset コマンドを使用して 1 つのスレッドに 1 つのコアを割り当てた．

表 1 調査環境

CPU	Xeon Gold 6152	POWER9
物理/論理 コア数	22/44	16/64
クロック	2.10GHz	2.20GHz
L1 データキャッシュ	32KB	32KB
L2 キャッシュ	1MB	512KB
L3 キャッシュ	30.25MB	10MB
メモリ	16GB	16GB
OS	ubuntu 16.04.5 LTS	ubuntu 18.04.2 LTS
コンパイラ	gcc version 5.5.0	gcc version 7.4.0
オプション	-O3	-O3

表 2 ベンチマークプログラムの実行時引数

Genome	-g16384 -s64 -n16777216
Intruder	-a10 -l128 -n262144 -s1
Kmeans	-m40 -n40 -t0.00001 -i random-n65536-d32-c16
Labyrinth	-i random-x512-y512-z7-n512
SSCA2	-s20 -i1.0 -u1.0 -l3 -p3
Vacation	-n2 -q90 -u98 -r1048576 -t4194304
Yada	-a15 -i ttimeu100000.2

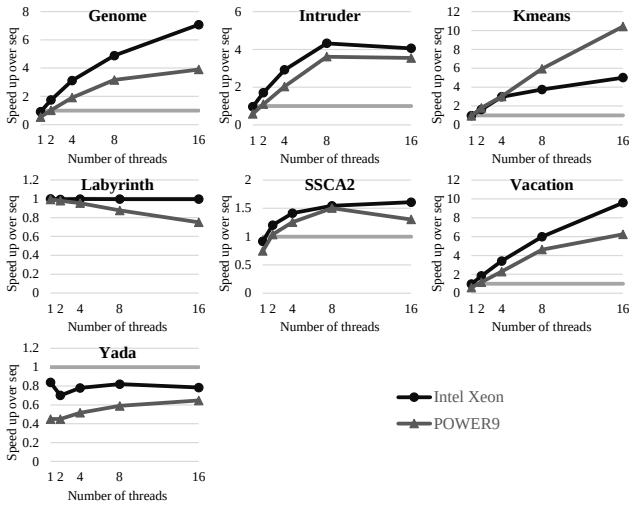


図 2 調査結果

調査の結果，Genome，Kmeans，Vacation では，両方のプロセッサにおいて，スレッド数の増加に伴い速度が向上している．特に，Kmeans では POWER9 で実行した場合に約 10.4 倍，Vacation では Intel Xeon で実行した場合に約 9.6 倍の速度向上であり，TM を使用して性能が大きく向上していることがわかる．また，Intruder，SSCA2 では，速度向上率が 8 スレッドで頭打ちになっているが，TM を使用して性能が向上していることがわかる．一方，Labyrinth では，Intel Xeon で実行した場合，スレッド数が増加しても，TM を使用しない場合と比較して速度が変化しておらず，POWER9 で実行した場合，速度が低下している．また，Yada では，両方のプロセッサにおいて，TM を使用しない場合よりも速度が低下している．以上の結果から，TM を使用して性能が向上するプログラムと向上しにくいプログラムがあることがわかる．TM を使用して性

Vol.2019-ARC-240 No.7
Vol.2019-SLDM-191 No.7
Vol.2019-EMB-53 No.7
2019/2/27

```
1 while (!queue.isEmpty()) {
2     node v = queue.pop();
3     /* に対する処理 v */
4     for (edge : v.edges))
5         node n = edge.getNode(v);
6         if (/* が未処理の場合 n */)
7             queue.push(n);
8 }
```

図 3 幅優先探索を簡略化したコード

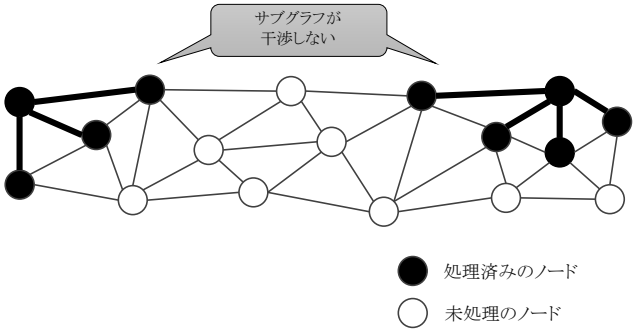


図 4 干渉しないサブグラフ

能が向上しなかった Labyrinth および Yada に着目すると，Labyrinth は経路探索を行うプログラムであり，Yada はメッシュ生成を行うプログラムであることから，ともにグラフ処理を含むことがわかる．したがって，グラフ処理の一部のアプリケーションには，TM を用いて高速化しづらい特徴が含まれると考えられる．

4. グラフ処理

3 章で示した調査結果より，グラフ処理に分類される一部のアプリケーションは，TM で高速化しづらい共通の特徴を持つと考えられる．そこで本章では，アルゴリズムの面から，グラフ処理に TM を適用した場合に起こりうることを考察する．

4.1 グラフ処理の特徴

グラフとはノードの集合とエッジの集合で構成されるデータ構造であり，様々なモデルを表現できる．例として，計算機のネットワークや道路網が挙げられる．グラフを処理するアルゴリズムとして，2 つのノード間における最短経路を求めるダイクストラ法や，最小全域木を求めるクラスカル法など，多数のアルゴリズムが存在し，幅広く研究されている．

このようなグラフ処理アルゴリズムの一部は，サブグラフに対する処理を繰り返すという特徴を持ち，あるノードから処理を開始し，隣接するノードにアクセスすることで，処理するサブグラフの範囲を広げていく．このような処理の例として，幅優先探索のコードを**図 3** に示す．

まず，1 行目で while ループ内に入ると，2 行目でキューから，処理するノード v を取り出す．ここではノード v とする．その後，v に対して処理を行い，v に隣接するノードが未処理であれば，キューに追加することで，処理するサブグラフの範囲を広げていく．キューが空になり，while ループを抜けると，また別のノードから処理を開始する．

このようなグラフ処理では，**図 4** のように処理するサブグラフ同士が干渉しない場合，サブグラフの並列処理が可能であるため，グラフ処理は高い並列性を持つ．一方，複数のノードをたどりながら処理を行うため，アクセスする変数の数が多くなる．

```
1  while (!queue.isEmpty()) {
2      node v = queue.pop();
3      mutex_lock(v);
4      /* に対する処理 v */
5      for (edge : v.edges))
6          node n = edge.getNode(v);
7          if (/* が未処理の場合 n */)
8              queue.push(n);
9  }
10 /* 獲得した全てのロックを解放 */
```

図 5 ロックを使用した実装の例

```
1  BEGIN_TRANSACTION();
2  while (!queue.isEmpty()) {
3      node v = queue.pop();
4      /* に対する処理 v */
5      for (edge : v.edges))
6          node n = edge.getNode(v);
7          if (/* が未処理の場合 n */)
8              queue.push(n);
9  }
10 END_TRANSACTION();
```

図 6 TM を使用した実装の例

4.2 並列処理時の問題点

4.1 節で述べたように，グラフ処理は高い並列性を持つ．グラフ処理を並列化する際には，サブグラフ同士の干渉を防ぐ必要があり，ロックを使用する排他的な方法と，TM を使用する投機的な方法の 2 つが考えられる．しかし，それぞれの方法において，グラフ処理でアクセスする変数の多さが問題点となる．図 3 に示した幅優先探索の処理を並列化する場合を例に説明する。

ロックを使用して排他的に処理する場合，**図 5** に示すように，それぞれのスレッドがノードごとにロックを獲得することで，他のスレッドから自身が処理するサブグラフへのアクセスを禁止する方法が考えられる．この場合，アクセスする全てのノードに対してロックを獲得しなければならず，多数のロック獲得に伴い膨大なオーバヘッドが発生する．更に，デッドロックを回避するために，どのような順序でノードのロックを獲得するかを制御する必要がある．科学技術計算に用いられるような大規模なグラフでは，こ

Vol.2019-ARC-240 No.7
Vol.2019-SLDM-191 No.7
Vol.2019-EMB-53 No.7
2019/2/27

表 3　キャパシティアボートの割合

	Intel Xeon	POWER9
Labyrinth	100%	100%
Yada	84.7%	89.9%

の問題点は顕著になり，プログラムの設計コストが膨大になる可能性がある．

一方，TM を使用する場合，**図 6** に示すように，サブグラフに対する処理をトランザクションとして定義する方法が考えられる．この場合，プログラマはトランザクションの開始と終了を記述するのみでよく，ロックを使用する場合と比較して，プログラム設計コストが小さい．しかし，トランザクション内でアクセスする変数の数が多いため，アクセス競合が発生しやすくなる．また，競合が発生しなかったとしても，膨大なメモリアクセスを監視する必要があるため，トランザクションをキャパシティアボートがする可能性がある．

そこで，Labyrinth および Yada で発生したアボートに占めるキャパシティアボートの割合について調査した．調査環境は 3 節に示したものと同様である．トランザクションの実行回数とキャパシティアボート回数を計測し，キャパシティアボート回数をトランザクションの実行回数で割ることで，割合を算出した．なお，キャパシティアボート回数の計測には，それぞれのプロセッサで用いられるレジスタを利用した．Intel Xeon および POWER9 では，トランザクションをアボートした際に，アボートの原因に対応した値をレジスタに格納する．この値を参照することで，キャパシティアボート回数を計測した．

調査結果を**表 3** に示す．調査の結果，発生しているアボートのうちキャパシティアボートである割合は，Labyrinth では，両方のプロセッサにおいて 100%であり，Yada では，Intel Xeon で実行した場合は 84.7%，POWER9 で実行した場合は 89.9%であることがわかった．したがって，実行したトランザクションの多くがロックにフォールバックされ，TM を使用しても並列に実行できる区間が短く，TM を使用しない場合と比較して性能が低下すると考えられる．グラフ処理に限らず，細粒度にロックを定義することが難しいプログラムに TM を使用する場合，同様の問題が発生しうると考えられる．

5. プログラムの改良

本章では，4.2 節で述べた問題点を解決するために，TM でグラフ処理を高速化するプログラミングについて検討し，Labyrinth と Yada の改良を試みる．

5.1 改良の方針

TM に適したプログラミングを検討するにあたり，小林らの並列メッシュ生成を TM で高速化するアルゴリズム研究 [11] に着目した．この研究は，メッシュ生成の並列化に TM を使用する場合，TM の性能を発揮するためにトランザクションとして定義する区間を短縮する必要があることを指摘している．従来の実装では，4 章で示したコードのように，トランザクションとして定義する区間に含まれるメモリアクセスが多いため，トランザクションが頻繁にアボートすることによって性能が低下してしまう．そのため，この研究は，メッシュ生成の処理を分割して整理し，一部の処理をトランザクションの外に出すことで，メッシュ生成に対し効果的に TM を使用するアルゴリズムを提案している．まず，メッシュ生成処理を以下の 4 フェーズに分割する．

traversal: 再構成するメッシュの範囲を決定する．

calculation: traversal フェーズで決定した範囲に含まれる三角形を削除し，新しい三角形を生成する．

validation: traversal フェーズおよび calculation フェーズでアクセスする三角形が他のスレッドによって削除されていないかどうかを検証する．

updating: calculation フェーズで生成した三角形でメッシュを更新する．

このうち，メモリアクセスが多く，処理時間の長い traversal フェーズおよび calculation フェーズをトランザクションから除外する．そして，新たに追加する validation フェーズで，除外した区間でアクセスする変数の一貫性を検証し，トランザクションとして定義する区間を短縮する．このアルゴリズムはメッシュ生成に対するアルゴリズムとして提案されたが，トランザクション内でのメモリアクセスを減らすという点に着目していることから，メッシュ生成以外の処理にも適用可能ではないかと考えられる．そこで，本論文では，このアルゴリズムを抽象化することで，TM に適したプログラミングとして一般化し，メッシュ生成以外のグラフ処理にも適用することを検討する．

まず，上記の 4 フェーズを，以下のように捉え直す．

traversal: 処理するサブグラフの範囲を決定する．

calculation: traversal フェーズで決定した範囲に含まれるデータを処理する．

validation: traversal フェーズおよび calculation フェーズでアクセスする変数の一貫性を検証する．

updating: calculation フェーズの処理結果に基づき共有変数を更新する．

```
1 BEGIN_TRANSACTION();
2 traversal();
3 calculation();
4 updating();
5 END_TRANSACTION();
```

図 7　3 つのフェーズをトランザクションとして定義する場合の疑似コード

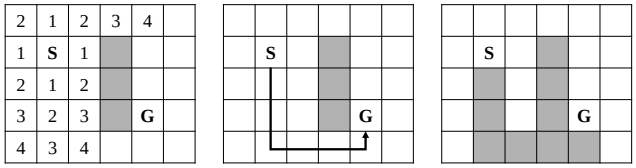
Vol.2019-ARC-240 No.7
Vol.2019-SLDM-191 No.7
Vol.2019-EMB-53 No.7
2019/2/27

```
1  RETRY:
2      traversal();
3      calculation();
4      BEGIN_TRANSACTION();
5      if (!validation()) {
6          END_TRANSACTION();
7          goto RETRY;
8      }
9      updating();
10     END_TRANSACTION();
```

図 8 4 つのフェーズを実装した場合の疑似コード

従来の実装では，**図 7** に示す疑似コードのように，3 つのフェーズを 1 つのトランザクションとして定義するが，トランザクションとして定義する区間を短縮するために，一部のフェーズをトランザクションから除外することを考える．これら 3 つのフェーズのうち，traversal フェーズでは，処理するサブグラフの範囲を決定するために，その後の calculation フェーズや updating フェーズではアクセスしないノードにもアクセスする必要がある．また，calculation フェーズでは，複数のデータを使用して処理を行うため，アクセスする変数の数が多くなる．したがって，これらのフェーズをトランザクションから除外することで，トランザクション内でアクセスする変数の数を削減し，競合およびキャパシティアボートを抑制できると考えられる．しかし，ただ除外するだけでは，2 つのフェーズでアクセスする変数に対して競合が検出されず，一貫性が失われる可能性がある．

そこで，calculation フェーズと updating フェーズとの間に，traversal フェーズおよび calculation フェーズでアクセスした変数の一貫性を検証する validation フェーズを追加し，validation フェーズと updating フェーズをトランザクションとして定義する．validation フェーズを追加した場合の疑似コードを**図 8** に示す．5 行目が validation フェーズであり，検証に成功した場合，処理するサブグラフの範囲が他のトランザクションと干渉していないため，8 行目の updating フェーズを実行する．検証に失敗した場合，自身が更新しようとしたサブグラフの範囲が，既に他のトランザクションによって更新されていることになるため，6 行目の命令でトランザクションを終了し，goto 文で RETRY ラベルまで戻り，traversal フェーズから処理をやり直す．このように，処理をフェーズに分割し，トランザクションとして定義する区間を短縮することで，一貫性を保証しながら，トランザクション内でアクセスする変数の数を削減することが可能となる．なお，変数の一貫性を検証するために，traversal フェーズおよび calculation フェーズでアクセスした変数を，各スレッドのスレッドローカルな領域に記憶しておく必要がある．以降，本論文では，traversal フェーズおよび calculation フェーズでアクセスし，一貫性を検証する必要がある変数の集合を **read-set** と呼ぶ．validation フェーズの追加および read-set の記憶により，プログラムの処理量およびメモリの消費量は増加するが，トランザクションのアボートを抑制することで，並列度が向上し，プログラムの性能が向上すると考えられる．



(a) 経路探索　(b) パスを発見　(c) ノードを更新

図 9 Labyrinth の動作概要

5.2 Labyrinth の改良

Labyrinth は経路探索を行うプログラムであり，格子状の 3 次元グラフ上で幅優先探索を行い，2 点間の最短パスを求める．このプログラムでは，電子回路の配線を求めるアルゴリズムを実装しており，あるパスで通行したノードは他のパスで通行不可能という制約がある．プログラムの動作概要を**図 9** に示す．図中の S および G はそれぞれ経路探索の開始ノードおよび目標ノードを表し，各数字は開始ノードからそのノードに至るまでの最小コストを表す．また，灰色のノードは通行不可能ノードを表す．図 9(a) のように経路探索が進行し，図 9(b) の矢印で示すパスが見つかった場合，図 9(c) のようにパスに含まれるノードを通行不可能ノードに更新する．

Labyrinth に含まれるトランザクションのうち，経路探索を行うトランザクションのコードを**図 10** に示す．まず，1 行目に示す命令によりトランザクションが開始されると，2 行目で共有変数のグラフを局所変数のグラフにコピーする．その後，3 行目でコピーした局所変数のグラフ上で経路探索を行う．経路探索の結果，パスが見つかった場合，if 文の条件式が真となる．4 行目では，経路探索で見つけたパスをたどり，パスが正しいかどうかを判断する．この処理をバックトラックと呼ぶ．バックトラックの結果，パスが正しいと判断された場合，6 行目の if 文が真となり，7 行目でパスに含まれるノードが通行不可能になったことを共有変数のグラフに書き込む．

このトランザクションでは，グラフをコピーする処理が含まれており，4.2 節で述べたように，トランザクションは全ての場合においてキャパシティアボートするため，ロックにフォールバックされる．したがって，TM を使用してもトランザクションとして定義した区間が排他実行されてしまい，逐次実行と速度が変化せず，TM を使用するオーバヘッドにより速度が低下することもあると考えられる．

そこで，Labyrinth の処理と 5.1 節で述べた 4 つのフェーズとの対応を考え，トランザクションとして定義する区間

Vol.2019-ARC-240 No.7
Vol.2019-SLDM-191 No.7
Vol.2019-EMB-53 No.7
2019/2/27

```
1  BEGIN_TRANSACTION();
2      grid_copy(gridPtr, myGridPtr);
3      if (doExpansion(myGridPtr)) {
4          pointVectorPtr = doTraceback();
5
6          if (pointVectorPtr) {
7              grid_addPath(gridPtr, pointVectorPtr);
8          }
9      }
10 END_TRANSACTION();
```

図 10 Labyrinth のトランザクションを簡略化したコード

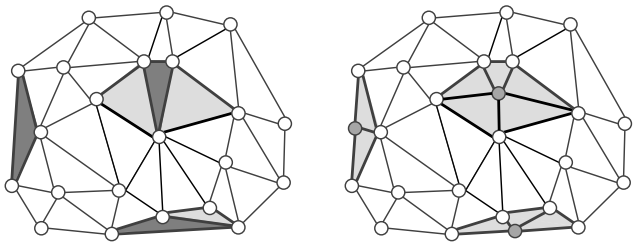
```
1  RETRY:
2      grid_copy(gridPtr, myGridPtr);
3      if (doExpansion(myGridPtr)) {
4          pointVectorPtr = doTraceback();
5
6          if (pointVectorPtr) {
7              BEGIN_TRANSACTION();
8              if (!grid_validation()) {
9                  END_TRANSACTION();
10                 goto RETRY;
11             }
12             grid_addPath(gridPtr, pointVectorPtr);
13             END_TRANSACTION();
14         }
15     }
```

図 11 Labyrinth のトランザクション定義を変更したコード

の短縮を検討する．グラフのコピー，経路探索およびバックトラックでは，処理するサブグラフの範囲を決定し，コストを計算している．次に，ノードを更新する処理では，バックトラックで決定したサブグラフに対応する共有変数を更新している．処理をこのように捉えることで，フェーズの考え方を適用できると考えられる．つまり，グラフのコピー，経路探索およびバックトラックを，traversal フェーズおよび calculation フェーズとみなし，これらのフェーズをトランザクションから除外する．そして，ノードの更新を updating フェーズとみなし，validation フェーズを追加することで，read-set の一貫性を保証する．以上のようにトランザクションとして定義する区間を変更したコードを**図 11** に示す．8 行目が新たに追加した validation フェーズである．validation フェーズでは，パスに含まれるノードが通行不可能ノードになっていないかを検証する．検証に失敗した場合，9 行目の命令でトランザクションをコミットし，10 行目の goto 文で 1 行目に戻ることで，グラフのコピーから処理をやり直す．なお，Labyrinth では見つけたパスが read-set となるため，パスとは別の領域に read-set を記憶しておく必要がない．

5.3 Yada の改良

Yada（Yet Another Delaunay Application）はメッシュ生成を行うプログラムであり，メッシュに含まれる三角形の最小内角が，実行時引数で指定するしきい値よりも大きくなるようにメッシュを再構成する．プログラムの動作概要を**図 12** に示す．しきい値よりも小さい内角を持つ三角形を開始地点として幅優先探索を行い，図 (a) のように，再構成する範囲を決定する．その後，図 (b) のように，新しいノードを追加することで，しきい値よりも小さい内角を持つ三角形を削除し，新しい三角形を生成する．メッシュに含まれる全ての三角形の最小内角がしきい値よりも大きくなるまで，この処理を繰り返す．



(a) 再構成する範囲の決定　(b) メッシュの更新

図 12 Yada の動作概要

```
1  while (!queue.isEmpty()) {
2      elementPtr = queue.pop();
3      isGarbage = elementIsGarbage(elementPtr);
4      if (isGarbage) {
5          continue;
6      }
7      BEGIN_TRANSACTION();
8      growRegion(elementPtr);
9      retriangulate();
10     END_TRANSACTION();
11
12     :
13
14 }
```

図 13 Yada のトランザクションを簡略化したコード

Yada に含まれるトランザクションのうち，メッシュ生成を行うトランザクションを簡略化したコードを**図 13** に示す．まず，2 行目でキューからしきい値よりも小さい内角を持つ三角形を 1 つ取り出す．その後，3 行目で取り出した三角形が削除されていないかどうかを判断する．なぜなら，他のスレッドがメッシュを再構成した際に，自身が取り出した三角形が削除されている可能性があるからである．三角形が削除されていた場合，再びキューから三角形を取り出すため，while 文の先頭に戻る．三角形が削除されていない場合，7 行目の命令によってトランザクションが開始され，8 行目では再構成するメッシュの範囲を決定する．取り出した三角形を開始地点として幅優先探索を行い，周囲の三角形を再構成する範囲に含むかどうかを判断する．範囲を決定した後，9 行目でしきい値よりも小さい内角を持つ三角形を削除し，新しい三角形を生成することで，共有変数のメッシュを更新する．

Vol.2019-ARC-240 No.7
Vol.2019-SLDM-191 No.7
Vol.2019-EMB-53 No.7
2019/2/27

このトランザクションでは，8 行目および 9 行目で範囲の決定および新しい三角形を生成する際に，三角形の頂点が位置する座標を用いて，三角形の外心を計算する必要がある．このとき，各座標を表す変数へのアクセスや，中間変数を多数使用することから，トランザクションをキャパシティアボートする可能性がある．さらに，トランザクション内の処理が長くなる場合，割込みによってトランザクションをアボートする可能性も高くなる．

そこで，Yada の処理と 5.1 節で述べた 4 つのフェーズとの対応を考え，トランザクションとして定義する区間の短縮を検討する．まず，8 行目では，幅優先探索の際に三角形の外心を計算し，再構成するメッシュの範囲を決定するため，この処理は traversal フェーズかつ calculation フェーズであると考えられる．次に，9 行目では，新しい三角形の生成およびメッシュの更新処理が 1 つの関数としてまとめられているため，生成処理と更新処理とを分離し，生成処理を calculation フェーズ，更新処理を updating フェーズとみなす．そして，calculation フェーズと updating フェーズとの間に validation フェーズを追加することで，read-set の一貫性を保証する．以上のようにトランザクションとして定義する区間を変更したコードを**図 14** に示す．9 行目および 15 行目の関数が，図 13 の 9 行目に示す処理を分離したものに相当する．また，11 行目が新たに追加した validation フェーズである．validation フェーズでは，read-set に含まれる三角形が削除されていないかを検証する．read-set に含まれる三角形のうち，どれか 1 つでも削除されていた場合，検証に失敗する．なお，validation フェーズでの検証に失敗した場合，キューから取り出した三角形が削除されている可能性がある．したがって，traversal フェーズに該当する 8 行目から処理をやり直すのではなく，4 行目の取り出した三角形が削除されていないかどうかを判断する処理からやり直す．

```
1 while (!queue.isEmpty()) {
2     elementPtr = queue.remove();
3 RETRY:
4     isGarbage = elementIsGarbage(elementPtr);
5     if (isGarbage) {
6         continue;
7     }
8     growRegion(elementPtr);
9     generate();
10    BEGIN_TRANSACTION();
11    if (!mesh_validation()) {
12        END_TRANSACTION();
13        goto RETRY;
14    }
15    update();
16    END_TRANSACTION();
17
18    :
19
20 }
```

図 14 Yada のトランザクション定義を変更したコード

5.4 評価結果および考察

以上のような変更を Labyrinth および Yada に実装し，3 章と同様の環境を用いて評価した．なお，変更前のプログラムでは，メモリアロケータに glibc の malloc を使用しているが，各スレッドがスレッド間で共通のヒープ領域からメモリを確保するため，トランザクション内で malloc 関数を使用した場合，メモリ確保時に競合が発生し，トランザクションがアボートする可能性がある．このようなメモリアロケータに起因するアボートを抑制するため，関連研究にならい，Google 社が公開している Thread-Caching Malloc（TCMalloc）を使用した．TCMalloc では，各スレッドがスレッド固有の領域からメモリを確保するため，トランザクション内で malloc 関数を使用しても，メモリ確保時に競合が発生しないという利点がある．

評価結果を**図 15** に示す．図では，各プログラムの実行結果を 4 本の折れ線で示しており，Original が 3 章で示した変更前のプログラム，Proposed が変更後のプログラムを表している．なお，図 2 同様，各グラフの縦軸は速度向上率を表し，横軸はスレッド数を表す．

Labyrinth では，スレッド数の増加に伴う速度向上が確認できる．特に，16 スレッドで実行した結果では，Intel Xeon で約 10 倍，POWER9 で約 7.3 倍の速度向上を達成した．この要因として，並列度の向上が考えられる．変更前のプログラムでは，トランザクションとして定義した区間の処理時間が長く，ロックにフォールバックした場合，排他実行する時間が長くなるため，処理を並列に実行できず，逐次実行と速度が変化していなかった．一方，変更後のプログラムでは，トランザクションとして定義した区間を短縮したことで，あるスレッドがロックにフォールバックしても，排他実行する時間が短くなると考えられる．また，ロックにフォールバックした際に，他のスレッドがトランザクションとして定義した区間を実行している可能性が低くなると考えられる．以上のことから，変更後のプログラムは変更前のプログラムよりも並列度が向上し，速度が向上したと考えられる．

また、Yada においても，Labyrinth と同様に速度向上が確認できる．特に，16 スレッドで実行した結果では，Intel Xeon で約 1.23 倍，POWER9 で約 1.36 倍の速度向上を達成した．以上の結果より，処理をフェーズに分割することで，TM の性能を引き出し、プログラムの性能が向上したことを確認した．

6. おわりに

本論文では，TM を使用して性能が向上するプログラムと向上しにくいプログラムがあることを確認した．そして，

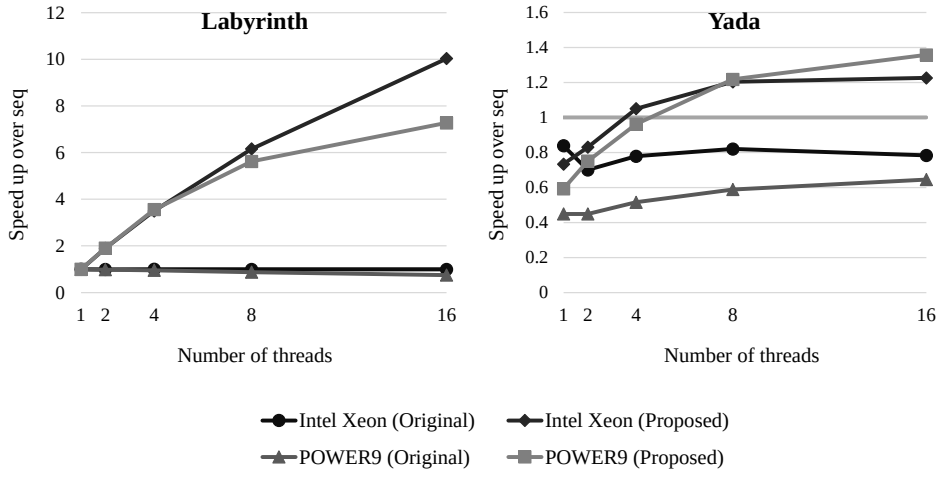


図 15 評価結果

TM を使用して性能が向上しにくいプログラムであるグラフ処理において，TM を適用した場合に起こりうることを考察し，トランザクション内でアクセスする変数の数が多いことに着目した．そこで，グラフ処理を 4 フェーズに分け，一部のフェーズをトランザクションから除外し，トランザクションとして定義する区間を短縮することで，トランザクション内でアクセスする変数の数を削減し，TM の性能をより引き出す方法を検討した．

トランザクションを短く定義したプログラムが性能向上しているかを確認するために，Intel Xeon および POWER9 に搭載されているプロセッサを用いて評価を行った．評価の結果，TM を使用せずに 1 スレッドで逐次実行した場合と比較して，Labyrinth では最大 10 倍，Yada では最大 1.23 倍の速度向上を達成した．

今後の課題として，以下の 2 つが挙げられる．まず 1 つ目の課題は，経路探索およびメッシュ生成以外のグラフ処理にもフェーズの考え方を適用することである．本論文では，フェーズの考え方が経路探索及びメッシュ生成に有効であることを確認できたが，これら以外のグラフ処理を含むプログラムにも有効であるかを評価する必要がある．2 つ目の課題は，グラフ処理を含まないプログラムにもフェーズの考え方を適用できるか検討することである．本論文では，グラフ処理を含むプログラムの改良を試みたが，グラフ処理を含まないが同様の特徴を持つプログラムは存在すると考えられるため，そのようなプログラムに対しても，フェーズの考え方が有効であるかを評価する必要がある．これらの課題を通じて，TM に適したプログラミング技法を体系化し，TM を活用する方法を模索していく．

謝辞 本研究の一部は，JSPS 科研費 JP17H01711，JP17H01764，JP17K19971 の助成を受けたものである．

参考文献

[1] Herlihy, M. et al.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. 20th Int'l Symp. on Computer Architecture (ISCA'93)*, pp. 289–300 (1993).
[2] Knight, T.: An Architecture for Mostly Functional Languages, *Proc. ACM Conference on LISP and Functional Programming (LFP'86)*, pp. 105–112 (1986).
[3] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C. and Olukotun, K.: Transactional Memory Coherence and Consistency, *Proc. 31st Annual Int'l Symp. Computer Architecture (ISCA'04)*, pp. 102–113 (2004).
[4] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D. and Wood, D. A.: LogTM: Log-based Transactional Memory, *Proc. 12th Int'l Symp. on High-Performance Computer Architecture (HPCA'06)*, pp. 254–265 (2006).
[5] Intel Corporation: *Intel Architecture Instruction Set Extensions Programming Reference, Chapter 8: Transactional Synchronization Extensions*. (2012).
[6] Intel Corporation: *Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 1. Chapter 15: Programming with Intel® Transactional Synchronization Extensions* (2015).
[7] International Business Machines Corporation: *IBM System BlueGene Solution BlueGene/Q Application Development*, 1 edition (2012).
[8] International Business Machines Corporation: Power ISA® Version 2.07, `https://www.power.org/documentation/power-isa-version-2-07/` (2013).
[9] Minh, C. C. et al.: STAMP: Stanford Transactional Applications for Multi-Processing, *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)* (2008).
[10] Nakaike, T., Odaira, R., Gaudet, M., Michael, M. M. and Tomari, H.: Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8, *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 144–157 (2015).
[11] Kobayashi, T., Sato, S. and Iwasaki, H.: Efficient use of hardware transactional memory for parallel mesh generation, *2015 44th International Conference on Parallel Processing*, IEEE, pp. 600–609 (2015).