

共有メモリ型TMアプリケーションの 分散環境への移植に基づくDTM記法の評価と改善

小谷 哲平¹ 佐藤 宏樹¹ 藤井 創悟¹ 伊原 楓¹ 小泉 透¹ 塩谷 亮太² 五島 正祐³ 津邑 公暁¹

概要：大規模計算基盤の開発環境では、一般に分散メモリ型と呼ばれるアーキテクチャが採用されているが、生産性の面で課題がある。一方で、共有メモリ環境において性能と生産性を両立し得る並行性制御機構のTM (Transactional Memory) がある。このTMを分散環境向けに導入することで大規模計算基盤における開発環境の課題を解決しようとする考えからDTM (Distributed Transactional Memory) の研究が行われている。しかし、生産性の面でDTMの記述は共有メモリ環境の記述と同程度に簡便なものとは言えない。そこで、TM向けアプリケーションをDTM向けに再実装する過程を通じて得た知見に基づき、DTMでも共有メモリ環境と同等の記述を可能とすることを目指し、現状のDTMの記法の課題を洗い出したうえでその改善案について検討する。

1. はじめに

現代の科学技術の発展を支える大規模計算基盤、特にその開発環境において、分散メモリ型と呼ばれるアーキテクチャから来る、開発の生産性の課題がある。分散メモリ型の並列計算機では、各プロセッサは独立したメモリ空間を持っており、あるプロセッサが別のプロセッサが保有するメモリ上のデータにアクセスする場合、通信が必要である。代表的な規格としてMPIがあるが、MPIを使用したプログラムでは、タスクやデータを計算機へどのようにマッピングするかを細かく指定したり、計算機の同期を明示的に行ったりするなど複雑な記述を行う必要がある。

一方で、マルチコアプロセッサなどの共有メモリ環境においては、ロックと比較して軽量な一貫性制御による性能と簡易な記述による高い生産性を実現し得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] が期待されている。大規模計算基盤における問題を解決するため、TMのソフトウェア実装であるSTM (Software Transactional Memory) と分散共有メモリ (DSM) を組み合わせた**分散トランザクショナルメモリ (Distributed Transactional Memory: DTM)** [2] の研究も行われている。

しかしDTM向けアプリケーションが、共有メモリ環境と同程度に簡便に記述できる状況にあるとは言えない。DTMは仮想的に作成した共有メモリ空間の一貫性制御にSTMを適用するため、ユーザがDTMアプリケーションを記述する際にもまた、STMを利用する際と同等な、簡易な記述と軽量な一貫性制御による恩恵を受けられることが望ましい。しかし、現存するDTM実装を用いた場合、形式的な識別子や明示的なフェッチ処理の明記などがプログラムには求められるため、これは共有メモリ環境と同等に簡便な記述であるとは言い難い。

そこで本稿では、TM向けベンチマークアプリケーションをDTM向けに改変する過程を通じて、DTM向けアプリケーション記述に関する知見を蓄積し、DTMでも共有メモリ環境と同等の記述を可能とすることを目指し、DTM向けの記法について検討する。

2. 研究背景

2.1 従来の大規模計算基盤

一般に大規模計算基盤は、単一のメモリ空間を共有するマルチコア・プロセッサを、高速なネットワークで相互接続した分散メモリ型の構造をとる。このようなアーキテクチャでは、ノード内の並列化にはOpenMP等を使用し、ノード間の通信にはMPI等を使用する、ハイブリッド並列プログラミング [3] を採用することが多い。しかし、このハイブリッド並列は、データの一貫性を保つという並列プログラミングにおける本質的な問題に対して、プログラムが明示的に同期を行うなどして対処する必要があるなど、

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

生産性の観点からは望ましいアプローチではない。

この分散メモリ型に対して、単一のメモリ空間を共有するハードウェアの形態は、共有メモリ型と呼ばれる。共有メモリ型アーキテクチャでは、全てのプロセッサが共有メモリ空間上の同一のメモリアドレスにアクセスすることで、プロセッサ間の通信を暗黙的に行うことができる。そのため、分散メモリ型アーキテクチャを保ちながら、共有メモリ型プログラミングモデルを提供することで性能と生産性を両立するという発想が以前から存在しており、そのような形態のシステムは DSM（分散共有メモリ: Distributed Shared Memory）と呼ばれる。しかし DSM は、コヒーレンスプロトコルのスケラビリティ向上が困難ということが次第に明らかになり 2000 年代にはほとんど研究されなくなった。一方、異なるアプローチから共有メモリ型に近い記述を可能にするものの 1 つとして PGAS (Partitioned Global Address Space) [4] がある。しかし PGAS はただ共有アドレス空間を提供しているに過ぎず、コヒーレンスのための同期処理の明示が必要であるうえ、競合制御の枠組みがないため既存のバリアやロックも必要とし、さらに性能チューニングのためには結局通信の明示が必要であるなど、生産性の問題を十分に解決できてはいない。

2.2 TM

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機構である。TM を使用する場合、クリティカルセクションを含む一連の処理をトランザクション (Tx) として定義し、TM はこれらを投機的に並列実行する。

TM では、以下の 2 つの性質を保証できる限り、トランザクション同士を投機的に並列実行する。

直列化可能性 (serializability) : 並列実行されたトランザクションの実行結果は、当該トランザクションを逐次実行した場合と同一であり、全てのスレッドにおいて同一の順序で実行されたように観測される。

不可分性 (atomicity) : トランザクションはその操作が完全に実行されるか、もしくは全く実行されないかのいずれかである。

以上の性質を保証するために、TM はトランザクション内で発生するメモリアクセスを監視するための機構を持つ。そして、複数のトランザクション内で同一アドレスに対するアクセスが確認され、これらのアクセスによってトランザクションの性質が満たせなくなる場合に、この状態を**競合 (Conflict)**として検出する。この競合検出を行うために、TM はトランザクションがどの変数に対し Read アクセスしているかを記録する。この記録に使用する領域のことを、**Read-set**と呼ぶ。また、トランザクションの性質を保証するためのこの操作を**競合検出 (Conflict Detection)**と呼ぶ。2 つ以上のトランザクションが同一

のメモリアドレスにアクセスしており、その内の少なくとも 1 つのアクセスが write アクセスである場合、write アクセスを行ったスレッド (Writer) もしくは Read アクセスを行ったスレッド (Reader) のどちらかがトランザクションのその時点までの実行結果を破棄する必要がある。この操作を**アボート (Abort)**という。トランザクションをアボートしたスレッドは、トランザクション開始前のメモリの状態を復元し、トランザクションを再実行する。一方、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する。この操作を**コミット (Commit)**という。

なお、トランザクションをアボートした際にトランザクション開始前の状態を復元するためには、トランザクション内で更新したデータと更新する前のデータの両方を保持しておく必要がある。そこで、TM ではトランザクション内で更新したデータ、もしくは更新前のデータをそのアドレスとともに別領域に退避する。この退避領域のことを、**Write-set**と呼ぶ。また、このようなデータの管理を**バージョン管理 (Version Management)**という。

このような競合検出は、アクセス競合が発生したか否かを確認するタイミングによって以下の 2 つに大別される。

Eager Conflict Detection (EagerCD) : トランザクション内でメモリアクセスが発生する度に、そのアクセスにより競合が発生するかどうかを確認する。

Lazy Conflict Detection (LazyCD) : トランザクションのコミットを試みる時点で、そのトランザクション内で行われた全てのアクセスに関して競合が発生しているかどうかを確認する。

TM では、共有変数に対する競合を検出しない限り、トランザクションが高並列に実行され、ユーザが共有変数へのアクセスを含む一連の処理をトランザクションとして定義するだけで並列処理での共有変数の一貫性を保証できる。このように、TM は簡易な記述による生産性と軽量な一貫性制御による性能を両立し得る並列性制御機構として期待されている。

2.3 DTM

マルチコアプロセッサ向けに研究されてきた TM を分散システムに適用することで、分散メモリ型の開発環境におけるプログラミングを容易にすることを目的として、分散トランザクショナルメモリ (DTM) の研究が行われている。

DTM はノード間の通信を暗黙的に行うことで、ユーザに仮想的な共有データストレージを提供し、ストレージの一貫性制御に TM の仕組みを活用することで、軽量な一貫性制御と簡易な記述による性能と生産性を両立したシステムの実現が期待される。ハードウェア TM (HTM) や STM の多くは、ワードを競合検出単位とするワード

ベース実装となっているのに対し、代表的な DTM である HyflowCPP [5] や当研究室で研究している DTM [6] など多くの DTM は、オブジェクトベースの実装を採用している。すなわち DTM において、仮想的な共有データストレージでは、共有データをオブジェクトとして定義し、オブジェクト単位で共有データを管理する。

なお本研究が対象とする DTM は、物理的な実体ノードを固定せず、データがキャッシュとしてノード間を自由に移動・複製される COMA (Cache-Only Memory Architecture) 的な構造を想定している。共有データの保持やノード間の通信などの操作をオブジェクト単位とすることで、無関係なデータによって偽の競合が検出されることを防いだり、意味のあるデータを一括で転送し、無駄な通信を削減したりできる利点がある。また、DTM の多くがオブジェクト指向のプログラミング言語で実装されており、プログラミング言語との親和性が高いことも、オブジェクトベースが採用される要因となっている。

3. TM アプリケーションの分散環境への移植

DTM の課題の一つとしてデファクトスタンダードなベンチマークの不在がある。そのため、現状の DTM 評価では、多様な実アプリケーションを用いた性能評価を行うことは難しい。このような状況では、異なるアクセスパターンや通信特性を持つアプリケーションに対する性能評価を十分行うことができず、提案された DTM が抱えるボトルネックや設計上の欠点を発見しづらくなるという問題がある。また、評価に用いるアプリケーションが特定の性能特性に偏ることは、提案された DTM の利点によって過小評価や過大評価を招く原因となり得る。

そこで、既存の単一ノードの TM 向けベンチマークである PANDORA [7] が持つアプリケーション群を DTM 向けに移植することでベンチマークの開発を試みた。PANDORA は様々な Tx 特性を持つ多様なアプリケーションで構成されており、同期なしでは容易に並列化できないため、TM による簡易な記述による恩恵を受けやすく、DTM の利点を活かせるアプリケーションである。本稿ではまず、そのうちの kmeans アプリケーションを対象として移植を行った。

移植の際に、主に共有データのオブジェクト化、オブジェクトへのアクセス方法の変更、ノード間の同期の記述の 3 点を行った。

- **共有データのオブジェクト化**：DTM がオブジェクトベースであることから、共有データをオブジェクトとして定義する。
- **オブジェクトへのアクセス方法の変更**：オブジェクトは仮想的なデータストレージで保管されており、ここへのアクセスは DTM が提供する関数やマクロを用いて行うため、オブジェクト化した共有データへのアクセス処理は DTM が提供する機能を用いて行うように

変更する。

- **ノード間の同期の記述**：分散環境では、自ノードのスレッドだけでなく、他ノードのスレッドの終了も待つ必要があるため、実装するアプリケーションによって適当な位置にノード間の同期を記述する必要がある。

上記の手順に従い kmeans を移植した結果、共有メモリ向け TM コード (図 2) と DTM コード (図 1) を比較したとき、後者が著しく冗長であることが判明した。具体的には、オブジェクト ID の生成 (図 1 の 4~6 行目) とフェッチ関数の明示的な呼び出し (8~9 行目) は、対応する STM コード (図 2 の 3~5 行目) には存在せず、記述の煩雑さの主因となっていた。加えて、DTM ライブラリのソースコードにユーザ固有の型名を直接記述しなければならないという、インターフェース上の問題も確認された。次章ではこれらの問題点を体系的に分析する。

4. 既存 DTM インターフェースの問題点

DTM は仮想的な共有データストレージを提供することで、分散メモリ型のアーキテクチャを保ちながら共有メモリ型のプログラミングモデルを提供し、分散メモリ型の開発環境におけるプログラミングを単純にすることを目的としている。したがって、DTM の記述は共有メモリ環境と同等であることが求められる。

しかし、3 章の移植作業を通じて、既存の DTM が前提とするアプリケーション記法には共有メモリ空間におけるそれと比較して以下の 4 つの問題点があることが明らかになった。

- (a) フェッチ処理の明示が必要
 - (b) 形式的なオブジェクト ID の使用
 - (c) Tx 区間外でのオブジェクトアクセス不可
 - (d) DTM ライブラリへのユーザ型名の直接記述 (密結合)
- (a)(b)(c) は仮想的な共有データストレージへのアクセス方法に関する問題であり、(d) はアプリケーションと DTM 間の連携方法に関する問題である。以下、それぞれを詳述する。なお、5 章で提案する改善はこれらに一对一に対応する。

4.1 仮想的な共有データストレージへのアクセス方法

仮想的な共有データストレージへのアクセスには、一般に DTM が提供する専用の関数を用いる。

図 1 は HyflowCPP を使って K-means プログラムを実装した際のコード片である。このコードではローカルで集計した結果をオブジェクトに反映させる処理をしている。まず、4~6 行目でアクセスしたいオブジェクトの ID を生成し、8~9 行目にある通り、アクセスしたいオブジェクトに該当する ID を writeobject 関数の引数として呼び出し、当該オブジェクトのコピーを取得している。ここで writeobject は、「書き込み用にオブジェクト

```

1 for(i=0; i<nclusters; i++){
2   HYFLOW_ATOMIC_START{
3     //ID 文字列の生成
4     std::ostringstream idStream_len;
5     idStream_len << i%nodeCount
6     << "-new_centers_len-" << i;
7     //更新処理
8     Int_value *new_cen_l
9     = writeobject<Int_value>(idStream_len.str());
10    new_cen_l->value
11    = new_cen_l->value + movecounts[i];
12  }HYFLOW_ATOMIC_END
13 }

```

図 1: HyflowCPP を使ったオブジェクトの更新コード

```

1 TM_BEGIN();
2   for (i = 0; i < nclusters; i++){
3     TM_SHARED_WRITE(*new_centers_len[i],
4     TM_SHARED_READ(*new_centers_len[i])
5     + movecounts[i]);
6   }
7 TM_END();

```

図 2: 図 1 に対応する TM 向けベンチマークのコード

```

1 // 共有オブジェクトの宣言
2 // Tracker 割当てと ID 生成が DTM 内部で行われる
3 ArrayObject new_centers_len[nclusters];
4
5 //途中の処理
6 ...
7
8 //更新処理
9 for(i=0; i<nclusters; i++){
10   ATOMIC_START{
11     //更新処理
12     Write_Object(new_centers_len[i],
13     Read_Object(new_centers_len[i]) +
14     movecounts[i]);
15   }ATOMIC_END
16 }

```

図 3: 望ましいと考えられるコード

をフェッチする」という動作をする, HyflowCPP 提供の関数である. HyflowCPP ではユーザは, 読み出し・書き込みを問わず, このような形で事前にローカルコピーを作成する操作が求められる. 最後に 10~11 行目で, 取得したオブジェクトコピーのメンバである value を更新している. この一連の処理を HYFLOW_ATOMIC_START と HYFLOW_ATOMIC_END で囲むことでトランザクションとして定義している.

HyflowCPP が提供するオブジェクトフェッチ関数 read-object/writeobject の内部処理を図 4 に示す. この図では, オブジェクトへアクセスするノードを AccessNode とし, ReadSet/WriteSet に対象のオブジェクトが登録されていない場合の処理を表している.

1. トランザクションの ReadSet/WriteSet に, アクセス対象のオブジェクトが登録済みかを確認する. 登録済

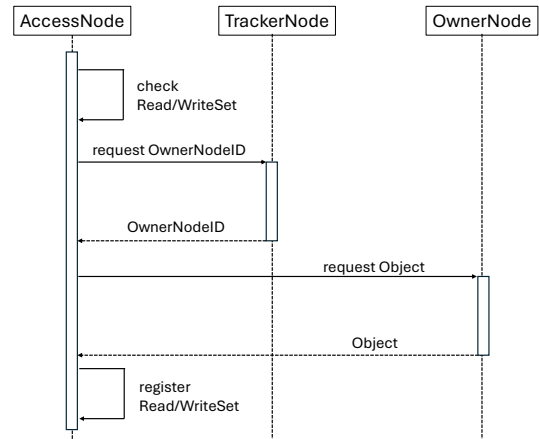


図 4: オブジェクトミス時のシーケンス図. Tracker/Owner への通信が支配的オーバヘッドとなる.

みであればフェッチ不要と判断し, そのオブジェクトへの参照を返す.

2. 未登録の場合, オブジェクトの所有権を管理するノード (以下, TrackerNode と呼ぶ) に所有者を問い合わせる. TrackerNode が自ノードの場合ローカル参照, 他ノードの場合はノード間通信が発生する.
3. 取得した所有者ノード (以下, OwnerNode と呼ぶ) に対し, オブジェクトの転送リクエストを送信する.
4. OwnerNode からオブジェクトのコピーを取得し, アクセス種別に応じて ReadSet もしくは WriteSet に登録する. この時, OwnerNode が他ノードであった場合にはノード間で通信が発生する. 最後に, 取得したオブジェクトのコピーに対する参照を返値として返す.

このフェッチ機構を前提として, 問題 (a)(b)(c) を順に説明する.

フェッチ処理の明示

共有メモリ環境向け STM は一般に, 図 2 に示すように, read 操作や write 操作をトランザクショナルに行うための関数を提供している. 4 行目に示す関数はトランザクショナルな read 操作を行う関数で共有変数 new_centers_len[i] の値を返す. また 3 行目にある関数はトランザクショナルな write 操作を行う関数で共有変数 new_centers_len[i] の値を更新する. このように, いずれもフェッチは関数内部で透過的に実行される.

一方, HyflowCPP では, ユーザが「その後読み書きする予定のオブジェクト」を事前にフェッチし, 得られたローカルコピーに対して操作を加えるという 2 段階の手順が必須となる (図 1 の 8~11 行目). 「オブジェクトのローカルコピーを取得する」といった分散環境を意識した操作は, 仮想的な共有データストレージが提供されている環境において直感的ではなく, 生産性を損なう. すなわち, ユーザの read/write アクセス操作に応じて, DTM システムが

透過的にオブジェクトのローカルコピー取得を行うべきである。

形式的なオブジェクト ID の使用

HyflowCPP はユーザに対し、「TrackerNodeID-name」という形式のオブジェクト ID を用いてオブジェクトを指示させる実装となっている。つまりユーザは、各オブジェクトの TrackerNode を意識する必要があり、これもまた共有データストレージによる仮想化の恩恵を大部分毀損するものである。

一方で共有メモリ環境では、ユーザは変数名を識別子として共有データにアクセス可能である(図 2 の 3, 4 行目)。DTM においては、オブジェクトインスタンス名が、共有メモリ環境における変数名に相当するものであり、オブジェクト ID は共有メモリ環境におけるメモリアドレスに相当するものと見なせる、後者を用いたアクセスのみ許される環境は、共有メモリプログラミングの抽象性を損なっており、生産性が高いとはいえない。

したがって、DTM でもユーザが使用する識別子を形式化されたオブジェクト ID ではなく自由に設定できる変数名とすべきである。この識別子から TrackerNode を特定する仕組みは DTM システム側で隠蔽して実現する(具体的なメカニズムは 5.3 節で後述する)。

この 2 つの問題 (a)(b) を解決した、DTM における望ましいコード例として図 3 を示す。2 行目に示すような、DTM の管理対象とするオブジェクト宣言により、DTM 内部で Tracker の割当て、ID の生成を行う。宣言されたオブジェクトインスタンスと、その Tracker/ID 情報との対応は DTM が管理し、以降はユーザがインスタンス名を用いて read/write 関数を呼び出すだけで、DTM が透過的にフェッチを解決する。

Tx 区間内に限定されたアクセス

もうひとつの既存 DTM の課題として、オブジェクトへのアクセスがトランザクションの区間内に限定されることがある。DTM が提供している関数を使用してオブジェクトをフェッチしたうえで、そのオブジェクトに対してアクセスする仕様となっているが、この関数はトランザクションとして定義された区間内でしか使用できない。

一方で共有メモリ環境における STM では、共有変数へのアクセスを含む処理をトランザクションとして定義することで共有変数の一貫性を保証している。つまり、共有変数へのアクセスを含む処理で一貫性制御が必要な場合にのみ一連の処理をトランザクションとして定義する。そのため、逐次処理や一切更新しない共有変数へのアクセスなどといった、一貫性制御の必要がない場面では、ユーザはそれらを含む処理をトランザクションとして定義する必要がない。

このように、共有データへのアクセスには競合検出のための検証操作をせずとも一貫性を保証しながらアクセスで

きる場合があり、共有メモリ環境における STM と同様に、DTM においても一貫性制御の不要な場面ではトランザクション区間外でもオブジェクトへアクセスできる柔軟性が必要である。この問題に対する具体的な解決策は 5.2 節で述べる。

4.2 アプリケーションと DTM の連携

HyflowCPP は、プログラマが定義・設定する情報を活用し、実行中に得られた情報をアプリケーション側に提供できる実装となっている。このような情報には、主に以下の 3 つがある。

パラメータ: DTM の詳細設定を行う変数や、アボート数・実行時間といったパフォーマンスカウンタが含まれる。これらはシステムのチューニングやアプリケーションの性能向上に活用可能な情報である。

パラメータに関わる関数: パフォーマンスカウンタの更新など、DTM 内部で呼び出される制御用関数である。

型登録の関数: オブジェクトとバイナリデータの相互変換(シリアライズ・デシリアライズ)を行うための関数である。メッセージの送受信直前に DTM 内部で呼び出されるが、対象となるオブジェクトの型はユーザが定義するため、その登録処理もアプリケーション側で行う必要がある。

問題 (d) は、この型登録関数の伝達方法に起因する。現状の実装では、DTM のライブラリ内に、アプリケーション側で定義された具体的な型名を直接記述することが想定されている。図 5 はオブジェクトのアクセスに関するメッセージをバイナリデータにシリアライズする、HyflowCPP 内部のコードを示している。この 4 行目を見ると、アプリケーション側で型登録のために定義された特定のクラス名や関数名が直接呼び出されていることがわかる。

```
1 void Messageserialize(Archive ar) {  
2     ar & base_object<BaseMessage>;  
3     // Register object pointers  
4     //型登録のための関数  
5     Userclass::registerObjectTypes(ar);  
6     //オブジェクト情報をシリアライズ  
7     ar & object;  
8     ar & id;  
9     ar & isRead;  
10 }
```

図 5: DTM で定義される関数のコード

このような実装では、プログラマが DTM を利用するたびにライブラリ自体のソースコードの一部を書き換える必要があり、インターフェースとして極めて不適切である。結果として、アプリケーションと DTM の境界が、曖昧な密結合の状態となっており、利用にあたって DTM ライブラリ内部の詳細な理解と修正を強いることが、開発生産性

を著しく低下させている．この問題に対する解決策は5.4節で述べる．

5. DTM インターフェースの改善

4章で述べた問題点から，オブジェクトに対する透過的なアクセス関数，Tx 区間外からのオブジェクトへのアクセス関数，プログラマからのオブジェクト ID の隠蔽を検討する．さらに，新たにアプリケーションの基底クラスも検討する．

```
1 //更新処理
2 for(i=0; i<nclusters; i++){
3     ATOMIC_START{
4         //更新処理
5         Write_Object(new_centers.len[i],value,
6                     Read_Object(new_centers.len[i],value
7                     ) + movecounts[i]);
8     }ATOMIC_END
9 }
```

図 6: DTM インターフェース改善後のコード

5.1 アクセス透過性を実現する関数

4.1 節で述べたように，従来の DTM では共有データへのアクセスに際し，プログラマが明示的にフェッチ処理（図 1 の 8, 9 行目）を記述しなければならないという課題があった（問題 (a)）．そこで本節では，フェッチ処理を DTM 内部に隠蔽し，共有メモリ環境における TM と同等の記述性を実現する透過的なアクセス関数を提案する．

提案するインターフェースでは，プログラマの要求（オブジェクト全体へのアクセスか，特定のメンバへのアクセスか）に応じ，表 1 に示す 4 種の関数を定義する．これらはトランザクション（Tx）区間内での使用を想定したものであり，内部で 3.1 節に示したフェッチ手順（レジスタ確認，Tracker への問い合わせ，Owner からのコピー取得）を自動的に実行する．

表 1: 提案するオブジェクトアクセス関数 (Tx 区間内)

機能 (API の役割)	引数
オブジェクト全体の取得	オブジェクトの識別子
特定のメンバの取得	オブジェクトの識別子，メンバ名
オブジェクト全体の更新	オブジェクトの識別子，更新データ
特定のメンバの更新	オブジェクトの識別子，メンバ名，更新データ

まず，オブジェクト全体を取得する関数は，引数に指定された識別子に該当するオブジェクト全体のコピーを返す．これは実質的に従来のフェッチ関数（read_object）と同じ処理を内部で行うが，戻り値をそのまま操作対象とできるため，記述が簡潔になる．

次に，オブジェクトの特定のメンバを取得する関数は，

フェッチ完了後に指定されたメンバのみを抽出して返す．各オブジェクトの内部構造はアプリケーションに依存するため，本関数の実現には，プログラマがメンバ名から該当データを判別する補助関数を定義する仕組みを導入する．この処理はノード間通信を伴うフェッチ処理と比較して極めて低コストであり，既存の実装と同等の性能を維持しつつ，より直感的な read 操作が可能となる．

オブジェクト全体を更新する関数は，識別子で指定されたオブジェクトをフェッチした後，引数で与えられたデータでその内容を完全に置き換える．図 1 の 8～9 行目でプログラマが手動で行っていた更新処理を DTM 側で肩代わりするものであり，性能を損なうことなく「書き込み」という意図を直接的に記述できる．

最後に，オブジェクトの特定のメンバだけを更新する関数は，フェッチ後に指定されたメンバのみを書き換える．前述のメンバ取得関数と同様，特定のメンバを操作するためのオーバーヘッドは微小であり，ノード間通信のコストが支配的な分散環境においては，性能への影響は極めて小さいと考えられる．

図 6 は図 1 の処理に DTM の改善を行った後のコードである．5,6 行目で，提案した関数で，特定のメンバだけを更新する関数を呼び出してオブジェクトを更新している．図 1 や図 2 と比較して，明示的なフェッチ処理を記述せず，共有メモリ環境に近い形式でアクセスすることが可能である．

これら 4 つの関数により，プログラマはフェッチという分散環境特有のオーバーヘッドを意識することなく，共有メモリ環境に近い形式でオブジェクトへのアクセスを記述できるようになる．従来の手動操作を DTM 内部に集約したことで，開発生産性を大幅に向上させつつ，実行性能を維持することが可能である．

5.2 Tx 区間外からのアクセス関数

4.1 節で述べたように，従来の DTM では共有データへのアクセスが Tx 区間内に限定されており，一貫性制御が不要な場面でも Tx を定義しなければならないという制約があった（問題 (c)）．この課題を解決するため，新たに Tx 区間外から共有データストレージへアクセスするための関数を定義する．共有メモリ環境と同等の記述性を実現するため，本節で提案する関数も 5.1 節と同様に，オブジェクト全体および特定のメンバを対象とした 4 つの形式をとる．

Tx 区間外アクセスのための関数内では，以下の手順で処理を行う：

- 4.1 節で述べた手順と同様に，対象オブジェクトのコピーを取得する．
- a. read 操作の場合，取得したオブジェクトのコピーから，対象のデータだけを抽出して返す．
- b. write 操作の場合は，オブジェクト内の対象のデータ

を、引数で与えられた内容に更新し、自ノードのグローバル領域に保存する。その後、所有者が更新されたことを TrackerNode に通達し、TrackerNode 側で所有者情報を更新する。

これらの関数と、5.1 節で述べた関数との違いは主に 2 点ある。1 つ目は、取得したオブジェクトのコピーを ReadSet や WriteSet に登録しない点である。ReadSet や WriteSet にあるデータは一貫性検証やアボート時のロールバックに使用されるが、Tx 区間外のアクセスは一貫性制御を必要としない処理を前提としているため、これらへの登録を省略できる。2 つ目は、write 操作の場合において、更新内容の自ノード領域への保存と TrackerNode への通知を関数内で実行する点である。Tx 区間内の関数では、これらの処理をトランザクション終了時のコミット処理に一任している。しかし、Tx 区間外では、コミット処理が介在しないため、関数内でこれらの処理を完結させる必要がある。TrackerNode への通達の直前には、後述するハッシュ計算をすることで識別子から TrackerNode を導出する。

これらの関数の提供によって、DTM はプログラマに対し、厳密な一貫性を必要としないオブジェクトへの高速なアクセスを提供できる。Tx の定義や競合検出のための検証走査をスキップすることができるため、パフォーマンスの最適化が期待できる。さらに、5.3 節から write 操作時に発生するハッシュ計算コストは通信コストと比較して無視できるほど小さく、関数のパフォーマンスへの影響は非常に少ない。ただし、この関数を利用する際には、対象となるオブジェクトへのアクセスに一貫性制御が本当に不要であるかをプログラマ自身が吟味する必要がある、開発生産性とのトレードオフとなる。

5.3 位置透過性を実現する識別子とハッシュベースの管理ノード特定メカニズム

4.1 節で述べたように、本研究ではプログラマが共有メモリ空間の変数のように自由な名称をオブジェクトの識別子（インスタンス名）として設定できる環境を目指している（問題 (b) への対応）。そのためには、識別子自体に管理ノード（TrackerNode）の情報が含まれていなくとも、システム内部で一意に TrackerNode を特定できる仕組みが必要になる。

これらを実現する内部メカニズムとして、本稿では (i) ハッシュ関数を導入して識別子から TrackerNodeID を導出する方法と、(ii) 所有権情報を一括管理する専用ノードを用意する方法の 2 つを比較検討する。

(i) ハッシュ関数を用いた方法では、プログラマが設定した識別子（文字列等）をハッシュ関数にかけることで TrackerNodeID を生成し、各オブジェクトの Owner 情報を当該ノードで管理する。オブジェクトにアクセスする際、DTM システムはプログラマから与えられた識別子を

内部でハッシュ計算し、導出された TrackerNode に所有者を問い合わせる。さらに、4.1 節で述べた Tx 区間外アクセスのための関数内の手順 2-b で、TrackerNode 側が所有者情報を更新する際の問い合わせでも、ハッシュ計算による TrackerNode の導出を行う。この方法の利点は、適切なハッシュ関数を用いることでオブジェクトの管理責任を全ノードに分散でき、特定のノードへのメッセージ集中を回避できる点にある。一方で、システム内部で識別子から ID を導出するための計算コストが追加される点がデメリットとなる。

(ii) 専用ノードを用意する方法では、すべてのオブジェクトの所有権管理（Tracker 機能）を特定の 1 ノードに固定する。すべての識別子に対して TrackerNodeID が固定されるため、計算コストなしに問い合わせ先を決定できるメリットがある。しかし、オブジェクトへのアクセス時には必ず当該ノードへの通信が発生するため、ノード数が増加するにつれて特定のノードに通信とデータが集中し、システム全体のボトルネックとなるリスクがある。

これら 2 つの方法を比較すると、スケーラビリティの観点から (i) ハッシュ関数を用いた方法が優れていると考える。(ii) 専用ノード方式では、単位時間あたりの通信量がノード数に比例して増加し、高負荷に耐えうる特殊な設計が求められる。対してハッシュ関数を用いた場合、リクエストが各ノードに分散されるため、計算資源を有効に活用できる。

性能面におけるハッシュ計算のオーバーヘッドについては、導入するアルゴリズムにもよるが、先行研究 [8] によると、一般的に用いられる SHA-256 や SHA-1 のようなハッシュ関数を用いたハッシュ値の計算コストは約 5~20cycles/byte である。このことから、現代のプロセッサを用いて一般的な長さの変数名（5~30 文字）をハッシュ関数にかけると、約数十ナノ秒から数百ナノ秒の範囲でハッシュ値が計算される考えられ、分散環境におけるノード間通信の遅延（数十マイクロ秒から数百マイクロ秒）と比較すれば、この計算コストは十分に無視できるほど小さい。

具体的に通信が発生する確率を $(N-1)/N$ （ N はノード数）とすると、通信による平均レイテンシはマイクロ秒オーダーとなる。ノード数が複数ある実行環境においては、通信遅延の影響が支配的であり、ハッシュ計算によるわずかな遅延はシステム全体の性能を大きく損なうものではない。

したがって、生産性の向上（位置透過性の実現）と性能の維持を両立する手段として、ハッシュ関数による所在解決メカニズムが最も望ましいと考える。

図 7 はハッシュベースの所在解決機構の概念図を表す。各ノードでは識別子と対応する OwnerNodeID をマップに格納しており、オブジェクトへアクセスする際は、識別子を各ノードが持つハッシュ関数にかけて、対応する

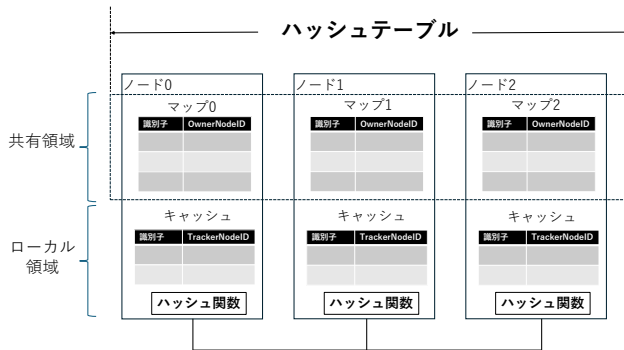


図 7: ハッシュベースの位置等価な所在解決機構

OwnerNodeID を持つノードを特定する。さらに、一度解決した TrackerNodeID を識別子と共にキャッシュで保持することで、ハッシュ関数にかけることなく TrackerNodeID を特定できる。

5.4 アプリケーションの基底クラスによる DTM とユーザの疎結合化

4.2 節で明らかになった問題 (d) から、ユーザが定義する具体的な型名を DTM ライブラリが関知することなく実行できるように設計する必要がある。また、DTM とアプリケーション間で伝達される情報は、ユーザビリティや拡張性の観点から一つのクラスで一元管理されることが望ましい。

そこで、アプリケーションの基底クラスを導入し、これが DTM とユーザプログラム間の情報の伝達を仲介する役割を担う構成とする。類似の手法としてテンプレートによる実装も考えられるが、テンプレートには動的なプログラムの切り替えが不可能であったり、DTM の内部実装を完全に隠蔽することが困難であるというデメリットがある。DTM がユーザの利用状況を制限することは生産性の面から望ましくないため、本研究ではより汎用性の高い基底クラスを選択する。

アプリケーションの基底クラスには、4.2 節で述べたパラメータ、パラメータに関わる関数、型登録の関数の 3 つをメンバとして定義する。本クラスは DTM フレームワークの一部として提供され、ユーザはこのクラスを継承して利用する仕組みとする。パラメータを基底クラスのメンバ変数として定義することで、ユーザが設定した値を DTM 側が基底クラスを介して参照可能となる。型登録の関数は基底クラスにおいて純粋仮想関数として定義する。これにより、ユーザに特定の型情報の定義を強制しつつ、DTM からは純粋仮想関数を介してユーザが定義したコンテキストを呼び出すことが可能になる。図 8 は DTM システム内のクラスが基底クラスを介してユーザが定義した関数を呼び出す流れを示す。まず、DTM システムは基底クラスにある static な関数 RegisterTypes を呼び出す。その関数

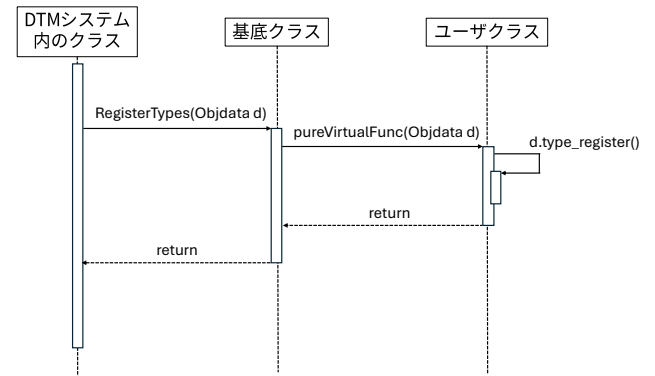


図 8: 型登録の関数を基底クラスを介して呼び出す際のシーケンス図。

の中で、純粋仮想関数を呼び出し、実際にはユーザクラスでオーバーライドした関数が実行される。基底クラスには static な基底クラスのインスタンスが定義されており、この関数の純粋仮想関数を呼び出している。

このように、アプリケーションの基底クラスを追加することで、図 5 に示したような、ユーザのクラス名を DTM 内部に直接記述する密結合な関係を防ぎ、両者の疎結合化を実現する。これにより、ライブラリ自体の書き換えを不要とし、生産性の高い開発環境をプログラマに提供できる。

6. ユーザのためのプログラミング指針

5 章で提案したインターフェース改善により、プログラマはフェッチや形式的なオブジェクト ID を意識することなく DTM アプリケーションを記述できるようになる。しかし、最終的な実行性能に関しては依然としてユーザによるデータ設計に依存する部分が残る。特に、オブジェクトベースの DTM においては、オブジェクトの定義そのものが通信回数や競合率に大きく影響する。本章では、5 章の改善を前提としたうえで、さらに性能を引き出すために、DTM を使用するプログラマが考慮すべきオブジェクト設計の指針について検討する。

6.1 データの性質に基づいたオブジェクト化の判断

分散環境において性能が悪化する主な原因の一つは通信のオーバーヘッドであり、DTM では主にオブジェクトの転送時に通信が発生する。したがって、全てのデータを一律に共有オブジェクト化するのではなく、その性質を吟味したうえでオブジェクト数を適切に絞り込むことが重要である。

具体的には、実行中に一度も更新されないデータや、更新されても他ノードの実行結果に影響を及ぼさないデータについては、DTM の管理対象（オブジェクト）とする必要はない。これらのデータをノードローカルな変数として

保持することで、不要な競合検出や一貫性維持のための通信を抑制できる。ただし、サイズの大きいデータに関しては、メモリ負荷の分散を目的として例外的にオブジェクト化し、複数ノードに分割して保持する等の検討が必要となる場合がある。

6.2 オブジェクトの粒度の最適化

オブジェクトの粒度、すなわちデータの分割単位は、競合率と通信効率のトレードオフを決定する重要な設計項目である。

細粒度なオブジェクト定義、例えば、配列全体ではなく各要素を個別のオブジェクトとした場合、異なる要素への同時アクセスによる偽の競合を抑制し、並列性を向上させる効果がある。しかしその反面、本来一括で転送できたはずのデータが細分化されるため、オブジェクトのフェッチに伴う通信回数が増大し、性能を損なう可能性がある。

したがってプログラマは、アプリケーションのアクセスパターンを考慮し、競合の抑制と通信回数の削減が両立する最適な粒度を選択する必要がある。

6.3 オブジェクトの初期配置戦略

COMA 的構造を持つ DTM において、アクセスするオブジェクトが自ノードにキャッシュされている状態を作ること、通信遅延を排除するための有効な手段である。

しかし、DTM はノード間の通信をユーザから隠蔽しており、ユーザが実行中のオブジェクトの配置を把握することは困難である。そのため、ユーザはオブジェクトの初期配置を工夫する。5.3 節で述べたように、DTM は動的なオブジェクト配置をプログラマから隠蔽しているため、実行中の配置を微細に制御することは困難である。しかし、オブジェクトの「初期配置」を工夫することで、定常状態における通信負荷を最適化できる。

具体的な戦略として以下の2つが挙げられる。1つ目は、オブジェクトにアクセスするノードが限定される場合、初期配置をその限定されたノードにすることである。これにより初回アクセス以降のノード間通信を最小限に抑えることができる。2つ目は、アクセスするノードが限定できないオブジェクト群を分散配置させる方法である。これにより、アクセスに伴う通信が特定のノードに集中することを防ぐ。

このような初期配置の最適化にはアプリケーションの特性に対する深い理解が求められ、開発生産性とはトレードオフの関係にあるが、性能を極限まで追求する場合には極めて重要な手段となる。

7. おわりに

本稿では、大規模計算基盤において生産性と性能を両立した実用的な開発環境が未だ確立していないことを指

摘し、その解決策として分散トランザクショナルメモリ (DTM) に着目した。既存の DTM 実装を詳細に検討した結果、フェッチ処理の明示的な記述や、形式的なオブジェクト ID による制約、さらに、アプリケーションと DTM 間の密結合な連携が、開発生産性を著しく損なう要因となっていることを指摘した。

これらの課題に対し、本稿ではフェッチ処理を内部に隠蔽した透過的アクセス関数の定義、トランザクション区間外からのアクセス機能の追加、ハッシュ関数を用いた位置透過なオブジェクト識別子メカニズム、および、アプリケーション基底クラスの導入によるユーザプログラムと DTM の疎結合化を提案した。これらの改善案によって、プログラマは共有メモリ環境における STM と同等の直感的な記述が可能となる。性能面においても、提案手法に伴うハッシュ計算や仮想関数呼び出しオーバーヘッドは、ノード間通信の遅延が支配的な分散環境においては十分に無視できる範囲に留まることを定性的に示した。さらに、kmeans の移植を通じて得た知見に基づき、データの性質に応じたオブジェクト化の是非の判断や、競合と通信のトレードオフを考慮した粒度設計、オブジェクト初期配置による最適化といったプログラミング指針を提示した。

今後の展望としては、検討した改善案を実際の DTM システムへ段階的に実装し、既存実装との詳細な性能比較を行う。加えて、提案手法が開発生産性の向上に与える影響を定量的に評価していきたい。

謝辞 本研究の一部は JSPS 科研費 21H03408, 23K21652, 25K03093 の助成を受けたものである。

参考文献

- [1] Herlihy, M. and Moss, J. E. B.: Transactional memory: Architectural support for lock-free data structures, *Proceedings of the 20th annual international symposium on Computer architecture*, pp. 289–300 (1993).
- [2] Manassiev, K., Mihailescu, M. and Amza, C.: Exploiting Distributed Version Concurrency in a Transactional Memory Cluster, *Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP’06)*, pp. 198–208 (2006).
- [3] Rabenseifner, R., Hager, G. and Jost, G.: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes, *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 427–436 (2009).
- [4] PGAS Forum: PGAS Partitioned Global Address Space, <http://www.pgas.org/>.
- [5] Mishra, S., Turcu, A., Palmieri, R. and Ravindran, B.: HyflowCPP: A Distributed Transactional Memory Framework for C++, *IEEE 12th Int’l Symp. on Network Computing and Applications*, pp. 219–226 (2013).
- [6] 伊原 楓, 酒井駿輔, 佐藤宏樹, 藤井創悟, 小泉 透, 塩谷亮太, 五島正裕, 津邑公暁: ソフトウェアキャッシュを備えた同期クロック不要な分散トランザクショナルメモリ, *情処研報*, Vol. 2025-ARC-261, No. 31, pp. 1–12 (2025).
- [7] 佐藤宏樹, 酒井駿輔, 伊原 楓, 小泉 透, 塩谷亮太, 五

島正裕, 津邑公暁: トランザクショナルメモリ性能を正確に評価可能なベンチマークスイートの開発, *Information Processing Society of Japan* (2023).

- [8] Gueron, S.: Speeding Up SHA-1, SHA-256 and SHA-512 on the 2nd Generation Intel Core™ Processors, *2012 Ninth International Conference on Information Technology - New Generations* (2012).