Vol.2015-HPC-150 No.5
2015/8/4

再配置可能な大域アドレス空間システムの設計とRDMAを用いた実装

遠藤 亘[†1,a)]　田浦 健次朗[†1,b)]

概要：大域アドレス空間は，分散メモリ環境において仮想的に共有されたメモリ空間のモデルであり，一例としてはPGASが挙げられる．大域的なメモリ空間は複数の大域ページに分割され，ある大域ページの一意な位置を示す値 (大域アドレス) によって全ノードからアクセスできる．動的で非定型な計算構造を持つアプリケーションでは，大域アドレス空間が提供する自動的な通信が特に有用であり，MPIのようなメッセージパッシングに替わるモデルとして注目されている．一方で，通常のPGAS処理系においては，各ノードへの大域ページ配置は初期化時に固定されるという問題がある．動的な計算処理をノード間で負荷分散する場合，ページが静的に配置されているためにデータ局所性が低下し，ノード間通信量が増大してしまう．そこで，本研究では，大域ページを再配置可能な大域アドレス空間システムについて取り上げる．大域アドレス空間の重要な役割は，大域アドレスと実アドレスの対応関係といったメタデータの管理である．通常のPGASとは異なり，再配置を認めるため，ノード間でメタデータを同期する必要がある．さらに，ハードウェアが提供するRDMAの機能を活用するため，メタデータを各ノードでキャッシュし，キャッシュヒットした際は即座にRDMAによるデータアクセスを実行できるように設計する．本研究では，主にメタデータの管理手法について論じ，実装とその簡易評価を行う．

キーワード：大域アドレス空間, RDMA, 動的負荷分散

1. 序論

HPC分野において，動的で非定型な計算構造を持つアルゴリズムの活用が進んでいる．そのようなアルゴリズムの並列化には，動的負荷分散を自動的に行う処理系が有用である．分散メモリ環境での動的負荷分散を行う処理系は発展途上であるが，研究事例 [1][2][3][4] が存在する．

分散メモリ環境のプログラムでは，ノード間通信がボトルネックとなる場合が多く，その高速化が重要である．ノード間通信にはMPI [5] の利用が一般的であるが，近年ではPGAS [6][7][8][9] が注目されている．PGASは，仮想的に共有されたアドレス空間を提供するモデルの一種である．PGASは，共有メモリ環境に近い抽象化を提供するため，特に動的で非定型なアルゴリズムの記述に有用である [10]．PGASは通信処理の発生時点が明確なため，手動での通信のチューニングを行いやすいという特長を持つ．

これらの背景から，分散メモリ環境において動的で非定型なアルゴリズムを記述するには，動的負荷分散を行う処理系にPGASを組み合わせることが有用である．しかし，PGASに動的負荷分散を組み合わせる場合，データを保持したノードとは異なるノードに計算処理が移動することで，データ局所性の低下が問題となる．負荷分散時のデータ局所性の改善手法としては，ノード間でのデータの再配置 [1] が知られている．PGAS上のデータも同様に，負荷分散に応じて別のノードに再配置することでデータ局所性を向上させることができる．本稿におけるPGAS上のデータ再配置とは，データを実際に保持しているノードから他のノードにデータを移動する一方で，PGASへのデータアクセス時には再配置前後であたかも同じメモリ領域を扱っているかのように見せる仕組みを指す．

ノード間通信に関して，通信ハードウェアの動向としては，近年ではRemote Direct Memory Access (RDMA) による片方向通信の利用が一般的となっている．RDMAによる通信処理は，CPUやOSの介在なしに行われるため，通信にかかるオーバーヘッドが低減される．PGASのAPIは片方向通信が基本であるためRDMAと相性が良く，PGAS処理系の中にはリモートノードのCPUの介在なしに即座にRDMAによるデータ転送が可能な処理系もある [11]．そのような処理系では，PGAS上のデータにアクセ

[†1] 現在，東京大学
Presently with The University of Tokyo
[a)] wendo@eidos.ic.i.u-tokyo.ac.jp
[b)] tau@eidos.ic.i.u-tokyo.ac.jp

スするために必要なメタデータを事前にキャッシュすることで，メタデータの問い合わせによるレイテンシを削減している．

一方で，PGAS に対するデータ再配置の導入は，RDMA を活用する上で問題となる．メタデータがキャッシュされた状態では，PGAS 上のデータは，複数のノードから RDMA によって任意の順番で読み書きされうる．データ再配置を行う場合は，それらの RDMA 転送との間の排他制御を適切に行う必要がある．例えば，データ再配置中に元データが RDMA で書き換えられると，再配置後に書き込みが反映されない可能性がある．そのため，再配置中の書き込み要求は適切に遅延させる必要がある．

本研究では，再配置可能な PGAS の設計として，RDMA を活用できる処理系を提案する．メタデータキャッシュを導入する一方で，再配置が発生した場合にはノード間で適切に排他制御を行う．これによって，PGAS 同様の RDMA による高速なデータアクセスと，動的負荷分散に適した再配置という機能を両立させることができる．

本稿の構成は次の通りである．2 章では，既存の PGAS や関連する処理系について解説し，動的負荷分散や RDMA といった各種技術と組み合わせた際の問題点について述べる，3 章では，2 章での議論を踏まえて，新たな PGAS 処理系の設計について提案する．4 章では，実際にその実装を用いて簡易的な評価を行い，結果と考察を示す．5 章で，関連研究としていくつかの処理系の例を紹介し，本研究との差異を述べる．そして，6 章が本稿の結論である．

2. 背景

2.1 大域アドレス空間 (GAS)

本稿における GAS とは，分散メモリ環境において仮想的な共有メモリを実現するモデルであり，後述する DSM や PGAS といったモデルを含むとする．これらの用語については，原ら [12] の定義を使用している．GAS のアドレス空間はあくまで仮想的であり，その実体は各ノードのメモリの集合体である．GAS と対照的なのは，MPI に代表されるメッセージパッシングモデルである．MPI のモデルは低レベルであり，実際の通信処理が明確となるため，手動での通信のチューニングが容易であるという利点を持つ．しかし，MPI は明示的な通信処理の記述が必要なため，GAS と比べてプログラミングが煩雑であるという欠点がある．

1990 年代頃に盛んに研究されていた GAS のモデルとして，分散共有メモリ (Distributed Shared Memory, DSM) がある．DSM は，原則的にユーザが扱う全てのメモリを全ノードで共有することが特徴である．多くの DSM 処理系は，OS のメモリ管理機構を活用し，ユーザから見て GAS への透過的なメモリアクセスを実現する．DSM 処理系は，必要に応じてノード間通信を自動的に行い，ローカルメモリをキャッシュとして利用することで通信削減も行っている．

DSM のキャッシュ管理手法は，ノード間通信の回数やデータ量に大きく影響するため，共有メモリ環境でのキャッシュ管理以上に重要である．そのため，積極的にキャッシュ一貫性を緩和する試みがなされており，代表例として Release Consistency [13]，Lazy Release Consistency [14]，Home-based Lazy Release Consistency [15] が挙げられる．

現状では，大規模環境でスケールする DSM 処理系は知られていない．DSM がスケーラブルでない原因として，次の 3 点が挙げられる．特に (1) は，DSM において最も深刻である．

(1) ノード間通信の発生時点が不明確である．全てのメモリアクセスが完全に同一に記述されるため，大域領域アクセスのコストは隠蔽され，プログラマによる手動の通信のチューニングが困難である．
(2) OS のページ管理を利用した実装では，キャッシュの粒度が調整できず，そのままでは通信集約が困難である．最適な通信粒度はプログラムの各局面によって異なっているため，それに合わせて調整可能であることが望ましい．
(3) 主要な実装例である Lazy Release Consistency では，ベクトルタイムスタンプという，プロセス数を N としたとき 1 プロセスあたりの空間計算量が O(N) となるようなキャッシュ管理手法を採用している．

こうした問題から，HPC 分野では DSM が普及せず，GAS のモデルとしては Partitioned Global Address Space (PGAS) が主流になっている．PGAS の特徴は，全ノードで共有される大域領域と，他ノードと共有されない局所領域を明確に区別することである．局所領域アクセスは，共有メモリ環境でのメモリアクセスと同様に実行される．大域領域のデータ配置に関してもプログラマに明示されており，ローカルに配置されたデータへのアクセスコストは低いと見積もれる．さらに，PGAS では大域領域のデータキャッシュを行わないため，リモートに配置された大域領域のアクセスは通信と一対一に対応する．このように，プログラマによるメモリアクセスコストの見積もりが容易であることが，PGAS の利点である．

一方で，ノード間の動的負荷分散を用いたプログラムから PGAS 上にデータアクセスを行う場合，負荷分散後にも大域領域がノード内に配置されているとは限らない．PGAS 上のデータは静的に配置されるため，動的負荷分散でデータ配置とは異なるノードに計算処理が移動することがありうる．そこで，PGAS 上のデータを再配置することで，動的負荷分散後でもデータをローカルに配置可能にする手法が考えられる．

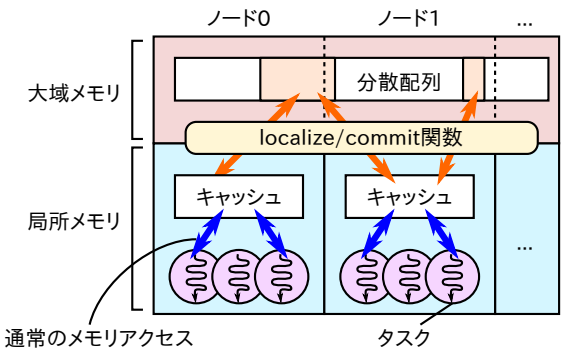

図 1 MGAS のメモリモデル

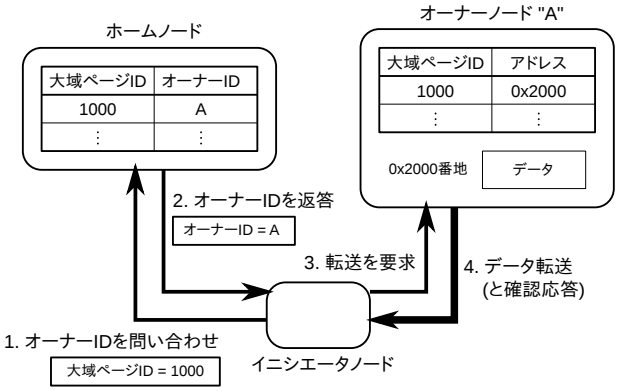

図 2 MGAS の get/put 関数の処理

2.2 MassiveThreads GAS (MGAS)

本研究の元となっている処理系は，秋山ら [16] の開発した GAS 処理系である MGAS である．MGAS は，通常の PGAS と同様に，大域メモリアクセスを行う get/put 関数を提供する他に，再配置を行う own 関数を追加している．

図 1 に，MGAS のメモリモデルを示す．MGAS における各ノードのメモリは，PGAS 同様に大域メモリと局所メモリに分割されている．大域メモリ上のデータは，大域ページという単位に分割され，各ノードに分散して保持されている．MGAS の低レベルな API として，大域メモリから局所メモリへのデータコピーを行う get 関数，逆向きのコピーを行う put 関数があり，PGAS と同様の機能を提供する．これらに加えて MGAS では，自ノードに大域ページの一部を再配置する own 関数を提供している．

図 2 に，MGAS の get/put 関数の処理を示す．MGAS において，大域ページにアクセスを行うノードは，イニシエータと呼んでいる．MGAS では，大域ページに対して静的に決定されるホームノードと，ある時点でデータを保持しているオーナーノードがある．これらノードの役割は全ノードで兼務しているため，それぞれが同じノードである可能性もある．イニシエータで get/put 関数が呼び出されると，大域ページ ID から静的にホームを決定し，オーナー ID を問い合わせる．ホームは，大域ページ ID とオーナー ID の対応表 (ホームディレクトリ) を集中管理しているので，対応するエントリを検索し，オーナー ID を返答する．イニシエータがオーナー ID を取得すると，次にオーナーに対してデータ転送を要求する．オーナーは大域ページ ID とデータアドレスの対応表 (オーナーディレクトリ) を保持しており，これを用いてデータ転送を実際に処理する．

次に，MGAS の own 関数の処理について述べる．own 関数の処理は get 関数の処理と似ているが，ディレクトリの変更を伴う点が異なる．own 関数では，最初にホームディレクトリで対応するエントリのロックを取得して，さらにオーナーディレクトリで対応するエントリも無効化する．新しいオーナーへのデータコピーが終わり次第，ホーム上のロックを解放する．

MGAS において，get/put 関数がホームから直接オーナーに問い合わせない理由は，イニシエータ上でオーナーごとに通信集約を行えるようにするためである．また，ホームディレクトリにアドレス情報を含めず，オーナーディレクトリを分離している理由は，オーナーが自身の所有する大域ページにアクセスする際にホームへの問い合わせを無くすためである．オーナーノード上で get/put 関数が実行されれば，ノード間通信は発生しない．

このように，MGAS は PGAS のモデルに再配置を組み合わせた処理系であるが，一方で性能面の課題がある．例えば，get/put 関数呼び出し時にイニシエータとオーナーが異なる場合，オーナー ID をホームノードへ毎回問い合わせる必要がある．その原因は，通常の PGAS 処理系とは異なり，再配置によってオーナー ID が変更される可能性があるためである．また，イニシエータではオーナー上のアドレスが不明なため，毎回オーナーに対してデータ転送を要求する必要がある．

データアクセスを要求するのは必ずしもオーナーではなく，参照局所性から一度アクセスしたイニシエータが再度アクセスする可能性が高い．そのため，イニシエータがディレクトリエントリを事前にキャッシュしていれば，その問い合わせによるノード間通信のオーバーヘッドを削減できるという点が本研究の元となっている．

2.3 RDMA とメタデータ

近年では，RDMA を搭載したインターコネクトが普及している．RDMA では，CPU や OS の介在なしにデータが転送されるため，高速なノード間通信が可能である．

RDMA が近年重要な理由として，CPU の周波数向上が限界に達した一方で，ネットワーク性能 (特にスループット) は依然として向上していることが挙げられる [17]．このハードウェア事情は，GAS の設計を決定する上で重要である．例えば，他ノードの CPU のポーリングを必要とするような通信処理は，本来のネットワーク性能を引き出せない場合がある．そのため，頻繁に必要となる通信処理

については，可能な限り片方向通信のみとすることが重要である．

一般的なインターコネクトにおいて RDMA を使用するには，まず送受信側双方のメモリ領域を API を介してハードウェアに登録しておく．その上で，実際に RDMA で転送する際には，以下の情報が必要となる．

- 送 (受) 信先のノード番号
- 送 (受) 信先ノード上のアドレス
- 送 (受) 信元ノード上のアドレス

GAS において RDMA を用いたデータ転送を行うには，大域アドレスからこれらの情報 (メタデータ) を計算する必要がある．

PGAS は特に RDMA と相性が良く，get/put 関数の高速化のために積極的に RDMA を活用する事例 [11][18] がある．PGAS においてデータは静的に分散されるため，原則的にメタデータは変化しないという仮定を置くことができる．そのため，大域メモリの領域を事前に登録し，全ノードにメタデータをキャッシュしておけば，任意の時点で全てのノードが大域メモリとの RDMA 転送を行うことが可能になる．

ここで，共有メモリ環境との対比について考えてみると，ホームディレクトリが OS が管理するページテーブルに相当し，メタデータキャッシュは TLB に相当する機能であるとみることができる．TLB 導入の目的が仮想アドレスから物理アドレスへの変換を高速化することであるのと同様に，メタデータキャッシュの目的は大域アドレスから実アドレスへの変換を高速化することにある．また，TLB とキャッシュメモリはあくまで独立した概念であるのと同様に，メタデータキャッシュとデータキャッシュも基本的には独立した概念である．

本研究では，データキャッシュの問題については扱わず，メタデータキャッシュについて取り上げる．メタデータをキャッシュできるような GAS の上に，データキャッシュを実装することも可能である．データやキャッシュのプロトコルによってメタデータの内容は異なるが，DSM のような GAS を実装する上でも同様の検討が成り立つ．例えば，ホームに最新データを保持する方式の DSM (例えば [15]) では，ホームとキャッシュとの間のデータ転送をするために，メタデータに相当するホーム上のアドレス情報が必要である．また，ホーム上に配置されたデータをホーム以外に再配置できれば，ノード間通信を減らすことができる．

再配置が必要となるのは，主に負荷分散が発生した場合であると述べた．例えば，時間発展する系をシミュレーションする場合，イテレーションごとに大域的配列を更新していくというプログラムになることが多い．そのようなプログラムでは，負荷分散は主にイテレーション間で発生し，イテレーション内では少数の動的負荷分散しか発生しない．大規模な負荷分散が必要となる局面に再配置とメタデータ更新を集中的に行い，それ以外で計算が集中的に発生している局面ではメタデータキャッシュを活用して高速にデータ転送を行うことで，データ局所性と負荷分散を両立することができる．

3. 提案手法

2 章の議論から，PGAS に対して再配置の機能を加えた GAS の設計を提案する．実装としては，RDMA を活用できるようにするため，メタデータキャッシュを導入し，それに対応したディレクトリ構造とする．

3.1 設計の方針

本研究の提案におけるメタデータ管理は，次のような方法で実現する．

- ホームノードに，RDMA 転送に必要なメタデータと，メタデータを共有しているノードのリスト (共有ノードリスト) を記録する．
- 任意のノードは，ホームに問い合わせることで，メタデータを自ノードにキャッシュし共有できる．
- イニシエータにおいてメタデータがキャッシュされている場合は，即座に RDMA 転送を実行できることを常に保証する．

図 3 のように，ディレクトリを持っているのはホームとイニシエータである．オーナーは，大域ページが配置されているだけのノードとして機能する．

3.2 通信プロトコル

図 3 に，get/put 関数の処理の流れを示す．get/put 関数の動作は，イニシエータ上のメタデータキャッシュの有無によって異なる．メタデータがキャッシュされていれば，即座に RDMA 転送を開始することが可能である．メタデータがキャッシュされていない場合は，ホームへの問い合わせが必要である．ホームは，この際にイニシエータを「共有ノード」としてディレクトリ上に追記する．これは，後述する再配置を行うためである．ホームからのメタデータの返答後に，イニシエータはメタデータのキャッシュを保存しつつ，RDMA 転送を開始する．

get/put 時にメタデータのキャッシュミスが発生した場合は，ホームからメタデータを取得することとなる．このため，最悪ではメタデータ取得分の 1 往復分のメッセージに加えて，RDMA 1 回分のレイテンシが発生する．メタデータのキャッシュは存在するが，データはリモートにある場合，RDMA 1 回分のレイテンシが発生する．最短なのはメタデータがキャッシュされていて，かつデータがローカルである場合で，ローカルコピーのみで終了する．オーナーとイニシエータが同じ場合は，メタデータがオーナー上にキャッシュされていれば，get/put 関数はノード間通信を必要としない．

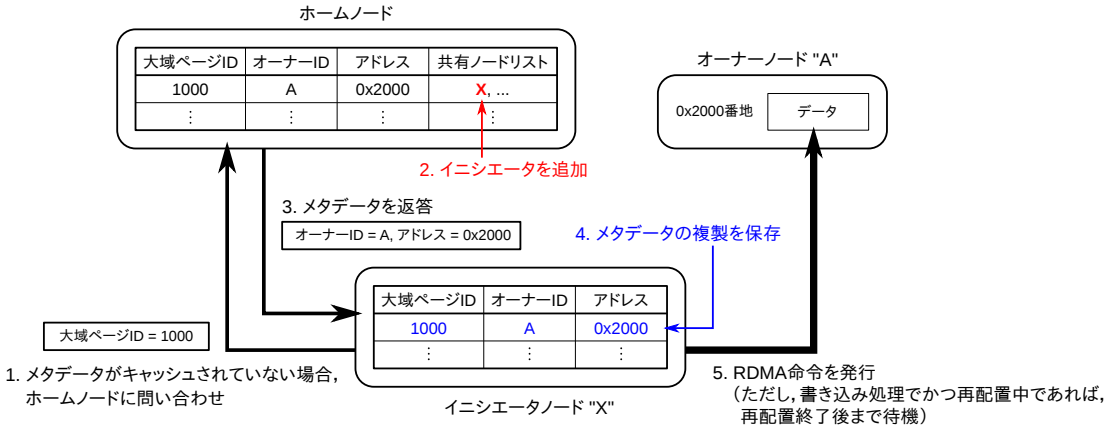

図 3 提案手法における get/put 関数の処理

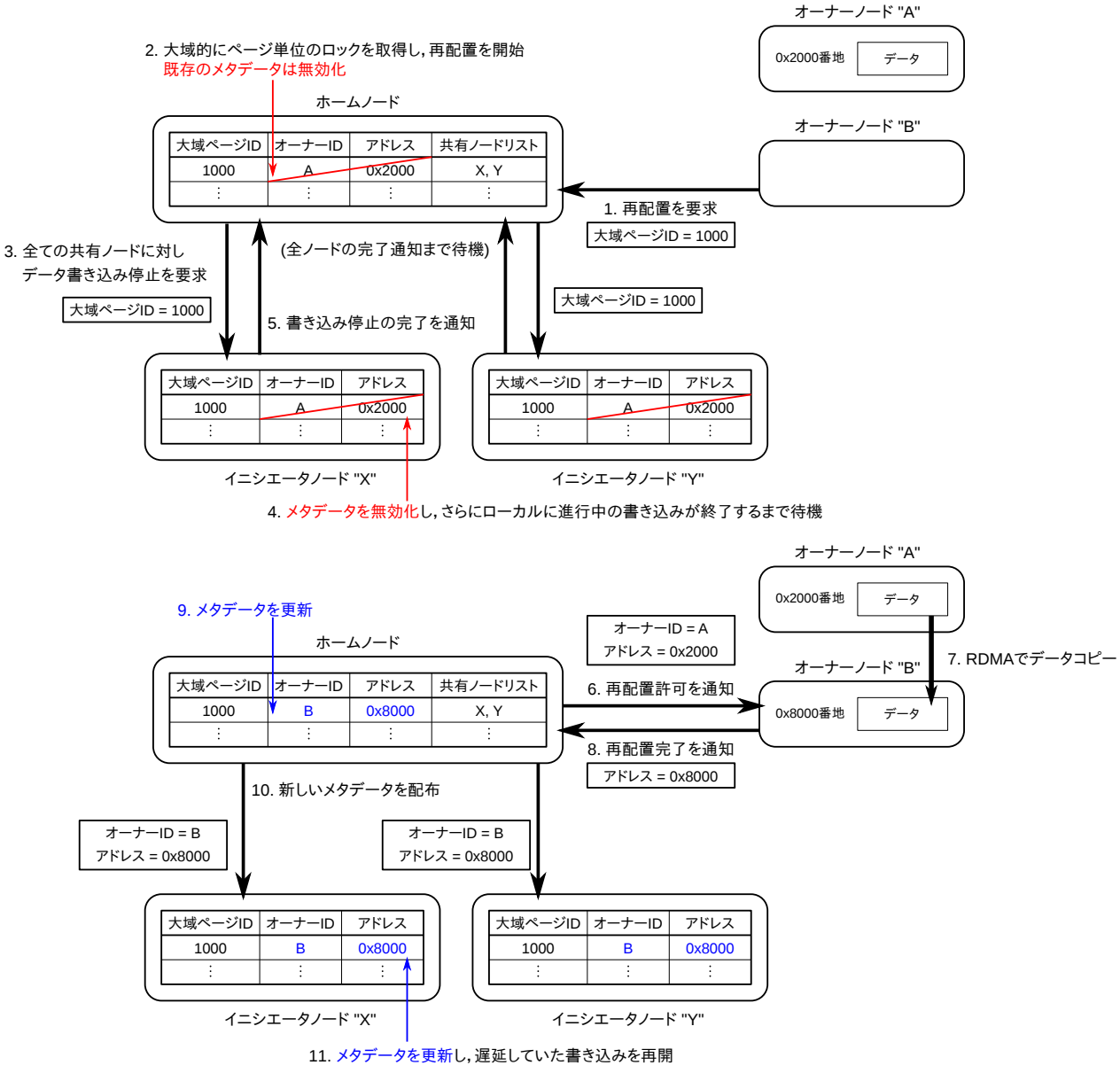

図 4 提案手法における own 関数の処理

表 1 評価環境 [19]

CPU	SPARC64 IXfx, 1.848 GHz
	1 ノードあたり 16 コア
メモリ	1 ノードあたり 16GB
コンパイラ	GCC 5.1.0 (オプション “-O3”)

図 4 に，own 関数の処理の流れを示す．own 関数の処理は，2 相コミットによって全てのメタデータのキャッシュを大域的に変更することであるといえる．再配置に関しては並列実行できないため，大域的なロックを管理する必要がある．提案手法では，ホームノード上でメタデータを無効化することで，再配置が複数スレッドから並列に実行されることを防ぐ．メタデータキャッシュ先はホームディレクトリ上で追跡されているため，全てのキャッシュ先ノードに対して RDMA によるデータ書き込みを中断するよう要求する．ホームが全てのキャッシュ先ノードから書き込み中断の完了通知を受け取れば，大域ページの状態は最新であり，再配置を開始できると判断する．再配置が完了したのち，ホームは新しいオーナー上のデータを示したメタデータを各キャッシュ先ノードに送信し，書き込み再開を指示する．この他，図 4 では省略しているが，全てのデータ読み込みが終了したことを確認して，元々のオーナー上のメモリを解放する機能も必要である．

再配置を実行している間，データへの書き込みを一時的に中断して，再配置が終了した後に再開する必要がある．一方で，データの読み込みだけであれば，再配置中であっても元のデータにアクセス可能である．ノード間でデータの読み書きの順序を保証する必要がある場合は，プログラマが明示的に排他制御を行う．

4. 簡易評価

3 章で述べた提案手法を実装し，簡易的な評価を行う．今回製作した処理系は実験的であり，通信機能などを含めて一部機能が効率化されていないため，簡易評価となっている．

マイクロベンチマークを用いてメタデータ管理の性能を評価する．**表 1** に，評価環境を示す．評価には，東大のスーパーコンピュータシステムである FX10 を用いた．評価に用いた RDMA の API である Fujitsu 拡張 RDMA インターフェイスは 4 つの NIC を選択可能であるが，今回は NIC0 のみ使用している．また，ノード内マルチスレッドは使用せず，1 ノード 1 プロセスで実行している．

4.1 get 関数のレイテンシ

事前評価として，本研究の GAS 処理系とは無関係に，Fujitsu MPI の RDMA Read のレイテンシを測定するマイクロベンチマークを行った．このベンチマークは 2 ノード間で行い，送受信とも同じアドレスに対して 4 バイトの RDMA Read を 1000 回実行続けるという状況で実行した．その結果，1 回の RDMA Read について 2.4[μsec] 程度のレイテンシが発生することが分かった．

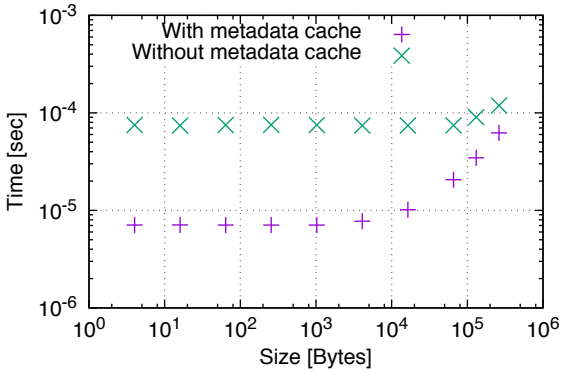

図 5 get 関数のレイテンシ

get 関数のレイテンシについては，ホーム，オーナー，イニシエータの 3 つをそれぞれ別々のノードとして測定を行っている．メタデータキャッシュを利用する場合と，メタデータキャッシュを利用せずに毎回破棄する場合で比較を行う．get 関数の呼び出しは 10000 回繰り返して 1 回分に換算している．

図 5 に，get 関数でリモートの大域メモリからローカルにコピーする際のレイテンシを示す．メタデータキャッシュを利用している場合のレイテンシは最短で 7.1[μsec] である．そのため，GAS を使用しない場合と比較して，4.7[μsec] のオーバーヘッドが生じていることが分かった．RDMA のみにかかるレイテンシよりも，ノード内オーバーヘッドの方が大きいという結果になっている．

メタデータキャッシュを利用しない場合のレイテンシは 75.1[μsec] であり，メタデータキャッシュを利用した場合の 10 倍以上の時間がかかっている．この結果から，メタデータキャッシュがレイテンシ改善に有用であることが分かる．ただし，現在のメッセージハンドラの実装は実験的なため，これを改良することによってメタデータキャッシュがない場合のレイテンシは改善する可能性がある．

4.2 own 関数のレイテンシ

own 関数の評価では，ホームとして 1 ノードと，オーナーとして 2 ノードを使用する．オーナー間で own 関数の呼び出しを交互に繰り返すことで，同じページに対する own 関数のレイテンシを計測する．own 回数の呼び出しは 1000 回繰り返して 1 回分に換算している．

図 6 に，own 関数のレイテンシを示す．get 関数と比べて，メッセージハンドラによるノード間の排他制御が必要であるという事情があるため，own 関数のレイテンシは大きくなっている．

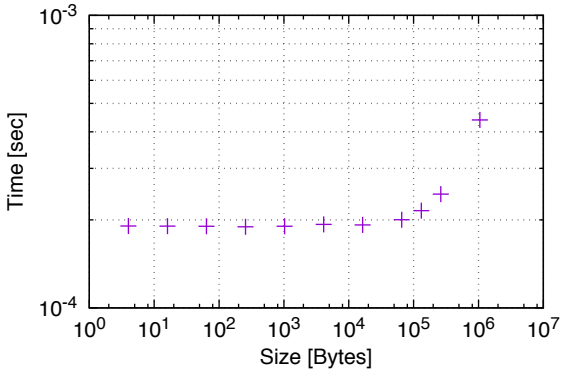

図 6 own 関数のレイテンシ

5. 関連研究

Farreras ら [11] は，PAGS 言語である UPC の処理系 (IBM XLUPC コンパイラ) での RDMA の活用について論じている．XLUPC の元々の実装ではメタデータの集中管理を採用していたため，get/put 関数のレイテンシが大きくなっていた．Farreras らは，メタデータキャッシュを導入することで get/put 関数を高速化している．

Chavarría-Miranda ら [18] は，PGAS ライブラリである Global Arrays でのメタデータ管理におけるスケーラビリティについて論じている．Global Arrays では元々全ノードにメタデータをキャッシュしていたため，メモリ使用量の観点からスケーラブルでなかった．そこで，一部のノードをメタデータサーバとして運用し，さらに計算ノードにもメタデータキャッシュを配置することで，メタデータのメモリ使用量を削減しつつ RDMA も活用できる PGAS を提案している．

本研究で提案する GAS では，再配置をサポートするため，PGAS における例よりも複雑となっている．PGAS では，メタデータは不変であるとして取り扱うことができるため，メタデータキャッシュの導入は比較的容易である．

近年では，RDMA を活用して，レイテンシの改善に着目した新たな DSM 処理系が登場している．例として，Argo [17] が挙げられる．本研究の方向性と同様に，これらの研究でも get/put 関数のレイテンシを削減することを目的としている．これらの DSM の研究では，主にデータキャッシュの課題に取り組んでおり，本研究で取り上げているメタデータキャッシュの問題とは異なっている．2 章で取り上げた問題点から，DSM 処理系が大規模環境でもスケーラブルであるかは疑問が残るが，データキャッシュも GAS に必要な機能の一つである．Argo については，RDMA のみでディレクトリ操作を実現するという取り組みを行っており，メッセージハンドラ無しで GAS が実現できることを示している点で特徴的である．

6. 結論

本研究では，再配置可能かつ RDMA を活用できる GAS の設計を提案した．RDMA に必要なメタデータを共有することで，データアクセスでは RDMA を活用しながら，必要に応じてデータ再配置も行うことができる．

評価においては，メタデータを使用した場合の方が，使用しない場合と比べて高速なデータアクセスが可能になることを示した．一方で，現行の実装では GAS の使用によるオーバーヘッドが大きいことが分かった．この問題については，コードレベルのチューニングを行う予定である．

今後の課題は，ディレクトリ操作を RDMA だけで行えるように改良することが挙げられる．これにより，メタデータキャッシュミスや再配置時の性能が向上すると考えられる．

参考文献

[1] Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., Lukasz Wesolowski and Kale, L.: Parallel Programming with Migratable Objects: Charm++ in Practice, *SC '14: Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 647–658 (online), DOI: 10.1109/SC.2014.58 (2014).

[2] Van Nieuwpoort, R. V., Wrzesiska, G., Jacobs, C. J. H. and Bal, H. E.: Satin: A high-level and efficient grid programming model, *ACM Transactions on Programming Languages and Systems*, Vol. 32, No. 3, pp. 1–39 (online), DOI: 10.1145/1709093.1709096 (2010).

[3] Min, S.-J., Iancu, C. and Yelick, K.: Hierarchical Work-Stealing Framework for Multi-core Clusters, *PGAS '11: Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models* (2011).

[4] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S. and Oskin, M.: Grappa : A Latency-Tolerant Runtime for Large-Scale Irregular Applications, Technical report (2014).

[5] The MPI Forum: MPI: A Message-Passing Interface Standard, Technical report (1994).

[6] El-Ghazawi, T. and Cantonnet, F.: UPC Performance and Potential: A NPB Experimental Study, *SC '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, p. 26 (online), DOI: 10.1109/SC.2002.10034 (2002).

[7] Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H. and Aprà, E.: Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit, *International Journal of High Performance Computing Applications*, Vol. 20, No. 2, pp. 203–231 (online), DOI: 10.1177/1094342006064503 (2006).

[8] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing, *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages,*

and applications, Vol. 40, pp. 519–538 (online), DOI: 10.1145/1094811.1094852 (2005).

[9] Chamberlain, B., Callahan, D. and Zima, H.: Parallel Programmability and the Chapel Language, *International Journal of High Performance Computing Applications*, Vol. 21, pp. 291–312 (online), DOI: 10.1177/1094342007078442 (2007).

[10] Zhang, J., Behzad, B. and Snir, M.: Optimizing the Barnes-Hut algorithm in UPC, *SC '11: Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11 (online), DOI: 10.1145/2063384.2063485 (2011).

[11] Farreras, M., Almási, G., Cacaval, C. and Cortes, T.: Scalable RDMA performance in PGAS languages, *IPDPS '09: Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, (online), DOI: 10.1109/IPDPS.2009.5161025 (2009).

[12] Hara, K. and Taura, K.: Parallel Computational Reconfiguration Based on a PGAS Model, *Journal of Information Processing*, Vol. 20, No. 1 (2012).

[13] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiprocessors, *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26 (online), DOI: 10.1109/ISCA.1990.134503 (1990).

[14] Keleher, P., Cox, A. and Zwaenepoel, W.: Lazy Release Consistency for Software Distributed Shared Memory, *ISCA '92: Proceedings the 19th Annual International Symposium on Computer Architecture*, IEEE, pp. 13–21 (online), DOI: 10.1109/ISCA.1992.753300 (1992).

[15] Zhou, Y., Iftode, L. and Li, K.: Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems, *OSDI '96: Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, New York, New York, USA, ACM Press, pp. 75–88 (online), DOI: 10.1145/238721.238763 (1996).

[16] 秋山茂樹，田浦健次朗：軽量マルチスレッディング向け大域アドレス空間ライブラリ，情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 135, No. 7, pp. 1–6 (2012).

[17] Kaxiras, S., Klaftenegger, D., Norgren, M., Ros, A. and Sagonas, K.: Turning Centralized Coherence and Distributed Critical-Section Execution on their Head, *HPDC '15: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, New York, New York, USA, ACM Press, pp. 3–14 (online), DOI: 10.1145/2749246.2749250 (2015).

[18] Chavarría-Miranda, D., Agarwal, K. and Straatsma, T. P.: Scalable PGAS Metadata Management on Extreme Scale Systems, *CCGrid '13: Proceedings of 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 103–111 (online), DOI: 10.1109/CCGrid.2013.83 (2013).

[19] 富士通株式会社：FX10 スーパーコンピューターシステム Oakleaf-FX / Oakbridge-FX 利用手引書 Version 1.6 (2014).