Vol.2013-HPC-140 No.42
2013/8/2

グローバルデータ構造のためのメモリ管理モデルの検討

安島雄一郎[†1,†2]　秋元秀行[†1,†2]　安達知也[†1,†2]　岡本高幸[†1,†2]
佐賀一繁[†1,†2]　住元真司[†1,†2]　三浦健一[†1,†2]

本論文では，将来の PGAS 言語の改良に貢献するグローバルメモリ管理機構を提案し，その実装方式を OS レベル，ランタイムシステムレベル，ライブラリレベルで検討する．また，グローバルメモリ管理機構に仮想メモリを導入する際にインターコネクトハードウェアに要求される機能も検討する．ライブラリレベル方式については通信ライブラリへの実装を想定し，ハードウェア抽象化層の設計，グローバルアドレスの構成，メモリ管理機構の階層設計について検討する．

A Study of Memory Management Model for Global Data Structures

YUICHIRO AJIMA[†1,†2]　HIDEYUKI AKIMOTO[†1,†2]　TOMOYA ADACHI[†1,†2]
TAKAYUKI OKAMOTO[†1,†2]　KAZUSHIGE SAGA[†1,†2]
SHINJI SUMIMOTO[†1,†2]　KENICHI MIURA[†1,†2]

In this paper, we propose a new memory management model called “global memory management,” which contributes to future Partitioned Global Address Space (PGAS) languages. We also discuss various implementation schemes of global memory management model such as operating system level, runtime system level and library level, and describe required features of an interconnect hardware to introduce virtual memory mechanism into an implementation of global memory management. A communication library including global memory management functions is further investigated from the points of view of a hardware abstraction layer, the composition of global address, and layered implementations.

1. はじめに

現在の計算機では複数のプロセッサコアを 1 チップに搭載しメモリを共有するマルチコア・プロセッサが主流である．High Performance Computing (HPC)分野ではこのようなマルチコア・プロセッサや，さらにコア数を増やしたメニーコア・プロセッサを多数接続し，並列計算により科学技術計算を高速に行う．並列計算機の主記憶には共有メモリと分散メモリがあるが，現在では分散メモリのクラスタ型並列計算機や超並列計算機が主流である．分散メモリ並列計算機では，メモリを共有し単一の Operating System (OS)が走行する単位をノードと呼ぶ. OS は仮想メモリアドレス空間機能により複数の仮想メモリアドレス空間を提供し，さらに各仮想メモリアドレス空間に複数の実行コンテキストを割当ることができる．本論文では仮想メモリアドレス空間を共有する実行コンテキストをまとめてプロセスと呼び，各実行コンテキストをスレッドと呼ぶ．コンテキスト切り替えはオーバヘッドが大きいので，HPC 分野では一般的にノードあたりのプロセッサコア数と実行コンテキスト数を一致させる．

プロセス間ではメモリ空間を共有しないので，プロセス間で明示的にデータ交換を行うプロセス並列プログラミングが必要である. HPC 分野では 90 年代より Message Passing Interface (MPI)ライブラリ[1]がプロセス並列の主流である．MPI は煩雑な送受信処理を関数呼び出しで記述するのでプログラム生産性が低い．しかしプロセス間データ転送は通信時間が長いため最適化が重要であり，データ分割等の指示に基づいた自動並列化では，プログラマが直接記述する送受信処理の性能に匹敵することは困難であった．

MPI よりも容易に明示的なプロセス間データ転送を記述するため，Partitioned Global Address Space (PGAS)言語と呼ばれる言語群が開発されてきた．PGAS 言語のデータ構造では複数プロセスに分割配置されるグローバル配列と，他プロセスの共有変数を参照するグローバルポインタが重要な構成要素である．どちらも配列の添え字やポインタの値にプロセス番号が埋め込まれており，値の範囲とプロセスが関連付けられている点が Partitioned の由来である．構造が単純なグローバル配列はデータ並列処理にもタスク並列処理にも適している．グローバルポインタを用いた複雑な並列データ構造とアルゴリズムは，タスク並列処理に適している．ここでグローバルアドレス空間に配置する並列データ構造をグローバルデータ構造と短く呼ぶ．Unified Parallel C (UPC) [2]や X10 [3]などの PGAS 言語はグローバルポインタとタスク並列に相当する機能を有するが，グローバルメモリの動的割当やオブジェクト生成は，グローバルアドレス空間のうち該当プロセスに関連付けられた領域に制限される．しかし，割当られるメモリや生成されるオブジェクトは，生成したプロセスではなく参照，更新するプロセスに関連付けられるのが望ましい．

†1 富士通株式会社　次世代テクニカルコンピューティング開発本部
Fujitsu Limited., Next Generation Technical Computing Unit
†2 独立行政法人科学技術振興機構　戦略的創造研究推進機能
Japan Science and Technology Agency (JST),
Core Research for Evolutional Science and Technology (CREST)

本論文では将来の PGAS 言語においてグローバルメモリ割当やオブジェクト生成の際に配置プロセス指定が可能になるように，下位層であるローカルメモリ管理機構，ランタイムシステム，通信ライブラリ，インターコネクトハードウェアに加えるべき変更を検討する．2 章では従来の並列処理系とメモリモデルを紹介し，3 章でグローバルデータ構造に適したメモリ管理モデルを提案する．4 章で提案モデルを実現するための低レベル通信ライブラリの構想を紹介し，最後に 6 章で今後の課題とまとめを述べる．

2. 従来の並列プログラミングとメモリモデル

2.1 MPI

MPI はメッセージパッシング通信でプロセス間データ交換を行う通信ライブラリである．送信元プロセスはメモリ上にメッセージを作成して MPI_Send 関数を呼ぶ．MPI の典型的な動作としては，送信元はメッセージを即座に宛先プロセスに転送する．宛先プロセスでは MPI_Recv 関数でメッセージを受信する．送信元のユーザープログラムが転送すべきメッセージを作成し，宛先のユーザープログラムもメッセージを解釈する必要があり，MPI の生産性が低い原因となっている．その一方，メッセージのような連続データはランダムアクセスに弱い IO バス経由のデータ転送や，プロトコルオーバーヘッドの大きい Ethernet 経由のデータ転送でもスループットが出ることは利点である．

受信側の MPI ライブラリは MPI_Recv 関数が呼ばれるまでメッセージをプールし，MPI_Recv 関数で目的のメッセージがプールにあるか検索する．MPI ライブラリがメッセージを蓄えるので，共有メモリで必要な同期やレーシングの考慮が不要になることは MPI の大きな利点である．

2.2 OpenMP

OpenMP [4]は逐次プログラムにデータ分割，ループ分割などの指示行を追加することで，コンパイラがスレッド並列化を行うスレッド並列処理系である．スレッド並列であるので共有メモリを前提としており，分散メモリのコンパイラ自動並列化とは違い実用的な並列化性能を有する．また，言語レベルのポータビリティを確保するため，OpenMP はデータ並列だけでなくタスク並列にも対応する．

HPC 分野でのマルチスレッドは基本的に，プロセス並列の 1 プロセス処理をさらにマルチスレッドで並列化するハイブリッド並列である．そのため，ベースのアルゴリズムを変えずにコンパイラがスレッド並列化を行う OpenMP は生産性の観点で望ましく，HPC 分野での主流となっている．

一般にマルチスレッド・プログラミングではメモリ上で変数を共有するためのプロセッサ命令を使用して，スレッド間での排他制御，同期，並列データ構造への並行参照・更新を行うが，この場合は元の逐次プログラムとは異なる並列アルゴリズムを記述する必要がある．

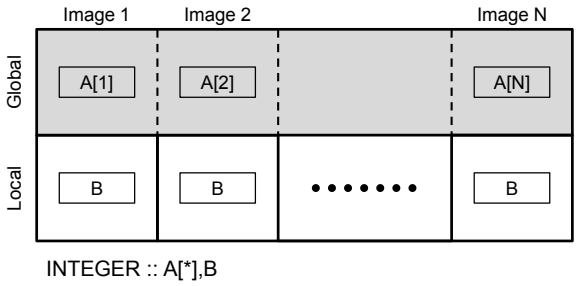


図 1　CAF の Co-Array の例

Figure 1　An example of co-array of CAF.

2.3 Co-Array Fortran

Co-Array Fortran (CAF) [5]は Fortran を Single Program Multiple Data (SPMD)モデルのプロセス並列に拡張した言語であり，Fortran2008 で標準規格となっている．co-array と呼ぶ，分散メモリを前提としたグローバル配列が導入されている．co-array は角括弧の添え字が追加された配列である．co-array の角括弧を除いた定義の配列を，全プロセス（CAF ではイメージと呼ぶ）で同期的に割付ける．どのプロセスも他のプロセスで割付けられた co-array を参照することができる．参照する際は角括弧でプロセス番号を指定する．図 1 に co-array の例を示す．

CAF では allocate，deallocate 文によりを co-array の動的割付を行うことが可能である．また，派生型の中に allocatable 配列を含むことで，プロセス毎にサイズの違う配列を動的割付することも可能である．一方，CAF ではグローバルな自動変数や，グローバルな返り値，そしてそもそもグローバルポインタは用意されていないため，Linked List などのデータ構造とアルゴリズムを実現するにはデータ構造全体を co-array 上に載せる必要がある．

2.4 XcalableMP

XcalableMP [6]は OpenMP のような指示行ベースで CAF 相当のグローバル配列を実現する処理系である．XcalableMP は Fortran だけでなく C 言語にも対応する．

2.5 Chapel

Chapel [7]は C，Modula 風の構文にイテレータ（yield），型推論（var）などの現代スクリプト言語風の要素を加えた新しい SPMD 並列処理系である．Chapel では domain というグローバルデータ構造が導入されている．Chapel プログラムでは domain を subdomain に分割する方法と，domain に対する逐次処理を記述する．Chapel コンパイラは domain に対する逐次処理記述を元にして，subdomain でデータ分割する自動並列化を行う．

Chapel は，各プロセスからのグローバルデータ構造への参照，更新を直接記述すると，境界データの交換，不規則なデータ分割などで場合分け処理が多くなる問題を解決する．その一方，コンパイラの自動並列化性能によって並列化性能が大きく左右される点が問題である．

Chapel の domain は多次元配列，subdomain はストライド

指定可能な部分配列が基本であるが，普通の配列のように扱える圧縮疎行列配列，グラフを格納する頂点配列，文字列をキーとする連想配列も利用できる．Chapel は CAF と同様にグローバルポインタに対応しないが，配列上に構築した複雑なデータ構造を言語仕様として用意することで生産性を改善している．

2.6 Unified Parallel C

UPC は C 言語を SPMD モデルのプロセス並列に拡張した言語であり，別プロセスから参照可能な共有変数の概念が追加されている．予約語 shared を付けて宣言した共有変数はグローバルメモリ空間に配置され，どのプロセスからでも参照できる．ただし共有変数はプロセス 0 から割当られるので，配列でないスカラの共有変数は全てプロセス 0 に置かれる．図 2 に共有変数の例を示す．

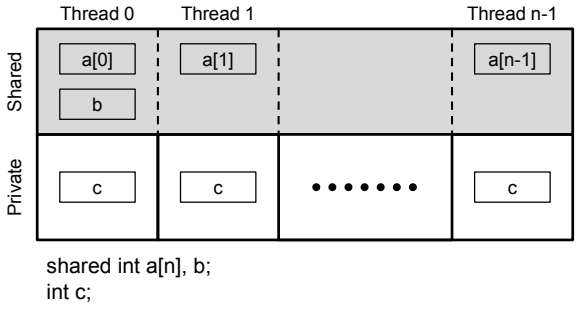


図 2　UPC の共有変数の例

Figure 2　An example of shared variables of UPC.

UPC はグローバルポインタに対応しており，UPC グローバルポインタの典型的な実装はプロセス ID，ローカルアドレス，配列内相対位置を格納する．グローバルポインタをグローバルアドレス空間に置くこともできる．また明示的なグローバルメモリ割当も可能である．upc_alloc 関数は該当プロセスのみでグローバルメモリを割当，upc_all_alloc 関数は全プロセスで同期してグローバルメモリを割当る．これらの仕様により，UPC ではグローバルアドレス空間上でリスト，ツリーのようなデータ構造も構築できる．図 3 にグローバルポインタの例を示す．ただし SPMD モデルは並列データ構造操作の記述に向かないので，アルゴリズムの記述はプログラマの負担が大きい．

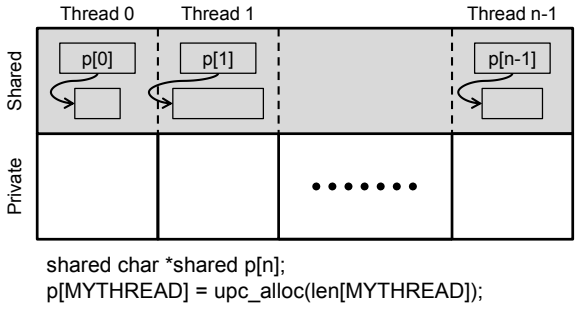


図 3　UPC のグローバルポインタの例

Figure 3　An example of global pointers of UPC.

2.7 X10

X10 は Java 言語仕様をベースとして開発された，オブジェクト指向のタスク並列言語である．全てのオブジェクトは所属プロセス属性を持っている．at 構文により，あらゆる処理に対して実行するプロセスを指定できる点がユニークである．なお X10 ではプロセスを Place，オブジェクトの所属プロセスを Home と呼ぶ．

at 構文だけでは実行プロセスが遷移する逐次プログラムであるが，ここで並列処理を導入するのが async 構文である．async が指定された構文は新しいスレッド（X10 では Activity と呼ぶ）で実行される．async と at を組み合わせることにより，任意のプロセスで任意の数のスレッドを起動することができる．X10 はグローバルアドレス空間上でリスト，ツリーのようなオブジェクトを生成し，タスク並列で自然に並列アルゴリズムを記述できる．また，データ並列を容易に記述するための分散配列オブジェクトもあり，分割パターンを生成するメソッドが実装されている．

UPC と X10 をメモリモデルの観点で比較すると，その差異はベースとなる言語に由来している．UPC は C 言語のようにポインタを扱い，ユーザーが malloc，free をするメモリモデルである．一方，X10 は Java に由来し，オブジェクトの参照を扱い，ユーザーはオブジェクトの生成，破壊を行う．UPC はガベージコレクションを行わないが，X10 はガベージコレクションを行う点もベース言語に由来する．

2.8 Charm++

Charm++ [8]は C++でランタイムシステムが実装された，並列オブジェクト実行モデルである．オブジェクトを扱う点はオブジェクト指向の PGAS 言語である X10 と共通しているので，以下に X10 と Charm++比較する．

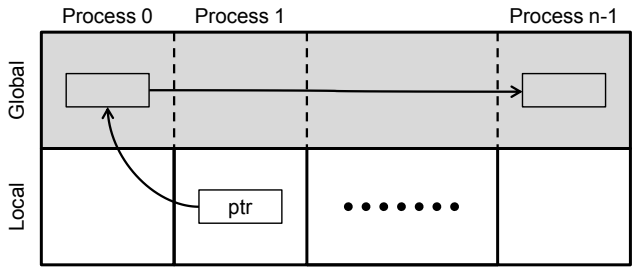
実行モデルでは，X10 は async 構文で明示的にタスク並列処理を行う．これに対し Charm++ではオブジェクトのメソッド呼び出しが暗黙的に並列実行される．典型的な Charm++実装では各プロセスのランタイムシステムが並列実行をスケジュールする．データ構造では，X10 はグローバルポインタによりグローバルアドレス空間上にグローバルデータ構造を構築し任意の参照，更新操作を行うのに対し，Charm++は多次元オブジェクト配列の分散と broadcast，reduce 操作が基本となる．

3. グローバルメモリ管理モデルの提案

3.1 グローバルメモリ管理モデル

従来の PGAS 言語では，メモリ割当やオブジェクト生成は全プロセスで同期して行われるか，該当プロセスのメモリに対してのみ行われる．グローバルアドレス空間全体で適切にデータ構造を配置するには，タスク並列で実行プロセスを指定してメモリ割当やオブジェクト生成を行う必要がある．しかしタスク並列はランタイムシステムにおけるタスク起動，コンテキスト切り替えなどのオーバヘッドが

大きい．SPMD でも場合分けで記述することは可能だが，プログラマの負荷が重い．この解決策として，別のプロセスに対するグローバルメモリの動的割当が可能な，グローバルメモリ管理モデルを提案する．図 4 に将来の PGAS 言語におけるグローバルメモリの動的割当の使用例として，グローバル・リストの例を示す．




```
at (1) {
    sturct list *shared ptr;
    ptr = pgas_alloc_at(sizeof(struct list), 0);
    ptr->next = pgas_alloc_at(sizeof(struct list), n-1);
}
```

図 4　将来の PGAS 言語におけるグローバル・リストの例
Figure 4　An example of a global list data structure in a future PGAS language.

インターコネクトハードウェアの Remote Direct Memory Access (RDMA)通信機能は別プロセスのメモリを直接参照するので, 原理的には RDMA で別ノードのメモリ管理情報を直接更新することで，グローバルメモリの割当が可能である．RDMA 経由でのメモリ割当は，該当プロセスでタスクを起動してローカルにメモリ割当を行うよりもオーバヘッドが小さいことが期待される．グローバルメモリ管理の実現にあたっては，ソフトウェア資産維持の観点から，現在プロセッサと OS が提供している複雑な仮想メモリ機能と共存できる仕組みであることが望ましい．グローバルメモリ管理の実現方法を以下に検討する．

3.2 空きページの割当，解放

OS は空きページを Buddy System のようなデータ構造で管理している．通常，連続した空きページはページフレームとしてまとめられ，ページフレームのサイズ毎にリスト構造で格納される．リストの要素はあらかじめ全ページ分のページ構造体配列として作成される．空きページフレーム・リストへの配列と統計情報等が管理構造体に格納される．以上に述べた管理情報は並列データ構造を使用していないので，並行参照が起きないように管理構造体の単位で排他制御が行われる．また，管理情報の更新中はプリエンプションおよび割込みも禁止される．

空きページを RDMA 経由でページの割当, 解放が可能なように管理データ構造を拡張できれば，グローバルメモリ管理の根幹部分を達成したことになる. もっとも簡単には，OS カーネルの空きページ割当と RDMA 経由の空きページ割当の間で排他制御を行い，管理構造体およびページ構造体配列を RDMA で更新すれば良い．

3.3 排他制御

複数のプロセッサコア間での排他制御に IO デバイスが参加する場合，排他制御方式を見直す必要がある．汎用性の高い方法は, IO デバイスの機能をプロセッサコアから制御して排他制御を行う方式である. IO デバイスは排他制御対象の処理中であるかどうかを表示し，排他制御対象処理の開始禁止指示を受付ける．プロセッサコアは IO デバイスに対する排他制御を実施する前にプロセッサコア間で排他制御を行い, IO デバイスに対する排他制御を実施するプロセッサコアは 1 つに限定される．しかしこの方式は IO デバイスのレジスタで制御するためオーバヘッドが大きい．

ここで，PCI Express 3.0 [9]で導入される Atomic Read Modify Write トランザクションを使用すると，オーバヘッドを削減できる可能性がある．Atomic Read Modify Write の一種である Compare and Swap を使用すれば, メモリ上の排他制御変数に対する 1 回のトランザクションで排他制御が行えるので，低遅延の排他制御を期待できる．この新機能を利用して別ノードから Atomic Read Modify Write 可能なインターコネクトを実現すれば，プロセッサと IO デバイス間の排他制御遅延は大きく削減される．なお，インターコネクトデバイスだけでなく PCI Express スイッチ, ルートコンプレックス，メモリコントローラも Atomic Read Modify Write に対応している必要があるので，PCI Express 3.0 対応システムであっても Atomic Read Modify Write が利用できない場合がある．

3.4 仮想アドレス

Buddy System をベースとした RDMA 経由のページ取得では，物理アドレスで連続した領域を確保できる．しかし Buddy System にはフラグメンテーションが進むと大きい領域を確保できなくなる問題や，領域サイズが 2 の冪乗に切り上げられるためメモリの使用効率が悪い問題がある．

そこで RDMA 経由のメモリ割当でも, 不連続な複数ページを仮想的に連続領域とみなす，仮想アドレスを導入すべきである．幸い，RDMA に対応する現在のインターコネクトデバイスは，各プロセスのメモリ空間に対して直接メモリ参照するための仮想アドレス機構を既に持っている．課題は，現状のインターコネクトデバイスでは該当ノードのデバイスドライバしかページテーブルを登録できないことと，そもそも登録すべき物理アドレスを取得したノードは該当ノードではないことである. ページを RDMA で取得しても，取得元ノードのデバイスドライバを動かさないと結局利用できないのでは意味がない．

そこで，インターコネクトデバイスに別ノードからページテーブルを登録する機能を検討する．ページテーブルを別ノードに書き込む事自体は RDMA で可能である. 課題は仮想アドレス空間の識別子取得と，ページテーブル格納のためのメモリ割当である．どちらも事前に該当ノードで資

源を確保してデータ構造に格納しておけば，別ノードから排他制御と資源取得を行うことができる．この場合の排他制御は該当ノードのプロセッサとは行わなくてよい．現在のインターコネクトデバイスの多くは同一インターコネクトデバイスを経由する参照間での排他制御機能を有するので，IO バスの Atomic Read Modify Write トランザクションが無くても低遅延な排他制御が可能と考えられる．しかし，ページテーブルを RDMA で更新できるだけでは不足で，別ノードからインターコネクトデバイスの TLB フラッシュを行うような，特別な制御通信も必要となる．

3.5 OS のページ管理を変更しない別方式

前節までに検討した方式は OS の基本機能であるページ管理方式を拡張しており，OS への影響が大きい．そこで別案として，OS を変更せずにグローバルメモリ管理を実現する方式を検討する．基本方針としては，ランタイムシステムがあらかじめ空きページを取得してしまい，ローカルメモリ管理とグローバルメモリ管理の共有空きページとして管理する．この場合，ランタイムシステムの管理下にあるプロセスは空きページを利用できるが，同一ノードで走行する他のアプリケーションからは使用できる空きメモリ容量は減少したように見える．

ページの管理データ，リモートメモリ管理のページ取得手段については，OS のページ管理拡張の場合と同様のデータ構造，アルゴリズムで問題ない．ローカルメモリ管理のページ取得手段に関しては，元が mmap システムコールで無名ページを取得しているならば，共有空きページ領域をマップするように変更すればよい．

3.6 ランタイムシステムも変更しない別方式

メモリ管理機構はランタイムシステムにおいても中核であり，malloc 関数や new 構文のようなローカルメモリアロケータに修正が入るのは大きなインパクトがある．そこで，OS もランタイムシステムも変更せずに，ライブラリレベルでグローバルメモリ管理を実現する別案を検討する．基本方針としては，ローカルの空きメモリをあらかじめ取得し，グローバルメモリ管理の空きページとして管理する．この場合，該当プロセスにおいても malloc 関数や new 構文で利用できる空きメモリ容量が減少するので，グローバルメモリ管理の空きページからローカルメモリ割当を行うライブラリ関数を用意する．

4. 低レベル通信ライブラリの検討

前章では PGAS 言語の将来の改良に貢献するグローバルメモリ管理機構を提案し，その実装方式を様々なレベルで検討した．そして，ライブラリレベル方式はローカルメモリ管理との空き容量共有に制約があるが，OS もランタイムシステムも変更せずにグローバルメモリ管理を実装できる見通しが得られた．本章ではグローバルメモリ管理の簡易実装として，通信ライブラリの一部としての実装を検討する．合わせて，将来の PGAS 言語の下位層に相応しいメモリモデルに基づいた低レベル通信ライブラリについても検討する．

4.1 ハードウェア抽象化層

低レベル通信ライブラリは最下層にインターコネクトデバイス仕様の差を吸収するハードウェア抽象化層を持つべきである．ハードウェア抽象化層の設計には大きく 2 つの方針がある．最低限の共通通信機能にオプションの高速通信機能を加える方針と，将来あるべきインターコネクトデバイスを想定した基本通信機能を定義し，機能が不足するインターコネクトデバイスに対応する際はランタイムシステム等によるエミュレーションを行う方針である．前者は最小限の手間で対応デバイスを増やすことを目指し，新しい言語処理系の普及を目指す戦略に適する．後者は将来のインターコネクトデバイス機能を最大限引き出すとともにスムーズに移行することを目指し，既存の言語処理系を将来のインターコネクトデバイスに適応して拡張する戦略に適する．また前者の抽象化層の上位に後者の抽象化層を設ける二重の抽象化も可能であるが，階層の増加はオーバヘッドの増加に繋がる．本論文では新しい PGAS 言語を生み出すことよりも現在の PGAS 言語を発展させることを目的としているので，後者のハードウェア抽象化層が適していると言える．

4.2 グローバルアドレス

本節では低レベル通信ライブラリが扱うべきグローバルアドレスについて検討する．プロセッサコアが認識する論理アドレスのみを考慮する場合，グローバルアドレスが含むべき情報はプロセス識別子と論理アドレスのみである．しかし，インターコネクトデバイスの RDMA 機能を想定する場合，もう 1 段階複雑な構成情報が必要になる．

プロセッサコアでは基本的に実行中コンテキストの論理アドレス空間に対してしか参照が発生しない．これに対しインターコネクトデバイスでは多数のプロセスの論理アドレス空間に対する参照が並行して発生する．しかし，OS が管理する全ての実行コンテキストのページテーブルをインターコネクトデバイスに登録するのは非現実的である．そこでインターコネクトデバイスは RDMA 対象になる部分メモリ領域のページテーブルのみ登録を受け付け，登録された部分メモリ領域毎に独自の識別子を発行する．すなわちプロセッサコアが認識する論理アドレスに対し，インターコネクトデバイスでは登録済みメモリ領域の識別子と，登録済みメモリ領域内オフセットの組が同等の情報となる．

ここで，エクサスケールやさらに将来を考慮して桁あふれを起こさないグローバルアドレスの構成を検討する．まず 1 億プロセスを識別できるプロセス識別子には 30 ビットが必要である．次にインターコネクトデバイスが発行する登録済みメモリ領域の識別子を 16～24 ビット，登録済みメモリ領域内オフセットを最大 1TB とすると 40 ビットとな

る．グローバルポインタが Compare and Swap 対象になることを想定するとグローバルアドレスは 64 ビット内に収めるべきであるが，以上の合計は 64 ビットを超える．そこで識別子やオフセットのビット数の削減が必要になるが，適切なビット幅の構成は実行環境によって変わるので，ライブラリではグローバルアドレスのビット構成をハードコーディングせずに設定可能とするべきである．

4.3 メモリ管理機構の階層設計

まずハードウェア抽象化層のメモリ管理機能として，インターコネクトデバイスの部分メモリ領域登録機能を想定する．これは機能としてはインターコネクトデバイスへの部分メモリ領域登録機能そのものであり，前節で検討したデバイス依存の識別子，オフセットをグローバルアドレスの形式で出力する．この際，グローバルアドレスのプロセス識別子には該当プロセスの識別子が入る．

グローバルメモリ管理機構はハードウェア抽象化層の 1 つ上の階層に実装する．ライブラリの初期化時にグローバルメモリ用の Buddy System を構築する．まず管理構造体，ページ構造体配列および空きページフレームのメモリを割当，次に初期化して空きページフレーム・リストを構築し，その次に部分メモリ領域として登録する．最後にこのグローバルメモリ管理データ構造へのグローバルアドレスを全プロセスで共有する AllGather 集団通信を行い，初期化を完了する．初期化完了後はどのプロセスも RDMA を使用して，任意のプロセスから空きページを取得することができる．ここでインターコネクトデバイスが 3.4 節で想定したような，別プロセスからページテーブルを読み込ませる拡張機能を備えるならば，Buddy System は仮想アドレス機構に置き換え可能である．また，小サイズのメモリ割当高速化やメモリ使用効率を上げるために，上位層にヒープやスラブアロケータといった，より複雑なメモリアロケータを配置することもできる．このような RDMA を経由して利用できるメモリアロケータの一種として，著者らは非同期グローバルヒープを提案した[10]．

5. まとめと今後の課題

本論文では，将来の PGAS 言語の改良に貢献するグローバルメモリ管理機構を提案し，その実装方式を OS レベル，ランタイムシステムレベル，ライブラリレベルで検討した．また，グローバルメモリ管理機構に仮想メモリを導入する際にインターコネクトハードウェアに要求される機能も検討した．ライブラリレベル方式については通信ライブラリへの実装を想定し，ハードウェア抽象化層の設計，グローバルアドレスの構成，メモリ管理機構の実装方針について検討した．

今後の課題としてはライブラリレベル方式のグローバルメモリ管理機構の試作とともに，RDMA 経由でのメモリ割当に適したメモリ管理アルゴリズムの研究や，ランタイムシステムレベル方式や OS レベル方式の課題整理と解決策の検討が挙げられる．

参考文献

1) Message Passing Interface (MPI) Forum, http://www.mpi-forum.org/
2) Berkeley UPC - Unified Parallel C, http://upc.lbl.gov/
3) X10: Performance and Productivity at Scale, http://x10-lang.org/
4) OpenMP.org, http://openmp.org/wp/
5) Co-Array Fortran, http://www.co-array.org/
6) XcalableMP, http://www.xcalablemp.org/
7) The Chapel Parallel Programming Language, http://chapel.cray.com/
8) Parallel Programming Laboratory, http://charm.cs.uiuc.edu/
9) PCI-SIG, PCI Express, http://www.pcisig.com/specifications/pciexpress/
10) 安島雄一郎，秋元秀行，岡本高幸，三浦健一，住元真司: 非同期グローバルヒープの提案と初期検討，情報処理学会 第 138 回 HPC 研究会 (2013).