Vol.2013-HPC-138 No.10
2013/2/21

非同期グローバルヒープの提案と初期検討

安島雄一郎[†1,†2]　秋元秀行[†1,†2]　岡本高幸[†1,†2]　三浦健一[†1,†2]　住元真司[†1,†2]

分散メモリ並列計算機における片側通信では，プロセス間でメモリ書き込み，読み出しを行う RDMA プロトコルが一般的に使用される．従来の片側通信ライブラリにおける遠隔メモリの割当は，識別子の共有などのノード間同期の必要性を生じる．ノード間同期を避けるため，従来の片側通信ライブラリでは，一度割当てた遠隔メモリはなるべく解放しないで使用する．しかし，複数プロセスのメモリに配置したグローバルデータ構造を主たるデータ格納先として利用する場合，一度割当てたメモリを解放しない使い方は空きメモリ容量を圧迫する．本稿では内部的に片側通信を使用し，リモートメモリを非同期に割当て，解放できる非同期グローバルヒープを提案する．さらにデータ構造と排他制御を含む管理アルゴリズムを検討し，API 関数の実行時間を評価する．

Proposal and Preliminary Evaluation of Asynchronous Global Heap

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

One-sided communications in a distributed memory parallel computer usually uses an RDMA protocol which writes and reads memory between processes. In existing one-sided communication libraries, an allocation of a remote memory involves synchronization between nodes to share information such as a memory identifier. In order to avoid synchronizations between remote nodes, existing libraries recommend users to reuse allocated remote memory regions and not to deallocate them. However, keeping remote memory regions allocated suppress the capacity of free memory. In this paper, we propose Asynchronous Global Heap which uses one-sided communication internally, allocates a remote memory asynchronously and deallocates it asynchronously. We also investigate the data structure and management algorithm of Asynchronous Global Heap including the mutual exclusion control, and evaluate the average execution time of API functions.

1. はじめに

現在の High Performance Computing (HPC)分野では，分散メモリ型の並列計算機が主流である．サーバー用，PC 用，組み込み用などの既存プロセッサの設計資産，部品，ソフトウェアを利用してノードを構成し，ノード間をネットワークで接続して大型計算機を構成する，分散メモリ型並列計算機ではノード毎に独立に OS が走行し，メモリ管理が分散している．並列計算機内でノード間を接続するために使用されるネットワーク，および各ノードがネットワークに接続するために使用されるデバイスは，合わせてインターコネクトと呼ばれる．分散メモリ型並列計算機のインターコネクトは，独立したメモリ空間の間でデータ転送を行う．各ノードの OS は独立した複数プロセスのメモリ空間を管理するため，大型分散メモリ型並列計算機の総プロセス数は，数十万から数百万と膨大な数になる．

分散メモリ型並列計算機を構成する，もっとも一般的かつ安価なインターコネクトは Ethernet である．Ethernet を介した通信では，一般的に OS の TCP/IP socket ソフトウェアスタックが使用される．TCP/IP socket はコネクション指向アーキテクチャを有し，なおかつ通信の両端でソフトウェア処理が必要な両側通信である．

両側通信では受信側でのソフトウェア処理が必要なため割込みがかかり，頻繁に計算が中断される．並列処理においてノード毎の計算時間のインバランスは並列化効率を低下させる．Ethernet は安価であるものの，TCP/IP socket が両側通信であるため，並列計算には適していない．

両側通信による計算の中断を解決するインターコネクトとして TCP/IP offload engine [1]や InfiniBand [2]がある．これらのインターコネクトは通信処理が複雑なプロセス間メッセージ送受信機能をハードウェアで実装する．加えて，これらは一般的に，通信処理が単純でハードウェア処理に適した Remote Direct Memory Access (RDMA)通信機能を有する．RDMA は，応答側のプロセッサ処理が不要な片側通信であることを前提とした通信プロトコルである．具体的には要求側がアドレスを指定して，応答側のメモリを直接書き込み(Put)，読み出す(Get)．

コネクション指向通信では各プロセスが全プロセスに対する通信状態を管理しなくてはならない．そのため通信ソフトウェアスタックは主記憶容量を圧迫し，通信処理ハードウェアは通信可能なプロセス数が制限される問題がある．コネクションによるプロセス数の制限を解消するため，近年の大型並列計算機専用インターコネクトの多くはコネクションレスのアーキテクチャを採用する．このようなインターコネクトには富士通の Tofu インターコネクト[3][4], Cray の SeaStar [5] / Gemini [6] / Aries [7]インターコネクト，IBM の Blue Gene/Q Network Device (BG/Q ND) [8]などがあ

†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)

る．特に Tofu，Gemini，Aries，BG/Q ND はハードワイヤードロジックで設計された通信処理回路を有しており，高性能，省トランジスタ，低消費電力を兼ね備えている．

このように HPC 向けインターコネクトのハードウェアは進歩が著しい一方，通信ソフトウェアスタックによる利用は不十分である．現状では，デファクトスタンダードの通信ライブラリである MPI が，ハードウェア依存の片側通信 API を利用しているに留まっている．MPI はメッセージパッシング通信ライブラリであり，必然的にコネクション指向の管理構造を内在するので，コネクションレス・アーキテクチャの高スケーラビリティを利用するには適していない．片側通信を基盤として先進的なインターコネクトの高スケーラビリティを利用し，なおかつポータブルな新しい通信ライブラリが必要である．

本稿では，片側通信ライブラリの基盤となる新しいメモリ管理技術，非同期グローバルヒープを提案する．非同期グローバルヒープは RDMA 参照可能なメモリを片側通信で取得可能であり，さらにローカルメモリを圧迫しない特徴を有する．以降では 2 章で既存の片側通信ライブラリの課題を指摘し，3 章で非同期グローバルヒープを提案し，4 章で初期評価を行い，5 章で関連研究を紹介する．6 章では今後の課題について述べ，最後に 7 章でまとめる．

2. 従来の片側通信ライブラリ

過去にポータビリティを目的とした片側通信ライブラリとして，ARMCI [9], DCMF [10], DMAPP [11], CCI [12], 拡張 RDMA インタフェースなどが提案，実装されてきた．これらの片側通信ライブラリでは，RDMA 参照可能なメモリを新しく割当てる API，もしくは指定したローカルメモリを RDMA 参照可能な状態に登録する API が用意されている．ここで，本稿では RDMA 参照可能なメモリをグローバルメモリと呼ぶ．

ARMCI のグローバルメモリ割当 API，ARMCI_Malloc 関数は全プロセスが同期する．各プロセスにおいてローカルメモリを確保し，インターコネクトハードウェアに登録し，さらに全プロセスのグローバルメモリ情報を共有する．ARMCI_Malloc 関数以降の処理では全プロセスのグローバルメモリが参照可能になる．

DCMF では DCMF_Memregion_create 関数がグローバルメモリ割当 API である．DCMF_Memregion_create 関数はローカルメモリ割当，インターコネクトハードウェアへの登録を行う．関数内部にプロセス間の識別子交換を含まないので，ARMCI_Malloc 関数に比べて低オーバヘッドである．DMAPP の dmapp_mem_register 関数，CCI の cci_rma_register 関数は，指定したローカルメモリをインターコネクトハードウェアにグローバルメモリとして登録する．登録 API はグローバルメモリ識別子を返す．DCMF_Memregion_create 関数と比べて API 内でローカルメモリの確保を行わない分だけ API 自体は低オーバヘッドである．

DCMF，GNI，CCI ではグローバルメモリ割当，登録 API が識別子を返す．片側通信にはグローバルメモリ識別子が必要なので，片側通信を開始する前に識別子を含む制御情報を交換する必要があり，そのための両側通信機能が併設される．一方，拡張 RDMA インタフェースは Tofu インターコネクトの機能により，FJMPI_Rdma_reg_mem 関数に識別子を与えてグローバルメモリを登録できるので，制御情報の交換も片側通信で行える．

3. 非同期グローバルヒープの提案

RDMA に対応したインターコネクトにおいてグローバルメモリを新規に割当て，利用するにはローカルメモリの確保，グローバルメモリとして登録，利用先へのグローバルメモリ識別子の通知が必要である．この一連の手続きは，オーバヘッドの大きい両側通信と見做すことができる．片側通信ライブラリの利便性を向上するには，グローバルメモリ割当 API 内部にローカルメモリ確保，グローバルメモリ登録，識別子通知のいずれも含むべきではない．そこで，新しいグローバルメモリ割当方式を実現する基盤として，非同期グローバルヒープ機構を提案する．

3.1 非同期グローバルヒープの概要

本稿では，予め各プロセスに 1 つ用意され，先頭から順に切り出して使用されるグローバルメモリの塊を，グローバルヒープと定義する．グローバルヒープの未使用領域の先頭位置をグローバルブレークと呼ぶ．グローバルヒープは初期化時にローカルメモリ確保，登録，全プロセスの識別子交換を済ませる．そのため，グローバルヒープの利用の際は集団同期や識別子発行は不要である．未使用グローバルメモリの取得はグローバルブレークの変更で実現されるので，低オーバヘッドである．

上に述べたようにローカルメモリから確保したグローバルヒープは，ローカルメモリの空きメモリ容量を圧迫する．そこで我々は，ローカルメモリアロケータが空き領域を利用できる，非同期グローバルヒープを提案する．非同期グローバルヒープは，C 言語標準ライブラリの malloc 関数などのプログラミング言語処理系のローカルメモリアロケータを使用しないで実装され，かつローカルメモリアロケータが空き領域からグローバルメモリを取得するためのインタフェースを備える．

ローカルメモリアロケータがグローバルメモリアロケータと同様にグローバルブレークを使用して非同期グローバルヒープの先頭から空き領域を使用すると，干渉によりフラグメンテーションが深刻化する．そこで，ローカルメモリアロケータは空き領域を末尾から使用する．末尾側の使用済みグローバルメモリの先頭位置をグローバルリミットと呼ぶ．末尾側からのメモリ取得もグローバルリミットの変更で実現されるので，低オーバヘッドである．

非同期グローバルヒープの空き領域はグローバルブレークからグローバルリミットの直前までである．ここで，グローバルブレークはグローバルメモリアロケータによって，グローバルリミットはローカルメモリアロケータによって変更され得る．よって，グローバルメモリアロケータおよびローカルメモリアロケータが非同期グローバルヒープの空き領域を確保する際は，排他制御が必要になる．

さらに我々の提案する非同期グローバルヒープは管理情報をグローバルメモリに置き，別のプロセスから未使用領域のメモリを取得できるようにする．この機能により，受信側がまだ待ち受けていない unexpected なメッセージを，送信側から動的に受信側グローバルメモリを確保して Put してしまうような，従来の片側通信ライブラリでは不可能であったアルゴリズムを実現できる．

以上に説明した非同期グローバルヒープのメモリマップを図 1 に示す．

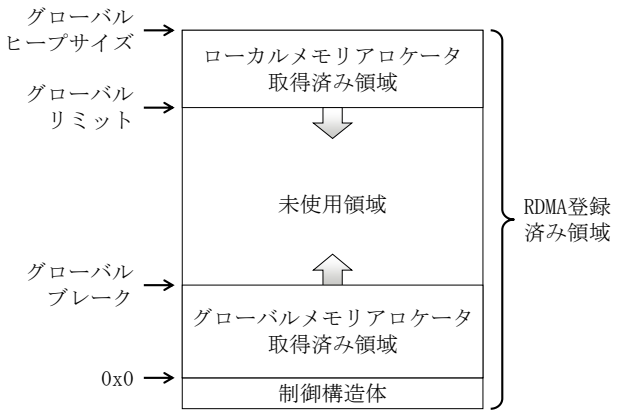


図 1　非同期グローバルヒープのメモリマップ

Figure 1　A memory map of an Asynchronous Global Heap.

3.2 API 関数の定義

我々が提案する非同期グローバルヒープの API 関数 5 つの定義を示す．

- ginit – 非同期グローバルヒープの初期化
  書式: int ginit(long size);
  返り値: 成功(0), エラー(-1)
- gbrk – グローバルブレークの変更
  書式: int gbrk(int proc, long old_gbrk, long new_gbrk);
  返り値: 成功(新しいグローバルブレーク),
  エラー(古いグローバルブレーク)
- sgbrk – グローバルメモリの取得
  書式: int sgbrk(int proc, long increment);
  返り値: 成功(変更前のグローバルブレーク),
  エラー(-1)
- gglimit – 未使用領域情報を取得
  書式: int gglimit(int proc, long* gbrk, long* glimit);
  返り値: 成功(0), エラー(-1)
- sglimit – グローバルリミットを変更
  書式: int sglimit(long new_glimit);
  返り値: 成功(0), エラー(-1)

非同期グローバルヒープ API は Linux の brk システムコールに似たインタフェースを採用した．ここで，グローバルブレークはポインタではなくグローバルヒープ先頭からの相対オフセットである．また，別プロセスからグローバルメモリを取得するため，gbrk，sgbrk，gglimit 関数はプロセス番号を指定できる．Sgbrk 関数が引数に古いグローバルブレークを取るのは，エラーチェックのためである．

3.3 管理情報の実装指針

非同期グローバルヒープ実装は全プロセスの非同期グローバルヒープについて，グローバルメモリ識別子およびグローバルメモリサイズを管理する．これらは初期化時に収集され，全プロセスにコピーが配布される静的な管理情報である．

各プロセスにおける非同期グローバルヒープの動的な状態は各プロセスが一元的に保持し，コピーは配布しない．動的状態を表現する変数は，ロック変数，グローバルブレーク変数(gbrk)，グローバルリミット変数(glimit)である．これら動的状態変数を制御構造体としてまとめ，グローバルメモリ上に置く．静的管理情報を小さくするため，データ領域と制御構造体を連続したメモリ領域に配置し，グローバルメモリ識別子を共有することが望ましい．

制御構造体は複数プロセスから並行して参照され，その際にロック変数で排他制御される．インターコネクトハードウェアの RDMA が Atomic Compare and Swap (CAS)参照機能を持たない場合，ロック変数は排他制御を行うための別のデータ構造に置き換える．

3.4 API 関数の実装指針

ginit 関数はローカルメモリ上で非同期グローバルヒープのデータ領域と動的状態変数領域を初期化し，グローバルメモリ登録する．データ領域は入力値 size の大きさを確保する．自プロセスの性的管理情報（グローバルメモリ識別子および非同期グローバルヒープサイズ）を全プロセスに通知し，全プロセスから通知された静的管理情報をライブラリ内で保存する．

gbrk 関数はプロセス番号 proc の制御構造体のロックを RDMA Atomic CAS で取得し，プロセス番号 proc の制御構造体の gbrk, glimit を RDMA Get で読み出す．入力値 old_gbrk が gbrk の値と一致し，入力値 new_gbrk が glimit の値を超えていなければ, 入力値 new_gbrk をプロセス番号 proc の制御構造体の gbrk に RDMA Put で書き込む. 最後にプロセス番号 proc の制御構造体のロックを RDMA Atomic CAS で解放する．

sgbrk 関数はプロセス番号 proc の制御構造体のロックを RDMA Atomic CAS で取得し，プロセス番号 proc の制御構造体の gbrk, glimit を RDMA Get で読み出す．gbrk と glimit の差が入力値 increment 以下ならば，gbrk の値に increment の値を足した値をプロセス番号 proc の制御構造体の gbrk

に RDMA Put で書き込む. 最後にプロセス番号 proc の制御構造体のロックを RDMA Atomic CAS で解放する.

gglimit 関数はプロセス番号 proc の制御構造体のロックを RDMA Atomic CAS で取得し，プロセス番号 proc の制御構造体の gbrk, glimit を RDMA Get で読み出す. 最後にプロセス番号 proc の制御構造体のロックを RDMA Atomic CAS で解放する.

sglimit 関数はローカルにある制御構造体のロックを RDMA Atomic CAS で取得し，入力値 new_glimit を制御構造体の glimit にローカルメモリ上で直接書き込む．最後に制御構造体のロックを RDMA Atomic CAS で解放する.

3.5 想定される使い方

3.1 節で述べたように非同期グローバルヒープはグローバルメモリアロケータから利用されることを想定している. ここでヒープのデータ構造とアルゴリズムはフラグメンテーションを生じる特性があるので，グローバルメモリアロケータのアルゴリズムが高度化すると，非同期グローバルヒープとそれ以外のグローバルメモリ供給源を併用すると想定される.

想定される非同期グローバルヒープ以外のグローバルメモリ供給源は，非同期グローバルヒープよりもオーバヘッドが大きい代わりにフラグメンテーションを起こさない機構，例えばローカルメモリにおける mmap システムコールに相当する機構である．このようなグローバルメモリ割当 API は, 内部で両側通信を行えば, ローカルメモリ確保，グローバルメモリ登録，識別子通知によって簡易に実装できる．また，将来的には OS によるローカルのメモリページ管理機構と連携して，両側通信による同期を避けることもできる.

以上に述べたような高度なグローバルメモリアロケータ・アルゴリズムにおいては，非同期グローバルヒープは小さいサイズの領域取得で使用されると想定される．一般的にメモリアロケータはアプリケーションプログラムから小さいメモリ領域が解放されても，メモリを実際には解放せず，メモリアロケータ自身のフリーリストで管理して再利用する．加えてヒープはフラグメンテーションへの対処が難しいので，メモリアロケータがブレークポインタを戻してヒープから取得したメモリを解放する機会は非常に少ない，と想定される．よって gbrk 関数，sgbrk 関数ともグローバルブレークが増える場合の遅延時間が小さくなるように最適化すればよい.

また，非同期グローバルヒープでは，ローカルメモリアロケータが sglimit 関数で空き領域を取得することを想定している．これは C 言語の malloc 関数，C++言語の new 演算子などのローカルメモリアロケータの中から呼び出される動作であるため，遅延時間を小さくする最適化が必須である.

4. 初期評価

提案した非同期グローバルヒープの初期評価として gbrk, gglimit，sglimit 関数の実行時間を評価する．前期 3 つの関数を選んだ理由は，上位のメモリアロケータから頻繁に呼ばれることが想定されることである．一方，ginit および sgbrk 関数の実行時間は評価しない．その理由は，ginit 関数については上位のメモリアロケータからの呼び出し回数が少ないと想定されることであり，sgbrk 関数については gbrk 関数とほぼ同じ処理であることである.

4.1 評価環境

評価環境にはスーパーコンピュータ「京」のプロトタイプ機を使用した．評価環境のプロトタイプ機は富士通の沼津工場に設置されており，搭載されたプロセッサは SPARC64TM VIIIfx, 動作周波数 2GHz, コア数は 8 である．インターコネクト Tofu インターコネクトであり，5GB×双方向のネットワーク・インタフェースを 4 つ搭載する．ネットワーク・トポロジーは 6 次元メッシュ／トーラスであり，ノードは 6 次元座標(X,Y,Z,A,B,C)で識別される.

Tofu ネットワーク・インタフェース(TNI)は Put および Get の RDMA 通信機能に対応し，TNI 1 つあたりカーネル用 1 本，ユーザー用 2 本の制御キュー(CQ)を搭載する．各 CQ は専用のパケット組み立てエンジンを持ち，TNI 1 つあたり最大 3 つの送信コマンドを並列に実行できる．また，Tofu インターコネクトの RDMA 通信はリモートメモリ参照順序保証するストロングオーダー･フラグ機能を備える.

4.2 測定プログラム

本評価の測定プログラムは，2 プロセスの MPI プログラムである．評価では RDMA 通信が必要であるので，MPI と併用して Tofu ライブラリ(tlib)を使用した. tlib は Tofu インターコネクトのハードウェア直接利用する低レベル API である．測定プログラムは 1 ノードあたり 1 プロセスを割当て, 2 ノードを使用して実行した. 1 ノードあたり 1 プロセスの場合，MPI は TNI あたりユーザー用 CQ を 1 本だけ使用する．tlib を使用した RDMA 通信では，各 TNI で MPI が使用していない, もう 1 本のユーザー用 CQ を使用した.

本評価ではランク 0 がランク 1 の非同期グローバルヒープに対して gbrk，gglimit 関数を実行する際の実行時間と，ランク 0 がランク 0 自身の非同期グローバルヒープに対して gbrk，gglimit，sglimit 関数を実行する際の実行時間を測定した．各関数は片側通信しか行わないので，測定対象はランク 0 における一連の RDMA 通信の実行時間である. 本評価の各関数は TNI 0 番を使用して RDMA 通信を行った. また，ランク 0 自身の非同期グローバルヒープに対する関数実行では排他制御のみ RDMA 通信で行い, 制御変数の参照はプロセッサ命令で行った.

評価では対象の関数を 1001 回連続で実行し，2 回目開始から 1001 回目終了までの 1000 回分の時間を測定して平均

値実行時間を求めた．全評価を通じて，非同期グローバルヒープにアクセスするプロセスはランク 0 のみであり，計測中に排他制御の競合は起きなかった．

(1) **RDMA Atomic CAS エミュレーション**

Tofu インターコネクトは RDMA Atomic CAS 機能を持たないので, 測定プログラムではプロセッサで RDMA Atomic CAS のエミュレーションを行う．このためにランク 0，ランク 1 の両プロセスで，RDMA Atomic CAS のエミュレーション用の子スレッドを作成した．子スレッドは受信バッファをポーリングする. Atomic CAS 要求が書き込まれると子スレッドはプロセッサの cas 命令で CAS メモリ参照を実行し，結果を要求元に Put 送信する．評価関数と干渉しないように子スレッドは親スレッドとは別の CPU コアに割り付け，結果の Put 送信には TNI 1 番を使用した．

(2) **リモートメモリ参照順序保証**

別プロセスにある，排他制御で保護された変数を参照する場合，通常は RDMA Atomic CAS でロックを取得し，成功したら RDMA Get で変数を読み出す．ここで RDMA 通信にリモートメモリ参照の順序保証機能があれば，RDMA Atomic CAS 要求と RDMA Get 要求を続けて送出することができる．ロックが取得できなかった場合は RDMA Get によるデータ転送が無駄になるが，非同期グローバルヒープの管理情報のように参照すべきデータ量が小さい場合には遅延短縮の手段として有効である．本評価ではリモートメモリ参照順序保証がある場合とない場合の両方の実行時間を測定した. リモートメモリ参照順序保証には Tofu インターコネクトのストロングオーダー･フラグ機能を使用した．

(3) **プロセッサ命令・RDMA 間のメモリ参照不可分性**

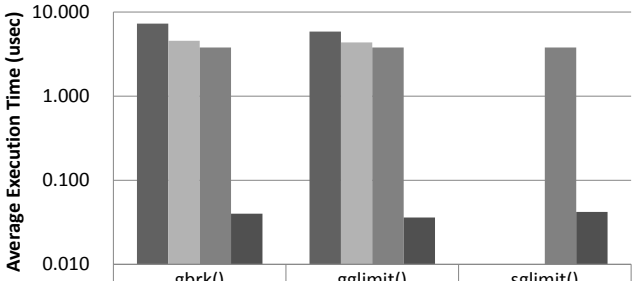
プロセッサ命令による CAS は一般的に, メモリを共有するプロセッサコア間でのみ読み出しと書き込みの不可分性が保証される．インターコネクトの RDMA Atomic CAS は一般的に，ネットワーク・インタフェースが読み出しと書き込みの不可分性を保証する．すなわち，通常はプロセッサ命令による CAS と RDMA Atomic CAS の間には不可分性の保証がない．

ここでプロセッサ命令による CAS と RDMA Atomic CAS の間で不可分性を保証できると，ランク 0 がランク 0 自身の非同期グローバルヒープの管理情報を参照する際に，プロセッサ命令による CAS で排他制御することが可能になる．本評価では RDMA Atomic CAS をプロセッサ命令の CAS でエミュレーションしているので不可分性が保証されている．不可分性が保証されていない場合と不可分性が保証されている場合の両方の実行時間を測定した．

4.3 結果

図 2 に評価結果を示す. 図は計測対象別の平均実行時間を示す縦の棒グラフであり，時間の単位はマイクロ秒である．各関数の平均実行時間は 4 種類測定した．Remote はランク 1 の非同期グローバルヒープに対する平均実行時間，Remote+はリモートメモリ参照順序保証がある場合の非ランク 1 の同期グローバルヒープに対する平均実行時間，Local はランク 0 自身の非同期グローバルヒープに対する平均実行時間，Local+はプロセッサ命令・RDMA 間のメモリ参照不可分性がある場合のランク 0 自身の非同期グローバルヒープに対する平均実行時間である．

Local の実行時間は Remote より 35～48%短い．また，Remote では gbrk 関数と gglimit 関数の実行時間に差があるが，Local ではほぼ同じ実行時間となっている．Remote+の実行時間は Remote より 25～37%短い．また，Remote+では gbrk 関数と gglimit 関数の実行時間の差が Remote より小さい．Local+は Local に比べて約 100 分の 1 と大幅に実行時間を短縮した．



	gbrk()	gglimit()	sglimit()
Remote	7.285	5.867	
Remote+	4.557	4.358	
Local	3.794	3.794	3.791
Local+	0.040	0.036	0.042

図 2　非同期グローバルヒープ API 関数の実行時間

Figure 2　Evaluated execution time of Asynchronous Global Heap API functions.

4.4 考察

Remote は管理情報も RDMA で参照するため，関数の処理内容で RDMA 回数が変わり, 実行時間に差が出ると考えられる．Local は排他制御のためのロック取得・解放で 2 回しか RDMA が必要でないため実行時間が短く, また関数の処理内容の種類の影響が小さい．Remote+ではロック取得と管理情報所得，管理情報更新とロック解放をそれぞれ連続要求できるため，Local に近い実行時間となった．Local+以外では関数実行に数マイクロ秒オーダーの時間がかかるのに対し，Local+では数十ナノ秒の実行時間で完了する. これは RDMA が不要でプロセッサ命令だけで処理できることが理由である．

以上の結果から，非同期グローバルヒープの取得は一般に数マイクロ秒オーダーの時間が必要であり，この実行時間を隠蔽できる用途での使用が望ましい．また，プロセッサ命令による CAS と RDMA Atomic CAS の間での不可分性保証はプロセス自身が持つ非同期グローバルヒープからメモリを取得するオーバヘッドを大幅に削減するので，ローカルメモリアロケータの非同期グローバルヒープ利用には大変効果がある．現在のシステムでは，IO バスがプロトコルとして Atomic DMA Read and Write トランザクションを

定義していても，プロセッサ側のキャッシュコヒーレンシプロトコルはプロセッサ命令による CAS と DMA の間に不可分性を保証する設計になっていない．プロセッサ命令による CAS と DMA の間に不可分性を保証するにはインターコネクトハードウェアが IO バスではなくキャッシュコヒーレントバスに接続するか，プロセッサ側のキャッシュコントローラ設計が変わる必要がある．

5. 関連研究

5.1 DMAPP Symmetric Heap

DMAPP はプロセス毎に Symmetric Heap と呼ばれる固定サイズのメモリを確保する. サイズは環境変数で指定する．Symmetric Heap を使用するメモリアロケータが全プロセス同期動作することを想定している．DMAPP は内部で Symmetric Heap をグローバルメモリ登録し，全プロセス分の識別子を共有する．Symmetric Heap のグローバルメモリ識別子はアプリケーションユーザーから隠蔽されており，暗黙に使用される． DMAPP で実装されているメモリアロケータ API の定義は void* dmapp_sheap_malloc(size_t size) であり，引数の数は malloc 関数と同じである．Symmetric Heap は本稿で定義した非同期グローバルヒープの機能は持たないが，グローバルヒープに近い機能を持つ機構と位置付けられる．

5.2 kmp_malloc

David Kuck [13]の Kuck and Associates, Inc. は共有メモリ計算機向けに，スレッド毎に独立したヒープを持たせてメモリアロケータの排他制御負荷を削減する kmp_malloc 関数を開発した．kmp_malloc 関数で確保したメモリは別のスレッドから解放できる特徴がある．非同期グローバルヒープ上に別スレッドのグローバルメモリを取得して返すグローバルメモリアロケータを構築する場合，kmp_malloc 関数と同様にどのプロセスからでも解放可能とすることが望ましい. kmp_malloc 関数は Intel OpenMP コンパイラに組み込まれている．

5.3 HeapCreate

Microsoft の Win32 API にはスレッド毎に個別のヒープを作成する HeapCreate 関数[14]がある．ただし HeapCreate 関数で作成したヒープ上でメモリ割当てを行った場合，kmp_malloc 関数を使用した場合とは異なり，同一スレッドで解放する必要がある．

5.4 Arena

ptmalloc [15]は共有メモリのスレッド間でメモリアロケータの排他制御負荷を分散させるため，Arena と呼ばれるサブヒープを導入した．グローバルメモリアロケータが複数プロセスの非同期グローバルヒープを一体として扱う場合，各プロセスの非同期グローバルヒープがサブヒープに相当するので，Arena のデータ構造と共通点がある．

6. 今後の課題

今後の課題としては，非同期グローバルヒープのフラグメンテーションを解決するグローバルメモリアロケータのアルゴリズム検討，そのためにグローバルメモリをページ単位で割当る mmap システムコール相当機能の検討が挙げられる．また，非同期グローバルヒープのサイズ拡張機能も検討課題である．

7. まとめ

本稿では，片側通信ライブラリの基盤となる新しいメモリ管理技術，非同期グローバルヒープを提案し，初期評価を行った. 非同期グローバルヒープは RDMA 参照可能なメモリを片側通信で取得可能であり，さらにローカルメモリを圧迫しない特徴を有する．

参考文献

1) Chelsio Communications, http://www.chelsio.com/
2) InfiniBand® Trade Association, http://www.infinibandta.org/
3) Ajima, Y., Inoue, T., Hiramoto, S., Shimizu, T., Takagi, Y.: The Tofu Interconnect. IEEE Micro, vol. 32, no. 1, pp.21-31 (2012).
4) Ajima, Y., Sumimoto, S., Shimizu, T.: Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers. IEEE Computer, vol. 42, no. 11, pp.36-40 (2009).
5) Brightwell, R., Pedretti K. T., Underwood, K. D. and Hudson, T.: SeaStar Interconnect: Balanced Bandwidth for Scalable Performance, IEEE Micro, vol. 26, no. 3, pp.41-57 (2006).
6) Alverson, R., Roweth, D. and Kaplan, L.: The Gemini System Interconnect, IEEE 18th Annual Symposium on High Performance Interconnects, pp.83-87 (2010).
7) Faanes, G., et al.: Cray Cascade: a Scalable HPC System based on a Dragonfly Network, In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Article No. 103 (2012).
8) Chen, D., et al.: Looking Under the Hood of the IBM Blue Gene/Q Network, In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Article No. 69 (2012).
9) ARMCI, http://www.emsl.pnl.gov/docs/parsoft/armci/
10) Kumar, S. et al.: The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer, In Proceedings of the 22nd Annual International Conference on Supercomputing, pp.94-103 (2008).
11) Bruggencate, M. T., Roweth, D.: DMAPP – An API for One-sided Program Models on Baker Systems, In Proceedings of the Cray User Group 2010.
12) Atchley, S., et al.: The Common Communication Interface (CCI), IEEE 19th Annual Symposium on High Performance Interconnects, pp.51-60 (2011).
13) David Kuck, http://www.computer.org/portal/web/awards/David-Kuck
14) HeapCreate function (Windows), http://msdn.microsoft.com/library/windows/desktop/aa366599(v=vs.85).aspx
15) Gloger, W.: Dynamic memory allocator implementations in Linux system libraries, http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html