Vol.2016-HPC-155 No.28
2016/8/9

PGAS向け低水準通信レイヤーのマルチスレッド実装

遠藤 亘[1,a)]　田浦 健次朗[1,b)]

概要：低水準通信レイヤー (Low-Level Communication Layer) は，分散メモリ型計算機におけるインターコネクトの API を抽象化するレイヤーであり，PGAS やタスクスケジューラといった高水準システムを移植可能な形で開発するのに必要とされる．低水準通信レイヤーの実装は高水準システムの性能を大きく左右するため，そのチューニングはシステム全体の高速化に不可欠である．しかし，既存の低水準通信レイヤーには問題点が多く，最新のハードウェア機能に対応できておらず，マルチコア向けのチューニングも充分でないものが多い．本報告では，インターコネクトの API に近い形で低水準通信レイヤーの API を再定義してオーバーヘッド削減を図るとともに，それに基づいた実装として特に Tofu インターコネクト向け実装を例に説明する．具体的には，マルチスレッド対応やオーバーヘッド削減のための実装手法について紹介し，マイクロベンチマークによる評価結果を報告する．

Multithreaded Implementation of Low-Level Communication Layer for PGAS Systems

Wataru Endo[1,a)]　Kenjiro Taura[1,b)]

Abstract: A low-level communication layer (LLCL) is an abstraction layer of interconnect APIs on distributed memory systems, which is required to develop portable high-level systems such as PGAS systems and task schedulers. Because the implementation of an LLCL heavily affects the performance of high-level systems, optimizing the LLCL is important to speed up the whole system. However, existing LLCLs have some issues including lacks of latest hardware features and optimization for multi-core environments. This paper tries to redefine a hardware-aware API for LLCLs to reduce the overhead and implement it especially for the Tofu interconnect. In particular, we show the methods to optimize LLCLs for multi-threading and reduce its overhead, and we also report the microbenchmark results of our implementation.

1. 序論

近年のスーパーコンピュータの動向として，クラスタ全体としての計算ノード数の増加だけではなく，1 ノード当たりコア数の増加も計算性能向上に寄与している．既に広く普及した数十コアのマルチコアプロセッサに留まらず，数百コアを持つメニーコアプロセッサも採用されつつあることから，1 ノード当たりの計算能力は今後も上昇し続けると予想される．

ノード内のコア数が増加する傾向の中，ノード間通信のシステムには新たな課題が突きつけられている．理想的にはノード内コア数に比例して 1 ノード当たりの通信資源も増加するべきであるが，現実のハードウェアはコア数よりも通信資源が少ないため，ノード内コアで通信資源を共有せざるを得ない．この際，コア間の排他制御に粗粒度のロックを用いるだけでは，通信発生毎に衝突が発生してノード内のスケーラビリティ低下の原因となる．この問題を解決するには，多数のコアからの通信要求を高速にスケジューリングできるランタイムシステムが必要である．

近年の通信ハードウェアの性能に着目すると，バンド幅の増加は依然として続いているのに対して，レイテンシの低減はあまり進んでいないという状況になっている．今後はレイテンシ削減よりも隠蔽が重要になると見込まれ，通信システムにはレイテンシ隠蔽に適した API や実装を提

[1] 東京大学 大学院 情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
[a)] wendo@eidos.ic.i.u-tokyo.ac.jp
[b)] tau@eidos.ic.i.u-tokyo.ac.jp

供していくことが求められる．

プログラミングモデルの観点では，現在の分散メモリ型計算機において Message Passing Interface (MPI) [1] によるプログラミングが一般的だが，最近では共有メモリに近いモデルとして Partitioned Global Address Space (PGAS) [2][3][4][5] も注目されており，高生産かつ高性能なモデルに段階的に移行していくと予想される．

高性能な PGAS システムの実現のために重要なのが，通信処理を実際に実行するモジュールであり，本稿で定義する低水準通信レイヤー (Low-Level Communication Layer) である．このような抽象化が必要な背景には，PGAS のような高水準な処理系を開発するにあたって，インターコネクトのハードウェアが更新される度に処理系全体を再開発することは非生産的であることが挙げられる．そして，低水準通信レイヤーの性能は，PGAS を含めた高水準システムの性能に大きく影響するため，その高速化はシステム全体にとって重要である．

本研究では，PGAS に特化した低水準通信レイヤーについて，マルチコアプロセッサ上での実行に適した実装を提案する．現在の PGAS 向け低水準通信レイヤーはマルチコアプロセッサ上で高速に動作するものが知られておらず，その問題がそのまま PGAS の性能にも影響している．そこで，低水準通信レイヤーに関して，マルチスレッド化やレイテンシ隠蔽に関する問題を中心に，API とその実装の両面から取り上げていく．

本研究での主な貢献は次の 3 つである．

- マルチスレッド環境で高速に動作し，オーバーヘッドの少ない実装を提供できるように低水準通信レイヤー API を定義した．
- Tofu インターコネクト [6] 上において別スレッドへの移譲機構を実装し，マイクロベンチマークの結果では，マルチスレッド対応によるレイテンシ増加を 19%に抑え，15 スレッドでのメッセージレートの低下を 12%まで抑えた．
- InfiniBand [7] において，通信要求を自動的に集約してインターコネクトに送出する機構を実装することでメッセージレートを向上させた．

本稿の構成は以下の通りである．2 章では，低水準通信レイヤー全般と，MPI のマルチスレッド対応に関する既存研究について述べる．3 章では，提案する API と，その実装方法について，各インターコネクトの詳細に立ち入らずに紹介する．4 章では，Tofu と InfiniBand を例に，各種インターコネクトに特化した実装技術について解説する．5 章では，評価手法として用いるマイクロベンチマークについて述べる．6 章では，ベンチマークの結果と考察を述べる．そして 7 章が結論である．

2. 関連研究

2.1 低水準通信レイヤー

GASNet [8][9] は PGAS 向けの低水準通信レイヤーとして広く知られており，PGAS 言語である Unified Parallel C (UPC) [2] を始めとして複数の使用例がある．GASNet は複数の（conduit と呼ばれる）実装を持ち，MPI-1 の上で動作する AMMPI に加えて，InfiniBand Verbs などのインターコネクトの API を直接使用する実装もある．

GASNet が提供する主要な機能は次の 3 つである．GASNet 以外のシステムもこれらの機能をサポートしている場合が多い．

Active Messages (AM) [10] メッセージを他ノードに送信し，そのメッセージを引数とした関数を実行させる機能で，Remote Procedure Call (RPC) とほぼ同義である．但し，AM と言った場合には HPC 向けの軽量な実装を指す場合が多い．AM の実装手法には 2 種類あり，CPU 割り込みによるものとポーリングによるものがある．

Remote Memory Access (RMA) 他ノード上のメモリに対して読み書きを実行する．インターコネクトが対応していれば，ハードウェア機能である Remote Direct Memory Access (RDMA) によって高速化が可能であるが，それが不可能な場合は AM を用いて代替実装が行われる．

集団通信 (Collective Communication) GASNet の実行モデルも MPI と同様に SPMD であるため，集団通信によって全プロセス間で同期しながらデータをやり取りすることができる．

GASNet の問題点として，RMA の一種であるリモートアトミック（他ノードのメモリをアトミックに書き換える機能）のような比較的新しいインターコネクトの機能がサポートされていないことが挙げられる．GASNet の登場時期は 2002 年頃であり，API は当時のハードウェア事情を反映したものとなっている．

ARMCI [11] も PGAS 向けの低水準通信レイヤーの一つで，PGAS ライブラリである Global Arrays [3] のために開発された．ARMCI の機能は GASNet に類似しているが，AM はサポートされていない．最近になって，Global Arrays の低水準通信レイヤーは ARMCI から ComEx [12] というライブラリに置換された．ComEx は MPI との相互運用性を目指して実装されている．ComEx は未だ開発途上のため，基本的な RMA の機能しか提供しておらず，ARMCI 同様に AM はサポートされない．

OpenFabrics [13] は InfiniBand ドライバを開発しているプロジェクトであり，そのサブプロジェクトである libfabric [14] というライブラリでは InfiniBand Verbs よりも抽象

度の高い API を提案している．プロジェクトの性質上，libfabric の API は InfiniBand に強く影響されており，サポートされているのは RMA のみである．

Unified Communication X (UCX) [15] も低水準通信レイヤーであり，GASNet と同様に AM，RMA，集団通信の 3 つを提供する他，MPI の実装に必要とされるタグマッチング機能を備えるなど，API としては網羅性が高い．

MPI は低水準通信レイヤーよりも上位に位置するが，それに相当する機能も含んでいる．MPI-2 からは RMA が導入され，MPI-3 ではより RDMA に近いモデルも登場したため，この上で PGAS を構築することもできる [16]．MPI の仕様は広範に渡るが，PGAS の下位レイヤーとして必要な機能はその一部にすぎない．また，MPI-3 の片方向通信は RDMA だけで実装できない場合もあり，リモート CPU の介在を要求する場合もある [17] など，現実のインターコネクトからの乖離も見受けられる．

2.2 通信のマルチスレッド化

2.1 節で取り上げた既存の低水準通信レイヤーのいずれも，マルチスレッド実行に関する特別な配慮はなされていない．GASNet や ARMCI が開発された当時はマルチコアプロセッサが一般的でなかったという事情があるが，近年になって登場したライブラリですらマルチスレッドの重要性を認識しているものは少ない．

一方，MPI の開発者の間では，通信とマルチスレッドの関係についての議論が盛んに行われている．そのきっかけの一つが Balaji ら [18][19] の研究で，MPI の実装の 1 つである MPICH [20] をマルチスレッド環境で実行すると，MPICH 内の粗粒度ロックによってスレッド数増加とともにメッセージレートが大幅に低下していくことを示した．

Vaidyanathan ら [21] は，“Software Offloading” という手法によって MPI における並行性と非同期性の両方を向上させることを提案している．その中心的なアイディアは，「コマンドキュー」と呼ばれるロックフリーな循環バッファに通信要求を溜めていき，通信専用スレッドがそれを処理するというものである．このような手法によって，通信専用スレッドのみが実際の通信を発行しているため，通信資源をロックで排他制御する必要がなくなって並行性が向上する．また，通信を要求しているスレッドは，コマンドキューへの挿入さえ完了すれば，`MPI_Isend()` のようなノンブロッキング関数において即座に呼び出し元に復帰できるため，非同期性も向上しているといえる．

Amer ら [22] は，OS が提供するミューテックスが非公平であるために，それに依存した MPI のランタイムの性能が低下することを示した．その解決策として，First-In First-Out (FIFO) で調停されるロックの使用に加えて，通信を進行 (progress) させる処理のために優先度をつけたロック手法を提案している．

Lu ら [23] は，ユーザーレベルスレッド (ULT) を用いて MPI の通信レイテンシを隠蔽する機構を提案している．ULT はコンテキストを高速に切り替えることができるので，ユーザのスレッド内で通信を待ち合わせる間に計算をオーバーラップさせることが容易になる．

PGAS システムである ACP [24][25] でも通信のマルチスレッド化に取り組んでおり，通信要求の管理には Vaidyanathan らと同様に循環バッファによる手法を採用している．ACP の Tofu 上の実装はソースコードが公開されていないが，設計上は本稿の Tofu 用実装と類似する点が多いため，6.1 節で比較を行う．

3. 提案手法

提案する処理系は，本研究グループで開発している PGAS 処理系である MGAS-2 [26] から，モジュールとして切り出されたものであり，Software Offloading の手法を低水準通信レイヤーとして実装する．MPI や PGAS よりも低水準なレイヤーとしてこの機構を実装することには，次のような利点がある．

- システムの直交性が向上し，並行性バグを減らせる．スレッドセーフなシステムを構築するにはシステムを細かいモジュールやオブジェクトに分割し，それらごとにスレッドセーフを満たすよう設計することが望ましい．
- 分散メモリ用の複数のシステムをマルチスレッド環境で両立させることが容易となる．通信を必要とするのは PGAS だけではなく，例えば分散タスクスケジューラも同様である．
- Offloading する通信の単位がインターコネクトの API に近くなるため，それを活かしたチューニングが可能である．4.2 章で述べる InfiniBand で通信を集約する機構がその一例である．

提案処理系の機能としては，GASNet と同様に RMA，AM，集団通信の 3 つを実装した．但し，プロセスの起動/終了など一部の処理に MPI を使用しており，集団通信の処理も基本的に MPI に移譲される．PGAS にとって重要なのは RMA であるため，本稿では主に RMA の実装手法を解説していく．

3.1 API

本研究では，既存の低水準通信レイヤーの API を参考にしながら，Tofu や InfiniBand で良好な性能が得られるよう設計を行った．**リスト 1** に，例として RDMA READ の API を示す．RDMA WRITE やリモートアトミックなどでもほとんど同様の API が提供され，それらも含めて全てスレッドセーフである．

提案する API の特徴的な点は次のように挙げられる．

- `try` という接頭辞は「この関数が失敗しうる」ことを

```
using process_id_t = /*integer*/;
struct remote_address { size_t offset; /*...*/ };
struct local_address  { size_t offset; /*...*/ };
struct callback { void (*f)(void*); void* d; };
struct read_params {
  process_id_t   src_proc;
  remote_address src_raddr;
  local_address  dest_laddr;
  size_t         size_in_bytes;
  callback       on_complete;
};
bool try_read_async(const read_params&);
```

リスト 1　RDMA READ の API

示す．このような仕様は，後述する Offloading の実装を踏まえている．通信要求失敗の原因は，通信資源やコマンドキューのサイズが有限であるため，通信要求が溜まりすぎてその処理が追いつかなくなることである．

- async という接尾辞は「完了通知が**コールバック関数**によって実現される」ことを意味する．この関数が成功して復帰することはあくまで通信要求が生成されたことを意味するのみで，実際の完了通知はコールバック関数の実行をもって行われる．MPI や GASNet などの既存のライブラリでは，“リクエスト” や “ハンドル” などと呼ばれるオブジェクトに問い合わせることで通信完了を調べるものが多いが，ユーザにとっては問い合わせの手法が限定されているために自由度が低い．最も単純な完了通知手法はフラグをセットすることであるが，状況によっては条件変数による待ち合わせも考えられるため，より柔軟な仕組みが必要といえる．ポーリングを実行したスレッドがコールバック関数を直接実行するモデルは柔軟でかつ軽量であるため，本提案ではこれによって完了通知を行うこととした．
- RDMA で読み書きされるアドレスは，**ポインタよりも大きな構造体**である．ユーザは offset メンバを増減することでアドレスを指定する．これは，多くのインターコネクトにおいて RDMA を実行する際，メモリレジストレーション時に割り当てられる ID が必要であるという事情が背景にある．API としてはポインタだけを授受し，内部では ID を逆引きするような実装では，逆引きによるオーバーヘッドが避けられない．
- リモートバッファだけでなく，ローカルバッファに対してもメモリレジストレーションを要求する．これに対し，既存処理系ではローカルバッファのレジストレーションを求めないものが多い．レジストレーションのコストは細粒度の転送時に問題となる [27] ため，予めレジストレーションした領域にコピーするといった高速化手法が広く用いられているが，そのようなコピーは上位層が予めレジストレーションしておけば不要であることもある．
- 実際の通信の実行順序と，コールバック関数の呼び出し順序について，通信要求時と異なっていてもよい．順序保証の導入は，通信のリオーダリングを行うインターコネクトにおける性能向上の機会を失わせ，ランタイムシステムのマルチスレッド化においても内部の実装上の制約となりかねないためである．

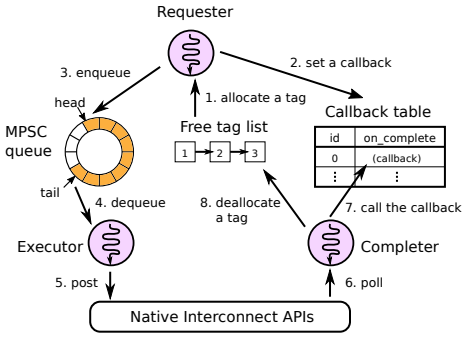


図 1　提案システムの設計

このように，既存処理系と比べてより低水準な API を提供し，各種インターコネクトを使う上での問題をシステムのプログラマにあえて明示することを意図している．一方で，インターコネクト間の差異が吸収されているため，下位の実装を入れ替えることで上位システムをほぼそのまま別環境で高速に実行できると期待される．また，通信要求の関数は全てスレッドセーフであり，上位層ではその層での排他制御のみに注力すればよい．

提案する API では，ユーザプログラム中で明示的に呼び出すポーリング関数は存在せず，通信要求時に内部でポーリングが行われるということもない．このことは，ポーリング処理が通信専用スレッドなどによって自動的に実行されることを意味する．ユーザが計算処理中に適切な間隔でポーリングを挟むことは困難であり，各種インターコネクトによってもその間隔は異なる．また，ポーリング中にも計算をオーバーラップできるため，並列性も低下しているといえる．一方で，ポーリング処理を別スレッドで実行する場合，完了通知時にコア間通信によるソフトウェアオーバーヘッドが発生する．ポーリングを同一スレッドで実行すればこれを削減できる可能性があるが，ポーリング時に使用する通信資源がコア間で共有されている場合は，ポーリングの結果が必ずしも同一スレッドの通信要求であるとはいえない点に注意する必要がある．

3.2　提案システムの設計

図 1 に，提案システムの設計を示す．次節以降で以下の 3 つのモジュールに分けて解説を行う．

通信要求の生成 (Requester)　通信要求をコマンドキューに投入し，通信を移譲する．通信要求が発生し

```
constexpr size_t N = /* # of queue elements */;
using tag_t = /*integer*/;
struct command { tag_t tag; /*...*/ };
struct element { command cmd; atomic<bool> vis; };
element elems[N];
callback cbs[/* # of tags */];
atomic<uint64_t> head, tail;
```

リスト 2　MPSC 循環バッファのデータ構造

たスレッド上で直接処理される．

通信実行 (Executor)　コマンドキューを監視し，コマンドがあればそれを取り出して実行し，インターコネクトに通信開始を指示する．

通信完了通知 (Completer)　インターコネクトのポーリング関数を呼び出し，完了通知を取得した場合は対応するコールバック関数を実行する．

インターコネクトの事情によって異なるが，これらのモジュールはそれぞれ複数存在する場合があり，それらが別のスレッドとして並列動作可能な場合もある．

```
tag_t alloc_tag();
void copy_params(const read_params&, command*);

bool try_read_async(const read_params& params) {
  uint64_t t;
  do {
    t = tail.load(memory_order_relaxed);
    if (t - head.load(memory_order_acquire) >= N)
      return false;
  }
  while (!tail.compare_exchange_weak(
    t, t+1, memory_order_acquire));

  element& e = elems[t % N];
  tag_t tag = alloc_tag();
  cbs[tag] = params.on_complete;
  e.cmd.tag = tag;
  copy_params(params, &e.cmd);
  e.vis.store(true, memory_order_release);
  return true;
}
```

リスト 3　Requester の実装例

3.3　通信要求の生成 (Requester)

通信処理は通信専用スレッドに移譲されるが，この際用いられるのはロックフリーな循環バッファである．一般的に，Requester 側のスレッドは複数あり，Executor のスレッドは（キュー 1 つに対して）1 つなので，Multiple-Producer Single-Consumer (MPSC) であると仮定することができる．

リスト 2 は，ロックフリーな MPSC 循環バッファのデータ構造の一例である．要素の配列に加えて，整数カウンタ 2 つ（`head` と `tail`）と，Producer が追加した要素を Consumer から可視にするフラグ（`vis`）が必要となる．各コマンドを表す構造体には，コマンドの種類に応じたデータの他に，完了通知を管理するためのタグが付与される．

通信要求の移譲処理は**リスト 3** のようなコードになる．Producer (= Requester) は Compare And Swap (CAS) によって `tail` を加算して伸長させることを試みるが，その際キューが満杯になっていないかを調べて，満杯であれば `false` を返して復帰する．CAS が成功した場合は，新規タグを確保してコールバックを書き込み，コマンドを設定して，最後にフラグ `vis` をセットして Consumer (Executor) に通知する．

ABA 問題を避けるため，`tail` は `N` を超えて加算し続けておき，上位ビット列の違いで CAS を失敗させる．有効なコマンドが存在するのは `[head, tail)` の区間（実際には N による剰余を取る）であり，`head == tail` の場合が空で，`head+(N-1) == tail` の場合が満杯と定義している．そのため，厳密には最大でも `N-1` 要素しか挿入できないが，この点は性能上重要でないので無視している．

通信要求関数が失敗できると規定されていることにより，通信システムの実装はキューが満杯になった際に `false` を返すことで速やかに呼び出し元関数に復帰できる．失敗通知を受け取ったユーザ側の関数には，次の 2 つの選択肢が与えられる．

- すぐさま再試行する．要求している通信処理のレイテンシを削減するのに有効である．この場合，要求が成功するまでスピンすることになるため，それを避けるために Pthreads の `sched_yield()` に相当する CPU の解放処理（以後，`yield()` と表記）を挟むことが可能だが，その場合はコンテキストスイッチを行うオーバーヘッドが加算されるために平均レイテンシは増加する．
- 別の計算/通信処理を実行した後で再試行する．要求している通信処理以外にも並行して実行可能な処理がある際に有効である．

どの手法が適しているかは低水準通信レイヤーのみでは判断不可能であるため，通信に関する最低限の処理のみを実行し，スケジューリングの方針は上位層に委ねるという方法を採用している．

タグ生成（`alloc_tag` 関数）には，使用中でないタグを区別できるフリーリストが必要となる．現時点の実装ではスレッドアンセーフな循環バッファをスピンロックで排他制御して使用しており，性能面で課題が残っている．解決策としては高速なアロケーション機構を実装する必要があり，各スレッドが独立した MPSC のキューを持つなどの手法が考えられる．4.1 節で述べる Tofu 上の実装では，Executor がタグ生成を行うことでロックの必要性を排除している．

```
void execute(const command&);
// ...
uint64_t h = head;
while (true) {
  while (h == tail.load(memory_order_acquire))
  { /*yield();*/ }
  element& e = elems[h % N];
  while (e.vis.load(memory_order_acquire)) { }

  execute(e.cmd);
  e.vis.store(false, memory_order_relaxed);
  head.store(++h, memory_order_release);
}
```

リスト 4　Executor の実装例

3.4　通信実行 (Executor)

通信を処理するスレッドは，リスト 4 のようなイベントループでコマンドキューを監視する．Executor (= Consumer) は Producer による `tail` の変更を監視し，コマンドが存在していればそれを実行し，`head` を前進させる．Single-Consumer なので `head` の衝突を考慮する必要はない．

キューに大量の通信要求が存在する場合，この関数は通信要求を実行し続ける．問題となるのは通信要求が無い場合であり，キューが空の際にこのスレッドはスピンすることになる．レイテンシ削減にはスピンが有効だが，一方でスピンしているスレッドが多すぎると CPU の利用効率が低下してしまうという問題点がある．これは，インターコネクトが通信要求に必要な資源を複数持ち，Executor を並列実行できるような場合に特に問題となる．

解決策の一つとしては `yield` を挟むことができるが，CPU の実行権限が回ってくるまで通信要求が遅延されるためにレイテンシの増大を招く．ロックフリーなキューではなくミューテックスと条件変数を用いたキューを使用することは可能だが，それらの同期プリミティブによるレイテンシ増加とスケーラビリティ低下が避けられない．

そこで，スピンを減らすための試みとして，ロックフリーキューと条件変数を組み合わせたキューについても検討した．通信要求がない場合は Consumer が条件変数によって待機し，その後最初の通信要求を行う Producer が Consumer に通知を行う．Consumer がコマンドを処理している間に Producer がさらなる通信要求を生成すると，ロックフリーキューに通信要求が追加されていく．このような仕組みにより，高負荷時の Requester 間の衝突を防ぎつつも，スピンするスレッドを減らすことができる．一方で，条件変数を使うことでレイテンシは増加することになる．

Consumer が条件変数で待機する場合は，コマンドが追加された際に Producer がそれに通知する必要があるが，高負荷時にはアトミック命令のみ動作させこの通知を削減することが性能上重要である．本稿の評価結果からは割愛するが，試験的な実装では `tail` の最下位 1 ビットを「待機中」であることを示すフラグとして利用することで，Producer が CAS を実行する際に Consumer に通知すべきか同時に判断できるようにしている．

```
bool poll(tag_t* tag_result);
void dealloc_tag(tag_t tag);
// ...
while (true) {
  tag_t tag;
  while (!poll(&tag)) { /*yield();*/ }
  callback& cb = cbs[tag];
  (*cb.f)(cb.d);
  dealloc_tag(tag);
}
```

リスト 5　Completer の実装例

3.5　通信完了通知 (Completer)

通信完了処理を行う Completer は，リスト 5 のようにインターコネクトに対してポーリングを行い，通知を取得できた場合はコールバック関数を実行し，そしてタグを解放して再利用できるようにする．

Completer に関しても，現在の実装では Executor と同様にスピンの問題を抱えている．InfiniBand の場合は，完了通知をファイルディスクリプタに関連付けて `select()` などで待ち合わせることが可能である．但し，レイテンシは増大する点に注意する必要がある．

現在の実装では Completer は 1 つだけであるが，ポーリングが並列に実行可能ならば複数に増やすことも可能である．但し，Completer が並列動作する場合は，タグの解放を正しく排他制御する必要がある．

API としてコールバック関数は Requester とは別スレッドで実行される可能性があるため，スレッドセーフでなければならない．また，API 上はポーリングスレッドは 1 つであると仮定していないため，ポーリング処理も並列に実行されている可能性がある．例えば，通信処理をまとめて待ち合わせるにはカウンタを加算して一定値になるまで監視すればよいが，コールバック関数内ではフェッチアンドアッド命令などで排他制御する必要がある．

4.　各インターコネクトごとの実装手法

3 章の設計と汎用的な実装を踏まえて，各種インターコネクトの仕様に起因する問題と，それらの解決策について述べる．

4.1　Tofu 用実装

Tofu の RDMA API には，次のような制約がある．

- 全ての API 関数がスレッドセーフではない．並列に

でなければ複数スレッドから呼び出すことはできる．
- 複数の NIC が存在するが，異なる NIC を使用する場合でも API 関数を並列に呼び出すことはできない．
- 通信要求関数とポーリング関数も並列に実行できない．
- MPI と内部のデータ構造が共有されており，全ての RDMA 関数に加えて MPI も並列に実行できない．

以上のような事情から，必然的に Tofu 上の実装では前述した設計において逐次化される範囲が広くなる．具体的には，次のような実装上の配慮を行っている．
- 通信要求関数を実行する Executor と，ポーリングを行う Completer は，単一の通信専用スレッド上でのみ動作させる．これによって，通信要求やポーリングが並列に実行されることを防ぐ．
- MPI の呼び出しが必要になる場合は，RDMA と同様にコマンドキューに投入してから通信専用スレッドで MPI を呼び出す．

現在の Tofu 用実装では，タグの確保を Executor 内で行っており，図 1 で示した設計とは若干異なっている．その場合は Requester 間の排他制御を減らせるため，マルチスレッドのベンチマーク性能としては安定するが，Executor スレッドの負荷が増大しているといえる．タグの確保を Requester 内で高速に実行する方法については現在検討中である．

4.2 InfiniBand 用実装

InfiniBand Verbs （以下，IBV と略す）は，InfiniBand を用いる際の標準的な API である．Tofu と異なり，IBV では API が全てスレッドセーフである．本稿に関係する範囲として，IBV は以下のような構成要素を持つ．

Queue Pair (QP) TCP/IP におけるソケットに相当し，通信要求が投入されるオブジェクトである．全体全で通信するには宛先のプロセスごとに QP が必要となる．実際の IBV の実装では，QP に対して通信要求を行う `ibv_post_send()` は QP ごとに独立したスピンロックで守られているため，QP が異なれば並列実行可能である．

Completion Queue (CQ) 通信完了通知が溜め込まれるオブジェクトであり，`ibv_poll_cq()` を呼び出すことで調べることができる．QP 同様に，CQ ごとに独立したスピンロックを持つ．

IBV はスレッドセーフな実装を提供しているが，内部ではスピンロックに頼っているのが実情である．スピンロックは通信レイテンシ削減には効果的であるものの，複数スレッドによって衝突が起きると 1 スレッドを除いて全スレッドがスピンするので，CPU の利用効率が低下する．通信レイテンシ隠蔽の観点からいって，IBV でも Offloading を用いる意義はあるといえる．

さらに，IBV で Offloading を用いることで，メッセージレート向上も期待できる．`ibv_post_send` には複数の通信要求（Work Request (WR) と呼ばれる）をリストにしてまとめて渡すことが可能である．`ibv_post_send` の実装では，所定のメモリ領域にアドレス情報などを書き込んだ後，doorbell と呼ばれる機構を用いてハードウェアに通信を要求する．この一連の動作を実行するのに少なくとも数百サイクルは消費するが，複数まとめて書き込むことでこのオーバーヘッドもまとめることができる．Offloading を行うと通信要求がコマンドキューに集約されるので，`ibv_post_send` を呼び出す際に複数のコマンドを一括して送り出すことが可能となる．

現在の実装では，Executor スレッドは 1 つとしているため，複数の QP が存在していて並列に実行できる場合も逐次化され，並列性が Offloading なしの場合と比べて低下する問題がある．この点については，Executor の設計で述べたように条件変数を併用するといった対策を含め，いくつかの実装手法を比較検討中である．

```
uint64_t get_clock(); // returns CPU clock
atomic<bool> flag;
void callback_func() {
  flag.store(memory_order_release);
}
// ...
while (/*...*/) {
  flag = false;
  uint64_t t0 = get_clock();
  while (!try_read_async(/*...*/))
  { /*yield();*/ }
  uint64_t t1 = get_clock();
  while (!flag.load(memory_order_acquire))
  { /*yield();*/ }
  uint64_t t2 = get_clock();
}
```

リスト 6 レイテンシとオーバーヘッドを測定するマイクロベンチマーク

5. 評価手法

評価にマイクロベンチマークを使用し，1 ノード 1 プロセスで 2 ノードで実験する．片方のプロセスがもう片方に RDMA READ を繰り返すことで，通信 1 回が完了するのにかかるレイテンシ，通信要求にかかる時間であるオーバーヘッド，単位時間あたりの通信回数であるメッセージレートを算出する．また，スレッド数を増やした際のそれぞれの変化も調べる．

レイテンシとオーバーヘッドは，**リスト 6** のプログラムのようなベンチマークで計測する．各スレッドは，通信要求を 1 つ送出して待機するということを一定時間経過するまで繰り返す．レイテンシは $t_2 - t_0$，オーバーヘッドは $t_1 - t_0$ で表される．

メッセージレートは**リスト 7** のようなプログラムで測定

```
atomic<uint64_t> count;
void callback_func() {
  count.fetch_add(1, memory_order_release);
}
// ...
while (/*...*/) {
  while (!try_read_async(/*...*/))
  { /*yield();*/ }
}
```

リスト 7　メッセージレートを測定するマイクロベンチマーク

表 1　Tofu 用実装の評価環境 (FX10) [28]

CPU	SPARC64™ IXfx, 1.848 GHz, 16 コア
メモリ	16GB
インターコネクト	Tofu
OS	XTCOS (GNU/Linux 2.6.25.8)
コンパイラ	GCC 4.6.3 (オプション “-O3”)
MPI	Fujitsu MPI Library

表 2　InfiniBand 用実装の評価環境

CPU	Intel® Xeon® E5-2680 v2 2.80GHz, 2 ソケット × 10 コア
メモリ	16GB
インターコネクト	Mellanox® Connect-IB® dual port InfiniBand FDR 2-port
ドライバ	Mellanox® OFED 2.4-1.0.4
OS	Red Hat® Enterprise Linux® Server release 6.5 (Santiago)
コンパイラ	GCC 4.4.7 (オプション “-O3”)

する．各スレッドは一定時間経過するまで通信要求を常に送信し続ける．終了時に完了している通信要求数を経過時間で割ったものがメッセージレートである．

Tofu 用実装の評価環境を**表 1**，InfiniBand 用実装の評価環境を**表 2** に示している．

現在の実装には一部バグが含まれており，高負荷時にしばしば異常終了することが確認されているが，それらの結果は含まれていない．

6. 評価結果と考察

評価結果におけるレイテンシの結果は全て往復分で算出している．グラフ上の各点に対して，5 秒間 RDMA READ を繰り返すプログラムを 32 回測定し，95%信頼区間を算出している．

6.1 Tofu 上での評価

図 2 に，Tofu 用実装のレイテンシのベンチマーク結果を示す，最小レイテンシとなるのは 1 スレッド時で，2.813 μsec（5200 サイクル）であった．

提案処理系を使用せずに Tofu の RDMA API を用いたマイクロベンチマークを実行したところ，RDMA READ の最小レイテンシは 2.365 μsec（4372 サイクル）であった．これらの差である 0.448 μsec（元のレイテンシに対して +19%）は，スレッドセーフティを保証し，かつオーバーラップの機会を増やすためのオーバーヘッドであるといえる．

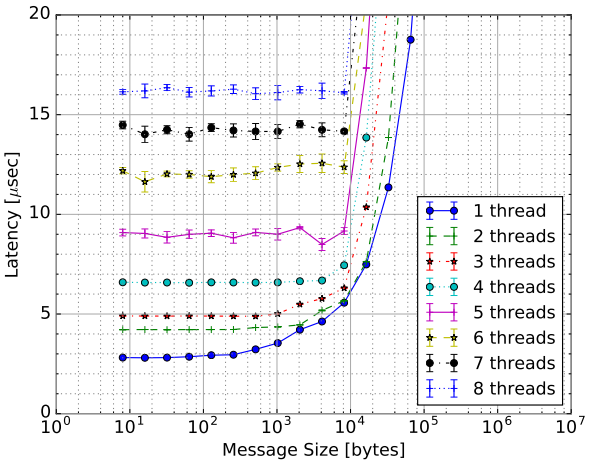


図 2　Tofu 用実装のレイテンシの測定結果

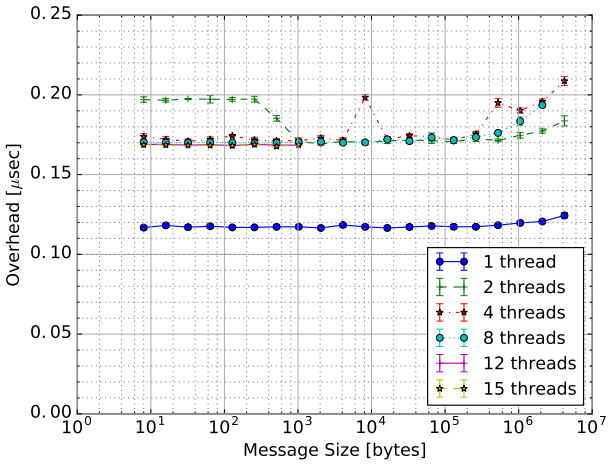


図 3　Tofu 用実装のオーバーヘッドの測定結果

参考までに，Tofu 2 上における ACP [25] のベンチマークでは，リモートアトミックのネイティブ API のみでレイテンシが 2.26 μsec であるのに対し，ACP を用いると 4.26 μsec まで増大すると報告しており，ACP のマルチスレッド対応には 2.00 μsec ものオーバーヘッドがあることが分かる．但し，ACP はレイヤーとして PGAS に属し，アドレス変換を行う機構が含まれているため，平等な比較ではないことに注意する必要がある．現段階では提案処理系を PGAS として評価できていないが，開発中の処理系においてアドレス変換のオーバーヘッドは約 0.7μsec （1300 サイクル）であったため，その差分を加味しても依然として ACP に対する優位性があることがわかった．

図 3 に，通信要求のオーバーヘッドの測定結果を示す．最小となるのは 1 スレッド時で，0.117μsec（216 サイクル）

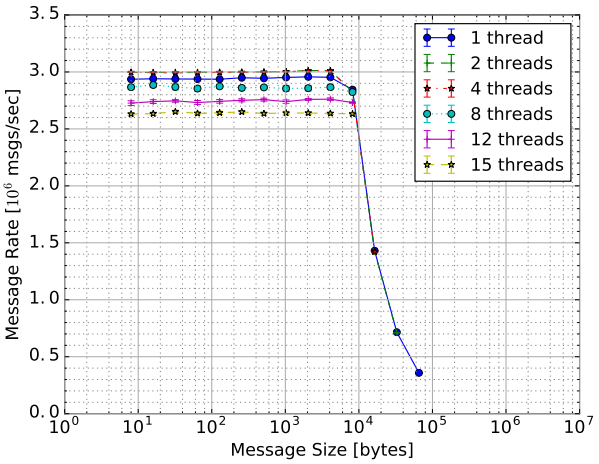


図 4　Tofu 用実装のメッセージレートの測定結果

である．スレッド数を増加させるとキューの操作が衝突するためオーバーヘッドは増加するが，最大でも 2 スレッド時に 0.196μsec（364 サイクル）程度である．通信レイテンシのうちオーバーヘッドが占める割合は 1 スレッド時に 4.19% である．残りの 96.81%の時間は，通信完了を待ち合わせる時間を除けばオーバーラップ可能であるといえる．

図 4 に，Tofu 用実装のメッセージレートのベンチマーク結果を示す．メッセージレートが最大となるのは 4 スレッドの時で，3.00×10^6 messages/sec となった．15 スレッドで実行しても依然として 2.64×10^6 messages/sec を保っており，スレッド間の衝突によるメッセージレートの低下を 12%に抑えることができた．

6.2　InfiniBand 上での評価

図 5 に，InfiniBand 用実装のレイテンシのベンチマーク結果を示す．Offloading を使用した際の最小のレイテンシは，1 スレッド時に 3.67 μsec（10288 サイクル）であった．Offloading を使用せず，Requester が直接 `ibv_post_send()` を呼ぶようにするとレイテンシは小さくなり，3.31 μsec（9276 サイクル）となるが，その差は 0.36 μsec と僅かである．

InfiniBand には perftest というベンチマークがあり，そのうちの ib_read_lat によって RDMA READ のレイテンシを計測したところ，4.03 μsec（11288 サイクル）であった．標準のベンチマークよりもレイテンシが 0.40 μsec 短縮された理由は不明だが，実行したノードの状態や，ポーリングを別スレッドで行っていることなどが影響している可能性がある．

図 6 に，InfiniBand 用実装のオーバーヘッドを示す．Offloading ありで最小オーバーヘッドとなるのは，1 スレッド時に 0.689 μsec（1930 サイクル）である．スレッド数が増加するとオーバーヘッドが増大しており，その原因は Requester が Completer からリクエストの ID（`wr_id`）を取得する際にスピンロックを使用しているためであるとみられ，今後修正する予定である．

Offloading なしの場合のオーバーヘッドの最小値は，1 スレッド時に 0.486 μsec（1361 サイクル）であった．その大半は `ibv_post_send()` を呼び出している時間であると推察される．スレッド数を増やすと急激にオーバーヘッドが増加しており，内部のスピンロックが原因とみられる．

図 7 に，InfiniBand 用実装のメッセージレートを示す．Offloading ありの際のメッセージレートの最大値は 2 スレッドのときで，11.8×10^6 messages/sec であった．スレッド数増加に伴いメッセージレートが低下するのもオーバーヘッド増加と同様の原因によるとみられる．Offloading がない場合でも，最大値は 1 スレッド時で 2048 bytes の際に 10.2×10^6 messages/sec になるが，8 bytes では 6.54×10^6 messages/sec まで逆に低下している．Offloading なし・小さいメッセージでのメッセージレート低下について明確な原因は不明だが，Offloading ありの場合は通信要求をまとめているために安定したメッセージレートが実現できていることがわかる．

7.　結論

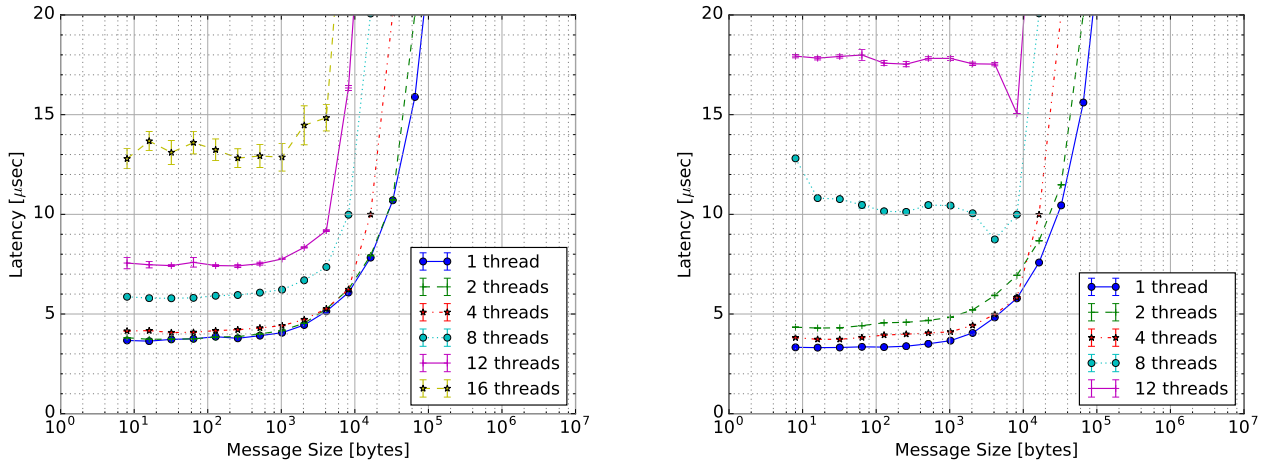
本稿では，PGAS 向けの低水準通信レイヤーについて，マルチスレッド実行を考慮しかつオーバーヘッドが少なくなるよう API を定義し，それに対して Tofu と InfiniBand を例に実装を行った．マルチスレッド環境でのマイクロベンチマークによる評価の結果，レイテンシ，オーバーヘッド，メッセージレートの 3 点について良好な結果が得られ，提案する実装手法がマルチスレッド対応，レイテンシ隠蔽，メッセージ集約の観点から有用であることを示した．

今後の方向性としては，今回実装した低水準通信レイヤーを基に PGAS 処理系を実装して，アプリケーションによるベンチマークを行う予定である．また，他のインターコネクトへの移植も検討し，API と実装の両面から汎用性を高めていく予定である．

謝辞　本研究の一部は科研費基盤研究 (A) 16H01715 の助成を受けて行われている．

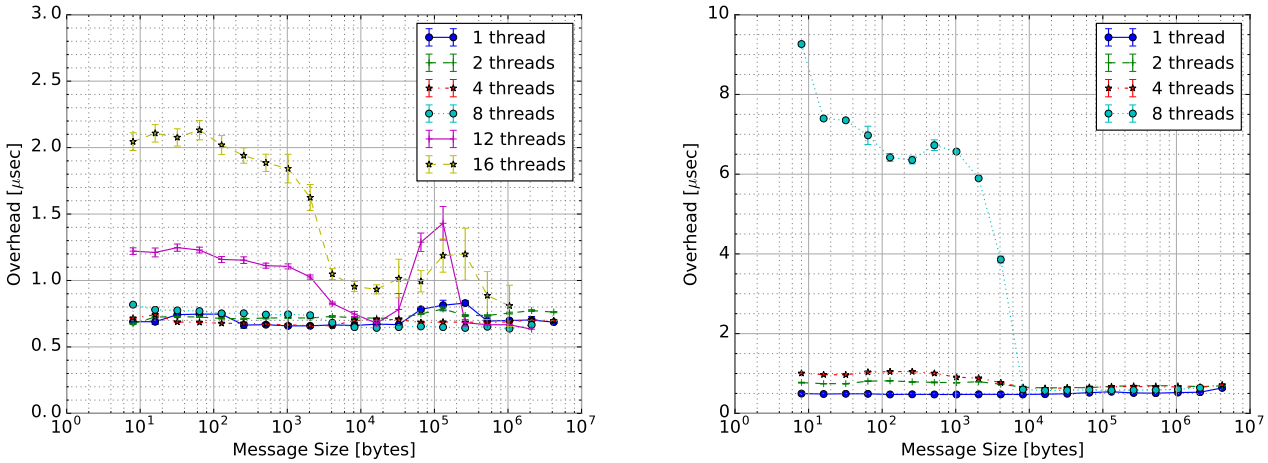
参考文献

[1] MPI Forum: MPI: A Message-Passing Interface Standard Version 3.1, (online), available from ⟨http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf⟩ (2015).
[2] 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).
[3] 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*



(a) Offloading あり　(b) Offloading なし

図 5　InfiniBand 用実装のレイテンシの測定結果



(a) Offloading あり　(b) Offloading なし

図 6　InfiniBand 用実装のオーバーヘッドの測定結果

Performance Computing Applications, Vol. 20, No. 2, pp. 203–231 (online), DOI: 10.1177/1094342006064503 (2006).

[4] 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).

[5] 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).

[6] Ajima, Y., Takagi, Y., Inoue, T., Hiramoto, S. and Shimizu, T.: The Tofu Interconnect, *HOTI '11: Proceedings of IEEE 19th Annual Symposium on High Performance Interconnects*, pp. 87–94 (online), DOI: 10.1109/HOTI.2011.21 (2011).

[7] InfiniBand Trade Association: Infiniband Architecture Specification (2007).

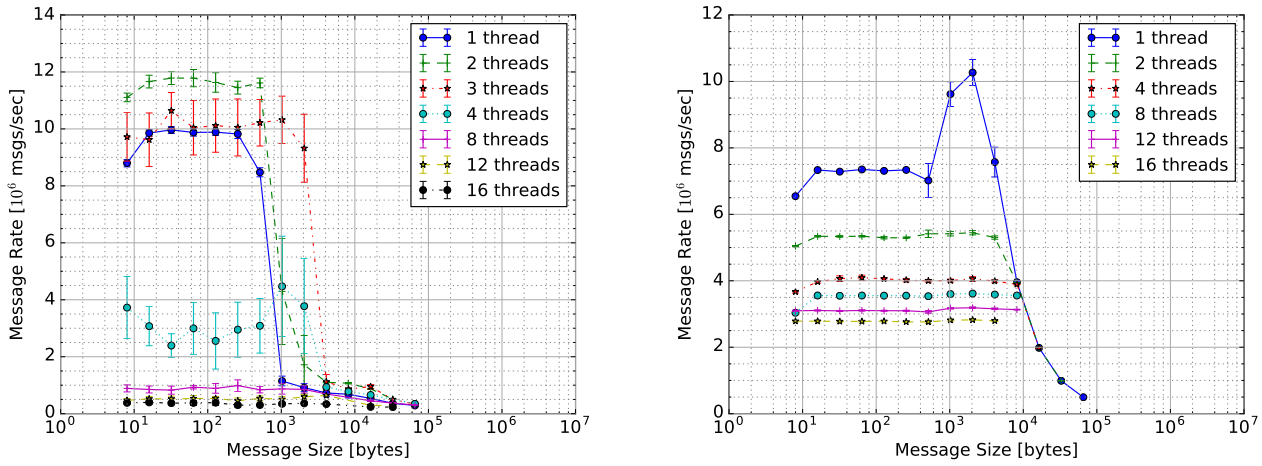
[8] Bonachea, D.: GASNet Specification Version 1.8, Technical report, EECS Department, University of California, Berkeley (2006).

[9] Bonachea, D. and Jeong, J.: GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages, *CS258 Parallel Computer Architecture Project*, pp. 1–27 (online), available from ⟨http://www.cs.berkeley.edu/ jaein/cs258/paper.pdf⟩ (2002).

[10] von Eicken, T., Culler, D. E., Goldstein, S. C. and Schauser, K. E.: Active Messages: A Mechanism for Integrated Communication and Computation, *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pp. 256–266 (online), DOI: 10.1109/ISCA.1992.753322 (1992).

[11] Nieplocha, J., Tipparaju, V., Krishnan, M. and Panda, D. K.: High Performance Remote Memory Access Communication: The Armci Approach, *International Journal of High Performance Computing Applications*, Vol. 20, No. 6, pp. 233–253 (online), DOI: 10.1177/1094342006064504 (2006).

[12] Daily, J., Vishnu, A., Palmer, B., van Dam, H. and Kerbyson, D.: On the Suitability of MPI as a PGAS Run-



(a) Offloading あり　　(b) Offloading なし

図 7 InfiniBand 用実装のメッセージレートの測定結果

time, *HiPC '14: The 21st annual IEEE International Conference on High Performance Computing*, (online), DOI: 10.1109/HiPC.2014.7116712 (2014).

[13] OpenFabrics Alliance: OpenFabrics, https://www.openfabrics.org/.

[14] Grun, P., Hefty, S., Sur, S., Goodell, D., Russell, R. D., Pritchard, H. and Squyres, J. M.: A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency, *HOTI '15: Proceedings of IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 34–39 (online), DOI: 10.1109/HOTI.2015.19 (2015).

[15] Shamis, P., Venkata, M. G., Lopez, M. G., Baker, M. B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R. L., Liss, L., Shahar, Y., Potluri, S., Rossetti, D., Becker, D., Poole, D., Lamb, C., Kumar, S., Stunkel, C., Bosilca, G. and Bouteiller, A.: UCX: An Open Source Framework for HPC Network APIs and Beyond, *HOTI '15: Proceedings of IEEE 23rd Annual Symposium on High-Performance Interconnects*, IEEE, pp. 40–43 (online), DOI: 10.1109/HOTI.2015.13 (2015).

[16] Zhou, H., Mhedheb, Y., Idrees, K. and Glass, C. W.: DART-MPI : An MPI-based Implementation of a PGAS Runtime System, *PGAS '14: Proceedings of the 8th International Conference on PGAS Programming Models*, (online), DOI: 10.1145/2676870.2676875 (2014).

[17] Si, M., Pena, A. J., Hammond, J., Balaji, P., Takagi, M. and Ishikawa, Y.: Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures, *IPDPS '15: IEEE International Parallel and Distributed Processing Symposium*, pp. 665–676 (online), DOI: 10.1109/IPDPS.2015.35 (2015).

[18] Balaji, P., Buntinas, D., Goodell, D., Gropp, W. and Thakur, R.: Toward Efficient Support for Multithreaded MPI Communication, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer Berlin Heidelberg, pp. 120–129 (2008).

[19] Balaji, P., Buntinas, D., Goodell, D., Gropp, W. and Thakur, R.: Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming, *International Journal of High Performance Computing Applications*, Vol. 24, No. 1, pp. 49–57 (online), DOI: 10.1177/1094342009360206 (2010).

[20] MPICH: MPICH, (online), available from ⟨http://www.mpich.org/⟩.

[21] Vaidyanathan, K., Kalamkar, D. D., Pamnany, K., Hammond, J. R., Balaji, P., Das, D., Park, J. and Joó, B.: Improving Concurrency and Asynchrony in Multithreaded MPI Applications using Software Offloading, *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 30:1–30:12 (online), DOI: 10.1145/2807591.2807602 (2015).

[22] Amer, A., Lu, H., Wei, Y., Balaji, P. and Matsuoka, S.: MPI+Threads: Runtime Contention and Remedies, *PPoPP '15: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 239–248 (online), DOI: 10.1145/2688500.2688522 (2015).

[23] Lu, H., Seo, S. and Balaji, P.: MPI+ULT: Overlapping Communication and Computation with User-Level Threads, *HPCC '15: IEEE 17th International Conference on High Performance Computing and Communications*, pp. 444–454 (online), DOI: 10.1109/HPCC-CSS-ICESS.2015.82 (2015).

[24] Saga, K., Ajima, Y., Nose, T., Miura, K. and Sumimoto, S.: ACP 基本層の実装と初期評価，研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2014-HPC-1, No. 10, pp. 1–6 (2014).

[25] Nose, T., Ajima, Y., Saga, K., Shida, N. and Sumimoto, S.: ACP ライブラリの性能最適化に関する検討，研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2015-HPC-1, No. 39, pp. 1–6 (2015).

[26] Endo, W. and Taura, K.: 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，研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2015-HPC-1, No. 5, pp. 1–8 (2015).

[27] Frey, P. W. and Alonso, G.: Minimizing the Hidden Cost of RDMA, *ICDCS '09: Proceedings of 29th IEEE International Conference on Distributed Computing Systems*, pp. 553–560 (online), DOI: 10.1109/ICDCS.2009.32 (2009).

[28] 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).