Vol.2016-HPC-155 No.15
2016/8/8

GPUクラスタにおける GPUセルフMPIシステムGMPIの予備性能評価

桑原 悠太[1,a)]　塙 敏博[2]　朴 泰祐[1,3]

概要：今日，CUDA (Compute Unified Device Architecture) は NVIDIA の GPU のプログラミング環境として一般的に用いられている．その高性能かつ低電力な特徴から，GPU クラスタに搭載された GPU は様々なアプリケーションの実行に用いられる．CUDA はシングルノード向けに開発されたため，ノード間での通信には MPI (Message Passing Interface) などを用いる必要がある．従来手法では，通信の度に制御を CUDA カーネルから CPU に戻す必要があり，カーネル関数の起動や同期に伴うオーバーヘッドが生じる他，プログラマビリティや生産性の低下も問題となっている．これらの問題を解決するために，我々は GPU カーネル内から MPI 通信の起動を可能とする並列通信システム “GMPI” を開発している．本稿では，この GMPI システムにおける並列実行モデルを定義し，Ping-Pong 通信および姫野ベンチマークの性能評価を実 GPU クラスタ上で行う．現状，Ping-Pong 通信は従来手法とほぼ同等の性能である．しかしながら，性能最適化やチューニングが十分でなく，姫野ベンチマークでは従来手法の約 66%の性能にとどまっており，この妨げとなる要因の考察を行う．

1. はじめに

近年，GPU (Graphics Processing Unit)，MIC (Many Integrated Cores)，FPGA (Field-Programmable Gate Array) などのアクセラレータは高性能かつ低電力な HPC システムにおいて重要な要素となっている．特に，汎用アクセラレータとして GPU (Graphics Processing Unit) が持つ高い浮動小数点演算能力とメモリバンド幅を利用した GPGPU (General Purpose computing on GPU) が注目されており，GPU を搭載した複数のノードで構成されたクラスタが盛んに開発されている．Top500 List[1] においても，GPU を活用したシステムが多数登場する．2016 年 6 月に発表された Top500 List では，3 位の Titan[2]，31 位の TSUBAME2.5[3] や 32 位の Tianhe-1A[4] が存在し，筑波大学でも HA-PACS[5]（初出場順位 41 位）が稼働中である．

現在，HPC 向けの GPU クラスタでは，NVIDIA 社製の GPU が標準的に用いられ，そのアプリケーションは CUDA (Compute Unified Device Architecture)[6] を用いて記述される．ほとんどの HPC アプリケーションは MPI[7] を用いて並列化されており，CUDA と MPI の組み合わせを用いて GPU プログラミングを行う．CUDA では，計算部分をカーネル関数として記述し，ホスト CPU からこれを呼び出して起動する．特に断らない限り，GPU カーネル関数を「カーネル関数」と述べる．現状，カーネル関数の内部ではノード間通信を開始することはできないため，通信が必要になる毎にカーネル関数を終了し，制御を CPU に戻す必要がある．この制限により，通信が発生する箇所でカーネル関数を切り分けてコーディングする必要があり，カーネル関数の起動に伴うオーバーヘッドが生じる．また，MPI 通信の前に，GPU で計算したデータの同期を行う上でもオーバーヘッドが生じ，このオーバーヘッドも通信コストに含まれる．この従来手法では，通信の粒度が細かくなる程，カーネル関数の起動回数も増え，オーバーヘッドも増加する．また何よりも，CPU 向けの MPI プログラムを CUDA+MPI 環境向けに書き換える際，プログラムを多数のカーネル関数と MPI 通信の組み合わせに書き換える必要がある．このプログラミングのコストが非常に大きく，ソースコードも煩雑になりやすい．これらの問題を解決するために，我々は GPU カーネル内からホスト CPU に制御を戻すことなく MPI 通信の起動を可能とする並列通信システム “GMPI” を開発している．このコンセ

[1] 筑波大学大学院 システム情報工学研究科
Graduate School of System and Information Engineering, University of Tsukuba
[2] 東京大学 情報基盤センター
Information Technology Center, The University of Tokyo
[3] 筑波大学 計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[a)] kuwahara@hpcs.cs.tsukuba.ac.jp

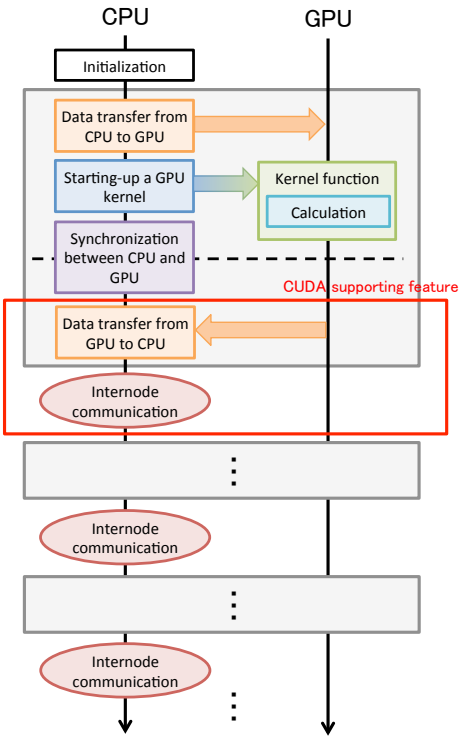

図 1 GPU クラスタにおけるノード間通信の流れ

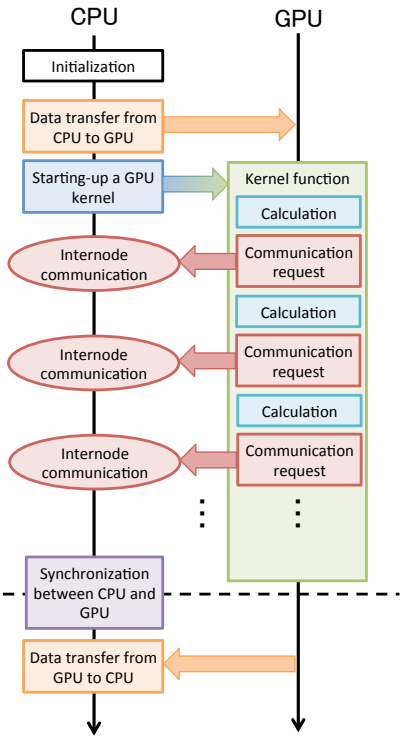

図 2 GMPI システムにおけるノード間通信の流れ

プトでは，カーネル関数内で CPU 向けの MPI プログラムのように，MPI 関数を呼び出すことができる．これにより，プログラマビリティや移植性の向上を目指す．本研究では，一般的に用いられている NVIDIA 社製 GPU を対象とし， CUDA 環境を前提に開発を行う．

2. GPU クラスタにおけるノード間通信

本節では， GPU クラスタにおける一般的なノード間通信とその問題点に関して述べる．以下，特に断らない限り，ホスト CPU を単に「ホスト」と呼ぶ．

通常の GPU クラスタでは異なるノードの GPU 同士は直接通信できないため，ホスト側で通信の起動や管理を行う．送信側では，ホストがデバイスメモリのデータをホストメモリにコピーをし，*MPI_Send()* などの関数を用いて通信を起動する．受信側では，ホストが *MPI_Recv()* などの関数を用いてデータを受信し，ホストメモリからデバイスメモリにデータをコピーする．MPI 環境ではそれぞれの MPI プロセスが CUDA カーネルとホストプログラムを実行し，すべての MPI 関数はホストメモリのみを扱うことができる．そのため，ホストメモリとデバイスメモリ間でのデータ転送が必要となる．CUDA では，*cudaMemcpy()* などの関数を用いてデータを転送するが，これを用いたメモリコピーにはレイテンシが大きいという問題がある．

最近の MPI 処理系には，MVAPICH2[8]，MVAPICH2-GDR[8]，Open MPI[9] などのように，MPI の送信バッファおよび受信バッファにデバイスメモリを直接指定できるものがあり，このような MPI を CUDA-Aware な MPI と呼ぶ．CUDA-Aware な MPI ライブラリを用いることで，プログラムの記述コストが低減する．

NVIDIA 社製の GPU では，GPUDirect RDMA (GDR) 機能 [10] を用いることで，PCIe デバイスが GPU のデバイスメモリにアクセスできるようになる．この機能を用いると，ホスト側で宣言されたデバイスメモリのアドレス空間は PCIe のアドレス空間にマッピングされる．マッピングされたアドレスに PCIe デバイスからアクセスすると，直接デバイスメモリにアクセスできる．この実用例として，Mellanox 社が提供する InfiniBand のネットワークインタフェースである InfiniBand HCA (Host Channel Adapter) が挙げられる [11]．HCA は対象のデータをホストメモリにコピーせずに，直接デバイスメモリにアクセスできる．前述の MVAPICH2-GDR や Open MPI は GDR 機能にも対応しており，これを利用することにより，全体の通信レイテンシも低減し，高い通信性能を達成できる．

CUDA+MPI の環境における GPU クラスタのノード間通信のプログラムの実行の流れを**図 1** に示す．一般的に，図中赤枠内の処理はユーザがプログラムに記述する必要があるが，CUDA-Aware な MPI を使用し，送信バッファや受信バッファにデバイスメモリを指定した場合は MPI ライブラリの内部で行われる．これにより，図中赤枠内の処理の記述を省略することができ，さらに GDR 機能に対応していればホストメモリへのデータ転送が不要となるためレイテンシが低減される．

表 1 GMPI がサポートする MPI 関数

name	corresponding MPI function
GMPI_Isend	MPI_Isend
GMPI_Irecv	MPI_Irecv
GMPI_Wait	MPI_Wait
GMPI_Waitall	MPI_Waitall
GMPI_Allreduce	MPI_Allreduce

しかしながら，CUDA-Aware な MPI で GDR 機能を使用した場合でも実際の通信はホストが行う．したがって，通信前にはカーネル関数が終了していなければならない他，通信後に新しくカーネル関数を起動する必要がある．さらに，通信の前後にはホストとデバイスの間でメモリ内のデータの同期が行われる必要があり，これによりオーバーヘッドが増加する．

3. GMPI

本節では，GPU カーネル内から MPI 通信の起動を可能とする並列通信システム “GMPI” について述べる．この詳細な実装については [12] を参照されたい．

3.1 概要

GMPI とは，実行中の CUDA カーネルがそのカーネル関数を終了することなく MPI 関数を直接実行できるシステムである．MPI 通信の実行中や実行後に，カーネル関数の実行を続けることができ，別の MPI 関数を呼び出すことなどもできる．現在，GMPI がサポートする MPI 関数は**表 1** に示すとおりである．

GMPI を利用した場合の GPU クラスタのプログラムの流れを**図 2** に示す．ただし，このプログラミングモデルはあくまでユーザ視点からのモデルであり，CUDA のフレームワークでは GPU 自身が主体となって実際の MPI 関数を起動することはできない．その代わりに， GMPI のシステムでは，ホストがポーリングによって GPU から通信リクエストを受け取る．ホストは，通信リクエストを受け取ると，GPU の代わりに実際の MPI 関数を実行し，その結果を GPU に戻す．このとき，通信リクエストと実行状態のみをホストメモリとデバイスメモリの間で転送し，実際のデータは GDR 機能を利用可能な通信デバイス (InfiniBand HCA) を用いて直接転送する．実際の MPI の関数は従来どおり実行されているため，GMPI 関数自身が CUDA+MPI の従来手法に比べて，通信に伴うオーバーヘッドを低減できるわけではない．ここで重要な論点は，カーネル関数がオーバーヘッドなしに実行し続けられることである．これにより，カーネル関数の起動と同期のステップを省略できる．通信リクエストの転送コストが，カーネル関数の起動と比べて十分小さくなれば，全体の通信コストが低減する．更に，ユーザは MPI 関数の呼び出し毎に，カーネル関数を複数に切り分けて記述する必要がなくなる．それゆえに，GMPI システムでは次の 2 つを目的としている．

(1) MPI による GPU 同士のノード間通信コストの低減
(2) MPI プログラムを CUDA+MPI のプログラムに移植するコストの低減

```
#pragma omp parallel for
for( i = 0; i < 100; i++ ) {
  ...Calculation...
  #pragma omp single
  {
    ...MPI Communication...
  }
  ...Calculation...
}
```

図 3 OpenMP+MPI プログラミングのコード例

3.2 同時実行スレッド数の制限

現在の GMPI の実装では，GMPI 関数が実行される前に *__threadfence()* を用いてカーネル関数を実行しているブロック数を数え上げ，スレッドブロック間の同期を行う．このとき，ブロック内では *__syncthreads()* を呼び出すことにより，全スレッドの実行が完了していることを保証する．我々が使用している Kepler 世代の GPU では，カーネル関数にプリエンプション機能が存在しない．同時実行可能なスレッド数は，ブロック辺りのスレッド数，スレッド辺りのレジスタ数，ブロック辺りの共有メモリの量に依存する．これらの制約から，SM (Streaming Multiprocessor) 毎に同時実行されるスレッド数が決定するため，最大の同時実行スレッド数には制限がある．そのため，現在の GMPI の実装には，同時実行可能なスレッド数に制限があり，ユーザが記述した GMPI を利用するカーネル関数は，この制限以下の合計スレッド数で実行される必要がある．

この制限の解決策として，GPU で実行しているカーネルから子供のカーネルを起動するダイナミックパラレリズムを利用するアイデアを検討している．通信を行う親カーネルと計算を行う子カーネルに分割し，全体の同期は親カーネルが行うことで，GPU 内での計算と通信が実現可能となる．このアイデアを実現する上で，ユーザがどのようにプログラムを記述できるようにするか検討する必要がある．更に，計算毎に親カーネル内で子カーネルを呼び出し，全体の通信コストが増加してしまうことが予想されるため，この問題の対策も重要となる．

また，Pascal 世代の GPU ではプリエンプション機能が実装されるため，この制限が解決される可能性がある [13]．この場合，Pascal 世代の GPU とともに異なる世代の GPU を利用した際にも，この制限が解決されることが，新たな課題として考えられる．

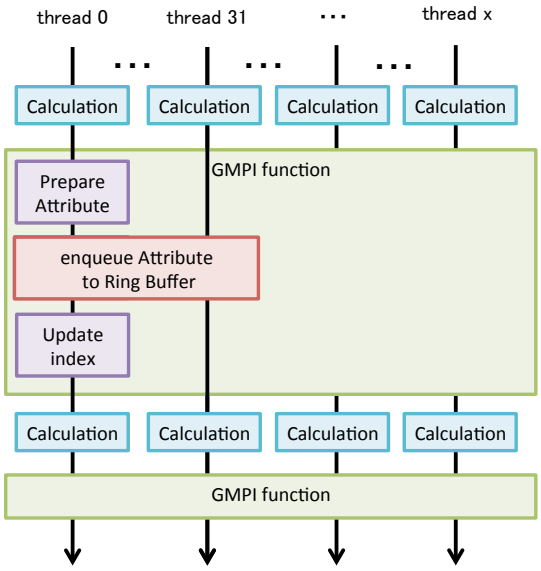

図 4 GMPI のセマンティックス

4. GMPI における並列実行モデル

本節では，GMPI を用いる際の GPU コンピューティングの並列実行モデルについて述べる．GMPI のセマンティックスを理解するために，まずは OpenMP を例にあげて説明する．

多くの OpenMP+MPI のコードでは，プログラマは 1 つのスレッドのみが MPI 通信を実行するシングルコンテキストのもと，MPI 関数を記述する．この記法では，**図 3** のように，OpenMP で並列実行される部分に single や master の指示文を記述することで，基本的に MPI 通信はシングルスレッドでのみ実行され，通信の後に，転送されたデータが複数スレッドによって参照される．OpenMP を利用する場合，計算速度を向上させるために，処理の重い部分に対してインクリメンタルに並列化していくのが一般的である．この場合，ユーザは個々の for ループに指示文を書いて並列化し，シーケンシャルなコンテキストのもと，MPI 関数が呼び出される．OpenMP プログラミングの最近のトレンドとして，作業単位が動的に変化する場合に有効で，スレッドセーフな MPI と組み合わせて用いられる task 構文の積極的利用が試みられている．最近の MPI はスレッドセーフにも対応しており，複数のスレッドがそれぞれ別々の MPI 関数を呼び出せるが，コードの可読性が損なわれ，通信の粒度が細かくなるため，通信性能も低下する場合が多い．しかしながら，こういったプログラミングスタイルは，GPU プログラミングのようなデータ並列の考え方とは異なっている．

一方，CUDA の SIMT (Single Instruction and Multiple Thread) 実行モデルの並列実行は，OpenMP のスレッド並列とは異なり，多数の並列なスレッドがカーネル関数起動時に実行され，それぞれの命令が 1 つの実行ストリームで並列に実行される．CUDA の SIMT 実行モデルでは，if 文などにより実行ストリームが 2 つに別れる場合があるが，この 2 つのストリームが同時に実行されることはない．実際，if 文側を実行している間は，もう一方のストリームは停止しているため，並列化効率が低下する．そのため，OpenMP と MPI のハイブリッドプログラミングと同様に，GMPI のためのアプリケーションフレームワークでは，並列に MPI 関数が呼び出された場合でも，シングルストリームで実行されることを想定する．

表 2 HA-PACS/TCA のノード構成

ハードウェア	
CPU	Intel Xeon-E5 2680v2 2.8 GHz × 2 sockets
Memory	DDR3 1866 MHz × 4 ch，128 GB
GPU	NVIDIA Tesla K20X × 4
Memory	GDDR5 2600 MHz，6 GB/GPU
InfiniBand	Mellanox ConnectX-3 QDR 4X Dual-port
ソフトウェア	
OS	CentOS 6.5
CUDA	CUDA 7.5
MPI	MVAPICH2-GDR 2.1a

このアイデアに基づいて，GMPI の並列実行モデルについて以下に述べる．ユーザは GMPI 関数の呼び出しをカーネル関数に記述でき，その関数は複数スレッドで平行して実行されない．これは複数スレッドが並列に MPI 関数を実行できず，シングルコンテキストのもとでのみ MPI 関数が実行されることを意味する．本稿では，この際に転送されるメッセージタイプ，メッセージサイズ，送信・受信バッファのポインタなどの通信リクエストに必要な情報を “Attribute” と呼ぶ．Attribute は，リングバッファを介してホストとデバイスの間を転送される．**図 4** に示すとおり，Attribute の用意を 1 スレッドで行った後，デバイスメモリ上のリングバッファの更新を CUDA の Warp の単位である 32 スレッドで行い，1 スレッドがリングバッファのインデックスを更新する．そのため，if 文などにより実行ストリームが別れている箇所では，GMPI 関数を呼び出すことはできない．このように，古典的な OpenMP と MPI のハイブリッドプログラミングとほぼ同様の考え方で GMPI を扱えるようにすることで，ユーザにとってプログラミングがしやすくなると考えた．

5. 性能評価

本節では， GMPI システムの性能評価を Ping-Pong 通信および姫野ベンチマーク [14] によって行う．

5.1 評価環境

筑波大学計算科学研究センターでは，TCA/PEACH2 の

```
__global__ void send_kernel(gmpi_buf_t *buf, int *src, int *dst, MPI_Request *request, MPI_Status
     *status) {
  for(int i = 0; i < N_ITER; i++) {
    GMPI_Isend(buf, src, DATA_SIZE, MPI_INT, 1, i, MPI_COMM_WORLD, request);
    GMPI_Irecv(buf, dst, DATA_SIZE, MPI_INT, 1, i, MPI_COMM_WORLD, request);
    GMPI_Wait(buf, request, status);
    GMPI_Synchronize(buf);
  }
}

__global__ void recv_kernel(gmpi_buf_t *buf, int *src, int *dst, MPI_Request *request, MPI_Status
     *status) {
  for(int i = 0; i < N_ITER; i++) {
    GMPI_Irecv(buf, dst, DATA_SIZE, MPI_INT, 0, i, MPI_COMM_WORLD, request);
    GMPI_Wait(buf, request, status);
    GMPI_Isend(buf, src, DATA_SIZE, MPI_INT, 0, i, MPI_COMM_WORLD, request);
    GMPI_Synchronize(buf);
  }
}
```

図 5 Ping-Pong 通信のコード例

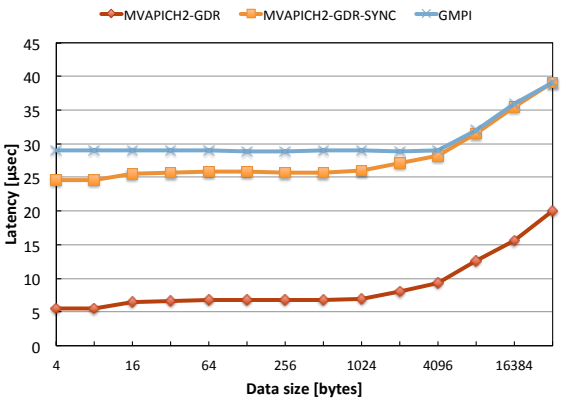

図 6 Ping-Pong 通信の測定結果

実験用システムとして，HA-PACS/TCA (Highly Accelerated Parallel Advanced system for Computational Sciences/TCA) が運用されている．このシステムのノード構成を**表 2** に示す．[12] では実験機上で測定を行っていたが，本稿における評価は，この実 GPU クラスタを用いて行う．本稿の測定では，QPI を跨がず，GDR 機能が適切に使える位置の GPU を 1 ノードあたり 1 台指定して使用する．また，InfiniBand HCA は QDR Dual-port だが，1 ポートのみ使用する．

5.2 Ping-Pong 通信の性能評価

まず，GMPI を用いた場合の Ping-Pong 通信のコードに関して説明する．実際に GMPI を用いたコード例をリスト**図 5** に示す．このコードでは，別々のノードがカーネル関数である *send_kernel()* または *recv_kernel()* を実行し，その間で Ping-Pong 通信を行う．はじめに，あるノードがデータを送信，もう一方のノードがデータを受信し，送信側と受信側を入れ替えてそれを繰り返す．GMPI 関数はデバイス上で実行される __device__ 関数として定義されている．上記のコードでは，GMPI 関数である *GMPI_Isend()*, *GMPI_Irecv()*, *GMPI_Wait()* を用いている．それぞれの __device__ 関数は，第一引数に GMPI のリングバッファのポインタ，それ以降の引数にはそれぞれの MPI の関数の引数を指定することで利用できる．コード例の src や dst は，*cudaMalloc()* などで確保されたデバイスメモリであり，カーネルの計算結果が格納されることを想定する．GMPI の実装には MVAPICH2-GDR を用いるため，CUDA-Aware な MPI を扱うようにデバイスメモリを直接指定できる．また，*GMPI_Synchronize()* により，これまでホストにリクエストしたすべてのリクエストが完了するのを待つ．これは，GMPI のリングバッファ構造体 “gmpi_buf_t” のポインタである “buf” 内で管理される．将来的には，この関数を呼び出さずに，*GMPI_Waitall()* のような GMPI 関数の内部でホスト側との同期が自動的に取れる実装を検討している．この例に示されているように，ユーザは MPI 通信を CPU 側で動作する MPI プログラムとほとんど同じ方法で記述できる．そのため，ユーザが CUDA+MPI のコードに移植するためのコストが低減し，カーネル関数を起動する際のオーバーヘッドの削減にもつながると考えられる．

次に，Ping-Pong 通信のレイテンシの測定結果を**図 6** に示す．図中の凡例における “MVAPICH2-GDR” は 1 イテレーション毎に同期を行わない場合のレイテンシ，“MVAPICH2-GDR-SYNC” は 1 イテレーション毎に *cud-*

aStreamSynchronize() を利用して同期を行う場合のレイテンシを示している．前者は MVAPICH2-GDR の基本性能を参考にするために示しており，後者が実アプリケーションでの実際のコストに対応する．転送するデータのサイズを 4 bytes から 32KB まで増やしていき，1000 回の平均時間を実行速度として測定した．測定の結果，データサイズが大きくなるほど，通信リクエストに必要な情報の転送などのオーバーヘッドが相対的に小さくなり，従来のホスト上での MPI 実行の場合と比べ，性能の差が縮まっていくことがわかった．実装のベースである MVAPICH2-GDR-SYNC と比較すると，4KB 以上では同程度の通信性能が得られた．データサイズが小さい場合の性能差の原因として，GMPI 関数の内部でデバイスメモリの内容を同期する必要があることが挙げられる．これにより，GMPI 関数を実行する毎に，コンスタントに 10 μs 程度のオーバーヘッドが生じてしまうことがわかっている．このオーバーヘッドを削減することによる GMPI 関数の高速化が今後の課題として考えられる．しかしながら，GMPI はほとんど期待どおりの性能で動作し，従来手法と比べてプログラムの記述が簡単であることが本測定よりわかった．

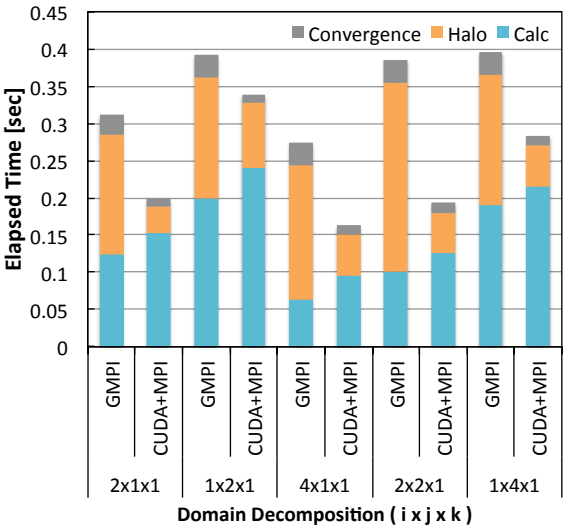

図 7 姫野ベンチマークの測定結果

表 3 姫野ベンチマークのコード行数

Method	Lines of code
CUDA+MPI version	1280
GMPI implementation	1033

5.3 姫野ベンチマークの性能評価

続いて，姫野ベンチマークの性能を評価することにより，実際のパフォーマンスを含む GMPI の動作を確認・評価する．姫野ベンチマークは，非圧縮性流体解析コードの性能評価のために開発されたもので，3 次元ポアソン方程式をヤコビ法で解く場合に主要なループの処理速度を測定するベンチマークである．姫野ベンチマークにおける通信は，袖領域交換と収束判定のための Allreduce である．本稿では，CUDA に移植した MPI 版の姫野ベンチマークを，NVIDIA Kepler アーキテクチャ向けに最適化し，i, j 次元方向で分割できるようにしたものを利用した [15][16]．i 次元方向の分割ではブロック転送，j 次元方向の分割ではブロックストライド転送が生じる．GMPI 版の姫野ベンチマークの実装では，元の実装において計算・袖領域交換・収束判定の 3 つを行う関数を 1 つのカーネル関数にした．更に，元の関数内で呼び出される関数を __device__ 関数化し，すべての MPI 関数を GMPI 関数に置き換えた．ベンチマーク中の通信では，Ping-Pong 通信で使用した *GMPI_Isend()* と *GMPI_Irecv()* に加えて，*GMPI_Waitall()* と *GMPI_Allreduce()* も使用する．CUDA デバイス上では各部分の詳細な実行時間を測定することが困難であるため，CUDA 内部のクロックカウンタを読み出す関数を用いて測定した結果とホスト側で測定した結果を用いて，実行時間を算出している．ヤコビ法の反復は 1000 回に固定し，問題サイズは SMALL($64 \times 64 \times 64$) を用い，最大で 4 ノードまで使用して評価を行った．前述の同時実行スレッド数の制限に依存するため，現状，姫野ベンチマークは問題サイズ SMALL のみしか実行できない．この制限に対する解決策を提案・実装し，問題サイズ MIDDLE や LARGE にも対応することは今後の課題である．

5.3.1 姫野ベンチマークの結果

測定結果を**図 7** に示す．図中の凡例における “Calc” は計算にかかった時間，“Halo” は袖領域交換にかかった時間，“Convergence” は収束判定の Allreduce にかかった時間を示している．CUDA+MPI 版と比較して，GMPI 版の計算時間は，66.2% から 87.9%となっている．しかしながら，袖領域交換と Allreduce を利用した収束判定の通信コストは従来手法より大きくなった．全体の実行時間を比較すると，何らかの理由により，1.16 倍から 1.99 倍程度の実行時間がかかっていることがわかった．

次に，CUDA+MPI の従来手法のコードと GMPI のコードの行数をそれぞれ**表 3** に示す．この表から， GMPI の実装は CUDA+MPI の実装と比べて，約 80.6%の行数となっていることがわかる．

5.3.2 姫野ベンチマークの考察

計算時間が減少した理由として，従来ホストで行われる同期のコストに比べて，カーネル関数内での同期コストが少ないことが考えられる．一方，メッセージサイズが十分に大きい場合，GMPI は従来手法とほぼ同程度の性能を達成することが予測されるため，通信時間が増加する結果は，Ping-Pong 通信の性能評価の結果と適合していない．CUDA のスレッドが必要とするレジスタ数が多くなると，GPU 内部のレジスタに収めきれなくなり，ローカルメモ

リに退避されるレジスタスピルが発生する．レジスタスピルの発生について調査したところ，CUDA+MPI 版ではスピルが発生していないのに対し， GMPI 版ではスピルが発生していた．更に，GMPI 版の GMPI 関数をコメントアウトすると，レジスタ使用数が 94 から 59 に減少した．このことから，GMPI 関数が使用するレジスタ数が多いため，実行時間に影響が出ていることが予想される．今後は，レジスタ数が増加する原因になる箇所の詳細を分析・改善し，実行時間がどのように変化するか調査していく予定である．この結果から，両方のベンチマークにおいて，CUDA+MPI の従来手法と比較して，プログラマビリティおよび MPI プログラムの CUDA 環境への移植性の高さを確認できた．このように，GMPI のアイデアにより， MPI のプログラムを GPU クラスタ向けに移植するコストが低減すると考えられる．

前述のとおり，同時実行可能なスレッド数の制限に対する解決策を検討し，スレッド数に関係なくカーネル関数内での同期を実現することが今後の課題である．更に，将来的には姫野ベンチマークの他の問題サイズをはじめ，様々なアプリケーションへの適用についても検討していく予定である．

6. 関連研究

電気通信大学の島らによって，カーネル関数内に MPI の関数を記述可能にする GPU プログラミングフレームワーク FLAT が提案されている [17]．FLAT はコンパイラのプリプロセッサとして実装されており，ソース to ソースのコード変換を行うことにより，カーネル関数に埋め込まれた MPI の関数の実際の処理を CPU コード上での処理に置き換える．したがって，FLAT のアイデアではカーネル関数の起動とデータの同期のコストの問題を解決することはできない．本システムは，カーネル関数に記述された MPI に準ずる関数の処理を，コンパイル時ではなく実行時にホスト側に依頼する機構で実現している．そのため，カーネル関数の起動と同期のステップを省略でき，カーネル関数がオーバーヘッドなしに実行し続けられる点が異なる．

本システムに似た実装として，Sapienza University of Rome で開発されていた CUQU (CUDA queue)[18], CUOS (CUDA Offloaded System services)[19] というライブラリが存在する．CUQU は CUDA を用いて開発されており，GPU カーネルとホスト側とのやりとりを Page-Locked Host Memory 上のリングバッファを介して行う機構である．CUOS は GPU カーネルからホストシステムのサービスを呼び出すためのフレームワークのプロトタイプであり， CUQU を利用してカーネル関数内から MPI の同期通信を行う例が実装されている．CUDA のバージョンは 4.0 前後を想定しており，特定の CUDA 環境に依存した実装となっている．これらのライブラリの開発は 2011 年 5 月で打ち切られており，その実用例も見当たらない．本研究では最近の CUDA 環境にも適用した，より高速かつ利便性の高いシステムの開発を目指す．

更に，The University of Texas at Austin の Kim らによって開発されている GPUnet[20] も本システムに似た実装として挙げられる．GPUnet は，RDMA over InfiniBand を用いて，GPU がソケット通信を行えるようにするライブラリである．CPU と GPU との間で共有しているリングバッファを介し，GPU が CPU にソケット通信を依頼する．CPU は共有しているリングバッファをポーリングし，GPU からのリクエストを処理する．CPU と GPU の間でデータを処理する方法が本システムとは異なるが，最も類似した実装である．GPUnet は ソケット通信を対象としているため，HPC アプリケーションに適用するには通信の粒度が細かく，プロトコルも詳細である．一方，GMPI は MPI レベルの通信を対象としており，通信をアプリケーション向けに，より抽象化することによって HPC 向けの性能向上を目指す．

7. おわりに

7.1 まとめ

本稿では，GPU カーネル内から MPI 通信を直接起動可能にする並列通信システム GMPI のユーザプログラミングモデルの定義と予備性能評価を行った．

GMPI システムでは，カーネル関数内で全スレッド間の同期を取る必要がある．GPU には同時実行可能なスレッド数に制限があり，現時点ではプリエンプション機能が扱えない．そのため，GMPI システムを用いて記述されるカーネル関数の総スレッド数は，同時実行可能な最大スレッド数以下に制限される．

GMPI の並列実行モデルを次のとおり定義した．ユーザは GMPI 関数の呼び出しをカーネル関数に記述でき，その関数は複数スレッドで平行して実行されない．GMPI は，古典的な OpenMP と MPI のハイブリッドプログラミングと比較し，ほぼ同様の考え方で扱うことができる．これにより，ユーザがプログラミングをしやすくなると考えた．

予備性能評価では，Ping-Pong 通信および姫野ベンチマークの性能評価を行った．Ping-Pong 通信の性能評価では，ユーザのプログラマビリティを損なうことなく，85%から同程度の相対性能を実現できた．一方，姫野ベンチマークでは，前述の制限により現状は問題サイズ SMALL しか動作しておらず，GMPI による実装は従来手法の約 66%の性能にとどまっている．この要因として，GMPI 関数がレジスタを多く使用するため，レジスタスピルが発生することが挙げられる．しかしながら，従来手法と比べて GMPI による実装のコード行数は減少しており，ユーザが GPU クラスタ向けのコードに移植する上でカーネル関数を分割

する必要がなくなり，移植性は改善していると考えられる．

7.2 今後の課題

GMPI システムには，レジスタ使用数が多く，レジスタスピルが発生してしまう問題など，最適化をしてオーバーヘッドを減らすべきいくつかの問題がある．現在，本システムは開発中であり，システムの冗長な部分やオーバーヘッドについて調査し，さらなるチューニングを行う予定である．特に，同時実行スレッド数の制限は重要な問題であるため，この解決策を検討・実装することで，姫野ベンチマークの問題サイズ MIDDLE，LARGE の測定も行っていく予定である．更に，より大規模なシステムでの詳細な実行条件についても検討していく予定である．

性能最適化やチューニングの後，姫野ベンチマークの他にも， NAS Parallel Benchmark[21] などのより複雑なベンチマークや実アプリケーションに対し本システムを適用することで，実用的な性能とプログラマビリティの評価を行う．更に，MPI の他に，筑波大学計算科学研究センターで開発が行われている密結合並列演算加速機構 TCA (Tightly Coupled Accelerators)[22] に対しても同様の機構を適用し，ライブラリの実装を行っていく予定である．最終的には，TCA に適用したライブラリを用いて，ベンチマークテストやアプリケーションを作成し，評価・比較を行う予定である．

謝辞

本研究の一部は，JST-CREST 研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」，研究課題「ポストペタスケール時代に向けた演算加速機構・通信機構統合環境の研究開発」による．また，本研究における HA-PACS/TCA の利用は，筑波大学計算科学研究センター学際共同利用プログラム・平成 27 年度課題「密結合演算加速機構アーキテクチャに向けたアプリケーションの開発と性能評価」による．

参考文献

[1] TOP500 Supercomputer Sites (online), http://top500.org/
[2] The Oak Ridge Leadership Computing Facility introduces Titan (online), https://www.olcf.ornl.gov/titan/
[3] GSIC. TSUBAME Computing Services (online), http://tsubame.gsic.titech.ac.jp/en
[4] The calculations were performed on the TianHe-1 (A) supercomputer located at National Supercomputer Center in Tianjin (online), http://nscc-tj.gov.cn/en/resources/resources_1.asp#TH-1A
[5] Center for Computational Science, University of Tsukuba : HA-PACS Project (online), http://www.ccs.tsukuba.ac.jp/eng/research-activities/projects/ha-pacs/
[6] CUDA C Programming Guide (online), http://docs.nvidia.com/cuda/cuda-runtime-api/index.html
[7] Message Passing Interface (MPI) Forum Home Page (online), http://www.mpi-forum.org/
[8] MVAPICH2: High performance MPI over InfiniBand/10GigE/iWARP and RoCE (online), http://mvapich.cse.ohio-state.edu/
[9] Open MPI: Open Source High Performance Computing (online)，http://www.open-mpi.org/
[10] NVIDIA Corp. : NVIDIA GPUDirect. (online), http://developer.nvidia.com/gpudirect
[11] Mellanox GPUDirect RDMA User Manual Rev 1.0 (online), http://www.mellanox.com/related-docs/prod_software/Mellanox_GPUDirect_User_Manual_v1.0.pdf
[12] 桑原 悠太, 塙 敏博, 児玉 祐悦, 朴 泰祐: GMPI：GPU クラスタにおける GPU セルフ MPI の提案, Vol.2015-HPC-151 No.12, 2015.
[13] NVIDIA Tesla P100 The Most Advanced Data Center Accelerator Ever Built. Featuring Pascal GP100, the World ' s Fastest GPU (online), https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
[14] Himeno benchmark, RIKEN, Japan. (online), http://accc.riken.jp/en/supercom/himenobmt/
[15] E. Phillips, M. Fatica, “Implementing the Himeno benchmark with CUDA on GPU clusters Parallel & Distributed Processing.” (IPDPS), 2010 IEEE International Symposium，pp.1 – 10, Apr. 2010.
[16] T. Hanawa, Y. Kodama, T. Boku, M. Sato, “Tightly Coupled Accelerators Architecture for Low-latency Inter-Node Communication Between Accelerators.” in SC14 poster, Nov. 2014.
[17] K. Shima, M. Yoshimi, T. Miyoshi, M. Kondo, H. Irie, H. Honda, T. Yoshinaga, “FLAT: An MPI Friendly GPGPU Programming Framework for GPU Clusters.” in IPSJ Transactions on Advanced Computing System (ACS) , Vol. 6，No.4，pp. 105 – 116 (2013) (in Japanese).
[18] cuqu A CPU ↔ GPU messaging queue (online), https://code.google.com/p/cuqu/
[19] cuos Offloaded System services for CUDA (online), https://code.google.com/p/cuos/
[20] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, M. Silberstein, “GPUnet: Networking Abstractions for GPU Programs.” in 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 201 - 216, Broomfield, CO, October 2014. USENIX Association.
[21] NAS Parallel Benchmark (online), http://mikilab.doshisha.ac.jp/dia/smpp/01_bench/naspara.html
[22] T. Hanawa, Y. Kodama, T. Boku, and M. Sato, “Tightly Coupled Accelerators Architecture for Minimizing Communication Latency among Accelerators.” in The Third International Workshop on Accelerators and Hybrid Exascale Systems (AsHES) in conjunction with IEEE 27th International Parallel and Distributed Processing Symposium (IPDPS), May 2013, pp. 1030 - 1039