Vol.2015-HPC-150 No.7
2015/8/4

# 演算加速機構を持つ並列クラスタ向けPGAS言語 XcalableACCの性能評価

田渕 晶大[1,a)]　中尾 昌広[2]　村井 均[2]　朴 泰祐[3,1]　佐藤 三久[2,1]

**概要**：GPU や Many Integrated Core (MIC) のような演算加速機構を持つクラスタが増加している．演算加速機構のプログラミングに OpenACC や OpenMP 4.0 を用いて MPI と組み合わせることで，比較的簡易にプログラムを記述できるようになったが，やはり MPI の記述が煩雑で生産性は低い．そこで我々は Partitioned Global Address Space (PGAS) 言語 XcalableMP と演算加速機構プログラミングモデル OpenACC を統合した XcalableACC (XACC) を提案している．XACC では逐次コードへの指示文の追加のみで，演算加速機構を持つクラスタ向けのプログラミングが可能である．XACC の指示文のうち reflect, reduction, gmove の一部を NVIDIA GPU 向けに実装しベンチマークで性能評価を行った．MPI+OpenACC と比較して Himeno Benchmark では最大で 94%，NAS Parallel Benchmarks (NPB) CG では最大で 97%の性能を達成した．また指示文による簡潔な記述により MPI+OpenACC と比較してコード行数を Himeno Benchmark では 49%，NPB CG では 73%に抑えられたことから，XACC は十分な性能と高い生産性があると言える．

## 1. 序論

ハイパフォーマンスコンピューティングの分野では計算性能を向上させるために Graphics Processing Unit (GPU) や Many Integrated Core (MIC) のような演算加速機構を搭載したクラスタが普及している．これら演算加速機構では数十から数千個の演算器を並列に動作させることで CPU よりも高い演算性能を達成できる．そのプログラミングには一般的に CUDA[1] や OpenCL[2] が使用されている．それらのプログラミングモデルでは，プログラマが演算加速機構上のデータの管理や実行する並列プログラムの作成を行う必要がある．そのため十分なチューニングが可能であるがコードが煩雑になってしまうため，生産性の低下が問題となっている．しかしながら OpenACC[3] や OpenMP 4.0[4] の登場により指示文を用いた記述が可能になり，演算加速機構の利用は容易になりつつある．

一方，分散メモリ環境上でのプログラミングには MPI が主に利用されている．MPI ではプログラマがすべての通信を明示するため十分なチューニングが可能であるが，記述量が多くコードが煩雑になるため，同じく生産性の低下が問題である．そこで MPI に代わるプログラミングモデルとして，Partitioned Global Address Space (PGAS) 言語である XcalableMP (XMP) [5] が提案されている．XMP では指示文により C・Fortran のコードをクラスタ上で並列に動作させ，ノード間の通信を行うことができるため，MPI よりも容易に開発が可能である．

また演算加速機構をもつクラスタ上では，例えば GPU クラスタであれば MPI と CUDA や MPI と OpenACC といった組み合わせによりプログラムを記述するのが一般的である．加えて先ほど述べた XMP と OpenACC を組み合わせた場合には，逐次コードに指示文のみ加えることで実装することが可能である．しかしながら，XMP と OpenACC の組み合わせには演算加速機構間の通信を直接記述する方法がないため，指示文の増加や通信性能の低下の問題がある．

そこで我々筑波大学 HPCS 研究室と理研 AICS プログラミング環境研究チームでは XMP と OpenACC を統合した言語仕様 XcalableACC (XACC) を提案している．XACC では XMP の通信指示文を拡張することで演算加速機構間の通信を記述可能とし，コンパイラが実行環境に合わせて適切な通信を行うようにする．XACC を用いることで MPI と OpenACC を組み合わせた場合と同等の性能を維

[1] 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
[2] 国立研究開発法人理化学研究所計算科学研究機構
RIKEN Advanced Institute for Computational Science
[3] 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[a)] tabuchi@hpcs.cs.tsukuba.ac.jp

持しつつ，より簡易にプログラミングが可能になると考えられる．

本稿では，NVIDIA GPU クラスタを対象にした XACC のコンパイラを実装し，Himeno Benchmark および NAS Parallel Benchmarks (NPB) の CG を用いて，その性能および生産性の評価を行う．

本稿の続きは次に示す構成となっている．2 章では関連研究を紹介する．3, 4 章では XMP および XACC についてコード例を交えて説明する．5 章では Omni XcalableACC Compiler の設計・実装を解説し，6 章でその評価を行う．7 章では結論と今後の課題を述べる．

## 2. 関連研究

XMP を GPGPU に対応させた XMP-dev が提案され，GPU への計算のオフロードと GPU 上のデータの袖領域通信が実装されている [6], [7]．XMP の拡張であるためオフロードに OpenACC を用いる XACC より簡潔な記述が可能であるが，GPU へのオフロードを制御する指示文が OpenACC より不足しており性能チューニングが難しいという問題がある．この点で XACC では演算加速機構へのオフロードに多くのコンパイラが対応している OpenACC を採用することで，豊富な記述が可能となると共にコンパイラの最適化による高い性能が期待できる．

Tightly Coupled Accelerators (TCA) を用いた XACC の通信の実装と評価が行われている [8]．袖領域通信を TCA により実装し，MPI を用いた実装よりも高い性能を達成している．しかしながら現在 TCA を利用できるクラスタは HA-PACS/TCA のみであるため，一般的な GPU クラスタでは利用することができない．本稿では MPI と CUDA を用いた汎用的な GPU 間通信の実装を行う．

他の PGAS 言語においては Unified Parallel C や OpenSHMEM で GPU のメモリへのアクセスや管理を行える拡張が行われている [9], [10]．さらに X10 や Chapel では構文を拡張することにより GPU のプログラミングを可能にしている [11], [12].

## 3. XcalableMP

XcalableMP[5] は PC クラスタコンソーシアムの XcalableMP 規格部会によって策定されている PGAS 言語の並列プログラミングモデルである．分散メモリ上での並列プログラミングのために C および Fortran に対して言語拡張を行っており，その多くは OpenMP に似た指示文である．XMP では指示文によりノード間の分散や通信を記述することで，コンパイラが分散と通信を行うコードを生成する．分散と通信をユーザが明示するため，プログラムの最適化を行いやすいという利点がある．

```
1  int a[N];
2  #pragma xmp template t(0:N-1)
3  #pragma xmp nodes p(4)
4  #pragma xmp distribute t(block) onto p
5  #pragma xmp align a[i] with t(i)
6
7  #pragma xmp loop (i) on t(i)
8  for(int i = 0; i < N; i++){
9      a[i] = i;
10 }
```

**図 1** XMP のグローバルビューモデルのコード例

### 3.1 実行モデルとメモリモデル

XMP の実行モデルは MPI と同じく Single Program Multiple Data (SPMD) である．すなわち通常は各ノードで重複して処理が実行され，指示文で指定されたときのみ異なるデータに対してそれぞれ処理が行われる．なお XMP におけるノードは MPI のプロセスのことであるため，物理的な 1 ノード上で複数の XMP ノードが実行されることもある．メモリに関しても MPI と同様に通常は各ノードで重複して確保される．そして指示文で指定されたときのみメモリを分割して各ノードで保持する．

### 3.2 グローバルビューモデル

XMP ではグローバルビューモデルとローカルビューモデルという 2 つのプログラミングモデルを提供しているが，本稿ではグローバルビューモデルのみ説明する．グローバルビューモデルは逐次プログラムに指示文でデータや処理の分散を指定することで並列プログラミングを行うプログラミングモデルである．グローバルビューモデルによるプログラム例を図 **1** に示す．この例を用いてプログラミング方法を説明する．

#### 3.2.1 データ・処理分散

`template` 指示文はテンプレートを定義する．テンプレートとは仮想インデックス空間であり，XMP ではテンプレートを用いてデータや処理の分散を記述する．例では 0 から N-1 までのインデックスをもつ 1 次元のテンプレートを定義している．`nodes` 指示文は XMP ノードの集合を定義する．`distribute` 指示文はテンプレートをノード集合に分散する．この際テンプレートの各次元の分散方法を指定でき，例では block 分散を指定している．`align` 指示文は配列をテンプレートに従ってノードに分散する．`loop` 指示文は for ループをテンプレートに従ってノードで分散して実行する．

#### 3.2.2 データ通信

XMP には定型的な通信を行うための指示文がいくつかあるが，ここでは XACC にて本稿で実装を行う `reflect`, `reduction`, `gmove` 指示文に関連する指示文のみ解説する．図 **2** に `shadow` と `reflect` 指示文の例を示す．`shadow` は分散配列の袖領域を定義する指示文である．袖領域とは隣

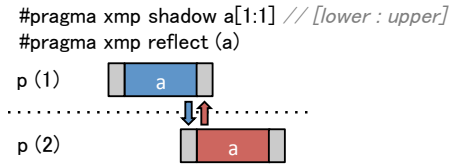



図 2　shadow・reflect の例

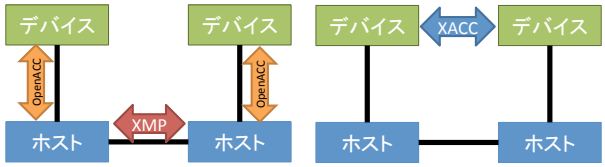
接するノードにある上端・下端の要素を保持するための領域である．分散配列の各次元の上端・下端に任意幅の袖領域を作成することが可能である．`reflect` は袖領域の更新を行う指示文である．隣接するノードと上端・下端を交換し，袖領域を最新の値に更新する．これらの指示文は主にステンシル計算で使用する．`reduction` は変数や配列の値をノード間でリデュースする指示文である．`gmove` は変数や分散配列に対する様々な代入処理を行うことが可能な指示文である．全対全通信が必要な分散配列から分散配列の代入などをコンパイラが自動で行う．

## 4. XcalableACC

XcalableACC は演算加速機構を持つクラスタで動作するプログラムを開発するためのプログラミングモデルである．既存のプログラミングモデルである XMP と OpenACC を組み合わせた際には，デバイス間の通信を行う指示文が存在しないため図 **3(a)** のように以下の 3 つの記述が必要となる．

(1) OpenACC 指示文によりデバイスからホストにデータをコピーする
(2) XMP 指示文によりホスト間で通信する
(3) OpenACC 指示文によりホストからデバイスにデータをコピーする

これらはそれぞれ別個に処理されるため，パイプライン化等の高速化が行えず通信性能低下の原因となる．XACC では既存の XMP に拡張を加えることで，図 **3(b)** のようにデバイス間の通信を記述可能とした．それにより XMP と OpenACC のハイブリッド並列化よりも簡潔に記述できるうえに，コンパイラが実行環境に合わせて最適な通信を行うことが可能になる．




(a) XMP + OpenACC　　(b) XACC

図 3　デバイス間通信の記述方法

### 4.1 デバイス間通信

XMP では `reflect`，`reduction`，`gmove` 等の通信指示文の対象はホスト上のデータである．XACC では以下のように XMP の通信指示文の最後に `acc` 節が存在する時はデバイス上のデータを対象に通信を実行する．

#pragma xmp *comm-directive* **acc**

図 **4** に XACC のコード例を示す．デバイスへのオフロードの部分は，基本的に XMP のコードに OpenACC 指示文を追加するだけである．16 行目では図 **5** のようにデバイス上にある分散配列 *a* に対して reflect を行っており，これによりデバイス間で袖領域の値が更新される．

```
1  int a[N];
2  #pragma xmp template t(0:N-1)
3  #pragma xmp nodes p(4)
4  #pragma xmp distribute t(block) onto p
5  #pragma xmp align a[i] with t(i)
6  #pragma xmp shadow a[1:1]
7
8  #pragma acc data copy(a)
9  {
10     #pragma xmp loop (i) on t(i)
11     #pragma acc parallel loop
12     for(int i = 0; i < N; i++){
13         a[i] = i;
14     }
15
16     #pragma xmp reflect (a) acc
17 }
```

図 4　XACC のコード例

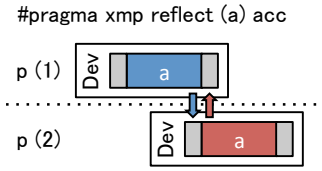



図 5　XACC によるデバイス間の reflect 通信

## 5. Omni XcalableACC Compiler

XMP のリファレンス実装である Omni XcalableMP Compiler を拡張し，NVIDIA GPU を対象とした XACC のコンパイラを実装する．本章では実装の概要と通信実装について述べる．

### 5.1 設計・実装方針

XMP のノード間のデータや処理の分散はそのまま利用し，デバイス間の通信機能を追加する．指示文のうち OpenACC の指示文は基本的にそのまま残し，OpenACC コンパイラでコンパイルを行うようにする．これによりデバイス用コードの生成等を既存の OpenACC コンパイラに委ねることが可能になる．またデバイス間の通信はライブラリ関数呼び出しに置き換えることで可搬性を保つ．図 **6** にコンパイル処理の流れを示す．XACC の指示文が書かれた C のコードを入力として与えると，そのコードを XACC のライブラリ関数呼び出しと OpenACC 指示文が含まれる

コードに変換する．それを OpenACC コンパイラを用いてコンパイルし，最後に XACC のライブラリとリンクすることで実行ファイルを作成する．

XACC のライブラリ関数では NVIDIA GPU 用に MPI と CUDA で通信を記述する．また MPI の send や recv バッファに GPU メモリのポインタを渡して通信することが可能な CUDA-Aware MPI を用いている場合には，実行時に環境変数を指定することでその機能を利用することができる．それにより，MVAPICH2[13] 等で対応している GPU Direct for RDMA を利用した低レイテンシな GPU 間通信が可能になる．

### 5.2 reflect 指示文

本来 reflect 指示文は通信の初期化や実行を適宜行うが，現在は処理を単純にするために通信の初期化を行う `reflect_init` 指示文と実際に通信を行う `reflect_do` 指示文に分けて実装されている．文献 [14] で XMP における reflect の効率的な実装が村井らによって行われており実装の参考とした．多次元配列を 1 次元目以外で分散した場合，袖領域は不連続（ストライドまたはブロックストライド）になる．XMP の reflect では袖領域が不連続の場合には MPI_Type_vector によって vector 型を定義して送る方法と pack してから送る方法の 2 種類が用意されており，通常は vector 型が使われる．しかしながら，MVAPICH2 を用いた GPU 間通信では pack する方が高速であったため，XACC の実装では通常は pack して送る実装となっている．データの pack,unpack には XACC のライブラリ内の CUDA カーネルが用いられる．

### 5.3 reduction 指示文

reduction の実装には XMP と同様に MPI_Allreduce を用いた．XMP および XACC の reduction 指示文では，あるアドレスのデータをリダクションした結果を同じアドレスに保存しなければならないので，MPI_Allreduce を呼び出す際には send buffer に MPI_IN_PLACE を指定する．CUDA-Aware MPI を用いている場合には，XMP の reduction 用ライブラリ関数に対して host_data 指示文により GPU のポインタを渡すように変更した．CUDA-Aware MPI でない場合には，GPU のデータをホストに確保したメモリにコピーし，そのデータに MPI_Allreduce を実行した後に GPU へ書き戻すように実装した．

### 5.4 gmove 指示文

gmove には非常に多くの通信パターンが存在するため，今回は評価に用いる NAS Parallel Benchmarks (NPB) CG で現れる 1 パターンについてのみ実装した．それは図 **7** のように，2 次元テンプレートのある次元で分散された 1 次元配列を同一テンプレートの他の次元で分散された 1 次元配列に代入するパターンである．

```
#pragma xmp nodes p(2,2)
#pragma xmp template t(0:N-1, 0:N-1)
#pragma xmp distribute t(block, block) onto p
double w[N], q[N];
#pragma xmp align w[i] with t(*,i)
#pragma xmp align q[i] with t(i,*)

#pragma xmp gmove acc
    q[:] = w[:];
```

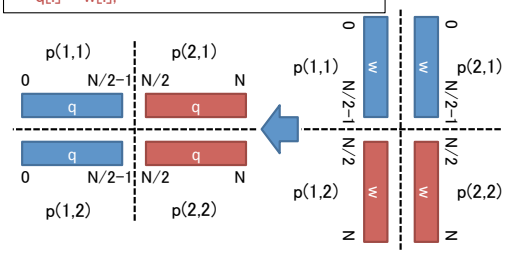



図 7　NPB CG にて用いられる gmove (4 ノード)

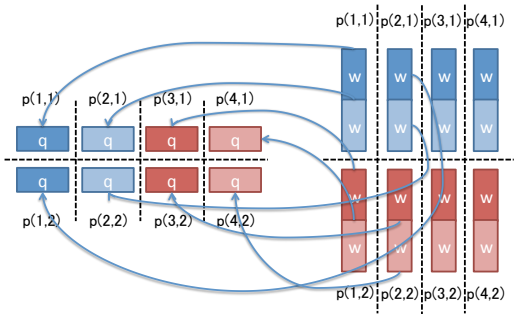



図 8　NPB CG の gmove の改善前 (8 ノード)

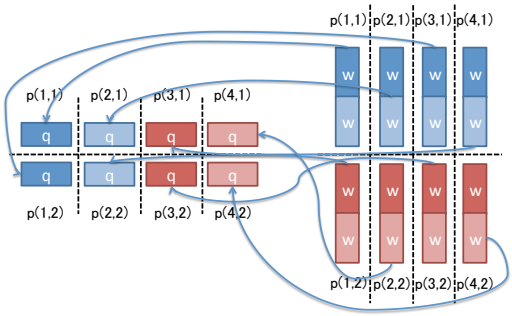



図 9　NPB CG の gmove の改善後 (8 ノード)

実装にあたり，最初に XMP の gmove の調査を行った．その結果，ノードの行数と列数が同じ場合には効率よく通信ができているが，ノードの行数と列数が異なる場合において非効率な通信が行われていた．例えば CG ではノードの列数が行数の倍となることがあり，その際に図 **8** に示す通信が行われていた．ノードの半分は 2 つのノードにデータを送る一方で，他の半分のノードはどこにもデータを送っていないためバランスが悪いうえに，同じノードで保持するデータも他のノードから送信しているので効率が悪い．そこで通信を図 **9** に示すように全ノードが 1 つのノードに送るように改善した．さらに現在の実装では通信相手を求める計算部分が非常に複雑になっているため，通信をキャッシュすることでその計算時間を短縮した．

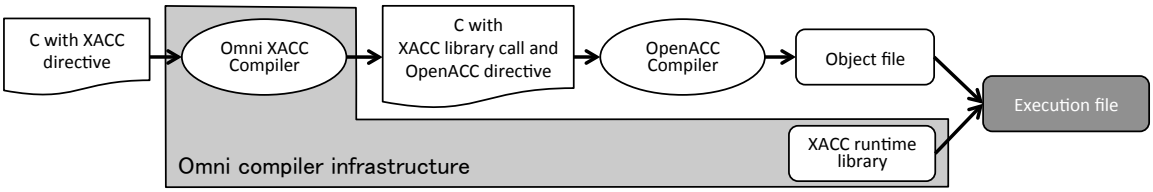



図 6　Omni XcalableACC Compiler のコンパイルの流れ

表 1　HA-PACS/TCA のノード構成

| | |
|---|---|
| CPU | Intel Xeon-E5 2680v2 2.8GHz x 2Socket |
| Memory | DDR3 1866MHz x 4 channel, 128GB |
| GPU | NVIDIA Tesla K20X x 4 (GDDR5 6GB) |
| Interconnect | InfiniBand: |
| | Mellanox Connect-X3 Dual-port QDR |
| Compiler | GCC 4.4.7, CUDA 6.5 |
| | MVAPICH2-GDR 2.1a |
| | Omni OpenACC Compiler 0.9.1a |

表 2　実行時に指定する MVAPICH2 用の環境変数

| |
|---|
| MV2_ENABLE_AFFINITY=0 |
| MV2_NUM_PORTS=2 |
| MV2_USE_CUDA=1, MV2_CUDA_IPC=0 |
| MV2_USE_GPUDIRECT_GDRCOPY=1 |
| MV2_USE_GPUDIRECT_RECEIVE_LIMIT=8192 |

## 6. 評価

XcalableACC の評価には筑波大学計算科学研究センターの HA-PACS/TCA を利用した．そのノード構成を表 1 に示す．1 ノードあたり 4 枚の GPU が搭載されているため，1MPI プロセス当たり 1GPU を割り当てて 1 ノードで 4MPI プロセスを実行し，最大で 16 ノード上で 64MPI プロセスを実行した．MPI には CUDA-Aware MPI である MVAPICH2-GDR 2.1a を利用し，mpicc のコンパイルオプションには “-O3” を指定した．OpenACC コンパイラには Omni OpenACC Compiler を用い，nvcc のコンパイルオプションには “-O3 -arch=sm_35” を指定した．実行時には MVAPICH2 用に**表 2** の環境変数を指定した．

### 6.1 Himeno Benchmark

Himeno Benchmark はポアッソン方程式解法をヤコビの反復法で解くベンチマークである [15]．複数の GPU を利用をするため，問題のサイズは Large を用いる．したがって，問題領域の大きさは $(i \times j \times k) = (256 \times 256 \times 512)$ である．k 次元の大きさは高々 512 であるうえ，分割すると袖領域がストライドになってしまうことから i と j 次元の 2 次元分割とした．GPU では i と j ループをスレッドブロックで，k ループをスレッドで並列化した．比較として Himeno Benchmark の MPI 版に OpenACC 指示文を追加した MPI+OpenACC 版を用意した．また元の MPI 版では不連続な袖の通信に vector 型を用いているが，それを pack してから送るようにした MPI+OpenACC (pack) も比較対象に加えた．

Himeno Benchmark の性能を**図 10** に，実行時間の内訳を**図 11** に示す．XACC の性能は MPI+OpenACC の 92.7〜98.2%，MPI+OpenACC (pack) の 91.9〜93.8%となった．実行時間の内訳から，ステンシル計算部分が性能差の主な原因となっていることが分かる．プロファイラで調査すると，スレッドブロックの SMX の占有率が MPI+OpenACC では 50%であるのに対して XACC ではカーネルで使用するレジスタ数が多いため 37.5%であった．使用するレジスタ数が増加した原因はループと配列アクセスの書き換えが原因である．XACC では分散するループの下限・上限・ステップが変数として置き換わる．また配列はポインタに置き換えられるため，配列の各次元のサイズを保持する変数も増加する．特に Himeno Benchmark では主演算カーネルで 7 個 (p,a,b,c,bnd,wrk1,wrk2) の配列にアクセスする必要があるため，配列の書き換えによる変数の増加がレジスタ数増加の主な原因となっている．加えて XACC のノード間の分散において演算が不均等に分散されているのも原因の 1 つと考えられる．Himeno Benchmark では Large の場合，実際に使用する配列サイズは [257][257][513] である．例としてこれを 1 次元目で 2 ノードに分散する場合を考えると，各ノードに割り当てられるテンプレートの 1 次元目のサイズは $\lceil 257/2 \rceil = 129$ となる．ステンシル計算のイテレーションはテンプレートの 1 次元目の (1:254) の部分に該当するため，ノード 1 は (1:128) = 128 要素，ノード 2 は (129:254) = 126 要素の計算をすることになる．このように演算が不均等に分割されることにより，均等に分割されている MPI+OpenACC よりもステンシル計算の実行時間が増加する．

次に袖通信時間を**図 12** に示す．XACC の袖通信時間は MPI+OpenACC の 91.9〜128%，MPI+OpenACC (pack) の 101〜128%である．MPI+OpenACC では不連続な袖を pack することで通信時間が削減されている．しかしながら，XACC では $2 \times 1 \times 1$ や $4 \times 1 \times 1$ 分割において通信時間が大きく増加している．これは通信量の違いによるものである．**表 3** は袖通信で送られる要素数の比較である．前述したように，データの分散が MPI+OpenACC と XACC で異なるため袖領域の大きさも異なる．XACC の方が各

表 3　Himeno Benchmark の袖通信の要素数

| 分割数 | MPI+OpenACC | | XACC | |
|---|---|---|---|---|
| i×j×k | jk 平面 | ik 平面 | jk 平面 | ik 平面 |
| 2 × 1 × 1 | 131328 | - | 132867 | - |
| 4 × 1 × 1 | 131328 | - | 132867 | - |
| 4 × 2 × 1 | 66177 | 33792 | 67203 | 34371 |
| 4 × 4 × 1 | 33858 | 33792 | 34371 | 34371 |
| 8 × 4 × 1 | 33858 | 17408 | 34371 | 17955 |
| 8 × 8 × 1 | 17442 | 17408 | 17955 | 17955 |

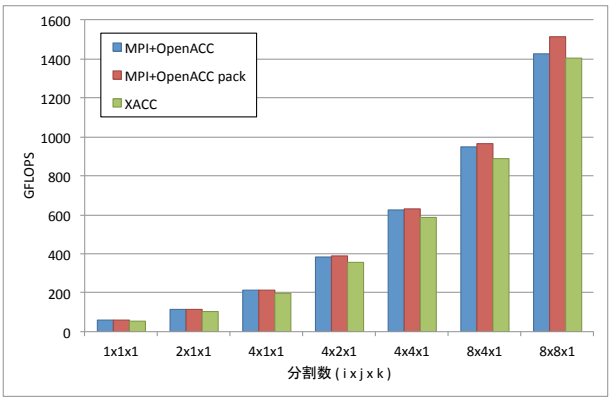



図 10　Himeno Benchmark の性能

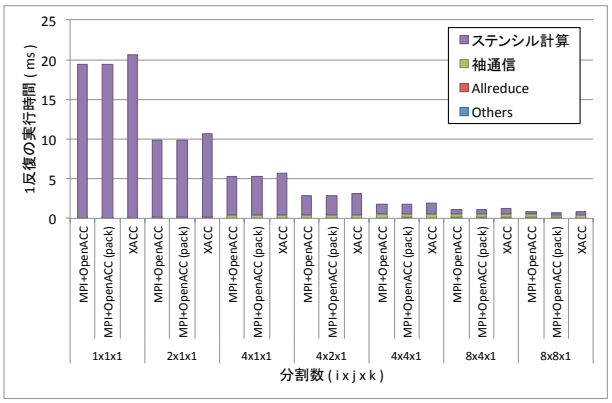



図 11　Himeno Benchmark の 1 反復の実行時間の内訳

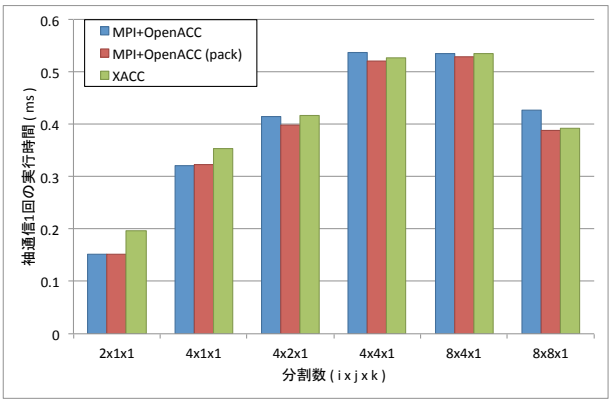



図 12　Himeno Benchmark の袖通信 1 回の実行時間

ノードの領域を大きく確保したため，XACC の方が最大で 3.1%多く通信していた．

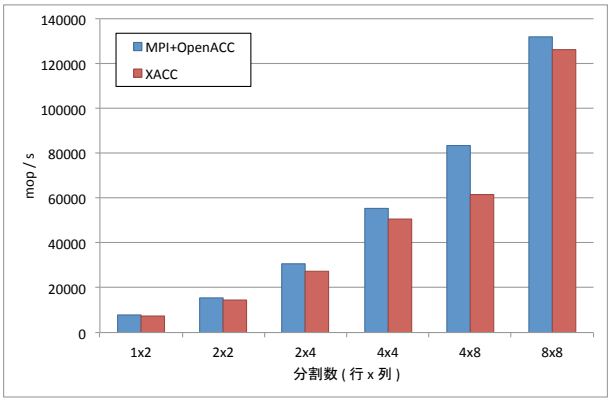



図 13　NPB CG の性能

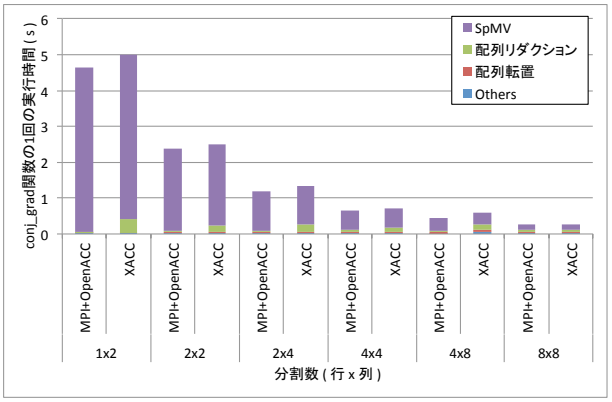



図 14　conj_grad 関数の 1 回の実行時間の内訳

### 6.2　NAS Parallel Benchmarks CG

NPB CG は正値対称な大規模疎行列の最小固有値を共役勾配法によって解くベンチマークである．複数の GPU を用いた際にスケールするよう，行列サイズが 1500000×1500000 である Class D を用いた．XMP による CG の実装は文献 [16] で行われている．その実装と同じく XACC の実装でも行列を 2 次元分割した．このとき主となる通信は

(1) 疎行列ベクトル積の結果を各行で足し合わせるための配列の Allreduce
(2) (1) の結果を行分割から列分割にするための配列の転置

の 2 つである．(1) は reduction 指示文で，(2) は gmove 指示文で記述される．

NPB CG の性能を図 **13** に，また主要な関数である conj_grad の実行時間の内訳を図 **14** に示す．XACC は MPI+OpenACC の 74.7%〜96.8%の性能が得られている．特に 4×8 分割のときの性能低下が大きい． 実行時間を比較すると SpMV や配列転置の実行時間は同等であるが，配列リダクションの実行時間に大きな差があり XACC は MPI+OpenACC の 1.0〜4.4 倍の時間がかかっている．この差は 2 つの要因から生じている．1 つ目はリダクションの実装の違いである．MPI+OpenACC では，send-recv と配列を加算するループによって Recursive-Doubling 法

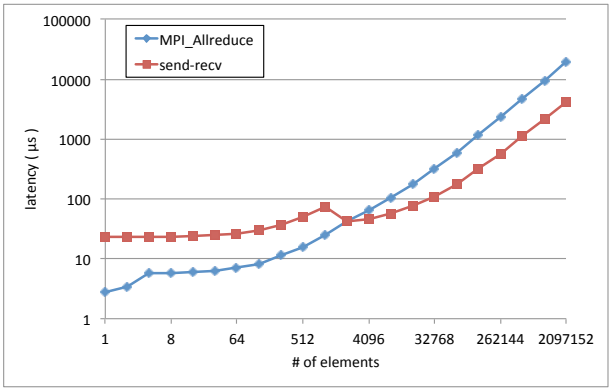



**図 15** Allreduce のレイテンシ (2 プロセス)

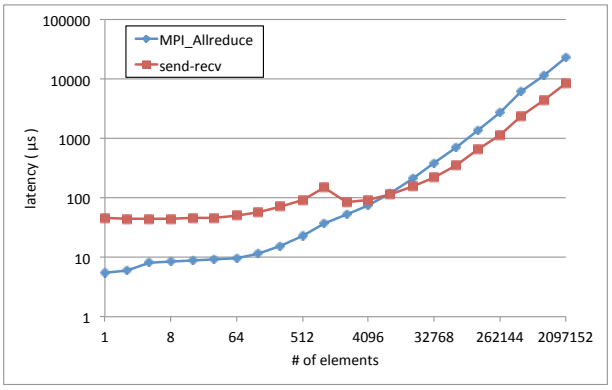



**図 16** Allreduce のレイテンシ (4 プロセス)

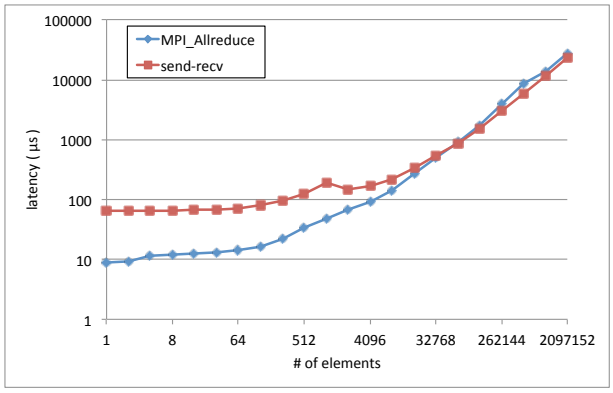



**図 17** Allreduce のレイテンシ (8 プロセス)

**表 4** CG の Class D における配列 w の長さ

| 分割数 (行 × 列) | 1 × 1 | 1 × 2 | 2 × 2 | 2 × 4 |
|---|---|---|---|---|
| 配列長 | 1500000 | 1500000 | 750000 | 750000 |
| 分割数 (行 × 列) | 4 × 4 | 4 × 8 | 8 × 8 | |
| 配列長 | 375000 | 375000 | 187500 | |

**表 5** Himeno Benchmark のコード行数

| | XMP | OpenACC | 指示文以外 |
|---|---|---|---|
| MPI+OpenACC | - | 13 | 315 |
| MPI+OpenACC (pack) | - | 27 | 369 |
| XACC | 28 | 9 | 155 |

によるリダクションを実現している．そのため，GPU メモリ間の通信と GPU での配列加算が行われる．一方で XACC では，MPI_Allreduce によりリダクションを行う．MVAPICH2 の実装では，GPU 上のデータの Allreduce はホストにコピーしてホスト上で Allreduce を行う．すなわち，ホストメモリ間の通信と CPU での配列加算が行われる．**図 15**，**図 16**，**図 17** は CG で用いられるのと同じ double 型配列の Allreduce を MPI_Allreduce を用いた場合と send-recv とカーネルで実現した場合のレイテンシの比較である．配列長が短い場合には通信レイテンシの低さから CPU で計算する MPI_Allreduce の方が高速であるが，配列長が長い場合には計算の速さから GPU で計算する send-recv を用いた実装の方が高速である．プロセス数が増えると共に send-recv を用いた方が良くなるサイズが大きくなるのは，GPU 間通信やカーネル実行のレイテンシが大きいからである．CG の各分割パターンでの配列 w の配列長を**表 4** に示す．これらのデータから，8×8 の分割ではレイテンシに大きな差は無いが，その他の分割では MPI_Allreduce よりも send-recv による実装の方が良いことが分かる．

2 つ目は処理する配列長の違いである．行の分割数と列の分割数が異なるとき図 9 のように配列の転置で必要になる配列要素は，偶数列のノードでは配列 w の前半分，奇数列のノードでは w の後半分のみである．したがって，MPI+OpenACC ではすべての要素をリダクションせずに必要な部分のみリダクションを行うようになっている．一方，XACC は必ず配列 w の全要素をリダクションするためこの場合には MPI+OpenACC の倍のデータの通信と演算が必要になる．2×4，4×8 分割にて性能低下が大きいのはこれが原因である．

### 6.3 生産性

XACC では逐次コードへの指示文の追加のみで，複数ノードおよびデバイスへのデータと処理のオフロードが可能である．またノード間の分散は XMP で，デバイスへのオフロードは OpenACC で記述するため，分散とオフロードを区別してプログラムを書きやすい．XMP+OpenACC で問題であったデバイス間の通信も XMP 指示文の拡張によってホストでの通信と同様に簡潔に記述可能である．

XACC の記述のしやすさを定量的に評価するためにベンチマークのコードの行数を計測した．Himeno Benchmark の行数を表 5 に，NPB CG の行数を表 6 に示す．Himeno Benchmark では XACC の行数は MPI+OpenACC の行数の 59%，MPI+OpenACC (pack) の 48%であり，NPB CG では 73%である．これは XACC ではノード間の分散や MPI による通信を指示文で記述することでコードが簡潔になったからであり，XACC の生産性の高さを示している．

表 6 NPB CG のコード行数

| | XMP | OpenACC | 指示文以外 |
|---|---|---|---|
| MPI+OpenACC | - | 23 | 748 |
| XACC | 47 | 23 | 493 |

## 7. 結論と今後の課題

本稿では，演算加速機構を持つクラスタ向けのプログラミングモデル XcalableACC のコンパイラを実装し，Himeno Benchmark および NAS Parallel Benchmarks の CG を用いてその性能を評価した．Himeno Benchmark を用いた評価では MPI+OpenACC の 92.7〜98.2%，MPI+OpenACC (pack) の 91.9〜93.8%の性能が得られた．不連続な袖を pack して送ることで MPI のベクトル型を用いるよりも通信時間を短縮できたうえ，同様に pack する MPI+OpenACC の実装にも近い通信性能が得られた．NPB CG では MPI+OpenACC の 74.7%〜96.8%の性能が得られた．配列の転置の通信は gmove を改善することで，MPI+OpenACC と同等の性能となった．これら 2 つのベンチマークによる評価から XACC は MPI と OpenACC を組み合わせたプログラミングと比較して十分な性能と高い生産性があると言える．

### 7.1 今後の課題

今後は分散配列の書き換えの改善を行う予定である．Himeno Benchmark での性能低下の原因となっていた使用する変数の増加を防ぐために，コンパイル時にノード数が決まっている場合にはポインタではなく固定サイズの配列にすることでインデックス計算に必要な変数をなくすことを計画している．また，多様な gmove の通信パターンへの対応や XMP のローカルビューモデルで使用される coarray への対応を進めた上で，実アプリケーションを用いた評価を行う必要がある．

**謝辞** 本研究に際しご協力頂いた理化学研究所計算科学研究機構プログラミング環境研究チームの皆様に深く感謝する．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」，ならびに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「アクセラレータクラスタにおける高生産言語 XcalableACC の開発と評価」による．

### 参考文献

[1] NVIDIA: *Parallel Programming and Computing Platform — CUDA*, http://www.nvidia.com/object/cuda_home_new.html.
[2] Khronos Group: *OpenCL - The open standard for parallel programming of heterogeneous systems*, https://www.khronos.org/opencl/.
[3] OpenACC-Standard.org: *OpenACC Home*, http://www.openacc-standard.org.
[4] OpenMP Architecture Review Board: *OpenMP Application Program Interface Version 4.0 - July 2013*, http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.
[5] XcalableMP Specification Working Group: *Xcalablemp specification version 1.2*, http://www.xcalablemp.org/download/spec/xmp-spec-1.2.pdf (2013).
[6] 李 珍泌，チャントゥアンミン，小田嶋哲哉，朴 泰祐，佐藤三久：PGAS 並列プログラミング言語 XcalableMP における演算加速装置を持つクラスタ向け拡張仕様の提案と試作，情報処理学会論文誌コンピューティングシステム（ACS），Vol. 5, No. 2, pp. 33–50 (2012).
[7] 野水拓馬，高橋大介，李 珍泌，朴 泰祐，佐藤三久：並列言語 XcalableMP のアクセラレータ向け言語拡張の OpenCL 実装，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]，Vol. 2012, No. 9, pp. 1–8 (2012).
[8] Nakao, M., Murai, H., Shimosaka, T., Tabuchi, A., Hanawa, T., Kodama, Y., Boku, T. and Sato, M.: XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters, *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, Piscataway, NJ, USA, pp. 27–36 (2014).
[9] Chen, L., Liu, L., Tang, S., Huang, L., Jing, Z., Xu, S., Zhang, D. and Shou, B.: Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Vol. 6548, Springer Berlin Heidelberg, pp. 151–165 (2011).
[10] Potluri, S., Bureddy, D., Wang, H., Subramoni, H. and Panda, D.: Extending OpenSHMEM for GPU Computing, *Parallel and Distributed Processing Symposium, International*, pp. 1001–1012 (2013).
[11] Cunningham, D., Bordawekar, R. and Saraswat, V.: GPU Programming in a High Level Language: Compiling X10 to CUDA, *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, New York, NY, USA, ACM, pp. 1–10 (2011).
[12] Sidelnik, A., Chamberlain, B. L., Garzaran, M. J. and Padua, D.: Using the high productivity language chapel to target gpgpu architectures, Technical report, Cray (2011).
[13] : *MVAPICH*, http://mvapich.cse.ohio-state.edu/.
[14] Hitoshi, M. and Mitsuhisa, S.: An Efficient Implementation of Stencil Communication for the XcalableMP PGAS Parallel Programming Language, 7th International Conference on PGAS Programming Models (2013).
[15] 理化学研究所情報基盤センター：姫野ベンチマーク，http://accc.riken.jp/2145.htm.
[16] Nakao, M., Lee, J., Boku, T. and Sato, M.: XcalableMP Implementation and Performance of NAS Parallel Benchmarks, *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, New York, NY, USA, ACM, pp. 11:1–11:10 (2010).