情報処理学会論文誌　コンピューティングシステム　Vol.9　No.1　17–29 (Mar. 2016)

演算加速機構を持つクラスタ向けPGAS言語 XcalableACCの評価

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

受付日 2015年7月31日, 採録日 2015年11月22日

概要：GPU や MIC のような演算加速機構を持つクラスタが広く使われている．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，比較的簡易に演算加速機構を持つクラスタ向けのプログラムを記述できるようになったが，それでもなお MPI の記述が煩雑であるため生産性が低いという問題がある．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 では逐次コードに指示文を追加することにより，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．本稿では，XACC の通信指示文の一部を 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 97%，NAS Parallel Benchmarks（NPB）CG では最大で 96%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 51%，NPB CG では 79%に抑えられたことから，XACC は高い性能と生産性があるといえる．

キーワード：演算加速機構，GPU，クラスタ，PGAS 言語

Evaluation of A PGAS Language XcalableACC for Accelerator Clusters

Akihiro Tabuchi[1,a)]　Masahiro Nakao[2]　Hitoshi Murai[2]　Taisuke Boku[1,3]
Mitsuhisa Sato[1,2]

Received: July 31, 2015, Accepted: November 22, 2015

Abstract: Clusters equipped with accelerators such as GPU and MIC are widely used. For these clusters, programmers can develop their applications relatively easily by combining MPI with OpenACC or OpenMP 4.0, but lower productivity due to complex MPI programming is still a problem. We have been proposing XcalableACC (XACC), which is an integration of a Partitioned Global Address Space (PGAS) language XcalableMP (XMP) and OpenACC. XACC enables programmers to develop applications for accelerator clusters just by adding directives to a serial version of the code. In this paper, we show the implementation of the XACC communication directives for NVIDIA GPU and evaluated their performance using two benchmarks. The performance of the XACC version against MPI+OpenACC version is up to 97% for Himeno Benchmark and up to 96% for NAS Parallel Benchmarks (NPB) CG. The code size of XACC version against MPI+OpenACC version is 51% for Himeno Benchmark and 79% for NPB CG. Therefore, XACC features fully high performance and productivity.

Keywords: accelerator, GPU, cluster, PGAS language

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．演算加速機構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可能である．そのプログラミングには

[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, Tsukuba, Ibaraki 305–8577, Japan
[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, Kobe, Hyogo 650–0047, Japan
[3] 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[a)] tabuchi@hpcs.cs.tsukuba.ac.jp

一般的に CUDA [1] や OpenCL [2] が使用されており，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．しかしながら標準規格である 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ったことで，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持つクラスタ上では，たとえば GPU クラスタであれば MPI と CUDA や MPI と OpenACC といった組合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を加えるだけで実装することが可能である．しかしながら，XMP と OpenACC の組合せには演算加速機構間の通信を直接記述する方法がなく，XMP によるホスト間の通信と OpenACC による演算加速機構とホスト間の通信により記述する必要があるため，指示文の増加や通信の分割による性能低下の問題がある．OpenACC が提案される以前に，XMP を演算加速機構を持つクラスタ向けに拡張した XMP-dev が李らによって提案され，XMP にデバイス上のデータ宣言，ホスト・デバイス間のデータ転送，デバイスでの並列ループ実行，およびデバイス間の通信を加えることで演算加速機構を持つクラスタで利用できる記述能力と性能を有することを明らかにした [6]．XMP と OpenACC の組合せと XMP-dev の違いはデバイス間の通信の有無であり，XMP-dev ではデバイス間の通信が性能向上に大きく貢献することが示された．そのため XMP と OpenACC の組合せにおいても，デバイス間の通信を追加することで性能向上が見込まれる．

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．

以上の背景に基づき本稿では Omni XcalableACC Compiler を NVIDIA GPU クラスタ向けに実装する．これは source-to-source コンパイラ用インフラストラクチャである Omni Compiler 上に実装された Omni XcalableMP Compiler の拡張である．その XACC コンパイラを用いて，HPC 分野でよく現れるステンシル計算の典型例である Himeno Benchmark と共役勾配法の典型例である NAS Parallel Benchmarks CG により，XACC の性能および生産性を評価する．

本稿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-dev はオフロードに独自の指示文を用いていたが，XACC は OpenACC が標準規格として提案されたことを受けて，XMP-dev を発展させ，オフロードを OpenACC に沿った指示文にすることにより OpenACC ユーザにより分かりやすくなるようにした．またとくにデータの管理に関して，XMP-dev ではデバイス上にデータがあるかないかにかかわらずデータを確保することしかできなかったが，XACC ではオフロードについて OpenACC を取り入れることにより，すでにある場合には確保しないなど，十分な操作ができるようになった．性能の点では，XMP-dev は CUDA に変換し，XACC は OpenACC に変換するという違いがあるが，OpenACC はさらに CUDA に変換（コンパイル）することができるので，同等な記述に関しては同等な性能が得られるはずである．ただし，OpenACC は CUDA よりも高レベルな記述であるため新規のプラットフォームに対応しやすいが，XMP-dev では開発当時の NVIDIA GPU のアーキテクチャ（Fermi）に合わせた CUDA コードになっており，現在主流のアーキテクチャ（Kepler）には適さない可能性がある．

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．本稿では一般的な GPU クラスタでも動作するよう GPU 間通信を MPI と CUDA により実装し，その場合でも XACC が有効であるかを調べる．

他の PGAS 言語においては Unified Parallel C（UPC）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おり，ともに GPU 間の通信に対応している [9], [10]．UPC は言語拡張，OpenSHMEM はライブラリベースの PGAS 言語であり，GPU のプログラミングに指示文ベースの OpenACC を用いる場合には，異なるプログラミング手法の組合せとなる．XACC では XMP が指示文ベースの PGAS 言語であるため，同じ指示文ベースの OpenACC との親和性が高い．そのため，既存の XMP のコードもしくは OpenACC のコードからシームレスに記述できるという利点がある．さらに UPC や 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], [13]．Chapel の拡張では 1 ノード上の GPU での実行にのみ対応している．UPC や X10 の拡張では複数ノード上の GPU の利用に対応しているが，ホスト・GPU 間の通信のみで GPU 間の通信については考慮されていない．XACC では複数ノード上のデバイスを利用可能であり，さらにデバイス間の通信にも対応することで，大規模システムにおいても高い性能向上が見込まれる．

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 並列プログラミング言語である．分散メモリ上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・同期を記述することで，コンパイラがそれら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，ユーザがプログラムの最適化を行いやすいという利点がある．

3.1 実行モデルとメモリモデル

XMP における実行の単位は “ノード” と呼ばれる．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並列処理，ノード間通信やノードごとの処理を行う．XMP におけるノードは通常の実装では MPI のプロセスに対応するため，物理的な 1 ノード上で複数の XMP のノードを実行することもできる．メモリモデルに関しても通常は MPI と同様に各ノードでデータは重複して確保されるが，指示文で分散が指定されたときのみデータ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，ここでは本稿で主に用いるグローバルビューモデルのみを説明する．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．

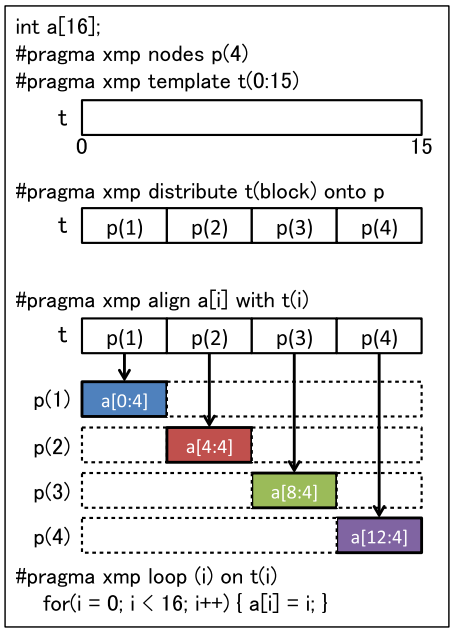


図 1　グローバルビューモデルのプログラム例
Fig. 1　Example of a global-view model.

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 15 までのインデックスを持つ 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どのように分散するかを指定する指示文であり，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存在するが，ここでは本稿で XACC 上で実装する `reflect`，`reduction`，`gmove` 指示文を解説する．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集計する指示文である．演算子には OpenMP と同様に+，*，

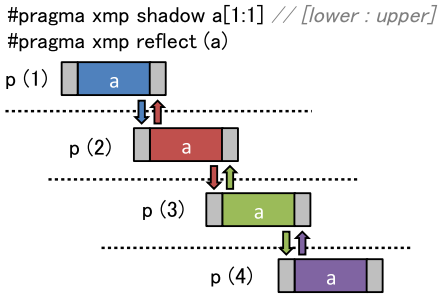


図 2　shadow・reflect の例

Fig. 2　Examples of the shadow and reflect directives.

```
#pragma xmp nodes p(4)
#pragma xmp template t(0:7)
#pragma xmp distribute t(block) onto p

int a[8], b, c[8][8], d[8][8];
#pragma xmp align a[i] with t(i)
#pragma xmp align c[*][i] with t(i)
#pragma xmp align d[i][*] with t(i)

// (a) send-recv
#pragma xmp gmove
    a[0:2] = a[4:2];

// (b) broadcast
#pragma xmp gmove
    b = a[0:1];

// (c) all-to-all
#pragma xmp gmove
    d[:][:] = c[:][:];
```


図 3　gmove の例

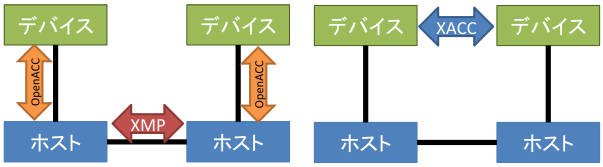
Fig. 3　Examples of the gmove directive.

min，max 等を指定できる．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．代表的な 3 種類の通信を**図 3** に示す．(a) は分散配列から分散配列への代入でこの例では send-recv の通信となる．(b) は分散配列からローカル配列への代入で，この例では broadcast の通信となる．(c) は分散配列から分散配列への転置で，all-to-all の通信となる．

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 PGAS 言語である．既存の PGAS 言語である XMP と演算加速機構のプログラミングモデルである 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 4** (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．



(a) XMP + OpenACC　(b) XACC

図 4　デバイス間通信の記述方法

Fig. 4　Description methods for inter-device communication.

```
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 5　XACC のコード例

Fig. 5　Example code of the XACC.

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 の通信指示文に拡張を加えることにより，図 4 (b) のように 1 つの指示文で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 1 行で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デバイス間の通信を行うことが可能になる．

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` 節が存在するとき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 5 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，11 行目のように基本的に XMP のコードに OpenACC 指示文を追加するだけである．デバイス間の通信は 16 行目のように XMP の reflect 指示文に `acc` 節を追加して記述することで，デバイス上の分散配列 a の袖領域が更新される．

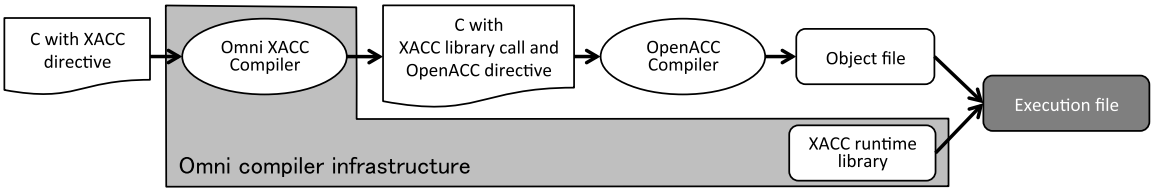


図 6　Omni XcalableACC Compiler のコンパイルの流れ
Fig. 6　Flow of compilation in the Omni XcalableACC compiler.

5. Omni XcalableACC Compiler

XMP のリファレンス実装である Omni XcalableMP Compiler [14] 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．**図 6** にコンパイル処理の流れを示す．

5.1　設計・実装方針

Omni XMP Compiler は source-to-source 変換を行うトランスレータであるため，Omni XACC Compiler も同様にトランスレータである．XACC の指示文のうち XMP 指示文に対してはコード変換を行い，OpenACC 指示文に対しては基本的に何も行わない．これにより OpenACC のコンパイルを既存の OpenACC コンパイラに委ねることが可能になる．通常の XMP 指示文による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．その際，デバイス間の通信はランタイム呼び出しに置き換えることで可搬性を保つ．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 のランタイム呼び出しと 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 のランタイムライブラリには通常の XMP のランタイムに加えデバイス間通信用ランタイムが含まれる．NVIDIA GPU 用の通信ランタイムは MPI と CUDA で記述する．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 [15] 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．そのほかに，Nakao ら [8] によって実装された TCA を用いた GPU 間通信ランタイムも含まれている．TCA が利用可能な環境ではそちらを利用することで，一般的な InfiniBand による通信よりも低レイテンシな通信が可能である．

5.2　reflect 指示文

XMP の reflect 指示文はそれのみで通信の初期化や実行を適宜行うが, XACC では通信の初期化を行う `reflect_init` 指示文と実際に袖通信を行う `reflect_do` 指示文に分けることにより，通信スケジュールを再利用して最適化する．文献 [16] で XMP における reflect の効率的な実装が Murai 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 して送る実装とした．データの pack,unpack には XACC ランタイムライブラリ内の CUDA カーネルを利用する．

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 には任意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．

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良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．たとえば 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

```
#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[:];
```

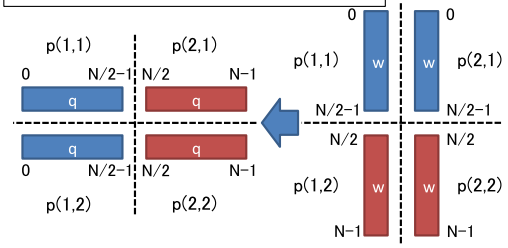


図 7　NPB CG にて用いられる gmove（4 ノード）
Fig. 7　A gmove in NPB CG (4 nodes).

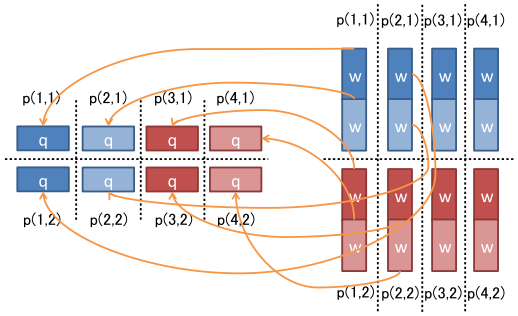


図 8　NPB CG の gmove の改善前（8 ノード）
Fig. 8　A gmove before improvement in NPB CG (8 nodes).

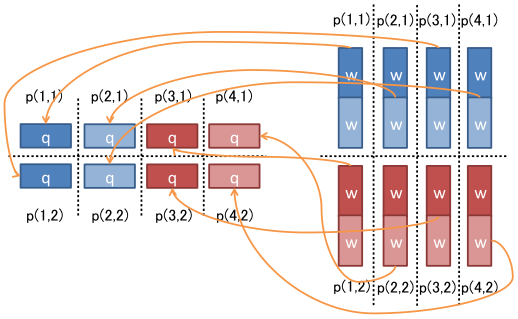


図 9　NPB CG の gmove の改善後（8 ノード）
Fig. 9　A gmove after improvement in NPB CG (8 nodes).

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や送受信の範囲を求める計算部分が非常に複雑になっているため，通信相手と送受信の範囲をキャッシュすることで 2 度目以降はただちに MPI 通信を開始できるようにした．

表 1　HA-PACS/TCA のノード構成
Table 1　Node configulation of HA-PACS/TCA.

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

表 2　実行時に指定する MVAPICH2 用の環境変数
Table 2　Environment variables specified at runtime for MVAPICH2.

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[*1] 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 プロセスあたり 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 [17] 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** に示す環境変数を指定した．

6.1　Himeno Benchmark

Himeno Benchmark は非圧縮流体解析コードの性能を評価するためのベンチマークである [18]．主な演算はポアッソン方程式解法をヤコビの反復法で解く 3 次元の 19 点ステンシルであり，主な通信は袖領域の交換である．複数の GPU を利用するため，問題のサイズは Large を用いた．Large では問題領域の大きさは $(i \times j \times k) = (256 \times 256 \times 512)$ である．**図 10** に XACC による Himeno Benchmark のコードの一部を示す．k 次元の大きさはたかだか 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ので i と j 次元の 2 次元分割とした．主計算であるステンシル計算は i, j, k の 3 重ループで構成されており，GPU で並列処理するために i, j ループをスレッドブロック（OpenACC における gang）で，k ループをスレッド（OpenACC における vector）で並列化した．比較として Himeno Benchmark の

[*1] 今回は TCA を用いていない．

```
#pragma xmp template t(0:MKMAX-1, 0:MJMAX-1, 0:MIMAX-1)
#pragma xmp nodes n(1, NDY, NDX)
#pragma xmp distribute t(block, block, block) onto n
#pragma xmp align p[k][j][i] with t(i, j, k)
#pragma xmp align bnd[k][j][i] with t(i, j, k)
#pragma xmp align wrk1[k][j][i] with t(i, j, k)
#pragma xmp align wrk2[k][j][i] with t(i, j, k)
#pragma xmp align a[*][k][j][i] with t(i, j, k)
#pragma xmp align b[*][k][j][i] with t(i, j, k)
#pragma xmp align c[*][k][j][i] with t(i, j, k)
#pragma xmp shadow p[1:2][1:2][0:1]
#pragma xmp shadow bnd[1:2][1:2][0:1]
#pragma xmp shadow wrk1[1:2][1:2][0:1]
#pragma xmp shadow wrk2[1:2][1:2][0:1]
#pragma xmp shadow a[0][1:2][1:2][0:1]
#pragma xmp shadow b[0][1:2][1:2][0:1]
#pragma xmp shadow c[0][1:2][1:2][0:1]
...
#pragma acc data copy(p,bnd,wrk1,wrk2,a,b,c),create(gosa)
{
#pragma xmp reflect_init(p) width(1,1,0) acc
    ...
  for(n=0 ; n<nn ; ++n){
    gosa = 0.0;
#pragma acc update device(gosa)

#pragma xmp loop (k,j,i) on t(k,j,i)
#pragma acc parallel loop firstprivate(omega) reduction\
      (+:gosa) collapse(2) gang vector_length(64) async
    for(i=1 ; i<imax-1 ; ++i)
      for(j=1 ; j<jmax-1 ; ++j){
#pragma acc loop vector reduction(+:gosa) private(s0,ss)
       for(k=1 ; k<kmax-1 ; ++k){
         s0 = a[0][i][j][k]*p[i+1][j][k]+a[1][i][j][k]
           *p[i][j+1][k]+a[2][i][j][k]*p[i][j][k+1]
           +b[0][i][j][k]*(p[i+1][j+1][k]-p[i+1][j-1][k]
           -p[i-1][j+1][k]+p[i-1][j-1][k])+b[1][i][j][k]
          *(p[i][j+1][k+1]-p[i][j-1][k+1]-p[i][j+1][k-1]
           +p[i][j-1][k-1])+b[2][i][j][k]
          *(p[i+1][j][k+1]-p[i-1][j][k+1]-p[i+1][j][k-1]
           +p[i-1][j][k-1])+c[0][i][j][k]*p[i-1][j][k]
           +c[1][i][j][k]*p[i][j-1][k]+c[2][i][j][k]
           *p[i][j][k-1]+wrk1[i][j][k];
         ss = ( s0 * a[3][i][j][k] - p[i][j][k] )
            * bnd[i][j][k];
         gosa += ss*ss;
         wrk2[i][j][k] = p[i][j][k] + omega * ss;
       }
     }
#pragma xmp loop (k,j,i) on t(k,j,i)
#pragma acc parallel loop collapse(2) gang \
                          vector_length(64) async
    for(i=1 ; i<imax-1 ; ++i)
      for(j=1 ; j<jmax-1 ; ++j){
#pragma acc loop vector
       for(k=1 ; k<kmax-1 ; ++k)
         p[i][j][k] = wrk2[i][j][k];
     }
#pragma acc wait
#pragma xmp reflect_do(p) acc
#pragma acc update host(gosa)
#pragma xmp reduction(+:gosa)
  } /* end n loop */
...
} //end of acc data
...
```

図 10　XACC による Himeno Benchmark のコード
Fig. 10　XACC Code of Himeno Benchmark.

MPI 版に OpenACC 指示文を追加した “MPI+OpenACC” 版を用意した．オリジナルの MPI 版では袖領域の通信に MPI の派生型であるベクトル型を用いているが，jk 平面では padding も含めて連続型として通信するように改変した．また，XACC では ik 平面を pack して連続データとして通信するため，同様に pack するようにした “MPI+OpenACC (pack)” も比較対象に加えた．さらに，XACC の機能を用いずに XMP と OpenACC で記述した “XMP+OpenACC” とも比較する．性能測定時の反復回数は 1000 回とし 10 回の計測のうち最良値を使用した．

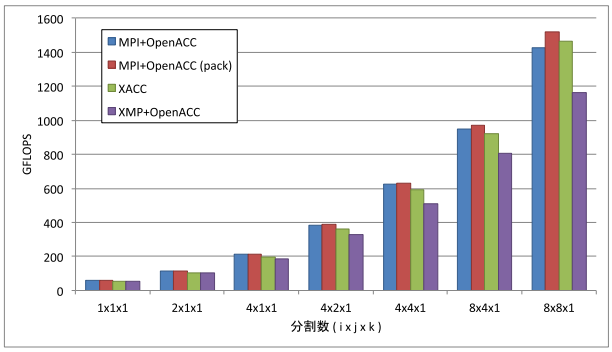


図 11　Himeno Benchmark の性能
Fig. 11　Performance of Himeno Benchmark.

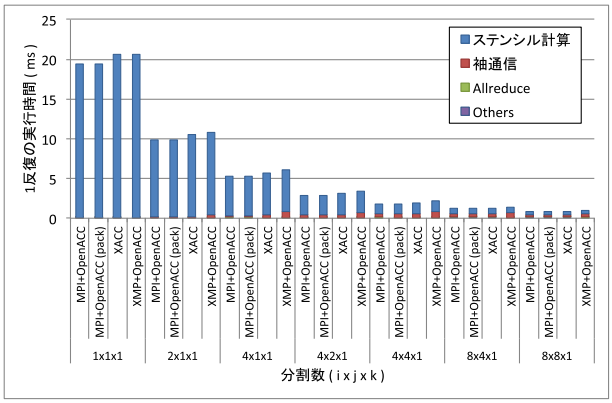


図 12　Himeno Benchmark の 1 反復の実行時間の内訳
Fig. 12　Breakdown of execution time per iteration of Himeno Benchmark.

Himeno Benchmark の性能を図 **11** に，1 反復の実行時間の内訳を図 **12** に示す．XACC における性能は MPI+OpenACC と比較して 93.7～102.8%，MPI+OpenACC (pack) と比較して 93.4～96.6%であった．一方，XMP+OpenACC では分割数が増えた際の性能向上率が悪く，8×8×1 分割では XACC の 8 割ほどの性能にとどまっている．実行時間の内訳から，ステンシル計算が全体の 4～9 割程を占めており，XACC や XMP+OpenACC では計算時間が MPI+OpenACC と比較して 6.5～13.5%長いことが分かる．

ステンシル計算時間の差は配列へのアクセス方法の違いによる．MPI+OpenACC では，実行ノード数から配列サイズを事前に決定するため，静的配列へのアクセスとしてコンパイルされる．XACC や XMP+OpenACC では，分散配列は malloc で動的に確保されるよう変換されるため，

表 3　Himeno Benchmark のカーネルによる占有率や性能の違い
Table 3　Difference of occupancy and performance of kernels in Himeno Benchmark.

	レジスタ数	占有率 (%)	GFLOPS
MPI+OpenACC	42	50.0	57.6
XACC	74	37.5	54.1
XACC (reg=72)	72	43.8	55.0
XACC (reg=64)	64	50.0	54.9
XACC (shrink)	66	43.8	57.3

ポインタへのアクセスとしてコンパイルされる．例として p[i][j][k] は * (p + acc0 * i + acc1 * j + k) に変換される．次元の大きさを保持する acc0 や acc1 は分散配列ごとに用意される．この変換により，
(1) インデックス計算の増加
(2) カーネルの実行率の低下
が生じる．(1) が生じるのは，分散配列 p 以外の分散配列は 3 次元配列なら [i][j][k]，4 次元配列なら [0-3][i][j][k] とほぼ同じインデックスでアクセスするにもかかわらず，そのオフセット計算に必要な変数 acc が分散配列ごとに異なるため，それぞれオフセット計算が必要になるからである．(2) が生じるのは，使用する変数の増加により GPU カーネルで必要なレジスタ数が増加し，同時実行可能なブロック数が減少するからである．これらの要因の影響を調査するため**表 3** に示すようにカーネルを変えて 1GPU での性能を測定した．占有率は GPU の SMX が実行可能なワープ数に対してどれだけ実行しているかを表す割合であり，この割合が高いほど良い性能が出やすい．“XACC (reg=72)” および “XACC (reg=64)” はカーネルコンパイル時にオプションによって最大レジスタ数を制限したもので，72 や 64 は占有率が変わる境界値である．レジスタ数を制限することで若干性能は向上するが，“XACC (reg=64)” ではレジスタスピルによる性能低下が起こった．“XACC (shrink)” は分散配列ごとに別々に使用していたオフセット計算用の変数を共有して削減したもので，MPI+OpenACC に匹敵する性能が得られたことからインデックス計算の増加が性能低下の主要な原因であることが明らかになった．

次に袖通信の時間を**図 13** に示す．XACC では MPI+OpenACC と比較して 93.4〜114%，MPI+OpenACC (pack) と比較して 100〜115% である．MPI+OpenACC (pack) や XACC 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 × 1 × 1 や 4 × 1 × 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 4** は袖通信で送られる要素数の比較である．MPI+OpenACC では分割しない次元には袖領域を確保せず，また分割する次元でも 2 プロセスでの分割の場合には片方だけしか袖領域を確保しない．一方 XACC では分割方法にかかわらず shadow で定義された袖領域を確保する．それにより，たとえば 2 × 1 × 1 分割では XACC の方が j 次元の大きさが 2 要素大きくなり，送る要素数としては 2 × 513 要素多くなる．このように分割が少ないときに XACC では MPI+OpenACC よりも多く通信することになり通信時間が増加した．XMP+OpenACC においては，通信量は XACC と同じであるがホストとデバイス間のデータコピーを OpenACC の update 指示文で記述したことにより通信時間が大幅に増加した．XACC の通信で用いている MVAPICH2 では，データがデバイスからホスト，ホストからホスト，ホストからデバイスへパイプライン転送されるが，XMP+OpenACC ではそれらを順次行うことになるからである．

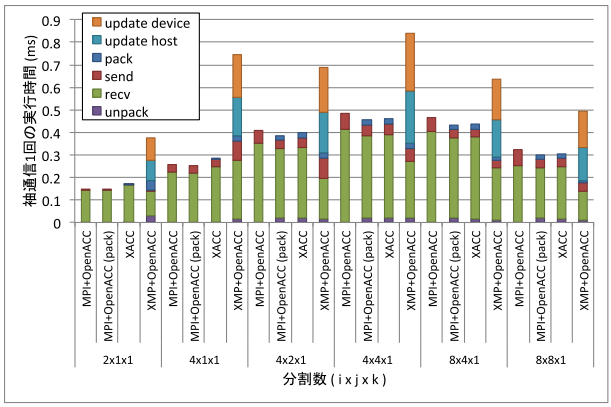


図 13　Himeno Benchmark の袖通信 1 回の実行時間
Fig. 13　Execution time of a halo communication in Himeno Benchmark.

表 4　Himeno Benchmark の袖通信の要素数
Table 4　Number of halo communication elements in Himeno Benchmark.

分割数	MPI+OpenACC		XACC	
i×j×k	jk 平面	ik 平面	jk 平面	ik 平面
2 × 1 × 1	256×513	-	258×513	-
4 × 1 × 1	256×513	-	258×513	-
4 × 2 × 1	129×513	66×512	130×513	66×512
4 × 4 × 1	66×513	66×512	66×513	66×512
8 × 4 × 1	66×513	34×512	66×513	34×512
8 × 8 × 1	34×513	34×512	34×513	34×512

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 NPB CG の実装は文献 [19] で行われている．MPI 版と同じく 2 次元分割であり，XACC の実装でも同様の分割とした．**図 14** に XACC による NPB CG のコードの一部を示す．主計算である疎行列ベクトル積は行と列の 2 重ループとなっており，GPU では行のループをスレッドブロックで，列のルー

```
1  #pragma xmp nodes p(NUM_PROC_COLS,NUM_PROC_ROWS)
2  #pragma xmp nodes sub_p(NUM_PROC_COLS)=p(:,*)
3  #pragma xmp template t(0:NA-1,0:NA-1)
4  #pragma xmp distribute t(block, block) onto p
5  #pragma xmp align w[i] with t(*,i)
6  #pragma xmp align q[i] with t(i,*)
7  #pragma xmp align r[i] with t(i,*)
8  #pragma xmp align p[i] with t(i,*)
9  #pragma xmp align x[i] with t(i,*)
10 #pragma xmp align z[i] with t(i,*)
11 ...
12 #pragma acc data copy(p,q,r,x,z,w,rowstr[0:NA+1]\
13                         , a[0:NZ], colidx[0:NZ])
14 {
15   ...
16   for (cgit = 1; cgit <= cgitmax; cgit++) {
17     rho0 = rho; d = 0.0; rho = 0.0;
18 #pragma xmp loop on t(*,j)
19 #pragma acc parallel loop gang
20     for(j=0; j < NA; j++){
21       double sum = 0.0;
22       int rowstr_j = rowstr[j];
23       int rowstr_j1 = rowstr[j+1];
24 #pragma acc loop vector reduction(+:sum)
25       for (k = rowstr_j; k < rowstr_j1; k++) {
26         sum = sum + a[k]*p[colidx[k]];
27       }
28       w[j] = sum;
29     }
30
31 #pragma xmp reduction(+:w) on sub_p(:) acc
32 #pragma xmp gmove acc
33     q[:] = w[:];
34
35 #pragma xmp loop on t(*,j)
36 #pragma acc parallel loop
37     for (j = 0; j < NA; j++)
38       w[j] = 0.0;
39 #pragma xmp loop on t(j,*)
40 #pragma acc parallel loop reduction(+:d)
41     for (j = 0; j < NA; j++)
42       d = d + p[j] * q[j];
43 #pragma xmp reduction(+:d) on sub_p(:)
44     alpha = rho0 / d;
45 #pragma xmp loop on t(j,*)
46 #pragma acc parallel loop
47     for (j = 0; j < NA; j++ ){
48       z[j] = z[j] + alpha*p[j];
49       r[j] = r[j] - alpha*q[j];
50     }
51 #pragma xmp loop on t(j,*)
52 #pragma acc parallel loop reduction(+:rho)
53     for (j = 0; j < NA; j++ )
54       rho = rho + r[j] * r[j];
55 #pragma xmp reduction(+:rho) on sub_p(:)
56     beta = rho / rho0;
57 #pragma xmp loop on t(j,*)
58 #pragma acc parallel loop
59     for (j = 0; j < NA; j++)
60       p[j] = r[j] + beta*p[j];
61   } /* end of do cgit=1,cgitmax */
62   ...
63 }//end of acc data
```

図 14　XACC による NPB CG のコード
Fig. 14　XACC code of NPB CG.

プをスレッドで並列化した．また主な通信は

(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

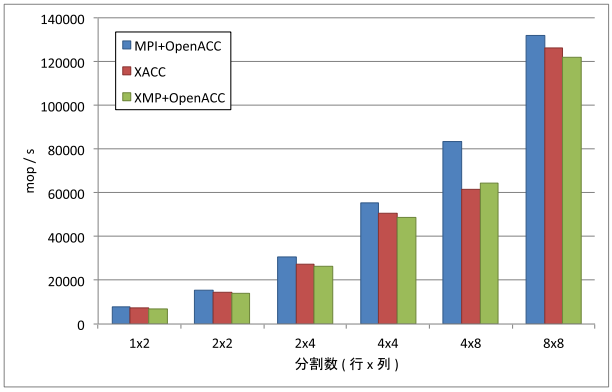


図 15　NPB CG の性能
Fig. 15　Performance of NPB CG.

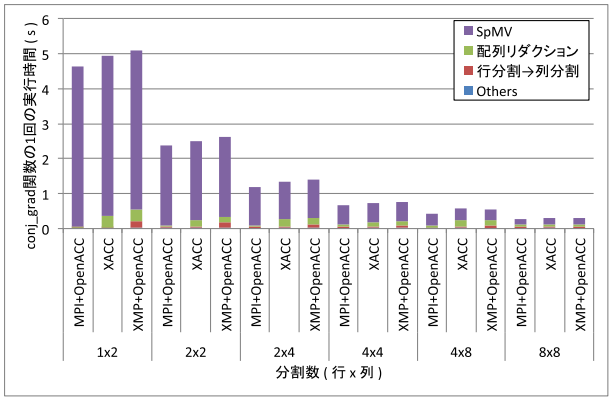


図 16　conj_grad 関数の 1 回の実行時間の内訳
Fig. 16　Breakdown of exection time of the conf_grad function.

(2) (1) の結果を行分割から列分割にするための通信の 2 つであり，(1) は 31 行目の reduction 指示文で，(2) は 32 行目の gmove 指示文で記述した．元の MPI 版は Fortran で記述されていたため，C で書き直してさらに OpenACC 指示文を追加した “MPI+OpenACC” を比較対象として実装した．また，Himeno Benchmark と同じく XMP と OpenACC の組合せで記述した “XMP+OpenACC” との比較も行う．なおこの問題サイズでは GPU のメモリ不足により 1 プロセスでの実行ができなかったため，2〜64 プロセスでの評価のみ行った．

NPB CG の性能を**図 15** に，また主関数である conj_grad の 1 回の実行時間の内訳を**図 16** に示す．XACC では MPI+OpenACC と比較して 73.5%〜95.7%の性能が得られている．とくに 4 × 8 分割のときの性能低下が大きい．また XMP+OpenACC では MPI+OpenACC と比較して 76.7%〜92.6%の性能となっており，XACC と比べると 4×8 分割では性能が 4%向上し，その他では 3〜4%低下している．

実行時間を比較すると XACC では SpMV や行分割から列分割にする通信の実行時間は MPI+OpenACC と同等であるが，配列リダクションの実行時間に大きな差があり 1.0〜4.4 倍の時間がかかっている．XMP+OpenACC ではホストとデバイス間の通信を OpenACC の update 指

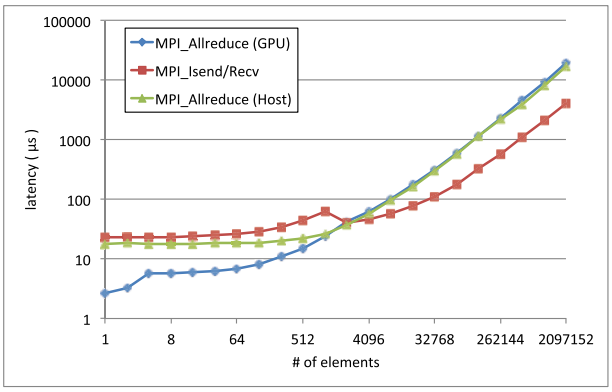


図 17　Allreduce のレイテンシ（2 プロセス）
Fig. 17　Latency of Allreduce (2 processes).

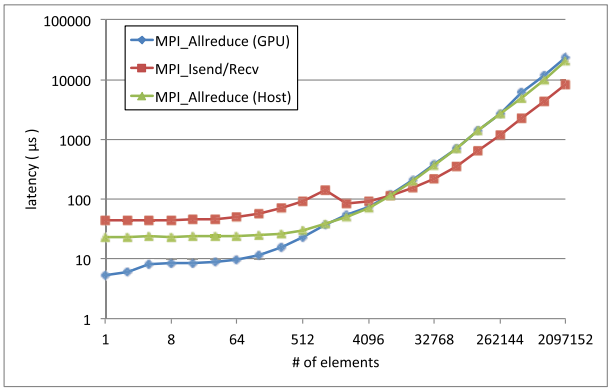


図 18　Allreduce のレイテンシ（4 プロセス）
Fig. 18　Latency of Allreduce (4 processes).

示文で記述したことにより，行分割から列分割にする通信は XACC と比べて増加しているが，一方で配列リダクションは同じか減少している．これらの差は 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，XACC では MPI_Allreduce によりリダクションを行う．MVAPICH2 では，GPU メモリ上の Allreduce はホストメモリにコピーしてホスト上で Allreduce を行うように実装されている．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．XMP+OpenACC はホストメモリと GPU メモリ間のコピーを OpenACC で行う以外は XACC と同じである．**図 17**，**図 18**，**図 19** は NPB CG で用いられるのと同じ GPU メモリ上の double 型配列の Allreduce のレイテンシの比較である．“MPI_Allreduce (GPU)” は MPI_Allreduce に GPU メモリのポインタを渡す場合，“MPI_Isend/Recv” は allreduce を send-recv とカーネルで実現した場合，“MPI_Allreduce (Host)” は CUDA でホストにデータを送り，MPI_Allreduce にホストメモリのポインタを渡す場合である．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 (GPU) 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ともに 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．NPB CG の各分割パターンでの配列 w の配列長を**表 5** 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ないが，その他の分割では MPI_Allreduce (GPU) よりも send-recv による実装の方が良いことが分かる．また同じ MPI_Allreduce を使う場合でも 512 要素以下の場合には GPU Direct for RDMA を用いる MPI_Allreduce (GPU) の方がレイテンシが短い．それ以上の要素数ではおおよそ同じではあるが，部分的に MPI_Allreduce (Host) のほうがレイテンシが短くなる．4×8 分割において XMP+OpenACC が XACC の性能を上回ったのはこれが原因である．

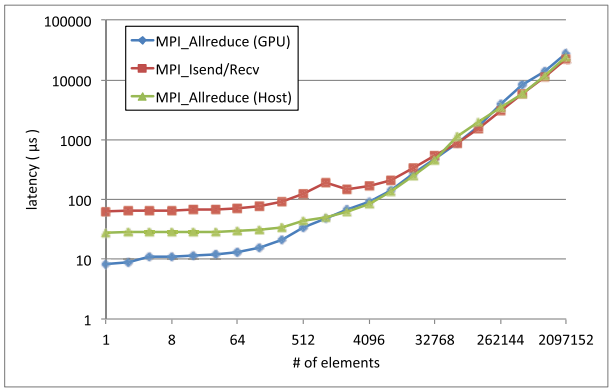


図 19　Allreduce のレイテンシ（8 プロセス）
Fig. 19　Latency of Allreduce (8 processes).

表 5　NPB CG の Class D における配列 w の長さ
Table 5　Length of array w at Class D of NPB CG.

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	

2 つ目の原因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 のように行分割から列分割にするための通信に必要な配列要素は，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．そこで 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ことで通信量を減らしている．一方，XACC や XMP+OpenACC は必ず配列 w の全要素をリダクションするため，行と列の分割数が異なる場合には MPI+OpenACC と比較して 2 倍の通信と演算が必要になる．それにより 2 × 4，4 × 8 分割で配列リダクションの時間が大きく増加した．

表 6　Himeno Benchmark のコード行数（内数は通信関連の行数）
Table 6　Code lines of Himeno Benchmark (included number is code lines related to communication).

	XMP	OpenACC	指示文以外
逐次プログラム	-	-	146
MPI+OpenACC	-	13 (4)	315 (108)
MPI+OpenACC (pack)	-	21 (12)	369 (162)
XMP+OpenACC	33 (7)	21 (12)	181 (26)
XACC	34 (9)	9 (0)	155 (0)

表 7　Himeno Benchmark の逐次コードへの変更行数（指示文は除く）
Table 7　Changed lines from serial code of Himeno Benchmark (exclude directives).

	追加	修正	削除
MPI+OpenACC	180	12	11
MPI+OpenACC (pack)	234	12	11
XMP+OpenACC	41	5	6
XACC	15	5	6

6.3　生産性

XACC では逐次コードへの指示文の追加のみで，複数ノードの分散，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．

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数えた．Himeno Benchmark の行数を**表 6** に，逐次コードからの変更行数を**表 7** に示す．XACC の全体の行数は MPI+OpenACC の 60%，MPI+OpenACC (pack) の 51%である．XACC や XMP+OpenACC では MPI+OpenACC に比べて非常に少ない修正で記述が可能である．MPI+OpenACC ではデータ・処理分散のために約 60 行ほど必要であるのに対して，XACC では指示文によるデータ・処理分散の記述により 25 行で済んでいる．また通信に関しては，袖通信は通信相手と回数が多いため MPI+OpenACC は全体の 33%，pack を加えたコードでは 42%の行数を必要としているのに対して，XMP+OpenACC は reflect 指示文により簡易に記述可能であるため全体の 19%に抑えられている．さらに XMP+OpenACC では袖通信の際に必要となる部分のみをホスト・デバイス間でコピーするために逐次コードへの変更と OpenACC 指示文が必要であるが，XACC ではデバイス間の reflect 指示文 1 行で済むため全体の 5%の行数となり，さらに記述が簡易となった．XACC で指示文以外の逐次コードに加えた変更は時刻取得関数の変更，XMP 指示文の効果範囲を指定するための中カッコ，ヘッダのインクルードの追加等であり，ほぼ逐次コードをそのまま利用できた．

表 8　NPB CG のコード行数（内数は通信関連の行数）
Table 8　Code lines of NPB CG (included number is code lines related to communication).

	XMP	OpenACC	指示文以外
逐次プログラム	-	-	444
MPI+OpenACC	-	24 (5)	748 (224)
XMP+OpenACC	48 (20)	28 (8)	541 (5)
XACC	48 (20)	20 (0)	541 (5)

表 9　NPB CG の逐次コードへの変更行数（指示文は除く）
Table 9　Changed lines from serial code of NPB CG (exclude directives).

	追加	修正	削除
MPI+OpenACC	304	43	0
XMP+OpenACC	109	16	12
XACC	109	16	12

次に NPB CG の行数を**表 8** に，逐次コードからの変更行数を**表 9** に示す．MPI+OpenACC に対する XMP+OpenACC の全体の行数は 80%, XACC は 79%である．XMP+OpenACC と XACC との差は通信時にホストとデバイス間でコピーを行うための OpenACC update 指示文のみである．分散や通信のために逐次コードに対して加えた変更は，MPI+OpenACC と比べて XMP+OpenACC や XACC では 200 行ほど少ない．全体に対する通信の割合を見ると，配列リダクションや行分割から列分割への代入処理を send/recv で記述した MPI+OpenACC は 30%であるのに対して，reduction，gmove 指示文で記述した XMP+OpenACC は 5.3%，さらにデバイス間の通信とした XACC では 4.1%であった．XMP+OpenACC や XACC における指示文以外の変更のうち，約半分はプロセス数が 2 冪であるかのチェックで，残りは分散する配列をグローバル変数として宣言するための変更，疎行列の列インデックスのオフセットの修正，2 つのノルムのリダクションを一度に行うための一時配列への代入と参照，ランク 0 のみが実行する部分の条件文，OpenACC 指示文の範囲を指定するための中カッコ等である．プログラムの実際の動作に大きく影響するのは疎行列の列インデックスのオフセットの修正のみであるため，記述のコストは低いといえる．

2 つのベンチマークにおけるコードの比較から，XACC や XMP+OpenACC では指示文によるデータ・分散の記述により各ノードのデータや処理の範囲を計算する処理が必要ないため，逐次コードへの変更が少なくなることや，袖通信のような典型的な通信パターンに対して，専用の指示文を用いることで非常に簡潔に記述可能であることが分かる．さらに XMP+OpenACC ではデバイス間の通信の際にホスト・デバイス間の通信を記述する必要があるが，XACC では必要がないため XMP+OpenACC よりも簡易に記述

できる．以上のことから，XACC は MPI+OpenACC や XMP+OpenACC によるプログラミングよりも生産性が高いといえる．

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 93.7〜102.8%，MPI+OpenACC (pack) の 93.4〜96.6%の性能が得られた．不連続な袖を pack して送ることで MPI の派生型である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 73.5%〜95.7%の性能が得られた．行分割配列から列分割配列への通信は 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いえる．

今後は XMP と XACC における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配列のインデックス計算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 通信への対応を進めたうえで，実アプリケーションを用いた評価を行う必要がある．さらに，OpenACC では基本的に 1 プロセスから 1 つの演算加速機構へのオフロードにのみ対応しており，複数の演算加速機構を用いるためには acc_set_device() による切り替えが必要であるため，それを簡易に記述するための拡張を検討している．

謝辞　本研究に際しご協力いただ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*, available from ⟨http://www.nvidia.com/object/cuda_home_new.html⟩.

[2] Khronos Group: *OpenCL — The open standard for parallel programming of heterogeneous systems*, available from ⟨https://www.khronos.org/opencl/⟩.

[3] OpenACC-Standard.org: *OpenACC Home*, available from ⟨http://www.openacc.org⟩.

[4] OpenMP Architecture Review Board: *OpenMP Application Program Interface Version 4.0 - July 2013*, available from ⟨http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf⟩.

[5] XcalableMP Specification Working Group: *Xcalablemp specification version 1.2*, available from ⟨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, *Proc. 1st Workshop on Accelerator Programming Using Directives*, *WACCPD '14*, Piscataway, NJ, USA, pp.27–36 (2014).

[9] Zheng, Y., Iancu, C., Hargrove, P.H., Min, S.-J. and Yelick, K.: Extending Unified Parallel C for GPU Computing, *SIAM Conference on Parallel Processing for Scientific Computing* (*SIAMPP*) (2010).

[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] 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).

[12] Cunningham, D., Bordawekar, R. and Saraswat, V.: GPU Programming in a High Level Language: Compiling X10 to CUDA, *Proc. 2011 ACM SIGPLAN X10 Workshop*, *X10 '11*, New York, NY, USA, ACM, pp.1–10 (2011).

[13] 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).

[14] RIKEN AICS and University of Tsukuba: *Omni Compiler Project*, available from ⟨http://omni-compiler.org⟩.

[15] The Ohio State University: *MVAPICH*, available from ⟨http://mvapich.cse.ohio-state.edu/⟩.

[16] Murai, H. and Sato, M.: An Efficient Implementation of Stencil Communication for the XcalableMP PGAS Parallel Programming Language, *7th International Conference on PGAS Programming Models* (2013).

[17] Tabuchi, A., Nakao, M. and Sato, M.: A Source-to-Source OpenACC Compiler for CUDA, *Euro-Par Workshops*, pp.178–187 (2013).

[18] 理化学研究所情報基盤センター：姫野ベンチマーク，入手先 ⟨http://accc.riken.jp/2145.htm⟩.

[19] Nakao, M., Lee, J., Boku, T. and Sato, M.: XcalableMP Implementation and Performance of NAS Parallel Benchmarks, *Proc. 4th Conference on Partitioned*

Global Address Space Programming Model, *PGAS '10*, New York, NY, USA, ACM, pp.11:1–11:10 (2010).

田渕 晶大　（学生会員）

1991 年生．2013 年筑波大学情報学群情報科学類卒業．2015 年同大学大学院システム情報工学研究科コンピュータサイエンス専攻博士前期課程修了．修士（工学）．同年 4 月より同大学院システム情報工学研究科コンピュータサイエンス専攻博士後期課程在籍．並列プログラミング言語，アクセラレータ等に興味あり．

中尾 昌広　（正会員）

2005 年同志社大学大学院工学研究科博士前期課程修了．同年 NTT アドバンステクノロジ株式会社入社．2010 年同志社大学大学院工学研究科博士後期課程修了．筑波大学計算科学研究センター研究員，理化学研究所計算科学研究機構特別研究員を経て，2015 年から同機構研究員．ハイパフォーマンスコンピューティングの研究に従事．2015 年情報処理学会山下記念研究賞受賞．

村井 均　（正会員）

1996 年 3 月京都大学大学院工学研究科情報工学専攻修士課程修了．1996 年 4 月～2010 年 3 月 NEC．2010 年 3 月筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻博士課程修了．2010 年 4 月より理化学研究所．現在，同計算科学研究機構・プログラミング環境研究チームにて HPC 向けプログラミング環境の研究に従事するとともに，同エクサスケールコンピューティング開発プロジェクト・アーキテクチャ開発チームにてポスト「京」の開発に従事．

朴 泰祐　（正会員）

1960 年生．1984 年慶應義塾大学工学部電気工学科卒業．1990 年同大学大学院理工学研究科電気工学専攻後期博士課程修了．工学博士．1968 年慶應義塾大学理工学部物理学科助手．1992 年筑波大学電子･情報工学系講師, 1995 年同助教授，2004 年同大学大学院システム情報工学研究科助教授，2005 年同教授，現在に至る．超並列計算機アーキテクチャ，ハイパフォーマンスコンピューティング，クラスタコンピューティング，GPU コンピューティングに関する研究に従事．筑波大学計算科学研究センターにおいて，超並列計算機 CP-PACS，PACS-CS，HA-PACS 等の研究開発を行う．2002 年および 2003 年度情報処理学会論文賞，2011 年 ACM ゴードンベル賞，2012 年度情報処理学会山下記念研究賞各受賞．IEEECS，ACM 各会員．

佐藤 三久　（正会員）

1959 年生．1982 年東京大学理学部情報科学科卒業．1986 年同大学大学院理学系研究科博士課程中退．同年新技術事業団後藤磁束量子情報プロジェクトに参加．1991 年通商産業省電子技術総合研究所入所．1996 年新情報処理開発機構並列分散システムパフォーマンス研究室室長．2001 年から 2015 年まで筑波大学システム情報系教授．2007 年度より 2012 年度まで同大学計算科学研究センターセンタ長．2010 年より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リーダ．2014 年より同機構エクサスケールコンピューティング開発プロジェクト副プロジェクトリーダ．理学博士．並列処理アーキテクチャ，プログラミングモデルと言語およびコンパイラ，計算機性能評価技術等の研究に従事．IEEE，日本応用数理学会各会員．