Vol.2017-HPC-158 No.26
2017/3/10

アクセラレータクラスタ向けPGAS言語XcalableACCの片側通信機能の実装と評価

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]

概要：GPU や MIC のようなアクセラレータを搭載したクラスタが広く使用されている．そのようなクラスタでは，MPI とアクセラレータ用のプログラミングモデルを組み合わせてプログラムを記述する必要がある．OpenACC によってアクセラレータのプログラミングは簡易になっているが，未だに MPI の複雑な記述により生産性が低いという課題がある．そこで我々は XcalableACC (XACC) という Partitioned Global Address Space (PGAS) 言語 XcalableMP (XMP) と OpenACC を統合した新しいプログラミング言語を提案している．XMP の PGAS プログラミングと OpenACC のアクセラレータプログラミングに加えてアクセラレータ間通信に対応することで高い性能と生産性を目指す．XMP のプログラミングモデルの 1 つである local-view モデルでは Fortran 2008 から標準に取り入れられた coarray により通信を記述することができるため，それを XACC のアクセラレータ間通信でも使用するための記法を提案する．XACC のコンパイラを設計・実装し，Himeno benchmark と NAS Parallel Benchmarks CG (NPB-CG) を用いて性能と生産性の評価を行ったところ，XACC の local-view モデルによる記述では MPI + OpenACC と比較して，Himeno benchmark で 85%以上，NPB-CG で 97%以上の性能を達成した．加えて，coarray 通信をノンブロッキングにすることで，Himeno benchmark における性能は 89%以上にまで改善することがわかった．また生産性の点では，local-view モデルは配列代入文形式による直感的な通信の記述が可能であるため MPI よりも優れていると言える．

1. 序論

高性能計算の分野では計算性能を向上させるために GPU や MIC のようなアクセラレータを搭載したクラスタが普及している．そのアクセラレータのプログラミングモデルとして OpenACC の利用が広がりつつある．OpenACC はアクセラレータ向けの指示文ベースプログラミングモデルであり，従来の CUDA や OpenCL に比べて簡易にアクセラレータに処理をオフロードすることが可能である．一方で，分散メモリプログラミングには現在でも MPI が一般的である．プログラマはデータや処理の分散に加えて MPI 関数を用いて通信を記述しなければならないため，記述が複雑で生産性が低下するという課題がある．MPI の代わりとして，我々は Partitioned Global Address Space (PGAS) 言語である XcalableMP (XMP) [1] を提案してきた．XMP は 2 つのプログラミングモデルを提供しており，1 つは指示文でデータと処理の分散や通信を記述する global-view モデル，もう一つは coarray で片側通信を記述する local-view モデルである．

さらに我々は XMP と OpenACC を垂直統合した XcalableACC (XACC) [2,3] を提案している．XACC では XMP と OpenACC のそれぞれの機能に加えて，アクセラレータクラスタ向けにアクセラレータ間の通信も提供している．XACC の global-view モデルにおいて高い性能と生産性があることはすでに示されている．そこで本稿では，XACC の local-view モデルにおける coarray を用いたアクセラレータ間の片側通信の記法を提案する．その機能を Omni XMP compiler [4] を拡張した Omni XACC compiler で実装し，2 つのベンチマークを用いて性能と生産性を評価する．

本稿の貢献は以下の 3 点である．
- OpenACC でオフロードされたデータに対する coarray の記述を提案する．
- XMP/XACC coarray の同期を削減する指示文を提案する．
- MPI を通信に用いた XACC コンパイラを設計・実装

[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

© 2017 Information Processing Society of Japan

し，ベンチマークを用いて XACC local-view モデルの片側通信の性能を評価する．

本稿の構成は次の通りである．2 章で関連研究を紹介し，3 章と 4 章で XMP と XACC をコード例を交えて説明する．5 章では Omni XACC compiler の設計と実装を述べる．さらに，6 章で 2 つのベンチマークを用いて性能と生産性を評価する．最後に，7 章で結論と今後の課題を述べる．

2. 関連研究

XMP-dev という XMP のアクセラレータクラスタ向け拡張が提案され，GPU クラスタ向けに実装された [5, 6]．XMP-dev はオフロードに独自の指示文を用いていたが，XACC では OpenACC を用いることで他のプラットフォームへの可搬性を向上させ，既に OpenACC を知っているプログラマが利用しやすいようにしている．また XMP-dev では global-view モデルしか考慮されていなかったが，XACC では local-view モデルにも対応し柔軟なプログラミングを可能としている．

XACC の global-view モデルの実装と評価が行われている [2, 3]．文献 [3] では通信ライブラリに MPI を，文献 [2] では Tightly Coupled Accelerators (TCA) を用いて実装している．アクセラレータ間通信が記述できることによって特に TCA を用いた実装では Himeno benchmark で OpenACC + MPI/InfiniBand よりも高い性能を達成している．

他の PGAS 言語では，Unified Parallel C (UPC) と OpenSHMEM において GPU メモリの管理・アクセス用の拡張が提案されており，どちらも GPU メモリ間の通信をサポートしている [7, 8]．UPC は XACC と同様に言語を拡張しているが，UPC では通信が暗黙に行われることがあるのに対して，XACC では必ず明示しなければならないことにより意図しない通信の発生を防ぐことができる．UPC や OpenSHMEM では関数呼び出しにより他のノードにデータを Put/Get することが可能であるが，XACC では coarray を用いることでより簡易に通信が記述可能である．

CAFe はヘテロジニアス環境向けの Coarray Fortran の拡張である [9]．その言語では各 image が保持する *subimage* というものを導入している．その subimage 上でのタスクや並列ループがアクセラレータにオフロードされる．CAFe は高レベルの記述によって，MPI + OpenACC のようなハイブリッドの記述よりもコンパイラ最適化が可能である．しかしながら，ある image はそれが保持する subimage と coarray により通信を行うことが可能であるが，異なる image が持つ subimage 間での通信には対応していない．XACC は異なるノードのアクセラレータ間の通信が可能であり，またアクセラレータプログラミングを OpenACC にすることで，プログラマがコード最適化を行う余地を残している．

文献 [10] では coarray をアクセラレータ向けに拡張している．“accelerated” 属性という，アクセラレータでの計算に適しており，ホストとアクセラレータの両方からアクセス可能であることを示す属性を導入している．したがって，この属性が付加された coarray に対する操作は同時にアクセラレータ側でも参照可能である．この機能は CUDA の managed memory というホストと GPU の両方からアクセス可能で CUDA 側でコヒーレントをとるメモリを利用して実装されている．XACC ではホストとアクセラレータのメモリは明確に区別することで，この手法よりも通信が明確になるという利点がある．

文献 [11] では Himeno benchmark を Fortran の coarray と OpenACC を用いて実装している．その文献ではデバイス間の通信は coarray によるホスト間の通信と OpenACC の `updata` 指示文により実装されており，本稿の評価ではデバイスメモリ間の通信を指示文で直接記述している．

3. XcalableMP

XcalableMP [1] は PC クラスタコンソーシアム XMP 規格部会 [12] が策定している PGAS 言語である．XMP は分散メモリシステム向け並列プログラミングのために C や Fortran を拡張しており，global-view モデルと local-view 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．この章では global-view モデルの概略と local-view モデルの詳細を述べる．

3.1 Global-view モデル

global-view モデルは High Performance Fortran (HPF) [13] に似た指示文ベースのプログラミングモデルである．逐次コードに対して指示文を追加することにより，分散メモリシステム向けのプログラムを記述することができる．HPF ではデータ分散の指示文から処理系が処理の分散や通信を自動的に生成するのに対して，XMP ではユーザがデータと処理の分散と通信をすべて指示文で指定する．それにより HPF よりもコードチューニングがしやすいという特徴がある．**図 1** に global-view モデルのプログラム例を示す．global-view モデルにおいて，ノードは XMP の実行単位であり，MPI のプロセスに相当するものである．`nodes` 指示文はそのノード集合を定義する．テンプレートは仮想インデックス空間であり，`template` 指示文により定義する．`distribute` 指示文はテンプレートをノード集合上にどのように分散するかを指定する．データや処理の分散はテンプレートに対応づけることで記述する．通信には `bcast`, `reduction`, `reflect`, `gmove` 指示文が用意されている．`bcast` 指示文はデータをブロードキャストし，`reduction` 指示文はデータを指定された演算子で縮約する．`reflect` 指示文は主にステンシル計算で用いられる通信である．分散配列には `shadow` 指示文で袖領域を定義す

© 2017 Information Processing Society of Japan

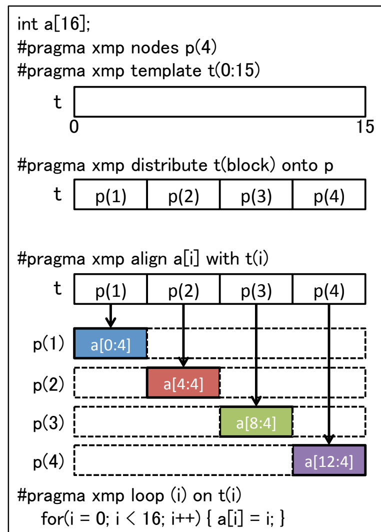


図 1: XMP global-view モデルプログラミングの例

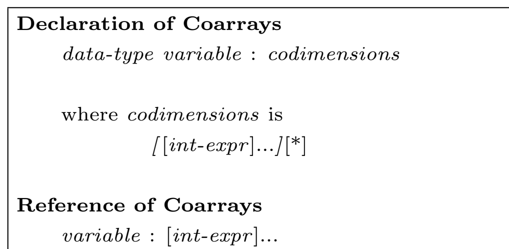


図 2: XMP/C における Coarray のシンタックス

ることができ，`reflect` 指示文は袖のデータを隣接ノードの値で更新する．`gmove` 指示文は変数や分散配列に対してグローバルな代入処理を行うための指示文である．

3.2 Local-view モデル

local-view モデルは各ノードのローカルデータに対して通信する機能を提供している．XMP では Fotran 2008 から採用されている Coarray Fortran [14] を local-view モデルでの通信に用いている．MPI と同様にユーザがデータや処理の分散を行う必要があるが，coarray 記法により簡易に通信を記述できる．XMP Fortran は Fortran 2008 との互換性があり，XMP C (XMP/C) は C の文法を拡張することで coarray に対応している．本稿では主に XMP/C の coarray を用いるため，その拡張記法について説明する．

3.2.1 XMP/C における Coarray 記法

XMP/C における coarray のシンタックスを図 **2** に示す．coarray は通常の変数宣言の最後に *codimension* を追加することで宣言できる．coarray は *image* 集合 (XMP におけるノード集合と等価) 上に宣言することが可能で，codimension は image 集合の形状を指定する．他のノードの coarray を参照する際には，coindex による対象 image の指定が必要であり，coindex がなければそれはローカルの参照となる．リモートの coarray に対する代入は Put となり，逆にリモートの coarray に対する参照は Get となる．また XMP/C では [*lower*[:*length*[:*stride*]]] という形式で配列の範囲指定を可能にしている．

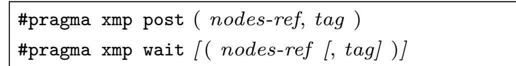


図 3: post/wait 指示文のシンタックス

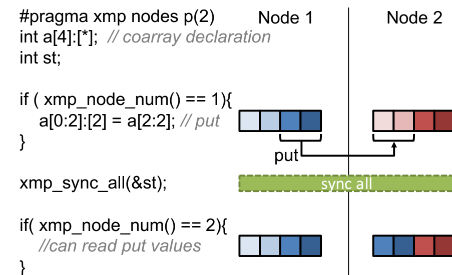


図 4: XMP local-view モデルプログラミングの例

3.2.2 XMP/C における Coarray の同期

XMP は coarray の同期のためにいくつかの関数を提供している．まず Fortran 2008 のように，xmp_sync_memory(), xmp_sync_image(), xmp_sync_images(), xmp_sync_all() が関数として利用できる．xmp_sync_memory() は先行する coarray 通信の完了を保証する．xmp_sync_image(), xmp_sync_images() は実行した image と指定した image において先行する通信の完了を保証する．xmp_sync_all() はすべての image に対して先行する通信の完了を保証する．

XMP/C にはさらにシグナルを送受信するための `post/wait` 指示文が用意されている．図 **3** は `post/wait` 指示文のシンタックスである．*nodes-ref* は global-view モデルにおけるノードを指定する．また *tag* を指定することで特定の post と wait を関連づけることが可能である．xmp_sync_memory() と `post/wait` の組み合わせによって xmp_sync_image() 等と同じように通信の同期を行うことが可能である．

3.2.3 XMP/C におけるコード例

図 **4** に XMP/C における local-view プログラミングの例を示す．*a[]* は codimension の指定により coarray として宣言されている．リモートの coarray への代入文により，ノード 1 の coarray *a[]* の 2–3 要素目をノード 2 の coarray *a[]* の 0–1 要素目に Put する．xmp_sync_all() 関数によってすべてのノードの同期がとられ，その後にノード 2 で Put された値を読むことが可能になる．

4. XcalableACC

XcalableACC (XACC) は XMP と OpenACC を統合したアクセラレータ向けの PGAS 言語である．XMP と同様に C と Fortran をベース言語としてサポートしている．XACC の主な追加機能はアクセラレータ間の通信である．

© 2017 Information Processing Society of Japan

```
1  int a[N]:[*]; /* coarray */
2  int b[N]; /* array */
3  #pragma acc declare create(a, b)
4
5  /* ... */
6
7  if(node_num == 1){
8      #pragma acc parallel loop
9      for(int i = 0; i < N; i++){
10         b[i] = i;
11     }
12     #pragma acc host_data use_device(a, b)
13         a[:]:[2] = b[:];
14 }
15
16 xmp_sync_all(&status);
17
18 if(node_num == 2){
19     #pragma acc parallel loop
20     for(int i = 0; i < N; i++){
21         b[i] = a[i] * 2;
22     }
23 }
```

図 5: XACC の coarray の例

デバイス上のデータを他のノードのデバイスと通信する際に，XMP ではホストメモリ上のデータに対する通信しかできないため，単に XMP と OpenACC を組み合わせた場合には OpenACC によるホストとデバイス間のコピーと XMP によるホスト間通信の記述が必要となる．XACC の global-view モデルでは XMP の通信指示文の最後に "acc" を付記することによりデバイス上のデータに対して通信を行うことが可能である．

local-view モデルの coarray に関しては Fortran の仕様として取り入れられているため，本来は OpenACC においてデバイス上の coarray の確保や通信の仕方を定めるのが望ましい．しかしながら，最新の OpenACC 2.5 においても coarray に関する記述は見られないため，XACC ではその記述を提案する．coarray の確保は OpenACC の `declare` 指示文で行う．デバイスメモリを確保する方法としては他に `data`，`enter data`，`exit data` 指示文が存在するが，RMA 通信を行えるデバイスメモリには制約がある可能性があるため，静的に確保が可能な `declare` 指示文のみを許すことにした．デバイス上の coarray に対する通信を行うには OpenACC の `host_data` 指示文の `use_device` 節で対象の coarray を指定する．通信の同期は通常の coarray と同じ関数により行う．

XACC の coarray を用いたコード例を図 **5** に示す．1 行目で coarray *a[]* を宣言する．これは XMP の coarray の宣言と同じである．3 行目で `declare` 指示文により coarray *a[]* と配列 *b[]* をデバイスメモリ上に確保するよう宣言する．12–13 行目でノード 1 のデバイスメモリ上の配列 *b[]* 全体をノード 1 のデバイスメモリ上の coarray *a[]* に Put している．16 行目では xmp_sync_all() によりすべてのノードの通信の完了を保証する．

5. Omni XcalableACC Compiler

この章では Omni XcalableACC compiler の設計と実装について述べる．

5.1 設計

Omni XACC compiler は source-to-source コンパイラとし，XACC のコードを OpenACC と XACC のライブラリ呼び出しに変換する．global-view モデルに対しては，分散する配列とループを変形し，XACC 指示文を対応する XACC ライブラリ呼び出しに置き換える．local-view モデルに対しては，coarray を配列に戻した上で Put/Get を XACC のライブラリ呼び出しに置き換える．これら 2 つのモデルに対する変換は基本的にそれぞれ独立である．OpenACC 指示文はコード変換での変数名の変更や coarray 確保によって若干書き換えるが，ほぼ元のコードと同じ指示文になる．生成されたコードは通常の OpenACC コードであるため，どの OpenACC コンパイラでもコンパイル可能である．この設計にはコード変換が単純になり可搬性が向上するというメリットがある．

5.2 実装

このコード変換を実現するため，理研 AICS と筑波大 HPCS 研が開発中の XMP リファレンスコンパイラである Omni XMP compiler [4] をベースに Omni XACC compiler を開発した．コンパイルの流れを図 **6** に示す．XACC コードは XACC translator により XACC ランタイム呼び出しを含む OpenACC コードに変換される．その OpenACC コードは OpenACC コンパイラ (Omni OpenACC, PGI, Cray コンパイラ等) でコンパイルされ，最後に Omni XACC ランタイムライブラリをリンクする．現在の実装では，C では global-view モデルと local-view モデルの両方に，Fortran では global-view モデルのみ対応している．XACC ランタイムライブラリは XMP および XACC のランタイムルーチンを含んでいる．XACC 用のデバイス間通信は NVIDIA GPU を対象として MPI と CUDA によって実装されており，特に coarray の実装には MPI 3.0 以降の片側通信機能を利用している．現在の実装では利用する MPI が送受信バッファに GPU メモリのポインタを指定可能な CUDA-aware である必要がある．MVAPICH2 [15] や OpenMPI [16] などがこの機能に対応している．

5.2.1 Coarray 用の初期化・終了時処理

MPI では *window* を通してリモートのメモリにアクセスする．すべての coarray は集団的に同期が取られるため，このコンパイラではすべての coarray をホストメモリとデバイスメモリ用の 2 つの window で管理する．XACC プログラムの初期化時には malloc() や cudaMalloc() によっ

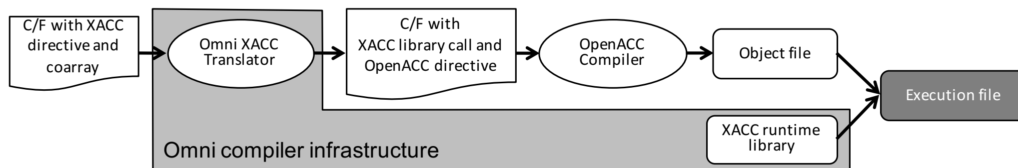


図 6: Omni XACC compiler のコンパイルの流れ

```
int A[64]:[*];
#pragma acc declare create(A)
```

(a) デバイスメモリ上の coarray 宣言

```
void * _XMP_COARRAY_DESC_A;
int * _XMP_COARRAY_ADDR_A;
int * _XMP_COARRAY_ADDR_DEV_A;

extern void xmpc_traverse_init_file_sample_pp()
{
 _XMP_coarray_malloc_info_1(0x40, sizeof(int));
 _XMP_coarray_malloc_image_info_1();
 _XMP_coarray_malloc_do(&(_XMP_COARRAY_DESC_A),
                          &(_XMP_COARRAY_ADDR_A));
 _XMP_coarray_malloc_do_acc(&(_XMP_COARRAY_DESC_A),
                          &(_XMP_COARRAY_ADDR_DEV_A));
 acc_map_data(_XMP_COARRAY_ADDR_A,_XMP_COARRAY_ADDR_DEV_A,
       _XMP_coarray_get_total_elmts(_XMP_COARRAY_DESC_A)
                                        *sizeof(int));
}
```

(b) 変換したコード

図 7: デバイスメモリ上の coarray 宣言のコード変換

て coarray 用のヒープを確保し，それらに関連付けた window を作成する．このヒープサイズは環境変数により指定可能である．MPI RMA は window の access epoch 内で行う必要があるため，プログラム初期化時に MPI_Win_lock_all() により access epoch を開始し，プログラム終了時に MPI_Win_unlock_all() により終了する．

5.2.2 Coarray の宣言

coarray の宣言は XACC のランタイム呼び出しに変換される．現在の実装では各 coarray は単純にヒープの先頭から順に確保される．また OpenACC の `declare` 指示文で指定された coarray に対してはデバイスメモリ側にもメモリが確保される．**図 7** は coarray 宣言のコード変換を示す．coarray *A[]* は `declare` 指示文で指定されているため，ホストとデバイスの両方に確保する必要がある．変数 _XMP_COARRAY_DESC_A は coarray の情報 (ホストアドレス，デバイスアドレス，形状等) を持つ構造体へのポインタである．変数 _XMP_COARRAY_ADDR_A と _XMP_COARRAY_ADDR_DEV_A は coarray へのローカルアクセスのためのポインタである．関数 xmpc_traverse_init_file...() は翻訳単位の初期化関数である．この関数では coarray のサイズや image の情報を設定した後，関数 _XMP_coarray_malloc_do() によりホストメモリを，関数 _XMP_coarray_malloc_do_acc() によりデバイスメモリを確

```
#pragma xmp nonblock [ ( remote ) ]
structured-block
#pragma xmp wait_nonblock [ ( local ) ]
```

図 8: nonblock, wait_nonblock 指示文のシンタックス

保する．最後に OpenACC の関数 acc_map_data() により coarray のホストメモリとデバイスメモリを対応付ける．

5.2.3 Coarray の通信

coarray の Put/Get 操作には MPI_Put/Get() がそれぞれ使われる．通信に使われるローカル側の配列もしくは coarray が `use_device` 節で指定されていたらデバイスのポインタを指定し，それ以外はホストのポインタを引数に指定する．また，リモート側の coarray が `use_device` 節で指定されていたらデバイス用の window を，そうでなければホスト用の window を引数に用いる．MPI_Put/Get() はノンブロッキング関数であるので，その完了を保証するために直後に MPI_Win_flush() を呼び出す．しかしながら，この実装には性能的に問題がある．例えば複数の Put/Get を連続して行う場合，それらに依存がない場合でも毎回同期を行うことで同時に通信を行うことができなくなる．このような最適化は処理系がコンパイル時・実行時に解析して行うべきである．文献 [17] では UPC のランタイムにおいてリモートへの書き込み・読み込みを記録することで自動的に RMA をノンブロッキングにしている．この手法ではリモート側のデータの依存は解析可能であるが，ローカル側は解析できないので必ずローカルではブロッキングとなる．このローカルの依存解析は難しいため，指示文を用いることが考えられる．例えば，Cray compiler では `defer_sync` 指示文により可能な限り同期を遅らせるよう指定することができる [18]．ところがこの機能の仕様は明確に定められていないため，我々は新たな指示文 `nonblock`, `wait_nonblock` を提案する．**図 8** にそのシンタックスを示す．`nonblock` 指示文は対象コード内の coarray 通信の完了を保証しなくても良いことをコンパイラに伝える．引数がない場合はローカルとリモート共に通信の完了を保証しなくてもよく，引数に remote が指定された場合はリモートのみ完了を保証しなくても良い．完了保証がされなかった通信は，次に出会う同期関数 (xmp_sync_...()) または `wait_nonblock` 指示文で完了を待つことができる．`wait_nonblock` 指示文は先行するノンブロッキングとなった通信の完了を保証する．引数がない場合はローカルとリ

```
//Array B on device is assigned to coarray A on device
#pragma acc host_data use_device(A, B)
A[:]:[2] = B[:];
```

(a) デバイスメモリ上の coarray への Put

```
unsigned long long _ACC_size_A
       = _XMP_coarray_get_total_elmts(_XMP_COARRAY_DESC_A);
#pragma acc host_data \
          use_device(_XMP_COARRAY_ADDR_A[0:_ACC_size_A], B)
{
_XMP_coarray_rdma_coarray_set_1(0, 64, 1);
_XMP_coarray_rdma_array_set_1(0, 64, 1, 64, sizeof(int));
_XMP_coarray_rdma_image_set_1(2);
_XMP_coarray_rdma_do_acc(701,_XMP_COARRAY_DESC_A,B,NULL,
                                                     1,1);
}
```

(b) 変換したコード

図 9: デバイスメモリ上の coarray 通信のコード変換

モートの完了を保証し，引数に local が指定された場合はローカルの完了のみを保証する．現在はまだこれらの指示文が実装できていないため，性能評価の際には簡易的に環境変数によって MPI_Put/Get() 直後の MPI_Win_flush() を行わないようにすることでノンブロッキングの効果を確認する．

図 9 にデバイス上の coarray の通信のコード変換を示す．この例では，coarray *A[]* と 配列 *B[]* は `use_device` 節で指定されているため，次の Put はデバイスメモリ間の通信となる．いくつかの関数で coarray のサイズ，image 番号を設定した後，関数 _XMP_coarray_rdma_do_acc() によりデバイスメモリ上の配列 *B[]* の全体を image 2 のデバイスメモリ上の *A[]* に Put する．

5.2.4 Coarray の同期

関数 xmp_sync_memory() では MPI_Win_sync() を用いて window を同期し，さらに最適化による sync_memory をまたいだ命令入れ替えを防ぐ．また Put/Get をノンブロッキングにした場合には MPI_Win_flush_all() により通信の完了を待機する．関数 xmp_sync_image() は xmp_sync_memory() と MPI_Send/Recv() により，関数 xmp_sync_all() は xmp_sync_memory() と MPI_Barrier() により実装されている．

6. 評価

6.1 ベンチマーク

評価には，Himeno benchmark [19] と NAS Parallel Benchmarks CG (NPB-CG) [20] を用いた．Himeno benchmark は非圧縮性流体解析コードの性能評価用ベンチマークである．主な演算はポアソン方程式をヤコビ法で解く際の 3 次元 19 点ステンシル計算である．NPB-CG は正定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．比較したコードは，Send/Recv を用いた MPI + OpenACC，Get を用いた MPI + OpenACC，XACC global-view モデル，Get を用いた XACC local-view モデル の 4 種類である．元の NPB-CG は Fortran で記述されているが，現在の Omni XACC compiler が Fortran coarray に対応していないため，C で書かれたものを使用した．

表 1: HA-PACS/TCA のノード構成

CPU	Intel Xeon-E5 2680v2 2.8GHz × 2 Socket
Memory	DDR3 1866MHz × 4 channel, 128GB
GPU	NVIDIA Tesla K20X × 4 (GDDR5 6GB)
Interconnect	InfiniBand: Mellanox Connect-X3 Dual-port QDR
Compiler	GCC 4.4.7, CUDA 7.5 MVAPICH2-GDR 2.2rc1 Omni XACC Compiler 1.0.3 相当

6.2 性能

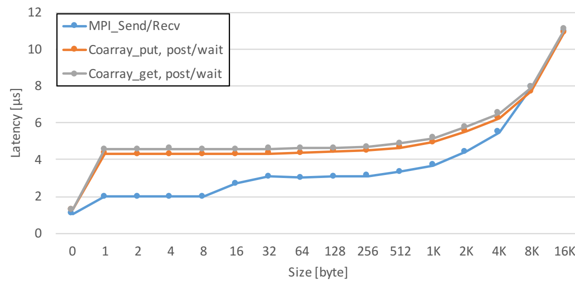
性能評価には筑波大学計算科学研究センターの HA-PACS/TCA [21] を用いた．そのノード構成を**表 1** に示す．なおこの評価では TCA 機構は使用せず，MPI を用いて IB で通信を行う．また 1 ノードあたり 4 台の GPU が搭載されているが，QPI をまたぐ通信を避けるために 2 台のみ使用している．MPI には MVAPICH2-GDR を用い，GPUDirect for RDMA (GDR) [22] や GDRCopy [23] を利用するよう設定した．OpenACC コンパイラには，我々が NVIDIA GPU 向けに開発した Omni OpenACC compiler [24] を用いた．

予備評価として coarray の性能を Ping-Pong ベンチマークで測定した結果を**図 10** に示す．coarray の実装では xmp_sync_memory() と `post/wait` 指示文により同期を行っている．データサイズが 8KB 未満の場合には MPI_Send/Recv よりもレイテンシが大きいが，それより大きなサイズでは性能はほぼ同じである．今回ベンチマークで用いたサイズやプロセス数では RMA で通信するデータサイズは 4KB 以上であるため，coarray と Send/Recv の性能差はそれほど現れないと考えられる．またベンチマークのいくつかのケースで Put だと MVAPICH2 のランタイムがエラーを出す場合があったので，RMA では Get を用いて通信を行う．

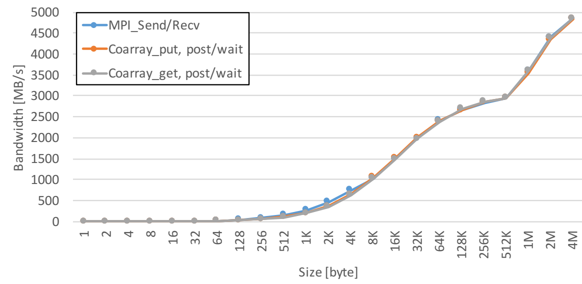
6.2.1 Himeno Benchmark

問題サイズには Medium $(i \times j \times k) = (128 \times 128 \times 256)$ と Large $(256 \times 256 \times 512)$ を用いた．領域分割では k 次元を分割すると袖領域が完全に不連続となってしまい pack-/unpack のコストが高くなるため，i 次元と j 次元の 2 次元分割とした．元々の Himeno benchmark では袖領域の送受信に MPI_Type_vector で作成した派生型を用いていたが，これを jk 平面に対してはパディングも含めることで，ik 平面に対しては pack/unpack を行うことで連続領域と

© 2017 Information Processing Society of Japan



(a) Latency



(b) Bandwidth

図 10: Ping-Pong ベンチマークの性能

```
#pragma xmp nodes n(1, NDY, NDX)
#pragma xmp template t(0:MKMAX-1, 0:MJMAX-1, 0:MIMAX-1)
#pragma xmp distribute t(block, block, block) onto n

float p[MIMAX][MJMAX][MKMAX], ...;
#pragma xmp align p[k][j][i] with t(i, j, k)
#pragma xmp shadow p[1:2][1:2][0:1]

/* ... */

/* halo exchange (ik-plane and jk-plane) */
#pragma xmp reflect(p) width(1,1,0) acc
```

図 11: Himeno benchmark XACC global-view 版の袖交換

して送受信するようにした．

XACC global-view 版の袖領域の通信を**図 11** に示す．配列 *p[]* は `align` 指示文により分散されており，かつ `shadow` 指示文により袖領域が追加されている [*1]．その袖領域は `reflect` 指示文により最新の値に更新される．

XACC local-view 版の袖領域の通信を**図 12** に示す．この図では pack/unpack が必要な *ik* 平面に関する部分のみ抜き出している．まず袖のデータを送信バッファにパックする．その後，xmp_sync_images() によって隣接プロセスと同期した後に coarray の Get を用いて隣接プロセスの送信バッファのデータを受信バッファに転送する．そして再度 xmp_sync_images() により通信完了を待ってから，受信バッファのデータをアンパックする．

Himeno benchmark の性能と 1 反復の時間内訳を**図 13** と**図 14** に示す．図の凡例，“MPI+ACC (s/r)”，“MPI+ACC (get)”，“XACC-G”，“XACC-L”，“XACC-L (nb)” はそれぞれ Send/Recv を用いた MPI + OpenACC，Get を用いた MPI + OpenACC，XACC global-view，Get を用いた XACC local-view，ノンブロッキングの Get を用いた XACC local-view，をそれぞれ示す．まず，XACC global-view 版の性能は安定して高く，MPI + OpenACC Send/Recv 版と比較してサイズ M で 97%以上，サイズ L で 98%以上の性能を達成している．次に，XACC local-view 版では プ

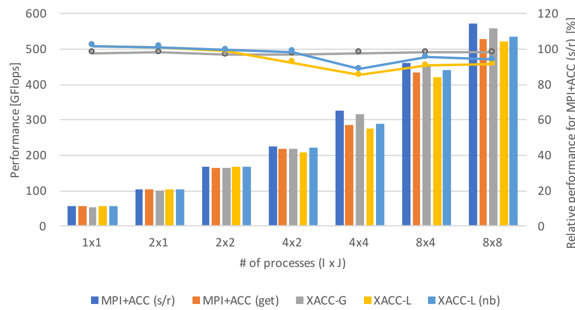
[*1] 本来，shadow 指示文による袖の指定は [1:1][1:1][0:0] で良いが，ここではパディングのために各次元に 1 要素追加している

```
float p[MIMAX][MJMAX][MKMAX]:[NDZ0][NDY0][*];
float sendp2_lo_sendbuf[MIMAX*MKMAX]:[NDZ0][NDY0][*];
float sendp2_lo_recvbuf[MIMAX*MKMAX]:[NDZ0][NDY0][*];
float sendp2_hi_sendbuf[MIMAX*MKMAX]:[NDZ0][NDY0][*];
float sendp2_hi_recvbuf[MIMAX*MKMAX]:[NDZ0][NDY0][*];
#pragma acc declare create(p, sendp2_lo_sendbuf, \
   sendp2_lo_recvbuf,sendp2_hi_sendbuf, sendp2_hi_recvbuf)

/* ... */

sendp2_pack();

#pragma acc host_data use_device(sendp2_lo_sendbuf,\
   sendp2_lo_recvbuf, sendp2_hi_sendbuf, sendp2_hi_recvbuf)
{
  int len = imax*kmax;
  xmp_sync_images(num_npy, npy, NULL);

  if(mey > 1){
    sendp2_lo_recvbuf[0:len]
      = sendp2_lo_sendbuf[0:len]:[mez][mey-1][mex];
  }
  if(mey < ndy){
    sendp2_hi_recvbuf[0:len]
      = sendp2_hi_sendbuf[0:len]:[mez][mey+1][mex];
  }

  xmp_sync_images(num_npy, npy, NULL);
}

sendp2_unpack();
```

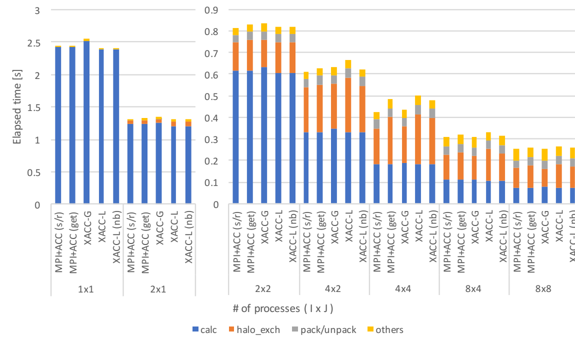
図 12: Himeno benchmark XACC local-view 版の袖交換 (*ik* 平面のみ)

ロセス数が 1×1 と 2×1 において，MPI + OpenACC Send/Recv 版よりも 1–2%高くなっている．これは XACC コンパイラが多次元の coarray を 1 次元の配列に変換したことによって計算時間が減少したからである．プロセス数が 4×2 から 8×8 の場合には，袖領域の通信時間の増加により性能が低下しており，MPI+OpenACC Send/Recv 版に対する性能はサイズ M で 85%，サイズ L で 92%まで低下している．袖通信は各次元の通信が順に行われるため，最大で 2 つの隣接プロセスとの通信を同時に行うことができる．しかしながら，5.2.3 章で触れた通り現在の実装では coarray がブロッキング通信であるため，同時に

© 2017 Information Processing Society of Japan

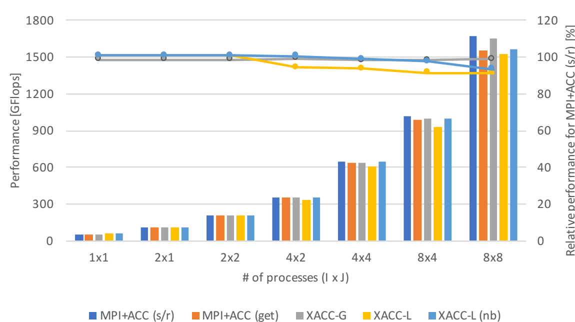


(a) 性能

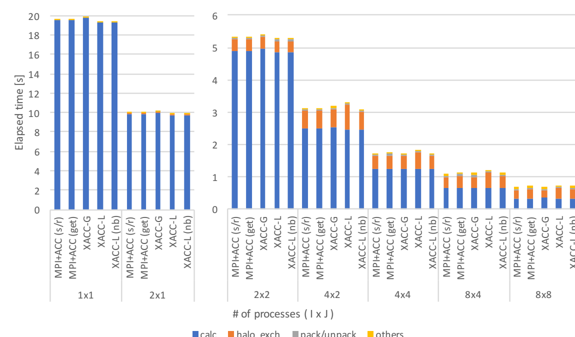


(b) 1 反復の時間内訳

図 13: Himeno benchmark の性能と時間内訳 (Size M)



(a) 性能



(b) 1 反復の時間内訳

図 14: Himeno benchmark の性能と時間内訳 (Size L)

1 つの隣接プロセスとしか通信ができない．環境変数を設定して coarray をノンブロッキングにした場合には，袖通信時間が短縮されて相対性能がサイズ M では 89%，サイズ L では 93%以上に改善した．それでもなお MPI+ACC Send/Recv 版よりも性能が低い部分があるが，MPI+ACC Get 版も同様に性能が下がっているため，Send/Recv と Get 自体の性能差によるものと言える．

6.2.2 NPB-CG

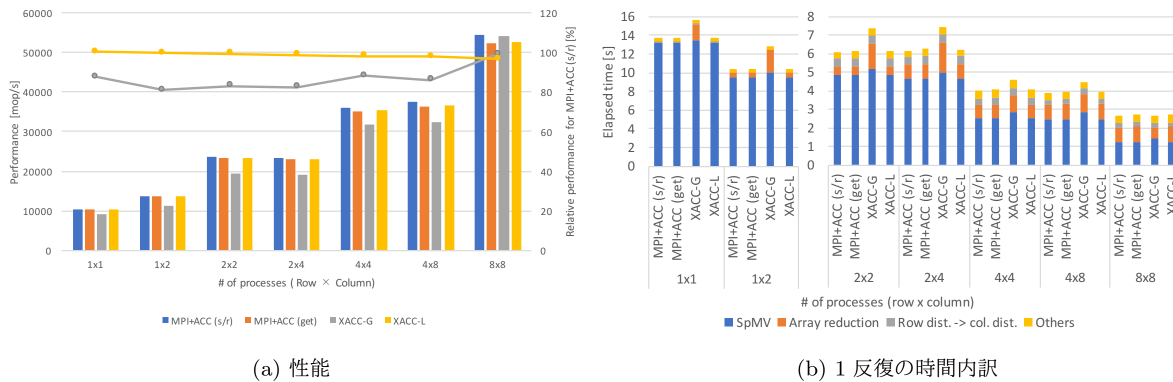
問題サイズには Class C (行列サイズが $750,000 \times 750,000$) と Class D ($1,500,000 \times 1,500,000$) を用いた．XACC global-view 版の通信部分を図 **15** に示す．配列 *w[]* は 2 次元ノードの行方向で分散されており，配列 *q[]* は列方向で分散されている．1 つ目の通信は `reduction` 指示文により，分散配列 *w[]* を行の部分ノード集合でリダクションしている．2 つ目の通信は `gmove` 指示文により，行分散の *w[]* を列分散の *q[]* に代入している．

XACC local-view 版の通信部分を図 **16** に示す．配列 *w[]* のリダクションは，Get と加算のカーネルを用いて Recursive-doubling 法により行っている．また行分散の *w[]* から列分散の *q[]* への代入も Get で記述している．なお，問題サイズ Class D においては GPU のメモリ量の制約により 1 プロセスで実行できなかったため，2–64 プロセスでの性能を計測した．また，XACC local-view 版においては coarray 通信は 1 プロセスと通信をして直後に同期をするパターンのみであるため，ノンブロッキングにすることによる効果はほぼないと考えられるので，ノンブロッキング版の計測は省いている．

```
#pragma xmp nodes proc(NUM_PROC_COLS, NUM_PROC_ROWS)
#pragma xmp nodes subproc(NUM_PROC_COLS) = proc(:,*)
#pragma xmp template t(0:NA-1, 0:NA-1)
#pragma xmp distribute t(block, block) onto proc
double w[NA], q[NA], ...;
#pragma xmp align w[i] with t(*,i)
#pragma xmp align q[i] with t(i,*)

/* ... */

/* array reduction */
#pragma xmp reduction(+:w) on subproc(:) acc

/* column-distributed array <- row-distributed array */
#pragma xmp gmove acc
  q[:] = w[:];
```

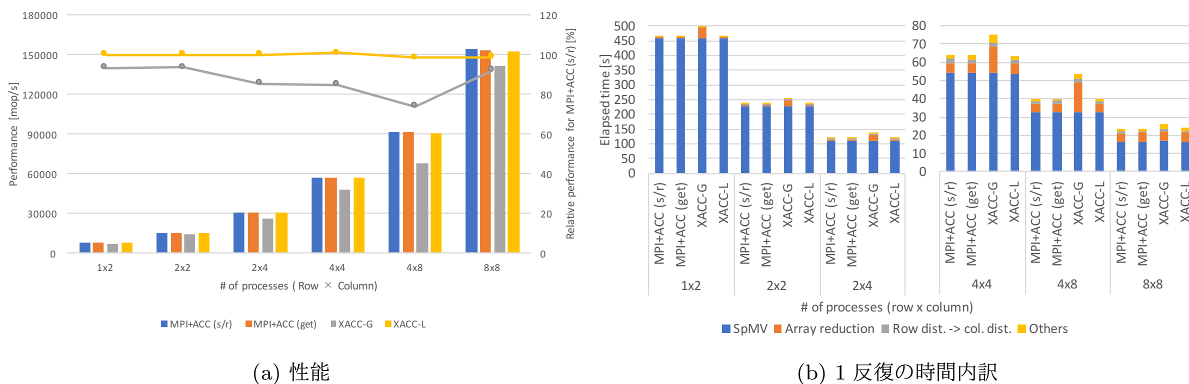
図 15: NPB-CG XACC global-view 版の主な通信

NPB-CG ベンチマークの性能と 1 反復の時間内訳を図 **17** と 図 **18** に示す．性能は 1 秒当たりの演算数を表す “mop/s” で示されている．XACC local-view 版の性能



(a) 性能　　(b) 1 反復の時間内訳

図 17: NPB-CG の性能と時間内訳 (Class C)



(a) 性能　　(b) 1 反復の時間内訳

図 18: NPB-CG の性能と時間内訳 (Class D)

は MPI + OpenACC Send/Recv 版に対して Class C では 97%以上，Class D では 99%以上と安定して高い性能を達成している．Class C の 8 × 8 プロセスにおいて若干性能が落ちているものの，MPI + OpenACC Get 版も同様に下がっているので通信方法の違いによるものと言える．Himeno benchmark で現れたような大きな性能低下が見られなかったのは NPB-CG では同時に 1 つの通信しか行わないためにブロッキングでも問題がなかったからである．

一方で，XACC global-view 版では全体的に local-view 版よりも性能が低い．性能低下は SpMV 計算と配列リダクションの時間の増加が原因である．SpMV の時間増加は XACC コンパイラによるループ書き換えが原因であると考えられるが，詳細は調査中である．配列リダクションの時間増加には 2 つの原因がある．1 つ目は使用する通信方法の違いで，global-view 版では CPU で加算を行う MPI_Allreduce() を用いているのに対して，その他のコードでは MPI_Isend/Recv() または MPI_Get() と，OpenACC カーネルにより GPU 上で加算を行う．2 つ目はリダクションする配列の長さの違いである．行プロセス数と列プロセス数が異なる場合，その次の *w[]* から *q[]* への代入で必要になる *w[]* はその前半分か後ろ半分のどちらかである．XACC global-view 版ではすべての場合で *w[]* の全体をリダクションするため特に，2×4 と 4×8 プロセスで遅くなっている．

6.3　生産性

MPI ではプログラマは MPI の関数にバッファ，要素型，要素数，タグ，ステータスなど非常に多くの引数を渡して通信を行わなければならない．また，多次元配列の一部を送る場合には pack/unpack や派生型を用いて送る必要がある．このような複雑さはプログラミングが難しくなるだけでなくバグを埋め込む原因にもなる．対照的に，XACC local-view モデルでは配列代入文形式で通信を記述できるため，プログラマは coarray とその範囲及び対象とする image 番号に気をつけるだけでよい．また，XMP + OpenACC ではデバイス間の通信を OpenACC によるホストとデバイス間の通信と XMP によるホスト間の通信により記述する必要があるが，XACC では直接記述が可能である．実際に，図 12 と 図 16 では coarray により非常に簡易に通信が記述できている．

生産性を定量的に測るために Source Lines Of Code (SLOC) を数えた結果を**表 2** に示す．XACC global-view 版はデータと処理の分散や通信を指示文で記述できたため MPI + OpenACC よりも非常に少ない行数になっている．

```
double w[na/num_proc_rows+2]:[*];
double q[na/num_proc_rows+2]:[*];
#pragma acc declare create(w,q)

/* ... */

#pragma acc host_data use_device(w, q)
{
  /* array reduction */
  for(i = l2npcols; i >= 1; i--){
    int image = reduce_exch_proc[i-1] + 1;
    int start = reduce_recv_starts[i-1] - 1;
    int length = reduce_recv_lengths[i-1];
    xmp_sync_image(image, NULL);
    q[start:length] = w[start:length]:[image];
    xmp_sync_image(image, NULL);

#pragma acc parallel loop
    for(j = send_start-1; j < send_start+lengths-1; j++){
      w[j] = w[j] + q[j];
    }
  }

  /* column-distributed array <- row-distributed array */
  if( l2npcols != 0 ) {
    xmp_sync_image(exch_proc+1, NULL);
    q[0:send_len] = w[send_start-1:send_len]:[exch_proc+1];
    xmp_sync_image(exch_proc+1, NULL);
  }else{
#pragma acc parallel loop
    for(j=0; j < exch_recv_length; j++){
      q[j] = w[j];
    }
  }
}
```

図 16: NPB-CG XACC local-view 版の主な通信

表 2: Himeno benchmark と NPB-CG の SLOC

	Himeno benchmark	NPB-CG
逐次版	146	444
MPI+OpenACC	398	769
XACC global-view	198	609
XACC local-view	395	768

一方 local-view 版は MPI と同じようにデータと処理の分散と通信をプログラマが記述するため，MPI+OpenACC と行数の差はほぼない．

6.4 考察

性能と生産性の評価から，global-view モデルは指示文で典型的な分割を行うプログラムを記述でき，Himeno benchmark の袖通信のような典型的な通信であれば十分な性能を得られるが，NPB-CG のような複雑な通信が必要なプログラムでは性能低下がおこりうる．一方で，local-view モデルは coarray により複雑なデータ・処理分散と通信を記述することが可能であるが，コード行数は MPI を用いた場合と同程度になるため global-view モデルよりは生産性は低い．

したがって，global-view モデルは典型的な分割や通信を行うプログラムをシンプルに記述場合に適している．対照的に，local-view モデルは複雑な分割や通信のプログラムもしくは性能が重要なプログラムに対して適している．また XMP では global-view モデルと local-view モデルを組み合わせて記述する hybrid-view モデルも提案されており，性能と生産性を両立している [25]．この手法は XACC でも有効であると考えられる．

7. 結論

本稿ではアクセラレータ向けの PGAS 言語 XACC の local-view モデルにおけるデバイスメモリ上の coarray 通信の記法を提案し，それを Omni XMP compiler をベースとした Omni XACC compiler に実装した．評価では，MPI + OpenACC と比較して XACC local-view モデルは Himeno benchmark で 85%以上，NPB-CG で 97%以上の性能を達成した．加えて，coarray の通信をノンブロッキングにすることで，Himeno benchmark における性能は 89%以上に改善することがわかった．生産性の観点では，local-view モデルはプログラマが MPI + OpenACC と同様にデータや処理分散を記述しないといけないために global-view モデルよりは複雑になるが，coarray によってより柔軟な通信が可能である．また MPI + OpenACC と比べると local-view モデルは coarray により配列のイメージを保ったまま通信が可能であるため MPI 関数を呼び出すより簡易である．

今後は，提案した coarray をノンブロッキングにする指示文の実装を行い，また Fortran への対応を進める．さらに Hybrid-view モデルでの XACC の評価を行う予定である．

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」，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 28 年度課題「アクセラレータおよびメニーコアを搭載したクラスタシステムのための高生産並列言語の開発と評価」による．

参考文献

[1] XcalableMP Specification Working Group. XcalableMP WebSite. http://www.xcalablemp.org.
[2] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, and Mitsuhisa Sato. XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pp. 27–36, 2014.
[3] 田渕晶大, 中尾昌広, 村井均, 朴泰祐, 佐藤三久. 演算加速機構を持つクラスタ向け PGAS 言語 XcalableACC の評価. 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 9, No. 1, pp. 17–29, Mar. 2016.

© 2017 Information Processing Society of Japan

[4] RIKEN AICS and University of Tsukuba. Omni Compiler Project. http://omni-compiler.org.
[5] Jinpil Lee, MinhTuan Tran, Tetsuya Odajima, Taisuke Boku, and Mitsuhisa Sato. An Extension of XcalableMP PGAS Lanaguage for Multi-node GPU Clusters. In *Euro-Par 2011: Parallel Processing Workshops*, Vol. 7155 of *Lecture Notes in Computer Science*, pp. 429–439. 2012.
[6] Takuma Nomizu, Daisuke Takahashi, Jinpil Lee, Taisuke Boku, and Mitsuhisa Sato. Implementation of XcalableMP Device Acceleration Extention with OpenCL. In *IPDPS Workshops*, pp. 2394–2403. IEEE Computer Society, 2012.
[7] Yili Zheng, Costin Iancu, Paul H. Hargrove, Seung-Jai Min, and Katherine Yelick. Extending Unified Parallel C for GPU Computing. In *SIAM Conference on Parallel Processing for Scientific Computing (SIAMPP)*, 2010.
[8] S. Potluri, D. Bureddy, H. Wang, H. Subramoni, and D.K. Panda. Extending OpenSHMEM for GPU Computing. *Parallel and Distributed Processing Symposium, International*, pp. 1001–1012, 2013.
[9] C. Rasmussen, M. Sottile, S. Rasmussen, D. Nagle, and W. Dumas. CAFe: Coarray Fortran Extensions for Heterogeneous Computing. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 357–365, May 2016.
[10] V. Cardellini, A. Fanfarillo, S. Filippone, and D. Rouson. Hybrid coarrays: A PGAS feature for many-core architectures. In *International Conference on Parallel Computing (ParCo 2015)*, 2015.
[11] A. Hart, R. Ansaloni, and A. Gray. Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers. *The European Physical Journal Special Topics*, Vol. 210, No. 1, pp. 5–16, 2012.
[12] PC Cluster Consortium. http://www.pccluster.org/en/.
[13] Charles H. Koelbel and Mary E. Zosel. *The High Performance FORTRAN Handbook*. MIT Press, Cambridge, MA, USA, 1993.
[14] John Reid. Coarrays in the next Fortran Standard. *ISO/IEC JTC1/SC22/WG5 N1824*, Apr. 2010.
[15] Network-Based Computing Laboratory. MVAPICH :: Home. http://mvapich.cse.ohio-state.edu/.
[16] The Open MPI Project. Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/.
[17] Wei-Yu Chen, Dan Bonachea, Costin Iancu, and Katherine Yelick. Automatic Nonblocking Communication for Partitioned Global Address Space Programs. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, pp. 158–167. ACM, 2007.
[18] Cray. Cray fortran reference manual (8.5). http://-docs.cray.com/PDF/Cray_Fortran_Reference_Manual_-85.pdf.
[19] 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/supercom/himenobmt/.
[20] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html.
[21] Center for Computational Sciences, University of Tsukuba. HA-PACS Project. http://www.ccs.tsukuba.ac.jp/eng/research-activities/projects/ha-pacs/.
[22] NVIDIA. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.
[23] NVIDIA. NVIDIA/gdrcopy: A fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology. https://github.com/NVIDIA/gdrcopy.
[24] Akihiro Tabuchi, Masahiro Nakao, and Mitsuhisa Sato. A Source-to-Source OpenACC Compiler for CUDA. In *Euro-Par Workshops*, pp. 178–187, 2013.
[25] Keisuke Tsugane, Taisuke Boku, Hitoshi Murai, Mitsuhisa Sato, William Tang, and Bei Wang. Hybrid-view programming of nuclear fusion simulation code in the PGAS parallel programming language XcalableMP. *Parallel Computing*, Vol. 57, pp. 37–51, 2016.

© 2017 Information Processing Society of Japan