Coarrays in the Context of XcalableMP

H. Iwashita and M. Nakao

Abstract @@@ XcalableACC (XACC) is an extension of XcalableMP for accelerated clusters. It is defined as a diagonal integration of XcalableMP and OpenACC, which is another directive-based language designed to program heterogeneous CPU/accelerator systems. XACC has features for handling distributed-memory parallelism, inherited from XMP, offloading tasks to accelerators, inherited from OpenACC, and two additional functions: data/work mapping among multiple accelerators and direct communication between accelerators.

Hidetoshi Iwashita
Fujitsu Limited, 140 Miyamoto, Numazu-shi, Shizuoka 410-0396, Japan, e-mail: iwashita.hideto@fujitsu.com

Masahiro Nakao
RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan, e-mail: masahiro.nakao@riken.jp

1 Introduction

Omni XMP は，PC クラスタコンソーシアムの XcalableMP 規格部会が制定する並列言語 XcalableMP（XMP）の実装である．XMP は，Fortran と C をベースとし，ディレクティブ行の挿入によって並列化を記述するが，Fortran 2008 で定義される coarray 機能も仕様として含んでいる．前者は「逐次プログラムに指示を与えて並列化する」という考え方からグローバルビューと呼ばれ，並列プログラミングが容易にできることを狙う．後者は「個々のノード（イメージ）の挙動を記述する」という考え方でローカルビューと呼ばれる。後者は、前者では記述困難な領域のアプリケーションを記述するため、それと、局所的に性能を出したい部分を MPI よりも容易なプログラミング言語という手段で記述するために導入された。そのため、XMP の文脈の中で自然な解釈ができて同時に使用できることと、同時に、MPI に匹敵する高い性能が求められた。

我々は、XMP コンパイラの一つの機能として、Fortran 2008 仕様で定義された Coarray 機能を実装した。また、言語仕様を Fortran に準じて、C 言語ベースの Coarray 機能もサポートした。Fortran で定義される image は XMP 仕様で定義される node に 1 対 1 に mapping されるため、coarray 変数の定義・参照は node 間の put/get の片側通信としてそれぞれ実現される。

To implement one-sided communication, the XcalableMP (XMP) runtime library selects one of the **communication libraries**, MPI-3, GASNet, or Fujitsu's native interface FJ-RDMA, as specified at build time. All coarray data must be **registered** to the communication library to be referenced or defined via the communication library.

本章では、coarray 機能を高性能で実現するために要求された技術と、その実現方法を示す。通信ライブラリの特質から、Coarray データの割付けは可能な限り利用者プログラムの実行前に集約すべきである。片側通信ベースなので、遠隔側のアドレス計算を低コストで実現する工夫が必要となる。通信が簡単に記述できることから、小粒度の頻繁な通信が起こりがちになるので、それらを aggregate して高速化する技術が要求される。

これらに対応した実装により、【評価の成果を】

【要修正】This chapter describes how the compiler and runtime techniques are implemented in the XMP compiler with some evaluation. In the rest of this chapter, Section 2 shows the basic implementation for coarray features, Section 3 describes how the implementation has solved the issues shown above, Section 4 evaluates the implementation with basic and practical applications, and Section 5 concludes.

2 Requirements from Language Specifications

XMP Fortran language specification supports a major part of coarray features defined in Fortran 2008 standard [5], and intrinsic procedures `CO_SUM`, `CO_MAX`, `CO_MIN` and `CO_BROADCAST` defined in Fortran 2018 standard [6] were supported. And also XMP C language specification extended to support coarray features.

This section introduces the coarray features and what is required to the compiler in order to implement the coarray features.

2.1 Images Mapped to XMP Nodes

In the Fortran standard, an **image** is defined as a instance of a program. Each image executes the same program and has its own data individually. Each image has a different image index k. While the Fortran standard itself does not specify where each image is executed, XMP specifies that images are mapped to executing nodes on a one-to-one basis. Therefore, image k is always executed on executing node k, where $1 \leq k \leq n$ and n is the number of images and also the number of the executing nodes. Since each MPI rank number of `MPI_COMM_WORLD` (0-origin) is always mapped to an XMP node number in order, image k is corresponding to rank $(k-1)$.

Note that the executing nodes can be a subset of the entire (initial) node set. For example, two distinct node sets can execute two coarray subprograms concurrently. The first executing images at the start of the program is the entire images. Coarray features are compatible to the ones of the Fortran standard unless the `TASK` and `END TASK` directives are used. If the execution encounters a `TASK` directive specified with a subset of nodes, the corresponding subset of the images will be the executing images for the task region. The current number of images and my image number, which are given by inquire functions `num_images` and `this_image`, also match with the executing images, and the `SYNC_IMAGES` statement synchronizes among the executing images. When the execution encounters the `END TASK` directive corresponding to the `TASK` directive, the set of executing image is reinstated.

Requirement to the implementation. The runtime library should manage the executing image set and the current image index in stack in order to reinstate them at the exit point of the task.

2.2 Allocation of Coarrays

A **coarray** or a coarray variable is a variable that can be referred from the other images. A coarray with the `ALLOCATABLE` attribute is called an **allocatable coarray**, otherwise called a non-allocatable coarray. A non-allocatable coarray may not be a pointer and must have an explicit shape and the `SAVE` attribute. In order to help intuitive understanding, we call a non-allocatable coarray as a **static coarray**. The lifetime of a static coarray is throughout execution of the program on all images even if the coarray is declared in a procedure called with a subset of images.

On the other hand, an allocatable coarray is allocated with the `ALLOCATE` statement and freed either explicitly with the `DEALLOCATE` statement or implicitly at the end of the scope in which the `ALLOCATE` statement is executed (**automatic deallocation**).

Static coarrays can be declared as scalar or array variables as follows:

```
real(8), save :: a(100,100)[*]
type(user_defined_type), save :: s[2,2,*]
```

The square bracket notation in the declaration distinguishes coarray variables from the others (non-coarrays). It declares the virtual shape of the images and the last dimension must be deferred (as '*').

Allocatable coarrays can be declared as follows:

```
real(8), allocatable :: b(:,:)[:]
type(user_defined_type), allocatable :: t[:,:,:]
```

A notable constraint is that at any synchronization point in program execution, coarrays must have the same dimensions (sizes of all axes) between all images (**symmetric memory allocation**). Therefore, an static coarray must have the same shape between all images during the program execution, and an allocatable coarray must be allocated and deallocated collectively at the same time with the same dimensions between the executing images. Thanks to the syn-metric memory allocation rule, all executing images can have the same symmetrical memory layout, which makes it possible to calculate the address of the remote coarray with no prior inter-image communication.

Requirement to the implementation. Static coarrays must be allocated and made accessible remotely before the execution of the user program, and made inaccessible remotely and be freed after the execution of the user program. In contrast, allocatable coarrays must be allocated and made accessible remotely when the `ALLOCATE` statement is encountered, and made inaccessible remotely and be freed when the `DEALLOCATE` statement or the exit point of the scope that the corresponding `ALLOCATE` statement is encountered is encountered.

2.3 Communications

Coarray features include three types of communications between images, i.e., reference and definition to remote coarrays, collective communications (intrinsic subroutines `CO_SUM`, `CO_MAX`, `CO_MIN` and `CO_BROADCAST`), and atomic operations (`ATOMIC_DEFINE` and `ATOMIC_REF`).

MPI library has the similar functions to

Collective communications and atomic operations are in the form of intrinsic procedures and similar ones are found in MPI library. Communication for reference and definition to remote coarrays are characteristic for coarray features.

PUT communication is caused by an assignment statement with a **coindexed variable** as the left-hand side expression, e.g.,

```
a(i,j)[k] = alpha * b(i,j) + c(i,j)
```

This statement is to cause the PUT communication to the array element $a(i, j)$ on image k with the value of the left-hand side. Using Fortran array assignment statement, array-to-array PUT communication can be written easily. E.g., the following statement causes MN-element PUT communication.

```
a(1:M,1:N)[k] = alpha * b(1:M,1:N) + c(1:M,1:N)
```

GET communication is caused by referencing the **coindexed object**, which is represented by a coarray variable with cosubscripts enclosed by square brackets, e.g., $s[1, 2]$ and $a(i, j)[k]$, where s and a are scalar and two-dimensional array coarrays, respectively. A coindexed object can appear almost in any expressions including array expressions.

Requirement to the implementation. To implement definition/reference to coindexed variable/object, PUT/GET one-sided communication is suitable to be used. To avoid costly processing such as remote procedure call, RDMA (Remote Direct Memory Access)-based implementation is desirable. On PUT/GET communication for large data, redundant multiple memory copies should carefully be avoided for all software layers, the communication library, the runtime, the Fortran library, and the object.

2.4 Inter-image Synchronization

The access order of coarrays between images must be explicitly controled by the programmer using the **image control statement**, such as `SYNC ALL` and `SYNC IMAGES` statements. It allows the compiler system to make PUT/GET communication asynchronous. The portion between such two image control statements is called as **segmemt**.

On the other hand, inside an image, the compiler system must maintain data dependency as before. It suppress the **non-blocking communication**, which postpones waiting for completion. In order to keep data dependency among the definitions and references to the same coarray in the same segment, the non-blocking communication should be restricted. The example bellow in which the same remote coarray is accessed some times inside the same segment.

```
1      if (this_image()==2) then
2          a[2]=
3          =a[2]
4          a[2]=
5      endif
```

In this case, the update of `a` in line 2 must be completed before the reference in line 3, which must be completed before the update in line 4.

Requirement to the implementation. In usual case, the completion point of the non-blocking is the end of the segment. However, the possibility to meet the condition shown above should be took account. If the PUT communication is performed

always in blocking, only the flow dependency should be care; the previous PUT communication (line 2) should be completed before the following GET communication (line 3).

2.5 Subarrays and Data Contiguity

In Fortran, if an array, except dummy argument, is declared as `real a(1:M,1:N)`, the reference to the whole array (i.e., `a` or `a(:,:)` or a(1:M,1:N)) is defined **contiguous**. We defined a term **contiguous length** as the length how long the data is partially contiguous. For example, the contiguous lengths of `a(2,3)` and `a(2:5,3)` are 1 and 4 respectively. `a(1:M,1:3)` is two-dimensionally contiguous and has contiguous length $2 \times$ `M`. `a(1:M-1,1:3)` is one-dimensionally contiguous and has contiguous length (`M` $-$ 1).

Requirement to the implementation. 通信の高速化のためには，十分に長い範囲の連続性を実行時に抽出する必要がある．一般にノード間通信で立ち上がりレイテンシ時間がほぼ無視できるようになるデータ量は数千バイトであることから，配列や部分配列の 1 次元め（Fortran では 1 番左の添字）が連続であっても数千要素に満たない場合には，次元を跨いだ連続性まで抽出して，十分に長い連続データとして通信する技術が必要である.【fx100-latency のグラフから数千バイトを言ってもよい。】

GET 通信と同様，a(i,j)[k] は代わりに全体配列にも部分配列にもなれるので，coindexed variable についても次元を跨いだ連続性の抽出が必要である．それに加えて、ローカル側、すなわち、右辺式データの連続性も意識に入れなければならない。高速な通信を実現するには，左辺と右辺で共通に連続な区間を検出してその単位で通信を反復するか，右辺データは連続区間に pack して左辺の連続区間を単位として通信を反復するなどの戦略がある。

2.6 Coarray C Language Specifications

C 言語にも配列式、配列代入を導入することで Fortran と同様な coarray features を仕様定義した。

coarray はデータ実体に限る。ポインタは coarray になれない。形式的には、1) 基本型、2) ポインタを含まない構造型、または、3) 1or2or3 の配列とする。C の coarray においても static と allocatable がある。ファイル直下で宣言するか static 指示した coarray は static coarray である。ライブラリ関数 xxx をつかって割付けたデータ領域は allocatable coarray であり、ライブラリ関数は allocatable coarray へのポインタを返す。C の coarray は、通常の C の変数と同じように、引数渡しや cast 演算によって自由にその型と形状の解釈を変えることができる。

これらの仕様と制限は、C プログラマにとっての使いやすさを考えて、C らしいプログラミングスタイルを認めた。Coarray C++とは違うアプローチである。

cf. air:/Users/iwashita/Desktop/coarray/Project_Coarray/の下にいくつか

3 Implementation

3.1 Omni XMP Compiler Framework

The CAF translator was added into the Omni XMP compiler as shown in Figure 1. The Omni XMP compiler is a source-to-source translator that converts XMP programs into the base language (Fortran or C). The component 'coarray translator' is located in front of the XMP translator to solve coarray features previously. The output of the decompiler is a standard Fortran/C program that may include calls to the XMP runtime library.

The following procedures are generated in advance or in the coarray translator to initialize static coarray variables prior to the execution of the user program:

- The built-in main program calls subroutine `xmpf_traverse_init`, the entry procedure of initialization subroutines, before executing the user main program.
- Subroutine `xmpf_traverse_init` is generated by the coarray translator to call initialization subroutines corresponding to all user-defined procedures.
- Each initialization subroutine `xmpf_init_`*foo* is generated from user-defined procedure *foo* by the coarray translator. It initializes all static coarrays declared in *foo*.

3.2 Allocation and Registration

To be accessed using the underlying communication library, the allocated coarray data must be registered to the library. The registration contains all actions to allow the data to be accessed from the other nodes, including pin-down memory, acquirement of the global address, and sharing information among all nodes.

3.2.1 Three methods of memory management

The coarray translator and the runtime library implements three methods of memory management.

- The **Runtime Sharing (RS) Method** allocates and registers a large memory for all static and dynamic coarrays at the initialization phase. The registered memory is shared by all static and allocatable coarrays.

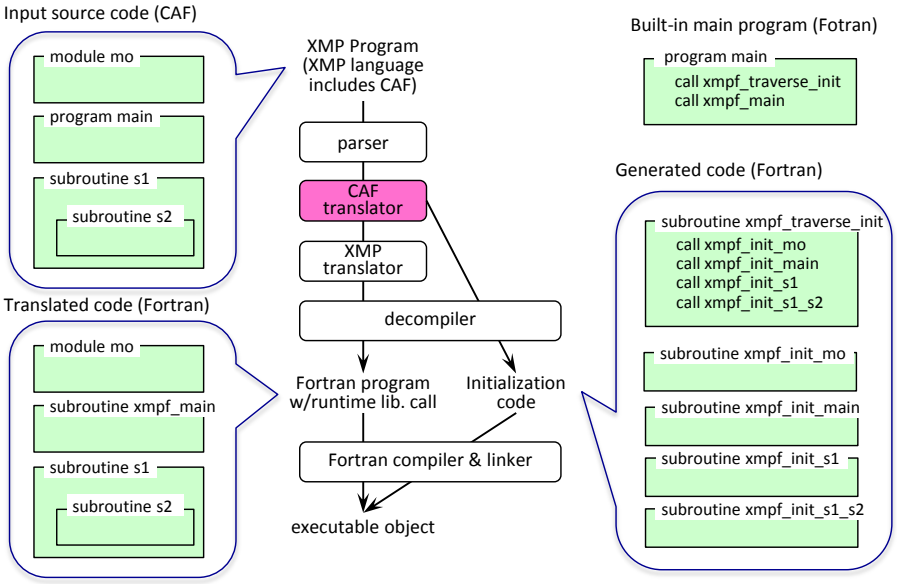


Fig. 1 XMP compiler and an example of coarray program compilation
CAF translator → coarray translator

- The **Runtime Allocation (RA) Method** allocates and registers a large memory for all static coarrays at the initialization phase. And it allocates and registers each allocatable coarray at runtime.
- The **Compiler Allocation (CA) Method** allocates all coarray objects by the Fortran system (at compile time or at runtime) and the address is passed to the runtime library to be registered.

For the RS and RA methods, because the allocated memory address is determined in the runtime library, the object code must accept the address allocated inside the runtime system as an address of a regal Fortran variable. To make this connection, it was necessary to use the Cray pointer, which is not in the Fortran standard. In the case of the CA method, the runtime library accepts the address allocated in the Fortran system, and registers to the communication library.

3.2.2 Initial Allocation for Static Coarrays

Static coarrays are allocated and registered in the initialization subroutines `xmpf_init_`*foo*.

On the RS and RA methods, static coarrays are initialized before the execution of the user program, as follows.

- In the first pass, all sizes of static (non-allocatable) coarrays are summed. The size of each static coarray is evaluated form the lower and upper bounds specified in the dimension declaration statement of each coarray. The lower and upper bound expressions, possibly including binary and unary operations, reference to name of constants, and basic intrinsic functions such as min/max and sum, are evaluated by constant folding techniques. Since the size of the structure that contains allocatable or pointer components differs depending on for the target compiler, the coarray translator get the necessary parameters to calculate the size of structures at the build time.
- Then, the total size of static coarrays is allocated and the address and the size is registered to the underlying communication library.
- In the second pass, the addresses of the all coarrays are calculated to share the registered data. Due to the language specification, sizes of the same coarray are the same among all images (nodes). So the offset from the base address of the registered data for each coarray can be the same among all images.

In the RS method, allocatable coarrays are also shared the registered memory. The total size of the memory to be registered should be specified with an environment variable by the user. While in the RA method, the total size is fully calculated by the runtime library and no information is required to the user because allocatable coarrays will be dynamically allocated on the other memories.

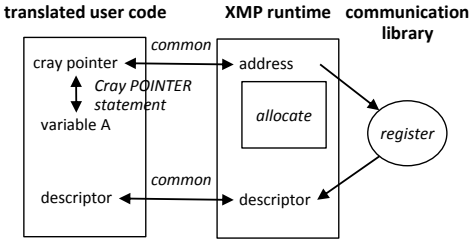
On the CA method, the Fortran processor allocates each coarray and then the runtime library registers the address. Each static coarray is converted into a common (external) variable to share between the user-defined procedure (say *foo*) and its initialization procedure (`xmpf_init_`*foo*). The data is statically allocated by the Fortran system similarly to the usual common variable. the address is registered in the initialization procedure via the runtime library.

3.2.3 Runtime Allocation for Allocatable Coarrays

For the RS method, the runtime library has a memory management system for cutting out and retrieving memory for each allocation and deallocation of coarrays.

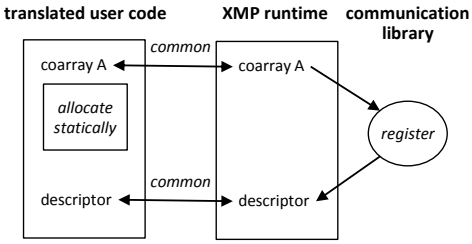
Figure 2 illustrates the memory allocation and registration for allocatable coarrays on the RA and CA methods.

These methods are properly used by the underlying communication library. On GASNet, only the RS method is adopted because its allocation function can be used only once in the program. On MPI-3, the CA method is not suitable because frequent allocation and deallocation of coarrays cause expensive creation and freeing MPI windows. Over FJ-RDMA, the RS method has no advantage over the other methods. Since the allocated address is used for registration to FJ-RDMA, no advantage was found for managing memory outside of the Fortran system. The unusual connection through the Cray pointer causes the degrade of the Fortran compiler optimization.



The runtime allocates and registers coarrays and passes the address to the user code.

(a) RA method



The user code allocates coarrays and causes the runtime to register with the address.

(b) CA method

Fig. 2 Memory allocation for coarrays in RA and CA methods

3.3 PUT/GET Communication

To avoid disturbing the execution on the remote image, PUT and GET communications are implemented always using Remote Direct Memory Access (RDMA) provided by the communication library (except coarrays with pointer/allocatable structure components). In contrast, local data access is selective between using Direct Memory Access (DMA) or using a local buffer. For the buffer scheme, one of four algorithms will be chosen.

3.3.1 Determining the possibility of DMA

coarray 変数は割付け時に registration することで RDMA アクセスが可能になる。一方で、PUT の source または GET の target となるローカル側データは、registration されていなければどの DMA の対象にならないので、あらかじめ registration されたバッファを介して通信することになる。しかし、ローカルデータも registration された coarray であることは比較的多く、その場合には DMA できる。ローカルデータが仮引数の場合、それが registrated かどうかは実行時にしか分からないので、runtime による判定が望ましい。

runtime library では、ローカルデータが registration されているか否かを binary tree で判定する仕組みを持っている。これは以下のように実現されている。

- When a chunk of coarray data is registered to the communication library, runtime library adds the set of the local address and the size into a sorted table called SortedChunkTable. The sort key is the local address.
- When a chunk of coarray data is deregistered from the communication library, runtime library deletes the record in SortedChunkTable.
- When a PUT or GET runtime library is called corresponding to a reference/definition to a coindexed object/variable, the local address is searched in SortedChunkTable with binary search. The local data is already registrated if $addr_i \leq addr < addr_i + size_i$ for any i, where $addr$ is the said local address and add_i and $size_i$ are the i-th address and size in SortedChunkTable, respectively.

データ量が大きい場合、この判定のコストは相対的に十分小さいので、まずこの判定を行い、可能なら DMA-RDMA 通信を行っている。データ量が小さいときには、この判定の時間よりバッファリングする時間の方が短くなるため、バッファリングを選ぶ。

3.3.2 Buffering Communication Methods

For the buffer scheme, one of four algorithms will be chosen depending on three parameters, the size of the local buffer B and the local and remote contiguous lengths N_L and N_R. B should be large enough to ignore communication latency overhead and we use about 400 kilo-bites in default. Unlike the case of MPI message passing, coarray PUT/GET communication requires only one local buffer for any numbers of other images. N_L and N_R can be evaluated at runtime. The Fortran syntax guarantees that N_L is a multiple of N_R or N_R is a multiple of N_L. An algorithm to get the contiguous length is shown in the paper [**?**].

Table 1 summarizes our algorithm for PUT/GET communication for five cases. The unit size is the chunk length of the PUT/GET communication. Case 0 shows the algorithm using RDMA-DMA PUT/GET communication and Cases 1 through 4 shows the algorithms using RDMA and local-buffering. Due to its strict condition, the DMA scheme is rarely used. And it is not always faster than the buffering scheme cases 2 and 3 because of the difference of the unit sizes. The merit of cases 2 and 3 is that the unit size is extended to a multiple of N_L by gathering number of short contiguous data in the buffer, or by scattering from the buffer into number of short contiguous data.

3.3.3 Non-blocking PUT communication

PUT 通信をできる限り non-blocking とし、そして、その完了待ちを可能なら次の image control statement まで遅延したい。Figure **??**に示したような、同じ

Table 1 Summary of the PUT/GET algorithm related to N_L, N_R and B

scheme	case	condition	unit size
DMA		Local data is registered.	$\min(N_L, N_R)$
buffering	1	$N_R \leq B,\ N_R \leq N_L$	N_R
	2	$N_L < N_R \leq B$	N_R
	3	$N_L < B < N_R$	multiple of N_L ($\leq B$)
	4	$B < N_R,\ B \leq N_L$	B (or less than B at last)

scheme	case	PUT action for each unit	GET action for each unit
DMA		put once	get once
buffering	1	buffer once and put once	get once and unbuffer once
	2	buffer for each N_L, and put once	get once, and unbuffer for each N_L
	3	buffer for each N_L, and put once	get once, and unbuffer for each N_L
	4	buffer once and put once	get once and unbuffer once

segment 内で同じリモートデータへ書いて読むようなケースは大変まれで、殆どの場合には次の image control statement まで遅延できる。添字式やイメージ番号が定数でないことが多いので、コンパイル時の判定では遅い方に倒れてしまう。まれなケースだけを軽いコストで正しく除外できる実行時判定が望まれる。

現在の実装では、実行時の環境変数によって blocking と non-blocking 通信を選択する。以下の条件に該当する場合には、低速となる blocking を選択しなければならない。　リモートの coarray データを更新した後で、同じセグメントの中で、同じデータを参照している。ほとんどの場合には該当しないことを利用者が分かるので、non-blocking のオプションを選択することを期待する。

3.3.4 Optimization of GET communication

A referrence to an array coindexed object is converted to a call of a runtime library function that returns a Fortran array value. For example, array assignment statement:

```
b(j1:j2) = a(i1:i2)[k]
```

is converted to:

```
b(j1:j2) = xmpf_coarray_get_generic(dp_a,k,a(i1:i2))
```

by the coarray translator, where `dp_a` is the descriptor of coarray `a`. The result value of the function is one-dimensional array whose size is (`i2-i1+1`). The issue is that it tends to cause many data copies in some library layers. As a contermeasure, we optimized specfic but common case by the translator. If a coindexed object is only the right-hand side of an array assignment statement, the entire assignment statement can be converted into one library call. The example above satisfies the condition, so it can be converted again as follows:

```
call xmpf_coarray_getsub_generic(dp_a,k,a(i1:i2),b(j1:j2))
```

In this runtime library subroutine, the variable `b(j1:j2)` is expected to be the local target of GET communication instead of the local buffer that would be generated by the Fortran runtime.

3.4 Runtime Libraries

The layer of the runtime libraries are shown in Figure 3. One of three communication libraries is selected at the build time of the Omni compiler. The coarray runtime consists of layered three libraries.

- The **lower-level runtime library (LRL)** abstracts the difference between the communication libraries except the memory management of coarray data.
- The **upper-level runtime library (URL)** solves each coarray features in the coarray program.
- The **Fortran wrapper** mediates the arguments and the result value of the translated user program (written in Fortran) and URL (written in C).

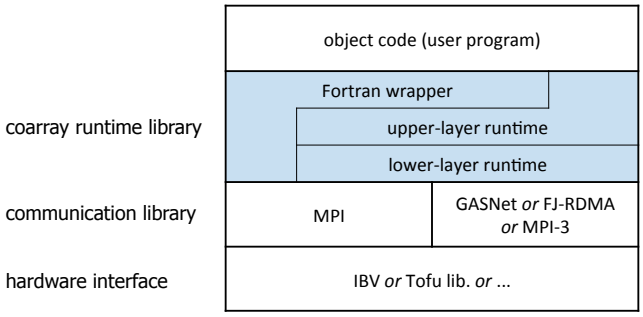


Fig. 3 Software stack for coarray features

3.4.1 Underlying Communication libraries

MPI-3 can be selected for all platform on which MPI-3 is implemented. Coarrays are registered and deregistered at the start and end point of the MPI window. Coarrays are performed one-sided communication by `MPI_Put` and `MPI_Get`, and synchronized by `MPI_Win_fence`. Implementation on MPI incurs certain costs for dynamic allocation of coarrays and waiting for communication completion.

GASNet can be selected for more advanced implementation over InfiniBand. Since allocation and registration of are inseparable and can be done only once on GASNet, the implementation allocates and registers a pool of memory whose size should be large enough to contain all static and allocatable coarrays. The XMP runtime should allocate and deallocate coarrays not using the Fortran library but using the memory manager made for the pool.

FJ-RDMA can be selected for the implementation over Tofu interconnect of the K computer and Fujitsu PRIMEHPC FX series supercomputers. Basically, each coarray is allocated by the Fortran library and registered the address with the FJ-RDMA interface `FJMPI_Rdma_reg_mem`. And it is deregistered with `FJMPI_Rdma_dereg_mem` before deallocated (freed) by the Fortran library. One-sided communication is performed with `FJMPI_Rdma_put` and `FJMPI_Rdma_get`.

3.4.2 Fortran wrapper

While the coarray runtime is written in C, coarray features are based on array notations specified in Fortran 90 or later. The Fortran wrapper is a part of the coarray runtime and mediates Fortran and C argument interfaces.

For example, suppose
tt a is a two-dimensional array coarray of 16-byte complex type. The value of a coindexed object:

$$\texttt{a(1:10,2:19)[}k\texttt{]}$$

should be a two-dimensional array shaped [10, 18]. The coarray translator converts it to a wrapper function call:

$$\texttt{xmpf\_coarray\_get\_generic(}desc\texttt{\_a, }k\texttt{, a(1:10,2:19))}$$

where *desc*_a is the descriptor of coarray a. Note that the generic function name is used in the output of the coarray translator. The corresponding specific name is selected in the following Fortran compiler as shown below:

$$\texttt{xmpf\_coarray\_get2d\_z16(}desc\texttt{\_a, }k\texttt{, a(1:10,2:19))}$$

where `xmpf_coarray_get2d_z16` means the wrapper function of GET communication for two-dimensional and 16-byte complex type. The last argument is a mold expression to get the communication pattern and the shape of the result value. At last, this function invokes the C-written runtime function:

$$\texttt{xmpf\_coarray\_get\_array(}desc\texttt{\_a, base, 16, }k\texttt{, dst, 2, skip, extent)}$$

where, `base` is the address of `a(1,2)` that is the base address of `a(1:10,2:19)`, `dst` is the result variable of 16-byte complex shaped [10, 18] in Fortran, or, the pointer to $16 \times 10 \times 80$-byte memory in C, and `skip` and `extent` are two-dimensional integer arrays to represent the pattern of the communication data.

As shown in this example, the Fortran wrapper converts Fortran multi-dimensional array segments into a set of contiguous data sequences, which can be handled in the

runtime library written in C. Conversely, it makes data returned from the runtime library an array, in the case of GET communication to an array coarray. It also converts a C pointer to a Fortran pointer with the shape, using the Cray pointer.

本実装では、Fortran の高級な仕様（配列記述や総称名手続き）を runtime library インタフェースとした。これにより、コンパイラが生成すべきオブジェクトインタフェースの数を数十分の一に削減してデータの型と形状を runtime に伝えることができるようになった。

For collective commnications, which are caused by intrinsic subroutines `CO_SUM`, `CO_MAX`, `CO_MIN` and `CO_BROADCAST`, the Fortran wrapper uses MPI library functions directly. For one-sided and collective communications, it uses URL.

3.4.3 Upper-layer runtime library (URL)

The major role of URL is performing the algorithm shown in Section 3.3 for one-sided PUT/GET communications. It uses LRL for blocking or non-blocking data transfer between pre-registered local and pre-registered remote data.

For atomic communications caused by intrinsic subroutines `ATOMIC_DEFINE` and `ATOMIC_REF`, URL calls the corresponding function of LRL after address calculation.

3.4.4 Lower-layer runtime library (LRL)

LRL basicly abstracts the difference between the communication libraries. The only exception is about allocation and registration of coarray data. The object code should exclusively select one of the following set:

- Functions to allocate and register a coarray data at the same time and functions to deregister and free it at the same time. The functions are supporsed to be used in the RS and RA methods and suitable for GASNet.
- Functions to register an already-allocated coarray data and functions to deregister it. The functions are supporsed to be used in the CA method and not suitable for GASNet.

  Other major LRL functions are shown bellow.
- A set of functions to allocate and register a specified size of coarray, and a set of functions to register an already-allocated coarray. They are alternatively used in the RS and RA methods and in the CA method. Corresponding to each, a set of functions to deregister and deallocate and a set of functions to deregister are provided.
- A function for RDMA-DMA GET communication with specified-length contiguous data, and the one of DMA-RDMA PUT communication. They assume that both remote and local data are previously registered. Blocking and non-blocking

can be switched and the only way to wait for the completion of the non-blocking communication is using the function corresponding to `SYNC MEMORY`.

- Functions corresponding to the `SYNC ALL`, `SYNC IMAGES` and `SYNC MEMORY` statements. The function corresponding to `SYNC MEMORY` waits for completion of all current non-blocking communications.
- Functions corresponding to the `ATOMIC_DEFINE` and `ATOMIC_REF` intrinsic subroutines. Each has two versions for self and remote images. Unlike PUT/GET functions, they always work in blocing.
- Some inquire functions.

LRL also has the features for multi-dimensional data developped for the C implementation, which are not used in the Fortran implementation because it is solved in URL.

3.5 Tips

- 引数の拡張。同期に visible coarray を並べる。MPI 参照。どこかの文を持ってくる。
- descriptor の逆引き。coarray の引数渡しで新しいインタフェースをつかわず non-coarray と同じインタフェースを使う。coarray 変数が引数渡しされる場合、その global address を含む descriptor を渡す。実引数が coarray 変数であっても、手続き側の仮引数は non-coarray 変数として受け取る場合、呼出し側は従来のインタフェース、つまり、先頭アドレスだけを渡すか、形状引継ぎの場合には dope vector を含む Fortran の descriptor を、実引数に載せる。呼出し側の翻訳でこのスイッチをするため、Fortran 文法では呼出し側手続きに明示的引用仕様を求める。しかし我々は、明示的引用仕様の如何に関わらず、従来インタフェースを用いることとした。手続き側では通常の変数として受け取り、これに対応する descriptor は、上述の仕組みで必要なときに runtime でその local address からハッシュ検索する。

4 Evaluation

We evaluated the Omni coarray compiler on the systems shown in Table 2.

Table 2 Specs of the compulters and evaluation environment

	RIKEN RCCS The K computer	RIKEN RCCS HOKUSAI GreatWare Fujitsu PRIMEHPC FX100	CCS, University of Tsukuba HA-PACS/TCA
CPU	SPARK64™VIIIfx, 2GHz, 128 Gflop/s, 8-core, 1CPU/node	SPARK64™XIfx, 1.975GHz, 1CPU/node, 4-SIMD × 32-core	E5-2680 v2 (Ivy Bridge), 10-core, 224Gflop/s, 2CPU/node
memory	16GB/node, bandwidth 64GB/s	32GB/node, bandwidth 480GB/s	128GB/node, 119.4GB/s
inter-connect	Tofu	Tofu2, 12.5GB/s × 2	InfiniBand FDR, 7GB/s
compiler	Fujitsu Fortran Ver. 2.0.0	Fujitsu Fortran Ver. 2.0.0	Intel Fortran 16.0.4
MPI	Fujitu MPI Ver. 2.0.0	Fujitu MPI Ver. 2.0.0	Intel MPI 5.1.3
comm. layer	Tofu library	Tofu library	GASNet 1.24.2 (IBV-conduit, built with Intel compilers)

4.1 Fundamental Performance

Using EPCC Fortran Coarray micro-benchmark [10], we evaluated ping-pong performance of PUT and GET communications compared with MPI_Send/Recv. The codes are shortly shown in Table 3.

Table 3 pingpong-code.pdf

	PUT version	GET version	MPI version

ping phase	if (me == 1) then
	x(1:n)[2] = x(1:n)
	sync images(2)
	else if (me == 2) then
	sync images(1)
	end if
		if (me == 1) then
		sync images(1)
		else if (me == 2) then
		x(1:n) = x(1:n)[1]
		sync images(1)
		end if
			if (id == 0) then
			call MPI_Send(x, n, … 1, …)
			else if (id == 1) then
			call MPI_Recv(x, n, … 0, …)
			end if

pong phase	if (me == 1) then
	sync images(2)
	else if (me == 2) then
	x(1:n)[1] = x(1:n)
	sync images(1)
	end if
		if (me == 1) then
		x(1:n) = x(1:n)[2]
		sync images(2)
		else if (me == 2) then
		sync images(1)
		end if
			if (id == 0) then
			call MPI_Recv(x, n, … 1, …)
			else if (id == 1) then
			call MPI_Send(x, n, … 0, …)
			end if

`me` is the image index. `id` is the MPI rank number.

Corresponding to the codes in Table 3, Figure 4 shows how data and messages are echanged between two images or processes. In coarray PUT (a) and GET (b), inter-image synchronization is necessary for each end of phases to make the passive image active and to make the active image passive. While, in MPI message-passing (c) and (d), such synchronization is not necessary because both processes are always

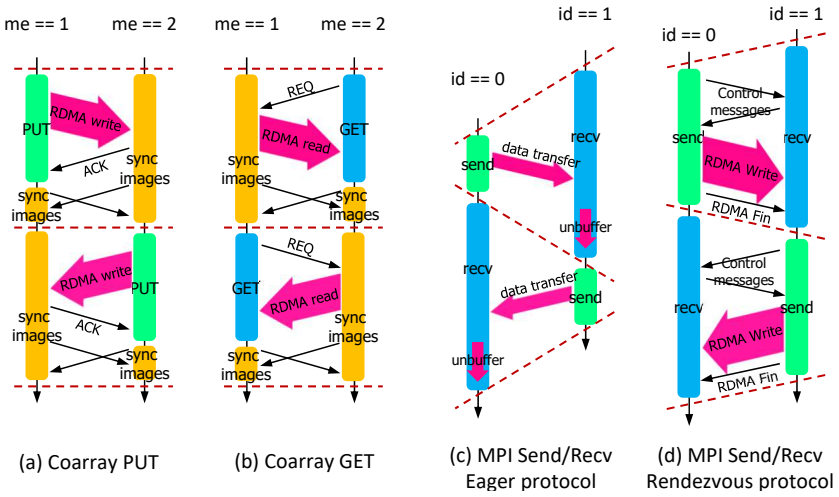


Fig. 4 pingpong-fig.pdf

active. On the other hand, MPI message-passing has its own overhead that coarray PUT/GET does not have. Because the eager protocol (c) does not use RDMA, the receiver must copy the received data in the local buffer to the target. The larger the data, the greater the overhead cost. In the rendezvous protocol (d), negotiations including remote address notification are required prior to the communication. The overhead cost is not negligible when the data is small.

The result of the comparision between coarray PUT/GET and MPI message passing is shown in Figure 5. As the underlying communication libraries, FJ-RDMA and MPI-3 are used on FX100 and GASNet and MPI-3 are used on HA-PACS. GET(a) and GET(b) are using the code without and with the optimization described in Section 3.3.4, respectively. Bandwidth is the communication data size per elapsed time, and latency is half of the ping-pong elapsed time. The difference between GET(a) and (b) is the compile-time optimization level of the coarray translator described in Section 3.3.4.

As the result, the following was found about coarray PUT/GET communication.

Bandwidth Coarray PUT and GET are slightly outperforms MPI rendezvous communication for large data on FJ-RDMA and MPI-3. On FJ-RDMA/FX100 (a), the bandwidth of PUT and GET(b) are respectively +0.1% to +18% and -0.4% to +9.3% higher than MPI rendezbous in the rendezvous range of 32k through 32M bytes. Also on MPI-3/HA-PACS, respectively +0.3% to +0.8% and +0.1% to +1.3% higher in the rendezvous range of 512k through 32M bytes.
It was confirmed from the runtime log that zero-copy communication was performed both in PUT and GET(b) by selecting DMA scheme described in Section 3.3.1.

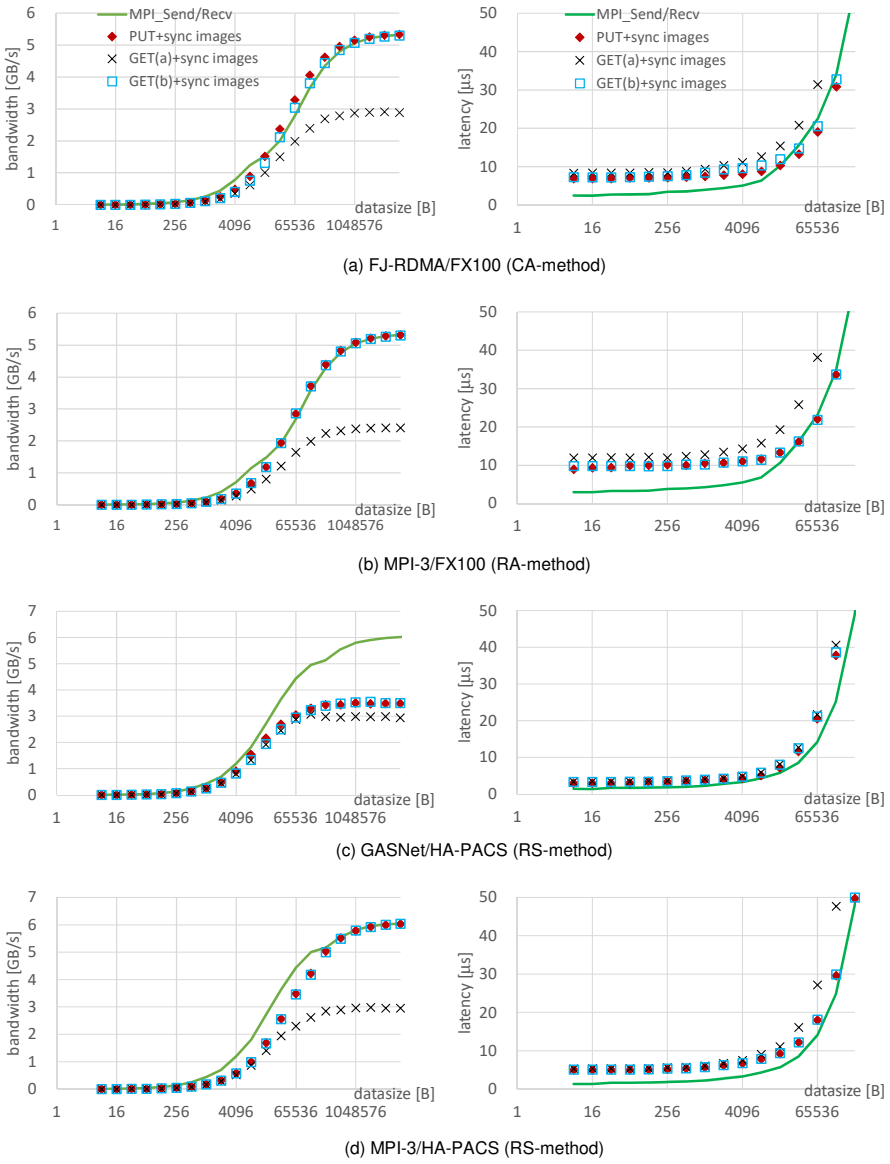


Fig. 5 Ping-pong performance on Fujitsu PRIMEHPC FX100 and HA-PACS/TCA

However, on GASNet/HA-PACS (c), PUT and GET(b) has only about 60% bandwidth for large data than MPI rendezvous. It is presumed that data copy was caused internally.

Latency On FJ-RDMA (a) and MPI-3 (b) and (d), PUT and GET(b) have larger (worse) latency than MPI eager communication, in the range of ≤16kB on FX100 and ≤256kB on HA-PACS.

Coarray on GASNet (c) behaves differently than other cases on (a), (b) and (d). Though the latency is larger than the one of MPI for all data sizes, the difference is smaller than the other cases. At data size 8B, the latency of PUT is 2.93μs and 2.1 times larger than the one of MPI while 5.73μs and 3.7 times larger on the case of MPI-3 (d).

Effect of GET optimization For all ranges of all cases, GET(a) has smaller bandwidth and larger latency than GET(b). On FJ-RDMA (a), the bandwidth is 1.41 to 1.85 times improved in the range of 32kB to 32MB by changing the object code of GET(a) to (b). We found GET(a) caused two extra memory copies; one is performing the array assignment by the Fortran library and the other is the copy from the communication buffer to the result variable of the array function `xmpf_coarray_get_generic`. The optimization described in Section 3.3.4 has eliminated these two data copies.

The issue is the large latency of coarray PUT/GET communication. In the next subsection, it is discussed how it should be solved by the compiler and the programming.

4.2 Non-blocking Communication

For latency hiding, asynchronous and non-blocking features can be expected on coarray PUT communication. The principle is shown in Figure 6. Figure 6 (a) illustrates the half pattern of the ping-pong PUT communication. Coarray one-sided communication is basically asynchronous unless synchronization is explicitly specified. Therefore, multiple communications without synchronization between them, as shown in (b), is closer to actural applications. In addition, it can be optimized using non-blocking communication as shown in (c). Blocking and non-blocking communications can be switched with the runtime environment variable in the current implementation of the Omni compiler. In MPI message passing, non-blocking communication can be written with `MPI_Isend`, `MPI_Irecv` and `MPI_Wait`.

Figure 7 compares blocking/non-blocking coarray PUT and MPI message passing communications. Two original graphs are the same as the ones of Figure 5 (a). Other four graphs display the result of 8-variable ping-pong program, which repeats the ping phase sending eight individual variables from one to the other in order and the pong phase doing similarly in the opposite direction. Each blocksize indicates the size of variables and latency includes the time for eight variables.

The following was found from the results:

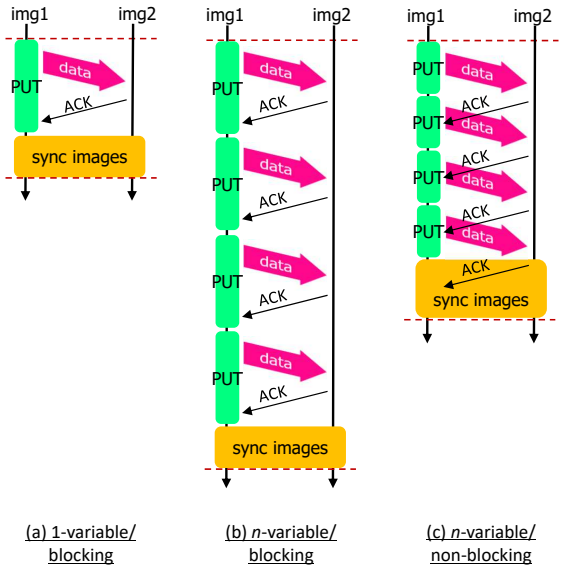


Fig. 6 nonblock-fig.pdf

- Non-blocking PUT significantly improves the latency of PUT communication. It is 4.63 times faster than blocking PUT on average from 8B to 8kB. Compared to the original PUT, it performs 8 times the communication in just 2.11 times longer on average from 8B to 8kB. Hiding completion wait behind communication (Figure 6 (c)) greatly improves the performance.
- Reduction of synchronization (Figure 6 (b)) itself does not improve the peformance. Compared to the original blocking PUT, 8-variable blocking PUT has 9.5 to 10.1 times larger latency for 8 times larger data.
- Unless data size exceeds about 8kB, the latency of non-blocking PUT is not depend on the amount of data. The graph of non-blocking PUT is very flat within ±4% over the range from 8B to 4kB.
- MPI eager communication has no effect with non-blocking for latency hiding. Eager protocol including receiver's unbuffering process seems not suitable for non-buffering.
- Non-blocking coarray PUT outperforms MPI eager message-passing except for very fine grain data. Latency of 8-variable non-blocking PUT is -9% to 54% and 18% to 61% compared to 8-variable blocking and non-blocking MPI eager,

respectively. Only at two plots of 8B and 16B, non-blocking PUT is 4% and 9% slower than blocking MPI. Otherwise, it is faster than MPI eager, and the more block size, the larger difference of the latencies.

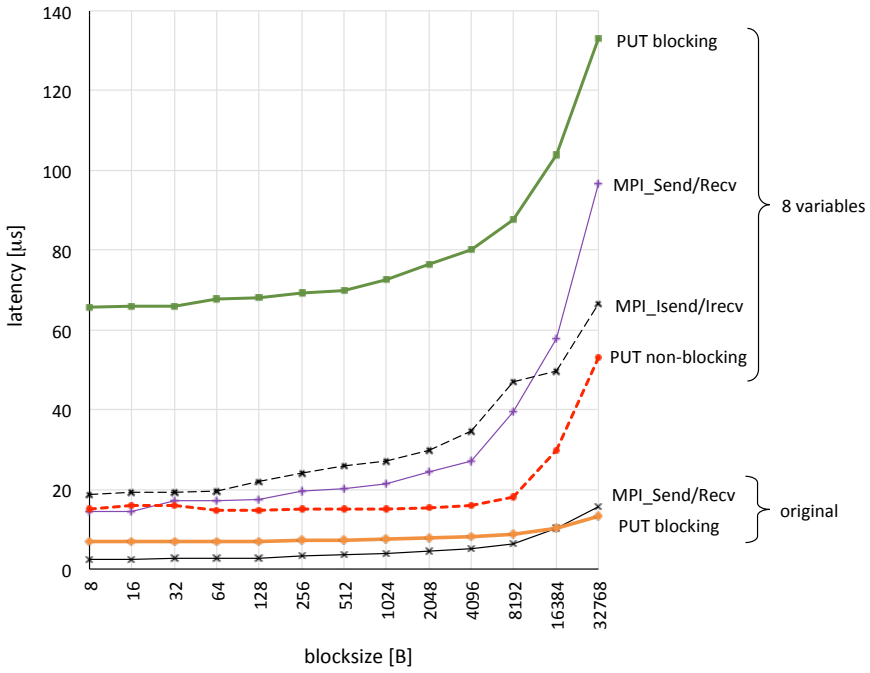


Fig. 7 8-variable ping-pong latency on PRIMEHPC FX100

4.3 Application Program

Himeno benchmark is a part of 2D Poisson's equation solver using the Jacobi iteration method [11]. The MPI version of Himeno benchmark is a strong scaling program distributed up to three-dimensional nodes. In this evaluation, we used it as a two-dimensionally distributed MPI program in the y and z axes and left the x axis to be SIMD-vectorized automatically by the Fortran compiler. The K computer and environment shown in Figure **??** was used.

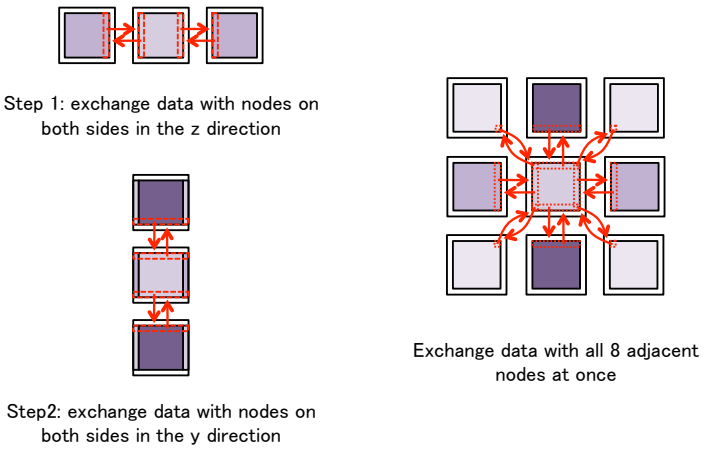
4.3.1 Coarray version Himeno benchmark

For competition, we prepared the following three versions of Himeno programs.

MPI/original The program executes computation and communication parts repetatively. The communication part consists of two steps, z-axis direction communicaiton, and then y-axis direction communication, as shown in Figure 8 (a). Each communication is written with non-blocking MPI message passing and completion wait at the end of each step.

MPI/non-blocking The 2-step communication was replaced with non-blocking scrambled communication, as shown in Figure 8 (b). With this replacement, the number of communications increases from 4 to 8 per node, but all communications become independent and can be non-blocking.

Coarray PUT/non-blocking The communication pattern is the same as the one of MPI/non-blocking. The data was declared as a coarray and each communication was written with a coarray PUT, i.e., an assignment statement with coindexed variable as the left-hand side. Since the right-hand side of the statement is the referrence to the same variable as the left-hand side coarray, the PUT communication was converted to zero-copy DMA-RDMA communication.



(a) Original MPI version (b) Non-blocking MPI and coarray versions

Fig. 8 Two algorithms of stencil communication in Himeno benchmark

4.3.2 Measurement Result

Figure 9 shows the measurement results for Himeno sizes M, L, and XL, executed on 1×1, 2×2, 4×4, ···, 32×32 nodes on the K computer. The following results were obtained:

- PUT non-blocking was the fastest in 76% of the measurement points of the graph. On 1024 nodes, it is 1.2%, 27% and 42% faster than MPI original for sizes M, L, and XL, respectively.
- As a result of analying the contents of elapsed time, it was confirmed that the difference of the performance is caused by the difference in communication time. As shown in (b) and (c), communication times of PUT non-blocking are 56% and 51% of the ones of MPI blocking on 256 nodes, respectively on L and XL sizes.
- MPI non-blocking is not always faster than MPI original. The effect of non-blocking seems to be limited in MPI.

4.3.3 Productivity

Table 4 compares the scale of the source codes. The following features can be found.

- PUT blocking requires less characters for programming than others, especially in subroutines `initcomm` and `initmax`. The MPI programmer must describe Cartesian to represent neighboring nodes in `initcomm`, and must declare MPI vector types to describe the communication pattern in `initmax`. In contrast, the coarray programmer easily represent neighboring images with coindex notation, e.g., `[i,j-1,k]`, and communication patterns with subarray notations, e.g., `p(1:imax, 1:jmax, 1)`.
- Fortran statement of MPI program tends to be longer than coarray's. Because, comparing PUT non-blocking to MPI non-blocking, the number of characters is one third while the number of statements is almost the same. This means that coarray program is more compact than the MPI program for each statement. MPI library functions often require long sequence of arguments.

5 Related Work

The Universicy of Rice has implemented coarray features with their own extension called CAF 2.0. It is a source-to-source compiler based on the ROSE compiler. GASNet is used as its communication layer. Houston University depeloped UH-CAF onto the Open64-base OpenUH compiler. It supports coarray features defined in the Fortran 2008 standard. As the communication layer, GASNet and ARMCI can be

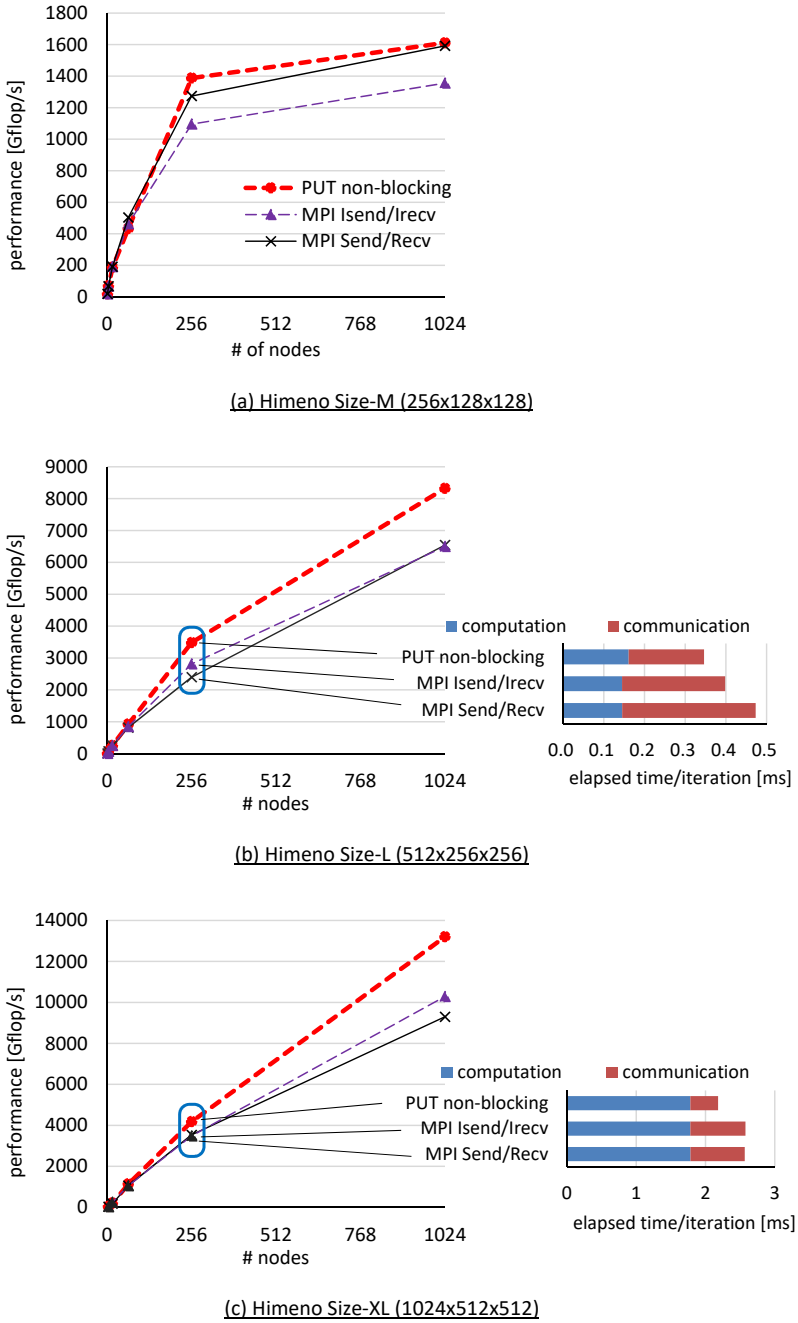


Fig. 9 himeno-graph-r2.pdf

Table 4 Comparison of source code scales for Himeno benchmark

subroutine	MPI original			MPI non-blocking			PUT non-blocking		
	LOC	SOC	chars	LOC	SOC	chars	LOC	SOC	chars
jacobi	50	33	1546	50	32	1546	43	31	1314
initcomm	65	39	1724	80	54	2380	19	19	421
initmax	95	77	2336	115	85	2939	71	71	1584
sendp	152	59	3724	299	91	7617	96	96	2435
others	248	225	5872	250	227	5923	232	231	5276
TOTAL	610	433	15202	794	489	29495	461	448	11030

LOC: Number of lines of codes excluding comment and empty lines.
SOC: Number of Fortran statements, which may span multiple lines.
chars: Number of characters excluding those in comment lines.

used selectively. OpenCoarrays [9] is an open-source software project. It is an library which can be used with GNU Fortran (gfortran) V5.1 or later. It supports coarray features specified in Fortran 2008 and a part of Fortran 2018. As the communiation layer, MPICH and GASNet can be used selectively. In the vendors, Cray and Intel fully and Fujitsu partially support the coarray features specified in Fortran 2008.

task 間は F2018 の coarray にある。違いは …

get の nonblocking は難しい。書式上、獲得したあたいがすぐに使用されることになるため、nonblocking の range を大きく取れない。put では完了を遅延させるのにたいし、開始を先行させる技術（prefetching）が考えられる。Cray はそのための directive をもつ。

Rice も Cray ポインタを使っている。Cray ポインタは alias analysis で性能を落とす。

6 Future Works

Coarray C++は Object oriented programming になれた人には自然な実装と感じるかもしれない。我々の目指す Coarray C はこれとは違い、HPC プログラムを C で書いている人向けに修正を最小限にするものである。また、coarray を抽象性の高い概念で定義するのではなく、単にアドレスと長さで特定できるメモリの範囲と定義したい。これによって一般の C プログラマがキャストや引数渡しを使って同じ領域を目的によって連続配列や多次元配列や単に先頭アドレスへの void ポインタなどとして自由に使い回すように扱いたい。これは美しくはないが、C の大きなプログラムを抱えている利用者には必要である。仕様書の定義には曖昧さが残っているので、まだ修正が必要である。Fortran と同じような性能が出せるもので、かつ、Fortran の Coarray との互換性を持たせたい。

Non-blocking PUT communication 正確な実行時判定を行うためには、現在 non-blocking PUT 通信中の coarray のアドレスの range と相手の image 番号をハッシュテーブルに記憶し、GET 通信を行う前にその range と image 番号に重なりがないことをチェックし、重なりがあったらそこで PUT 通信を完了させ

る、という方法が考えられる。しかしこれでは、重なりが無い通常のケースでも重なりのチェックのコストが大きい。コンパイラでの解析と動的なチェックを併用する、正確さが劣ってもより高速な方法が望ましい。

Other Static allocatable coarray がスタックを乱す。

7 Conclusion

本章では、XMP の文脈の中における coarray features の位置付けを示し、Omni XMP compiler の中での特徴的な実装を紹介した。

評価では、同じローカルビューを記述するプログラミングモデルである MPI message passing と比較し、性能の特徴を

実測を元にして処理系の性能を評価した。Coarray features は

XMP の中でローカルビューを担う coarray features は、性能

その価値と言えるローカルビュープログラミングの性能を評価した。

8 MEMO

9 Fusen

synmetric memory
リモートのアドレスをローカルで計算できる
これがあるから、片側通信が。。。

下位層：通信 lib の差異を吸収し、上位層にプリミティブを提供する。
XMP coarray 実装のプリミティブ i/f
put operation
get operation
- どちらも連続データのみ/nonblocking。
sync memory operation
- 発行した put をすべて remote に書き込み、get をすべて local に読み込むまで待つ。

put には nonblocking 技術：runtime で十分
get には prefetch 技術：コンパイラ技術

生産性の観点の評価

1. リンク時に static 配列を集めるところが対 MPI 片側通信で重要。言語仕様／言語処理系だからできる。単にライブラリ群／ライブラリ呼出しではできない。
2. F90 配列記述が MPI_Type の定義を不要にしている。

```
klogin7$ which xmpf90
~/Project/OMNI-clone/bin/xmpf90
klogin7$ xmpf90 --version
Version:1.3.1, Git Hash:4a5736e
klogin7$
```

sync memory がいつ必要か
- ユーザ記述
- get のとき、1 subarray 参照に対して syncmemory を 1 回にする。
- get の前に、同じ remote アドレスへの put があれば syncmemory で待つ

alignment の問題か
64 バイト未満で put の latency が高い。
src5, RUN5 で改善　… しなかった
メモリ割付けの alignment の問題ではない。
残るは Tofu の put の単位の問題と推測
マスク付き書き込みになる？
この点では RDMA よりバッファリングが有利

A Source-to-Source Translation of Coarray Fortran with MPI for High Performance
https://dl.acm.org/citation.cfm?id=3155888&dl=ACM&coll=DL

Preliminary Implementation of
Coarray Fortran Translator Based on Omni XcalableMP
https://ieeexplore.ieee.org/document/7306099

Acknowledgments

The present research used the computational resources of HA-PACS provided by the Interdisciplinary Computational Science Program at the Center for Computational Sciences at the University of Tsukuba.

Part of the results is obtained by using the K computer at the RIKEN Advanced Institute for Computational Science (Proposal number ra000002).

References

1. XcalableMP Language Specification, http://xcalablemp.org/specification.html
2. MPI: A Message-Passing Interface Standard, http://mpi-forum.org (2015).
3. John Reid, JKR Associates, UK. Coarrays in the next Fortran Standard. ISO/IEC JTC1/SC22/WG5 N1824, April 21, 2010.
4. ISO/IEC TS 18508:2015, Information technology – Additional Parallel Features in Fortran, Technical Specification, December 1, 2015.
5. Fortran 2008
6. Fortran 2018
7. Rice University. Coarray Fortran 2.0. http://caf.rice.edu/download.html
8.
9. OpenCoarrays http://www.opencoarrays.org/
10. EPCC Fortran Coarray micro-benchmark suite. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-co-array-fortran-micro
11. Himeno Benchmark. http://accc.riken.jp/en/supercom/himenobmt/

HPCAsia

[1]John Reid. 2010. Coarrays in the next Fortran Standard. ISO/IEC JTC1/SC22/WG5 N1824, (Apr. 21, 2010)
[2]ISO/IEC 2015. Information technology ^^e2^^80^^93 Additional Parallel Features in Fortran. ISO/IEC TS 18508:2015(E). (Dec. 2015)
[3]Rice University. Coarray Fortran 2.0. http://caf.rice.edu/download.html
[4]HPCTools Research Group at the University of Houston. OpenUH Open-source UH Compiler. http://web.cs.uh.edu/~openuh/
[5]OpenCoarrays. http://www.opencoarrays.org
[6]XcalableMP Specification Working Group. 2014. XcalableMP Language Specification Version 1.2.1. http://www.xcalablemp.org/specification.html
[7]Omni Compiler Project. http://omni-compiler.org
[8]Hidetoshi Iwashita, Masahiro Nakao, Mitsuhisa Sato. 2015. Preliminary Implementatio of Coarray Fortran Translator Based on Omni XcalableMP. Proc. of 9th International Conference on PGAS Programming Models (PGAS2015), P.70-75, Washington, D.C. USA (Sep.
[9]Naoyuki Shida, Shinji Sumimoto, and Atsuya Uno. 2012. MPI Library and Low-Level Communication on the K computer, FUJITSU Scientific & Technical Journal, Vol.48, No.3,
[10]David Henty. 2012. A Parallel Benchmark Suite for Fortran Coarrays. In Applications, Tools and Techniques on the Road to Exascale Computing, Advances in Parallel Computing, Vol.22, pp.281-288.
[11]Himeno Benchmark. http://accc.riken.jp/en/supercom/himenobmt/
[12]Cristian Coarfa. 2007. Portable High Performance and Scalability of Global Address Space Languages. Ph.D. Thesis, Rice University (Jan. 2007).
[13]John Mellor-Crummey, Laksono Adhianto, William N. Scherer III, and Guohua Jin. 2009. PGAS '09, Proc. of 3rd Conference on Partitioned Global Address Space Programming Models. Ashburn, VA, (Oct. 2009)
[14]Deepak Eachempati, Hyoung Joon Jun, and Barbara Chapman. 2010. An Open-Source Compiler and Runtime Implementation for Coarray Fortran. PGAS '
10, 4th Int'l
Conf. on PGAS Programming Models, No.13.

[15]Shiyao Ge, Deepak Eachempati, Dounia Khaldi, and Barbara Chapman. 2015. An Evaluation of Anticipated Extensions for Fortran Coarrays, PGAS' 15, 9th
International Conference on Partitioned Global Address Space Programming Models, P47-58, Washington, D.C. USA (Sep. 2015).
[16]Alessandro Fanfarillo, Tobias Burnus, Valeria Cardellini, Salvatore Filippone, Dan Nagle, and Damian Rouson. 2014. OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers, PGAS '14, Proc. of 8th International Conference on Partitioned Global Address Space Programming Models, No. 4.