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 Coarray Features and the Key of the Implementation

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 (SPMD: Single Program/Multiple Data). Each image has a different image index, which is one of $1, \cdots, n$, where n is the number of images. While Fortran standard does not specify actually where each image is executed, XMP defines that the images are mapped in tern to the executing nodes. Therefore, image k is always executed simply 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.

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. When 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 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. Unless the `TASK` and `END TASK` directives are used, coarray features are compatible to the ones of the Fortran standard.

Requirement to the compiler. Memory management is necessary to cope with changes of the executing images.

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 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 compiler. The language specification suggests the compiler to maintain the symmetric memory allocation rule and to use it for efficient remote address calculation.

automatic deallocation の生成が必要なことをどこかに。deregister と一緒に。片側通信を使って remote coarray をアクセスするためには、register がいることを先に。

2.3 Communications

Coarray features include three types of communications between images, i.e.,

- reference and definition to remote coarrays (GET and PUT communication),
- collective communication (intrinsic subroutines `CO_SUM`, `CO_MAX`, `CO_MIN` and `CO_BROADCAST`), and
- atomic operations (`ATOMIC_DEFINE` and `ATOMIC_REF`).

Omni compiler implements the latter two communications using the corresponding functions in MPI. The rest of this section describes the former communication.

2.3.1 PUT communication

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.

Fortran の配列代入文を使えば、配列から配列への PUT 通信を記述することもできる。例えば以下の配列代入文は、MN 要素の PUT 通信を生じさせる。

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

Requirement to the compiler. 配列代入文では、一般には右辺の評価結果を Fortran システムが一時領域に置くが、通信バッファをできる限り多重にしない工夫が必要である。右辺が配列の参照だけの場合はゼロコピー通信とすることが望ましい。また、次の image 制御文によって同期が行われるまで PUT されたデータは他の image から参照されないので、実行は PUT 通信の完了を待たないで次に進むことができる（nonblocking communication）。

2.3.2 GET communication

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. Coindexed objects can appear in most expressions.

値を配列とする配列式を使うと配列に対する GET 通信を行うことができる。例えば coindexed-object $a(1 : M, 1 : N)[k]$ の値は a と同じ型で形状 $[M, N]$ の 2 次元配列である。

Requirement to the compiler. ローカル側は、獲得したデータがすぐに消費されることから non-blocking 化は適さない。できる限りバッファリングを回避してゼロコピーとすることで効率化したい。受信側（local 側）をゼロコピーにするため、最適化が有効。右辺に 1 つの coindexed-object1 しかない配列代入文、例えば、

```
a(1:M,1:N) = b(1:M,1:N)[k]
```

では、GET 通信の受信先をバッファでなく直接変数 a にすることで、ゼロコピーにできる場合がある。

2.4 Inter-image Synchronization

Basically, the Fortran language specification does not guarantee the inter-image data dependency unless any image control statements are explicitly used. Figure 1 shows an example of program fragment describing inter-image communication and synchronization by the reference and definition of coarrays and `SYNC ALL` statement. Here, all variables `a`, `b`, `c`, `x`, `y` and `z` are coarrays. An **image control statement**, such as `SYNC ALL` and `SYNC IMAGES` statements, separates a procedure into **segments**. In Figure 1, portions between lines 2 and 9 and between lines 11 and 20 are segments. The synchronization between images does not need to complete until encountering

the image control statements. And the compiler optimization does not allowed to perform the code motion across the image control statement. In Figure 1, `sync all` in line 10 should wait for completion of the definition to coarray `a` in line 4, and should prevent the code motion of lines 4 and 5 and lines 16, 17 and 19 from crossing the `sync all` in line 10.

緩い同期である。ここに最適化のチャンスがある。

```
 1      sync all
 2
 3      if (this_image()==1) then
 4          a[2]=
 5          b[2]=
 6          =c[2]
 7          x=
 8          y=
 9          =z
10      endif
11
12      sync all
13
14      if (this_image()==2) then
15          =a
16          b=
17          c=
18          =x[1]
19          y[1]=
20          z[1]=
21      endif
22
23      sync all
```

Fig. 1 Example of inter-image synchronization

各イメージの中では、従来の Fortran 仕様に従う依存関係の維持が必要である。non-coarray データに関しては従来とおりだが、coarray について注意が必要。In order to keep data dependency among the definitions and referrences in the coarray, the nonblocking communication should be restricted. Figure 2 に data dependency に気をつけなければならない例を示す。同じリモートデータに対して同じ segment 内で複数回アクセスする場合である。

Requirement to the compiler. To reduce the latency overhead, nonblocking one-sided communication is effective. The compiler should generate nonblocking GET communication as long as possible. In usual case, the completion point of the nonblocking is the end of the segment, as shown in Figure 1. However, the posibility to meet the condition shown in Figure 2 should be took account. If the PUT communication is 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).

```
1      if (this_image()==2) then
2          a[2]=
3          =a[2]     ! should wait for the completion of line 2 to keep flow dependency.
4          a[2]=     ! should wait for the completion of line 3 to keep anti dependency.
5      endif
```

Fig. 2 Example of intra-image communication blocking

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 compiler. 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 Compiler Implementation

3.1 Omni XMP Compiler Framework

The CAF translator was added into the Omni XMP compiler as shown in Figure 3. 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.

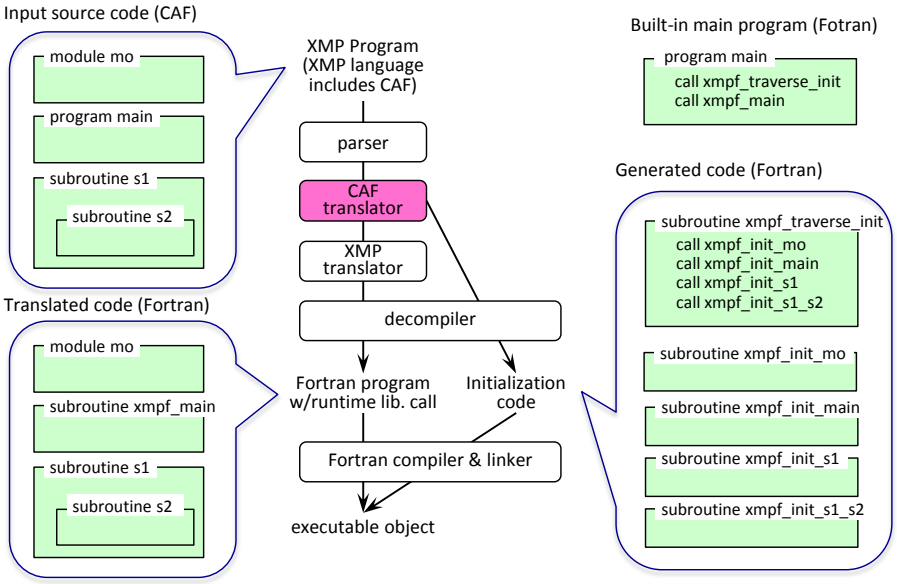


Fig. 3 XMP compiler and an example of coarray program compilation
CAF translator → coarray translator

- 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.
- 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.

3.3 Initial Allocation

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

On the **Runtime-library Sharing (RS) method** and on the **Runtime-library Allocation (RA) method**, 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 declaration statement of each coarray. Because the dimensions may have any integer constant expression, the coarray translator evaluates name of constants, binary and unary operations, and basic Fortran intrinsic functions such as min/max and sum.
- 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.

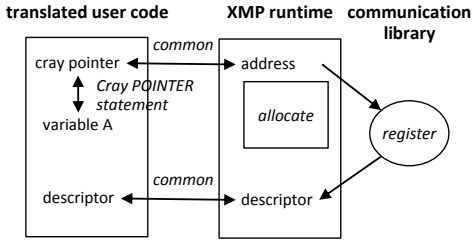
On the **Compiler Allocation (CA) method**, the Fortran processor allocates each coarray and then the runtime library registers the address. Each static coarray is converted into a common (external) variable to share between the user-defined procedure (say *foo*) and its initialization procedure (`xmpf_init_`*foo*). 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.

3.3.1 Allocation at Runtime

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.

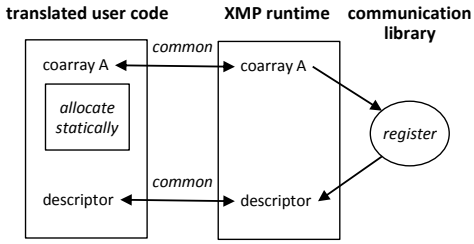
Figure 4 illustrates the memory allocation and registration for allocatable coarrays on the RA and CA methods.

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



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. 4 Memory allocation for coarrays in RA and CA methods

found for managing memory outside of the Fortran system. The unusual connection through the Cray pointer causes the degrade of the Fortran compiler optimization.

3.4 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 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.

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 every unit	GET action for every 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

3.5 Non-blocking one-sided communication

GET 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 2 に示したような、同じ 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。まれなケースを除外するための実行時判定が望まれる。

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。

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。　明示的な同期なしで、リモートの coarray 変数に対して定義後の参照がある。

4 Runtime Libraries

The layer of the runtime libraries are shown in Figure 5. One of the three underlying 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).

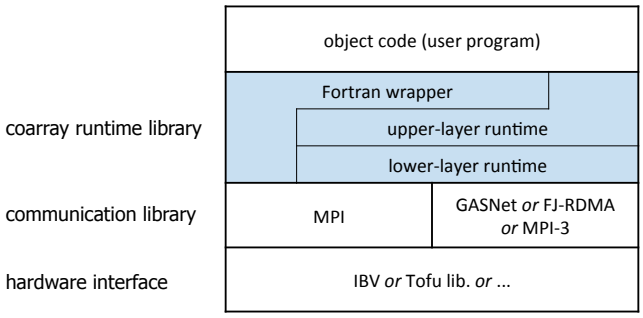


Fig. 5 Software stack for coarray features

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`.

4.2 Lower-layer runtime library (LRL)

LRL abstracts the difference between the communication libraries except which allocates the coarray data between the coarray runtime (by the RS and RA methods) and the Fortran system (by the CA method). The features for multi-dimensional data developped for the C implementation are not used for the Fortran implementation.

The LRL functions currently used in the Fortran implementation are as follows.

- 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.

4.3 Upper-layer runtime library (URL)

URL implements three kinds of coarray communications as follows:

- For one-sided PUT/GET communications caused by the referrence of coindexed variables/objects, the algorithm shown in Section 3.4 is implemented in URL. The algorithm requires only the contiguous communicaton between pre-registered local and remote data. The Fortran wrapper mediates the argument interfaces.
- For collective commnications caused by intrinsic subroutines `CO_SUM`, `CO_MAX`, `CO_MIN` and `CO_BROADCAST`, URL lost the role in the latest implementation. The Fortran wrapper uses MPI library functions directly.
- For atomic communications caused by intrinsic subroutines `ATOMIC_DEFINE` and `ATOMIC_REF`, URL calls the corresponding function of LRL after address calculation.

4.4 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:

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

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:

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

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:

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

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 × 10 × 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 solves multi-dimensional array segments of Fortran into C, and vise visa. It also converts a C pointer to a Fortran pointer with the shape, using the Cray pointer and tricky coding.

5 Evaluation

測定環境 FX100 HA-PACS

5.1 Ping-pong

EPCC CAF benchmark

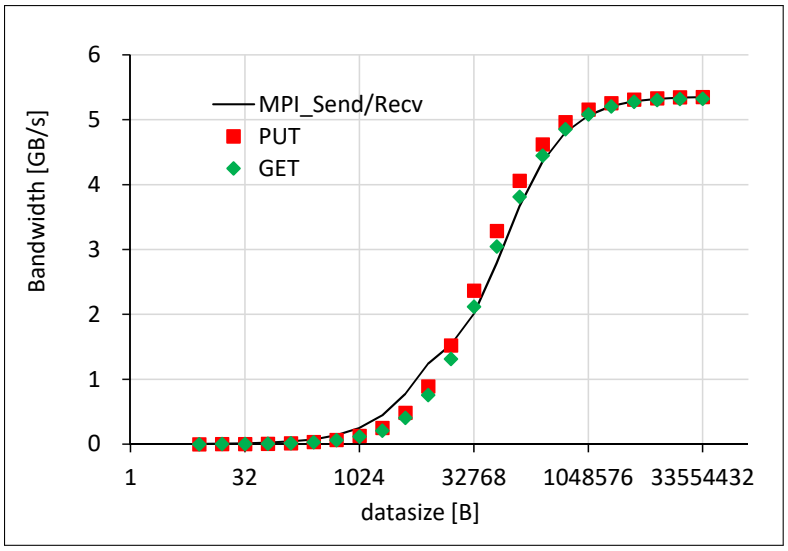


Fig. 6 fx100-bw.pdf

As the result, coarray PUT and GET are slightly outperforms MPI rendezvous message-passing for large data. In Figure 6, the bandwidth of PUT and GET are respectively +0.1% to +18% and -0.4% to +9.3% higher than MPI rendezbous in the

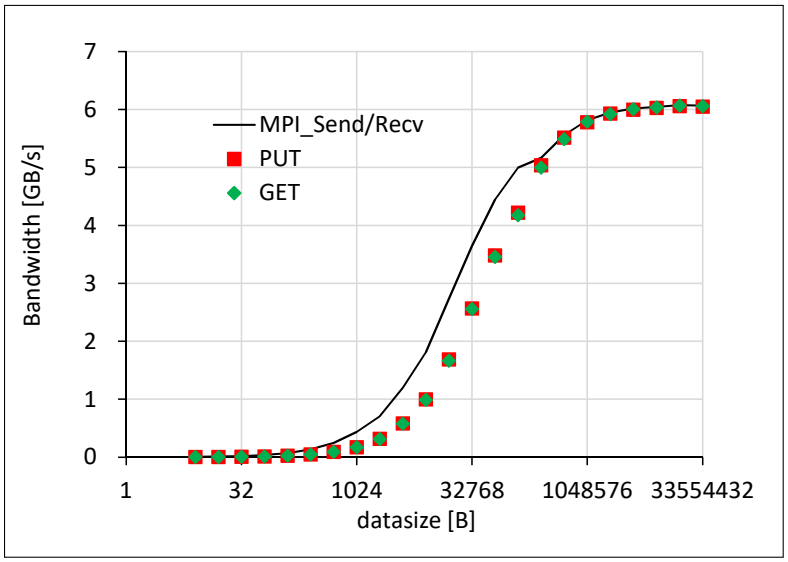


Fig. 7 HAPACS-bw.pdf

rendezvous range of 32k through 32M bytes. Also in Figure 7, respectively +0.3% to +0.8% and +0.1% to +1.3% higher in the rendezvous range of 512k through 32M bytes.

However, coarray PUT and GET are clearly slower than the MPI eager communication in the range of shorter length data.

分析 PUT/GET が eager に比べて遅い理由は、ack 待ちと sync all のオーバヘッド eager にはこのようなオーバヘッドがない。RDMA でないのでコピーがあるけれども。

5.2 OTHERS

6 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 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

	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

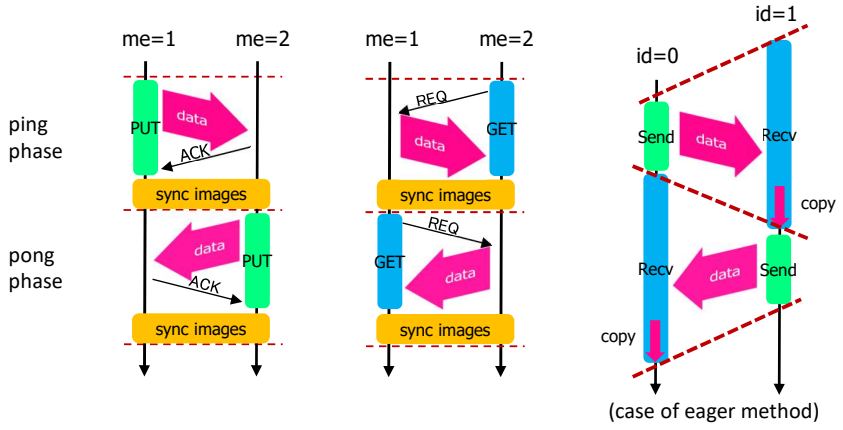


Fig. 8 pingpong-code.pdf

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.

Coarray C++は

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。

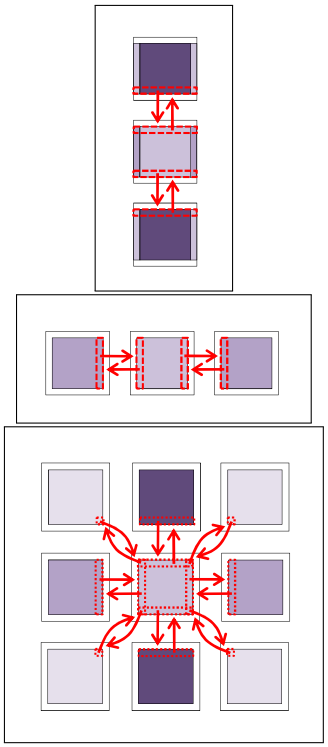


Fig. 9 3cell-y.pdf, 3cell-z.pdf, 9cell-yz.pdf

7 Conclusion

8 MEMO

MPI は PGAS に向かない、という論文がどこかに。

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。

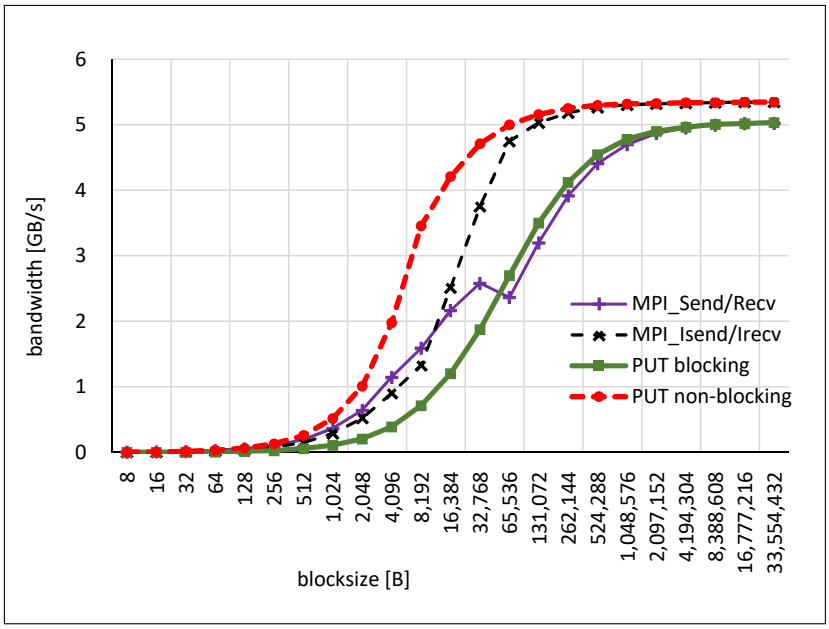


Fig. 10 8var-pipo-bw.pdf

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

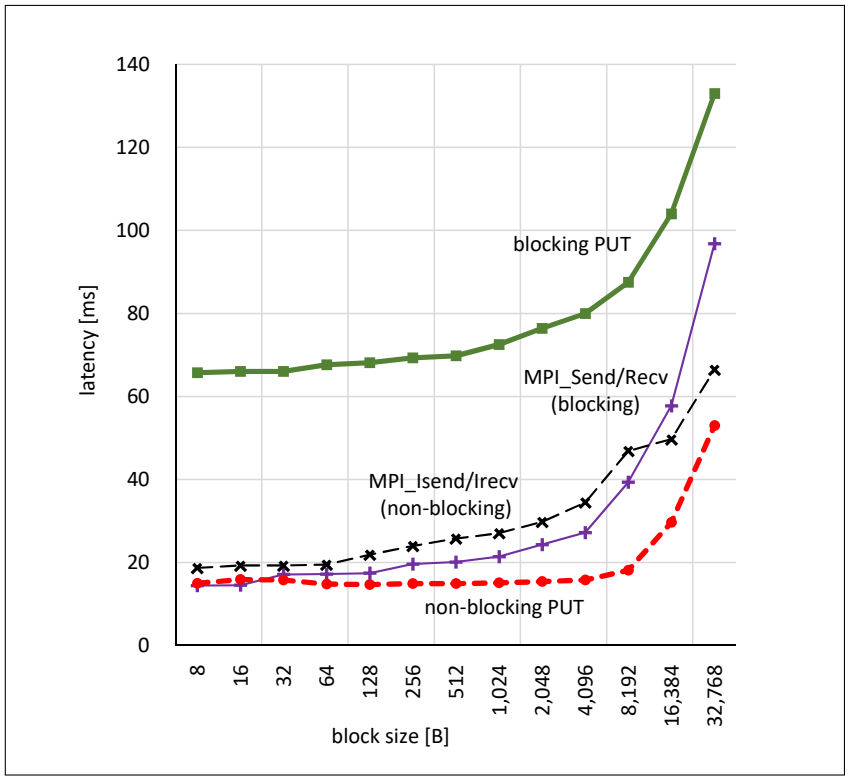


Fig. 11 8var-pipo-latency.pdf

```
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？

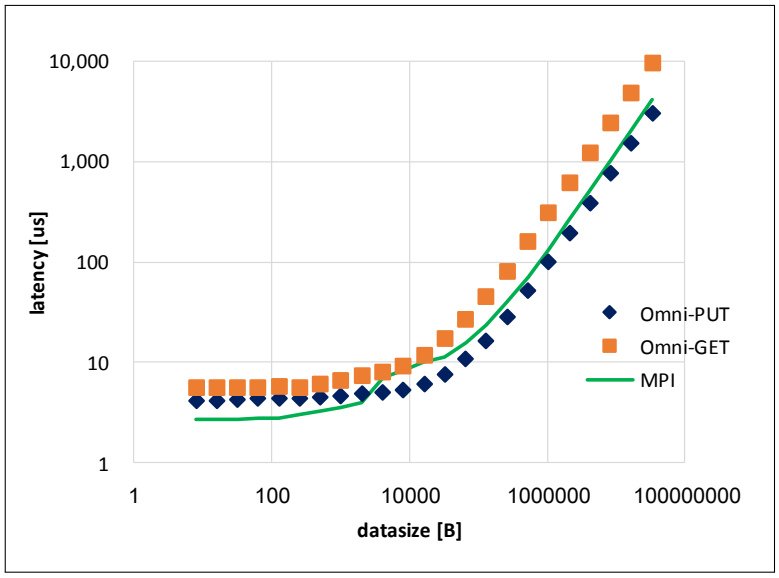


Fig. 12 fx100-pipo.pdf

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

10 STATUS-CAF

```
1.2  Interoperability with the global-view features (NEW since V1.0)
  Coarray features can be used inside the TASK directive blocks. As default,
  each coarray image is mapped one-to-one to a node of the current executing
  task. I.e., num_images() returns the number of nodes of the current executing
  task and this_image() returns each image index in the task.
     There are two directives to change the default rule above. A COARRAY
  directive corresponding to a coarray declaration changes the image index set
```

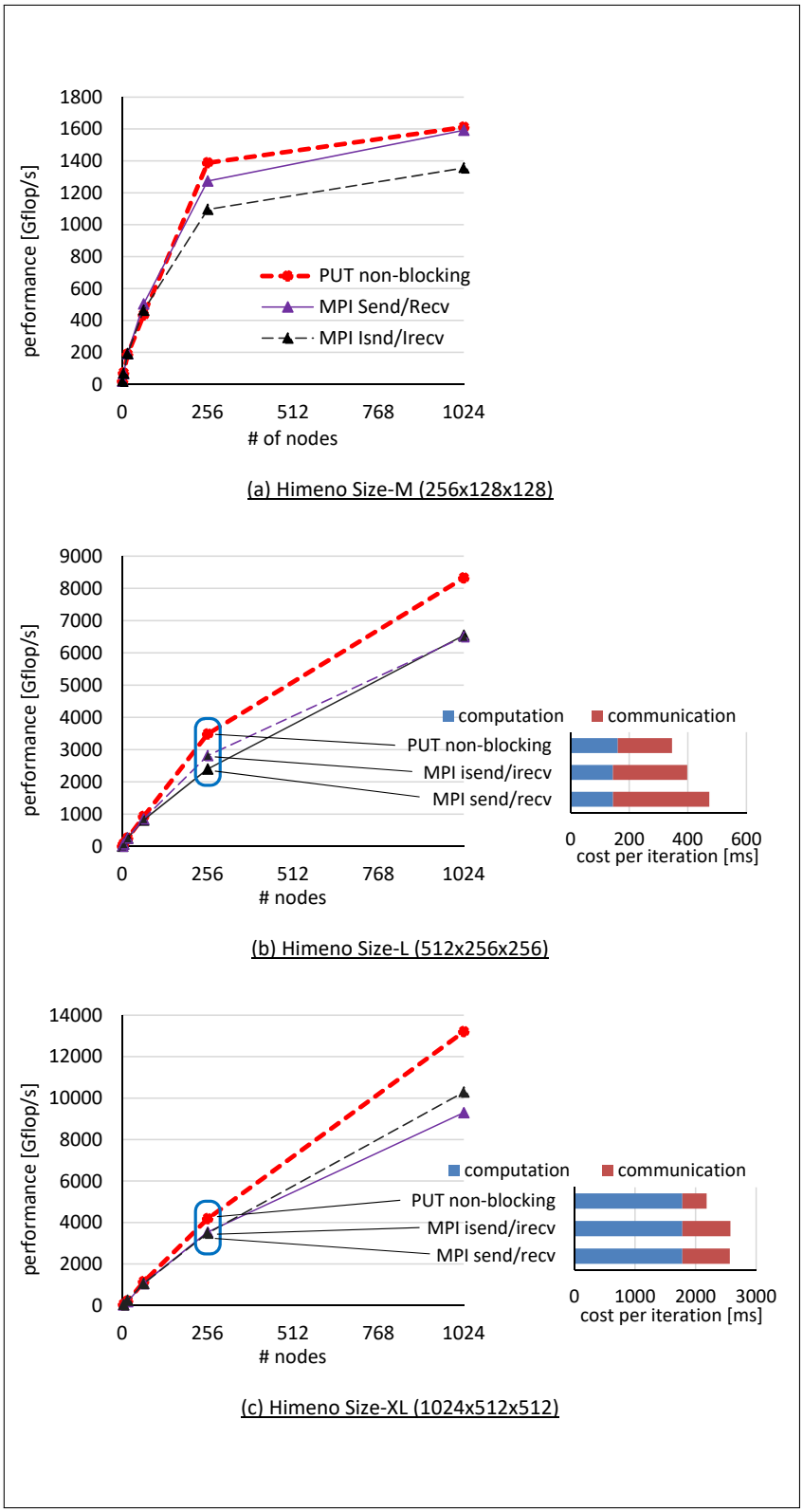


Fig. 13 himeno-graph-r2.pdf

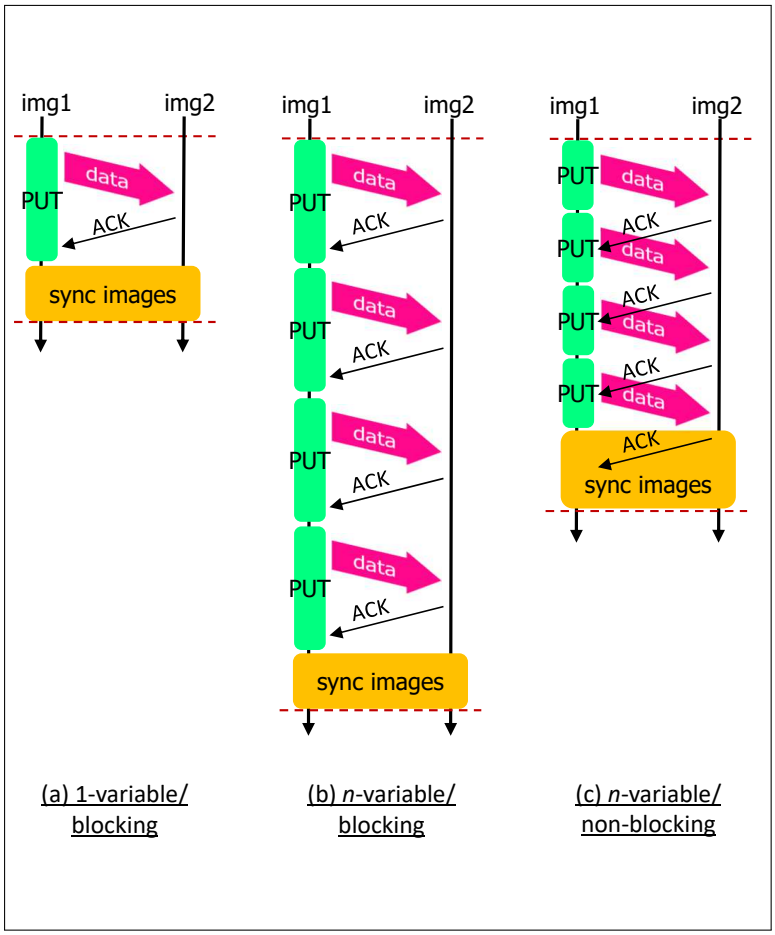


Fig. 14 nonblock-fig.pdf

```
    of the specified coarray with the one of the specified nodes. An IMAGE
    directive corresponding to one of a SYNC ALL statement, a SYNC IMAGES
    statement, a call statement calling CO_SUM, CO_MAX, CO_MIN or CO_BROADCAST
    changes the current image index set with the one of the specified nodes.
    See the language spacifications [3].

2. Declaration
    Either static or allocatable coarray data objects can be used in the program.
    Use- and host-associations are available but common- or equivalence-
    association are not allowed in conformity with the Fortran2008 standard.
    Current restrictions against Fortran2008 coarray features:
      * Rank (number of dimensions) of an array may not be more than 7.
```

```
    * A coarray cannot be of a derived type nor be a structure component.
    * A coarray cannot be of quadruple precision, i.e., 16-byte real or 32-byte
      complex.
    * Interface block cannot contains any specification of coarrays. To describe
      explicit interface, host-association (with internal procedure) and use-
      association (with module) can be used instead.
    * A pointer component of a derived-type coarray is not allowed.
    * An allocatable component of a derived-type coarray cannnot be referrenced
      as a coindexed object.
    * A derived-type coarray cannot be defined as allocatable.

2.1  Static Coarray
  E.g.
      real(8) :: a(100,100)[*], s(1000)[2,2,*]
      integer, save :: n[*], m(3)[4,*]
  The data object is allocated previously before the execution of the user
  program.  A recursive procedure cannot have a non-allocatable coarray without
  SAVE attribute.
  Current restrictions against Fortran2008 coarray features:
    * Each lower/upper bound of the shape must be such a simple expression that
      is an integer constant literal, a simple integer constant expression, or
      a reference of an integer named constant defined with a simple integer
      constant expression.
    * A coarray cannot be initialized with initialization or with a DATA
      statement.

2.2  Allocatable Coarray
  E.g.
      real(8), allocatable :: a(:,:)[:], s(:)[:]
      integer, allocatable, save :: n[:], m(:)[:,:]
  The data object is allocated with an ALLOCATE statement as follows:
      allocate ( a(100,100)[*], s(1000)[2,2,*] )
  The allocated coarray is deallocated with an explicit DEALLOCATE statement or
  with an automatic deallocation at the end of the scope of the name unless it
  has SAVE attribute.
  Current restrictions against fortran2008 coarray features:
    * A scalar coarray cannot be allocatable.
    * An allocatable coarray as a dummy argument cannot be allocated or
      deallocated inside the procedure.

3. Reference and definition of the remote coarrays
  For the performance of communication, it is recommended to use array
  assignment statements and array expressions of coindexed objects as follows:
      a(:) = b(i,:)[k1] * c(:,j)[k2]    !! getting data from images k1 and k2
      if (this_image(1))  d[k3] = e     !! putting data d on k3 from e
```

```
    Current restrictions on the K computer and Fujitsu PRIMEHPC FX10:
      * The coindexed object/variable must be aligned with the 4-byte boundary
        and the size of the array elements of them must be a multiple of 4 bytes.

  4. Image control statements
    SYNC ALL, SYNC MEMORY and SYNC IMAGES statements are available.
    Current restrictions against Fortran2008 coarray features:
      * LOCK, UNLOCK, CRITICAL and END CRITICAL statements are not supported.
      * STAT= and ERRMSG= specifiers of image control statements are not
        supported.
      * ERROR STOP statement is not supported.

  5. Incrinsic Functions
    Inquire functions NUM_IMAGES, THIS_IMAGE, IMAGE_INDEX, LCOBOUND and LUBOUND
    are supported.
    Current restrictions against Fortran2008 coarray features:
      * ATOMIC_DEFINE and ATOMIC_REF subroutines are not supported.

  6. Intrinsic Procedures in Fortran2015
    Argument SOURCE can be a coarray or a non-coarray.
    Intrinsic subroutines CO_BROADCAST, CO_SUM, CO_MAX and CO_MIN can be used
    only in the following form:
      * CO_BROADCAST with two arguments SOURCE and SOURCE_IMAGE
        E.g.,  call co_broadcast(a(:), image)
      * CO_SUM, CO_MAX and CO_MIN with two arguments SOURCE and RESULT
        E.g.,  call co_max(a, amax)
```

11 mac-air memo

```
introduction
coarrays in XcalableMP
support F2008 coarrays
image set を task に対して map
natural extension to C
basic implementation and issues
lower-level interface
evaluation
related work
Coarray C++は
```

task 間は F2018 の coarray にある。違いは …
Cray は get を directive で実装
conclusion

implementation for high performance
static coarray の高速化
全部まとめて先に alloc するコンパイラ技術
定数評価、構造体の大きめな見積り
allocatable coarray の高速化：lowlevel に合わせて選択
RuntimeLibShare 事前に巨大領域を取る
実行時 alloc のコストが大きいとき
RuntimeLibAllocation 実行時 alloc のコストが小さいとき
contiguouity 検出による高速化
次元を跨ぐ連続性抽出
バッファリング
有限サイズ、基礎データから、Nlocal, Nremote, Nbuf の関係でアルゴリズム
CommpilerAllocationFortran が allocate して register できるとき

implementation software layer
３種:one sided, collective, atomic. このうち collective と atomic は MPI の機能をそのまま使う。それ以上の工夫はない。one sided は allocate/free と put/get と同期。lower-level 通信層のバリエーションを吸収するライブラリ層を設けた。階層の図。
次元の概念と contiguity を上位層で解決するため結果的に必要なインタフェースは少なくなった

12 nigun

12.1 intro より related work の方がよいか？

PGAS 言語の一つである Coarray Fortran（CAF）は，Fortran 2008 仕様の一部として採用されたことも追い風となって，近年研究開発が盛んに進められている．フリーのものでは，Houston 大学の OpenUH コンパイラを開発基盤とした UH-CAF[5] と，Rice 大学の ROSE コンパイラを開発基盤とした caft[6] が有名である．近年リリースされた OpenCoarray は，GNU gfortran にリンクできるライブラリである．我々の開発する Omni XMP もまた，CAF コンパイラとして利用することができる．ベンダでは Cray と Intel が古くから提供しており，近年富士通からもリリースされている．これら 3 社は，Fortran 2008 と 2015 に含まれる coarray 機能の実装を Fortran2003 のフル実装よりも先行させたこと

になる．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）のパイロット実装である．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，MPI に匹敵する性能が MPI よりも容易に出せることを狙っている．

References

1. XcalableMP Language Specification, http://xcalablemp.org/specification.html (2017).
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.
8.
9. http://www.opencoarrays.org/