Coarrays in the Context of XcalableMP

H. Iwashita and M. Nakao

Abstract Coarray features have been implemented on the Omni XcalableMP compiler with a source-to-source translator and layered runtime libraries. Three memory allocation methods for coarrays were implemented for the GASNet and MPI-3 communication libraries and the native interface of Fujitsu. For the coarray PUT/GET communication, algorithms using DMA (zero-copy) and buffering were introduced. Important techniques for achieving high performance were the non-blocking PUT communication implemented in the runtime library and the optimization for the GET communication in the translator. Using the ping-pong benchmark and the modified version, the fundamental performance was evaluated and analyzed. The MPI version of the Himeno benchmark was ported to the coarray version and modified for fully using the non-blocking PUT. As a result of the evaluation, the non-blocking coarray version clearly outperformed the original and non-blocking MPI versions.

1 Introduction

XcalableMP (XMP) [1] has complementary global-view and local-view programming models. The former is a directive-based language extension to the base languages Fortran and C, and the latter adopts the coarray features defined in Fortran 2008 [2] and a part of the coarray features defined in Fortran 2018 [3]. The purpose of the coarray features as the local-view part of XMP is 1) writing application programs, which is difficult for global-view programming and 2) writing important parts of programs that are critical to performance with an easier programming model than

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

MPI message passing. Therefore, the coarray features in XMP must be naturally merged into the global-view XMP language and must exhibit high performance, comparable to that of MPI.

The Omni XMP compiler is an open-source implementation developed at RIKEN and the University of Tsukuba [4]. The kernel of the Omni XMP compiler is a source-to-source compiler that converts an XMP program into a Fortran program by calling a runtime library. The coarray translator has been implemented on the Omni XMP compiler. Since the images are mapped one-to-one to XMP nodes, each image was implemented as a process, and the definition and reference to coarrays were implemented as inter-node one-sided communications.

This chapter describes the techniques used in the coarray compiler and the runtime library, and a comparison to MPI message passing. The remainder of this chapter is organized as follows. Section 2 introduces the requirements of the coarray features. Section 3 describes the implementation used to solve the requirements, and Section 4 evaluates the performance and productivity of coarray programs. Related research is described in Section 5, and Section 6 concludes this chapter.

2 Requirements from Language Specifications

The XMP Fortran language specification [1] supports many of the coarray features defined in the Fortran 2008 standard [2], and intrinsic procedures `CO_SUM`, `CO_MAX`, `CO_MIN`, and `CO_BROADCAST` defined in the Fortran 2018 standard [3]are supported. In addition, the XMP C language specification was extended to support coarray features.

This section introduces the coarray features and what is required of 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 an instance of a program. Each image executes the same program and has its own individual data. 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 as well as the number of 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 corresponds 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 are entire images. Coarray features are compatible with those 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, then the corresponding subset of 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 the executing images, and the SYNC_IMAGES statement synchronizes the executing images. When the execution encounters the END TASK directive corresponding to the TASK directive, the set of executing images is reinstated.

Requirement to the implementation. The runtime library should manage the executing image set and the current image index in a 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**, and is 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 refer to a non-allocatable coarray 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 brackets in the declaration distinguish coarray variables from other (non-coarray) variables. The declaration 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) for all images (**symmetric memory allocation**). Therefore, a static coarray must have the same shape for all images during program execution, and an allocatable coarray must be allocated and deallocated collectively at the same time with the same dimensions for the executing images. Thanks to the symmetric 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 execution of the user program and made inaccessible remotely and be freed after 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 freed when the `DEALLOCATE` statement is encountered or the exit point of the scope to which the corresponding `ALLOCATE` statement is encountered.

2.3 Communication

Coarray features in XMP 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`). Collective communications and atomic operations are similar to those in MPI library. Communications 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 causes the PUT communication to the array element `a(i,j)` on image `k` with the value of the left-hand side. Using the Fortran array assignment statement, array-to-array PUT communication can be written easily, e.g., the following statement causes `M`×`N`-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 in almost any expression, including array expressions.

Requirement to the implementation. In order to implement definition/reference to a coindexed variable/object, PUT/GET one-sided communication is suitable for use. In order to avoid costly processing, such as a remote procedure call, remote direct memory access-based (RDMA-based) implementation is desirable. In 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 Synchronization

The access order of coarrays between images is explicitly controlled by the programmer using the **image control statement**, such as `SYNC ALL` and `SYNC IMAGES` statements. The statement allows the compiler system to make PUT/GET communication asynchronous. The sequence of execution between the image control statements is called as a **segment**. An asynchronous communication must be completed by the end of the segment.

Inside each image, the compiler must maintain data dependency as before, even if the compiler contains coarray communications. The compiler suppresses the **non-blocking communication**, which postpones waiting for communication completion. In order to keep data dependency among the definitions and references to the same coarray in the same segment, non-blocking communication should be restricted. The example below in which the same remote coarray is accessed a number of times inside the same segment.

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

Between lines 2 and 3, the completion wait for PUT communication is necessary in order to avoid referencing data that is not defined completely. Similarly, between lines 3 and 4, the completion wait for GET communication is necessary in order to avoid referencing data that is being updated. However, between lines 4 and 5, the completion wait is not necessary. The issue of race condition on image 2 cannot be avoided by the completion wait on image 1 in general, and avoiding this issue is up to the programmer.

Requirement to the implementation. Unless the same remote data is accessed from the same segment, non-blocking completion can be delayed until the end of the segment. Since the data received by the GET communication is usually referenced soon, non-blocking GET communication is hard to use. Therefore, if GET communication is always on blocking, then only the flow dependency (between lines 2 and 3) should be considered.

2.5 Subarrays and Data Contiguity

Except for a dummy argument, an array is fully **contiguous** across the dimensions. A subarray of the array can be fully or partially contiguous or non-contiguous. For example, if an array is declared with the shape `a(1:M,1:N)`, then the whole array (referenced as `a` or `a(:,:)` or a(1:M,1:N)) is fully contiguous and a subarray `a(2:5,3)` is partially contiguous. We defined a term **contiguous length** as the length for which 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 a contiguous length (`M` $-$ 1).

Requirement to the implementation. For high-performance communication, it is important to find the contiguous length across the dimensions, because thousands of bytes of contiguous data is needed in order to be comparable to the communication latency in general, and only the first dimension of the array is not always long enough.

2.6 Coarray C Language Specifications

The XMP language specification extends the C language to support coarray features. Array notations, such as subarray and array assignment statements, are adopted in the C language. In XMP/C, a coarray is a data object but is not a pointer. A coarray is either 1) of basic type, 2) a structure in which no component is a pointer, or 3) an array of 1, 2, or 3.

XMP/C also has static and allocatable coarrays. Coarray variables declared directly in the file and declared with the `static` attribute are static. Coarray variables can be allocated with intrinsic functions.

3 Implementation

3.1 Omni XMP Compiler Framework

The CAF translator was added to the Omni XMP compiler [4], 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, which 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, which initializes all static coarrays declared in *foo*.

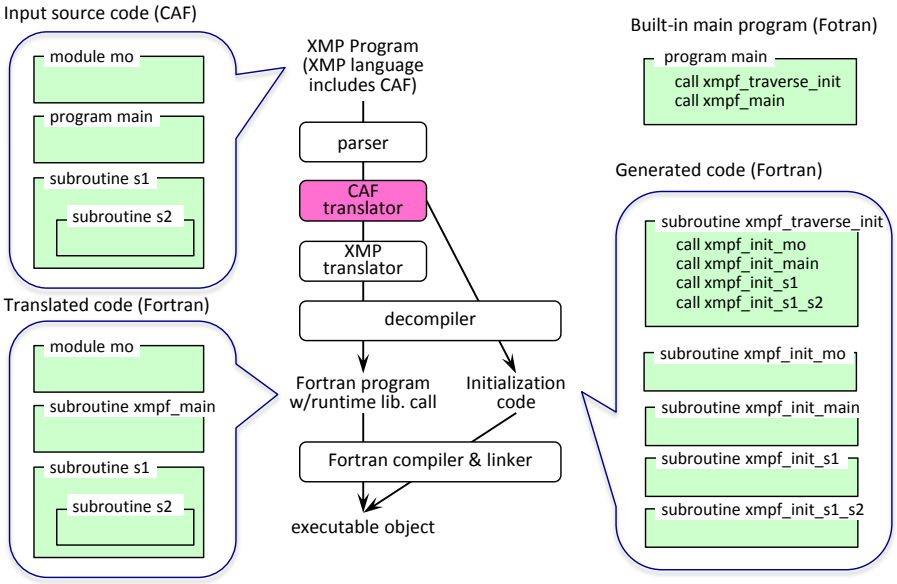


Fig. 1 XMP compiler and an example of coarray program compilation
CAF translator → coarray translator

3.2 Allocation and Registration

In order 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.
- The **runtime allocation (RA) method** allocates and registers a large memory for all static coarrays at the initialization phase. The RA method also 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, since 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*.

Using the RS and RA methods, static coarrays are initialized before 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, references to names of constants, and basic intrinsic functions, such as min/max and sum, are evaluated by constant folding. Since the size of the structure that contains allocatable or pointer components differs depending on the target compiler, the coarray translator obtains the necessary parameters to calculate the size of structures at build time.
- Then, the total size of static coarrays is allocated and the address and size are registered to the underlying communication library.
- In the second pass, the addresses of all of the coarrays are calculated to share the registered data. Due to the language specification, the sizes of the same coarray are the same among all images (nodes). Therefore, the offset from the base address of the registered data for each coarray can be the same among all images.

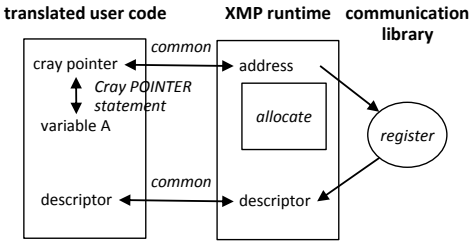
In the RS method, allocatable coarrays also share the registered memory. The total size of the memory to be registered should be specified with an environment variable by the user. In the RA method, the total size is fully calculated by the runtime library, and no information is required of the user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

In the CA method, the Fortran processor allocates each coarray, and the runtime library then 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 in a manner similar 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

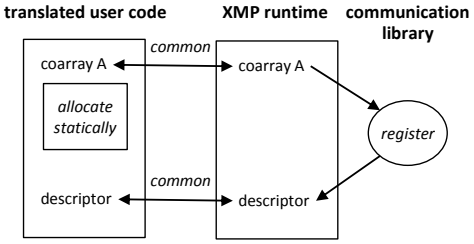
For the RS method, the runtime library has a memory management system for cutting out and retrieving memory for each allocation and deallocation of coarrays.

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



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

These methods are properly used by the underlying communication library. For GASNet, only the RS method is adopted because its allocation function can be used only once in the program. For MPI-3, the CA method is not suitable because frequent allocation and deallocation of coarrays cause expensive creation and freeing of MPI windows. In the case of 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 degradation of the Fortran compiler optimization.

3.3 PUT/GET Communication

In order to avoid disturbing the execution on the remote image, PUT and GET communications are always implemented 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 variables must be registered when allocated to be the target of RDMA communication. In contrast, since the local data, which is the source of PUT or the destination of GET, was not registered or linked to registered information, the data could not be the target of DMA communication and had to be communicated via the registered buffer.

When the local data is an entire coarray or a part of coarray, the coarray must be registered, and efficient DMA-RDMA communication can be made. Since the analysis at compile time is limited, we implemented the detector in the runtime library using binary-tree search, as follows.

1. When a chunk of coarray data is registered to the communication library, the runtime library adds the set of the local address and the size to a sorted table called `SortedChunkTable`. The sort key is the local base address of the data.
2. When a chunk of coarray data is deregistered from the communication library, the runtime library deletes the record in `SortedChunkTable`.
3. 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 registered 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, respectively, in `SortedChunkTable`.

If the communication data is large, then the cost of procedure 3 is relatively small and is worth using. If the data is small, then the buffering algorithm, as shown in Section 3.3.2, may be better.

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, respectively. Here, B should be large enough to ignore communication latency overhead and we use approximately 400 kilo-bytes by default. Unlike the case of MPI message passing, coarray PUT/GET communication requires only one local buffer for any number of other images. Both 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 obtain the contiguous length is shown in a previous paper [5].

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 show the algorithms using RDMA and local-buffering. Due to its strict condition, the DMA scheme is rarely used. In addition, this scheme is not always faster than the buffering scheme for Cases 2 and 3 because of the difference in the unit sizes. The advantage of Cases 2 and 3 is that the unit size is extended to a multiple of N_L by gathering a number of short contiguous data in the buffer, or by scattering from the buffer into a 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	$N_R \le B,\ N_R \le N_L$	N_R
	2	$N_L < N_R \le B$	N_R
	3	$N_L < B < N_R$	multiple of N_L ($\le B$)
	4	$B < N_R,\ B \le N_L$	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

3.3.3 Non-blocking PUT Communication

For higher performance, the PUT communication should be non-blocking, and the completion wait should be delayed until the end of the segment. Writing and reading the same remote data from the same image in the same segment appears to be a very rare case, as described in Section 2.4. However, this is difficult to detect with low cost. Since the subscripts and image indices are often variable expressions, the compiler rarely select non-blocking communication and usually generates safe but slow code. We do not have a reasonable solution for this issue.

In the current implementation, the user selects blocking or non-blocking for PUT communication at runtime with the environment variable.

3.3.4 Optimization of GET Communication

A reference 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 issue is that the result of the library function is an array value, which causes several memory copies. As a countermeasure, we optimized a specific but common case by the translator. If a coindexed object is only the right-hand side of an array assignment statement, then the entire assignment statement can be converted into a single library call. The above example satisfies this condition and so 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 is shown in Figure 3. One of three communication libraries is selected at the build time of the Omni compiler. The coarray runtime consists of three layered libraries. The **Fortran wrapper** mediates the arguments and the result value of the translated user program (written in Fortran) and the upper-layer runtime (ULR) (written in C). The **(ULR) library** performs the algorithms described above in this section. The **lower-layer runtime (LLR) library** abstracts the difference between the communication libraries, except for the memory management of coarray data.

3.4.1 Fortran Wrapper

Each set of Fortran wrapper procedures has a generic name and dozens of corresponding specific names. For example, the object code contains a call to a function with the generic name `xmpf_coarray_get_generic`. If the data is a two-dimensional array of the 16-byte complex type, the Fortran compiler selects the corresponding specific name `xmpf_coarray_get2d_z16` at compile time and generates the object code by calling a ULR function by the specific name.

The Fortran wrapper accepts Fortran array notations as the arguments and the result variable and converts these notations into structures that can be handled in

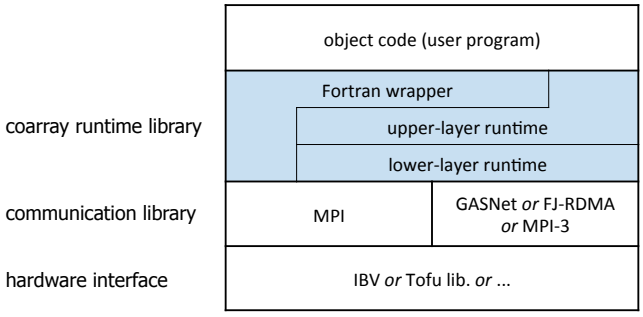


Fig. 3 Software stack for coarray features

a runtime library written in C. The Fortran wrapper also converts a C pointer to a Fortran pointer with the shape using the Cray pointer.

The Fortran wrapper calls ULR procedures basically and calls MPI library functions directly for collective communications.

3.4.2 Upper-layer Runtime (ULR) Library

The major role of ULR is performing the algorithms for coarray data allocation/registration (Section 3.2) and PUT/GET communications (Section 3.3). Additionally, for atomic communications caused by intrinsic subroutines `ATOMIC_DEFINE` and `ATOMIC_REF`, ULR calls the corresponding function of LLR after address calculation.

3.4.3 Lower-layer Runtime (LLR) Library

The LLR basically abstracts the difference between the communication libraries. The only exception is the allocation and registration of coarray data. Major functions are shown below.

- Functions to allocate and register coarray variables, and functions to register coarray variables that are already allocated. They are alternatively used in the RS and RA methods and in the CA method. Correspondingly, a set of functions to deregister and deallocate and a set of functions to deregister are provided.
- Fundamental functions for RDMA-DMA GET communication and DMA-RDMA PUT communication. It is assumed that both remote and local data are previously registered. Blocking and non-blocking can be switched.

- Functions corresponding to image control statements, atomic subroutines, and inquire functions.

The LLR also has the features for multi-dimensional data developed for the C implementation, which are not used in the Fortran implementation because this implementation is solved in ULR.

3.4.4 Communication Libraries

MPI-3 can be selected for all platforms on which it is implemented. Coarrays are registered and deregistered at the start and end point of the MPI window. Coarrays perform one-sided communication by `MPI_Put` and `MPI_Get` and are 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 performed only once on GASNet, the implementation allocates and registers a pool of memory, the size of which should be large enough to contain all static and allocatable coarrays. The XMP runtime should allocate and deallocate coarrays without using the Fortran library but using the memory manager constructed for the pool.

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

4 Evaluation

We evaluated the Omni XMP coarray compiler in the environments shown in Table 2.

4.1 Fundamental Performance

Using the EPCC Fortran Coarray micro-benchmark [6], we evaluated the ping-pong performance of PUT and GET communications compared with MPI_Send/Recv. The codes are briefly shown in Table 3.

Corresponding to the codes in Table 3, Figure 4 shows how data and messages are exchanged between two images or processes. In coarray PUT (a) and GET (b), inter-image synchronization is necessary for each end of the phases to make the passive

Table 2 Specifications of the computers 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
coarray	Omni XcalableMP 1.3.1	Omni XcalableMP 1.3.1	Omni XcalableMP 1.3.1
Fortran	Fujitsu Fortran 2.0.0	Fujitsu Fortran 2.0.0	Intel Fortran 16.0.4
MPI	Fujitu MPI 2.0.0	Fujitu MPI 2.0.0	Intel MPI 5.1.3
comm. layer	Tofu library	Tofu library	GASNet 1.24.2 (IBV-conduit, built with Intel compilers)

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.

image active and to make the active image passive. Whereas, in MPI message-passing (c) and (d), such synchronization is not necessary because both processes are always active. On the other hand, MPI message passing has its own overhead that coarray PUT/GET does not have. Since 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 communication. The overhead cost is not negligible when the data is small.

The result of the comparison 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 is used on HA-PACS. GET (a) and GET (b) use the code without and with the optimization described in Section 3.3.4,

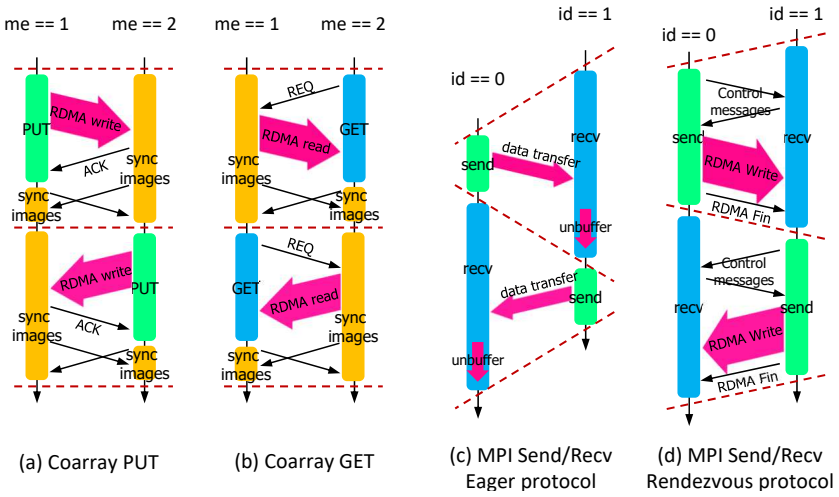


Fig. 4 pingpong-fig.pdf

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 GET (b) is the compile-time optimization level of the coarray translator described in Section 3.3.4.

The following was found regarding coarray PUT/GET communication.

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

However, on GASNet/HA-PACS (c), PUT and GET (b) were only approximately 60% of the bandwidth of MPI rendezvous for a large amount of data. 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). Although the latency is larger than that for MPI for all data sizes, the difference is smaller than in the other cases. At a data size of 8B, the latency of PUT is 2.93 μs and 2.1 times larger than the one of MPI while 5.73μs and 3.7 times larger for the case of MPI-3 (d).

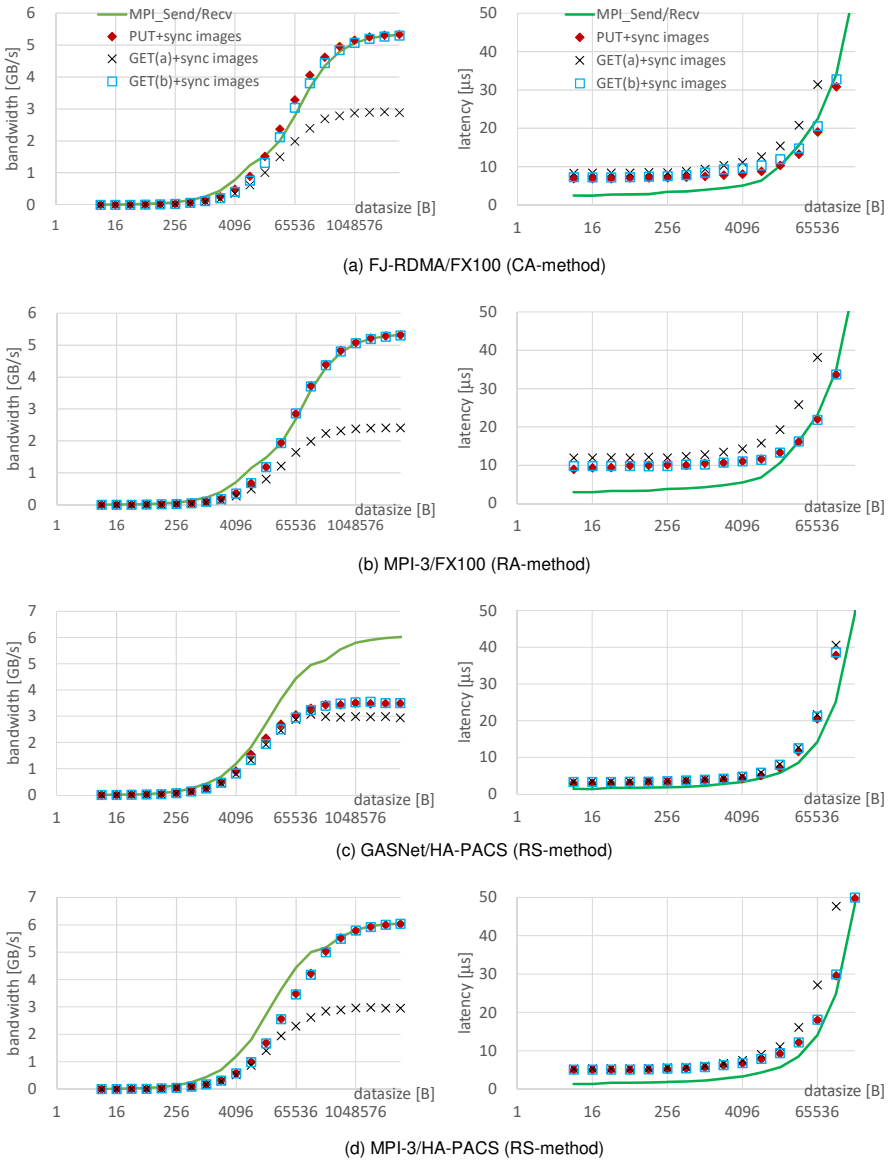


Fig. 5 Ping-pong performance on Fujitsu PRIMEHPC FX100 and HA-PACS/TCA

Effect of GET optimization For all ranges in all cases, GET (a) has a smaller bandwidth and a 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 GET (b). We found GET (a) caused two extra memory copies. One copy performs the array assignment by the Fortran library, and the other copy is from the communication buffer to the result variable of the array function `xmpf_coarray_get_generic`. The optimization described in Section 3.3.4 eliminated these two data copies.

The large latency of coarray PUT/GET communication is problematic. In the next subsection, we discuss how this problem 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 in coarray PUT communication. The principle is shown in Figure 6.

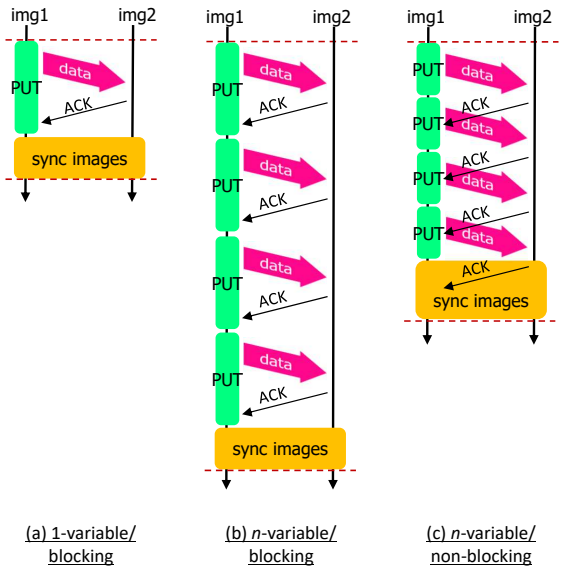


Fig. 6 Blocking and non-blocking PUT communications

Figure 6(a) shows 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, as shown in (b), are closer to actual applications. In addition, coarray one-sided communication 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. The two original graphs are the same as those of Figure 5(a). Four other graphs display the results of the eight-variable ping-pong program, which repeats the ping phase, sending eight individual variables from one to the other in order, and, similarly, the pong phase in the opposite direction. Each block size indicates the size of variables, and latency includes the time for eight variables.

The following was found from the results:

- Non-blocking PUT significantly improves the latency of PUT communication. From 8B to 8kB, the latency of non-blocking PUT communication is 4.63 times faster on average than blocking PUT. Compared to the original PUT, from 8B to 8kB, it performs communication eight times for a period of time 2.11 times longer, on average. Hiding completion wait behind communication (Figure 6(c)) greatly improves the performance.

- Reduction of synchronization (Figure 6(b)) itself does not improve the performance. Compared to the original blocking PUT, eight-variable blocking PUT has 9.5 to 10.1 times larger latency for a data set that is eight times larger.

- Unless data size exceeds approximately 8kB, the latency of non-blocking PUT does 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 on non-blocking for latency hiding. The eager protocol, including the unbuffering process of the receiver, appears not to be suitable for non-buffering.

- Non-blocking coarray PUT outperforms MPI eager message-passing, except for very fine grain data. The latency of eight-variable non-blocking PUT is -9% to 54% and 18% to 61%, as compared to eight-variable blocking and non-blocking MPI eager, respectively. At only two plots for 8B and 16B, the non-blocking PUT is 4% and 9% slower than the values for blocking MPI. Otherwise, non-blocking PUT is faster than MPI eager, and the more block size, the larger difference in the latency.

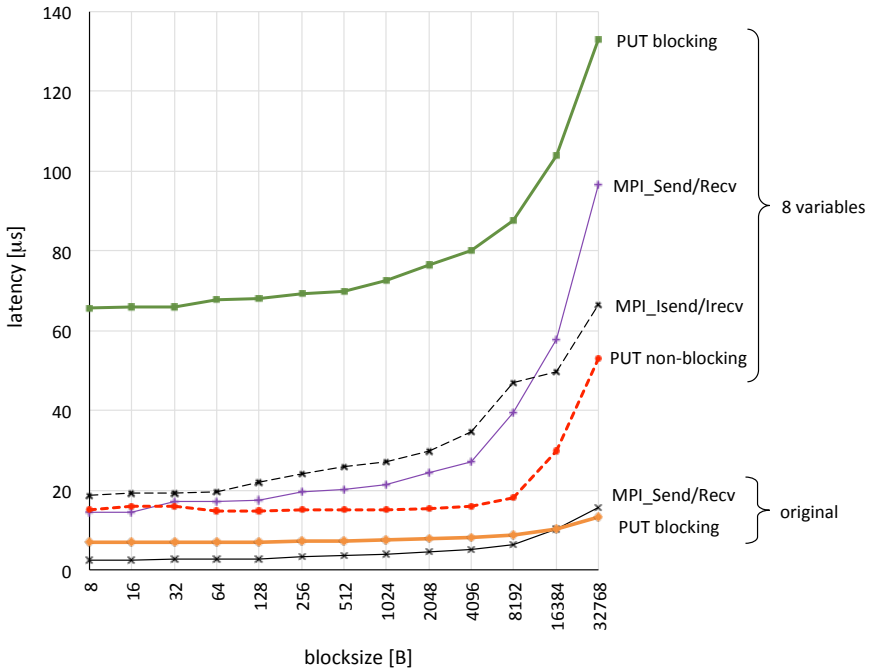


Fig. 7 Eight-variable ping-pong latency on PRIMEHPC FX100

4.3 Application Program

The Himeno benchmark is a part of the 2D Poisson equation solver using the Jacobi iteration method [7]. The MPI version of the Himeno benchmark is a strong scaling program distributing up to three-dimensional nodes. The K computer shown in Table 2 was used in this evaluation.

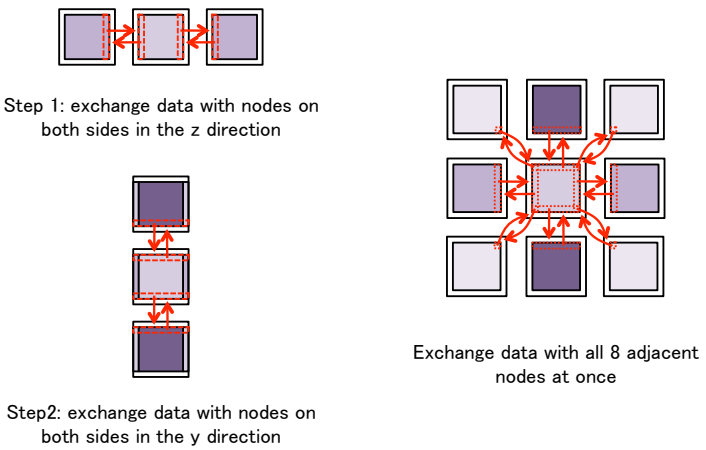
4.3.1 Coarray Version of the Himeno Benchmark

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

MPI/original The original MPI version of Himeno benchmark was used as a two-dimensionally distributed in the y and z axes. The x axis was automatically SIMD-vectorized by the Fortran compiler. The program executes the computation and communication parts repetitively. The communication part consists of two steps: z-axis direction communication and 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 two-step communication was replaced by non-blocking scrambled communication, as shown in Figure 8(b). With this replacement, the number of communications increases from four to eight per node, but all communications become independent and can be non-blocking.

Coarray PUT/non-blocking The communication pattern is the same as that 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 a reference 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 the 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 at 76% of the measurement points of the graph. On 1,024 nodes, PUT non-blocking is 1.2%, 27%, and 42% faster than MPI original for sizes M, L, and XL, respectively.
- As a result of analyzing the contents of elapsed time, it was confirmed that the difference in the performance is caused by the difference in communication time.

As shown in (b) and (c), the communication times of PUT non-blocking are 56% and 51% of those of MPI blocking on 256 nodes on L and XL Himeno sizes, respectively.

- MPI non-blocking is not always faster than the MPI original. The effect of non-blocking seems to be limited in MPI.

4.3.3 Productivity

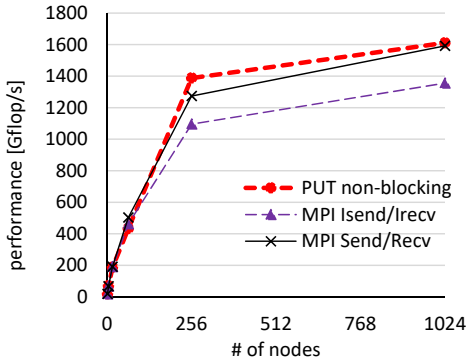
Table 4 compares the scale of the source codes. The following features can be found.

- PUT blocking requires fewer characters for programming, especially in subroutines `initcomm` and `initmax`. The MPI programmer must describe the Cartesian coordinates 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 represents 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)`.
- The Fortran statement of the MPI program tends to be longer than that of coarray. Since, 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 the coarray program is more compact than the MPI program for each statement. MPI library functions often require long sequences of arguments.

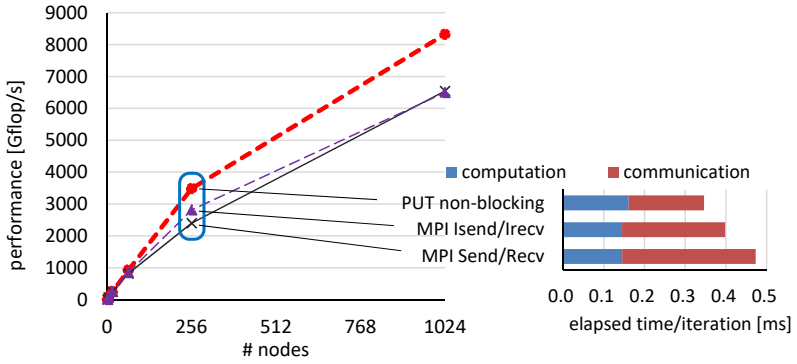
Table 4 Source code scales for the Himeno benchmark

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

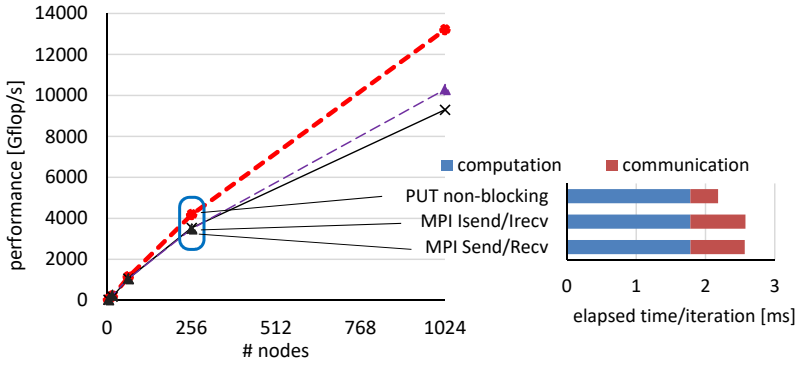
LOC: Number of lines of codes excluding comment and empty lines.
SOC: Number of Fortran statements, which may span multiple lines.
chars: Number of characters excluding those in comment lines.



(a) Himeno Size-M (256x128x128)



(b) Himeno Size-L (512x256x256)



(c) Himeno Size-XL (1024x512x512)

Fig. 9 himeno-graph-r2.pdf

5 Related Work

The University of Rice has implemented coarray features with their own extension called CAF2.0 [8]. CAF2.0 is a source-to-source compiler based on the ROSE compiler. GASNet is used as its communication layer. Similarly to our RS and RA methods, the Cray pointer is used to pass the data allocated in C to Fortran. Houston University developed UH-CAF on the Open64-base OpenUH compiler [9]. UH-CAF supports the coarray features defined in the Fortran 2008 standard. As the communication layer, GASNet and ARMCI can be used selectively. OpenCoarrays is an open-source software project [10]. OpenCoarrays is a library that can be used with GNU Fortran (gfortran) V5.1 or later and supports the coarray features specified in Fortran 2008 and a part of Fortran 2018. As the communication layer, MPICH and GASNet can be used selectively. In the vendors, Cray and Intel fully support and Fujitsu partially supports the coarray features specified in Fortran 2008.

In the latest Fortran standard, Fortran 2018, a subset of coarrays is referred to as a team. It is similar to the executing images in the term of XMP, but does not affect the parallel execution among images.

While non-blocking PUT communication is effective, non-blocking GET communication is difficult to put into practical use because the acquired data is used immediately. Cray has the directive extension for prefetching a remote coarray corresponding to the GET communication.

Coarray C++ is a coarray implementation in C++. The coarray features are implemented with the template library, unlike XMP/C, which is based on the C language.

6 Conclusion

This chapter described the coarray features in the context of XMP and the characteristic implementation of the coarray translator.

For memory allocation and registration, the RS, RA, and CA methods were implemented corresponding to the communication library GASNet, FJ-RDMA, and MPI-3.

For the coarray PUT and GET communications, DMA and four buffering methods were described. The effect of the non-blocking PUT communication was analyzed, and the knowledge is used to make the coarray version of the Himeno benchmark from the original MPI version. The measurement results on 1,024 nodes of the K computer, the coarray version is 27% and 42% faster than the original MPI version for Himeno sizes L and XL, respectively. The effect of the optimization of GET communication was also obvious on the ping-pong benchmark on HA-PACS/TCA and Fujitsu PRIMEHPC FX100.

As an evaluation of productivity, the coarray program uses fewer than half as many characters as the MPI message passing program to write the same algorithm as the Himeno benchmark.

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.

The results were obtained in part using the K computer at the RIKEN Advanced Institute for Computational Science.

References

1. XcalableMP Language Specification, http://xcalablemp.org/specification.html
2. John Reid, JKR Associates, UK. Coarrays in the next Fortran Standard. ISO/IEC JTC1/SC22/WG5 N1824, April 21, 2010.
3. ISO/IEC TS 18508:2015, Information technology – Additional Parallel Features in Fortran, Technical Specification, December 1, 2015.
4. Omni Compiler Project. http://omni-compiler.org
5. Hidetoshi Iwashita, Masahiro Nakao, Mitsuhisa Sato. 2015. Preliminary Implementation of Coarray Fortran Translator Based on Omni XcalableMP. PGAS2015, Proceedings of 9th International Conference on PGAS Programming Models, pp.70-75, Washington, D.C., USA. September, 2015.
6. EPCC Fortran Coarray micro-benchmark suite. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-co-array-fortran-micro
7. Himeno Benchmark. http://accc.riken.jp/en/supercom/himenobmt/
8. John Mellor-Crummey, Laksono Adhianto, William N. Scherer III, and Guohua Jin. PGAS'09, 3rd Conference on Partitioned Global Address Space Programming Models. Ashburn, VA. October, 2009.
9. Deepak Eachempati, Hyoung Joon Jun, and Barbara Chapman. An Open-Source Compiler and Runtime Implementation for Coarray Fortran. PGAS'10, 4th Conference on Partitioned Global Address Space Programming Models, No.13. 2010.
10. Alessandro Fanfarillo, Tobias Burnus, Valeria Cardellini, Salvatore Filippone, Dan Nagle, and Damian Rouson. 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. 2014.