To: J3

From: Hidetoshi Iwashita

Subject: Use cases of generic coarray dummy arguments

Date: 2024-December-23

References: [24-139r2](https://j3-fortran.org/doc/year/24/24-139r2.txt), [24-168](https://j3-fortran.org/doc/year/24/24-168.txt), [24-187](https://j3-fortran.org/doc/year/24/24-187.txt), [24-188r1](https://j3-fortran.org/doc/year/24/24-188r1.txt)

# 1. Introduction

In 24-168 (3), we proposed the following constraint and specification in 24-139r2 to be removed:

C8nn A generic dummy argument shall not be a coarray.

sNN A generic dummy argument cannot be a coarray.

In response to this, a straw vote was taken at the October 2024 meeting on whether to keep or remove these, but the result was very close (24-188r1).

We believe that generic dummy arguments for coarrays are necessary. In this paper, we will introduce two use cases for generic coarray dummy arguments and show that they are indispensable.

# 2. Use cases

## 2.1 Example of application: stencil communication

Himeno benchmark solves the Poisson equation by Jacobi's iterative method [1], whose communication pattern is as follows.

DO loop=1,nn

computation

z+ and z- direction communication (sendp3)

y+ and y- direction communication (sendp2)

x+ and x- direction communication (sendp1)

all reduce

ENDDO

In this section, the sendp2 above is introduced (1), expanded to type-generic (2), and changed from a MPI program to a coarray program (3). An example of the generic dummy argument for coarray is shown in (3).

### (1) Original subroutine sendp2

List 1 is the original code of the sendp2 above, added inline comments for explanation. Its communication pattern is demonstrated in Figure 1.

List 1: Subroutine sendp2 in the Himeno benchmark

module pres

real(4),dimension(:,:,:),allocatable :: p ! Main data

end module pres

module others ! (Irrelevant names were deleted.)

integer :: mimax,mjmax,mkmax !! the allocated size of p

integer :: imax,jmax,kmax !! the size actually used in p

end module others

module comm ! (Irrelevant names were deleted.)

integer :: ndx,ndy,ndz !! 3-D node (image) size

integer :: npx(2),npy(2),npz(2) !! node numbers of adjacent nodes

integer :: ijvec,jkvec,ikvec !! representing the shape of communication data

integer :: mpi\_comm\_cart !! communicator defined by MPI\_Cart\_create

end module comm

subroutine sendp2()

use pres

use others

use comm

implicit none

include 'mpif.h'

integer :: ist(mpi\_status\_size,0:3),ireq(0:3)=(/-1,-1,-1,-1/)

integer :: ierr

!

call mpi\_irecv(p(1,1,1), & ! initial address of receive buffer

1, & ! number of elements

ikvec, & ! representing shape [mimax, 1, mkmax]

npy(1), & ! source node (neighboring in y- direction)

2, & ! tag

mpi\_comm\_cart, & ! communicator

ireq(3), & ! communication request (intent(out))

ierr)

call mpi\_irecv(p(1,jmax,1), &

1, &

ikvec, &

npy(2), & ! source node (neighboring in y+ direction)

1, &

mpi\_comm\_cart, &

ireq(2), &

ierr)

call mpi\_isend(p(1,2,1), & ! initial address of send buffer

1, &

ikvec, &

npy(1), & ! destination node (neighboring in y- direction)

1, &

mpi\_comm\_cart, &

ireq(0), &

ierr)

call mpi\_isend(p(1,jmax-1,1), &

1, &

ikvec, &

npy(2), & ! destination node (neighboring in y+ direction)

2, &

mpi\_comm\_cart, &

ireq(1), &

ierr)

call mpi\_waitall(4, &

ireq, &

ist, &

ierr)

return

end subroutine sendp2



Figure 1. MPI message passing in List 1.

(2) An example of making sendp2 generic

The original data type of Himeno benchmark is REAL(4). List 2 shows a generic version of sendp2, which expanded the data type to all kinds of REAL type supported by the processor.

List 2: Generic subprogram version of sendp2

module others

integer :: mimax,mjmax,mkmax

integer :: imax,jmax,kmax

end module others

module comm

integer :: ndx,ndy,ndz

integer :: npx(2),npy(2),npz(2)

integer :: ijvec,jkvec,ikvec

integer :: mpi\_comm\_cart

end module comm

module himeno\_sendp2\_mpi

use comm

implicit none

contains

generic subroutine sendp2(p)

real(\*),dimension(mimax,mjmax,mkmax) :: p

include 'mpif.h'

integer :: ist(mpi\_status\_size,0:3),ireq(0:3)=(/-1,-1,-1,-1/)

integer :: ierr

call mpi\_irecv(p(1,1,1), 1, ikvec, npy(1), &

2, mpi\_comm\_cart, ireq(3), ierr)

call mpi\_irecv(p(1,jmax,1), 1, ikvec, npy(2), &

1, mpi\_comm\_cart, ireq(2), ierr)

call mpi\_isend(p(1,2,1), 1, ikvec, npy(1), &

1, mpi\_comm\_cart, ireq(0), ierr)

call mpi\_isend(p(1,jmax-1,1), 1, ikvec, npy(2), &

2, mpi\_comm\_cart, ireq(1), ierr)

call mpi\_waitall(4, ireq, ist, ierr)

return

end subroutine sendp2

end module himeno\_sendp2\_mpi

Comparing to List 1, sendp2 was modified as follows.

1. Change sendp2 from an external subprogram to a module subprogram with the GENERIC prefix,
2. Change p from a module variable to a generic dummy argument with REAL(\*) type specifier.
3. Change dummy argument p from an allocatable array to an explicit-shape array (optional).

The important point here is that in order to make the subroutine generic, at least one dummy argument that is declared in a generic type declaration statement is necessary (item 2). Item 3 doesn't have much significance in sendp2, but since there is a high-load computational loop in the jacobi subroutine which calls sendp2, higher performance can be expected by making the argument p of jacobi and of all the procedures it calls explicit-shape.

### (3) An example of changing sendp2 to use coarray

List 3 shows a coarray version of sendp2 modified from the code in List 2. In this example, the generic dummy argument p must be a coarray because it is referenced as coindexed objects.

List3: A coarray version of sendp2 with a generic coarray dummy argument

module others

integer :: mimax,mjmax,mkmax

integer :: imax,jmax,kmax

end module others

module comm

integer :: ndx,ndy,ndz

integer :: ihalo,jhalo,khalo ! new variables

end module comm

module himeno\_sendp2\_coarray

use comm

implicit none

contains

generic subroutine sendp2(p)

real(\*),dimension(mimax,mjmax,mkmax), codimension[ndx,ndy,\*] :: p

integer :: me(3)

me = this\_image(p)

sync all

if (me(2)>1) then

p(:, jhalo, :)[me(1), me(2)-1, me(3)] = p(:, 2 , :)

end if

if (me(2)<ndy) then

p(:, 1 , :)[me(1), me(2)+1, me(3)] = p(:, jmax-1, :)

endif

sync all

return

end subroutine sendp2

end module himeno\_sendp2\_coarray



Figure 2. Coarray assignment in List 3.

Figure 2. shows the communication caused by the two coarray assignment statements in List 3. Instead of the message passing used in Lists 1 and 2, one-sided communication is used in List 3. The newly-appeared variable jhalo is the index of the halo in the y+ direction on the image [me(1), me(2)-1, me(3)]. This is a pre-calculated value, and if the data is equally allocated to all images, it will be the same value as jmax.

## 2.2 Example of creating a library procedure: CO\_BROADCAST

Generic subprograms are suitable to write Fortran intrinsic procedures and intrinsic module procedures with generic names, at least for the entry layers of those procedures. The same can be said for highly generic user-defined procedures. This is because generic subprograms can achieve both high performance and high productivity for developing highly generic procedures. And then, coarray dummy arguments are necessary to use coarray one-to-one communication.

List 4 shows an example of writing the CO\_BROADCAST intrinsic subroutine assuming the argument is coarray. Using generic type declaration, dummy argument A can be any intrinsic type with any kind supported by the processor and any rank. For the sake of simplicity, A cannot be a derived type and arguments STAT and ERRMSG are omitted. Figure 3 displays the communication and synchronization in this program. We assume that the function MAX\_RANK proposed in 24-187 is included in the module ISO\_FORTRAN\_ENV.

List 4: CO\_BROADCAST specialized for coarrays as dummy arguments

01 GENERIC SUBROUTINE co\_broadcast\_coarray(a, source\_image)

02 USE iso\_fortran\_env

03 IMPLICIT NONE

04 TYPE(INTEGER(\*),REAL(\*),COMPLEX(\*),LOGICAL(\*),CHARACTER(kind=\*)), &

05 RANK(0:MAX\_RANK(1)), INTENT(INOUT):: a[\*]

06 INTEGER, INTENT(IN):: source\_image

07 INTEGER:: n\_images, dist, i

08 INTEGER:: this\_img, that\_img, this\_id, that\_id

09

10 n\_images = num\_images()

11 this\_img = this\_image()

12 this\_id = modulo(this\_img - source\_image, n\_images)

13

14 SYNC ALL

15 dist = 1

16 DO

17 IF (this\_id < dist) THEN ! This image is a sender.

18

19 !-- find receiver or exit the loop

20 that\_id = this\_id + dist

21 IF (that\_id >= n\_images) EXIT ! This image exits the loop.

22 that\_img = modulo(this\_img + dist - 1, n\_images) + 1

23

24 !-- send the data

25 a[that\_img] = a

26

27 !-- 1-by-1 synchronization

28 SYNC IMAGES (that\_img)

29

30 ELSE IF (this\_id < 2 \* dist) THEN ! This image is a receiver.

31

32 !-- find sender

33 that\_id = this\_id - dist

34 that\_img = modulo(this\_img - dist - 1, n\_images) + 1

35

36 !-- 1-by-1 synchronization

37 SYNC IMAGES (that\_img)

38

39 END IF

40 dist = 2 \* dist

41 END DO

42 SYNC ALL

43

44 END SUBROUTINE co\_broadcast\_coarray



Assumed that the number of images is 7, and the value of source\_image is 3.

Figure 3. Broadcast communication and synchronization pattern

Subroutine co\_broadcast\_coarray in List 4 assumes that the actual argument corresponding to a is a coarray. If not, a coarray communication buffer, for example as shown in List 5. In this case, dynamic coarray allocation and round-trip full data copying may cause a significant overhead cost. So, the processor should select co\_broadcast\_coarray if the actual argument is coarray, and co\_broadcast\_noncoarray otherwise.

List 5: CO\_BROADCAST for non-coarrays using the subroutine in List 4.

01 GENERIC SUBROUTINE co\_broadcast\_noncoarray(a, source\_image)

02 IMPLICIT NONE

03 TYPE(INTEGER(\*),REAL(\*),COMPLEX(\*),LOGICAL(\*),CHARACTER(kind=\*)), &

04 RANK(0:MAX\_RANK), INTENT(INOUT):: a

05 INTEGER, INTENT(IN):: source\_image

06 TYPE(REAL), ALLOCATABLE, DIMENSION(:):: tmp[:]

07

08 ALLOCATE (tmp(SIZE(a))[\*])

09 tmp(:) = RESHAPE(a, [SIZE(a)])

10 CALL co\_broadcast\_coarray(tmp, source\_image)

11 a = RESHAPE(tmp, SHAPE(a))

12 RETURN

13 END SUBROUTINE co\_broadcast\_noncoarray

# 3. Discussions

In this section, the need for a generic coarray dummy argument is discussed.

## 3.1 Execution performance

Coarray one-to-one communication has the potential to achieve high performance through zero-copy communication by implementing it as one-sided communication using DMA (Direct Memory Access) and RDMA (Remote DMA) provided by communication layers such as GASNet [2]. In order to apply such high performance to dummy arguments, it must be declared as a coarray to receive the global address and other information (if any) from the corresponding coarray actual argument. This is true regardless of whether the subprogram is generic or not.

## 3.2 Programming and maintenance

It is doubtful whether adding such a constraint that only apply to generic subprograms, and not to non-generic subprograms, will lead to simplification. We think a typical programmer would first design an algorithm for a specific type/kind/rank and then expend it to a generic type/kind/rank. If the programmer encounters the constraint when expanding it to generic, they must either give up to make it generic, or go back to reconsider the algorithm.

We don't think the idea of "setting it to constraint for now and then releasing it later" is appropriate in this case. Programs that get around the constraint in strange ways will become established as assets.

# Acknowledgments

We would like to thank to John Reid for his detailed review of the program codes.

# References

[1] Himeno benchmark. <https://i.riken.jp/en/supercom/documents/himenobmt/>

[2] Iwashita, H., Nakao, M. (2021). Coarrays in the Context of XcalableMP. In: Sato, M. (eds) XcalableMP PGAS Programming Language. Springer, Singapore. <https://doi.org/10.1007/978-981-15-7683-6_3>