# Co-Array Collectives: Refined Semantics for Co-Array Fortran

Matthew J. Sottile[1], Craig E Rasmussen, and Richard L. Graham

Los Alamos National Laboratory,
Los Alamos NM 87545, USA
{matt, rasmussn, rlgraham}@lanl.gov

**Abstract.** Co-array notation provides a compact syntax for programming parallel programs. Co-array Fortran (CAF) introduced and implements this notation, and CAF is currently proposed as an extension to the Fortran language standard. We believe that co-array notation requires a revised semantic definition beyond that specified by CAF for both pragmatic reasons within Fortran and to make the notation attractive for incorporation into other programming languages. The revised semantics make the language model easier to understand and reduces the potential for programmer error. Furthermore, these revised semantics allow CAF to be extended to capture collective operations in co-array notation.

## 1 Introduction

Co-Array Fortran (CAF) is a simple, parallel extension to Fortran [3]. For CAF, a single program is replicated `p` number of times in SPMD fashion (Single Program, Multiple Data). Each copy of the program (referred to as an *image*) executes asynchronously with its own set of data objects. The co-array notation extends the normal array syntax with an additional array dimension called the *co-dimension*. The local portion of a co-array is denoted with the normal parentheses while the co-dimension is specified with square brackets.

For example, if `a` and `b` both have local rank 1, local size 100, and a co-rank 1, they are declared as

```
integer :: a(100)[*], b(100)[*]
```

The co-dimension is declared to be assumed size, *i.e.* `[*]`, as the co-size depends on the number of participating images (a compile time or run-time parameter).

Indexing in the co-dimension is the same as in the local dimension. For example, given constants `i` and `j` that are identical on all images, `a(1)[i] = b(1)[j]` states that the first element of the array `b` on image `j` is to be copied to the first element of the array `a` on image `i`. Because this assignment, as written, would be executed on every image, it is actually illegal (a non-conforming program) because concurrent writes into the same storage location are not allowed.

To emphasis this point, consider the simple assignment, `a(1)[i] = 3`. Without additional context, all images will execute this statement and cause the

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part II, LNCS 3992, pp. 945–952, 2006.

constant 3 to be stored in the first element of `a` on image `i`. As this is a concurrent write and therefore illegal, the programmer must embed the statement within other control structures to ensure that it adheres to the rules established for CAF. One must explicitly declare *which* image is to cause the store to occur, e.g.,

```
if (this_image() == j) a(1)[i] = 3
sync_all()
```

If image `j` is not equal to `i`, image `j` *puts* the value 3 to image `i`.

We suggest that this is an unfortunate definition for the result of the CAF assignment operator. Assuredly all images must execute an assignment statement (unless restricted by an if statement), but there is no reason all images must *cause* data to be transferred to image $i$. Certainly in this case, the value of `a(1)[i]` will be 3 after the `sync_all` is executed, no matter which image caused the store to occur. (The intrinsic function `sync_all` is a barrier that all images must reach before any image may continue execution.)

In particular, this execution model causes problems if array sections are allowed in the co-dimension (co-sections). For example, the assignment statement `a(1)[:] = 3` would require that *all* images store 3 into co-memory on *all* images. Indeed, although co-sections were allowed in the original Reid and Numrich definition [3], they were removed in a recent CAF specification [4]. One reason they were removed, is that they are not present in CAF implementations and limiting co-subscripts to scalars simplifies the language specification [5]. The goal of this paper is to examine the ramifications of permitting co-sections, with the hope that they can be restored to the language.

Specifically, we propose that one-sided *get* semantics be adopted in all assignment statements involving co-array variables, making the image that "owns" the memory the only one allowed to initiate a write operation to it. This simplifies the language specification and automatically satisfies the requirement that concurrent stores to the same memory location should not occur. In this paper we examine the consequences of this change. The rules defining the result of a co-variable assignment statement are described in Section 3 and then applied to collective operations in Section 4. Section 5 contains two programming examples employing these changes and Section 6 summarizes the paper.

## 2 Refined CAF Assignment Operators

In this section we propose a set of semantic rules for co-array notation that define the result of a co-variable assignment statement. These rules are intended to provide the compiler with as much freedom as reasonable to generate efficient parallel code by removing the burden of explicit delegation of message passing initiation from the user. Like CAF, as defined in [4], these rules are such that the compiler is able to make decisions solely on the basis of an analysis of the program as a sequential entity, without requiring cross image analysis to perform messaging optimizations. In addition, they allow the use of array sections in the co-dimension.

The Fortran grammar defines an assignment statement as *variable* = *expr* [2]. The first rule describes the result of a co-array expression: For any co-array expression `b(:)[begin:end:stride]` executed on any image `i`, the result will be as if a temporary rank 2 array `tmp` were created on `i` and initialized as,

```
do k = begin, end, stride
  do j = 1, nrows
    tmp(j,k) = b(j)[k]
  end do
end do
```

This rule generalizes to other co-array ranks. In general, the rank of the temporary array will be equal to the total rank (the sum of the local rank and the co-rank) of the co-array. Note that the elements of `tmp` are obtained using get semantics.

The second rule defines the result of an assignment statement if the variable is a co-array and is sectioned (using subscript triplet notation) in the co-dimension. We define this as a parallel assignment statement. Consider `a(:)[begin:end:stride] = tmp(:)`, where `tmp` is a local array and may be a temporary array resulting from the evaluation of a co-array expression, as defined above (note the change in the rank of the temporary). The results of the assignment statement, when executed on all images, is

```
do i = begin, end, stride
  if (this_image() == i) then
    a(:)[i] = tmp(:)
  end if
end do
```

This rule also generalizes to other co-array ranks. In general, the local rank of the co-array variable must be equal to the rank of the array temporary and their shapes must conform.

In addition, we propose that a *put* assignment operator be provided. This would allow third party transfers, not allowed with get semantics. While not intending to provide guidance for the syntax of a put assignment, the following,

```
if (this_image() == 2) then
  a(1)[1] .recvs. b(1)[3]
end if
```

could be adopted to cause image two to copy memory from image three to image one. The same code segment with standard assignment syntax,

```
if (this_image() == 2) then
  a(1)[1] = b(1)[3]
end if
```

results in no change to the state of the program because of the implied `if (this_image() == 1)` with the standard assignment operator.

These are the only two changes proposed for CAF: 1. Get semantics as embodied in the second rule above; and 2. a put assignment statement. Nothing has been removed from the language. Only a clear distinction between the syntax for the two assignment statements, thus removing the overloading of the assignment operator. Furthermore, this removes ambiguity with respect to the meaning (get vs. put) of co-array notation when it appears on different sides of the `=` operator. With serial execution this distinction is unimportant. However, this is not true for parallel execution (multiple images). The put assignment (with no implied if statement) can lead to unnecessary and repetitious memory transfer, *.e.g.,* `a(1)[1] = 3`, and concurrent stores. Therefore, it is suggested that the distinguishing syntax is helpful to the programmer.

## 3 Collective Assignment Statements

Given the rules specified in Section 2, we can now enumerate the outcome of the 16 possible combinations (of array subscript versus array section) that can occur in the assignment statement `a(i)[j] = b(k)[l]`. This is done in order to insure that the results of any statement containing a co-section, *i.e.* `[:]`, is well understood.

The examples are shown in two columns: the left-hand column is written using array section notation in both the local and the co-dimensions, while the right-hand column is written using it only for the local dimension. The statements all assume that `me = this_image()`, `p = num_images()`, and that the local size of the arrays equals `p`.

Simple scalar and array assignment are well understood, as can be seen below:

```
a(1)[1] = b(3)[3]   ;   if (me == 1) a(1) = b(3)[3]
a(:)[1] = b(3)[3]   ;   if (me == 1) a(:) = b(3)[3]
a(:)[1] = b(:)[3]   ;   if (me == 1) a(:) = b(:)[3]
```

Likewise are the scalar and array broadcasts:

```
a(1)[:] = b(3)[3]   ;   a(1) = b(3)[3]
a(:)[:] = b(3)[3]   ;   a(:) = b(3)[3]
a(:)[:] = b(:)[3]   ;   a(:) = b(:)[3]
```

Broadcast statements execute on all images, with result that the same data are stored in the local portion of `a` on each image. Strictly speaking, these are not broadcasts, as the assignment has get semantics, though the implementation may be a broadcast by recognizing the array section in the co-dimension.

The gather collectives read data from all images and store the results either on one image (gather) or on all images (allgather).

```
a(:)[1] = b(3)[:]   ;   if (me == 1) then  ! gather
                          do i = 1, p; a(i) = b(3)[i]; end do
                        end if
a(:)[:] = b(3)[:]   ;   do i = 1, p;  a(i) = b(3)[i];  end do
```

There is no collective assignment statement corresponding to a scatter operation. It would have to be expressed explicitly:

```
a(1)[me] = b(me)[3]
```

There are several statements (of the 16 combinations) that are well defined but result in illegal assignments:

```
a(1)[1] = b(:)[3]   ;   if (me == 1) a(1) = b(:)[3]
a(1)[1] = b(3)[:]   ;   if (me == 1) a(1) = b(3)[:]
a(1)[1] = b(:)[:]   ;   if (me == 1) a(1) = b(:)[:]
a(:)[1] = b(:)[:]   ;   if (me == 1) a(:) = b(:)[:]
a(1)[:] = b(:)[3]   ;   a(1) = b(:)[3]
a(1)[:] = b(3)[:]   ;   do i = 1, p;  a(1) = b(3)[i];  end do
a(1)[:] = b(:)[:]   ;   do i = 1, p;  a(1) = b(:)[i];  end do
a(:)[:] = b(:)[:]   ;   do i = 1, p;  a(:) = b(:)[i];  end do
```

All of these statements could be made legal by increasing the rank of `a` by one, or by a reduction of the right-hand side, *.e.g.,* `a(1)[:] = sum(b(:)[3])`.

## 4 Examples

Two algorithms are implemented using co-arrays. The first example requires a high degree of synchronization while the second does not. These examples were chosen to compare and contrast the differences between the current CAF definition of the assignment operator, with that of our proposed definition.

### 4.1 Maximum Value of a Co-Array

The original Reid and Numrich Co-Array Fortran paper [3] contains several examples. One in particular, an algorithm that finds the maximum value of a co-array, was chosen because it highlights many important points. It should be noted that this example is inefficient and therefore, the actual CAF intrinsic function `CO_MAXVAL` is preferred.

Consider the following subroutine that will return the maximum value of co-array `a` in the co-scalar `great`:

```
subroutine greatest(a, great)  ! find maximum value of a(:)[*]
  real, intent(in)  :: a(:)[*]
  real, intent(out) :: great[*]
  great[:] = maxval(a(:))      ! all calculate local maxima
  sync_all()                   ! wait for all to calc local max
  great[1] = maxval(great[:])  ! gather to 1, then find maximum
  sync_all()                   ! wait for 1 to calc global max
  great[:] = great[1]          ! all collect the results
  sync_all()                   ! wait for all to finish
end subroutine greatest
```

The algorithm first computes local maxima on all local portions of `a`. It then gathers the local maxima to image one, where the global maximum is calculated. This global maximum is then copied by the other images.

Note the three synchronization calls in the example. The second is required because all images (other than one) must wait for the global maximum to be calculated before getting a copy of it. The code section containing the outer `sync_all` calls could be replaced by:

```
sync_all()                    ! wait for all to finish
great[1] = maxval(great[:])   ! gather to 1, then find maximum
if (this_image() == 1) then
  great[:] .recvs. great[1]   ! broadcast the results to all
end if
sync_all()                    ! wait for all to finish
```

This implementation using puts requires one less synchronization call because image one broadcasts the global maximum, only after it has calculated it. This example points out subtle differences between put and get semantics. Different control flow and synchronization patterns may be required, as demonstrated.

The relevant code segment implementing `greatest` using the original CAF execution model is shown below:

```
real :: work(num_images())
great = maxval(a(:))          ! all calculate local maxima
sync_all()                    ! wait for all to calc local max
if (this_image(great) == 1) then
  do i = 1, num_images()      ! gather local maxima
    work(i) = great[i]
  end do
  great = maxval(work)
  do i = 1, num_images()
    great[i] = great          ! broadcast the results to all
  end do
end if
sync_all()                    ! all wait for image 1 to finish
```

Note the lack of explicit, temporary arrays (`work`), if statements, and do loops in the first implementation. We suggest that the lack of these constructs (in the first implementation) makes the code more readable and their overuse can get in the way of program understanding. The first implementation is expressed at a higher level of abstraction; the programmer instructs the compiler regarding *what* should be done, without unnecessarily informing the compiler *how* it should be accomplished. This is left up to the compiler to decide.

### 4.2 Producer/Consumer

The second example we consider is that of an agent producing information that another consumes. This example was chosen to examine implementation of an

algorithm that is more loosely coupled than the first, requiring no algorithmic synchronization.

A genetic algorithm (GA) [1] implementation could be run in parallel by instructing each image to work on an entirely separate and distinct population, with no communication between images. This would be as if a real population were evolving on an island, separated by other populations of the same species. A population is a set of individual genomes that exchange genetic information during the creation (breeding) of a new generation. We define a population as an integer co-array `pop` with both a local and a co-rank of one, `integer :: pop(pop_size)[*]`.

For better results, it is desirable to exchange genetic information between separate populations. The algorithm does not require synchronization between images because the timing of the information exchange is not crucial. For each population and at each generation, we choose to replace one individual chosen randomly with another individual, also chosen randomly from the global population:

```
do n = 1, num_generations
  call new_generation(pop) ! create new generation, local memory
  i = random_image();  k = random_index();  l = random_index()
  pop(k) = pop(l)[i]  ! (*) replace local individual
end do
```

The implementation of this GA is identical, whether using the original or our revised CAF assignment operator definition.

Note that while no synchronization is required by the algorithm, two `sync_all` calls would be necessary around the statement `(*)` to ensure that no remote reads can occur while the local population is being modified. This becomes clear when the elements representing individuals in the `pop` array are not atomic with respect to local memory stores operations. Global synchronization is an expensive operation. Thus, this example points out the need for a CAF language construct to provide mutually exclusive access to non-atomic data structures. To do this, however, consideration must be given to performance aspects, because such a language construct could harm the performance of all remote read operations.

## 5 Conclusions

We have proposed an alternative definition of the assignment operator for Co-Array Fortran. This new definition was required to allow collective operations to be expressed using co-array section notation while ensuring that the language specification does not imply that excessive and redundant messages would result. Our alternative definition specifies that assignment statements involving cooperating images have one-sided, get semantics. Get semantics allow the use of collective assignment statements without causing inefficiencies to arise, as unneeded data is transferred, or causing illegal, concurrent writes to memory. Put semantics are allowed only via a different syntax. The proposed changes provide the following advantages:

- Array-section notation may be specified in either the local or the co-dimensions, thus unifying its usage.
- `If` statements denying concurrent writes are not needed. A programmer is not able to *express* concurrent writes to a single address in the language, unless the separate `.recvs.` syntax is used.
- The programmer is able to express collectives without explicit loops.
- Explicit temporaries can be eliminated when expressing collective all-reduce and gather-scatter operations.
- The programmer does not need to be artificially aware of memory affinity, as may occur with the explicit use of `if` blocks to restrict execution to a single image; get semantics allows a programmer to think of the co-dimension as a memory address, rather than an image index.
- While get semantics is specified as the pattern to which a programmer designs, the compiler is free to choose the optimal method for data transfer, provided that these semantics hold for the program.
- Collective notation allows the programmer to explicitly specify which images are expected to participate within an expression or assignment statement.

Perhaps most importantly, the new Co-Array Fortran semantics leave room for future extensions to the language in ways that the original semantics do not. For example, suppose CAF could be extended to allow a generalized vector subscript notation (or generalized section notation), that would select co-array elements from the *combined* local and co-dimensions.

## Acknowledgments

Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36, LA-UR-05-9203.

## References

1. J. H. Holland. *Adaptation in natural and artificial systems.* The University of Michigan Press, Ann Arbor, MI, 1975.
2. American National Standards Institute. Information technology – programming languages – Fortran – part 1: Base language. Technical Report ISO/IEC 1539-1:2004, 2004.
3. Robert W. Numrich and John K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998. http://www.co-array.org.
4. Robert W. Numrich and John K. Reid. Co-arrays in the next Fortran standard. Technical Report ISO/IEC JTC/SC22/WG5 N1642, May 2005.
5. John K. Reid, Personal communication, Delft, May 2005.