ISO/IEC JTC1/SC22/WG5 N1702

Rationale for Co-Arrays in Fortran 2008

Aleksandar Donev

20th September 2007

Abstract

This paper explains some of the technical decisions made during the incorporation of co-arrays in Fortran 2008. Familiarity with John Reid's summary of co-arrays [R07] is assumed. This document is by no means exhaustive: It is a selection of issues from my recollection. This article is not an official document and has not been approved by J3 or WG5.

Many thanks to John Reid and Bill Long for helping me write this Rationale.

1 Design Principles

Some fundamental design principles guided much of the technical decision. Some of these principles are from the perspective of language design, and others are specifically meant to ensure that the language can be implemented efficiently.

1. *Co-array programs will follow a simple SPMD model*
   The simplest parallel extension of Fortran 2003 is to replicate a traditional serial Fortran program on a set of processors and provide a means of accessing data among them. This is the Single Program Multiple Data (SPMD) programming model. The SPMD model has been widely adopted in existing parallel codes due to the popularity of MPI-1. There are a constant (NUM_IMAGES) number of equivalent but distinct and fully asynchronous execution sequences each with its own memory, throughout the execution of the program. There is no concept of master storage (unlike UPC, for example, where objects have affinity to threads but otherwise exist in a shared memory space) or of a (sequential) master execution sequence (unlike HPF) or of spawning/forking (OpenMP, MPI-2) of new execution sequences. The goal has been simplicity and easy integration (for both compilers and programmers) with existing (serial) Fortran. Simple, basic facilities are least likely to become obsolete and provide the greatest flexibility to the programmer. If desired, other models for parallel computing, such as global arrays, data-parallel models, or task-parallel models, can be implemented using the capabilities provided by the co-array model.

2. *Transparency of remote data accesses*
   All references or definitions of remote data can easily be identified (by both the compiler and the programmer) by looking for co-indexed data references (square brackets). There is no hidden "communication", thus enabling the programmer to identify the performance-critical code. An example of a technical decision guided by this design principle is the decision to not allow co-indexed objects as *selector*s in the ASSOCIATE construct. If we had allowed this, one could disguise a remote object to appear as if it is an ordinary local variable. Similarly, local pointers cannot be associated with remote memory.

3. *All synchronization should be explicit*
   Unlike HPF or other higher-level parallel languages where the compiler is expected to perform some magic and insert data-movement and synchronization in places not obvious to the programmer, co-arrays gives full control (and responsibility) to the programmer. This design principle has influenced lots of technical decisions, for example, automatic allocation/deallocation of allocatable co-arrays has been avoided in most cases by various restrictions not present for local allocatable arrays or allocatable components.

4. *A co-array exists and is always of the same type and shape on* ***all*** *images*
   A co-array exists on all images by virtue of the fundamental SPMD execution model. The same program (containing the same declarations) is executed on all images and non-dummy non-allocatable co-arrays are restricted to always be saved (static) and have the same declared bounds on all images (this is unlike UPC, for example). For deferred type or shape allocatable co-arrays there are runtime restrictions requiring that the same dynamic type, type parameters, and bounds be specified, and allocation is a collective operation. Therefore, even in the presence of different execution sequences on different images, each image has the same set of co-arrays. Support for direct references and definitions of irregular data distributions among images is provided through allocatable and pointer components of co-arrays.
   This design principle enables an efficient implementation of co-arrays to use a *symmetric heap* even in distributed memory settings. Basically, co-arrays can be placed at the same virtual address on each image, or at least at some easily-computed offset from some known base virtual address. On shared memory machines, the stride between the virtual memory base addresses can be made constant. An image can thus determine the virtual address of a co-indexed object on another image without any information from other images: the address is the same as its local address with possibly an easily-computed offset. This kind of optimization is unique to co-arrays and can make it much more efficient than other paradigms (e.g., one-sided communication in MPI-2). Furthermore, unlike library approaches like MPI, because co-arrays are integrated into the typing system, the compiler has enough information to perform routine optimizations, assignments with implicit type conversions, and of course all the usual type-safety checks.
   Copy in/out of actual arguments associated with dummy co-arrays is fully avoided by restrictions that guarantee that address- or descriptor-based argument passing is easily achieved. If copy in/out were allowed different execution sequences on different images could lead to copies existing on some of the images and not the others.

5. *A single segment should be optimizable as if it were traditional serial Fortran code*
   The set of statements executed in-between synchronization (image-control) statements is called a segment and is amenable to full local optimizations (e.g., non-aliasing assumptions). Image control statements will in most cases act as barriers to code motion as would, for example, operating system calls. At least in principle, from the perspective of a single image, remote memory can be treated as yet another memory hierarchy above local memory and caches. Of course, lots of additional work might be required to teach the optimizer how to fully optimize remote data references and definitions, depending on the existing hardware and operating system support (e.g., modern hardware provides caches for local memory which it may not do for remote memory).
   The memory consistency model in co-arrays is intentionally simple (unlike, for example, UPC) in order to support this design principle. In effect, except for references to certain VOLATILE variables (which would not get optimized anyway), there can be no conflicting or ambiguous references/definitions of a co-variable or a variable with the TARGET attribute by multiple images (TARGETs can still be aliased locally).

2 Corresponding Co-Arrays

The concept of a corresponding co-array is central to co-arrays even though it does not feature prominently in the text of the standard. This concept replaces the concept of distributions of an array as exist in HPF or UPC (they call it affinity but it is essentially a block-cyclic distribution of a global array among threads). For every co-array, which is a purely local object (an object residing and belonging to this image only), there is a corresponding co-array on other images. Co-indexed objects reference co-arrays, possibly on other images. The co-indexed data reference is to the co-array corresponding to the local array that would be referenced if the square bracket were deleted. This correspondence is essential for both the semantics, in the sense that it enables the programmer to reason about what object a co-indexed data reference really refers to, and for implementations, since it enables efficient compilation of remote data references.

For named non-dummy, non-allocatable co-arrays correspondence is established at the level of declarations: Corresponding arrays are declared with the same declaration. Such arrays must have the SAVE attribute so there is effectively only a single (static) co-array per declaration. For saved allocatable co-arrays correspondence is also established at the level of declarations.

2.1 Non-Saved Allocatable Co-Arrays

Allocatable co-arrays do not have to be SAVEd; in particular, they can be local procedure variables, or declared in a BLOCK construct. If the procedure is not RECURSIVE, the local allocatable co-array exists only during the execution of the procedure (it is automatically deallocated upon return from the procedure). Since there can be only one active instance of a non-recursive procedure, correspondence is trivially established just as for SAVEd variables. Similar considerations apply to co-arrays declared in a BLOCK construct. Some compilers may support threaded execution in which case on each image there may be multiple active instances of the same procedure; the standard does not cover this. Compilers may implement extensions but it would seem that for now threads should not use any of the co-arrays features other than remote data accesses (which should still obey the memory consistency requirements).

A recursive procedure may have multiple active instances on each image, all ordered on the stack. For such procedures correspondence between local allocatable co-array variables is partially established at runtime when ALLOCATEing the variable, since there is an implicit synchronization. At first we wanted to restrict the allocation to occur at the same recursion depth so that corresponding co-arrays are all at the same recursion depth (this would make the semantics simpler). However, important iterative algorithms may require different iteration counts on different images and so such a restriction was not added after all. The proper nesting of the multiple active instances enables implementations to not have to do anything special about RECURSIVE procedures. Similarly, programmers do not have to worry about the automatic deallocations causing deadlock so long as the allocations were properly done.

Identical considerations apply to allocatable co-array components of local scalar variables of a derived type.

2.2 Non-Allocatable Dummy Co-Arrays

Because each image executes asynchronously, including procedure calls, argument association is purely local, even for co-array dummy arguments. As explained by Reid [R07], copy in/out is not going to occur for co-array dummies due to added restrictions on actual arguments. Because of this, co-array dummies are *directly* associated with the actual argument, and by following the call chain eventually the dummy is associated with an *ultimate argument* that is a subobject of a non-dummy named co-array or allocated co-array component. Therefore there are co-arrays corresponding to the ultimate argument on all other images, and these co-arrays also correspond to the dummy argument after being interpreted to have the same type, rank and bounds as the dummy argument (remember that sequence association may change the rank, bounds and even type of a dummy relative to the associated actual argument).

Implementations need not do anything special about dummy co-arrays. Co-indexed references can be interpreted based on the properties of the dummy only. If a symmetric heap is used, co-array dummies can be passed exactly as are normal explicit, assumed-size, or assumed-shape dummy arrays. Scalar co-arrays may be passed differently by some implementations that treat array and scalar arguments differently. Similarly, shared memory implementations that represent co-arrays as arrays of higher rank will need to pass additional information about the stride between the base virtual addresses of the corresponding co-arrays (or simply pass a higher-rank descriptor for arguments passed by descriptors). Because of this possible change in passing conventions and the restriction that the actual must be a co-array (co-arrays do not exist in C), interoperable procedures may not have co-array dummies (even though co-arrays *can* be interoperable as a local array).

2.3 Allocatable Dummy Co-Arrays

A procedure with an allocatable co-array dummy argument may be called by a single image and the argument association is local just as for ordinary co-array arguments. However, if one actually wants to change the allocation status inside the procedure (rather than just test it), then all images must execute that procedure since (de)allocation of co-arrays is collective. Furthermore, there is an additional restriction that the ultimate argument must be the same non-dummy allocatable array on all images. This rule maintains proper correspondence between allocatable co-arrays on different images even when different images execute different procedure calls:

```
REAL, ALLOCATABLE, DIMENSION(:)[:] :: x, y
IF(THIS_IMAGE()==1) THEN
   CALL Sub(x) ! The dummy is allocatable and will be allocated
ELSE ! This is not allowed!
   CALL Sub(y) ! All procedures must execute with the same actual
END IF
```

3 Allocatable Co-Arrays

Fortran 2008 allows allocatable co-arrays. Pointer arrays are not as efficient and widely used in scientific codes as allocatable arrays, and therefore pointer co-arrays were not added at this time.

```
REAL, ALLOCATABLE, DIMENSION(:)[:] :: x
```

Allocatable arrays can be implemented as an array descriptor that describes the base address and bounds of the actual memory holding the array, once it is allocated. Additionally, the descriptors for allocatable co-arrays need to hold information about the co-bounds (no co-strides). It is expected that the descriptor for an allocatable co-array, which is a local object, contains all information needed to identify the corresponding co-arrays on other images, without consulting descriptors on other images.

Memory for allocatable co-arrays must be allocated identically on all images (same type, type parameters, bounds, and most importantly, the same co-array). For this reason, allocation of co-arrays is a collective operation (i.e., the same ALLOCATE statement must be executed by all images for the same co-array(s) and involves implicit synchronization). These restrictions ensure that allocatable co-arrays can be placed on a symmetric heap and that implementations can use the same descriptors for allocatable co-arrays as it does for ordinary allocatable arrays, with the addition of co-bounds as an additional piece of the “co-descriptor”.

3.1 Co-Array Components

The original co-arrays design by Numrich and Reid [NR98] did not include co-array components of derived types. The reason is that co-array components only become usable when they are allocatable, and allocatable components were not fully standardized in Fortran until Fortran 2003.

```
TYPE :: DisallowedType
    INTEGER, DIMENSION(2)[*] :: i
    REAL, DIMENSION(3)[*] :: x
END TYPE

TYPE(DisallowedType) :: object ! Must exist on all images
```

Consider a derived type with a non-allocatable co-array component (not allowed in F2008!). What that means is that an object of such a derived type must exist on all images, so that the co-array component exists on all images. This effectively makes objects of such derived types as restricted as co-arrays. In fact, the same functionality can be achieved by making the component a regular array component and then declaring a co-array of such a derived type. Multiple such non-allocatable components could be grouped together into a single derived type and a co-array of that type used instead of making the components co-arrays:

```
TYPE :: ClassicType
    INTEGER, DIMENSION(2) :: i
    REAL, DIMENSION(3) :: x
END TYPE

TYPE(ClassicType), DIMENSION[*] :: object
```

The situation is different for allocatable co-array components. Allocatable co-arrays are implemented as (co-)array descriptors. The descriptors themselves are purely local objects that need not be (directly) accessible to other images and therefore they need not be on a symmetric heap. Allocation of co-array components is a collective operation just as for top-level allocatable co-arrays. The co-array component, once allocated, exists on all images with the same type and bounds and can be made to have the same virtual address:

```
TYPE :: AllowedType
    INTEGER, DIMENSION(:)[:], ALLOCATABLE :: i
    REAL, DIMENSION(:)[:], ALLOCATABLE :: x
END TYPE

TYPE(AllowedType) :: object ! Must be a "plain" scalar
ALLOCATE(object%i(2)[*]) ! Collective operation
ALLOCATE(object%x(3)[*]) ! Implicit sync
```

It is important to point out that having a local object (even if it exists on all images) of a derived type with an allocatable co-array component is different from a co-array of a derived type containing a traditional allocatable component. In the second case, the components can be allocated independently on each image and an image cannot calculate virtual addresses on other images without first consulting the descriptor on that other image (see Section 3.2).

3.1.1 Restrictions on Co-Array Components

Because the allocation/deallocation of co-array components is a collective operation and requires synchronization, numerous restrictions have been added to ensure that automatic allocation/deallocation of such components does not occur (e.g., during intrinsic assignment). This put some burden on the user to learn these additional rules, however, the rules are not any different from those already in existence for top-level (non-component) co-arrays.

The most important and much less intuitive difference between co-array and regular allocatable components is the restriction that arrays of a derived type with co-array components cannot be declared. More specifically, only non-allocatable, non-pointer and non-co-array scalars of an object with co-array components can be declared. Reid [R07] gives one reason for not allowing co-arrays with co-array components. The very meaning and utility of such a feature are questionable, and the complications it would cause are great. The reason allocatable or pointer scalars with co-array components are disallowed is because such objects must exist on all images so that one can establish correspondence between co-arrays on different images. A similar argument applies to arrays of co-arrays. Such arrays would have to be exist on all images and have the same bounds (this is analogous to the restrictions needed to allow non-allocatable co-array components). In effect, co-arrays are not hierarchically composable as are derived types (i.e., arrays of arrays). Co-arrays exist at the *top level* across all images. One can extend this view in future revisions but the gain is not obvious and the complications are great.

Since having a co-array component changes the properties of a type in important ways (e.g., one cannot declare allocatable objects of such a derived type), one is not allowed to extend a derived type and add co-array components unless the parent type already has co-array components. Without such a restriction the basic *is-a* relationship of inheritance would be compromised for derived types containing co-array components. Nevertheless, despite all of the additional restrictions, one can use derived types to group allocatable co-arrays (data encapsulation) and even use inheritance and polymorphism together with co-arrays. This is an important feature facilitating object-oriented programming and was one of the reasons for adding co-array components.

3.2 Non-Symmetric Objects

The restriction that co-arrays be identical on all images may be limiting in certain situations. In particular, one may want to allocate memory for a co-array only on one image (e.g., a master), a subset of images (i.e., a team), or one may want to have different sizes on different images (e.g., imperfect partitioning). It is clear that one must sacrifice some of the efficiency of symmetric co-arrays in order to use such non-symmetric co-arrays. In particular, direct addressing of co-indexed objects cannot be achieved, instead, images must exchange some addressing information before being able to modify each-other's data. There will also be an additional performance impact on architectures where not all memory is efficiently accessible by other images. On such architectures it would be beneficial to store all local arrays in purely local memory and put objects accessed by other images in special "shared" memory.

Fortran 2008 offers two ways to achieve non-symmetric "shared" arrays: co-arrays of a derived type with allocatable or pointer components. The obvious implementation is that co-indexed objects referencing such non-symmetric data will require first examining the array descriptor from the referenced image, then calculating the virtual address based on that information, and finally referencing or defining the remote object using that virtual address. This will make such references more expensive so programmers will need to be careful in deciding whether to use ordinary symmetric co-arrays or non-symmetric arrays. However, for multiple references within a segment, the compiler might cache the remote descriptor information locally and thus amortize the extra cost.

Implementations must support direct access to allocatable components that may be exposed to other images through a parent co-array, as well as targets that may be associated with a pointer component of some co-array. This may incur an additional performance penalty. However, compiler analysis may be able to eliminate many variables from consideration based on scoping and type rules. Runtime analysis may also help, for example, a field can be added to descriptors of allocatable components to indicate if they are accessible to other images (the field can be set when the co-rank of the top-most parent object is declared).

4 Memory Consistency

Memory consistency for a serial Fortran program is handled automatically by the compiler. The order of execution of statements that reference or define variables determines the order of the associated memory operations. The memory consistency model in co-arrays is as relaxed as possible in order to permit maximal optimizations. As in serial Fortran, there are restrictions in place that make the semantics unaffected by optimizations.

When executing with more than one image, statements executed on other images may affect or depend upon the values of local variables that are co-arrays or have the TARGET attribute. In this context, additional coordination is required to ensure that the definition of a variable occurs before a reference that depends on that new value, and that another definition does not occur in between. This is accomplished using image control statements. Two statements executed by two images can be *ordered* with respect to each other or *unordered*. This partial ordering can be determined without any reference to actual execution speed (timing) during a particular run of a program. Therefore, a standard conforming program will (given the same execution sequence) execute correctly regardless of the actual speed of the different images. In fact, the memory consistency restrictions that the standard imposes can be checked by a serial debugging tool that emulates a parallel execution.

4.1 Segment Ordering

Statement ordering can be achieved by built-in image control statements. The execution sequence is broken into segments, each segment spanned between two successive image control statements. The statements within a segment are only ordered locally on the executing image (as in serial Fortran) but they appear unordered to other images. That is, the compiler has full freedom to do any serially-consistent code motion within a segment without worrying about what other images are doing and treat segment boundaries as (traditional) barriers to code motion. In the UPC vocabulary, all co-arrays co-indexed references are "relaxed". In this scheme all co-indexed definitions are automatically "atomic" since they complete at segment boundaries and there cannot be any conflicting (concurrent) references or definitions.

The segment ordering itself is a partial ordering induced by a graph of the execution sequence where each one-way synchronization (e.g., a "follows" relation) is a directed edge and segment boundaries are nodes. Here the execution of each image is a linear graph, that is, loops are unrolled. For example, if at a segment boundary image i executes NOTIFY(j), there is an edge from that node on image i to the corresponding node on image j. Similarly for QUERY and SYNCs, and SYNC MEMORY if there is user-defined segment ordering (see below). The resulting graph is a directed acyclic graph (DAG) and therefore induces a "follows" (or "precedes") partial ordering of the segments. The ordering has no relation to actual timing during the execution.

4.2 Memory Consistency Restrictions

The basic restriction that ensures memory consistency is that if an image defines a variable, no other image may reference or define that variable in unordered segments. Two images may reference the same variable in unordered segments since this causes no problems. Procedure calls may cause a copy of an INTENT(OUT) actual argument to be made when the dummy is not a co-array. Therefore, other images are not allowed to define or reference the associated argument until the procedure completes execution and a new segment starts (and thus the copy out is completed), regardless of any ordering of the segments during the execution of the procedure. In a sense, the execution of the procedure is considered a super-segment where all segments comprising the super-segment effectively define the actual. No attempt was made to only apply this stronger ordering restriction in cases where copy in/out *could* apply, since the rules would be confusing and programmers should not have to worry about copy in/out explicitly. Example:

```
REAL, DIMENSION(10)[*] :: x
SYNC ALL ! Start super-segment S
CALL Sub(x(1:10:2)) ! Dummy is not a co-array
! Other images may not reference or define x(1:10:2)
SYNC ALL ! End super-segment S
! Other images may reference or define x(1:10:2)
```

4.3 User-Defined Segment Ordering

The standard provides a reasonably complete set of synchronization statements; however, users will sometimes need special user-defined segment ordering either for increased efficiency or in order to express certain algorithms more naturally. For example, special data-structure locks may be implemented to enable conflict-free access to a shared priority queue. Furthermore, during migration of codes or in specialized situations users may use synchronization mechanisms outside the scope of the standard.

Note that user-defined ordering follows the same memory-consistency rules as standard-specified synchronization.

4.3.1 SYNC MEMORY

Storing a value from the processor to memory involves, conceptually, two steps. The destination address and data value are sent into the memory system, and later a signal comes back indicating either that the operation completed, or it failed. Similarly, loading a value may be done in two steps, the first requesting the value, and later waiting until the value is received. If the memory address is on a remote image the details of how this happens might involve extra steps compared to the process for a local address, but the concept is the same.

For local memory accesses (e.g., in Fortran 2003) the compiler and hardware (e.g., caches or speculative execution machinery) are expected to respect the ordering of loads and stores and ensure memory consistency, so we don't have any memory flushing or refreshing statements in Fortran 2003. When interactions with other co-executing entities (processes, OS kernel, other images, etc.) exist, however, additional actions are required to ensure that there is a consistent view of memory and files. The VOLATILE attribute serves this purpose for loads of variables that may be changed externally to a program, and FLUSH serves the purpose for writes to files. The problem with VOLATILE is that it affects every reference to the variable so that it effectively destroys optimizations.

The SYNC MEMORY statement suspends the execution sequence until all locally initiated memory operations involving co-arrays or objects with the TARGET attribute have completed. The effect of SYNC MEMORY is included implicitly in all of the standard synchronization statements, so that most programmers will not need to concern themselves with SYNC MEMORY. In UPC the function of SYNC MEMORY is achieved by upc_fence (a null strict memory access). When a SYNC MEMORY statement is executed remote memory definitions that occurred in the preceding segment will need to be flushed to memory so they become visible to all other images. Similarly, local temporary copies of co-array variables (for example, in registers, caches, or buffers) will need to be refreshed from main memory (as for VOLATILE variables) since they might have been changed by other images. In the absence of additional code analysis, the SYNC MEMORY statement will effectively act as a barrier to code motion.

4.3.2 Volatile Variables

The memory consistency rules are relaxed for single-word (one numeric storage unit) VOLATILE variables. This is to allow the coding of user-defined segment ordering with constructs such as spin loops. If such ordering appears, a SYNC MEMORY statement must be used to delimit segments.

The relaxed restriction allows images to reference a VOLATILE variable even if some other image is defining it in an unordered segment. The implementation will reload such variables from memory at every reference. Variables of types occupying other than a single word are excluded because stores and loads of such variables may not be atomic. Two images may not both define a volatile variable in unordered segments. The rationale is that Fortran 2003 only changes the meaning of references to volatile variables (namely it forces reloading from memory). It was argued that compilers do not treat definitions of volatile variables any differently from ordinary variables (e.g., flushing them to main memory instantly), and we did not want to change this.

One can code all of the synchronization primitives using VOLATILE locks and SYNC MEMORY, but of course the vendor-provided version will often be significantly more efficient.

5 Synchronization

Several synchronization primitives are provided in the form of statements. The original design used intrinsic subroutines. However, it was realized that statements supported more flexible syntax (example, a star for a team). Additionally, synchronization is very tightly integrated into the semantics of co-array programs through the memory consistency model and therefore statements seemed more appropriate.

Note that, for flexibility, synchronization statements do not follow a "textual" or source-code matching, that is, different statements may be executed by different images and match each other. In fact, there is no way to enforce (for debugging or safety purposes) matching of a particular SYNC statement with another (unlike, for example, Titanium, where barriers must be textually aligned, or UPC, were one can provide a barrier value that the runtime can match). There are some situations in which it is required that all images execute the same statement (e.g., ALLOCATE, call of an intrinsic procedure, OPEN, etc.). This is required for simplicity: If it is the same statement on all images than one does not have to design complex rules for matching. For example, it did not seem wise to allow some images to specify an ACTION keyword in OPEN and others to not do so, or to allow some images to allocate two co-arrays in an ALLOCATE statement and others to do it in two ALLOCATE statements.

5.1 Synchronization Statements

The SYNC ALL statement performs a global barrier, which is the most commonly used form of image synchronization. Because of its widespread use and the possibility of hardware-specific special optimization, this operation is implemented as a separate statement. Other statements that have a similar effect to that of SYNC ALL, for example, a SYNC IMAGES(*), cannot always be compiled to use the optimal barrier mechanism (e.g., a hardware barrier) because not all images are required to execute the same synchronization statement.

The SYNC IMAGES statement suspends execution of the local image until each of the images in a list has executed a SYNC IMAGES statement that specified this image in its image list. SYNC IMAGES is used to synchronize a subset of the images and it is very flexible. The lists may vary from image to image. The only requirement is that the image numbers in the lists match up on a pairwise basis. SYNC IMAGES is particularly useful if the number of images involved in the synchronization is small, with the synchronization of just two images with each other the most common case. SYNC IMAGES is also useful for so called wavefront synchronizations and cases where the image lists are highly asymmetric, such as one master image specifying all other images, and each of the others specifying only the master. Because of its generality, SYNC IMAGES may not be very efficient for synchronization of large numbers of images.

Bidirectional synchronization, as provided by SYNC IMAGES, can result in lots of wasted CPU cycles as an image waits for its partners to check into the barrier. Additionally, there are situations in which a unidirectional synchronization is sufficient, for example, a producer need only notify a consumer that it has completed some action but it does not need to wait for a confirmation. For this reason a split-phase barrier is provided through the NOTIFY and QUERY statements. They allow overlapping (local) computation with waiting for other images to check into a barrier. A SYNC statement can be represented as a NOTIFY/QUERY pair, in fact, UPC defines its barrier this way.

The SYNC TEAM statement is discussed in Section 5.3.

5.2 Mutual Exclusion: CRITICAL

The CRITICAL block supports a semantically-clean form of mutual exclusion similar to that provided by mutex variables and other types of locks in other programming paradigms (e.g., pthreads). The advantage of CRITICAL is that it eliminates explicit management of locks by the programmer and helps avoid common errors such as forgetting to release a lock. The CRITICAL section restricts the execution of a segment of code to a single image at a time, as if a lock was acquired and locked at the beginning and then released at the end of execution of the block. The mutual exclusion is based on source-code, that is, other images are not excluded from executing statements inside other CRITICAL sections. Users that want to lock access to shared data from any statement can use CRITICAL to implement their own locks (but schemes based on VOLATILE and spin locks may be more efficient on some platforms).

The body of a CRITICAL section forms a segment and these segments are ordered among images in some unspecified way. The actual ordering depends on runtime conditions (for example, which image gets to the section first), and the conformance of a program (with respect to memory consistency) cannot rely on the actual ordering. Arbitrary local operations can be carried inside a CRITICAL section, notably, file I/O that involves only the executing image is allowed.

5.3 Teams

It is common in applications to partition the set of images into subsets, termed *teams*, that work on a sub-problem (for example, ocean and atmosphere in a climate model, or molecular dynamics and finite elements in a materials problem). There is also use of teams in obtaining maximal efficiency on hierarchical machines like clusters of SMPs: The processors in the SMP or the cores in a multicore could be a team.

Synchronization of a team can be performed scalably (to very large numbers of processors), safely, and efficiently, with SYNC TEAM. The same effect can be achieved with SYNC IMAGES. However, for large systems, the list of images in the team can get very large and the cost of merely storing and examining the list of images will be too large. It becomes difficult and error prone for the user to carry around this list and guard against errors. Finally, efficient execution of collective operations often requires some startup work, where, for example, the communication topology is examined and a binary reduction tree is constructed. Such startup work will be carried by the runtime (and hardware) for the set of all images (i.e., for SYNC ALL), but it cannot be done for teams unless the compiler is told to do so. The user cannot code such things purely from within co-arrays, rather, they need to rely on hardware details or mechanisms outside the standard.

5.3.1 IMAGE_TEAM

To resolve the problems discussed above, a concept similar to an MPI communicator was added to the language, that is, a team descriptor stored in a scalar object of type IMAGE_TEAM. This object is initialized in a call to the intrinsic FORM_TEAM in a collective operation that checks against errors and does any pre-computations needed to later efficiently execute collective operations. The team descriptor is then passed on to SYNC TEAM, I/O statements, or collective intrinsics. The implementation has full freedom over what to store inside such a team descriptor. It may (on small SMPs, for example), merely store a list of the images as an array of a certain size (e.g., the maximal number of images allowed). On other machines, the team may be a bitmask where the set of bits corresponding to images in the team are ones and the rest zeros (this may be passed directly on to hardware synchronization primitives). On very large systems each image may only store a list of its immediate team neighbors to be used in tree operations (in which case, however, the intrinsic TEAM_IMAGES must do some extra work). Or, the descriptor may simply contain one pointer to some external runtime/OS data-structure like an MPI Communicator.

The standard says that IMAGE_TEAM has pointer components and no allocatable components. This of course cannot be tested by programmers so a compiler is free to ignore it; however, it does restrict programmers from trying to do non-portable operations with teams such as copying a team from one image to another (the pointer components would become undefined). Teams can be copied (duplicated) using assignment, and the compiler must ensure that this works transparently for the programmer. There is no intrinsic procedure to explicitly "destroy" a team; this is left to the compiler to figure out from scoping rules if needed (e.g., destroy an unsaved team when it goes out of scope, but keep any copies remaining in scope).

5.4 Collective Intrinsics

In addition to synchronizations, scientific codes often perform collective reduction operations on co-arrays over a set of images (a team). The most efficient implementation is platform-dependent, and therefore the standard provides some basic reduction operations in the form of collective intrinsics. Only all-to-all reductions are supported, i.e., the result is distributed back to all participating images. More general collective operations such as user-specified reduction operators were not considered at this stage but future revisions may consider them.

The collective intrinsics (e.g., CO_SUM) are subroutines instead of functions (like SUM) because the result is a co-array for maximal efficiency (see below) and functions cannot return co-arrays. Additionally, the synchronization inherent to collective operations interacts with optimizations of expression evaluations in ways difficult for the programmer to predict. In the following example, with optimization turned on, only images for which MASK is true would evaluate the presumed intrinsic function CO_SUM_FUN and this would result in a deadlock:

```
IF (MASK.AND.(CO_SUM_FUN(co_scalar)>0)) ...
```

The collective intrinsics are designed to offer maximal efficiency on any platform, rather than maximal flexibility. The input arguments is required to be the same co-array on all images, just as for ALLOCATE in the case of allocatable co-arrays. The output argument is similarly required to be the same co-array on all images. These restrictions ensure that no communication of virtual addresses among images is needed and that direct memory referencing can be used to complete the reduction without allocating additional temporaries. Future revisions or compiler extensions can allow non-coarray arguments, and for this reason we chose the rather general name SOURCE for the input argument. The user can easily write wrappers that allow the input or the result to not be a co-array, or even to be written in the forms of functions (the use of such functions is limited to expressions that require evaluation of the function). These wrappers would need to have local allocatable co-array temporaries and require an additional copy to or from those temporaries (but compilers can optimize these away if they wish):

```
FUNCTION MY_CO_SUM(array, team) RESULT(csum)
   REAL, INTENT(IN) :: array(:)
   TYPE(IMAGE_TEAM), INTENT(IN) :: team
   REAL :: csum(SIZE(array))

   REAL, ALLOCATABLE, DIMENSION(:)[:] :: input, result

   ALLOCATE(input(1:SIZE(array))[*], &
      result(1:SIZE(array))[*]) ! Synchronize
   input=array ! Copy in
   CALL CO_SUM(array,result,team) ! Efficient reduction
```

```
    csum=result ! Copy out
END FUNCTION ! Automatic deallocation with implicit sync
```

6 I/O

Reid's summary of co-arrays [R07] describes I/O well and I do not have much to add. As explained there, file consistency, for direct-access files, is treated identically to memory consistency, namely, ordering of segments. The compiler is not expected, however, to flush all buffers at every segment boundary. Instead, it is required that a FLUSH statement be executed following each write and preceding each READ, in ordered segments, if one image needs to read a record written by another. In this sense FLUSH is like SYNC MEMORY but its effect is not bundled with synchronization statements. Implementation of direct access parallel files should not be very demanding given existing OS support for shared files. Processors may need to implement some form of record-level locking.

Sequential files are very restricted (essentially to log files) so there are no complications. Processors only need to implement some form of record locking to ensure records from different images do not get mixed up, otherwise no specific ordering among images is imposed. Support for this is likely already in the system or runtime.

One of the central issues in discussions about co-arrays I/O was whether to make unit numbers local to each image or global (shared among images). The argument for making them global is that this is more restrictive and thus more likely to be compatible with existing compiler extensions for parallel I/O. However, this is not consistent with the design goal to not modify the semantics of serial Fortran (and thus local I/O) as little as possible and to keep objects local by default. It is also more flexible for the programmers to allow the same unit number to be used on different images for local file I/O, especially when incrementally parallelizing existing serial code. Therefore units are now local.

The original intention was to make all image control statements known at compile time. However, OPEN and CLOSE can sometimes perform operations on files with connect teams of more than just the local image without this being visible at compile-time (certainly not visible in the syntax). We did not want to make all OPENs and CLOSEs image-control statements because we did not in any way want to impact purely serial programs, and because we did want to allow local file operations inside CRITICAL blocks. Therefore, only OPEN and CLOSE statements for a file connected to a team of more than the executing image are image control statements. In practice, on most systems, OPEN and CLOSE would be system calls and cause the effects of SYNC MEMORY even if the team consists of only the current image. On other systems compilers can do a runtime test and include the effects of SYNC MEMORY only if the team does contain more than one image.

References

[R07] *Co-arrays in the next Fortran Standard*, John Reid, UK ISO/IEC JTC1/SC22/WG5 document N1697 (2007)

[NR98] *Co-Array Fortran for parallel programming*, R. W. Numrich and J. K. Reid, ACM Fortran Forum (1998), 17, 2 (Special Report) and Rutherford Appleton Laboratory report RAL-TR-1998-060 (1998)