A Critique of Co-array Features in Fortran 2008 *Working Draft J3/07-007r3* [*]

John Mellor-Crummey Laksono Adhianto William Scherer

Department of Computer Science
Rice University
Houston, TX, USA

February 10, 2008

Abstract

This note offers a critique of support for parallelism in Fortran 2008 based on co-arrays. We believe that there are some significant shortcomings in current design of co-array features that affect their suitability for mapping onto a range of parallel systems, expressing a wide range of parallel applications, supporting the development of parallel libraries, and providing an extensible framework for developing sophisticated parallel applications. Based on these shortcomings, we believe that it is premature to recommend to the WG5 committee that the collection of co-array features described in *Working Draft J3/07-007r3* be incorporated into the language standard without significant refinements.

1 Introduction

As the J3 committee considers adding support for parallelism based on co-arrays to Fortran 2008, it is important that the collection of features be carefully designed so that the language yields the maximum benefit for the target audience of developers. Below we list a few questions that we believe are worth considering as part of this process.

- *What types of parallel systems are viewed as the important targets for Fortran 2008?*

  Over the next few years, small-scale parallel systems built out of one or a few multicore processors will become ubiquitous. Moderate-scale parallel systems such as clusters loosely coupled with Gigabit Ethernet or more tightly coupled with a higher performance interconnect fabric such as Infiniband will also be common. Large-scale MPP systems, such as Cray's XT series, IBM's Blue Gene series, and SciCortex's SC series will be prominent, but much fewer in actual number than other types of systems.

  We believe that for the parallelism support in Fortran 2008 to be viewed as successful, its language features must support writing code that maps well onto multicore systems with small-scale parallelism. However, the language should also support writing codes that gracefully scale from multicore processors to clusters and MPPs with high efficiency.

- *Does Fortran 2008 provide the set of features necessary to develop parallel scientific libraries that will help catalyze development of parallel software using the language?*

  We consider, the collection of capabilities a parallel language needs to support construction of useful parallel libraries. Quoting from version 1.1 of the MPI message-passing interface standard [11]:

[*]J3 paper number 08-126. This critique arrives at the beginning of the J3 Meeting on Feb. 10-15, 2008. The authors of this critique view it as an unfinished, incomplete draft, but are submitting it anyway in the hope that there might be an opportunity to consider our opinions before the committee votes to forward a proposed revision of the Fortran standard to WG5.

The key features needed to support the creation of robust parallel libraries are as follows:

- *Safe communication space, that guarantees that libraries can communicate as they need to, without conflicting with communication extraneous to the library,*
- *Group scope for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved processes (potentially running unrelated code),*
- *Abstract process naming to allow libraries to describe their communication in terms suitable to their own data structures and algorithms,*
- *The ability to "adorn" a set of communicating processes with additional user-defined attributes, such as extra collective operations. This mechanism should provide a means for the user or library writer effectively to extend a message-passing notation.*

In addition, a unified mechanism or object is needed for conveniently denoting communication context, the group of communicating processes, to house abstract process naming, and to store adornments.

We discuss how Fortran 2008 features measure up against these high-level requirements in Section 2.

- *What types of parallel applications is Fortran 2008 intended to support and is the collection of features proposed sufficiently expressive to meet those needs?*

  Co-arrays were first proposed a decade ago as a simple extension to Fortran to facilitate construction of SPMD parallel applications based on a static decomposition of work by providing a facility for one-sided communication. Scientific applications have changed over the past decade—they have grown more sophisticated, they rely heavily on libraries, and coupled applications are increasingly common. To meet the needs of coupled applications, multi-block applications, or even parallel libraries for linear algebra, a parallel programming model must provide effective support for computing on processor subsets; we discuss this issue further in Section 2.2.2. Solving large problems involves manipulation of large data structures distributed over the nodes in the target system. Ideally, a programming model that supports one-sided operations on shared mutable data should provide expressive mechanisms for one-sided modification of such structures and for one-sided operations to extend the structures dynamically as needed. We address the shortcomings of Fortran 2008 for extending and manipulating shared mutable data structures in Section 2.1.

- *Will the collection of co-array features described provide Fortran 2008 facilitate writing portable parallel programs that deliver high performance on systems with a range of characteristics?*

  Having appropriate support for synchronization and coordination as part of the language will be essential to avoid having application developers hand-craft their own synchronization primitives based on co-array reads and writes. The problem with this approach is that without extreme care, such shared-memory synchronization won't scale well from tightly-coupled multicore platforms to larger, more loosely-coupled systems. We discuss aspects of this issue in Sections 2.1 and 3.5.

In Section 2, we explore issues raised by these questions and evaluate the support co-array features provide for parallel programming. In Section 3, we offer some recommendations for modest changes that we believe enhance the capabilities of co-array-based parallelism. Section 4 offers some suggestions for major changes that are worth considering as support for co-arrays in Fortran evolves. In Section 5 offer some final thoughts.

2 Shortcomings of the proposed co-array features

This section outlines several key capabilities Fortran 2008 lacks that we believe are necessary to support implementation of parallel libraries and applications.

2.1 Lack of support for one-sided manipulation of shared data structures

A strength of the co-array model is that it supports programs that use one-sided communication to read and write remote data rather than relying on two-sided message passing. Also, it enables application developers to manage locality explicitly by distinguishing between local and remote data. However, as described in *Working Draft J3/07-007r3*, support for one-sided communication in Fortran 2008 is not very expressive. In particular, shared-memory parallel programming depends upon support for extending and manipulating shared data flexibly and efficiently. There are two parts to this problem: allocation and synchronization.

Data allocation. Allocation of shared data (co-arrays) in Fortran 2008 is collective. Thus, an image can not allocate shared data in another image to extend a remote portion of a shared data structure in a one-sided fashion. Since there is no support for global pointers, one can't allocate new shared data locally and store a pointer on the remote node either. Thus, there is insufficient language support to write routines that extend shared data structures using one-sided operations. In contrast, Unified Parallel C (UPC) has flexible support for one-sided manipulation of shared data [2]. Besides collective allocation of shared data, UPC also permits one-sided allocation of both distributed shared data and local shared data.

Synchronization. Named critical sections overly limit concurrency when updating mutable shared data. Since names for critical sections are static labels, critical sections don't provide support for fine-grain concurrency control over disjoint-access parallelism. It is unreasonable to expect users to implement their own primitives for finer-grain mutual exclusion.

NOTE 8.42 in *Working Draft J3/07-007r3* shows how one might use SYNC MEMORY to implement one type of lock abstraction; however, without great care, lock implementations can result in busy-waiting on remote data, which will cause an unacceptable level of network traffic on anything but a cache-coherent shared machine. Although it is known how to implement scalable synchronization primitives using shared variables (*e.g.*, [9]), Fortran 2008 lacks read-modify-write operations such as swap, compare-and-swap, and atomic-add that are essential for implementing synchronization primitives that will work well on large-scale systems or supporting lock-free concurrent modification of data structures.

2.2 Lack of support for encapsulation

Almost universally, application developers partition large software systems into separable (and often reusable) libraries and components to manage complexity. Libraries depend upon *encapsulation*; namely, they provide a stable interface that insulates the rest of the program from the details of their implementation. Here, we consider the collection of capabilities a parallel language needs to support encapsulation to support construction of parallel libraries. (Section 1 previously quoted the MPI 1.1 Standard's review of the capabilities needed to support encapsulation for parallel libraries.) Here, we focus on three deficiencies of Fortran 2008: the lack of a safe communication space, the lack of full-featured support for processor subsets, and the lack of a support infrastructure for user-defined collective communication primitives.

2.2.1 Lack of a safe communication space

As described in *Working Draft J3/07-007r3*, `notify/query` synchronization provides only a single communication context between each pair of images; namely, synchronization occurs between image pairs themselves. As previously recognized by Sjkellum et al. [15], a single communication context is insufficient to support encapsulation for parallel library primitives. There is no reasonable way for parallel libraries to isolate themselves from the ongoing point-to-point communication in a parallel computation with only a single communication context. Quoting again from the MPI 1.1 standard [11]:

> *The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications ... and avoids the need to synchronize entry or exit into library code.*

Using Fortran 2008, if one tries to overlap communication and synchronization latency with computation by calling a library routine while one or more non-blocking `notify` operations are pending, a `notify` performed before the library call can inadvertently pair with a `query` performed by the library primitive.[1] Appendix A presents a detailed example that illustrates how having a single communication context for `notify`/`query` interferes with encapsulation.

2.2.2 Lack of support for processor subsets

Co-array allocation and deallocation. A key maxim for writing parallel libraries [15] is that libraries should work over a processor subset as well as they work over all of the processors. This essential if one wishes to employ library operations within disjoint processor subsets in a coupled application, *e.g.*, `ocean` and `atmosphere` processor subsets in a climate application. However, as defined in *Working Draft J3/07-007r3*, co-array `ALLOCATE` and `DEALLOCATE` operations are global. Paragraph 4 of Section 6.6.1.2 of *Working Draft J3/07-007r3* states that and 6.6.3 state that "There is implicit synchronization of all images in association with each `ALLOCATE` statement that allocates one or more co-arrays." Thus, co-array allocation cannot be employed within a processor subset. Hence, a library operation executing on a processor subset cannot allocate co-array data.

Named critical sections also contribute to an inability to work effectively on processor subsets. Since the names are static in the program text, they are effectively global. For instance, if a parallel library uses named critical sections within its implementation, one can't use the library independently on `atmosphere` and `ocean` processor subsets without the critical sections forcing global mutual exclusion rather than just within the processor subsets as intended.

Multi-dimensional co-arrays. Multi-dimensional co-arrays are intended to provide a convenient abstraction for data distributed over a multi-dimensional grid of process images. However, because allocation of co-arrays is global, one cannot conveniently employ multi-dimensional co-arrays over a processor subset, *e.g.* within the `ocean` part of a coupled climate application.

Image indexing. As described in *Working Draft J3/07-007r3*, images index co-array data and synchronize using absolute image numbers and image TEAMs are described by a set of absolute image numbers. Sjkellum et al. [15] state that "libraries don't want to describe point-to-point communication using hardware-dependent names, in fact, many algorithms are more natural if described in terms of point-to-point calls relative to a virtual topology naming scheme." They advocate for abstract names for processors based on virtual topologies, or at least rank-in-group names (*i.e.*, image numbers relative to a subset of the executing images). Such support would constitute an alternative to using global rank to name images. While co-arrays with multiple dimensions provide a way of establishing a virtual mesh topology, they provide no help for establishing a virtual topology within a processor subset. In contrast, MPI supports Cartesian and graph topologies (as well as extensions for user-defined topologies) for process groups.

2.2.3 Lack of support for extensible collective communication

Section 13.5 of *Working Draft J3/07-007r3* describes nine collective subroutines including `CO_ALL`, `CO_ANY`, `CO_COUNT` and `CO_SUM` (). The set of primitives provided is far from being all-inclusive and there are no extension points that help users add more. Leaving it to application developers to implement collective extensions entirely from scratch using co-array reads and writes is a recipe for disaster; it is extremely unlikely that programs extended in this way will run well on clusters, machines with different topologies, and large-scale systems.

[1]We consider non-blocking semantics for `notify` here because Section 10.5 of Reid's technical note [13] states that `notify` is intended to be non-blocking to enable communication/computation overlap. In Section 2.4, we discuss the problem that according to the definition in *Working Draft J3/07-007r3*, that seems to preclude it from being non-blocking.

2.3 Lack of a precise memory model

A memory model provides a reference that can be used to resolve questions about what reorderings compilers may perform on parallel programs and hardware may perform on executions of parallel program. Programmers need to know what to expect from their programs. Compiler writers need to understand what optimizing transformations are legal. Runtime system implementers need to know what orderings must be enforced by the hardware or software of the runtime layer. Unfortunately, *Working Draft J3/07-007r3* lacks a detailed memory model for Fortran 2008.

A memory model for Fortran 2008 would carefully define the set of legal behaviors for a program. The memory model for Fortran 2008 should be as relaxed as possible to allow compilers and runtime systems the freedom to favor interleavings of co-array accesses that deliver the highest performance on a parallel system. NOTE 8.30 clearly states that this is the intent; "operations within an image may be reordered as if it is alone in the execution". The language on page 199 is unfortunately not precise enough and a more formal description of a memory model is necessary for users to understand the semantics of the programs they write. Java defines a memory model very precisely [8, 7]. Unified Parallel C (UPC) also has a formal memory model [2].

In *Working Draft J3/07-007r3*, it is unclear what reordering transformations are legal. Can one reorder two accesses to different volatile variables? Can accesses to non-volatile variables be reordered with respect to accesses to volatile variables? Suppose image A writes volatile co-array variables `x` and `y` in that order; must all images see the writes as having occurred in the same order? Are dependences between accesses (read after write, write after write, and write after read) always preserved for accesses executed by the same image, regardless of whether the accesses touch local or remote memory and independent of the syntax of the accesses (*e.g.*, `x` vs. `x[p]`, where `p = this_image()`)?

2.4 Lack of support for hiding synchronization latency

Point-to-point synchronization with `notify`/`query` To overlap the latency of remote writes and synchronization with computation (*e.g.*, for producer/consumer, wave front communication patterns), it is essential to have a non-blocking primitive for signaling one's communication partner. Fortran 2008's `notify` primitive seems to be intended for this purpose (see the description by Reid in Section 10.5 of his technical note [13]). However, page 205, line 4 of *Working Draft J3/07-007r3* indicates that all image control operations include the effect of a SYNC MEMORY. If a `notify` includes the effect of a SYNC MEMORY, the semantics of SYNC MEMORY (as described in NOTE 8.41) "ensures that all memory operations initiated before the SYNC MEMORY its image complete before any memory operations in the subsequent segment in its image are initiated. This appears to mean that any remote co-array writes must *globally complete* before the `notify` can return. If this indeed what is intended, this can result in exposed communication latency.

Barrier synchronization The SYNC ALL primitive described in *Working Draft J3/07-007r3* only has a blocking form that makes it impossible to overlap the latency of barrier synchronization with computation. Split-phase barriers, which separate signaling barrier arrival from waiting for barrier completion, enable one to overlap computation with waiting for other process images to arrive at the barrier. Split-phase barriers can be implemented in hardware (one can set a special barrier register and to signal arrival, and await the desired state at barrier completion) or software [14].

2.5 Lack of support to exploit machine topology for locality

There is no language support that enables one to construct programs that observe latency between image pairs and adjust the image to processor mapping to exploit locality in a platform's topology to achieve better performance. Consider the case of mapping a Fortran program onto a scalable shared-memory multiprocessor composed out of multicore processors. There is no way for a Fortran program to manage the mapping of image numbers to processor cores in the system so that one can exploit locality between cores within a processor. In contrast, a process group abstraction supported by MPI enables programs to precisely control the order in which processes are mapped into a process group to exploit topology awareness. An MPI program can adapt to a topology by running tests to determine latency between processor pairs and constructing a new communicator that is a topology-informed permutation of the processes in MPI_COMM_WORLD.

3 Modest recommendations

3.1 Event counts to provide a safe communication space for `notify/query`

As mentioned in Section 2.2.1, `notify/query` synchronization on global image numbers fails to provide a safe communication space. Rather than having `notify/query` operate on images directly, we propose that `notify` and `query` be applied to *event count* variables. Event counts could be statically or dynamically allocated as needed to provide separate synchronization contexts. A parallel library could allocate its own event counts to isolate synchronization within the library from actions in the application in which library calls are embedded.

3.2 Locks to support fine-grain disjoint access parallelism

Our view is that critical sections, even named ones, have no place in a language that is designed in part to map to scalable parallel systems. Locks that can be dynamically allocated provide a more expressive mechanism that will enable application developers to exert fine-grain control over mutual exclusion to support disjoint access parallelism. At present, we recommend that locks replace critical sections entirely. Ongoing research is exploring approaches for supporting mutual exclusion using atomic blocks based on software transactional memory. Atomic blocks based on transactional approaches are likely to provide a superior implementation technology for mutual exclusion in the future, but we believe that further research is necessary before this deserves consideration as part of the Fortran standard.

Unlike critical sections, dynamically allocated locks could be used by instances of a parallel library running on disjoint sets of images without causing unwanted interference between independent image teams.

3.3 Atomic operations to support data structures and synchronization

Language design should allow for the implementation of as-yet undiscovered synchronization paradigms. To do this, it is sufficient to provide direct support for a small number of atomic read-modify-write instructions. From the success of the Java Concurrency library, it is clear that `compare-and-swap` (`CAS`) — both for integers and pointers — is sufficient to allow implementation of new synchronization features in a garbage collected language. Since Fortran is not garbage collected, additional support is needed to avoid the ABA problem [2]; Hazard Pointers [12] are one good option for this. In addition to `CAS`, the Cray SHMem library also supports atomic `swap`, `fetch-and-increment` (`FAI`), and `fetch-and-add` (`FAA`) operations; additionally supporting these would be ideal from the perspective of a synchronization library writer.

3.4 Split-phase barriers for hiding barrier latency

Split-phase barriers [4], a very common idiom for synchronization, should be supported explicitly. In this idiom, the operation of a barrier is divided into two first-class components: `signal`, a notification that the image has reached the point in its execution where other images may proceed safely beyond the barrier; and `wait`, a declaration that the image has completed all work prior to the barrier and must await the time when all other images have signaled it.

[2]The ABA problem arises when a thread reads a pointer to an object and then, for example, while it is preparing an update to the object, another thread reclaims the object and the block of memory is reallocated for another object pointed to from the same original location. If the first thread proceeds to attempt a `CAS` operation, this will succeed, though it is almost certainly incorrect.

Split-phase barriers enable programmers to overlap the latency associated with barrier synchronization with computation on data independent of that protected by the barrier synchronization. This allows "real work" to proceed in parallel with barrier communication.

3.5 Richer and extensible support for collective communication

Scatter and gather collective communication is often used and should be exported explicitly. Fortran 2008 should include extensible reduction primitives analogous to `MPI_Reduce`, `MPI_Allreduce`, and `MPI_Scan` and that accept a user-defined reduction operator analogous to those defined with MPI's `MPI_Op_create` [10]. Providing language level extension points for reductions enables developers to benefit from the runt-time system's efficient mapping of reductions to the underlying topology of the machine, perhaps exploiting a reduction tree or dimensional reductions in a mesh topology. By adding such language support, it is possible to write reduction extensions that are simple, concise, and deliver portable high performance.

3.6 Allocation and deallocation should not involve *all* of the process images

Sections 6.6.1 and 6.6.3 of the *Working Draft J3/07-007r3* state that an implicit global synchronization is performed for each allocation and deallocation. This prevents a processor subset from managing allocation and deallocation of its own co-array data. In our opinion, the standard should allow one to allocate and deallocate data among image teams. Also, non-collective forms of shared data allocation are also necessary; however they can't be considered without also considering the introduction of global pointers.

3.7 Eliminate SYNC ALL

Working Draft J3/07-007r3 supports both SYNC ALL (Section 8.5.2) SYNC TEAM (Section 8.5.3). SYNC TEAM statement has no default *image-team.* It is preferable to introduce a default team which is equal to a team of all images, eliminate SYNC ALL as a redundant construct, and use SYNC TEAM with the default *image-team* in its place. Skjellum et al. [15] argues that one should write libraries to run on arbitrary processor subsets and not just all processes; following his approach, SYNC ALL would be unused by libraries.

3.8 Refine the formation and use of image teams

Teams are an extremely useful concept for parallel programming. Section 13.7.68 of the proposed standard describes the `FORM_TEAM` subroutine. The specification of the `FORM_TEAM` subroutine introduces several difficulties:

- Team formation requires explicit enumeration of all images involved in the team on each processor. On a large scale system (such as a Blue Gene with 128K processors), having each image enumerate all images in the team is unwise.
- Images in the team are all specified by absolute image numbers. This makes it somewhat awkward to form subset teams.

MPI provides a model for a better way to form and manage teams, known as *process groups* in MPI parlance. Operations on MPI process groups include methods to interrogate the group size, one's rank within the group, and compare groups. Operations on groups include union, intersection, difference. Methods for creating groups include creating a group from only listed members of an existing group, all but listed members of an existing group, as well as methods for creating groups based on inclusion or exclusion of strided patterns. Also, one can map between a processor's rank in different groups.

Fortran 2008 teams would substantially benefit from a team abstraction more like MPI process groups. The ability to index team members by rank within a team rather than using absolute image rank better supports encapsulation. Many parallel algorithms on processor subsets are more natural if described in terms of point-to-point communication relative to rank in group [15] rather than an absolute coordinate such as global image index. Providing explicit control over image order within teams enables users to create teams that renumber images to arrange for a better embedding of logical topology into physical topology.

4 Major changes worth considering

4.1 Eliminate multiple co-dimensions

Multiple co-dimensions only provide the right abstraction for programs that use a k-D logical topology without periodic boundary conditions across *all* of the process images. In our opinion, multiple co-dimensions introduce more problems than they solve. Having to cope with the case when the number of processors is not divisible by the product of the leading co-dimensions is particularly awful. The abstraction that multiple co-dimensions provide is better provided by introducing an abstraction for logical communication topologies, as described next.

4.2 Logical communication topologies

In their paper *Writing MPI Libraries* [15], Sjkellum et al. advocate for virtual topologies as an alternative to simple rank naming for processors. MPI supports Cartesian and graph topologies, as well as extensions for user-defined topologies. Adding such virtual topology abstractions to Fortran 2008 would both simplify programming and facilitate compiler analysis.

We propose limiting co-arrays to 1D and adding lightweight language abstractions for an ordered group of processes (like CAF's image teams), functions for constructing new process groups, and support for Cartesian and graph virtual topologies. A neighbor function applied to a topology can be used to index communication partners, e.g. left and right along a dimension of a Cartesian topology, or predecessors and successors in a graph topology. Properties of topologies can be exploited for synchronization optimization. More details about a concrete proposal for logical topologies can be found elsewhere [3].

4.3 Global pointers for transparent manipulation of distributed data

In a highly dynamic and chaotic application, it may not always be clear on which image a particular datum resides. By providing a flattened view of co-arrays, globalized pointers offer a convenient way for programmers to construct and manipulate the dynamic complex data structures needed for this class of application.

4.4 Co-functions for efficient manipulation of remote data

Efficient manipulation of remote data on scalable parallel systems is a challenge. Rather than trying to tolerate latency, it is better to avoid exposing it. This can be achieved by function shipping (AKA remote procedure call, active messages) spawn foo(a,b,c,d)[p] could execute a call to co-function foo on image p. However, this introduces multithreading, which has implications for synchronization and parallelization.

Where latency tolerance allows a sending image to proceed without blocking if a dependent image has not yet reached a critical point in execution, latency avoidance requires that a dependent image be able to proceed quickly even if sender is not yet ready. This can be supported by function shipping. The syntax `spawn foo(a,b,c,d)[p]` could be used to indicate that the program should execute co-function `foo` with parameters `a,b,c,d` on image `p`.

There are two potential execution models for co-functions: interrupt-like activities or long-lived threads created upon demand. Function shipping raises the issue of what to do with references to variables referenced by a shipped function outside the scope of its parameters. A downside of allowing function shipping is that this admits concurrency within images, which will impose an additional burden on programmers. More details about function shipping in co-array Fortran can be found elsewhere [3].

4.5 Multi-version variables

Multi-version variables to which producers commit values and consumers retrieve them, provide a simpler abstraction for producer/consumer communication (e.g. halo updates and wave front computations). Otherwise, hiding communication latency when using a shared memory programming model can be difficult, requiring multiple buffers and overlapping the latency of "buffer full/empty" notifies with computation. In our experience, using multi-version variables dramatically simplified and streamlined application code. More details about multiversion variables in co-array Fortran can be found elsewhere [3].

5 Summary

In this white paper, we have presented some drawbacks of the proposed integration of co-arrays in Fortran 2008 and offered some recommendations about how to address these shortcomings. We believe that further refinement is needed of co-array support in Fortran 2008 before co-array features should be frozen in the Fortran standard.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. Technical Report 95-7, Digital Western Research Laboratory, Palo Alto, CA, 1995.

[2] T. U. Consortium. UPC language specification (v 1.2). `http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf`, June 2005.

[3] Y. Dotsenko. *Expressiveness, Programmability and Portable High Performance of Global Address Space Languages*. PhD thesis, Rice University, Houston, TX, Jan 2007.

[4] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 54–63, New York, NY, USA, 1989. ACM.

[5] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[6] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[7] J. Manson. Advanced topics in programming languages: The Java memory model. `http://video.google.com/videoplay?docid=8394326369005388010`, March 21 2007.

[8] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM.

[9] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[10] Message Passing Interface Forum. MPI_Op_Create. `http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Op_create.html`.

[11] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 1.1. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`, June 1995.

[12] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[13] J. Reid. Co-arrays in the next Fortran standard. J3 Committee WG5 Note N1647, Sept. 21 2007.

[14] M. L. Scott and J. M. Mellor-Crummey. Fast, contention-free combining tree barriers for shared-memory multiprocessors. *Int. J. Parallel Program.*, 22(4):449–481, 1994.

[15] A. Skjellum, N. E. Doss, and P. V. Bangalore. Writing libraries in MPI. In A. Skjellum and D. S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993.

A Loss of Encapsulation with a Single Synchronization Context

In *Working Draft J3/07-007r3*, `notify`/`query` synchronization provides each processor with only one logical synchronization channel between image pairs— `notify` and `query` each specify a partner image. Having only one logical channel prevents encapsulation. Here, we present an example to illustrate how the use of non-blocking `notify` and `query` fail to support encapsulation for library code.

To overlap communication and synchronization latency with computation, one might issue a non-blocking `notify` before a library call and `query` for its completion afterward. A loss of encapsulation occurs if a `notify` posted before a library call may be matched by a `query` inside the library call. The rest of this appendix provides a detailed example of how having only a single synchronization context can lead to loss of synchronization encapsulation for library code.

Figure 1(a) shows a pair of processes each executing the same code fragment. Assume that `me = thisimage()` and `P = numimages() = 2`. To keep the example's code simple, we assume that images are numbered 0 and 1, despite the fact that *Working Draft J3/07-007r3* says image numbers begin with 1. Let the left image in the figure be image 0 and the right image be image 1. Each image's executes the following sequence of operations:

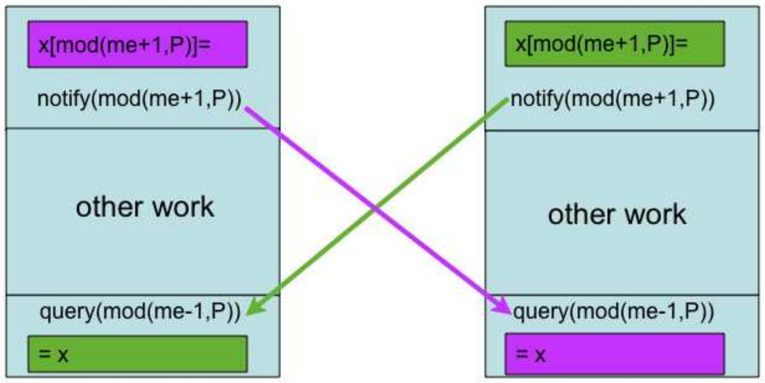
- The image assigns a value to the co-array `x` on its successor image on a ring, i.e., image `i` writes to `x` on image `mod(i+1, P)`.
- The image issues a `notify` to its successor, indicating that its co-array write is complete.
- The image overlaps communication and synchronization latency with computation by performing unspecified "other work" before waiting for the data from its predecessor.
- The image waits for a notification from its predecessor
- The image reads the value written by its predecessor.

In Figure 1(a), reads and writes of corresponding data elements are assigned the same color. For example, the magenta box in image 0 shows a write to `x` on image 1, and the magenta box in image 1's fragment shows the read of the same data. A colored arrow connects each corresponding `notify/query` pair; the color of the arrow corresponds to the color of the data element for which the `notify` signals write completion. This program fragment is properly synchronized. Neither image will read its local co-array data for `x` until after it is written by its predecessor; the `notify/query` pairing shown prevents each image's read of a co-array element from overlapping the write by its predecessor.

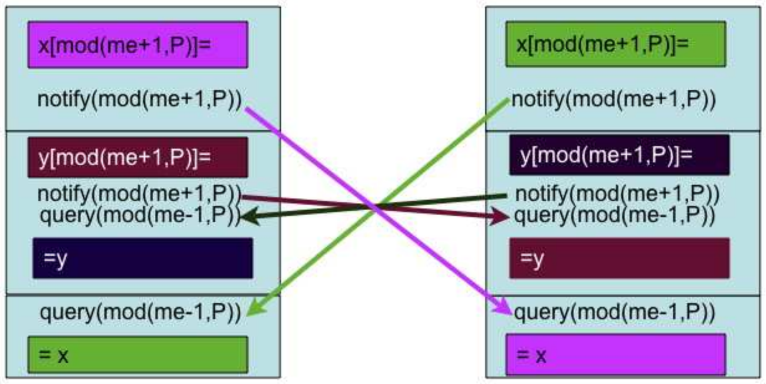
Figure 1(b) is a refinement of the program fragment shown in 1(a) in which the "other work" in 1(a) is replaced by a call to a library procedure that performs the same communication pattern as shown for the code 1(a), except with a different variable `y`. For expository purposes, Figure 1(b) shows the library procedure's code directly embedded in the fragment rather than separately. In (b), the correspondence shown by `notify/query` pairs shown is what the program designer intended.

Figure 1(c) shows the actual dynamic pairing of notifies and queries that occurs when a pair of images execute the program fragments shown in 1(b) and 1(c). Since there is only a single communication context, the i^{th} `notify` by each image pairs with the i^{th} `query` by its successor. Notice that the pairing of synchronization is not what the program designer intended. The `notify` operations before calls to the library code pair with `query` operations inside the library code. With this pairing of synchronization, the reads and writes of `y` by each image are not properly synchronized—they may occur concurrently.

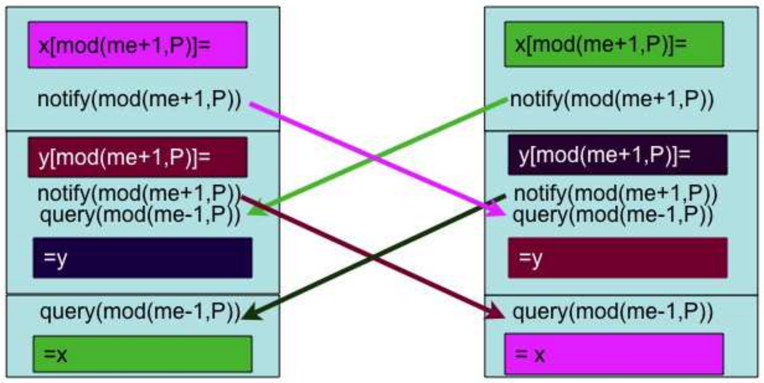
Thus, Figure 1 shows that with a single communication context, one cannot overlap communication and synchronization with a call to a library procedure without knowing that on no image does the library procedure perform notify/query synchronization between the same processor pairs as the synchronization in the user's code. Thus, this example shows that having only a single communication context per processor breaks encapsulation. *Multiple* synchronization contexts are necessary to support encapsulation.



(a)



(b)



(c)

Figure 1: Having only a single synchronization context per image leads to loss of encapsulation.