The Journal of Supercomputing, 22, 231–250, 2002

A Comparison of Co-Array Fortran and OpenMP Fortran for SPMD Programming

ALAN J. WALLCRAFT wallcraf@ajax.nrlssc.navy.mil

Naval Research Laboratory, Stennis Space Center, MS 39529

Abstract. Co-Array Fortran, formally called F^{--}, is a small set of extensions to Fortran 90/95 for Single-Program-Multiple-Data (SPMD) parallel processing. OpenMP Fortran is a set of compiler directives that provide a high level interface to threads in Fortran, with both thread-local and thread-shared memory. OpenMP is primarily designed for loop-level directive-based parallelization, but it can also be used for SPMD programs by spawning multiple threads as soon as the program starts and having each thread then execute the same code independently for the duration of the run. The similarities and differences between these two SPMD programming models are described.

Co-Array Fortran can be implemented using either threads or processes, and is therefore applicable to a wider range of machine types than OpenMP Fortran. It has also been designed from the ground up to support the SPMD programming style. To simplify the implementation of Co-Array Fortran, a formal Subset is introduced that allows the mapping of co-arrays onto standard Fortran arrays of higher rank. An OpenMP Fortran compiler can be extended to support Subset Co-Array Fortran with relatively little effort.

Keywords: Co-Array, Fortran, OpenMP, SPMD

1. Introduction

In Single-Program-Multiple-Data (SPMD), a single program is replicated a fixed number of times, each replication having its own set of data objects. For example, in SPMD domain decomposition a region is divided into nonoverlapping sub-regions and each program replication is assigned its own sub-region. The best known SPMD Application Programming Interface (API) is the Message Passing Interface (MPI) library [10]. Like most subroutine library based SPMD APIs, an apparently stand-alone program is compiled as if for a single processor but replicated on MPI startup with each replication executing asynchronously except when communicating with another replication by calling a MPI library routine. A key feature of a message passing library, such as MPI, is that two replications must cooperate (i.e. both make a subroutine call) if communication is to take place between them. This approach is widely used, but there are inconsistencies caused by the compiler assuming a single replication when in fact there are many. In addition, message passing is intrinsically slower and harder to program than direct memory copies on machines with a hardware global memory. MPI-2 provides a put/get capability that can in principle take advantage of direct memory copies, but it relies on C procedures whose calling mechanism differs from that of Fortran and the put/get syntax is further obscured by the need to be compatible with similar message passing subroutines [5]. Both Co-Array Fortran and OpenMP Fortran replace message passing with direct

memory access expressed via assignment statements. They are therefore potentially faster than MPI on machines with a hardware global memory. In both cases, the compiler is to some extent aware that the program is SPMD which avoids either all (Co-Array Fortran) or some (OpenMP Fortran) of MPI's incompatibilities with Fortran.

Section 2 is a brief overview of Co-Array Fortran and OpenMP Fortran. Section 3 describes a simple example program. Section 4 compares and contrasts the languages in more detail. Section 5 introduces Subset Co-Array Fortran, and describes how the Subset can be automatically translated into OpenMP Fortran.

2. Language overview

2.1. Co-Array Fortran

Co-Array Fortran is a simple syntactic extension to Fortran 90/95 that converts it into a robust, efficient parallel language [6]. It looks and feels like Fortran and requires Fortran programmers to learn only a few new rules. The few new rules are related to two fundamental issues that any parallel programming model must resolve, work distribution and data distribution.

First, consider work distribution. A single program is replicated a fixed number of times, each replication having its own set of data objects. Each replication of the program is called an **image**. Each image executes asynchronously and the normal rules of Fortran apply, so the execution path may differ from image to image. The programmer determines the actual path for the image with the help of a unique image index, by using normal Fortran control constructs and by explicit synchronizations. For code between synchronizations, the compiler is free to use all its normal optimization techniques, as if only one image is present.

Second, consider data distribution. The co-array extension to the language allows the programmer to express data distribution by specifying the relationship among memory images in a syntax very much like normal Fortran array syntax. One new object, the **co-array**, is added to the language. For example,

```
REAL, DIMENSION(N)[*] :: X,Y
X(:) = Y(:)[Q]
```

declares that each image has two real arrays of size `N`. If `Q` has the same value on each image, the effect of the assignment statement is that each image copies the array `Y` from image `Q` into its local array `X`. Co-Arrays cannot be constants or pointers or automatic data objects, and they always have the SAVE attribute unless they are allocatable or dummy arguments.

Array indices in parentheses follow the normal Fortran rules within one memory image. Array indices in square brackets provide an equally convenient notation for accessing objects across images and follow similar rules. Bounds in square brackets in co-array declarations follow the rules of assumed-size arrays since co-arrays are

always spread over all the images. The programmer uses co-array syntax only where it is needed. A reference to a co-array with no square brackets attached to it is a reference to the object in the local memory of the executing image. Since most references to data objects in a parallel code should be to the local part, co-array syntax should appear only in isolated parts of the code. If not, the syntax acts as a visual flag to the programmer that too much communication among images may be taking place. It also acts as a flag to the compiler to generate code that avoids latency whenever possible.

Input/output has been a problem with previous SPMD APIs, such as MPI, because standard Fortran I/O assumes dedicated single-process access to an open file and this constraint is often violated when the API assumes that I/O from each image is completely independent. Co-Array Fortran includes only minor extensions to Fortran 90/95 I/O, but all the inconsistencies of earlier APIs have been avoided and there is explicit support for parallel I/O. In addition I/O is compatible with implementations that map images onto either threads or processes (or a combination of the two).

The only other additions to Fortran 90/95 are several intrinsics. For example: the integer function `NUM_IMAGES()` returns the number of images, the integer function `THIS_IMAGE()` returns this image's index between 1 and `NUM_IMAGES()`, and the subroutine `SYNC_ALL()` is a global barrier which requires all operations before the call on all images to be completed before any image advances beyond the call. In practice it is often sufficient, and faster, to only wait for the relevant images to arrive. `SYNC_ALL(WAIT)` provides this functionality. There is also `SYNC_TEAM(TEAM)` and `SYNC_TEAM(TEAM,WAIT)` for cases where only a subset, `TEAM`, of all images are involved in the synchronization. The intrinsics `START_CRITICAL` and `END_CRITICAL` provide a basic critical region capability. It is possible to write your own synchronization routines, using the basic intrinsic `SYNC_MEMORY`. This routine forces the local image to both complete any outstanding co-array writes into "global" memory and refresh from global memory any local copies of co-array data it might be holding (in registers for example). A call to `SYNC_MEMORY` is rarely required in Co-Array Fortran, because there is an implicit call to this routine before and after virtually all procedure calls including Co-Array's built in image synchronization intrinsics. This allows the programmer to assume that image synchronization implies co-array synchronization.

2.2. OpenMP Fortran

OpenMP Fortran is a set of compiler directives that provide a high level interface to threads in Fortran, with both thread-local and thread-shared memory. Most compilers are now complient with version 1.1 of the specification [8], which will be discussed here unless otherwise noted. Version 2.0 [9] was released in November 2000 but is not yet widely available. OpenMP can also be used for loop-level directive based parallelization, but in SPMD-mode N threads are spawned as soon as the program starts and exist for the duration of the run. The threads act like Co-Array images (or MPI processes), with some memory private to a single thread and other

memory shared by all threads. Variables in shared memory play the role of co-arrays in Co-Array Fortran, i.e. if two threads need to "communicate" they do so via variables in shared memory. Local non-saved variables are thread private, and all other variables are shared by default. The directive `!$OMP THREADPRIVATE` can make a named common private to each thread.

Threaded I/O is well understood in C [3], and many of the same issues arise with OpenMP Fortran I/O. A single process necessarily has one set of I/O files and pointers. This means that Fortran's single process model of I/O is appropriate. I/O is "thread safe" if multiple threads can be doing I/O (i.e., making calls to I/O library routines) at the same time. OpenMP Fortran requires thread safety for I/O to distinct unit numbers (and therefore to distinct files), but not to the same I/O unit number. A SPMD program that writes to the same file from several threads will have to put all such I/O operations in critical regions. It is therefore not possible in OpenMP to perform parallel I/O to a single file.

The integer function `OMP_GET_NUM_THREADS()` returns the number of threads, the integer function `OMP_GET_THREAD_NUM()` returns this thread's index between 0 and `OMP_GET_NUM_THREADS()-1`. The compiler directive `!$OMP BARRIER` is a global barrier which requires all operations before the barrier on all threads to be completed before any thread advances beyond the call. The directives `!$OMP CRITICAL` and `!$OMP END CRITICAL` provide a critical region capability, with more flexiblity than that in Co-Array Fortran, and in addition there are intrinsic routines for shared locks that can be used for the fine grain synchronization typical of threaded programs [3]. The directives `!$OMP MASTER` and `!$OMP END MASTER` provide a region that is executed by the master thread only, `!$OMP SINGLE` and `!$OMP END SINGLE` identify a region executed by a single thread. Note that all directive defined regions must start and end in the same lexical scope. It is possible to write your own synchronization routines, using the basic directive `!$OMP FLUSH`. This routine forces the thread to both complete any outstanding writes into memory and refresh from memory any local copies of data it might be holding (in registers for example). It only applies to "thread visible" variables in the local scope, and can optionally include a list of exactly which variables it should be applied to. `BARRIER, CRITICAL`, and `END CRITICAL` all imply `FLUSH`, but unlike Co-Array Fortran it is not automatically applied around subroutine calls. This means that the programmer has to be very careful about making assumptions that thread visible variables are current. Any user-written synchronization routine should be preceeded by a `FLUSH` directive every time it is called.

A subset of OpenMP's loop-level directives, that automate the allocation of loop iterations between threads, are also available to SPMD programs but are not typically used.

Unlike High Performance Fortran (HPF) [4], which has compiler directives that are carefully designed to not alter the meaning of the underlying program, the OpenMP directives used in SPMD-threaded programming are declaration attributes or executable statements. They are still properly expressible as structured comments, starting with the string "`!$OMP`", because they have no effect when the program has exactly one thread. But they are not "directives" in the conventional sense. For example "`!$OMP BARRIER`" does not allow any thread to continue until all

have reached the statement. When there is more than one thread, SPMD OpenMP defines a new language that is different from uni-processor Fortran in ways that are not obvious by inspection of the source code. For example:

1. Saved local variables are always shared and non-saved local variables are always threadprivate. It is all too easy to inadvertently create a saved variable. For example, in Fortran 90/95 initializing a local variable, e.g., `INTEGER :: I=0`, creates a saved variable. A DATA statement has a similar effect in both Fortran 77 and Fortran 90/95. In OpenMP such variables are always shared, but often the programmer's intent was to initialize a threadprivate variable (which is not possible with local variables in version 1.1).
2. In version 1.1, only common can be either private or shared under programmer control. Module variables, often used to replace common variables in Fortran 90/95, are always shared. Version 2.0 allows individual saved and module variables to be declared private.
3. ALLOCATE is required to be thread safe, but because only common variables can be both private and non-local, it is difficult to use ALLOCATABLE for private variables. A pointer in THREADPRIVATE common may work, but is not a safe alternative to an allocatable array.
4. It is up to the programmer to avoid race conditions caused by the compiler using copy-in/copy-out of thread-shared array section subroutine arguments.
5. There is no way to document the default case using compiler directives. There is a `!$OMP THREADPRIVATE` directive but no matching optional `!$OMP THREADSHARED` directive. Directives that imply a barrier have an option, `NOWAIT`, to skip the barrier but no option, `WAIT`, to document the default barrier.
6. Sequential reads from multiple threads must be in a critical region for thread safety and provide a different record to each thread. In all process-based SPMD models sequential reads from multiple processes provide the same record to each process.

SPMD OpenMP is not a large extension to Fortran but OpenMP programs cannot be maintained by Fortran programmers unfamiliar with OpenMP. For example, a programmer has to be aware that adding a DATA statement to a subroutine could change the multi-thread behavior of that subroutine. In contrast, adding a DATA statement, or making any other modifications, to a Co-Array Fortran program is identical in effect to making the same change to a Fortran 90/95 program providing no co-arrays are involved (i.e., providing no square brackets are associated with the variable in the local scope).

Version 2.0 of the specification adds relatively few capabilities for SPMD programs, but the extension of `THREADPRIVATE` from named common blocks to saved and module variables will provide a significantly improved environment particularly for Fortran 90 programmers. It is unfortunate that there is still no way to document the default via a similar `THREADSHARED` directive. If this existed, the default status of variables would cease to be an issue because it could be confirmed or overridden with compiler directives. The lack of fully thread safe I/O places an unnecessary

burden on the SPMD programmer. The standard should at least require that thread safe I/O be available as a compile time option. This is much easier for the compiler writer to provide, either as a thread-safe I/O library or by automatically inserting a critical region around every I/O statement, than the application programmer. The sequential read limitation is a basic property of threads, and is primarily an issue because many Fortran programmers are familiar with process-based SPMD APIs. Version 2.0 has a `COPYPRIVATE` directive qualifier that handles this situation cleanly. For example:

```
!$OMP SINGLE
      READ(11) A,B,C
!$OMP END SINGLE, COPYPRIVATE(A,B,C)
```

Here "`A,B,C`" are threadprivate variables that are read on one thread and then copied to all other threads by the `COPYPRIVATE` clause at the end of the single section. Co-Array Fortran I/O is designed to work with threads or processes, and a proposed extension can handle this case:

```
READ(11,TEAM=ALL) A,B,C
```

All images in the team perform the identical read and there is implied synchronization before and after the read. If images are implemented as threads, the I/O library could establish a separate file pointer for each thread and have each thread read the file independently or the read could be performed on one thread and the result copied to all others.

The limitations of OpenMP are more apparent for SPMD programs than for those using loop-level directives, which are probably the primary target of the language. SPMD programs are using *orphan* directives, outside the *lexical* scope of the parallel construct that created the threads [8]. OpenMP provides a richer set of directives within a single lexical scope, which allow a more complete documentation of the exact state of all variables. However, it is common to call a subroutine from within a do loop that has been parallelized and the variables in that subroutine have the same status as those in a SPMD subroutine. Also, almost all OpenMP compilers support Fortran 90 or 95, rather than Fortran 77, but version 1.1 directives largely ignore Fortran 95 constructs. Version 2.0 has more complete Fortran 95 support, which provides an incentive for compilers to be updated to version 2.0.

3. A simple example

The calculation of π was used as an example in the original OpenMP proposal [7], which presented three versions using OpenMP's loop level parallelization constructs, using MPI, and using pthreads. SPMD versions using Co-Array Fortran and OpenMP Fortran are presented here. First Co-Array Fortran:

```
program compute_pi
double precision :: mypi[*],pi,psum,x,w
integer          :: n[*],me,nimg,i
```

```
nimg = num_images()
me   = this_image()

if (me==1) then
   write(6,*) 'Enter number of intervals'; read(5,*) n
   write(6,*) 'number of intervals = ',n
   n[:] = n
endif
call sync_all(1)

w = 1.d0/n; psum = 0.d0
do i= me,n,nimg
   x = w * (i - 0.5d0); psum = psum + 4.d0/(1.d0+x*x)
enddo
mypi = w * psum
call sync_all()

if (me==1) then
   pi = sum(mypi[:]); write(6,*) 'computed pi = ',pi
endif
call sync_all(1)
end
```

The number of intervals and the partial sums of π are declared as co-arrays, because these must be communicated between images. All other variables are local to each image. The number of intervals is input on image 1 and broadcast to all images. Note that `n` without square brackets refers to the local part, `n[me]`. All images wait at the first `sync_all` for image 1 to arrive, signaling that `n` is safe to use. Each image then waits at the second `sync_all` for all images to complete the calculation. Finally, the first image adds the co-array of partial sums and writes out the result. The final `sync_all` prevents the other images from terminating the program before image 1 completes the write.

In OpenMP Fortran this becomes:

```
     program main
     call omp_set_dynamic( .false.)
     call omp_set_nested( .false.)
!$omp parallel
     call compute_pi
!$omp end parallel
     stop
     end
     subroutine compute_pi
     double precision   :: psum,x,w  ! threadprivate
     integer            :: me,nimg,i ! threadprivate
     double precision   :: pi
     integer            :: n
     common       /pin/    pi,n
```

```
!*omp threadshared(/pin/)
      integer omp_get_num_threads,omp_get_thread_num

      nimg = omp_get_num_threads()
      me   = omp_get_thread_num() + 1

!$omp master
        write(6,*) 'Enter number of intervals'; read(5,*) n
        write(6,*) 'number of intervals = ',n
        pi = 0.d0
!$omp end master
!$omp barrier

      w = 1.d0/n; psum = 0.d0
      do i= me,n,nimg
         x = w * (i - 0.5d0); psum = psum + 4.d0/(1.d0+x*x)
      enddo
!$omp critical
        pi = pi + (w * psum)
!$omp end critical
!$omp barrier

!$omp master
        write(6,*) 'computed pi = ',pi
!$omp end master
!$omp barrier
      end
```

All SPMD OpenMP programs start with the same main program. It spawns the number of threads specified by the environment variable `OMP_NUM_THREADS`, then immediately calls the top level subroutine that represents the actual program to replicate. On exit from this subroutine all threads except the master thread are freed and the program then exits. The number of intervals and π are declared in named common and are therefore global (thread-shared) variables by default. There is no compiler directive available to confirm the default, so a pseudo-directive, `!*omp threadshared`, is used to document that the common is shared. All other variables are local to the subroutine and therefore private to each thread (no saved variables). The number of intervals is input on the master thread, and since `n` is a global variable it is automatically available on all threads. All threads wait at the first `!$omp barrier` for the master thread to arrive, signaling that `n` is safe to use. Each thread then independently calculates its part of π and adds it to the total π. Updating π is in a critical region, so that only one thread at a time can access π. Each thread then waits at the second `!$omp barrier` for all threads to complete the calculation. Finally, the master thread writes out the result. The final `!$omp barrier` prevents the other threads from terminating the program before the master completes the write. This is probably unnecessary, since it is the master that will execute `stop` in the main program.

A relatively minor difference between the two versions is that Co-Array Fortran has a richer set of synchronization operations. In many cases, `sync_all(1)` is significantly faster than `sync_all()` because the former allows the image to continue as soon as image 1 arrives and the latter requires the image to wait for all images to arrive. OpenMP's `!$omp barrier` is the only synchronization of its kind provided by OpenMP and is equivalent to `sync_all()`. A synchronization routine like `sync_all(wait)` can be written in OpenMP, provided it is always called in conjunction with a `!$omp flush` directive. The primary difference between the two versions is that global variables are co-arrays spread across all images in Co-Array Fortran, but are standard variables in global memory (not assigned to any particular thread) in OpenMP Fortran. However, the difference is more one of style than substance. The OpenMP version can be rewritten in Co-Array Fortran, by only using the part of each co-array on image 1:

```
program compute_pi
double precision   :: psum,x,w
integer            :: me,nimg,i
double precision   :: pi[*]  ! only use pi[1]
integer            :: n[*]   ! only use n[1]

nimg = num_images()
me   = this_image()

if (me==1) then
   write(6,*) 'Enter number of intervals'; read( 5,*) n
   write(6,*) 'number of intervals = ',n
   pi = 0.d0
endif
call sync_all()

w = 1.d0/n[1]; psum = 0.d0
do i= me,(n[1]),nimg
   x = w * (i - 0.5d0); psum = psum + 4.d0/(1.d0+x*x)
enddo
call start_critical()
   pi[1] = pi[1] + (w * psum)
call end_critical()
call sync_all()

if (me==1) then
   write(6,*) 'computed pi = ',pi
endif
call sync_all()
end
```

In order to emulate shared variables, the Co-Array Fortran code replicates them on all images but only uses the part on image 1. All references to such variables must end in `[1]`. In the case of large shared arrays, it would be possible to avoid the space this wastes by defining a co-array of a derived type with a pointer component and

then only allocating an array to the pointer on image 1. This sounds complicated, but is in fact the standard way for Fortran 90/95 to handle an array of arrays (or in this case a co-array of arrays). The master directive in OpenMP is replaced by a test for the first image. Co-Array Fortran does not treat the first image any differently than the others (i.e., it has no master image). However, standard input is available on the first image only, so if the master's tasks include reading standard input Co-Array Fortran must use the first image as the master.

Similarly, the Co-Array version can be expressed in OpenMP by adding a per-thread dimension to each shared variable:

```
      program main
      call omp_set_dynamic( .false.)
      call omp_set_nested(  .false.)
!$omp parallel
      call compute_pi
!$omp end parallel
      stop
      end
      subroutine compute_pi
      integer, parameter :: max_threads=128
      double precision   :: pi,psum,x,w
      integer            :: me,nimg,i
      double precision   :: mypi
      integer            :: n
      common       /pin/    mypi(max_threads),n(max_threads)
!*omp threadshared(/pin/)
      integer omp_get_num_threads,omp_get_thread_num

      me   = omp_get_thread_num() + 1
      nimg = omp_get_num_threads()

      if (me==1) then
        if (nimg>max_threads) then
          write(6,*) 'error - too many threads ',nimg
          stop
        endif
        write(6,*) 'Enter number of intervals';
          read(5,*) n(me)
        write(6,*) 'number of intervals = ',n(me)
        n(1:nimg) = n(me)
      endif
!$omp flush
      call caf_sync_all(1)

      w = 1.d0/n(me); psum = 0.d0
      do i= me,n(me),nimg
        x = w * (i - 0.5d0); psum = psum + 4.d0/(1.d0+x*x)
      enddo
```

```
     mypi(me) = w * psum
!$omp barrier
     if (me==1) then
       pi=sum(mypi(1:nimg)); write(6,*) 'computed pi=',pi
     endif
!$omp flush
     call caf_sync_all(1)
     end
```

In order to emulate co-arrays, the OpenMP code puts them in named common (i.e. makes them shared variables) and converts co-array dimensions into additional regular array dimensions. Since array size has to be known at compile time, the parameter `max_threads` is introduced which has to be no smaller than the actual number of threads at run time. If this is set to a safe value, e.g., the number of processors on the machine, it is probably an over estimate and hence wastes memory. Co-Array Fortran allows the local part of a co-array to be referenced without square brackets, but all references to emulated co-arrays must include the co-dimensions, e.g. `n(me)`. It also "knows" that the co-size is `num_images()`, so `mypi[:]` is legal Co-Array Fortran but must become `mypi(1:nimg)` in OpenMP Fortran. The routine `caf_sync_all` is assumed to be an OpenMP implementation of `sync_all` but it can only synchronize threads, the `!$omp flush` is also required to synchronize shared objects.

4. A comparison

The features of SPMD OpenMP Fortran and Co-Array Fortran are summarized in Table 1. OpenMP Fortran is only applicable to systems with a single global memory space, and perhaps only to those with flat single level addressing and cache coherence across the entire memory space (i.e., systems such as the Cray T3E are not candidates for OpenMP). However, this includes a wide range of SMP and DSM systems with from 2 to 256 processors. OpenMP is a relatively new "standard," but it has wide vendor and third party support is available on almost all machines with a suitable global shared memory, from PC's to MPP's. Compilers with partial support for OpenMP typically do not support it in SPMD-mode, but most compilers now claim full version 1.1 complience. Version 2.0 complient compilers are not yet typically available, but for SPMD programmers only the extension of `THREADPRIVATE` to saved and module variables and the new `COPYPRIVATE` clause are significant, so even partial support for version 2.0 may be sufficient.

Co-Array Fortran can take full advantage of a hardware global memory, but it can also be used on shared nothing systems with physically distinct memories connected by a network. However, performance is expected to only be about as good as MPI on such systems. Co-Array Fortran can be implemented using threads or processes, or on a cluster of SMP systems it could even use threads within a SMP system and processes between systems. It is therefore more widely applicable than OpenMP Fortran. However, the Cray T3E is the only machine with a Co-Array Fortran compiler today and it implements only a subset of the language. There is

Table 1. Features of SPMD OpenMP Fortran and Co-Array Fortran

Feature	SPMD OpenMP Fortran	Co-Array Fortran
availability	wide-spread	Cray T3E only
implementable using	threads	threads and/or processes
target memory architecture	cache-coherent global	any
on single thread/image, get	standard Fortran	extended Fortran
incompatible Fortran extension	copy-in/out	none
local variables	private	private
saved and module variables	shared	private (or co-array)
common variables	shared (or private)	private (or co-array)
pointers	local or global	local
communication	shared variables	co-arrays
memory synchronization	local scope	global scope
memory layout control	automatic for private none for shared	automatic for private automatic for co-arrays
synchronization	global critical regions, locks	global or team (local short cut) critical region
I/O namespace	single and shared	single but private
I/O operations	unsafe to same unit	safe

a definite need for a source to source compiler that will allow Co-Array Fortran to run on the same systems as OpenMP Fortran. This is discussed in more detail in Section 5.

Any Co-Array Fortran program is translatable into OpenMP Fortran and vice versa. However, the differences between co-arrays and shared arrays tend to steer programmers to alternative solutions to the same problem. For example, suppose there are `P` images or threads and we need to perform operations both on an array, `A(1:M,1:N)`, and its transpose, `AT(1:N,1:M)`. For simplicity further assume that both `M` and `N` are multiples of `P`. In Co-Array Fortran we would probably store both as co-arrays, `A(1:M,1:N/P)[*]` and `AT(1:N,1:M/P)[*]`, and write a routine to copy between them. Since `A` and `AT` are co-arrays, the routine can do a direct copy without using intermediate buffers. In OpenMP, for efficiency of memory layout we might similarly store both as private arrays on each thread, `A(1:M,1:N/P)` and `AT(1:N,1:M/P)`, but now we have to provide a shared buffer to copy between them. The simplest shared buffer to use is the entire array, `B(1:M,1:N)`. Then the copy routine is just copy each private `A` into the shared `B`, barrier, copy the shared `B` into each private `AT`. An alternative in OpenMP is to always store the array as a whole shared array, `A(1:M,1:N)`. It may then be unnecessary to store the transpose at all, although cache effects may make it advisable to also have a shared transpose, `AT(1:N,1:M)`. The shared array approach is also available in Co-Array Fortran by placing arrays on one image, but to avoid wasting memory a co-array of arrays, `CA[1]%A(1:M,1:N)`, would probably be used rather than a simple co-array, `A(1:M,1:N)[1]`. In either case, the co-array syntax makes clear that accessing the "shared" array is a potentially expensive remote memory operation. The shared array approach is sometimes the easiest to use, and is more cleanly expressible in OpenMP Fortran, but it comes at the cost of less programmer control over performance.

OpenMP Fortran contains no directives to control the layout of shared arrays in memory. This is not an issue on SMP systems with uniform memory access, but where in memory shared arrays are placed may have a large effect on performance on non-uniform memory access (NUMA) systems. This limits OpenMP's scalability to large numbers of nodes, since large node-count systems tend to be NUMA. OpenMP Fortran is primarily designed for fine grained loop parallelization, which is typically appropriate for small node counts. Therefore the lack of layout control is less of an issue for OpenMP in its primary domain of interest, but it is a concern for SPMD programs. All memory in Co-Array Fortran is associated with a local image, so memory placement on NUMA systems is simple to arrange and does not effect scalability. Since Co-Array Fortran always knows when remote communication is involved, the global memory does not need to be cache coherent and in fact each image's memory can be physically and logically distinct with only a fast network connecting them. Overall, Co-Array Fortran has clear advantages on systems with large node counts (above about 32 processors).

Co-Array Fortran is a simple set of extensions to Fortran 90/95. The features that are compatible with Fortran 77 do not produce a viable subset language. OpenMP compilers typically support Fortran 90 or 95, but the version 1.1 compiler directives really only apply well to Fortran 77 programs. The lack of support for thread private module variables and for ALLOCATABLE are two examples of this. Fortran 77 is probably still the dominant variant for SPMD programs, but large projects, in particular, are increasingly migrating to Fortran 95 and will need version 2.0 OpenMP compilers.

One example of Co-Array Fortran's reliance on Fortran 90/95 features is that any subroutine with a co-array dummy argument must have an explicit interface. Hence a Fortran 77 subset would either have to ban co-array dummy arguments or provide an extension to the existing language that distinguishes between co-array actual arguments and local part actual arguments without an explicit interface. Explicit interfaces make Co-Array Fortran significantly safer to use. The compiler always has complete knowledge of co-arrays except in the special case when the local part of a co-array is passed to a dummy argument of co-rank zero. The programmer is responsible for co-array safety in this special case, and must make sure that no other image references the passed piece of the co-array between the subroutine call and return. A synchronization call in Co-Array Fortran always implies that all co-arrays are up to date (except those passed to co-rank zero dummies, and these must not be referenced from other images anyway). All the co-array "machinery" works behind the scenes to allow the programmer to do the obvious thing and in fact get the expected result.

OpenMP Fortran provides a significantly lower level programmer interface. Once an object has been passed to a procedure through its argument list there is no way to tell if it is a shared object or a private object. Pointers can be shared or private and both can reference either shared or private variables, so it is possible (although unsafe) for one thread to access the private memory of another thread. Also, synchronization primitives only apply to shared objects in the local scope. All subroutine arguments are potentially "thread visible" (i.e. shared), so all have to be flushed even though some may actually be private. Local scope synchronization

has several pitfalls to trap the unwary programmer. One of the most obvious is copy-in/copy-out:

```
      common/shared/ i(100)
!$omp master
      i(1:100) = 0
!$omp end master
!$omp barrier
      call sub1(i(2:100:2))
!$omp master
      write(6,*) i(4)
!$omp end master
      end
      subroutine sub1(i2)
      integer :: i2(0:49),omp_get_thread_num
      i2(omp_get_thread_num()) = omp_get_thread_num()
!$omp barrier
      end
```

The barrier in `sub1` synchronizes `i2`, but, since it is not an assumed shape array, `i2` is probably only a local contiguous copy of `i(2:100:2)` and a different local copy on every thread (the only practical alternative a compiler has is to in-line `sub1`). If a local copy is used, the barrier has no effect on `i` and when each thread returns from `sub1` they will independently copy back their entire local version of `i(2:100:2)` into the shared original and perhaps also update a register holding i(4) from the local version. This means that there is no way to tell if the write prints the value 1, as expected, or 0. This is not just a local scope issue, since the problem remains even if the second barrier is moved from inside `sub1` to just after the call to `sub1`. The value of `i(4)` then depends on which thread exits `sub1` last. The only safe approach is for the programmer to manually implement steps similar to those that Co-Array Fortran takes. Issue a `!$omp flush` before and after every subroutine call that might contain synchronization, and if a shared array section that is not contiguous in array element order is passed to a subroutine the associated dummy array argument must be assumed shape (and the subroutine interface therefore explicit). The OpenMP specification makes the above example illegal, i.e. it places the responsibility onto the programmer to avoid such copy-in/copy-out race conditions. Co-Array Fortran guarantees that copy-in/copy-out is never required for co-array dummy arguments, e.g., if `i2` were a co-array an array section actual argument would be illegal (and detectable as an error at compile time) unless `i2` is declared assumed shape. All library-based SPMD APIs have similar consistency problems. The MPI-2 standard [5] has a good discussion of these issues, which can cause optimization problems in Fortran 77 but are much more serious for Fortran 90/95. Co-Array Fortran may be unique among SPMD APIs in having no known conflicts with Fortran 90/95. High Performance Fortran is also consistent with Fortran 95, but is not formally a SPMD API although it is often implemented using SPMD.

The previously listed limitations make OpenMP Fortran a less than optimal choice for very large SPMD programs, but it has the important advantage of being widely available. There is a preliminary port of the NRL Layered Ocean Model (NLOM) to OpenMP Fortran [11]. NLOM already ran in SPMD-mode using MPI or SHMEM. The original code is 69,000 lines of Fortran 77 including 22,000 comment lines of which 500 are compiler directives (many are repeats in different dialects). Ignoring communication routines, adding support for OpenMP required 900 OpenMP compiler directives, 500 to characterize all COMMON's (could be reduced using INCLUDE) and 400 primarily to handle I/O. This illustrates a general property of compiler directive based APIs, they are very verbose. Other, extensive, changes were required to allow sequential I/O to be compatible with either SPMD processes or SPMD threads. Programs that do all sequential I/O from a single image would not require these modifications. If the `COPYPRIVATE` directive qualifier had been available the sequential I/O modifications would have been greatly simplified. Shared variables were added to handle sequential I/O, but in general variables outside communication routines are `THREADPRIVATE`. Fortunately, NLOM does not use modules and most saved local variables had already been placed in common. However, `DATA` statement initialization had to be modified to make sure there were no implied saved local variables. Since this was a prototype port, the required communication routines were generated by replicating the existing 4,000 line SHMEM version and making as few changes as possible to support OpenMP.

The native OpenMP port of NLOM has been made obsolete by adding support to NLOM for a dialect of Co-Array Fortran that can be automatically translated into OpenMP Fortran using a nawk [2] script (described in more detail below). Outside communication routines, this involved adding macros (that are null except when using Co-Array Fortran) to 230 I/O statements and adding Co-Array syntax to a single subroutine that defines arrays (co-arrays) used by communication routines. Inside communication routines, this involved replicating and modifying SHMEM-specific code fragments for Co-Array Fortran (with each SHMEM library call mapping to a single co-array assignment statement), and adding a macro identifying the local part of a co-array to 375 assignment statements. The latter are only required due to limitations in the nawk script and are null when using a true Co-Array Fortran compiler. The total effort involved in writing the nawk script and adding Co-Array Fortran support to NLOM was significantly less than required to add native OpenMP support. The nawk script adds all required OpenMP compiler directives and emulates Co-Array Fortran I/O, thus removing the two most time consuming aspects of the port.

NLOM is typical of many SPMD codes in using a program-specific interface to handle all communication. This greatly simplifies porting to a new SPMD API, but reduces the opportunity for optimization with a low latency API (such as either Co-Array and OpenMP Fortran). Porting an existing SPMD program, e.g. one using MPI, that did not separate out communication to OpenMP Fortran would be difficult, because MPI allows communication between what are in OpenMP terms `THREADPRIVATE` objects and in fact has no concept of `THREADSHARED` objects. Porting any existing SPMD program to Co-Array Fortran would be much easier, because all objects including co-arrays can be treated as local to an image

and co-arrays need only be introduced at all for objects that are involved in communication.

5. Translation

5.1. Subset Co-Array Fortran

The full Co-Array Fortran language [6] provides support for legacy SPMD programs based on Cray's SHMEM put and get library [1]. It requires that all variables in named COMMON be treatable either as standard variables or as co-arrays, and which objects in the COMMON block are co-arrays is allowed to vary between scoping units. This makes it difficult to implement co-arrays as if they were Fortran arrays of higher rank. The following relatively minor restrictions on the full language define a formal Subset that significantly widens the implementation choices, and in particular allows a simple mapping from Co-Array Fortran to OpenMP Fortran.

1. If a named COMMON includes a co-array, every object in that COMMON must be a co-array. The objects in the COMMON must agree in size, type, shape, co-rank and co-extents in all scoping units that contain the COMMON.
2. The EQUIVALENCE statement is not permitted for co-arrays.
3. The sum of the local rank plus the co-rank of a co-array is limited to seven.
4. A dummy co-array argument cannot have assumed size local dimensions.
5. A dummy co-array argument cannot have assumed shape local dimensions, unless the co-rank is one. The actual argument shall then also have co-rank one.
6. If a dummy argument has both nonzero local rank and nonzero co-rank and does not have assumed shape local dimensions, the actual argument must agree in size and type with the dummy argument.

The restrictions on COMMON are similar to nonsequence COMMON in HPF [4]. The restrictions on EQUIVALENCE are more severe than in HPF, for simplicity, but in Subset Co-Array Fortran the restrictions only apply to co-arrays. COMMON can be largely replaced by MODULE for new programs, so the restrictions are easily met except when migrating legacy Fortran 77 programs that make heavy use of COMMON and either EQUIVALENCE or different layouts for the same named common in different scopes. If the objects in a legacy COMMON already agree in size, type, and shape in all scoping units (which is good programming practice), then every object in that COMMON can be converted to a co-array without changing the meaning of the program. This is because a reference to a co-array without square brackets is always a reference to the local part of the co-array. Migration to the Subset is therefore easy in this case.

In Co-Array Fortran both the local rank and the co-rank of a co-array can be seven, but the local rank plus co-rank of any co-array subobject that is actually used in an executable statement must be no more than seven (because the rank and co-rank are merged and the object treated as a standard array subobject). Thus the Subset's restriction on local rank plus co-rank to seven is not typically a severe

additional constraint. It would be helpful if Fortran 2000 increased the limit on the rank from seven to, say, ten, since this would give more room for rank plus co-rank.

The restrictions on dummy co-array arguments may require the programmer to explicitly pass additional array dimension information through the argument list. The restriction on assumed shape is probably the most severe of all for new programs, since assumed shape arrays are a significant simplifying factor in Fortran 90/95 programs and Co-Array Fortran requires some kinds of co-array actual arguments to only be associated with assumed shape dummy arguments. It is a consequence of the fact that co-size is always `NUM_IMAGES()` and therefore that, when the co-rank is greater than one, the co-array has no final extent, no final upper bound, and no co-shape.

The Subset does not allow any kind of array element sequence association for co-arrays. It therefore prohibits an element of a co-array being passed to a subroutine and treated there as a co-array of non-zero rank. Only entire co-arrays can be passed to explicit-shape co-array dummy arguments and the size of the actual and dummy argument must be identical.

5.2. Subset Co-Array Fortran into OpenMP Fortran

Subset Co-Array Fortran has been designed to be implementable by mapping co-arrays onto arrays of higher rank. In particular, they are implementable as shared OpenMP Fortran arrays. The translation of the Co-Array Fortran π program into OpenMP Fortran presented in Section 3 illustrates what a compiler is required to do. Any saved or module local variables must be placed in `THREADPRIVATE` named common (or just declared `THREADPRIVATE` in version 2.0). Any named common that does not contain co-arrays must be made `THREADPRIVATE`. All co-arrays must be shared objects. Square brackets are merged to create arrays of higher rank. The convention that references to a co-array without square brackets is a reference to the local part of the co-array requires first expanding the reference to include both round brackets and square brackets, and then merging square brackets to create an array subobject. References to co-arrays in procedure calls do not typically include square brackets, but the intent is always unambiguous because the interface must be explicit when the dummy argument is a co-array. If the dummy argument is not a co-array, the reference must be expanded to explicitly pass the local part of the co-array to the procedure. If the dummy argument is an assumed shape co-array (with co-rank one), the dummy is translated to an assumed shape array with one higher rank and special handling may also be required on the calling side. Co-Array intrinsic procedures can be implemented as an OpenMP module. All *caf-procedure* calls [6], i.e., calls to procedures that could contain synchronization, must be bracketed by `FLUSH` directives that explicitly name all actual co-arrays in the local scope. A generic `FLUSH` without arguments would also be sufficient, but is less efficient because OpenMP would then flush objects that the original Co-Array source has identified as not being thread visible. All of Co-Array I/O maps directly onto thread-safe OpenMP I/O, so the translator may have to explicitly make I/O thread safe,

using critical directives, but the mapping is otherwise straight forward. The translation process has been presented as if performed by a Subset Co-Array Fortran source to OpenMP Fortran source compiler. Many of the steps are trivial if actually performed by retargeting an existing native OpenMP Fortran compiler to support Subset Co-Array Fortran. So on machines with a cache-coherent shared memory and an OpenMP compiler it would take very little effort on the vendors part to support Subset Co-Array Fortran. A single compiler is typically already used for standard Fortran and OpenMP Fortran, with the target language specified at compile and link time. With minor upgrades the same compiler can also support Subset Co-Array Fortran. There would be a single compiler but three distinct languages, so linking standard Fortran and Subset Co-Array Fortran objects together would not be supported (just as linking standard Fortran and OpenMP Fortran objects is not supported now).

As a "proof of concept" a nawk script has been developed to translate Subset Co-Array Fortran directly into OpenMP Fortran. Since this is a pattern matching script, rather than a compiler, it treats some keywords as reserved and requires some statements be expressed in one of the several alternatives that Fortran provides. In order to implement `TEAM` read, I/O unit numbers are restricted to be less than 100,000. The only other significant variances from the Subset Co-Array Fortran language are those made necessary by a lack of a symbol table identifying modules and co-arrays by name. The most serious of these is that the local part of a co-array cannot be referenced without square brackets. To simplify local parts, the script will automatically translate a copy of the co-array's declaration square brackets, with `"*"` replaced by `"@"` into square brackets identifying the local part. For example:

```
COMMON/XCTILB4/ B(N,4)[0:MP-1,0:*]
SAVE  /XCTILB4/
CALL SYNC_ALL( WAIT=(/IMG_S,IMG_N/) )
B(:,3)[0:MP-1,0:@] = B(:,1)[I_IMG_S,J_IMG_S]
B(:,4)[0:MP-1,0:@] = B(:,2)[I_IMG_S,J_IMG_N]
CALL SYNC_ALL( WAIT=(/IMG_S,IMG_N/) )
```

Only the local part of `B(:,3)` and `B(:,4)` is used, but square brackets are still required and have been provided by replicating the square bracket declaration of `B` with `"*"` replaced by `"@"`. The advantage of this extension to the language is that these square brackets can be removed by a batch stream editor to produce a legal Subset program. Absent a symbol table tracking explicit interfaces, passing co-arrays to subroutines also requires extensions to the Subset language. A whole co-array can be passed to a co-array dummy just as in the Subset, but all other cases rely on an extension to the Subset to allow co-array sections to be passed as arguments. A co-array section passed to a co-array dummy must include square brackets that cover the entire co-extent. A local part passed to a dummy of co-rank zero must use square brackets to form the corresponding co-array section.

The nawk script obviously provides a way of running Co-Array Fortran programs (after some manual tweaking) via an OpenMP compiler. But it can also be simply viewed as a pre-processor that provides an improved SPMD interface for OpenMP. It has several major advantages over native OpenMP Fortran for SPMD programs.

For example, I/O is consistent with process-based SPMD APIs and the mapping of variables onto shared and private memory is greatly enhanced (because the script automatically places variables in `COMMON` as necessary). Co-Array Fortran intrinsic procedures provide a much richer set of synchronization options than OpenMP, and the special handling of caf-procedures ensures that synchronization of threads implies synchronization of co-arrays. A disadvantage of the nawk script is that it provides no error checking. Legal Co-Array Fortran programs are translated to legal OpenMP Fortran programs, but illegal programs will also be translated and it is up to the OpenMP compiler to detect the error. Error messages are likely to be obscure, but relatively few lines are modified in the translation so inspection of the OpenMP source should provide an indications of the error. A true Subset Co-Array compiler, either provided as an addition to an OpenMP compiler or as a stand-alone source to source compiler, would not have any of the restrictions of the nawk script and would be able to provide clear and relevant error diagnostics for non-conforming syntax.

One advantage that OpenMP has over Co-Array Fortran is that if an OpenMP program is designed to work when there is exactly one thread, it is then also a legal Fortran 90/95 program. The compiler directives have no effect on one thread, and are ignored by the Fortran 90/95 compiler. A library of a few standard procedures is required, but is trivial to implement for a single thread. This is not the case for Co-Array Fortran. Obviously, a Subset Co-Array Fortran source to OpenMP Fortran source compiler would also be a Subset Co-Array Fortran source to Fortran 90/95 source compiler in the special case of one image. A much simpler source to source compiler is sufficient in this special case, and a public domain implementation would provide a useful service to the Co-Array Fortran programming community. This is not quite just a matter of deleting all references to square brackets, because the effective rank of a co-array subobject is the sum of its local rank and co-rank. If the square brackets are deleted the effective rank may change, giving rise to illegal ranks for intrinsic procedure arguments and non-conforming ranks in some array assignment statements involving co-arrays.

6. Conclusions

Both Co-Array Fortran and OpenMP Fortran are viable languages for SPMD programs. Version 2.0 of the OpenMP Fortran specification provides more SPMD support than previous versions, but still has limitations, particularly in ease of use and code maintainability, that could be removed with relatively small additions to the existing suite of compiler directives. OpenMP Fortran is a thread-based language and provides comprehensive support for the fine grain synchronization typical of threaded programs. It is therefore the better candidate when mixing SPMD and threaded programming styles. Co-Array Fortran has been designed exclusively for SPMD programming, and has distinct advantages over OpenMP Fortran for such programs. It is also based on Fortran 90/95, rather than Fortran 77, and Fortran 90/95 features are becoming increasingly common in large scalable scientific packages. Co-Array Fortran scales to large distributed shared memory machines,

because, unlike OpenMP Fortran, all memory is associated with a particular image. In fact, Co-Array Fortran can be used on shared nothing systems with physically distinct memories connected by a network. However, performance is expected to only be about as good as MPI on such systems. Only one machine, the Cray T3E, has a Co-Array Fortran compiler today and it is for a subset of the language. OpenMP Fortran is becoming increasingly widely available on SMP and DSM systems. A formal Subset Co-Array Fortran language is therefore introduced that simplifies mapping co-arrays onto standard arrays of higher rank. An OpenMP Fortran compiler can be extended to support Subset Co-Array Fortran with relatively little effort, as illustrated by a "proof of concept" nawk script that translates a Subset Co-Array Fortran like language into OpenMP Fortran. Alternatively, a source to source compiler translating Subset Co-Array Fortran into either OpenMP Fortran or Fortran 90/95 plus a threads library would provide portability to all SMP and DSM systems.

Acknowledgments

This is a contribution to the 6.2 Global Ocean Prediction System Modeling Task. Sponsored by the Office of Naval Research under Program Element 62435N. Also to the Common HPC Software Support Initiative project Scalable Ocean Models with Domain Decomposition and Parallel Model Components. Sponsored by the DoD High Performance Computing Modernization Office. Special thanks to John Reid of the Rutherford Appleton Laboratory for his careful reading of early drafts of this paper and his many helpful suggestions.

References

1. Cray Research Inc. *Application Programmer's Library Reference Manual*. Cray Research SR-2165, 1996.
2. D. Dougherty. *Sed & Awk*. O'Reilly and Assoc., Sebastopol, CA, 1990.
3. S. Kleiman, D. Shah, and B. Smaalders. *Programming with Threads*. SunSoft Press, Prentice Hall, Upper Saddle River, NJ, 1996.
4. C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.
5. The Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, http://www.mpi-forum.org/docs/docs.html, 1997.
6. R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *Fortran Forum*, 17(2):1–31, 1998.
7. The OpenMP Organization. OpenMP: A Proposed Industry Standard API for Shared Memory Programming, http://www.openmp.org, 1997.
8. The OpenMP Organization. OpenMP Fortran Application Programming Interface version 1.1, http://www.openmp.org, 1999.
9. The OpenMP Organization. OpenMP Fortran Application Programming Interface version 2.0, http://www.openmp.org, 2000.
10. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.
11. A. J. Wallcraft. SPMD OpenMP vs MPI for ocean models. *Concurrency: Practice and Experience*, 12:1155–1164, 2000.