The Complete Compendium on Cooperative Computing using Coarrays.

October 29, 2008

Preface

Over the last several decades, the speed of computing has increased exponentially, a phenononom called Moore's Law. Of course, this can't go on forever. At the time of this writing, the size of wires on microchips is around 100 nanometers. Since this is only several hundred atoms, quantum effects and leakage currents are becoming more dominant, making fabrication and design vastly more difficult. Several schemes are being proposed to get around this small difficulty, but one of the simplest is to use more than one processor in a program.

There are currently several systems for addressing problems like this. They typically involve programming libraries that are linked in with the user program and involving calling subroutines and functions exported by the library.

Coarrays are a solution along these lines, with the difference being much tighter intregration with the Fortran language itself. There are still functions to be called, but statements are used for increased legibility. The biggest advantage is that remote memory is referred to directly instead of having to call a subroutine for loading and storing data. These subroutines still exist "under the hood", but coarrays are designed with legibility in mind. A program that runs over several cores is inherently more complicated than a single-threaded program and anything that makes the processes clearer is to be welcomed.

This guide is meant to be a practical introduction to using coarrays in general, and how to use them with G95 in particular.

Introduction

“With great power comes great responsibility”— ’Uncle Ben’ (Stan Lee)

Parallel programs are just flat out hard to write. If more than one program accesses the same data at once, all kinds of subtle interactions become possible. And yet, there has to be some form of communication between programs if useful work is to be done. Coarray Fortran uses the model of single-program multiple-data (SPMD). A single program runs on multiple machines with different local data that is occasionally shared across machines. Each execution context is called an “image”. Images are numbered, starting at one and going up to however many images are specified at runtime.

In coarray Fortran, variables can have the `CODIMENSION` attribute, which is similar to an array dimension. This means that such a variable not only exists on all images, but also that the content of this variable on any given image can be accessed by all other images. Such a variable is then called a “coarray”.

The declaration of a coarray looks like:

```
integer :: a[*]
```

The `*`-syntax is reminiscient of an assumed size array from `FORTRAN 77`, which means that the upper bound is undefined until runtime. By itself, `a` is just an integer variable that exists on each image. The expression `a[2]` refers to the value of `a` on image two, `a[i]` refers to the value of `a` on image `i`. The statement

```
a[2] = a[3] + 1
```

means to load `a` from image three, add one to it, then store the result to the `a` on image two. This statement has the same effect no matter which image executes it.

The term “`a[2]`” is a legal Fortran expression. You can print it, add it, multiply it, square it, compare it, use it as an array index, take the inverse hyperbolic cotangent of it or anything else. Access to it can be from memory shared by local images, or from a machine across the world over a network. You don’t have to worry about where it is, you just use it. The square brackets provide a quick visual indicator of the cross-image communication.

There are a couple of standard intrinsic functions that help support coarrays. The two most important of these are `THIS_IMAGE()` and `NUM_IMAGES()`, which return the image number of the image that executes it and the total number of images respectively. Using `a` is equivalent to `a[THIS_IMAGE()]`. The `a` form is more efficient because the compiler knows that `a` is always a local reference instead of having to compute the image number and possibly load a value from somewhere else.

The coarray in the previous example has a corank of one. Coarrays can have a larger corank by declaring them that way. For example:

```
real :: b[2, 4:*], c[-10:10, 4:10, *]
integer, CODIMENSION [2, *] :: x, y, z
```

In the same manner that the upper bound of assumed size arrays must be designated by an asterisk, the upper bound of last codimension must be `*`, leaving it indefinite, but a lower bound can be specified. Like a regular assumed-size array, the lower bound defaults to one if not given.

Coarray references work in a similar manner as a regular Fortran array. Instead of referencing an array element, an image number is ultimately calculated. From the previous declaration, `b[1,4]` is image one, `b[2,4]` is image two, `b[1,5]` is image three and so on. The elements of a coarray reference must always refer to a valid image number.

Depending on how images are allocated among actual processors, the codimension can be used to reflect arrangement of the processors. For example, a cluster of dual-core machines might have a coarray that is declared as `a[2,*]`, where coarrays `a[1,m]` and `a[2,m]` are stored on the same physical machine, and may be accessed much more efficiently than between coarrays `a[1,m]` and `a[1,m+1]` which might require a network transaction. The actual assignment of images is processor dependent. The G95 Coarray Console allows some flexibility in how image numbers are assigned.

Coarrays cannot be pointers (including `C_PTR` and `C_FUNPTR`), but pointers can be components of derived types that are coarrays. An image cannot allocate or deallocate memory on another image but it can reference pointers on a remote image. Once the local image associates a pointer (which cannot be done remotely), any image can load or store the target of the pointer. This facility is particularly powerful with components that are allocatable arrays.

Another powerful application is with procedure pointers. When an image dereferences a procedure pointer on another image, the procedure on the local image is called. Like a data pointer, remote images must wait for a local image to associate the pointer before it can be used. This allows one image to initiate control transfers on another image.

Coarrays can also be arrays. Here are three examples of correct declarations of array coarrays and one incorrect one:

```
real :: a(10)[*]
real, CODIMENSION[10,*] :: b(3,3)
real, CODIMENSION(10)[*] :: c    !  Syntax error!
integer, DIMENSION(10), CODIMENSION[*] :: d, e(15)[*]
```

If both rank and corank are used without an attribute specification, i.e. the `a` and `e` coarrays in the previous example, the array specification always precedes the coarray specification. Similarly in a coarray reference, the array reference always precedes the codimension reference. When referencing a full coarray, the `(:)` notation must be used. For example:

```
integer :: a(10)[*]
...
a(:)[5] = 1      !  Set a(1), a(2), ...  a(10) on image 5 to one
a(5)[6] = 2      !  Set a(5) on image 6 to two
```

Coarrays can have default initializations the same as regular variables and with the same syntax:

```
integer :: a[*] = 123
integer :: b(5)[2,*] = (/ 1, 2, 3, 4, 5 /)
```

Each image starts out with the same initial value.

The `CO_LBOUND()` and `CO_UBOUND()` intrinsics act like `LBOUND()` and `UBOUND()` intrinsics, except that they operate on cobounds instead of regular bounds. The exact arguments to these intrinsics can be found in the coarray function reference, but the simplest form with just a coarray argument returns a rank-one integer array with the appropriate cobounds.

Note that the upper bounds returned by `CO_UBOUND()` do not necessarily correspond to a valid image. For example, if we ran the code fragment

```
real :: b[10,*]
...
print *, CO_UBOUND(b)
```

with 15 images, the result would be the array `(/ 10, 2 /)`, which are not valid coindices. The last valid image is actually `b[5,2]`. It is the user's responsibility to make sure that only existing images are referenced.

The `THIS_IMAGE()` intrinsic can also be called with a coarray argument, in which case a one dimensional integer array is returned that gives the coarray indices for the coarray argument. If the optional `DIM` argument is also given, only the single integer index of that codimension is returned.

The `IMAGE_INDEX()` intrinsic provides the inverse calculation– Given a coarray and a one-dimensional array that represents co-indices, the image number is returned, or zero if the coindices are invalid. For our `b[10,*]` coarray with 15 images, `IMAGE_INDEX(b, [5,2])` returns 15 and `IMAGE_INDEX(b, [6,2])` returns zero. See the function reference for more details.

Synchronization

When a coarray program starts, all images start running at the same time in the PROGRAM program unit. Different images usually branch to different parts of the program to carry on different parts of the calculation. Coarray Fortran offers several forms of synchronization to coordinate execution of different images. The simplest is the SYNC ALL statement.

Executing a SYNC ALL statement amounts to waiting until all other images have executed a SYNC ALL statement. Once this happens, all images continue execution. A simple example is:

```
integer :: a[*]

if (THIS_IMAGE() == 1) then
   read(*,*) m
   do i=1, NUM_IMAGES()
      a[i] = m
   enddo
endif

SYNC ALL     ! Wait until 'a' is set
call foo(a)
...
```

Here, the SYNC ALL is necessary to force all images to wait until a is initialized by image one before proceeding. SYNC ALL can be inefficient, though. It forces all images to wait for all other images. SYNC ALL acts as a barrier to further program execution by any image. No image will call foo() until the a coarray is fully initialized.

In a coarray program, any image may write to standard output, but only image one may read from standard input. The G95 Coarray Console allows several options for segregating data from different images and also allows images other than one to read from standard input.

The SYNC IMAGES statement is more fine-grained. It causes pairs of images to synchronize in the same way. A short two-image example is:

```
integer :: a[*]

if (THIS_IMAGE() == 1) then
   SYNC IMAGES(2)   !  Wait for image 2
   print *, a       !  a[1]

else if (THIS_IMAGE() == 2) then
   a[1] = 123
   SYNC IMAGES(1)   !  Tell image 1 to proceed
endif
```

In this example, image one waits for image two to execute a SYNC IMAGES(1) before it can proceed. The opposite is also true– image two will also wait for image one to execute SYNC IMAGES(2) before continuing. The first image to reach the SYNC IMAGES waits for the second, then both continue.

There are two additional forms of the SYNC IMAGES statement. SYNC IMAGES(*) refers to all other images except the executing image. The image specification may also be a rank-one integer array that is a list of images, e.g., SYNC IMAGES([1,3,5]). The *-form is useful for initializations:

```
integer :: a[*]

if (THIS_IMAGE() == 1) then
   read(*,*) m
   do i=1, NUM_IMAGES()
      a[i] = m
   enddo

   SYNC IMAGES(*)
else
   SYNC IMAGES(1)
endif

call foo(a)
```

This differs from the previous SYNC ALL example in that all images except image one wait for image one instead of each other. Once image one is done, it effectively releases all of the other images. The foo subroutine is not called in any image until the a coarray has been initialized.

The NOTIFY and QUERY statements are very similar to the SYNC IMAGES statement, but they are asymmetric. The NOTIFY statement sends a notification to a set of images without waiting for a reply. The QUERY statement waits for notification from a set of images. More than one notification can be outstanding at any time. If QUERY is used with a ready tag, it will always return immediately and set the ready value to indicate whether receipt of all notifications are complete. If true, all notifications are accepted. If not, no notifications are accepted. Without a ready tag the program waits for all notifications. This is a non-blocking synchronization which allows a program to do something else instead of waiting. Our previous initialization program can be written

```
integer :: a[*]

if (THIS_IMAGE() == 1) then
   read(*,*) m
   do i=1, NUM_IMAGES()
      a[i] = m
   enddo

   NOTIFY(*)
else
   QUERY(1)
endif

call foo(a)
```

An alternative program is

```
integer :: a[*]
logical :: flag

if (THIS_IMAGE() == 1) then
   read(*,*) m
   do i=1, NUM_IMAGES()
      a[i] = m
   enddo

   NOTIFY(*)
else
   do
      QUERY(1, ready=flag)
      if (flag) exit
      call do_something_worthwhile()
   enddo
endif

call foo(a)
```

Another synchronization primitive is the `SYNC MEMORY` statement. `SYNC MEMORY` forces any pending coarray writes to finish before proceeding as well as flushing any cached reads. If a coarray is written to, additional statements can be executed before other images can read the new value. `SYNC MEMORY` waits until preceding writes are visible to other images. Another way of putting it is that coarray writes are always asynchronous, and the `SYNC MEMORY` serves to synchronize any preceding transfers.

```
integer :: a[*]
...
if (THIS_IMAGE() == 1) then
   a[2] = 123
   print *, 'Image two does not necessarily have a == 123, ...  yet'
   SYNC MEMORY
   print *, 'Image two now has a == 123'
endif
```

All other image control statements (`SYNC`'s, `NOTIFY`, `QUERY`) imply a `SYNC MEMORY`.

Coarray Fortran also allows statements to be grouped in a `CRITICAL` block. These are blocks of statements that begin with a `CRITICAL` and end with an `END CRITICAL`. Only one image at a time is permitted inside these blocks. Other images wait at the beginning of the block until the currently executing image leaves it. The main use for this primitive is for atomic updates of data structures that involve more than single CPU words. Other image control statements are not permitted inside `CRITICAL` blocks. The G95 coarray implementation puts images entering a `CRITICAL` block into a first-in first-out queue, so that an image will not wait indefinitely.

It's easy to ascribe more synchronization to `CRITICAL` blocks than they actually provide. Suppose the following program is run with three images:

```
program crit
    integer, CODIMENSION[*] :: i_sum = 0

    CRITICAL
       i_sum[1] = i_sum[1] + THIS_IMAGE()
    END CRITICAL

    if (THIS_IMAGE() == 1) write(6, *) 'Sum = ',i_sum
end program crit
```

This program can print one, three, four or six. One way that one can be printed is by image one entering the CRITICAL block first, doing the addition, exiting the block and printing i_sum before the other images can update i_sum. The problem here is synchronization of the WRITE statement, not the actual addition. Adding a SYNC ALL barrier prior to the IF statement forces image one to wait until the additions are complete. Note that the CRITICAL block is still necessary– otherwise the same value of i_sum[1] can be loaded by multiple images before being updated.

A neat trick worth knowing is described by John Reid and uses SYNC IMAGES to emulate a CRITICAL block. It is stronger than a CRITICAL block in that images execute the block in sequence, instead of a processor-dependent order.

```
me = THIS_IMAGE()

if (me > 1) SYNC IMAGES(me - 1)
print *, 'Image', me, 'here'
if (me < NUM_IMAGES()) SYNC IMAGES(me + 1)
```

If the me image is greater than one, it waits until the (me − 1)-th image is done. Once done with the critical section, all images except the last synchronize with the next image, allowing it into the block.

The last thing a program does is to stop, which it can do "normally" or "abnormally". Normal termination happens if a program executes the END PROGRAM or STOP statements. In this case, the coarrays associated with an image remain available to other images.

Abnormal termination is anything but normal termination– a program crash, an I/O error without an error tag, a memory allocation failure without the stat tag, or any similar program termination. An explicit way to initiate abormal termination is the ALL STOP statement. In abnormal termination, all images are stopped as quickly as possible. This is useful for shutting down a calculation without any additional logic in the program itself. The ALL STOP has the same syntax as the STOP statement, and can include an optional constant stop-code that can be a numeric or string constant.

Scoping

Most coarrays are static coarrays, meaning that they always exist while the program is running. Most coarrays live in modules, where they implicitly or explicitly have the `SAVE` attribute. Coarrays can also be declared within procedures with the explicit `SAVE` attribute, or in a `PROGRAM` unit, in which case the `SAVE` attribute is implicit.

Automatic coarrays, i.e. coarrays local to a subroutine without the `SAVE` attribute are not permitted– if they were, they would have have to follow the same rules as allocatable coarrays and be the same size on all images as well as requiring synchronization between images when they came into scope. Coarray function results are not permitted for the same reason. The following example has three legal declarations and one illegal declaration:

```
module mod
   integer :: a[*]          !  Legal
   ...
end module mod

subroutine sub()
   integer, save :: b[*]    !  Legal
   integer :: c[*]          !  Illegal!
   ...
end subroutine sub

program p
   integer :: d[*]          !  Legal
   ...
end program
```

Coarrays and parts of coarrays can be passed as arguments to procedures. Coarray parts follow the same rules as regular fortran arguments. When a full coarray is passed, the codimension part can be freely redefined in the dummy argument in a manner analogous to an assumed-size array. The interface to a procedure with a full coarray argument must be explicit. Otherwise, the local part of the coarray is passed, not the coarray itself. An example of passing coarrays through a procedure call is

```
module m
   integer :: v[*]
   external :: sub1

contains
   subroutine sub2(b)
      integer :: b[10,*]

      do i=1, 10
         print *, b[i,1]
      enddo
   end subroutine sub2
```

```
      subroutine sub
         call sub1(v)    !  Implicit interface, passes local part of v
         call sub2(v)    !  Explicit interface, passes whole coarray
      end subroutine sub
   end module m
```

Coarrays can also be allocated and deallocated dynamically during the run of a program. Allocatable coarrays are declared in much the same way as regular allocatable array, all parts of an allocatable coarray must be deferred until runtime.

```
   real, allocatable :: a[:], b(:)[:], c(:,:)[:,:,:]
   real, allocatable :: d(20)[:]            !  Illegal
```

In an `ALLOCATE` statement, the coarray specification appears after the regular array specification. Like a static size coarray, the upper bound of the last codimension must always be `*`. The actual upper bound is always determined at runtime. A complicated example of coarray allocation is:

```
   allocate( a[*], b(i)[*], c(r1,r2)[c1,c2,*] )
```

When a coarray is allocated, it must be allocated by all the images in the program and the bounds and cobounds must be the same on all images. All dimensions and codimensions must be allocated at once. Furthermore, all the allocations happen at the same time– memory has to be allocated on all images before any other image can proceed, since any image may reference data on any other image. This requires synchronization like a `SYNC ALL`, where each image has to wait for all other images.

The deallocation of a dynamic coarray with the `DEALLOCATE` statement happens in the same way– the memory must be deallocated on all images at the same time. Even in cases where the array is deallocated automatically, synchronization still takes place. The syntax for the deallocation of a coarray is the same as deallocation of a regular array. The `DEALLOCATE` statement just takes a list of variables to deallocate, without subscripts.

Monte Carlo example

Let's consider a simple, but complete coarray code, a Monte Carlo code for calculating π, the ratio of the circumference to diameter of a circle. The idea is to pick random points on a unit square, and then use the Pythagorean theorem to test whether or not the point is within a unit circle. The area of the unit square is one, and the area of the unit circle within the unit square is $\pi r^2/4 = \pi/4$. If we pick random points within the unit square, the fraction $\pi/4$ should be in the unit circle. So if we compute the fraction of points within the circle and multiply by four, we should get π.

The overall idea of the program is to have image one be the "master image" that collects results from the other images. Each "worker image" has its own local total of how many points have been picked and how many points were inside of the unit circle. Once per second, the master accumulates the results and prints them out. The first part of the code is:

```
module pi_calc
  use, intrinsic :: iso_c_binding

  type results
    integer(kind=8) :: in, total
  end type results

  type(results) :: acc[*]

  interface
     subroutine usleep(u) bind(c)
       use, intrinsic ::  iso_c_binding
       integer(kind=c_long), value :: u
     end subroutine usleep
  end interface
```

The `INTERFACE` block allows us to call the standard unix `usleep()` subroutine, which takes a long integer argument in microseconds to sleep. The last thing we want is a master image that hogs the CPU. It would be possible to have the master image do work of its own and periodically display results, but this keeps our program simple. This is an example of where more images should be run than the number of processors you are willing to saturate. The number of images is selected at runtime with the G95 Coarray Console, see its documentation for details.

The next section is the master subroutine:

```
contains
  subroutine master
    type(results) :: sum

    do
       sum%total = 0
       sum%in    = 0

       do n=2, NUM_IMAGES()
          sum%total = sum%total + acc[n]%total
          sum%in = sum%in + acc[n]%in
        enddo

        print *, sum%in, sum%total, 4*dble(sum%in) / dble(sum%total)
        call usleep(1000000_c_long)
      enddo
    end subroutine master
```

The master consists of an infinite loop that initialize the `sum` accumulators to zero, then adds them up from image two to however many images we have, prints out the current estimate for π, then goes to sleep for a second before starting over again.

```
    subroutine worker
      real :: x, y

      acc%in     = 0
      acc%total  = 0
      call random_seed()

      do
        call random_number(x)
        call random_number(y)

        if (x*x + y*y < 1.0) acc%in = acc%in + 1
   !  Location A
        acc%total = acc%total + 1
      enddo
    end subroutine worker

  end module pi_calc
```

The worker subroutine also has an infinite loop, but it contains the heart of the Monte Carlo algorithm. An important requirement is that the sequence of random numbers be different on each image. Otherwise we are recycling the same series of numbers, messing up our results. This isn't a problem in G95, where the random number generator is seeded by the current time in microseconds. A best practice is to seed the random generator using the `RANDOM_SEED()` intrinsic subroutine with a value involving the image number.

The final part of the program is the main program:

```
  program m
    use pi_calc

    if (NUM_IMAGES() == 1) stop 'Need more images!'

    if (THIS_IMAGE() == 1) then
      call master()
    else
      call worker()
    endif
  end
```

The main program includes a safety check at the start, since more than one image is required to run the program. If only one image is running, no results will be created for the master image to collect. After this, the master or worker images are called and the program runs.

If you run it, the program works. However, there are some serious theoretical problems with it. The main one is the most common bane of all multiprocess programs, the race condition. A race condition exists where the behaviour of a program changes depending on the execution of one image relative to another. This usually involves an image getting to some place first, in which case it is said to have "won" the race. You, the programmer, lose. Race conditions exist on single processor machines as easily as on machines with multiple processors.

For example, suppose a worker image happens to pick a point within the unit circle. It then increments `acc%in`. Suppose that just after the increment, but still before the increment of `acc%total` (location `A`) the master image comes along and reads both of these and accumulates them into its totals. Well, now the total is wrong. Because it's only wrong by one out of billions or trillions, the error is almost unnoticable, but it is still there. The same principle can have much nastier effects in programs more sensitive to these sorts of bugs.

How would we fix this program? The first option is to put the updates and reads inside a `CRITICAL` / `END CRITICAL` block. This would make the master loop look like:

```
do
   call usleep(1000000_c_long)
   sum%total = 0
   sum%in    = 0

   CRITICAL
      do n=2, NUM_IMAGES()
         sum%total = sum%total + acc[n]%total
         sum%in = sum%in + acc[n]%in
      enddo
   END CRITICAL

   print *, sum%in, sum%total, 4*dble(sum%in) / dble(sum%total)
enddo
```

and the worker loop look like:

```
do
  call random_number(x)
  call random_number(y)

  CRITICAL
    if (x*x + y*y < 1.0) acc%in = acc%in + 1
    acc%total = acc%total + 1
  END CRITICAL
enddo
```

The problem is that the CRITICAL block ends up being part of the inner loop of the program, and the overhead of entering and leaving the block is very high. Our program ends up spending more time entering and leaving the block than performing actual computation. This is exactly the sort of thing you want to avoid at all costs.

If the situation were the other way around– the computation taking much longer than processing a CRITICAL, then this is a perfectly fine solution. It's small codewise and easy to understand. It doesn't scale very well– if we had hundreds of images, then the contention for the global lock will start to slow things down. But it's a quick and easy solution that usually works fine.

One can imagine the worker images furiously updating the acc coarray to be sporadically read by the master image. One possible solution is to have acc updated less frequently. This doesn't really solve the problem, though, it just makes garbled reads less likely.

Suppose each worker image kept its intermediate results private and only posted the results when requested? Let's add a LOGICAL coarray named flag to the pi_calc module. When this flag is set, it is a request to a worker image for its current results. The master subroutine would set this coarray periodically and wait for the worker images to notice. Once all the workers have noticed that flag is set, they provided current totals, acknowledge the master, which can then safely add the current sums. The new version of the master subroutine is:

```
subroutine master
  type(results) :: sum

  do
     call usleep(1000000_c_long)

     do i=2, NUM_IMAGES()
        flag[i] = .true.
     enddo

     SYNC IMAGES(*)

     sum%total = 0
     sum%in = 0

     do n=2, NUM_IMAGES()
        sum%total  = sum%total  + acc[n]%total
        sum%in = sum%in + acc[n]%in
     enddo

     print *, sum%in, sum%total, 4*dble(sum%in) / dble(sum%total)
  enddo
end subroutine master
```

After delaying, the flag coarray is set on all worker images. The SYNC IMAGES waits for all of the worker images to notice that the flag is set and store their results. We wait until all other images have done this, then pull results from the worker images and print the total as usual. The worker subroutine becomes:

```
subroutine worker
  real :: x, y
  type(results) :: local

  local%in = 0
  local%total  = 0

  do
     call random_number(x)
     call random_number(y)

     if (x*x + y*y < 1.0) local%in = local%in + 1
     local%total = local%total + 1

     if (flag) then
        acc = local
        flag = .FALSE.  !  Before SYNC IMAGES(1)!
        SYNC IMAGES(1)
     endif
  enddo
end subroutine worker
```

The two changes here are the creation of a `local` variable for intermediate results, and a check on the `flag` coarray to see if we should report our results. If so, we copy the results, reset the `flag` and synchronize with image one. The main program is unchanged.

As in any parallel program, there are several subtle points that deserve more detailed comments, both here and in the source code, lest someone change it heedlessly. The main one is the fact that `flag` is reset before the `SYNC IMAGES` statement.

Suppose we were to reset it after the `SYNC IMAGES`. If a worker image is delayed for more than one second, after `SYNC IMAGES` but before the assignment to `flag`, then the master image would have time to set `flag` back to `.TRUE.`. The worker image would then reset `flag`, which would remain `.FALSE.` forever, and the master image would wait forever for synchronization that would never come.

This also highlights one way to try and find race conditions– at different points in one image, suppose it were delayed at that point for a very long time and see how this affects other images. For all points within the worker image, the master image eventually ends up at the `SYNC IMAGES` statement waiting for results. For all points within the master image, nothing interesting can happen in other images until `flag` is set, and shortly after that we're in the `SYNC IMAGES` waiting for other images.

There are several efficiency-related things about our final Monte Carlo calculation that should be pointed out. The first is that although the worker images wait in the `SYNC IMAGES` they don't spend long there compared to the total time spent computing something useful. Even if the `SYNC IMAGES` is being processed across a network, the delay over a local network is a millisecond or so out of a thousand milliseconds. Even over a wide area network, a latency of 30 milliseconds, only leads to a 3% reduction in CPU usage.

Another important point is that accessing a local version of a coarray is much more efficient than accessing data on another image. An alternative for our code could involve each worker image looking at `flag[1]`. The problem is that the check for the `flag` being set is in the inner loop. Across a network, this would cause a huge number of packets flying back and forth. Also notice that when the master image retrieves intermediate results from remote images, it is only doing so once per second, making the overhead negligible.

Noticing excessive image communication is straightforward. If you run "`top`" on your system and see that the G95 Coarray Console is taking up lots of CPU, this means that communication between images is taking up a lot of time. In the Monte Carlo example, the coarray console uses no measurable CPU at all, which is how you want it.

A very nice thing about this code is that it is not dependent at all on the speed of individual images. Each image proceeds as fast as it can. A fast machine can freely compute multiple results compared with a slow machine. It is only the intermediate results that are printed as slowly as the slowest machine can compute them. A major purpose of the `SYNC IMAGES` in our code is to cleanly turn off the `flag` variables, and this will happen quickly even for slow machines.

Another way of implementing the code would be to have each image loop for a fixed number of trials before writing out the final result, but then we are left with a situation where we end up waiting for the slowest machine. This would be good in the sense that a run would be repeatable (assuming that the random number generator was seeded in the same way). Our current code is not repeatable– running the program twice will produce different results, depending on exactly when the workers are sampled.

Yet another way is to have the worker images check the time using the `DATE_AND_TIME()` intrinsic. When the value of the current second changes from what it last was, we could write our current results, then `SYNC IMAGES` against image one, which would do a `SYNC IMAGES(*)` to wait for all the results to come in before calculating and printing a total. The problem is that the calculation would proceed as fast as the slowest image. Fast images would end up waiting on the master image, which in turn would wait for the slowest machines. A fragment could look like:

```
integer, save :: now, skip = 0
integer ::  time_values(8)
...

   skip = skip + 1
   if (skip > 1000) then
      skip = 0
      call date_and_time(values=time_values)
      if (time_values(7) /= now) then
         now = time_values(7)
         call periodic_event()
      endif
   endif
```

This code avoids calling `DATE_AND_TIME` at every trial.

The final point is that it is not really fair to call the "master" image "master" any longer. It does not control the workers at all and in fact works in concert with them to tally the totals. It is more of a "collector" in the final version.

Even in a relatively simple code like this, there are a multitude of tradeoffs to be considered. Even from this first code, a couple of basic principles seem clear:

- Keep data local where possible
- Keep interactions as simple as possible
- Localize interactions to small sections of code

Job Dispatcher

The next example illustrates a job dispatcher. The idea is to run multiple instances of a non-coarray program on multiple images. We do this in a manner that allows the easy extension of a non-coarray program. The basic compatibility module is

```
module myjob
  type input
     ...
  end type input

  type output
     ...
  end type output
```

The input and output structures hold inputs and outputs for the single-image program.

```
contains
  subroutine run(in, out)
    type(input),  intent(in)  :: in
    type(output), intent(out) :: out
    ...
  end subroutine run
```

The run() subroutine is passed an input structure, that it uses to calculate an output structure.

```
  subroutine load_inputs(inputs)
    type(input), allocatable, intent(out) :: inputs(:)
     ...
  end subroutine load_inputs
```

The load_inputs() subroutine allocates an array of input structures and loads each element with input values. It is run only in image one when the program starts. The lower bound of the allocated array is one.

```
  subroutine save_output(outputs)
    type(output) :: outputs(:)

    ...
  end subroutine save_output
end module myjob
```

The last subroutine of the module is save_output() which takes the array of output structures and saves them in an appropriate format. The main reason this happens at the end of the program is to make the output independent of the order in which the images finish. The user can modify this module to their own program.

The main PROGRAM unit actually runs the multi-image program. The program starts out with the declarations:

```
program runall
   use myjob

   type input_ptr
     type(input), allocatable :: p(:)
   end type input_ptr

   type output_ptr
      type(output), allocatable :: p(:)
   end type output_ptr

   integer :: j, t, next_job[*] = 1
   type(input_ptr)  :: in_ptr[*]
   type(output_ptr) :: out_ptr[*]
```

The coarrays in_ptr and out_ptr are scalar coarrays consisting of a single component that is a pointer to arrays of input and output structures, respectively. The arrays are only allocated on image one. The next_job integer variable is likewise only used on image one. next_job[1] is used as the number of the next input to run. Initialization comes next:

```
   if (this_image() == 1) then
      call load_inputs(in_ptr % p)
      allocate(out_ptr % p(ubound(in_ptr % p, 1)))
      SYNC IMAGES(*)
   else
      SYNC IMAGES(1)
   endif

   t = ubound(in_ptr[1] % p, 1)    !  t = Number of jobs
```

Image one calls load_inputs() and then allocates the output array to match the input array. Image one then synchronizes against all other images. All images then calculate t, the number of jobs to run and start the main loop:

```
   do
      CRITICAL
         j = next_job[1]
         if (j <= t) next_job[1] = j + 1
      END CRITICAL

      if (j > t) exit
      call run(in_ptr[1] % p(j), out_ptr[1] % p(j))
   enddo
```

The code in the CRITICAL block copies the next job number into j and updates next_job[1]. The CRITICAL block is necessary to prevent the race condition where two images load the same value from next_job[1]. If j is greater than the number of input jobs, we exit, otherwise we call the subroutine that executes the job. The call to run illustrates that pointer arrays can be dereferenced on remote images. The loop exits when no more jobs are available to run.

The final part of the program is cleanup

```
    if (this_image() == 1) then
       SYNC IMAGES(*)
       call save_output(out_ptr % p)
       deallocate(in_ptr % p, out_ptr % p)
    else
       SYNC IMAGES(1)
    endif
 end program runall
```

Synchronization is necessary to force image one to wait for all the other images to finish before writing the output. We do this from image one because this is where the output array is stored.

A variation on this code would be to add a procedure pointer to the `input` structure. Instead of calling the `run()` subroutine, the procedure pointer is called, branching to the right processing subroutine automatically.

Coarray Statement Reference

`ALL STOP [` *stop-code* `]`

The optional *stop-code* is a scalar integer or character initialization expression. The `ALL STOP` statement causes all images to terminate abnormally as soon as possible.

`SYNC ALL [ ( [` *sync-stat-list* `] ) ]`

The *sync-stat* is `STAT =` *stat-variable*
or `ERRMSG =` *errmsg-variable*
stat-variable is a scalar integer variable that is set to zero if the synchronization succeeds, `STAT_STOPPED_IMAGE` from the `ISO_FORTRAN_ENV` intrinsic module if another images has terminated or another positive value if something else goes wrong. If the `STAT` specifier is not present and an error occurs, the executing image is terminated. *errmsg-variable* is a scalar character variable that is set to an error message if an error occurs. Otherwise, *errmsg-variable* is not changed.

The `SYNC ALL` statement causes an executing image to wait until all other images are executing a `SYNC ALL` before continuing.

`SYNC IMAGES (` *image-set* `[ ,` *sync-stat-list* `] )`

If *image-set* is `*`, then all images are selected except the executing image. Otherwise, *image-set* is of type `INTEGER`. If *image-set* is scalar, then that image is selected. If *image-set* is not scalar, it must be a rank-one array that specifies a list of images to be selected. The image numbers must be valid and unique.

The `SYNC IMAGES` statement causes an image to stop until all images specifed in the *image-set* execute a `SYNC IMAGES` statement that specifies the original image.

`SYNC MEMORY [ ( [` *sync-stat-list* `] ) ]`

The `SYNC MEMORY` statement causes any pending coarray writes to finish and any cached reads to be flushed.

`NOTIFY (` *image-set* `[ ,` *sync-stat-list* `] )`

The `NOTIFY` statement sends notification to the images specified by *image-set*, and then continues execution without waiting.

`QUERY (` *image-set* `[ , READY =` *scalar-logical-var* `] [ ,` *sync-stat-list* `] )`

If the `READY` tag is not present, `QUERY` waits for notification via the `NOTIFY` statement from the images in *image-set*. If the `READY` tag is present and the `QUERY` would wait, then the *scalar-logical-var* is set to `.FALSE.` and execution continues without processing any notifications. Otherwise, the notifications are processed, *scalar-logical-var* is set to `.TRUE.` and execution continues. The `QUERY` and `NOTIFY` statements are an asymmetric version of `SYNC IMAGES`. They are unique in that they have a mode that does not block execution.

Coarray Function Reference

`LCOBOUND ( COARRAY [ , DIM ] [ , KIND ] )`

Without the optional scalar integer `DIM` parameter, returns a one dimensional integer array that gives the lower cobounds of the `COARRAY`. If the `DIM` parameter is present, a scalar integer giving the lower cobound for that dimension is returned. The optional `KIND` parameter is a scalar integer initialization expression that specifies the integer kind returned.

`UCOBOUND ( COARRAY [ , DIM ] [ , KIND ] )`

Without the optional scalar integer `DIM` parameter, returns a one dimensional integer array that gives the upper cobounds of the `COARRAY`. If the `DIM` parameter is present, a scalar integer giving the upper cobound for that dimension is returned. The optional `KIND` parameter is a scalar integer initialization expression that specifies the integer kind returned.

`CO_LBOUND ( COARRAY [ , DIM ] [ , KIND ] )`

Legacy synonym for the standard `LCOBOUND` intrinsic. G95 Extension.

`CO_UBOUND ( COARRAY [ , DIM ] [ , KIND ] )`

Legacy synonym for the standard `UCOBOUND` intrinsic. G95 Extension.

`IMAGE_INDEX ( COARRAY, SUB )`

The `SUB` parameter is a rank-one integer array with a size equal to the corank of the `COARRAY`. If the `SUB` array is a valid set of coindexes for `COARRAY`, the image number of the corresponding to the coindexes is returned. Otherwise, zero is returned.

`NUM_IMAGES ( )`

Return the number of images in the program as a default integer.

`THIS_IMAGE ( [ COARRAY [ , DIM ] ] )`

Without any arguments, `THIS_IMAGE` returns a default scalar integer that is the image number of the invoking image. If the `COARRAY` argument is present, an integer array of size equal to the corank of `COARRAY` is returned. The array are the cosubscripts of the coarray associated with the invoking image. If the scalar integer `DIM` parameter is also used, only that dimension of the subscript array is returned.