Parallel programming with Fortran 2008 coarrays

Anton Shterenlikht

Mech Eng Dept, The University of Bristol, Bristol BS8 1TR
mexas@bris.ac.uk

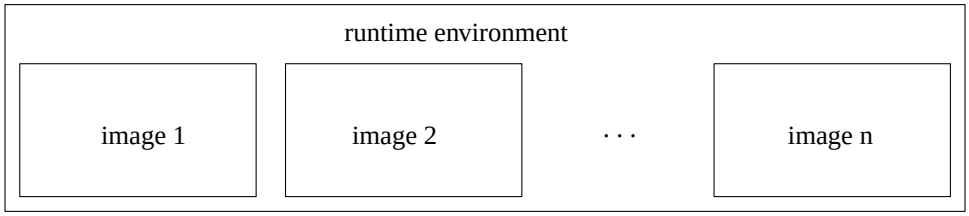
ABSTRACT

Coarrays are a Fortran 2008 standard feature intended for single program - multiple data (SPMD) type parallel programming. The runtime environment starts a number of identical executable images of the coarray program, on multiple processors, which could be actual physical processors or threads. Each image has a unique number and its private address space. Ordinary variables are private to an image. Coarray variables are available for read/write access from any other image. HPC literature coarrays are sometimes considered to be an example of partitioned global address space (PGAS) parallel programming model. Coarray communications are of "single sided" type, i.e. a remote call from image A to image B does not need to be accompanied by a corresponding call in image B. This feature makes coarray programming a lot simpler than MPI. The standard provides synchronisation intrinsics to help avoid race conditions or deadlocks. Any ordinary variable can be made into a coarray - scalars, arrays, intrinsic or derived data types, pointers, allocatables are all allowed. Coarrays can be declared in, and passed to, procedures. Coarrays are thus very flexible and can be used for a number of purposes. For example a collection of coarrays from all or some images can be thought of as a large single array. This is the opposite of the model partitioning logic, typical in MPI programs. A coarray program can exploit functional parallelism too, by delegating distinct tasks to separate images or teams of images. Coarray collectives are expected to become a part of the next version of the Fortran standard. A major unresolved problem of coarray programming is the lack of standard parallel I/O facility in Fortran. In this talk several program fragments and complete coarray programs are shown. Comparison is made with alternative parallel technologies - OpenMP, MPI and Fortran 2008 intrinsic "do concurrent". Inter image communication patterns and data transfer are illustrated.

28 January 2015

1. Coarray images

The runtime environment spawns a number of identical copies of the executable, called *images.* Hence coarray programs follow SPMD model.




```
$ cat one.f90
use iso_fortran_env, only: output_unit
implicit none
 integer :: img, nimgs
   img = this_image()
 nimgs = num_images()
 write (output_unit,"(2(a,i2))") "image: ", img, " of ", nimgs
end
$
$ ifort -o one.x -coarray -coarray-num-images=5 one.f90
$ ./one.x
image:  1 of  5
image:  3 of  5
image:  4 of  5
image:  2 of  5
image:  5 of  5
$
```

`iso_fortran_env` is the intrinsic module, introduced in Fortran 2003, and expanded in Fortran 2008. The module provides several named constants, such as `input_unit`, `output_unit` and `error_unit`, and a derived type.

All I/O units, except `input_unit`, are private to an image. However the runtime environment typically merges `output_unit` and `error_unit` streams from all images into a single stream. `input_unit` is preconnected only on image 1.

`this_image` and `num_images` are new intrinsics in Fortran 2008. `this_image()` with no arguments returns the index of the invoking image, starting from 1. `num_images()`, used always without arguments, returns the total number of images.

With Intel compiler one can set the number of images with the environment variable:

```
$ FOR_COARRAY_NUM_IMAGES=3
$ export FOR_COARRAY_NUM_IMAGES
$ ./one.x
image:  1 of  3
image:  2 of  3
image:  3 of  3
$
```

Note: as with MPI the order of output statements is unpredictable.

2. Intel Fortran compiler on BlueCrystal

The course materials are available via subversion, so you need to load this module:

```
tools/subversion-1.8.4
```

Make a directory for the course, and download the materials, e.g.

```
$ mkdir zzz
$ cd zzz
$ svn co https://svn.ggy.bris.ac.uk/subversion-open/pgas .
```

We will use ifort v.15 on phase 3. Load these modules:

```
languages/intel-compiler-15
languages/intel-compiler-15-impi
```

Also set the environment variables for ifort. For Bourne shell and related shells do:

```
source /cm/shared/languages/Intel-Compiler-XE-15/bin/compilervars.sh intel64
```

For C shell and related shells do:

```
source /cm/shared/languages/Intel-Compiler-XE-15/bin/compilervars.csh intel64
```

The example problems and the solutions are under

```
examples/coarray
examples/coarray/sol
```

3. Example program: printing image number

```
cd examples/coarray/1img
make
```

Tasks:

- Try running the compiled program a number of times. Do the messages appear in image order? Is the order of the messages the same for all runs? Explain these observations.
- Change the number of images using `FOR_COARRAY_NUM_IMAGES` environment variable. Do you need to recompile the program?
- Change the number of images using `-coarray-num-images` compiler switch. Recompile and re-run the program. Does the number of images match what you set? From `ifort(1)` man page:

  > "Note that when a setting is specified in environment variable `FOR_COARRAY_NUM_IMAGES`, it overrides the compiler option setting."

- Have a look at the `ifort(1)` man page. Search for `coarray`.

4. Coarray syntax

The standard[1, 2, 3] uses the square brackets `[]`, to denotes a coarray variable. A coarray variable can be also declared with `codimension` attribute. Any image has read/write access to all coarray variables on all images. It makes no sense to declare coarray parameters.

The last upper cobound is always an `*`, meaning that it is only determined at run time.

Examples of coarray variables:

```
integer :: i[*]                 ! scalar integer coarray with a single
                                ! codimension
integer, codimension(*) :: i    ! equivalent to the above

!                 lower  upper
!               cobound  cobound
!                     |  |
!                     |  |
!               upper |  |
!               bound |  |
!          lower |    |  |
!          bound |    |  |
!              | |    |  |
complex :: c(7,0:13) [-3:2,5,*] ! complex array coarray of corank 3
!            |  |       |  | |
!      subscripts       cosubscripts
!
```

Similar to ordinary Fortran arrays, *corank* is the number of cosubscripts. Each *cosubscript* runs from its *lower cobound* to its *upper cobound.* New intrinsics are introduced to return these values: `lcobound` and `ucobound`.

---

[1] ISO/IEC 1539-1:2010, *Information technology - Programming languages - Fortran - Part 1: Base language.*

[2] ISO/IEC 1539-1:2010/Cor 1:2012, *Information technology - Programming languages - Fortran - Part 1: Base language TECHNICAL CORRIGENDUM 1.*

[3] ISO/IEC 1539-1:2010/Cor 2:2013, *Information technology - Programming languages - Fortran - Part 1: Base language TECHNICAL CORRIGENDUM 2.*

5. Cosubscript sets

`this_image` can take a coarray variable as an argument. In this case it returns a *set of cosubscripts* corresponding to the invoking image. New intrinsic `image_index` is the inverse of `this_image`. Given a valid set of cosubscripts as an input, `image_index` returns the index of the invoking image. Note that there can be subscript sets which do not map to a valid image index. For such *invalid* cosubscript sets `image_index` returns `0`.

```
% cat cob.f90
program cob
implicit none
character( len=10 ) :: i[-2:2,2,1:*]
 if ( this_image() .eq. num_images() ) then
    write (*,*) "this_image()", this_image()
    write (*,*) "this_image( i )", this_image( i )
    write (*,*) "lcobound( i )", lcobound( i )
    write (*,*) "ucobound( i )", ucobound( i )
    write (*,*) "image_index(ucobound(i))", image_index( i, ucobound( i ) )
 end if
end program cob
% ifort -o cob.x -coarray cob.f90
% setenv FOR_COARRAY_NUM_IMAGES 20
% ./cob.x
 this_image()          20
 this_image( i )           2           2           2
 lcobound( i )          -2           1           1
 ucobound( i )           2           2           2
 image_index(ucobound(i))          20
% setenv FOR_COARRAY_NUM_IMAGES 24
% ./cob.x
 this_image()          24
 this_image( i )           1           1           3
 lcobound( i )          -2           1           1
 ucobound( i )           2           2           3
 image_index(ucobound(i))           0
%
```

6. Example program: cobounds and cosubscript sets

```
cd examples/coarray/2cob
make
```

Tasks:

- What number of images must be used for `ucobound(i)` to return a valid cosubscript set?
- Change the cobounds of coarray `i` to make sure `ucobound(i)` will return a valid cosubscript set when run on 8 images.

7. Remote operations, execution segments and image control statements

Remote operations are easily expressed in coarray notation. If a coarray variable does not have the square brackets, `[]`, then the reference is to the variable of the invoking image. The syntax thus clearly indicates which statements involve remote operations.

```
integer :: i[*], j
real :: r(3,8) [4,*]
!
i[5] = i              ! remote write
r(:,:) = r(:,:) [3,3] ! remote read
i = j                 ! both i and j taken from the invoking image
```

There are several rules governing remote calls. Only one image can be referenced in each statement. For array coarray the bracket notation, `()`, must be used if a coarray variable is an array. A valid set of cosubscripts must be used to refer to an image, not the image index.

A Coarray program consists of one or more *execution segments.* The segments are separated by *image control statements.* If there are no image control statements in a program, then this program has a single execution segment. `sync all` is a simple image control statement. To use it, each image must execute this statement. On reaching this statement each image waits for each other. Its effect is in ordering the execution segments on all images. All statements on all images before `sync all` must complete before any image starts executing statements after `sync all.` In other words all images *synchronise* with each other. Thus `sync all` is a global barrier, similar to MPI routine MPI_Barrier.

```
integer :: i[*]               ! Segment 1 start
if (this_image() .eq. 1 ) &   ! Image 1 sets its value for i.
  i = 100                     ! Segment 1 end
                              !
                              ! All images must wait for image 1 to set its i,
                              ! before reading i from image 1.
sync all                      ! Image control statement
i = i[1]                      ! Segment 2 start - all images read i from image 1
end                           ! Segment 2 end
```

Note that not using the image control statement in this example will result in a race condition - some images might try to read `i` from image 1 before image 1 finished setting its value. However, the standard does not allow this:

> "if a variable is defined on an image in a segment, it shall not be referenced, defined or become undefined in a segment on another image unless the segments are ordered"

Thus a standard conforming coarray program should not suffer from races.

All coarray programs implicitly synchronise at start and at termination.

Another new image control statement is `sync images.` It provides a more flexible means for image control. `sync images` takes a list of image indices with which it must synchronise:

```
if ( this_image() .eq. 3 ) sync images( (/ 2, 4, 5 /) )
```

There must be *corresponding* `sync images` statements on the images referenced by `sync images` statement on image 3, e.g.:

```
if ( this_image() .eq. 2 ) sync images( 3 )
if ( this_image() .eq. 4 ) sync images( 3 )
if ( this_image() .eq. 5 ) sync images( 3 )
```

Asterisk, `*`, is an allowed input. The meaning is that an image must synchronise with all other images:

```
if ( this_image() .eq. 1 ) sync images( * )
if ( this_image() .eq. 1 ) sync images( 1 )
```

In this example all images must synchronise with image 1, but not with each other, as would have been the case with `sync all`.

For cases when there are multiple `sync images` statements with identical sets of image indices, the standard sets the rules which determine which `sync images` statements correspond:

> "Executions of SYNC IMAGES statements on images M and T correspond if the number of times image M has executed a SYNC IMAGES statement with T in its image set is the same as the number of times image T has executed a SYNC IMAGES statement with M in its image set. The segments that executed before the SYNC IMAGES statement on either image precede the segments that execute after the corresponding SYNC IMAGES statement on the other image."

Here's an example of swapping coarray values between two images.

```
$ cat swap.f90
integer :: img, nimgs, i[*], tmp
                                  ! implicit sync all
  img = this_image()
nimgs = num_images()
   i = img                        ! i is ready to use

 if ( img .eq. 1 ) then
   sync images( nimgs )           ! explicit sync 1 with last img
    tmp = i[ nimgs ]
   sync images( nimgs )           ! explicit sync 2 with last img
      i = tmp
 end if

 if ( img .eq. nimgs ) then
   sync images( 1 )               ! explicit sync 1 with img 1
    tmp = i[ 1 ]
   sync images( 1 )               ! explicit sync 2 with img 1
      i = tmp
 end if
 write (*,*) img, i
                                  ! all other images wait here
end
$ ifort -coarray swap.f90
$ setenv FOR_COARRAY_NUM_IMAGES 5
$ ./a.out
           3           3
           1           5
           2           2
           4           4
           5           1
$
```

How many execution segments are there on each image?

Which `sync images` statements correspond?

8. Example program: segments and image control statements

```
cd examples/coarray/3swap
make
```

Tasks:

- Make the program standard conforming with `sync all` image control statements.
- Make the program standard conforming with `sync images` statements.
- Can you make the program deadlock?

9. Example program: deadlock

```
cd examples/coarray/4deadlock
make
```

Tasks:

- Try to run the program. Terminate with CTRL/C if stuck.
- Explain why the program deadlocks in terms of execution segments and image control statements.
- Modify the program to avoid the deadlock.

10. New Fortran 2008 construct: `do concurrent`

This do loop is intended for cases when the order of loop iterations is of no importance. The idea is that such loops can be optimised by a compiler.

```
integer :: i, a1(100)=0, a2(100)=1
do concurrent( i=1, 100 )
  a1(i) = i                ! valid, independent
  a2(i) = sum( a2(1:i) )   ! invalid, order is important
end do
```

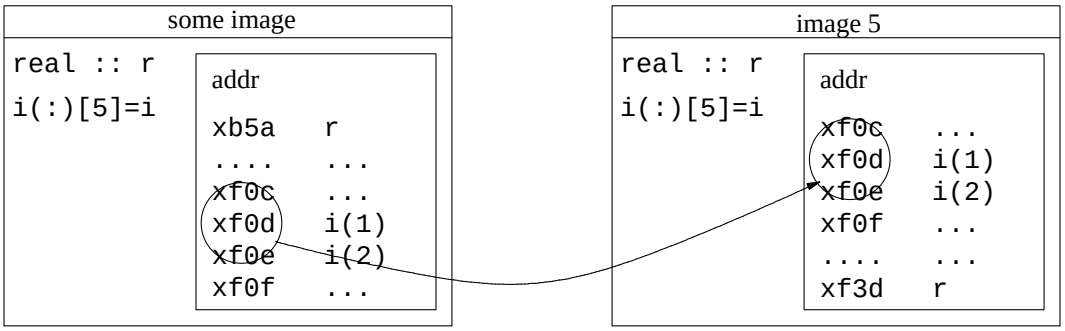
The exact list of restrictions on what can appear inside a `do concurrent` loop is long. These restrictions severely limit the usefulness of the `do concurrent` construct. While this new construct is potentially a portable parallelisation tool, there might or might not be a performance gain, depending on the implementation. In this tutorial `do concurrent` is used for comparison with coarrays in the π calculation example.

11. Implementation and performance

The standard deliberately (and wisely) says nothing on this.

A variety of underlying parallel technologies can be, and some are, used - MPI, OpenMP, SHMEM, GASNet, ARMCI, etc. As always, performance depends on a multitude of factors.[4]

The Standard *expects*, but does not require it, that coarrays are implemented in a way that each image knows the address of all coarrays in memories of all images, something like the integer coarray `i` in the illustration below. This is sometimes called *symmetric memory.* An ordinary, non-coarray, variable `r` might be stored at different addresses by different processes. Cray compiler certainly does this, other compilers likely do too.

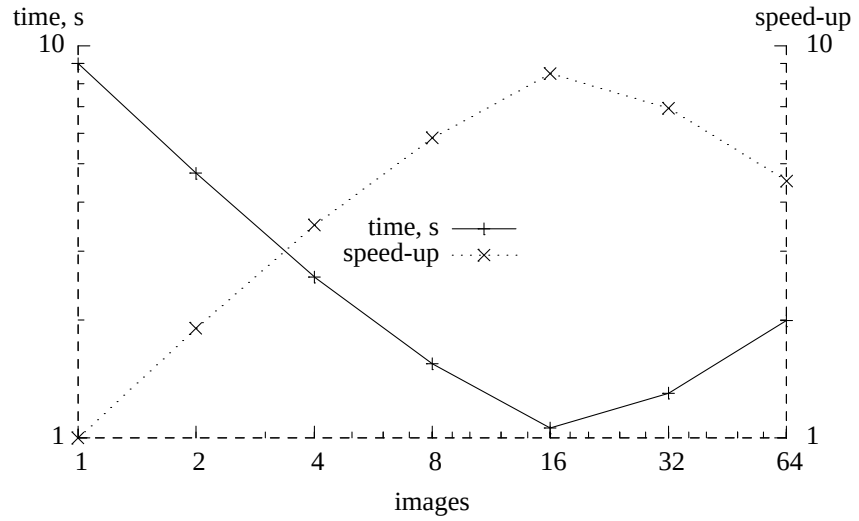


Example: calculation of π using the Gregory - Leibniz series:

$$\pi = 4\sum_{n=1}^{\infty}\frac{(-1)^{n-1}}{2n-1}$$

Given the series upper limit, each image sums the terms beginning with its image number and with a stride equal to the number of images. Then image 1 sums the contributions from all images. The segments are ordered by `sync all` to make sure all images finish calculating their partial sums before image 1 reads the values from all other images and adds those together.

Below is a sample scaling performance with ifort on 16-core nodes with 2.6Hz SandyBridge cores. As always, a great many things affect performance, coarrays are no exception.



[4] A. Fanfarillo, T. Burnus, S. Filippone, V. Cardellini, D. Nagle, and D. W. I. Rouson, "OpenCoarrays: open-source transport layers supporting coarray Fortran compilers" in *PGAS conf.* (2014). http://opencoarrays.org/yahoo_site_admin/assets/docs/pgas14_submission_7.30712505.pdf.

The key segment of the code, - the loop for partial π, and the calculation of the total π value, is shown below for the coarray code, and also for MPI, Fortran 2008 new intrinsic `DO CONCURRENT` and OpenMP.

Coarrays

```
do i = this_image(), limit, num_images()
 pi = pi + (-1)**(i+1) / real( 2*i-1, kind=rk )
end do
 sync all              ! global barrier
if (img .eq. 1) then
  do i = 2, nimgs
    pi = pi + pi[i]
  end do
  pi = pi * 4.0_rk
end if
```

MPI

```
do i = rank+1, limit, nprocs
 pi = pi + (-1)**(i+1) / real( 2*i-1, kind=rk )
end do
call MPI_REDUCE( pi, picalc, 1, MPI_DOUBLE_PRECISION, &
                 MPI_SUM, 0, MPI_COMM_WORLD, ierr )

picalc = picalc * 4.0_rk
```

`DO CONCURRENT`

```
loops = limit / dc_limit
do j = 1, loops
  shift = (j-1)*dc_limit
  do concurrent (i = 1:dc_limit)
    pi(i) = (-1)**(shift+i+1) / real( 2*(shift+i)-1, kind=rk )
  end do
  pi_calc = pi_calc + sum(pi)
end do

pi_calc = pi_calc * 4.0_rk
```

OpenMP

```
!$OMP PARALLEL DO DEFAULT(NONE) PRIVATE(i) REDUCTION(+:pi)
do i = 1, limit
 pi = pi + (-1)**(i+1) / real( 2*i-1, kind=rk )
end do
!$OMP END PARALLEL DO

pi = pi * 4.0_rk
```

Coarray implementation is closest to MPI. When coarray collectives are in the standard, the similarity will be even greater.

The table below is a subjective comparison of these four parallelisation methods.

Parallel method/language	Fortran standard	shared memory	distributed memory	ease of use	flexibility	performance
coarrays	yes	yes	yes	easy	high	high
`do concurrent`	yes	possibly	possibly	easy	poor	uncertain
OpenMP	no	yes	no	easy	limited	medium
MPI	no	yes	yes	hard	high	high

12. Example program: calculation of π

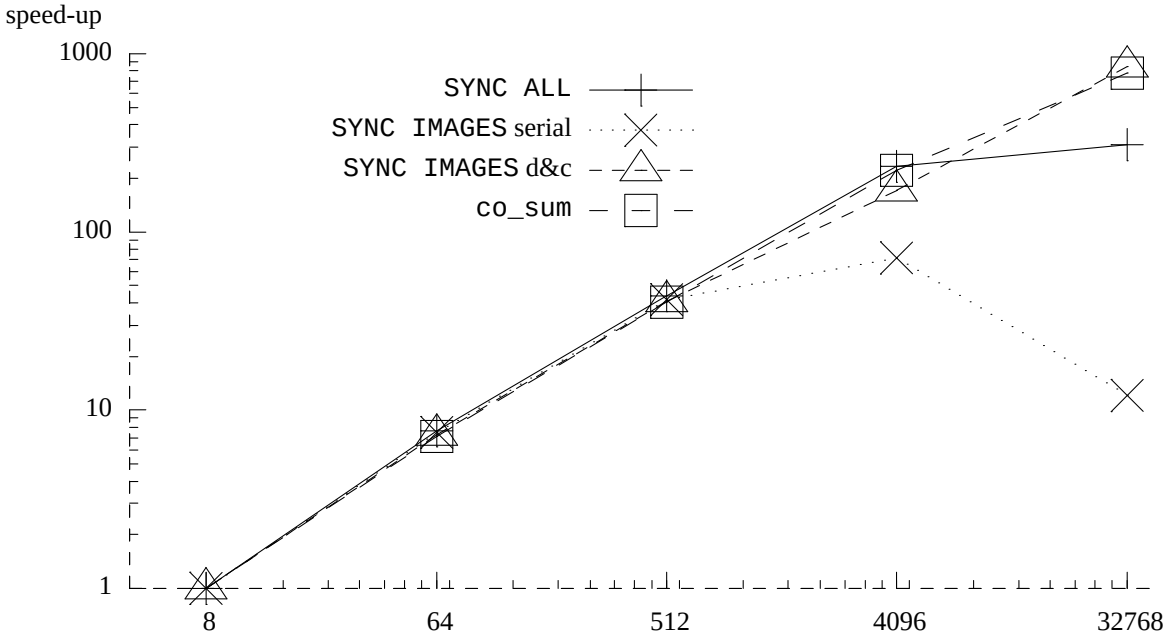
```
cd examples/coarray/5pi
make all
make run
```

For comparison, the same problem is solved with coarrays, `do concurrent` OpenMP and MPI. The source code files are: (1) coarray - `pi_ca.f90`, (2) do concurrent - `pi_dc.f90`, (3) OpenMP - `pi_omp.f90` and (4) MPI `pi_mpi.f90`.

Tasks

- Examine the coarray source code, `pi_ca.f90`, and add the necessary image control statements where required.
- Change the number of images and rerun the program, noting down the run time. What speed-up can you achieve?
- What is the relative speed of the 4 programs?
- Split the work between the images in an alternative fashion, by changing the `do` loop to `do i = (img-1)*cs+1, img*cs` where `cs=limit/nimgs`. Is the accuracy of the code maintained? Does the code run faster or slower?
- What happens when the series limit is not an exact multiple of the number of images?
- Try using several nodes. For this you need to submit your job to the queue. `pbs.sh` is the template job submission script. Make sure to modify the path to where your course files are.
- Intel compiler requires different options for shared and distributed memory compilation. The shared memory executable is `pi_ca.xs`. The distributed memory executable is `pi_ca.xd`. Examine the `Makefile` for more details. For distributed memory the Intel compiler needs a configuration file. The template is provided in `ca.conf`. The file must contain the number of processor and the name of the executable as the last argument. This file is updated from `pbs.sh`.

13. Another scaling example



This scaling data was obtained on Hector, the previous generation UK national supercomputer. The code is a microstructure simulation cellular automata model.[5] A three order magnitude speed has been achieved between 8 and 32k cores, an efficiency of about 25%.

`co_sum` is at present a Cray extension to the standard. This is a *collective* sum operation. Note that even `sync`all shows very impressive scaling, despite being the simplest image control statement, a global barrier.

14. Allocatable coarrays and coarray components of derived types

Coarray variables can be allocatable. Allocatable coarrays are declared with `:` for each dimension and each codimension:

```
real, allocatable :: r(:) [:]      ! real allocatable array coarray
complex, allocatable :: c[:]       ! complex allocatable scalar coarray
integer, allocatable :: i[:,:,:]   ! integer allocatable scalar coarray
                                   ! with 3 codimensions
```

As with non-allocatable coarray, the last upper codimension must be an asterisk on allocation, to allow for the number of images to be determined at runtime:

```
allocate( r(100) [*], source=0.0 )
allocate( c[*], source=cmplx(0.0,0.0) )
allocate( i[7,8,*], source=0 )
```

Sourced allocation was added in Fortran 2003.

Allocation and deallocation of coarrays involve implicit image synchronisation. Hence coarray allocation/deallocation must appear only in contexts which allow image control statements. This means that all

[5] A. Shterenlikht, "Fortran coarray library for 3D cellular automata microstructure simulation" in *Proceedings of the 7th International Conference on PGAS Programming Models*, ed. M. Weiland, A. Jackson & N. Johnson, The University of Edinburgh, UK (2013). ISBN: 978-0-9926615-0-2 http://www.pgas2013.org.uk/sites/default/files/finalpapers/Day2/R4/1_paper2.pdf.

images must allocate and deallocate a coarray. All allocated coarrays are automatically deallocated at program termination.

Coarrays must be allocated with the same bounds and cobounds on all images.

The following coarray allocations are **not** valid because the bounds or cobounds are not identical on all images. However, the processor (compiler) is not required to detect this violation of the standard.

```
allocate( r(10*this_image()) [*], source=0.0 )   ! not valid
allocate( i[7*this_image(), 8,*], source=0 )     ! not valid
```

Coarrays can be passed as arguments to subroutines. If a coarray is allocated in a subroutine, the dummy argument must be declared with `intent(inout)`. The bounds and cobounds of the actual argument must match those of the dummy argument.

```
module coalloc
 contains
 subroutine coal(i, b, cob)
   integer, allocatable, intent(inout) :: i(:) [:,:]
   integer, intent(in) :: b, cob
   allocate( i(b) [cob,*], source=0 )
 end subroutine coal
end module coalloc

program z
 use coalloc
 integer, allocatable :: i(:) [:,:]
 call coal( i, 8, 4 )
end program z
```

If arrays with different bounds are needed on different images, a simple solution is to have coarray components of a derived type:

```
$ cat pointer.f90
program z
implicit none
  type t
    integer, allocatable :: i(:)
  end type
    type(t) :: value[*]
    integer :: img
 img = this_image()
 allocate( value%i(img), source=img )   ! not coarray - no sync
 sync all
 if ( img .eq. num_images() ) value%i(1) = value[ 1 ]%i(1)
 write (*,*) "img", img, value%i
end program z
$ ifort -coarray -warn all -o pointer.x pointer.f90
$ setenv FOR_COARRAY_NUM_IMAGES 3
$ ./pointer.x
 img           1           1
 img           2           2           2
 img           3           1           3           3
$
```

15. Termination

In a coarray program a distinction is made between a *normal* and *error* termination.

Normal termination on one image allows other images to finish their work. `STOP` and `END PROGRAM` initiate normal termination.

New intrinsic `ERROR STOP` initiates error termination. The purpose of error termination is to terminate *all* images as soon as possible.

Example of a normal termination:

```
$ cat term.f90
implicit none
integer :: i[*], img
real :: r
img = this_image()
  i = img
if ( img-1 .eq. 0 ) stop "img cannot continue"
do i=1,100000000
 r = atan(real(i))
end do
write (*,*) "img", img, "r", r
end
$ ifort -coarray term.f90 -o term.x
$ ./term.x
img cannot continue
 img           2 r   1.570796
 img           4 r   1.570796
 img           3 r   1.570796
$
```

Image 1 has encountered some error condition and cannot proceed further. However, this does not affect other images. They can continue doing their work. Hence `STOP` is the best choice here.

Example of an error termination:

```
$ cat errterm.f90
implicit none
integer :: i[*], img
real :: r
img = this_image()
  i = img
if ( img-1 .eq. 0 ) error stop "img cannot continue"
do i=1,100000000
 r = atan(real(i))
end do
write (*,*) "img", img, "r", r
end
$ ifort -coarray errterm.f90 -o errterm.x
$ ./errterm.x
img cannot continue
application called MPI_Abort(comm=0x84000000, 3) - process 0
rank 0 in job 1  newblue3_53066   caused collective abort of all ranks
  exit status of rank 0: return code 3
$
```

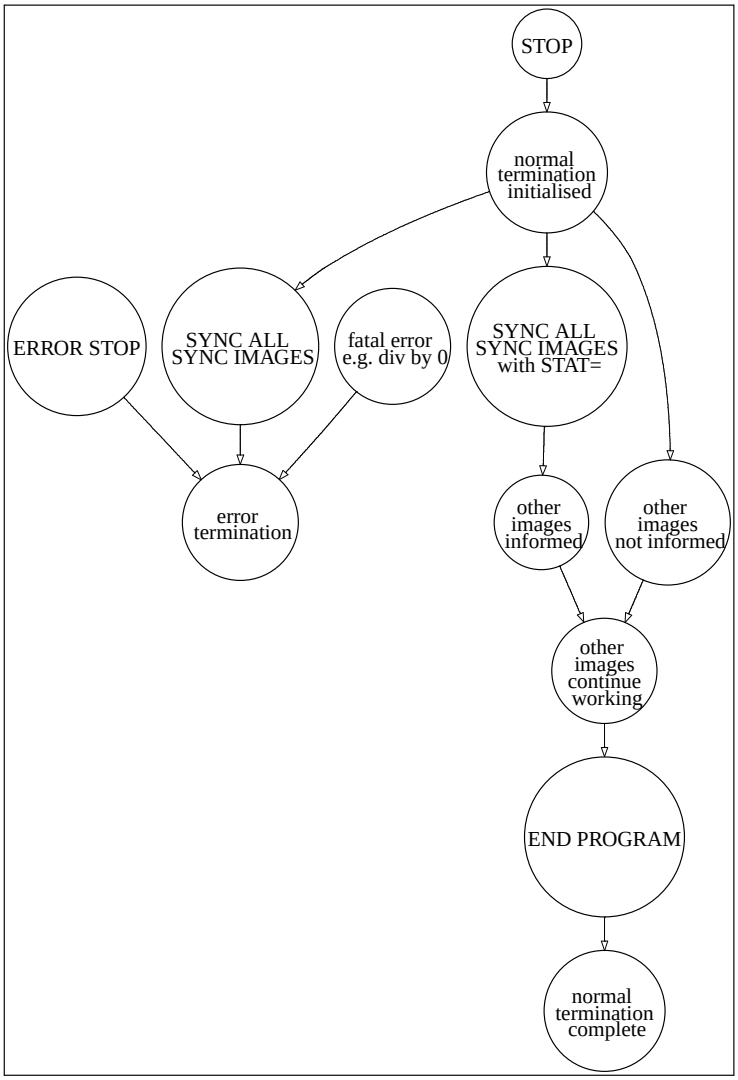
Here the error condition on image 1 is severe. It does not make sense for other images to continue. `ERROR STOP` is the appropriate choice here.

Note: the following does not seem to be supported by the Intel compiler v.15.

The standard provides a way to determine, via image control statements `sync images` and `sync all`, whether any image has initiated normal termination. For this both statements can use `stat=` specifier. If at the point of an image control statement some image has already initiated normal termination, then the integer variable given to `stat=` will be defined with the constant `stat_stopped_image` from the intrinsic module `iso_fortran_env`. The images that are still executing might decide to take a certain action with this knowledge:

```
use, intrinsic :: iso_fortran_env
integer :: errstat=0
  ! all images do work
sync all( stat=errstat )
if ( errstat .eq. stat_stopped_image ) then
  ! save my data and exit
end if
! otherwise continue normally
```

Below is a schematic flowchart illustrating steps taken during normal and error termination.



16. Example program: allocatable component of a derived type

```
cd examples/coarray/6pointer
make
```

Tasks

- Is image synchronisation necessary in this example? Why? Where?
- Add the necessary image synchronisation statement.
- Does the program work as expected on different numbers of images?

17. Example program: allocatable coarray

```
cd examples/coarray/7alloc
make
```

Tasks

- How many execution segments does the program have?
- What would happen if only one image called subroutine `coal`?
- Does `coal` need to be deallocated at the end of the program?

18. Example program: normal and error termination

```
cd examples/coarray/8term
make
```

Tasks

- Change the program to use error termination.

19. Next standard

The next Fortran standard is expected in 2015. It will have new coarray features, detailed in the technical specification TS 18508, "Additional Parallel Features in Fortran", WG5/N2033.[6] This is the 6th draft of this TS. It was approved in NOV-2014, subject to further corrections. TS 18508 includes:

- Teams - subsets of images working on independent tasks. This feature helps exploit functional parallelism in coarray programs. Proposed new statements are: `FORM TEAM`, `CHANGE TEAM` and `SYNC TEAM`. Proposed new intrinsics are: `GET_TEAM` and `TEAM_ID`.
- Events - similar to locks? Proposed new statements are: `EVENT POST` and `EVENT WAIT`. Proposed new intrinsic is `EVENT_QUERY`.
- Facilities to deal with failed images - think exascale... Proposed new statements are: `FAIL IMAGE`. Proposed new intrinsics are: `FAILED_IMAGES`, `IMAGE_STATUS` and `STOPPED_IMAGES`.
- New atomic intrinsics, such as: `ATOMIC_ADD`, `ATOMIC_OR` or `ATOMIC_XOR`.
- Collectives: `CO_MAX`, `CO_MIN`, `CO_SUM`, `CO_REDUCE` and `CO_BROADCAST`.

20. Coarray resources

The standard is the best reference. Draft version is available online[7] for free.

A more readable, but just as thorough, resource is the MFE[8] book.

Sections on coarrays, with examples, can be found in several further books.[9, 10, 11, 12]

At this time Fortran 2008 coarrays are fully supported only by the Cray compiler. The Intel v.15 coarray support is nearly complete. I've found bugs in both Cray and Intel compilers though.

	Fortran 2008 Features	Absoft	Cray	g95	gfortran	HP	IBM	Intel	NAG	Oracle	Pathscale	PGI
	Compiler version number	14	8.3.0		4.8		15.1	15	6.0	8,7, 32	4	14.4
2	**Submodules**	N	Y		N	N	Y	N	N	N	N	N
3	**Coarrays**	N	Y	P	P, 200	N	N	Y	N	N	N	N

(From ACM Fortran Forum[13])

G95 and GCC compilers support syntax, but until recently lacked the underlying inter-image communication library. However, a recent announcement of the OpenCoarrays project (http://opencoarrays.org) for "developing, porting and tuning transport layers that support coarray Fortran compilers" is likely to change this. The developers claim that GCC5 can already be used with OpenCoarrays.

In addition there are claims[14] that Rice Compiler (Rice University, USA) and OpenUH (University of Houston, USA) also support coarrays.

The Fortran mailing list, `COMP-FORTRAN-90@JISCMAIL.AC.UK`, and the Fortran Usenet newsgroup, `comp.lang.fortran`, are invaluable resources for all things Fortran, including coarrays.

---

[6] ISO/IEC JTC1/SC22/WG5 N2033, *TS 18508 Additional Parallel Features in Fortran* (6-NOV-2014).

[7] ISO/IEC JTC1/SC22/WG5 WD1539-1, *J3/10-007r1 F2008 Working Document.* http://j3-fortran.org/doc/year/10/10-007r1.pdf.

[8] M. Metcalf, J. Reid, and M. Cohen, *Modern Fortran explained*, Oxford, 7 Ed. (2011).

[9] I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*, Springer, 2 Ed. (2012).

[10] A. Markus, *Modern Fortran in practice*, Cambridge (2012).

[11] R. J. Hanson and T. Hopkins, *Numerical Computing with Modern Fortran*, SIAM (2013).

[12] N. S. Clerman and W Spector, *Modern Fortran: style and usage*, Cambridge (2012).

[13] I. D. Chivers and J. Sleightholme, "Compiler support for the Fortran 2003 and 2008 standards," *ACM Fortran Forum* **33**(2), pp. 38-51, revision 15 (AUG-2014).

[14] A. Fanfarillo, T. Burnus, S. Filippone, V. Cardellini, D. Nagle, and D. W. I. Rouson, "OpenCoarrays: open-source transport layers supporting coarray Fortran compilers" in *PGAS conf.* (2014). http://opencoarrays.org/yahoo_site_admin/assets/docs/pgas14_submission_7.30712505.pdf.

21. Example: parallel image processing

This example implements a halo exchange algorithm to speed up an image processing program.

We will need to view images on screen. Please connect to BlueCrystal with `ssh -X`.

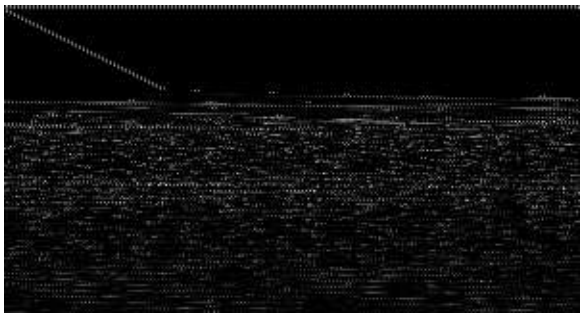
```
cd examples/coarray/9laplace
make
```

This directory contains several programs, each producing a separate output file.

	program	output
serial	edge.f90	edge.pgm
	back.f90	back.pgm
coarray fragmented along 1	co_edge1.f90	co_edge1.pgm
	co_back1.f90	co_back1.pgm
coarray fragmented along 1 and 2	co_edge2.f90	co_edge2.pgm
	co_back2.f90	co_back2.pgm

File `ref.pgm` is the reference picture:

To avoid confusion with coarray images, we use "picture" in this example to refer to a graphical image. This is a photo of Cray XC30, similar to the one installed as Archer, the current UK national supercomputer. `ref_edge.pgm` is a reference edge file:



`pgmio.f90` is a module dealing with reading and writing of the PGM files.

If the picture is read into a 2D array `p`, then the 2D array of edges, `e`, can be calculated like this:

```
e(i,j) = p(i-1,j) + p(i+1,j) + p(i,j-1) + p(i,j+1) - 4 * p(i,j)
```

where `i` takes values between `lbound(p, dim=1)` and `ubound(p, dim=1)` and `j` takes values between `lbound(p, dim=2)` and `ubound(p, dim=2)`.

Note that the expression for `e` uses values outside of the actual data array. So we need to extend the array sizes by one in each direction to store the "halo" elements.

You might recognise that the expression for `e` is a Laplacian of the original picture intensity:

$$\Delta p = p_{,ii} = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2}$$

Given the edges of a picture, it is possible to reconstruct the original picture, i.e. solve the Laplace equation, e.g. using the iterative Jacobi method. Inverting the expression for `e` we get the following simple iterative algorithm:

```
pnew(i,j) = 0.25 * ( p(i-1,j) + p(i+1,j) + p(i,j-1) + p(i,j+1) - e(i,j) )
        p = pnew
```

which is repeated until convergence.

However, convergence of this algorithm is very slow. Typically $10^5 - 10^6$ iterations are required. We will use coarrays to speed-up the execution.

The key idea is to partition the picture into smaller fragments, and delegate processing of each fragment to a separate processor (image). The picture may be partitioned along dimension 1:

or along dimension 2:

or along both dimensions 1 and 2:

Tasks:

- Study and run the serial edge detection program `edge.f90`. It produces file `edge.pgm`. Make sure it matches the reference edge file `ref_edge.pgm`. You can use UNIX command `diff`, possibly with `-q` switch.

- Study and run the serial picture reconstruction program `back.f90`. This program produces `back.pgm`. Try different number of iterations, `niter`, in `back.f90` until you get the exact match with `ref.pgm`. Write down the execution time of `back.f90`, required to achieve convergence. You can use UNIX command `time`, for example as `/usr/bin/time -fE`.
- Study `co_edge1.f90`. This is a coarray edge detection program, implementing picture fragmentation along direction 1. Run it. It produces the edge file `co_edge1.pgm`. Does it agree with `ref_edge.pgm`? Why? Add the missing image control statements to achieve the desired execution order. Make sure `co_edge1.pgm` agrees with `ref_edge.pgm`.
- Study `co_back1.f90`. This is a coarray picture reconstruction program, implementing fragmentation along direction 1. Use the same value for `niter` that you found with `back.f90`. Run `co_back1.f90`. It produces the reconstructed picture file `co_back1.pgm`. Does it agree with `ref.pgm`? Why? Add the missing image control statements to achieve the desired execution order. Make sure `co_back1.pgm` agrees with `ref.pgm`.
- Collect the run times of `co_back1.f90` for different numbers of images.
- Do `co_edge1.f90` or `co_back1.f90` work with just one image? Why?
- Study `co_edge2.f90`. This is a coarray edge detection program, implementing picture fragmentation along both directions 1 and 2. Run it. Note that this program takes the number of images along 1 as its only argument. So you need to make sure this argument is consistent with the total number of images.
- `co_edge2.f90` produces the edge file `co_edge2.pgm`. Does it agree with `ref_edge.pgm`. Why? Add the missing image control statements to achieve the desired execution order. Make sure the resulting `co_edge2.pgm` agrees with `ref_edge.pgm`.
- Study `co_back2.f90`. This is a coarray picture reconstruction program, implementing fragmentation along both directions 1 and 2. Run it. Note that this program takes the number of images along 1 as its only argument. It produces the reconstructed picture file `co_back2.pgm`. Does it agree with `ref.pgm`. Why? Add the missing image control statements to achieve the desired execution order. Make sure the resulting `co_back2.pgm` agrees with `ref.pgm`.
- Do `co_edge2.f90` or `co_back2.f90` work with just one image? Why?
- What is the highest speed-up you can achieve?