Fortran 2008 coarrays

Anton Shterenlikht

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

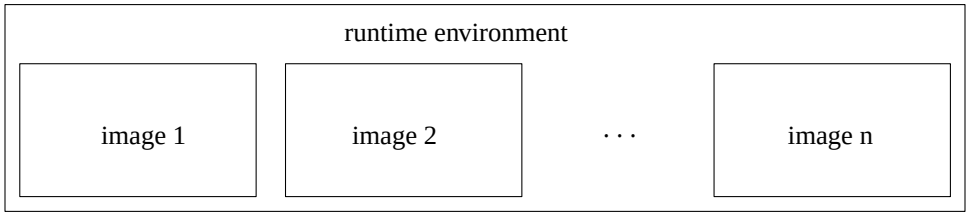
ABSTRACT

Coarrays are a Fortran 2008 standard feature intended for 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. 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 precisely the inverse 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 simple complete coarray programs are shown and compared to alternative parallel technologies - OpenMP, MPI and Fortran 2008 intrinsic "do concurrent". Inter image communication patterns and data transfer are illustrated. Finally an example of a materials microstructure simulation coarray program scaled up to 32k cores is shown. Problems with coarray I/O at this scale are highlighted and addressed with the use of MPI-I/O.

1 October 2014

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
$
```

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.

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

```
$ FOR_COARRAY_NUM_IMAGES=9
$ export FOR_COARRAY_NUM_IMAGES
$ ./one.x
image:  1 of  9
image:  2 of  9
image:  9 of  9
image:  8 of  9
image:  6 of  9
image:  3 of  9
image:  7 of  9
image:  4 of  9
image:  5 of  9
$
```

These results were obtained with Intel compiler 15.0.0 20140723.

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

2. Coarray syntax and remote calls

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

Examples of coarray variables:

```
integer :: i[*]                 ! scalar integer coarray with a single
                                ! codimension
integer, codimension(*) :: i    ! equivalent to the above
real :: r(100) [:]              ! real allocatable array coarray


!                 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 return these values: `lcobound`, `ucobound`, `this_image`, `image_index`

Remote calls are indicated by explicit reference to an image index using the square brackets, `[]`:

```
$ cat x.f90
integer :: img, i[*]
 img = this_image()
   i = img
 if ( img .eq. 1 ) i = i[ num_images() ]
 if ( img .eq. num_images() ) i = i[ 1 ]
 write (*,*) img, i
end
$ ifort -coarray x.f90
$ setenv FOR_COARRAY_NUM_IMAGES 4
$ ./a.out
           1           4
           3           3
           2           2
           4           1
$
```

This is a actually a race condition. Synchronisation between images is required here.

---

[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.*

The last upper cobound is always an *, meaning that it is only determined at run time. 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 z.f90
character( len=10 ) :: i[-3:2,5,*]
 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
$ ifort -coarray z.f90
$ setenv FOR_COARRAY_NUM_IMAGES 60
$ ./a.out
 this_image()          60
 this_image( i )           2           5           2
 lcobound( i )          -3           1           1
 ucobound( i )           2           5           2
 image_index(ucobound(i))          60
$ setenv FOR_COARRAY_NUM_IMAGES 55
$ ./a.out
 this_image()          55
 this_image( i )          -3           5           2
 lcobound( i )          -3           1           1
 ucobound( i )           2           5           2
 image_index(ucobound(i))           0
$
```

Coarrays must be of the same shape on all images. If arrays of different shape/size 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
$
```

3. Synchronisation

All images synchronise at program initialisation and at program termination.

`sync all` is a global barrier - all images wait for each other.

`sync images` is for more flexible synchronisation.

```
$ cat y.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 y.f90
$ setenv FOR_COARRAY_NUM_IMAGES 5
$ ./a.out
           3           3
           1           5
           2           2
           4           4
           5           1
$
```

A deadlock example:

```
$ cat deadlock.f90
if ( this_image() .eq. num_images() ) sync images( 1 )
end
$ ifort -coarray deadlock.f90
$ ./a.out
```

deadlock!

CTRL/C

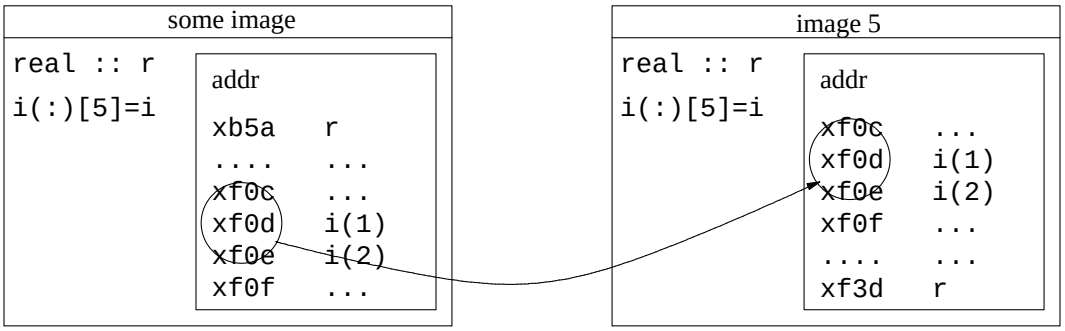
Allocation and deallocation of allocatable coarrays always involves *implicit* synchronisation.

4. 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.

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.

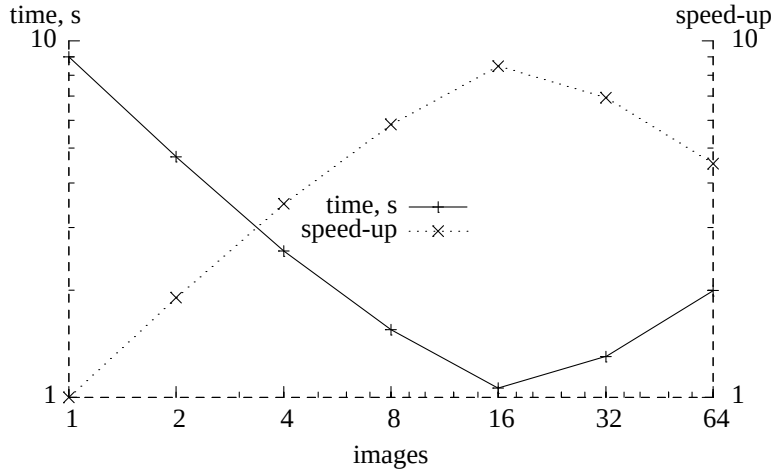


Cray compiler certainly does this, other compilers likely do too.

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

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

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.



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. To avoid the race condition, image 1 must make sure that all images have completed their calculations, before attempting to read the values from them. Hence synchronisation between the images is required. Here we use `sync all,` the global barrier.

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.

5. 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
$
```

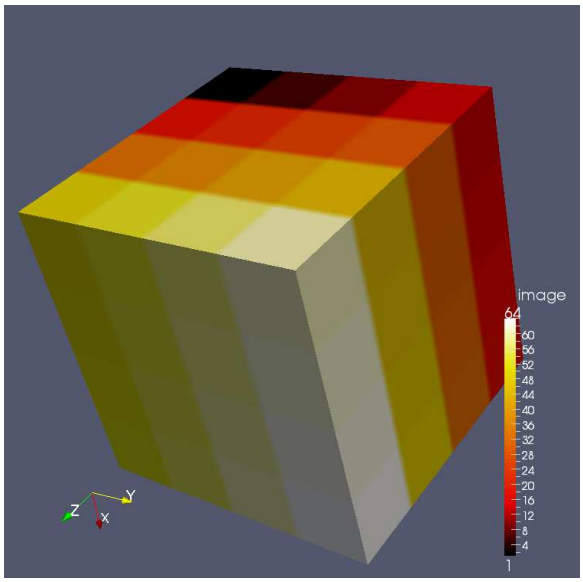
Here the error condition on image 1 is severe. It does not make sense for other images to continue. `ERROR STOP` is the appropriate choice here.

6. Cellular automata materials library

The library† is used to simulate the evolution of polycrystalline microstructures, including grain coarsening and grain boundary migration, and transgranular cleavage. The main coarray variable is:

```
integer :: space(:,:,:,:) [:,:,:]
```

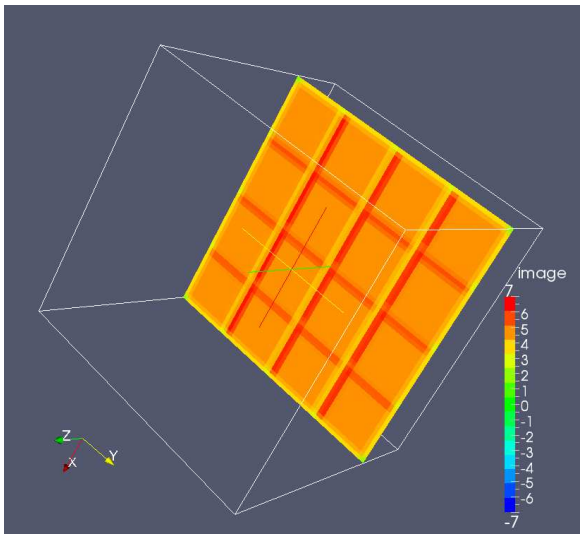
The model space is a "box" made from coarrays on all images. This model is made of 64 images.



Halo exchange is simple with coarray syntax. This command reads model cells from an image with cosubscript one lower than this image, along codimension 2, into halo cells on this image:

```
if ( imgpos(2) .ne. lcob(2) )                              &
  space( lbr(1):ubr(1), lbv(2), lbr(3):ubr(3), : ) =       &
  space( lbr(1):ubr(1), ubr(2), lbr(3):ubr(3), : )         &
    [ imgpos(1), imgpos(2)-1, imgpos(3) ]
```

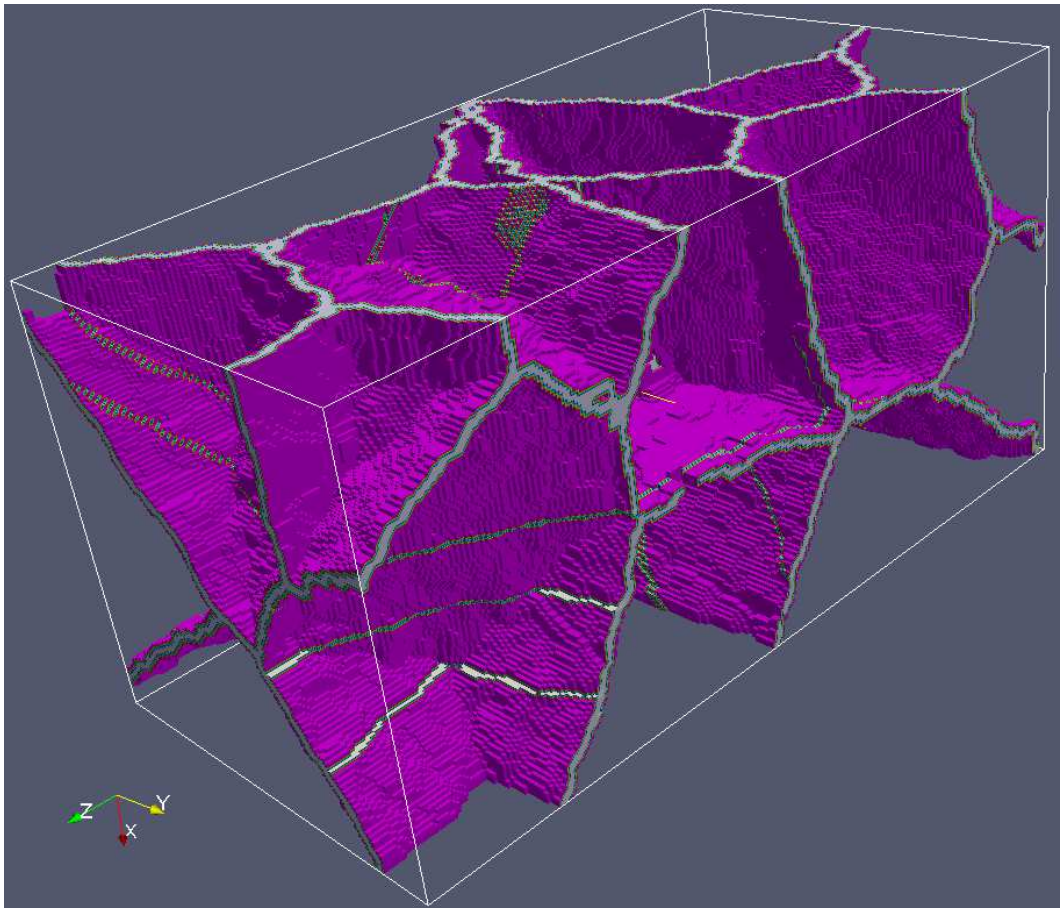
This is a slice of the model space. The halo cells are highlighted.



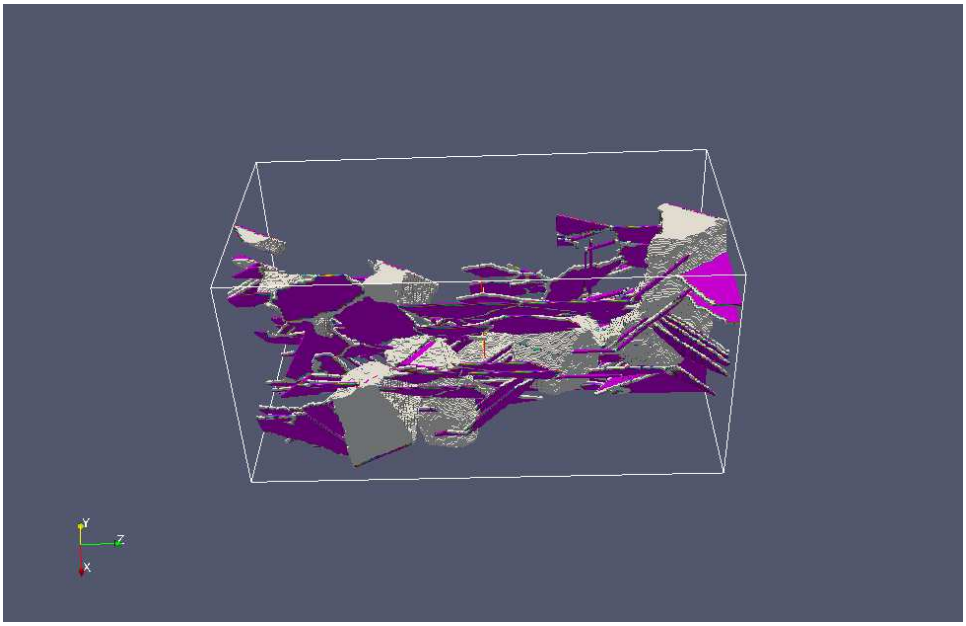
† http://eis.bris.ac.uk/~mexas/cgpack/ - Cellular Grains PACKage, freely available under BSD 2-clause licence.

Model results and performance

Grain boundaries in a random equiaxed microstructure:



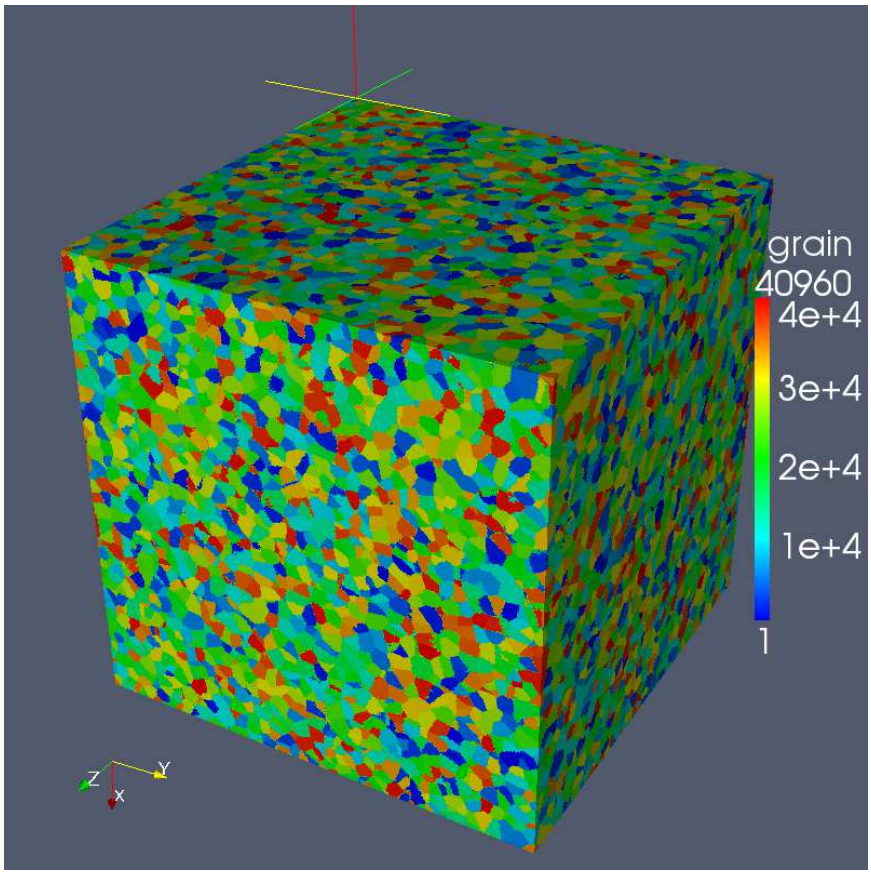
A cleavage crack in a polycrystalline microstructure:



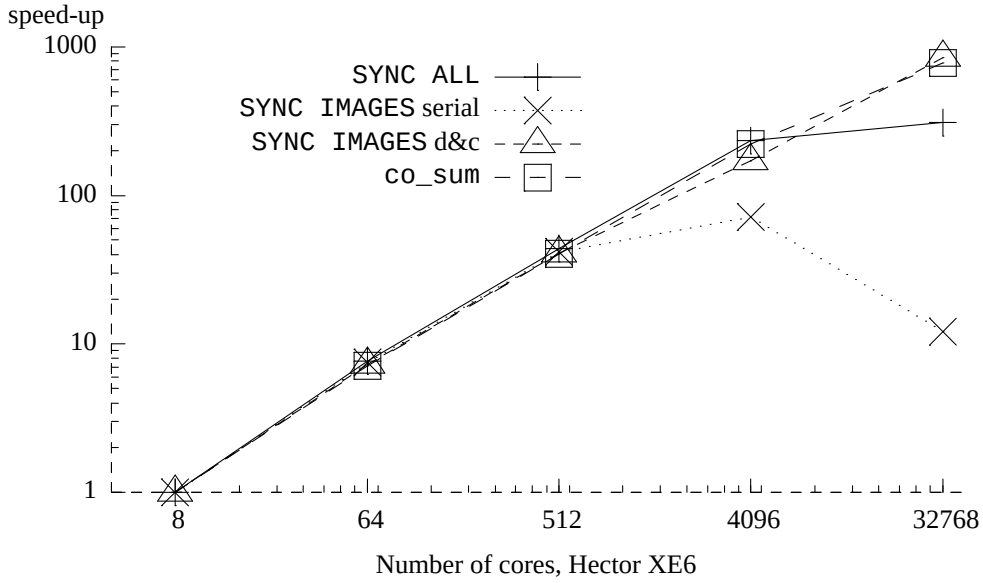
A Microstructure simulated with the space coarray allocated as

```
allocate( space( 200, 200, 200 ) [ 8, 8, * ], source=0, stat=errstat )
```

on 512 images. The last upper codimension is 8. In total there are 4×10^9 cells within 40,960 grains. The HECToR wall time was 5m, of which most most I/O!



The code shows linear scaling, with 1/4 efficiency, up to 32k cores (images).



7. I/O

Lack of parallel I/O provision in Fortran standard is, in my opinion, a major problem.

There are several ways to output the results from the model:

- A single image acts as a single writer. It reads data from all images and writes it into a single file in the correct order, ready for post-processing. This is very slow!

```
do coi3 = lcob(3), ucob(3)
 do i3 = lb(3), ub(3)
  do coi2 = lcob(2), ucob(2)
   do i2 = lb(2), ub(2)
    do coi1 = lcob(1), ucob(1)
     write( unit=iounit, iostat=errstat ) &
       space( lb(1):ub(1), i2, i3, stype ) [ coi1, coi2, coi3 ]
    end do
   end do
  end do
 end do
end do
```

- Each image writes its own data into a separate file. This put a lot of pressure on the OS, e.g. forcing the OS to create many file descriptors simultaneously. A lot of work is then still needed to put the data into a single file in the correct order. Alternatively, readers must be designed which can read the model from multiple files, in the right order. Either way, this is a lot of work.
- Each image writes its data into a shared file in the right place. This is not supported by Fortran, but is supported by MPI/IO†.

```
call MPI_Type_create_subarray(                    &
  arrdim, arraygsize, arraysubsize, arraystart,   &
  MPI_ORDER_FORTRAN, MPI_INTEGER, filetype, ierr )

call MPI_File_set_view( fh, disp, MPI_INTEGER,    &
  filetype, 'native', MPI_INFO_NULL, ierr )

call MPI_File_write_all(fh, coarray(1,1,1, stype), &
  arraysubsize(1)*arraysubsize(2)*arraysubsize(3), &
  MPI_INTEGER, status, ierr )
```

On Lustre file system (lfs, a popular parallel file system), after some optimisation of `lfs stripe size` and `lfs stripe count`, I/O rates up to 2.3 GB/s were achieved on HECToR. This is a speedup of over 20 compared to using a single writer.

† This MPI/IO code was written by David Henty, EPCC, d.henty@epcc.ed.ac.uk, Copyright 2013 The University of Edinburgh.

8. 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/N2027.[4] This is the 5th draft of this TS. It was approved in SEP-2014, subject to further corrections. The final draft is due by NOV-2014. Ask John Reid for the latest status.

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`.

The language will be a lot reacher, but more complex to learn and use.

[4] ISO/IEC JTC1/SC22/WG5 N2027, *TS 18508 Additional Parallel Features in Fortran* (22-AUG-2014).

9. Coarray resources

The standard is the best reference. Draft version is available online[5] for free.

A more readable, but just as thorough, resource is the MFE[6] book.

Sections on coarrays, with examples, can be found in several further books.[7, 8, 9, 10]

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[11])

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[12] 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.

---

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

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

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

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

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

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

[11] 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).

[12] A. Fanfarillo, *Coarrays in GNU Fortran* (JUN-2014). http://opencoarrays.org/yahoo_site_admin/assets/docs/Coarrays_GFortran.217135934.pdf.