Advanced Features

Parallel Programming with Fortran Coarrays

MSc in HPC

David Henty, Alan Simpson (EPCC)
Harvey Richardson, Bill Long (Cray)

CRAY THE SUPERCOMPUTER COMPANY | epcc

Advanced Features: Overview

- Execution segments and Synchronisation
- Non-global Synchronisation
- Critical Sections
- Visibility of changes to memory
- Other Intrinsics
- Miscellaneous features
- Future developments

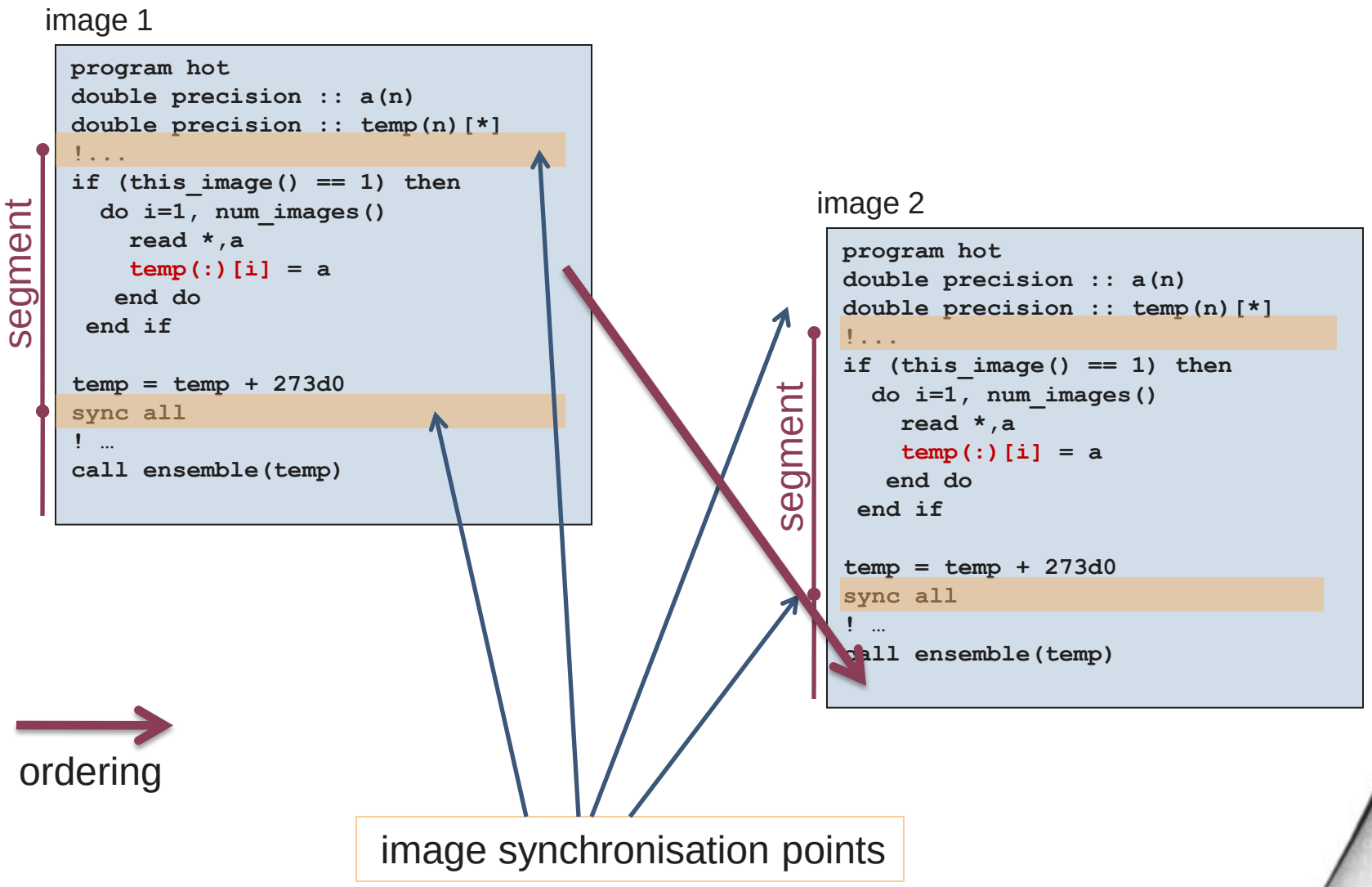
More on Synchronisation

- We have to be careful with one-sided updates
  - If we read remote data, was it valid?
  - Could another process send us data and overwrite something we have not yet used?
  - How do we know when remote data has arrived?
- The standard introduces **execution segments** to deal with this: segments are bounded by image control statements

The standard can be summarized as follows:

- If a variable is defined in a segment, it must not be referenced, defined, or become undefined in another segment unless the segments are ordered – *John Reid*

Execution Segments



Synchronisation mistakes

- This code is wrong

```
subroutine allreduce_max_getput(v,vmax)
  double precision, intent(in)  :: v[*]
  double precision, intent(out) :: vmax[*]
  integer i
  sync all

  vmax=v
  if (this_image()==1) then
    do i=2,num_images()
      vmax=max(vmax,v[i])
    end do
    do i=2,num_images()
      vmax[i]=vmax
    end do
  end if
  sync all
```

CRAY THE SUPERCOMPUTER COMPANY | epcc

Synchronisation mistakes

- It breaks the rules

```
subroutine allreduce_max_getput(v,vmax)
  double precision, intent(in)  :: v[*]
  double precision, intent(out) :: vmax[*]
  integer i
  sync all

  vmax=v
  if (this_image()==1) then
    do i=2,num_images()
      vmax=max(vmax,v[i])
    end do
    do i=2,num_images()
      vmax[i]=vmax
    end do
  end if
  sync all
```

Synchronisation mistakes

- This is ok

```
subroutine allreduce_max_getput(v,vmax)
  double precision, intent(in)  :: v[*]
  double precision, intent(out) :: vmax[*]
  integer i
  sync all

  if (this_image()==1) then
    vmax=v
    do i=2,num_images()
      vmax=max(vmax,v[i])
    end do
    do i=2,num_images()
      vmax[i]=vmax
    end do
  end if
  sync all
```

More about `sync all`

- Usually all images execute the same **`sync all`** statement
- But this is not a requirement...
  - Images execute different code with different **`sync all`** statements
  - All images execute the first **`sync all`** they come across and….
    - this may match an arbitrary **`sync all`** on another image
    - causing incorrect execution and/or deadlock
- Need to be careful with this ‘feature’
  - Possible to write code which doesn’t deadlock but gives wrong answers

CRAY THE SUPERCOMPUTER COMPANY | epcc |

More about `sync all`

- e.g. Image practical: wrong answer

```
! Do halo swap, taking care at the upper and lower picture boundaries

    if (myimage < numimage) then
      oldpic(1:nxlocal, nylocal+1) = oldpic(1:nxlocal, 1)[myimage+1]

    sync all
    end if

! ... and the same for down halo
! Now update the local values of newpic
  ...

! Need to synchronise to ensure that all images have finished reading the
! oldpic halo values on this image before overwriting it with newpic
  sync all
  oldpic(1:nxlocal,1:nylocal) = newpic(1:nxlocal,1:nylocal)

! Need to synchronise to ensure that all images have finished updating
! their oldpic arrays before this image reads any halo data from them

  sync all
```

All images NOT executing this sync all

All images ARE executing this sync all

More about **sync all**

- **sync images(*imageList*)**
  - Performs a synchronisation of the image executing **sync images** with each of the images specified in ***imageList***
  - ***imageList*** can be an array or a scalar

```
    if (myimage < numimage) then
      oldpic(1:nxlocal, nylocal+1) = oldpic(1:nxlocal, 1)[myimage+1]
    end if
    if (myimage > 1) then
      oldpic(1:nxlocal, 0) = oldpic(1:nxlocal, nylocal)[myimage-1]
    end if

! Now perform local pairwise synchronisations

    if (myimage == 1 ) then
      sync images( 2 )
    else if (myimage == numimage) then
      sync images( numimage-1 )
    else
      sync images( (/ myimage-1, myimage+1 /) )
    end if
```

Other Synchronisation

- Critical sections
  - Limit execution of a piece of code to one image at a time
  - e.g. calculating global sum on master image

```
integer :: a(100)[*]
integer :: globalSum[*] = 0, localSum
... ! Initialise a on each image

localSum = SUM(a) !Find localSum of a on each image

critical
   globalSum[1] = globalSum[1] + localSum
end critical
```

CRAY THE SUPERCOMPUTER COMPANY | epcc

Other Synchronisation

- **`sync memory`**
  - Coarray data held in caches/registers made visible to all images
    - requires some other synchronisation to be useful
    - unlikely to be used in most coarray codes
- Example usage: Mixing MPI and coarrays

  loop: coarray operations

  sync memory

  call MPI_Allreduce(...)
- **`sync memory`** implied for **`sync all`** and **`sync images`**

Other Synchronisation

- **`lock`** and **`unlock`** statements
  - Control access to data defined or referenced by more than one image
    - as opposed to **`critical`** which controls access to lines of code
    - **`USE iso_fortran_env`** module and define coarray of **`type(lock_type)`**
  - e.g. to lock data on image 2

```
type(lock_type) :: qLock[*]

lock(qLock[2])
!access data on image 2
unlock(qLock[2])
```

Other Intrinsic functions

- **`lcobound(z)`**
  - Returns lower cobounds of the coarray **`z`**
  - **`lcobound(z,dim)`** returns lower cobound for codimension **`dim`** of **`z`**
- **`ucobound(z)`**
  - Returns upper cobounds of the coarray **`z`**
  - **`lcobound(z,dim)`** returns upper cobound for codimension **`dim of z`**

- **`real :: array(10)[4,0:*]`** on 16 images
  - **`lcobound(array)`** returns **`[ 1, 0 ]`**
  - **`ucobound(array)`** returns **`[ 4, 3 ]`**

More on Cosubscripts

- `integer :: a[*]` on 8 images
  - cosubscript `a[9]` is not valid

- `real :: b(10)[3,*]` on 8 images
  - `ucobound(b)` returns `[ 3, 3 ]`
  - cosubscript `b[2,3]` is valid (corresponds to image 8)…
  - …but cosubscript `b[3,3]` is invalid (image 9)

- Programmer needs to make sure that cosubscripts are valid
  - `image_index` returns 0 for invalid cosubscripts

CRAY THE SUPERCOMPUTER COMPANY | epcc |

Assumed Size Coarrays

- Codimensions can be remapped to corank greater than 1
  - useful for determining optimal extents at runtime

```
program 2d
real, codimension[*] :: picture(100,100)
integer :: numimage, numimagex, numimagey
numimage = num_images()

call get_best_2d_decomposition(numimage,&
       numimagex, numimagey)
! Assume this ensures numimage=numimagex*numimagey

call dothework(picture, numimagex, numimagey)
...
contains
   subroutine dothework(array, m, n)
   real, codimension[m,*] :: array(100,100)
   ...
end subroutine dothework
```

I/O

- Each image has its own set of input/output units
- units are independent on each image
- Default input unit is preconnected on image 1 only
  - **`read *,…`** , **`read(*,…)`**…
- Default output unit is available on all images
  - **`print *,…`** , **`write(*,…)`**…
  - It is expected that the implementation will merge records from each image into one stream

Program Termination

- STOP or END PROGRAM statements initiate *normal termination* which includes a synchronisation step
- An image's data is still available after it has initiated normal termination
- Other images can test for this using STAT= specifier to synchronisation calls or allocate/deallocate
  - test for STAT_STOPPED_IMAGE (defined in ISO_FORTRAN_ENV module)
- The ERROR STOP statement initiates error termination and it is expected all images will be terminated.

Coarray Technical Specification TS18508

- These are described in TS18508: Additional Parallel Features in Fortran
- Due for publication and will be incorporated into Fortran 2015
- This provides for:
  - image teams
  - collective intrinsics
  - Atomics
  - Events
  - Failure handling

TS: Teams

- Often useful to consider subsets of processes
  - e.g. MPI communicators
- Subsets not currently supported in Fortran, e.g.
  - **`sync all`**: all images
  - **`sync images`**: pairwise mutual synchronisation

- Extension involves TEAMS of images
  - user creates teams and can change execution context to new team which must be a subset of current team
  - Collectives then apply within the team
  - **`sync all`** applies to the current team; there is a new **`sync team`** statement

TS: Teams…

- Operating within a new **team**

```
type(team_type) :: odd_even !From ISO_FORTRAN_ENV

me = this_image()
form subteam( 2-mod(me,2) , odd_even)
change team (odd_even)
  select case (team_id())
    case (1)
      ! Code for odd images
    case (2)
      ! Code for even images
    end select
  end team
```

TS: Teams...

- Image selectors can reference a **team**

```
type(team_type) :: odd_even

! Access via team variable
a(:) = b(:)[image, team = odd_even]
! Access using team_id
a(:) = c(:)[image, team_number = id]
```

- **this_image() now accepts a TEAM argument**

- Can define a team variable for the current, parent, or initial team.

CRAY THE SUPERCOMPUTER COMPANY | epcc |

TS: Collectives

- Collective operations a key part of real codes
  - broadcast
  - global sum
  - ...
- Supported in other parallel models
  - OpenMP reductions
  - **`MPI_Allreduce`**
- Not currently supported for coarrays
  - efficient implementation by hand is difficult
  - calling external MPI routines rather ugly

CRAY THE SUPERCOMPUTER COMPANY | epcc |

TS: Collective intrinsic subroutines

- Collectives, with in/out arguments, invoked by same statement on all images (or team of images)
- Routines
  - CO_BROADCAST
  - CO_MAX, CO_MIN, CO_SUM and CO_REDUCE
    - Optional RESULT_IMAGE argument to direct result to one image only
  - basically reproduce the MPI functionality

- Main data argument is overwritten with result
- There is no separate synchronization at begin/end
- The Cray Fortran compiler provides co_sum, co_bcast and co_min/max: see manpages

TS: Atomic operations

- Critical or lock synchronisation sometimes overkill
  - `counter[1] = counter[1] + 1`
- Simple atomic operations can be optimised
  - e.g. OpenMP `atomic`

```
!$OMP atomic
sharedcounter = sharedcounter + 1
```

- New variable types and operations for coarrays

TS: Atomic variables

- Fortran already includes some atomic support (define,ref)
- TS expands on this to supports atomic compare and swap, fetch and add, bitwise operations, …

```
integer (atomic_int_kind) :: counter[*]
integer old

call atomic_define(counter[1],0)
call atomic_add(counter[1],1)
call atomic_fetch_add(counter[1],1,old)
call atomic_ref(countval,counter[1])
```

CRAY THE SUPERCOMPUTER COMPANY | epcc |

TS: Events

- Events provide very simple synchronization support
- An image can post an event to another image to inform it that it can proceed
- Event variables are used and posted events increment the event variable
- Images can query events (to obtain the event count)
- Images can wait on events reaching a count,
  when the wait returns the count is reset
- The EVENT POST and WAIT statements are image control statements (this means events can be used to indicate that variable updates have happened)

TS: Image Failures

- Failed images do not support definition or reference of variables and are not executing statements
- The program can determine if image failure support is provided
- Various image control statements, collective and atomic intrinsic subroutines can return a status value to indicate a failed image was detected
- It is possible to test for failed or stopped images


CRAY
THE SUPERCOMPUTER COMPANY
epcc