Further coarray features in Fortran 2015

John Reid, ISO Fortran Convener,
JKR Associates and
Rutherford Appleton Laboratory

BCS Fortran Specialist Group
London, 1 October 2015
Items removed in 2008

In February 2008, it was decided to move the following features into a Technical Specification (a mini Standard) on *Additional Parallel Features in Fortran*

Teams and features that require teams.
The collective intrinsic subroutines.
The `notify` and `query` statements.
File connected on more than one image, unless preconnected to the unit specified by `output_unit` or `error_unit`.
The Technical Specification (TS)

In 2011, it decided that the choice of features should be reviewed while not altering the overall size of the addition.

This choice was made during the 2012 WG5 meeting and modified during the WG5 meetings in 2013 and 2014.

The TS was completed at the 2015 WG5 meeting and will be published soon.

I will describe the set of additional features it contains. They will all be included in Fortran 2015.
Teams

Needed for independent computations on subsets of images.

Code that has been written and tested on whole machine should run on a team.

Therefore, image indices need to be relative to team.

Collective activities, including syncs and allocations, need to be relative to team.
The intrinsic module iso_fortran_env contains a derived type team_type. A scalar object of this type identifies a team of images.

The same form team statement must be executed on all images of a team to form subteams

    form team(number,new_team)

Images with the same value of number form a new team. All images of the current team synchronize.
Change team construct

change team (team, local[*] => corray)
  ! Block executed as a team
  if (team_number() == 1) then ! New intrinsic
  : ! Code for team 1
else
  :
end team

Associating local with coarray allows corank and cobounds to change. Other attributes are unchanged.

The new teams synchronize at change team and end team.

Changing teams is likely to be costly – avoid doing it often.
Example
This code splits images into two groups and implicitly synchronizes each of them:

use iso_fortran_env
integer :: i
type(team_type) :: team
i = 1 + 2*this_image()/num_images()
form team(i,team)
change team (team)
    :
end team
Allocation in a team

On leaving a team construct, any allocated coarrays that were allocated in the construct are deallocated, even if they have the SAVE attribute.

This is needed to preserve *symmetric memory* – a vendor can arrange for the address of a coarray to be the same on all images of a team.
Accessing parent or sibling team

: type(team_type) ::= initial, block

initial = get_team()

i = ...

form team(i, block)

change team (block)

: sync team(initial)

s = a(1)[team=initial::mep+1]

t = a(1)[team_number=2::me2+1]

end team
Failed images

The probability of a particular image failing is small, but if the number of images is huge, the probability that one or more fails is significant.

Hence the concept of continuing execution in the presence of failed images.

If an image is considered to be failed, it remains so for the rest of the program execution.
failed_images intrinsic function

failed_images()
Returns an integer array holding image indices of failed images in the current team.

failed_images(team)
Returns an integer array holding image indices of failed images in team.
Testing for failed image in image control statement

parent = get_team()
change team (team_a)

: sync_all(parent, stat=st)
if (st==stat_failed_image) exit
end team

sync_all(stat=st)
if (st==stat_failed_image) then
: Deal with failure
end if
Testing for failed image in a remote reference

use iso_fortran_env
  :
  a = b[image,stat=st]
if (st==stat_failed_image) then
  : Deal with failure
end if
fail_image statement

The statement

    fail_image

causes an image to behave as failed. Useful for debugging.
Collectives

The collective subroutines are reduced in number, but a general reduction is added. They are `co_broadcast`, `co_max`, `co_min`, `co_sum`, `co_reduce`.

Invoked by the same statement on all images of the team and involve synchronization within them, but not at start and end.

The main argument is not required to be a coarray.
co_broadcast and co_max

call co_broadcast (a, source_image)
Copy source from a to all images of the current team.

call co_max (a)
On all images, replace a by maximum value of a on all images of the current team.

call co_max (a, result_image)
On result_image, replace a by maximum value of a on all images of the current team.
co_min, co_sum, co_reduce

co_min and co_sum are just like co_max.

co_reduce is also just like co_max but has extra argument

call co_reduce(a, operator) or
call co_reduce(a, operator, source_image)

operator is a pure function with two arguments of the same type as source. Applied just like max, min, sum.
Keeping source for collectives

If you want to retain the source, it is easy:

```python
result = a
    call co_max (result)
```
Events

Events are useful if one or more images need to do something before another image can continue.

For example, in the multifrontal method for factorizing a sparse matrix, work at a node of the assembly tree has to wait for all the work at its child nodes to be completed.
Event variable

An event variable is a scalar coarray of type `event_type`. It contains a count which increases by 1 each time the event is “posted”.

```fortran
use iso_fortran_env

_type(event_type), save :: event[*]

: event post(event[i]) ! Atomic
  : if(this_image()==i) then
    event wait(event)
    ! Waits until count >= 1, then atomically
    ! decreases it by 1 and continues
```
Wait for count

Can wait until count reaches given value:

```c
event wait(event, until_count=value)
    ! Waits until count >= value and then
    ! atomically decreases it by value
```

Useful if several images need to perform their actions before the executing image can continue.
Query count value

call event_query (event, count)
Sets count to the count value of event.

It is an atomic intrinsic subroutine.
More intrinsic atomic subroutines

call atomic_add(atom, value)
  ! atom = atom + value

call atomic_fetch_add(atom, value, old)
  ! old = atom; atom = atom + value

call atomic_cas(atom, old, compare, new)
  ! old = atom;
  if (atom == compare) atom = new

Also

atomic_and, atomic_fetch_and,
atomic_or, atomic_fetch_or,
atomic_xor, atomic_fetch_xor
Summary of features in TS

Teams
Collectives
Events
Failed images
More atomics