Dynamic coarrays

Only dynamic form: the allocatable coarray.

```
real, allocatable :: a(:)[:], s[:,:]
    :
allocate ( a(n)[*], s[-1:p,0:*] )
```

The bounds, cobounds, and length parameters must not vary between images.

All images synchronize at the same **`allocate`**, **`deallocate`** or **`call move_alloc`** statement so that they can all perform their allocations and deallocations in the same order.

Coarray dummy arguments

A dummy argument may be a coarray.

It may be of explicit shape, assumed size, assumed shape, or allocatable:

```
subroutine subr(n,w,x,y,z)
   integer :: n
   real,save :: w(n)[n,*]  ! Explicit shape
   real,save :: x(n,*)[*]  ! Assumed size
   real,save :: y(:,:)[*]  ! Assumed shape
   real, allocatable :: z(:)[:,:]
```

Coarray dummy arguments (cont)

```
subroutine subr(n,w,x,y,z)
   integer :: n
   real,save :: w(n)[n,*]  ! Explicit shape
   real,save :: x(n,*)[*]  ! Assumed size
```

Where the bounds or cobounds are declared, there is no requirement for consistency between images. The local values are used to interpret a remote reference. Different images may be working independently.

There are rules to ensure that copy-in copy-out of a coarray is never needed.

Coarrays and `save`

Unless allocatable, a coarray that is local to a procedure must be given the **`save`** attribute.

This is to avoid such an coarray going out of scope on return from a procedure, which would require synchronization.

Not required for allocatables because of their use in recursive procedures.

Structure components

Coarray not allowed to be a pointer but may be of a derived type with allocatable or pointer components.

Provides a simple but powerful mechanism for cases where the size varies from image to image, avoiding loss of optimization.

Pointers must have targets in their own image:

```
q => z[i]%p        ! Not allowed
allocate(z[i]%p) ! Not allowed
```

Program termination

Normal termination (**`stop`** or **`end`**) occurs in three steps: *initiation*, *synchronization*, and *completion*. Data on an image is available to the others until they all complete execution and synchronize.

Error termination occurs if any image hits an error condition or executes an **`error stop`** statement. All other images that have not initiated error termination do so as soon as possible.

Input/output

Default input (*) is available on image 1 only.

Default output (*) and error output are available on every image. The files are separate, but their records will be merged into a single stream or one for the output files and one for the error files.

The **open** statement connects a file to a unit on the executing image only.

Whether a file on one image is the same as a file with the same name on another image is processor dependent.

Optimization

Most of the time, the compiler can optimize as if the image is on its own, using its temporary storage such as cache, registers, etc. (even for remote data).

There is no coherency requirement while unordered segments are executing.

The programmer is required to follow the rule: 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.

The compiler also has scope to optimize communication.

Planned extensions

The following features are planned for a Technical Specification on *Additional Parallel Features in Fortran*:

- **Teams** - subdivide the images into sets that execute independently.
- **Collective intrinsic subroutines** for the operations **`CO_ADD, CO_MAX, CO_MIN, CO_REDUCE, CO_BROADCAST`**.
- **One-sided synchronization** using tagged events that may be posted, tested or waited for.
- **Failed images** – continue execution in the face of failed images.

A comparison with MPI

A colleague (Ashby, 2008) converted most of a large code, SBLI, a finite-difference formulation of Direct Numerical Simulation (DNS) of turbulance, from MPI to coarrays.

Since MPI and coarrays can be mixed, he was able to do this gradually, and he left parts of the code in MPI.

Most of the time was taken in halo exchanges and the code parallelizes well with 64 processors. The speeds were very similar.

The code clarity (and maintainability) was much improved. The code for halo exchanges, excluding comments, was reduced from 176 lines to 105 and the code to broadcast global parameters from 230 to 117.

Advantages of coarrays

- References to local data are obvious as such.
- Easy to maintain code - more concise than MPI and easy to see what is happening
- Integrated with Fortran - type checking, type conversion on assignment, ...
- The compiler can optimize communication
- Local optimizations still available
- Does not make severe demands on the compiler, e.g. for coherency.

References

Ashby, J.V. and Reid, J.K (2008). Migrating a scientific application from MPI to coarrays. CUG 2008 Proceedings. RAL-TR-2008-015, see http://www.numerical.rl.ac.uk/reports/reports.shtm

ISO/IEC (2010). Information technology - Programming languages - Fortran -Part 1: Base language. ISO/IEC 1539-1:2010(E), ISO, Geneva.

Metcalf, M., Reid, J., and Cohen, M. (2011). Modern Fortran Explained. OUP, Oxford.