Coscalars

A suggestion for supporting distributed structures in Fortran

Declaration / establishment

- coarray:
  - exists on every image
  - dynamic memory: requires program-wide synchronization

x - scalar coscalar
y - array coscalar

Corank is zero

- coscalar:
  - exists on exactly one image („host image“)

```
integer :: n[]
```

  - dynamic memory: exactly one image allocates

→ becomes the host image

```
real, allocatable :: x[],
                     y(:)[]

allocate(x[5],(y(20)[7]))
```

→ no sync, atomic semantics

(invoke ALLOCATED())

→ same image deallocates

Definition / Reference

- Explicit bracket
  - always present („statistics“)
  - **all** accesses coindexed

- Synchronization rule
  - same as for coarrays

```
integer :: n[]

if (this_image(n) == this_image()) then
    n[] = …
    sync images(*)
else
    sync images(this_image(n))
    … = n[] …
end if
```

Coscalar pointer

- add the attribute

```
real, pointer :: p[]
```

  - itself a coscalar
  - target: must be coindexed

```
real :: t1[*]
real :: t2[]

if (X) p[] => t1[5]
sync images (X)
… = p[]
sync all
if (…) p[] => t2[]
```

- expensive operations → 3 (or more) images involved
  - host image of pointer
  - image **X** executing the pointer assignment
  - host image of target

(to some extent, have this

problem also for coarrays)

- require TARGET attribute?

Distributed binary tree

```
type :: tree
  type(lock_type) :: lk
  type(content) :: entry
! entities of type content have „<“
! and possibly assignment overloaded
  logical :: defined = .false.
  type(tree), pointer :: left[] => null()
  type(tree), pointer :: right[] => null()
end type
```

- all entities must be coscalars (then %lock is)
- support a tasking-like, recursive programming style
  - lock needed for population, not read-only traversal
  - therefore preference for read operations
- allocation and deallocation remain purely local operations
- with coarrays, implementation much more clumsy

Rice „CAF 2.0“ Copointers

A more full-featured approach

Declaration / establishment

- uses the COPOINTER attribute

```
real, dimension(:), &
      copointer :: px
```

  - entity exists on one image only

- target must have the COTARGET attribute

```
real, dimension(10), &
      cotarget :: x
```

  - exists on one image only

- Copointer association

```
if (this_image() == 1) &
  px => x
```

- additional attributes like CONTIGUOUS are also allowed

- dynamic (de)allocation of anonymous cotarget (shared area, one image only) is possible

Copointers to coarrays

```
real, dimension(:), copointer :: px
real, dimension(10), cotarget :: x[*]
```

- local portion

```
px => x
```

(remote operation wrt. copointer location)

- coindexed entity

```
px => x[9]
```

Copointers to local pointers

```
real, dimension(:), pointer :: lpx

lpx => x  ! cotarget implies target (?)
…
px => lpx ! converts local pointer to copointer
```

Local casts

- remote RHS for local LHS remains disallowed

```
lpx => px[]   ! forbidden
```

- Only allowed if RHS target is local

```
if (imageof(px) == this_image()) then
  lpx => px   ! OK
end if
```

this also illustrates how target location is identified

Copointers that are coarrays

- gives you a bunch of num_images() copointers, each hosted on an image of its own

```
real, dimension(:), copointer :: pxc[*]
```

- make each of these point to a coarray

```
real, dimension(10), cotarget :: xc[*]

pxc => xc ! all images execute
```

Referencing / defining

- Requires the co-dereference operator

```
real, dimension(:), copointer :: px
real, dimension(10) :: a


if (this_image() == 1) then
  allocate(px(4))
  px(1)[] = a(5)

  px(2:)[] = a(1:3)
end if
! same for RHS
```

- Exception: target is local to the executing image

Conclusion

- Feature is for functionality, not for performance
- better support for implementation of object-based parallel software patterns
  - mostly of the kind write rarely, read often
  - e.g. load balancing algorithms
- Locks, Events may be more flexibly used