A New Vision for Coarray Fortran

John Mellor-Crummey, Laksono Adhianto
William Scherer III

Department of Computer Science
Rice University

johnmc@cs.rice.edu
Lessons from HPF

• Good parallelizations require proper partitionings
  —in inferior partitionings will fall short at scale
• Excess communication undermines scalability
  —both frequency and volume must be right!
• Must exploit what smart users know
  —allow the power user to relax consistency
• Single processor efficiency is critical
  —node code must be competitive with serial versions
  —must use caches effectively on microprocessors
Coarray Fortran (CAF)

- Explicitly-parallel extension of Fortran 95 (Numrich & Reid)
- Global address space SPMD parallel programming model
  - one-sided communication
- Simple, two-level memory model for locality management
  - local vs. remote memory
- Programmer has control over performance critical decisions
  - data partitioning
  - computation partitioning
  - communication
  - synchronization
- Suitable for mapping to a range of parallel architectures
  - shared memory, clusters, hybrid
Classic CAF Programming Model

- SPMD process images
  - fixed number of images during execution: `num_images()`
  - images operate asynchronously: `this_image()`

- Both private and shared data
  - `real x(20, 20)` a private 20x20 array in each image
  - `real y(20, 20) [*]` a shared 20x20 array in each image

- Simple one-sided shared-memory communication
  - `x(:,j:j+2) = y(:,p:p+2) [r]` copy columns from `p:p+2` into local columns

- Synchronization intrinsic functions
  - `sync_all` – a barrier and a memory fence
  - `sync_mem` – a memory fence

- Asymmetric dynamic allocation of shared data

- Weak memory consistency
Why a New Vision?

Fortran 2008 draft specification characteristics

- Coarrays must be allocated over all images
  —no support for process subsets
- Coarrays must be declared as global variables
  —no support for dynamic non-global coarrays
- No remote pointers
- No support for collective communication
- Synchronization is not expressive enough
- ... and so on ... (see our critique)
  —www.j3-fortran.org/doc/meeting/183/08-126.pdf
Outline

• Coarray Fortran 2.0
  — Process subsets: teams
  — Topologies
  — Copointers
  — Synchronization
  — Collective communication

• Summary and ongoing work
Coarray Fortran 2.0 Goals

• Facilitate construction of sophisticated parallel applications and parallel libraries
• Support irregular and adaptive applications
• Hide communication latency
• Colocate computation with remote data
• Scale to petascale architectures
• Exploit multicore processors
• Enable development of portable high-performance programs
• Interoperate with legacy models such as MPI
Process Subsets: Teams

- Teams are first-class entities
  - ordered sequences of process images
  - namespace for indexing images by rank \( r \) in team \( t \)
    - \( r \in \{0..\text{team}_\text{size}(t) - 1\} \)
  - domain for allocating coarrays
  - substrate for collective communication

- Teams need not be disjoint
  - an image may be in multiple teams
Creating New Teams

```python
team_split (existing_team, color, key, new_team,
            [new_color=result_color, err_msg=emsg_var])
```

- Images supplying the same `color` are assigned to the same team
- Each image’s rank in the new team is determined by `key` order
- `result_color ≠ color` gets handle for another team
  — used to arrange inter-team communication
  — alternative to MPI’s process groups
- `emsg_var` receives any error result message
- Predefined team: `TEAM_WORLD`
Accessing Coarrays on Teams

• Accessing a coarray relative to a team
  \[ x(i,j)[p@ocean] \]
  \( ! \) \( p \) names a rank in team ocean

• Accessing a coarray (default)
  \[ x(i,j)[p] \]
  \( ! \) \( p \) names a rank in team_world (default)

• Simplifying processor indexing using “with team”

  with team atmosphere \( ! \) make atmosphere the default team
  \( ! \) \( p \) is wrt team atmosphere, \( q \) is wrt team ocean
  \[ x(:,0)[p] = y(:)[q@ocean] \]
  end with team
Teams and Coarrays

real, allocatable :: x(:,;)[*] ! 2D array
real, allocatable :: z(:,;)[*]
team :: subset
integer :: color, rank

! each image allocates a singleton for z
allocate( z(200,200) [@team_world] )

color = floor((2*team_rank(team_world)) / team_size(team_world))

! split into two subsets:
!   top and bottom half of team_world
team_split(team_world, color, &
    team_rank(team_world), subset)

! members of the two subset teams
! independently allocate their own coarray x
allocate( x(100,n)[@ subset])

team_rank(team): returns the relative rank of the current image within a team

team_size(team): returns the number of images of a given team
Topology

• Motivation
  —a vector of images may not adequately reflect their logical communication structure
  —multiple codimensions only support grid-like logical structures
  —want a single mechanism for expressing more general structures

• Topology
  —augments a team with a logical structure for communication
  —more expressive than multiple codimensions
Using Topologies

- **Creation**
  - Graph: `topology_graph(n,e)`
  - Cartesian: `topology_cartesian(/e1,e2,.../)`

- **Modification**
  - `graph_neighbor_add(g,e,n,nv)`
  - `graph_neighbor_delete(g,e,n,nv)`

- **Binding:** `topology_bind(team,topology)`

- **Accessing coarrays using a topology**
  - **Cartesian**
    - `array(:) [ (i1, i2, ..., in)@ocean ]`! absolute index wrt team ocean
    - `array(:) [ +(i1, i2, ..., in)@ocean ]`! relative index wrt self in team ocean
    - `array(:) [ i1, i2, ..., ik ]`! wrt enclosing default team
  - **Graph:** access $k^{\text{th}}$ neighbor of image $i$ in edge class $e$
    - `array(:) [ (e,i,k)@g ]`! wrt team g
    - `array(:) [ e,i,k ]`! wrt enclosing default team
Cartesian Topology Example

Topology :: Cart
Integer, Allocatable :: X(:)[*], Y(:)[*]
Team :: Ocean, SeaSurface

! create a cartesian topology 2 (cyclic) by 3
Cart = Topology_cartesian( /-2, 3/ )

! bind Cart to teams Ocean and SeaSurface
Call Topology_bind( Ocean, Cart )
Call Topology_bind( SeaSurface, Cart )

allocate( X(100)[@SeaSurface])
allocate( Y(100)[@Ocean])

! Ocean is the default team in this scope
With Team Ocean
  Y(:) [1, 1] = X(:)[ (-1, 2)@SeaSurface ]
End With Team
Topology :: graph
graph = topology_graph( 6, 2 )
integer :: red, blue, myrank

myrank = team_rank(team_world)

read *, blue_neighbors, red_neighbors
! blue edges
call graph_neighbor_add( graph, blue, myrank, blue_neighbors )

! red edges
call graph_neighbor_add( graph, red, myrank, red_neighbors )

! bind team with the topology
call topology_bind( ocean, graph )

allocate( x(100)@ocean )
y(:) = x(20:80) [ (myrank, blue, 2)@ocean ]
Copointers

• Motivation: support linked data structures

• copointer attribute enables association with remote shared data

• imageof(x) returns the image number for x

• useful to determine whether copointer x is local

```fortran
integer, allocatable :: a(:,:,1)
integer, copointer :: x(:,:,1)
allocate(a(1:20, 1:30)[@ team_world]
!
associate copointer x with a remote section of a coarray
x => a(4:20, 2:25)[p]
!
imageof intrinsic returns the target image for x
prank = imageof(x)
!
x(7,9) = 4 ! assumes target of x is local
x(7,9)[ ] = 4 ! target of x may be remote
```
Synchronization

- **Lockset**: ordered sets of locks
  - convenient to avoid deadlock when locking/unlocking multiple locks -- uses a canonical ordering

- Point-to-point synchronization via *event variables*
  - like counting semaphores
  - each variable provides a synchronization context
  - a program can use as many events as it needs
    - user program events are distinct from library events
  - `event_notify()` / `event_wait()`
Collective Communication

- Collective operations:
  - usual suspects: broadcast, all/gather, permute, all/reduce, scan, scatter, segmented_scan, shift

- Flavors
  - traditional: two-sided synchronous
  - new modalities
    - two-sided asynchronous: all start it and later finish it
    - one-sided synchronous: one starts it and blocks until done
    - one-sided asynchronous: one starts it and later finishes it

- A new twist: all/select for min, max, max_copy, min_copy

- User-defined reduction and selection operators

- Split-phase barriers
Summary and Ongoing Work

• CAF 2.0 supports many new features
  — process subsets (teams), coarrays allocated on teams, dynamic allocation of coarrays, collectives on teams
  — topologies
  — copointers
  — events for safe pair-wise synchronization
  — locksets

• Provides expressiveness, simplicity and orthogonality

• Source-to-source translator is a work in progress
  — requires no vendor buy-in
  — will deliver node performance of mature vendor compilers

• Coming attractions:
  — cofunctions: remote procedure calls for latency avoidance
  — coarray binding interface for inter-team communication