Coarray Fortran (CAF) 2.0

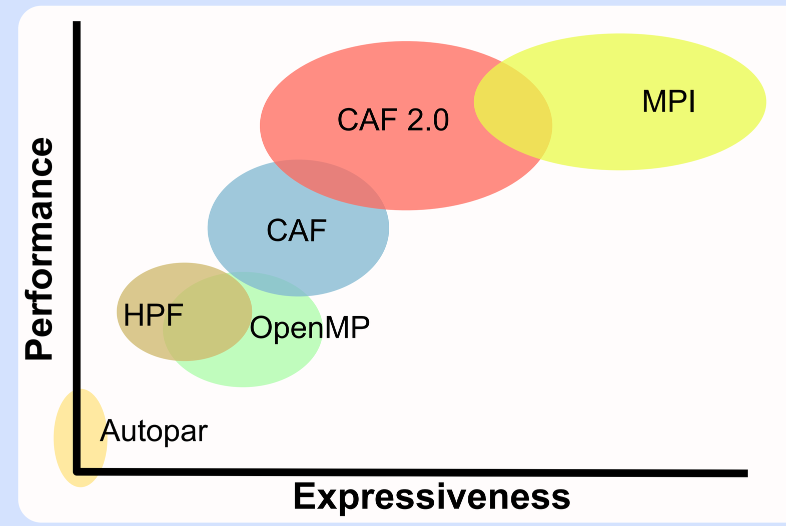
Department of Computer Science, Rice University, Houston, TX

http://caf.rice.edu

Objectives

Expressiveness	Support irregular and adaptive applications; support construction of sophisticated parallel applications and parallel libraries
Scalability	Scale to petascale architectures
Orthogonality	Provide a powerful model in the form of a small set of composable features
Multithreading	Exploit multicore processors
Performance	Deliver top performance: enable users to avoid exposing or overlap communication latency
Portability	Support development of portable high performance programs
Interoperability	Interoperate with legacy parallel computing models such as MPI and OpenMP

Coarray Fortran (CAF) 2.0 supports higher expressiveness than CAF features in Fortran 2008, with performance comparable to MPI



Key Features

Participation

`team`: ordered sequence of process images
- Create arbitrary subsets of any team as necessary
- Support coupled applications with multiple teams (e.g. separate teams for ocean and atmosphere)
- Allow multiple overlapping views (e.g., row and column teams overlaid on a grid of images)
- Index images in a team using team-relative rank r ∈ {0..team_size(t) - 1} with team t

Accessing a coarray from a specific team:
- `b(1:100)[1]`: accessing elements in coarray **b** of image 1 within the current default team
- `b(1:100)[1@a_team]`: accessing elements in coarray **b** of image 1 within team *a_team*

Some team intrinsics and statements:
- `team_world`: a predefined team that consists of all process images (analogous to `MPI_COMM_WORLD`)
- `team_default`: the team in the current scope (by default is `team_world`)
- `team_rank([a_team])`: returns the team-relative rank of a given image process (`team_default` if not specified)
- `team_size([a_team])`: returns the number of images in a given team (`team_default` if not specified)
- `team_split(parent_team, color, key, new_team)`: forming a new team from an existing one
- `with team a_team ... end with team [a_team]`: changing the default team to *a_team* within its scope

Organization

`Topology`:
- Augments a team with a logical structure for communication
- More expressive than multiple codimensions
- Support for cartesian and graph virtual topologies

Creation:
- Cartesian: `topology_cartesian(/e1,e2, .../)` *! e_i are the sizes in the i-th dimension*
- Graph: `topology_graph(n, e)` *! n is the number of nodes, e is the number of edge classes*

Modification (graph topology only):
- `graph_neighbor_add(g, e, n, nv)`
- `graph_neighbor_delete(g, e, n, nv)`

Binding:
- `topology_bind(team, topology)`

Accessing coarrays using a cartesian topology:
- `array(:) [(i1, i2, ..., in)@ocean]` *! **absolute** index w.r.t. team* **ocean**
- `array(:) [+(i1, i2, ..., in)@ocean]` *! **relative** index w.r.t. self in team* **ocean**
- `array(:) [i1, i2, ..., in]` *! w.r.t. enclosing default team*

Accessing **k**th neighbor of image **i** in edge class **e** using a graph topology
- `array(:) [(e,i,k)@ocean]` *! w.r.t. team* **ocean**
- `array(:) [e,i,k]` *! w.r.t. enclosing default team*

Mutual exclusion

`lock`: support fine grain mutual exclusion
- `lock_acquire(l)` *! acquire lock* **l**
- `lock_release(l)` *! release lock* **l**

`lockset`: a set of locks help avoid deadlock when acquiring multiple locks by transparently acquiring them in an appropriate order
- `lockset_acquire(ls)` *! acquire lockset* **ls**
- `lockset_release(ls)` *! release lockset* **ls**

`critical([lock])`: a structured construct for mutual exclusion
- `critical(l) ... end critical` *! block protected by lock* **l**

Coordination

`event`: synchronization object for anonymous pairwise coordination
- Safe synchronization space: can allocate as many events as desired
- `event_init`: event initialization
- `event_notify`: a non-blocking signal to an event; serves as a pairwise fence between the sender and target image
- `event_wait`: blocking wait for notification on an event
- `event_trywait`: non-blocking wait for notification on an event
- `event_getid`: retrieve an event ID

`eventset`: multi-events synchronization
- Set manipulation: `eventset_init`, `eventset_add`, `eventset_addarray`, `eventset_remove`, `eventset_destroy`
- Events manipulation: `eventset_waitany`, `eventset_waitany_fair`, `eventset_waitall`, `eventset_notifyall`

Collective operations

Support development of portable high performance programs synchronization and communication among a team of images

Two-sided collectives
- Each process image in a team calls the collective operation
- The two-sided style enables each process image to specify where the result will be received

All-to-one communication:
- `team_reduce`, `team_gather`

One-to-all communication:
- `team_broadcast`, `team_scatter`

All-to-all communication:
- `team_allreduce`, `team_allgather`, `team_alltoall`, `team_barrier`, `team_sort`, `team_scan`, `team_shift`

Asynchrony

Predicated asynchronous copy: optionally wait for an event before starting the copy; optionally post an event upon completion
- `copy_async(var_dest, var_src [, event_after] [, event_before])`

Two-sided asynchronous collective operations: two-sided design facilitates flow control
- `team_barrier_async`, `team_broadcast_async`, `team_gather_async`, `team_allgather_async`, `team_reduce_async`, `team_allreduce_async`, `team_alltoall_async`

Multithreading & function shipping

`spawn`: create local or remote asynchronous threads by calling a procedure
- Local threads can exploit multicore parallelism
- Remote threads can be created to avoid latency when manipulating remote data structures

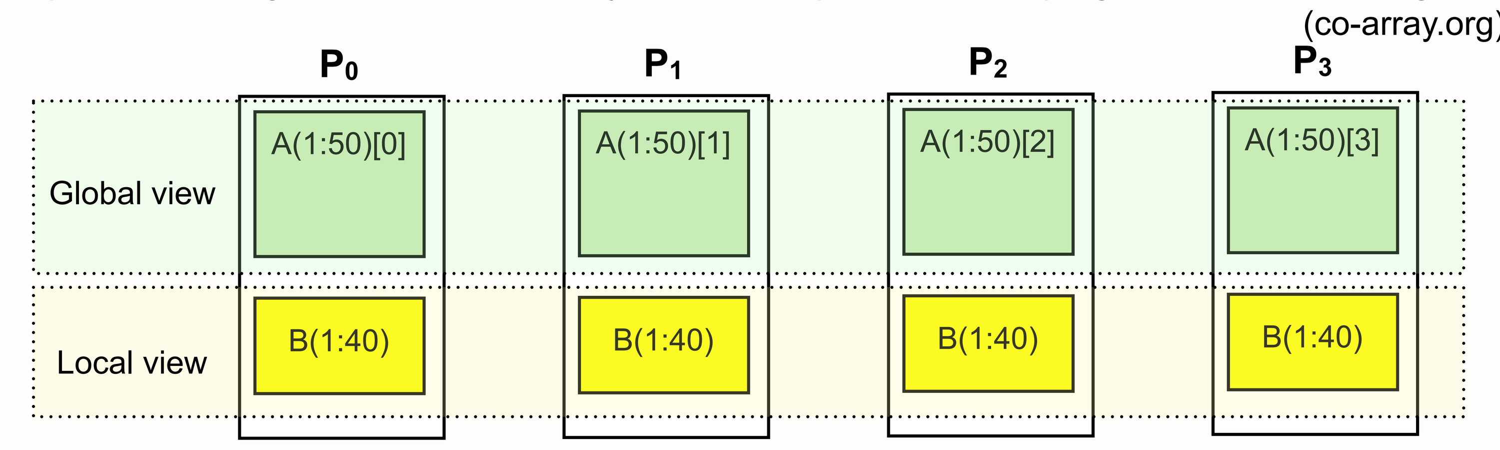
`finish [t]`: terminally strict synchronization for (nested) threads spawned across team t (or the default team)
- Orthogonal to procedures (like X10 and unlike Cilk)

Remote pointers

`copointer`: an attribute to associate with shared data that may be remote
- Support for remote manipulation of data structures
- `imageof`: get the target image for a copointer

Memory view

"First, consider work distribution. A single program is replicated a fixed number of times, each replication having its own set of data objects. Each replication of the program is called an image" (co-array.org)




```
integer :: A(1:50)[*] ! declare coarray A , which is accessible by all process images
integer :: B(1:40)    ! declare a local array B, which is inaccessible to other process images
```

Array allocation in CAF 2.0:

```
integer, allocatable :: C(:)[*], D(:)[*]      ! declare allocatable coarrays
allocate(C(1:100)[@ocean])       ! allocate a 100-element coarray C within team ocean
allocate(D(1:100)[])             ! allocate a 100-element coarray D within the default team
...
C[1@ocean] = D[2]                ! copy coarrray D from image 2 within the default team to
                                 ! coarray  C from image 1 within team ocean
```

Memory model

By default, CAF 2.0 programs are sequentially consistent. One may obtain relaxed semantics for a section of code by marking it with '`!$caf consistency(relaxed=on/off)`'

Principles guiding the CAF 2.0 memory model:

Sequential consistency is provided by default so that it is easy to reason about possible executions
- '*Delay Set Analysis*' [Shasha, Snir TOPLAS88] can make sequential consistency cheaper at runtime

In sections of code marked for relaxed consistency:
- No program order guarantee between coarray reads and writes
- Ordering can be enforced via '`cofence`' or synchronization primitives

`cofence(allow_downward=PUT|GET, allow_upward=PUT|GET)`, based on SPARC V9 MEMBAR:
- Acts as a memory barrier for synchronous coarray operations, except as relaxed by arguments
- Acts as a release barrier for implicit asynchronous operations, guaranteeing their completion
- Provides no guarantee that explicit asynchronous operations have completed

`event/eventset` notify/wait and `copy_async` operations always act as release barriers

Example 1: Team and Coarray allocation

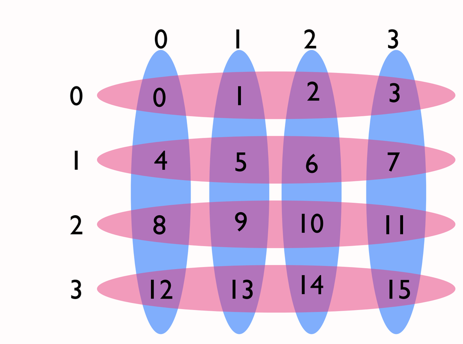
```
team :: row_team, col_team

rank = team_rank() ! get the relative rank
size = team_size() ! get the number of images
p = rank / 4       ! determine row position
q = mod( rank, 4 ) ! determine column position

! split into rows and columns
team_split(team_world, p, rank, row_team)
team_split(team_world, q, rank, col_team)

allocate(rowdata(1000000)[@row_team])   ! allocate across process images within my row_team
allocate(coldata(1000000)[@col_team])   ! allocate across process images within my col_team

with team row_team                      ! row_team is the default team
  ...
  rank_row = team_rank()                ! get the relative rank within row_team
  size_row = team_size()                ! get the number of images within row_team
  buffer   = rowdata[mod(rank_row-1, size_row)]  ! get the data from the "left"
end with team
```



Example 2: Function shipping

```
subroutine update_table(table, index, value)
  integer :: table(:)[*]
  ! update local table
  table(index) = value
end subroutine

subroutine apply_updates(table, buffer)
  integer :: buffer(:), table(:)[*]
  finish
    do i=1,size
      buffer(i) = ...
      ! ask remote process to update an element in its table with a given value
      spawn update_table(table, index, buffer(i))[remote_proc]
    enddo
  end finish
end subroutine
```

The spawn shown below is semantically equivalent to the copy_async shown below:
`copy_async(table(index)[remote_proc],buffer(i))`

Contributors

- John Mellor-Crummey (PI)
- Laksono Adhianto
- Guohua Jin
- Karthik Murthy
- Dung Nguyen
- Mark Krentel
- William N. Scherer III
- Scott K. Warren
- Chaoran Yang

SciDAC Scientific Discovery through Advanced Computing

Development of CAF 2.0 is supported by the Department of Energy's Office of Science under cooperative agreements DE-FC02-07ER25800 and DE-FC02- 06ER25754

RICE