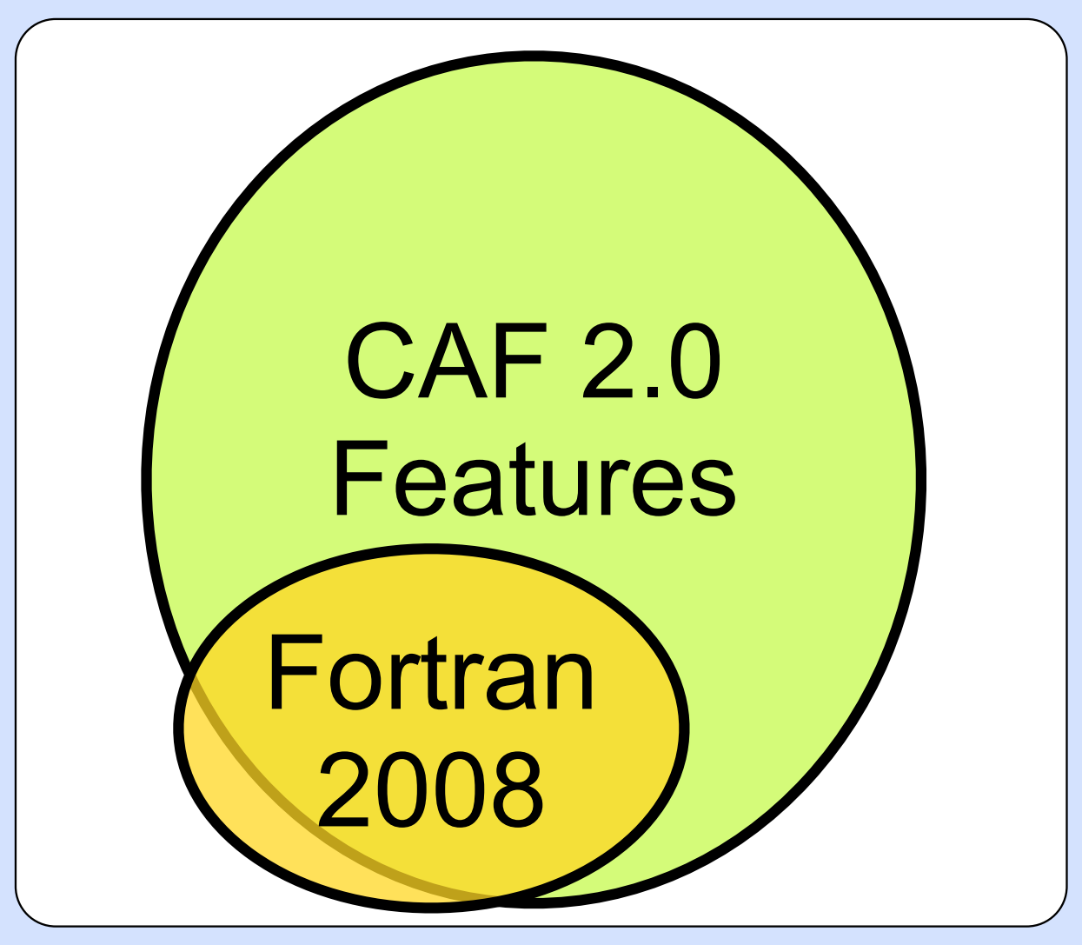
Coarray Fortran (CAF) 2.0

Department of Computer Science, Rice University, Houston, TX
Project URL: 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 and beyond
Orthogonality	Support complex concepts with minimal language extensions
Multithreading	Exploit multicore and multi-threaded processors
Performance	Deliver top performance: enable users to overlap communication latency with computation
Portability	Support development of portable high performance programs
Interoperability	Interoperate with legacy parallel computing models such as MPI, OpenMP, and CUDA

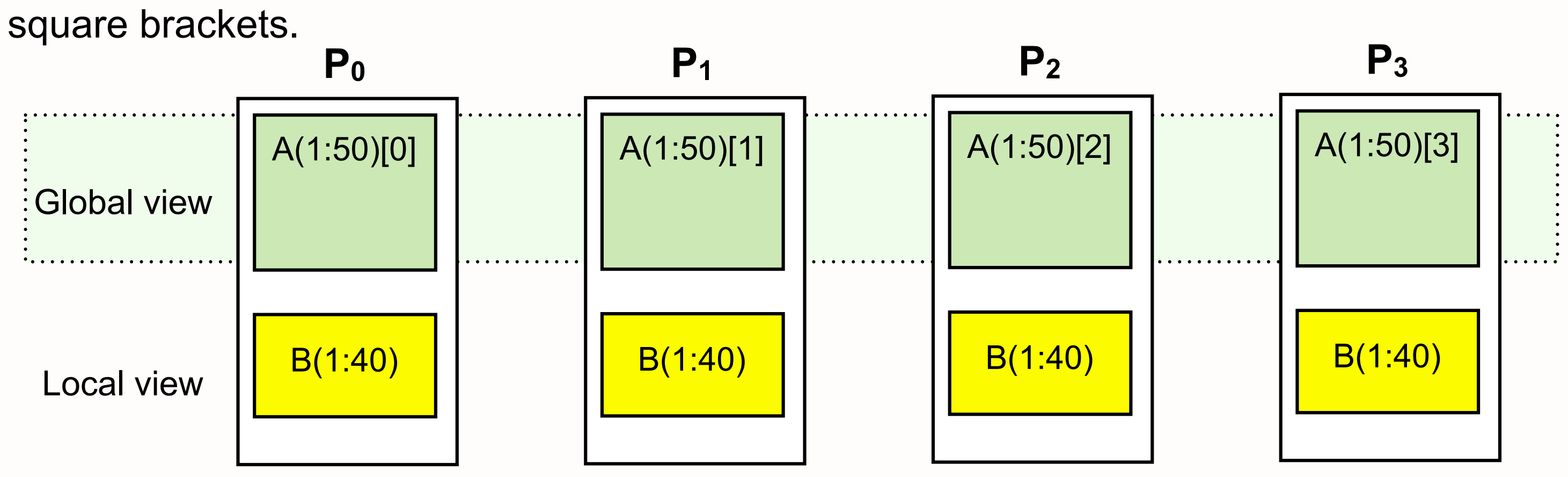
CAF 2.0 offers greater expressiveness than the coarray features in Fortran 2008 yet it still yields performance comparable to that of MPI



Key Features

Partitioned Global Address Space (PGAS) memory view

Like Unified Parallel C (UPC) and Chapel, CAF 2.0 features a two-level partitioned view of memory in which data is either local or remote. Unlike them, however, accesses that may touch remote memory are always explicitly flagged with square brackets.

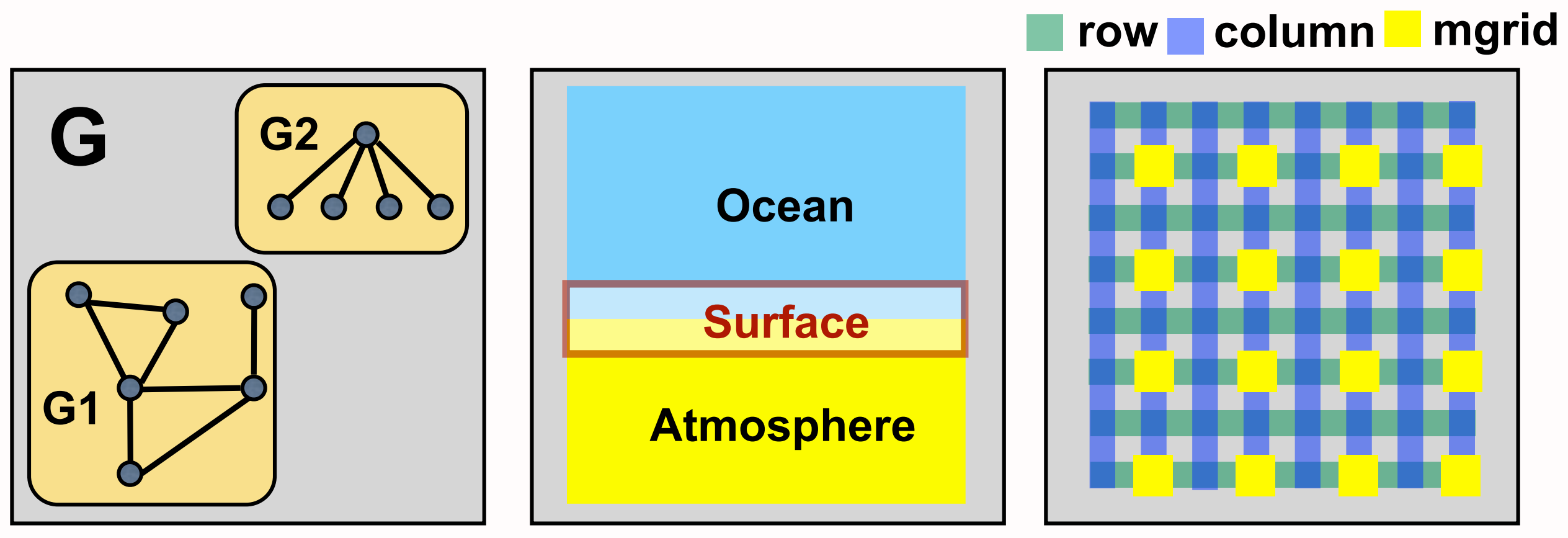


```
integer :: A(1:50)[*]   ! declares coarray A accessible to all image processes
integer :: B(1:40)      ! declares local array B, not accessible to other image process
```

Array allocation in CAF 2.0:

```
 integer, allocatable :: C(:)[*]       ! declares an allocatable coarray
 allocate(C(1:100)[@some_team])        ! allocates coarray C in members of some_team
 allocate(D(1:100)[])                  ! allocates coarray C in members of the default team
```

Process subsets: Teams



`team`: ordered sequence of process images
- Dynamically create arbitrary subsets of any team
- 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 $r \in [0..\mathrm{team\_size}(t_0) - 1]$ with team t_0

Accessing a coarray from a specific team:
- `b(1:100)[1]`: accesses elements in coarray `b` on image 1 of the current default team.
- `b(1:100)[1@myteam]`: accesses elements in coarray `b` on image 1 on team *`myteam`*.

Team intrinsics and statements:
- `team_world`: predefined team that consists of all images (equivalent to `MPI_COMM_WORLD`).
- `team_default`: the default team for the current scope (initially `team_world`).
- `team_rank(myteam)`: returns the team-relative rank of a given image process.
- `team_size(myteam)`: returns the number of images in a given team.
- `team_split(parent_team, color, key, new_team)`: forms new teams as subsets of an existing one; equivalent to `MPI_COMM_SPLIT`.
- `with team myteam … end with team myteam`: sets the default team to *`myteam`* within its scope.

A rich set of collective operations

Portable, high performance synchronization and communication among images within a team

Two-sided design for collectives:
- Requires only *O(1)* memory where one-sided would require *O(p)* with *p* participating processes.
- Receivers can manage flow control by specifying their willingness to participate.

All-to-one communication: all processes contribute to the result, but only one process receives it
- `team_reduce`
- `team_gather`

One-to-all communication: one process contributes the result; all processes receive it
- `team_broadcast`
- `team_scatter`

All-to-all communication: all processes contribute to the result; all processes receive it
- `team_allreduce`, `team_allgather`, `team_alltoall`, `team_alltoallv`, `team_barrier`
- `team_sort`, `team_scan`, `team_shift`

Other CAF 2.0 features

- **Topologies** (cartesian, graph)
- **Multithreading**: Fortran 2008 `do concurrent` statement implemented via work stealing
- **Function shipping**: synchronous invocation of remote functions using a `spawn` statement
- **Synchronization**: block-structured `finish … end finish` construct as in X10
- **Mutual exclusion**: locks, critical sections, and locksets
- **Memory consistency**: `cofence` for local completion of operations and asynchronous events

Events for point-to-point synchronization and asynchrony

`event`: synchronization object for anonymous pairwise coordination
- Safe synchronization space: can allocate as many events as desired
- `event_init`: event initialization
- `event_notify`: nonblocking signal to an event; a pairwise fence between sender and target image
- `event_wait`: blocking wait for notification of an event
- `event_trywait`: nonblocking check to see if an event has been signaled

Asynchrony support
- Completion of asynchronous operations managed two ways:
  - ➡ Explicit model: notify an event upon completion
  - ➡ Implicit model: both `cofence` and `finish` block “round up” outstanding operations
- Asynchronous collectives signal completion either explicitly or implicitly

- Predicated asynchronous copy overlaps computation and communication
  - ➡ `copy_async(dest, src, cr, sr, dr)`
  - ➡ `cr` (copy ready; optional): an event indicating that the data may now be copied from src to dest
  - ➡ `sr` (source ready; optional): an event indicating that the source data may be safely overwritten
  - ➡ `dr` (destination ready; omitted to use implicit asynchrony): an event indicating that the copy has completed

Global pointers: copointer and cotarget

- `copointer`: support irregular data decompositions, distributed linked data structures, parallel model coupling
- `cotarget`: marks entities that may be targeted by a copointer.
- `=>`: same symbol is used for pointer and copointer assignment

```
 integer, dimension(:), allocatable, cotarget :: A[*]
 integer, dimension(:), copointer :: p, q

 p => A          ! copointer p points to coarray A
 q => A[2]       ! copointer q points to portion of coarray A on image 2
 p(5) = 42       ! assign to local data using copointer p
 q(5)[] = 42     ! assign to remote data on image 2 using copointer q
 p => q          ! reassign copointer p with a copy of copointer q
```

CAF 2.0 halo exchange in the Parallel Ocean Program (POP)

Initialize copointers

```
do n = 1, size(boundary%out)
 face => boundary%out(n)
 p = face%partner
 face%local_ptr => A(
   face%src_bounds(1,1): face%src_bounds(2,1),&      1
   face%src_bounds(1,2): face%src_bounds(2,2),&
   face%src_block_id)
 face%remote_ptr => A(
   face%dest_bounds(1,1): face%dest_bounds(2,1),&    2
   face%dest_bounds(1,2): face%dest_bounds(2,2),&
   face%dest_block_id)[p]
 end do
```

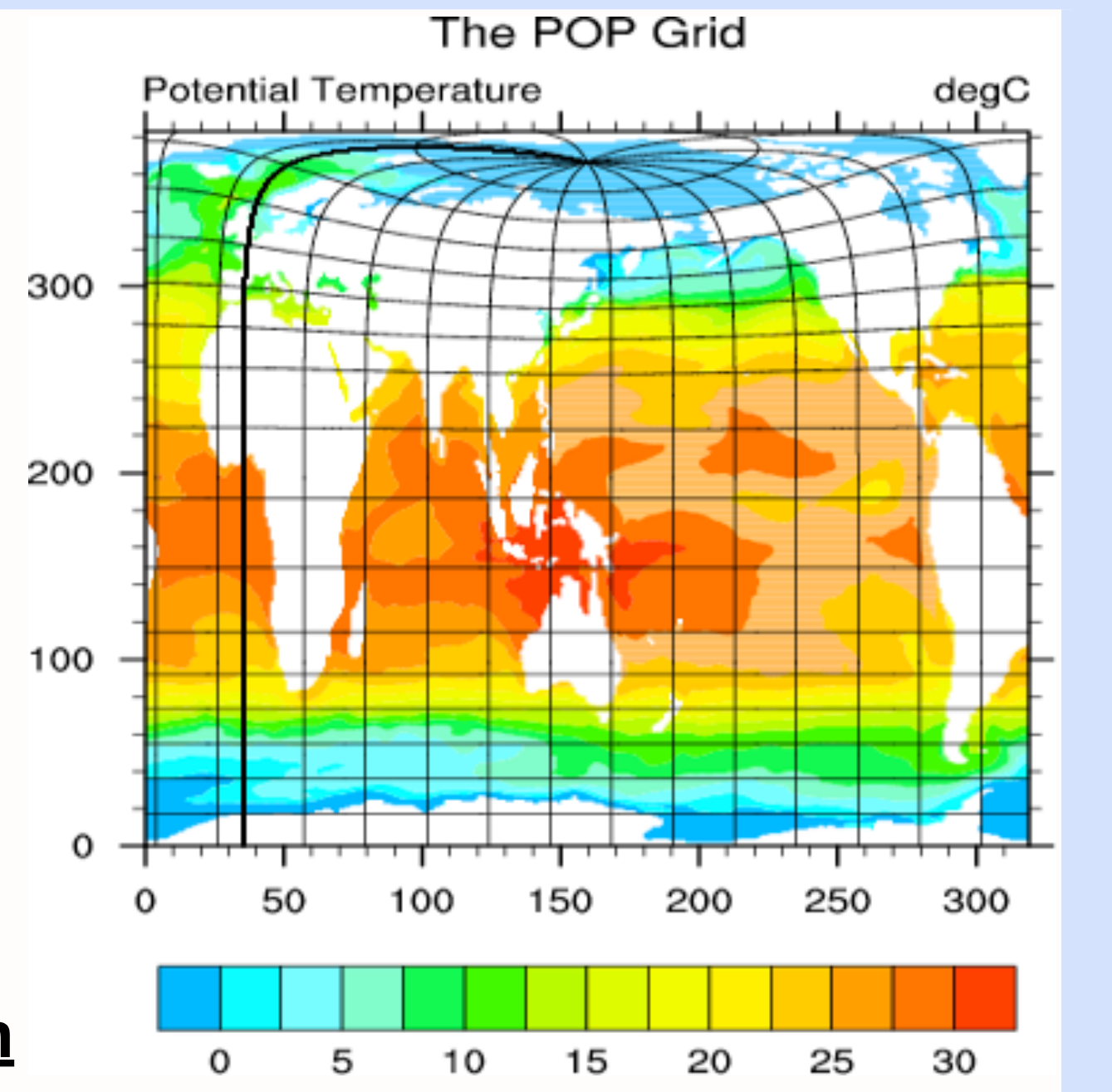


Image Credit: UCAR

Replace block-synchronous updates with one-sided communication

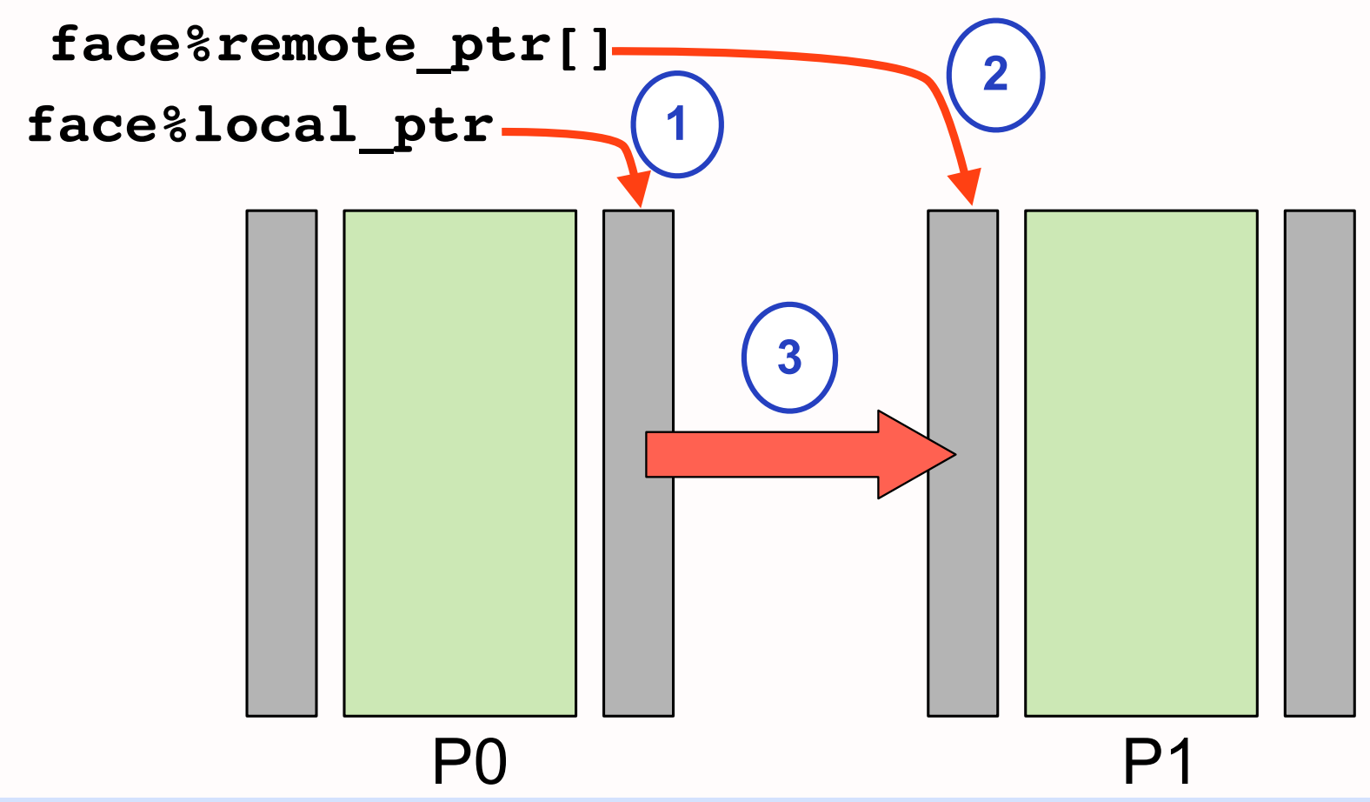
```
 !  notify my partners that my block is ready by posting an event for each ready face
 do n = 1, size(boundary%in)
  event_notify(boundary%in(n)%event_dest_ready[])
 end do

 !  for each face, initiate a data copy when the destination is ready and signal when complete
 do n = 1, size(boundary%out)
  face => boundary%out(n)
3 copy_async(face%remote_ptr[], face%local_ptr, face%event_dest_ready &
             face%event_src_done, face%event_dest_done[])
 end do

 !  wait for local completion of copies initiated locally
 do n = 1, size(boundary%in)
  event_wait(boundary%out(n)%event_src_done)
 end do

 !  perform  local updates
 …

 !  wait for all incoming faces to arrive from partners
 do n = 1, size(boundary%in)
  event_wait(boundary%in(n)%event_dest_done)
 end do
```



Contributors

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

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.

The Rice CAF 2.0 compiler uses the Rose compiler (Lawrence Livermore National Lab), the Open Fortran Parser (Los Alamos National Lab) and the GASNet communications library (University of California Berkeley).

SciDAC
Scientific Discovery through Advanced Computing

RICE

Friday, November 4, 2011