Coarray Fortran 2.0: A Productive Language for Scalable Scientific Computing

John Mellor-Crummey
Department of Computer Science
Rice University

johnmc@rice.edu
CAF 2.0 Project Team

- Laksono Adhianto
- Guohua Jin
- Mark Krentel
- Karthik Murthy
- Dung Nguyen
- William Scherer
- Scott Warren
- Chaoran Yang
Outline

• Coarray Fortran
  —original 1998 version
  —Fortran 2008 - a standard with coarrays

• Coarray Fortran 2.0 (CAF 2.0)
  —features
  —experiences - HPC challenge benchmarks + performance
  —implementation notes

• Status and plans
Partitioned Global Address Space Languages

- Global address space
  - one-sided communication (GET/PUT)
  - Programmer has control over performance-critical factors
    - data distribution and locality control
    - computation partitioning
    - communication placement
  - Data movement and synchronization as language primitives
    - amenable to compiler-based communication optimization

Examples: UPC, Titanium, Chapel, X10, Coarray Fortran
Coarray Fortran (CAF)

- Explicitly-parallel extension of Fortran 95 (Numrich & Reid 1998)
- 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 shared and distributed memory systems
Coarray Fortran (1998)

- **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

- **Coarrays with multiple codimensions**
  - `real y(20, 20) [4,*]`

- **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_team(notify, wait)`
    - notify = a vector of process ids to signal
    - wait = a vector of process ids to wait for
  - `sync_memory` – a memory fence
  - `start_critical/end_critical`

- **Asymmetric dynamic allocation of shared data**

- **Weak memory consistency**
integer a(10,20)[*]

me = this_image()

if (me > 1) a(1:5,1:10) = a(1:5,1:10)[me-1]
A CAF Finite Element Example (Numrich)

subroutine assemble(start, prin, ghost, neib, x)
    integer :: start(:), prin(:), ghost(:), neib(:), k1, k2, p
    real :: x(:) [*]
    call sync_team(neib)
do p = 1, size(neib) ! Add contributions from ghost regions
    k1 = start(p); k2 = start(p+1)-1
    x(prin(k1:k2)) = x(prin(k1:k2)) + x(ghost(k1:k2)) [neib(p)]
endo
call sync_team(neib)
do p = 1, size(neib) ! Update the ghosts
    k1 = start(p); k2 = start(p+1)-1
    x(ghost(k1:k2)) [neib(p)] = x(prin(k1:k2))
endo
call sync_all
end subroutine assemble
• 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

• Coarrays with multiple codimensions
  – real y(20, 20) [4,*]

• 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, sync images(image vector)
  – sync memory
  – critical sections, locks
  – atomic_define, atomic_ref

• Asymmetric dynamic allocation of shared data
• Weak memory consistency
CAF on Hopper in 2011

GTS Particle Shifter (LBNL, Cray, PPPL) [SC11]
Preissl, Wichmann, Long, Shalf, Ethier, Koniges
GTS Particle Shifter in MPI

!(1) Prepost receive requests
do i=1,nr_dests
    MPIIRECV(recv_buf(i),i,req(i),tor_comm,..)
enddo

!(2) compute shifted particles and fill buffer
$omp parallel
pack(p_array,shift,holes,send_buf)
$omp end parallel

!(3) Send of particles to destination process
do j=1,nr_dests
    MPIISEND(send_buf(j),j,req(j+i),tor_comm,..)
enddo
MPIWAITALL(2*nr_dests,req,..)

!(4) fill holes with received particles
$omp parallel do
do m=1,min(recv_length,shift)
    p_array(holes(m))=recv_buf(src,cnt)
    if(cnt.eq.recv_buf(src,0)) {cnt=1; src++}
enddo

!(5) append remaining particles or fill holes
if(recv_length < shift) {
    append_particles(p_array,recv_buf)
} else {
    fill_remaining_holes(p_array,holes)
}
GTS Particle Shifter in CAF

1. Compute shifted particles and fill the receiving queues on destination images.
2. Shift remaining particles.
3. Sync with images from same toroidal domain.
4. Fill holes with received particles.
5. Append remaining particles or fill holes.

# Code Snippet

```fortran
!$(omp parallel do schedule(dynamic,p_size/100))$
!$omp private(s_buf,buf_cnt) shared(recvQ,q_it)$
do i=1,p_size
dest=compute_destination(p_array(i))
if(dest.ne.local_toroidal_domain) {
    holes(shift++)=i
    s_buf(dest,buf_cnt(dest)++)=p_array(i)
    if(buf_cnt(dest),eq, sb_size) {
        q_start=afadd(q_it[dest], sb_size)
        recvQ(q_start:q_start+sb_size-1)[dest] &
        =s_buf(dest,1: sb_size)
        buf_cnt(dest)=0
    }
}
enddo

!$(omp parallel do)$
empty_s_buffers(s_buf)
$omp end parallel$

sync images([my_shift_neighbors])

fill holes with received particles
length_recvQ=q_it-1
$omp parallel do$
do m=1,min(length_recvQ, shift)
p_array(holes(m))=recvQ(m)
enddo

append remaining particles or fill holes
if(length_recvQ–min(length_recvQ, shift),gt,0) {
    append_particles(p_array,recvQ)
}
else { fill_remaining_holes(p_array, holes) }
```
GTC Particle Shifter Performance

(a) 1 OpenMP thread per instance

(b) 6 OpenMP threads per instance
GTS Weak Scaling Performance

Figure 8: Weak scaling GTS experiments with CAF-atom & MPI-ms as particle shift algorithms (6 OpenMP threads per instance)

52% speedup
Why a New Vision?

Fortran 2008 characteristics

- No support for process subsets
- No support for collective communication
- No support for latency hiding or avoidance
  —rendezvous synchronization: sync all, sync images
- No remote pointers for manipulating remote linked data structures
- ... and so on ... (see our critique)
  —www.j3-fortran.org/doc/meeting/183/08-126.pdf
Coarray Fortran 2.0 Goals

• Exploit multicore processors
• Enable development of portable high-performance programs
• Interoperate with legacy models such as MPI
• Facilitate construction of sophisticated parallel applications and parallel libraries
• Support irregular and adaptive applications
• Hide communication latency
• Colocate computation with remote data
• Scale to leadership computing facilities
Coarray Fortran 2.0 (CAF 2.0)

- Teams: process subsets, like MPI communicators
  - formation using team_split (like MPI_Comm_split)
  - collective communication

- Topologies

- Coarrays: shared data allocated across processor subsets
  - declaration: double precision :: a(:,;)[*]
  - dynamic allocation: allocate( a(n,m)[@row_team] )
  - access: x(:,n+1) = x(:,0)[mod(team_rank()+1, team_size())]

- Latency tolerance
  - hide: asynchronous copy, asynchronous collectives
  - avoid: function shipping

- Synchronization
  - event variables: point-to-point sync; async completion
  - finish: SPMD construct inspired by X10

- Copointers: pointers to remote data
Process Subsets: Teams

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

• Teams need not be disjoint
  — an image may be in multiple teams
Teams and Operations

• Predefined teams
  — team_world
  — team_default
    – used for any coarray operation that lacks an explicit team specification

• Operations on teams
  — 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
  — team_split (existing_team, color, key, new_team)
    – images supplying the same color are assigned to the same team
    – each image’s rank in the new team is determined by lexicographic order of
      (key, parent team rank)
Teams and Coarrays

- Coarray allocation occurs over teams
  - storage is allocated over each member of the specified team

- Example
  - integer :: a(:, :)[*]
  - allocate (a (10, 100)[@team_world])

- Allocation is a collective operation
  - need barrier after an allocation to know that a coarray is available on other team members before accessing their data
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] )
Accessing Coarrays on Teams

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

- Accessing a coarray relative to the default team
  \[ x(i,j)[p] ! p \text{ names a rank in team_default} \]
  \[ x(i,j)[p@team_default] ! p \text{ names a rank in team_default} \]

- Simplifying processor indexing using “with team”
  ```
  with team atmosphere ! set team_default to atmosphere within
  ! p is wrt team atmosphere, q is wrt team ocean
  x(:,:,0)[p] = y(:,:,q@ocean]
  end with team
  ```
Communication Topologies

• 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
  —shamelessly patterned after MPI Topologies
  —logical structure for communication within a team
  —more expressive than multiple codimensions
Using Topologies

• Creation
  — Cartesian: `topology_cartesian((/e1,e2,.../), (/ w1, w2, ... /))`
  — Graph: `topology_graph(e)`
  \[ \text{graph_neighbor_add}(g,e,n,nv) \]
  \[ \text{graph_neighbor_delete}(g,e,n,nv) \]

• Binding: `topology_bind(team,topology)`

• Accessing a coarray using a topology
  — Cartesian
    \[ \text{array}(:) [ +(i1, i2, \ldots, in)@ocean ] ! \text{relative} \text{ index wrt self in team ocean} \]
    \[ \text{array}(:) [ (i1, i2, \ldots, in)@ocean ] ! \text{absolute} \text{ index wrt team ocean} \]
    \[ \text{array}(:) [ i1, i2, \ldots, ik ] ! \text{wrt enclosing default team} \]
  — Graph: access \( k^{th} \) neighbor of image \( i \) in edge class \( e \)
    \[ \text{array}(:) [ (e,i,k)@g ] ! \text{wrt team } g \]
    \[ \text{array}(:) [ e,i,k ] ! \text{wrt enclosing default team} \]
Synchronization

• 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()
  —event_notify is non-blocking

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

• **Hide** latency for accessing remote data by overlapping it with computation

• **Avoid** exposed latency when manipulating remote data structures

• Asynchrony models
  — explicit: signal an event to indicate when an asynchronous operation has completed
  — implicit: programmer specifies a point when program must block until outstanding asynchronous operations have completed
  — interactions between models are subtle!
Predicated Asynchronous Copy

copy_async(var_dest, var_src [, ev_dest] [, ev_src] [, ev_pred])

– var_dest: data target
– var_src: data source
– ev_src: event to be triggered when the read of var_src is complete
– ev_dest: event to be triggered when the write of var_dest is complete
– ev_pred: optional event indicating that var_src is ready
Collective Communication

• Why provide collectives?
  — application programmers want them
  — avoid having programmers roll their own (non scalable) versions

• Collective operations
  — alltoall, barrier, broadcast, all/gather, permute, all/reduce, scatter, segmented/scan, shift

• User-defined reduction operators

• Potential flavors
  — two-sided synchronous
    – all execute it together
  — two-sided asynchronous
    – all team members will execute a call to start it
    – all will later wait for it to complete
  — one-sided synchronous: one starts it and blocks until done
  — one-sided asynchronous: one starts it and later finishes it
Two-sided vs. One-sided Collectives

• Issues with one-sided collectives
  — where does the data get delivered?
    – does the initiator specify an address for each recipient?
    – does data get delivered to the same offset in a coarray for each recipient?
  — how do I know when I can overwrite it?

• Two-sided collectives address these issues
  — each participant receiving a value specifies where to deliver it
  — each participant can decide how many asynchronous collectives can be outstanding at once
    – based on the number of buffers available for receiving values
  — an asynchronous collective initiated before some recipients are ready will have (at least part of) its execution deferred until recipients are ready

Coarray Fortran 2.0 supports two-sided synchronous and asynchronous collectives
Asynchronous Collective Operations

- Synchronization:
  - `team_barrier_async([event] [, team])`

- Communication:
  - `team_broadcast_async(var, root [, event] [, team])`
  - `team_gather_async(var_src, var_dest, root [, event] [, team])`
  - `team_allgather_async(var_src, var_dest [, event] [, team])`
  - `team_reduce_async(var_src, var_dest, root, operator [, event] [, team])`
  - `team_allreduce_async(var_src, var_dest, operator [, event] [, team])`
  - `team_scatter_async(var_src, var_dest, root [, event] [, team])`
  - `team_alltoall_async(var_src, var_dest [, event] [, team])`
  - `team_sort_async(var_src, var_dest, comparison_fn [, event] [, team])`
  - ...

Function Shipping

- Reduce communication overhead by moving computation to the data instead of moving data to computation
- Implicit asynchrony

```plaintext
finish (team)
  spawn fxn(table(i,j)[p], n)[p]
  ...
end finish
```
CAF 2.0 Finish

• X10 finish
  finish {
    ...
  }
  —synchronization model
    – Cilk: fully strict - all spawned children reports directly to their parent
    – X10: terminally strict
      all asyncs report to an enclosing finish scope
      the enclosing finish scope may be in a different procedure

• CAF 2.0 finish
  —SPMD construct defined over teams
    finish (team)
    ...
    end finish
  —all members of a team enter a finish block
  —any functions that team members ship to one another from within a finish block must complete before any node will exit the corresponding finish block
CAF 2.0 Cofence

• Finish is a heavyweight mechanism
  —manages global completion across a team
  —sometimes only local completion is needed
    – e.g. an asynchronous copy has delivered a value locally

• Cofence manages local completion
  —asynchronous copies with implicit completion
  —asynchronous collectives with implicit completion

• Can use a cofence within a finish block to demand early completion of asynchronous operations
Copointers: Global Pointers

- 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

```
integer, allocatable :: a(:,:, :)[*]
integer, copointer :: x(:,:, :)[*]

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
```
LANL’s Parallel Ocean Program

- Data partitioning of ocean blocks
  - cartesian, balanced, space-filling curve distributions
- Data communication
  - boundary updates between neighboring processors
  - collective communications (gather, scatter, reduction)
- Different boundary types
  - cyclic, closed, tripole
post a receive
do n=1,in_bndy%nmsg_ew_rcv
  bufsize = ny_block*nghost*in_bndy%nblocks_ew_rcv(n)
call MPI_Irecv(buf_ew_rcv(1,1,1,n), bufsize, mpi_dbl, &
     in_bndy%ew_rcv_proc(n)-1, &
     mpitag_bndy_2d + in_bndy%ew_rcv_proc(n), &
     in_bndy%communicator, rcv_request(n), ierr)
end do

pack data and send data
do n=1,in_bndy%nmsg_ew_snd
  bufsize = ny_block*nghost*in_bndy%nblocks_ew_snd(n)
  partner = in_bndy%ew_snd_proc(n)-1
  do i=1,in_bndy%nblocks_ew_snd(n)
    ib_src = in_bndy%ew_src_add(1,i,n)
    ie_src = ib_src + nghost - 1
    src_block = in_bndy%ew_src_block(i,n)
    buf_ew_snd(:,:,i,n) = ARRAY(ib_src:ie_src,:,src_block)
  end do
  call MPI_Isend(buf_ew_snd(1,1,1,n), bufsize, mpi_dbl, &
     in_bndy%ew_snd_proc(n)-1, &
     mpitag_bndy_2d + my_task + 1, &
     in_bndy%communicator, snd_request(n), ierr)
end do

local updates
! wait to receive data and unpack data
call MPI_WaitAll(in_bndy%nmsg_ew_rcv, rcv_request, rcv_status, ierr)

! wait send to finish
call MPI_WaitAll(in_bndy%nmsg_ew_snd, snd_request, snd_status, ierr)

! notify each partner that my face is ready
do face=1,bndy%rcv_faces
  call event_notify(bndy%incoming(face)%rcv_ready)
end do

! when each partner face is ready
!   copy one of my faces to a partner's face
!   notify my partner's event when the copy is complete
do face=1,bndy%snd_faces
  copy_async(bndy%outgoing(face)%remote[], &
    bndy%outgoing(face)%local, &
    bndy%outgoing(face)%snd_done[], &
    bndy%outgoing(face)%snd_ready)
end do

! wait for all of my incoming faces to arrive
do face=1,bndy%rcv_faces
  call event_wait(bndy%incoming(face)%rcv_done)
end do

CAF 2.0
Multithreading

• Where can asynchronous threads of control arise in CAF 2.0?
  — spawned procedures
  — parallel loops
    – Fortran 90’s “do concurrent”

• Work in progress to employ Cilk-like lazy multithreading
  — generate continuations when spawning functions
  — generate a continuation when blocking for synchronization
Outline

• **Coarray Fortran**
  — original 1998 version
  — Fortran 2008 - a standard with coarrays

• **Coarray Fortran 2.0 (CAF 2.0)**
  — features
  — experiences - HPC challenge benchmarks + performance
  — implementation notes

• **Status and plans**
HPC Challenge Benchmark Goal: Productivity

• Priorities, in order
  — performance
  — source code volume

• Productivity = performance / (lines of code)

• Implications
  — EP STREAM Triad
    – outlined a loop to assist compiler optimization
  — Randomaccess
    – used software routing for higher performance
  — FFT
    – blocked packing/unpacking loops for bitreversal (8x gain for packing kernel)
  — HPL
    – tuned code to make good use of the memory hierarchy
double precision, allocatable :: a(:)[*], b(:)[*], c(:)[*]

...!

! each processor in the default team allocates their own array parts
allocate(a(local_n)[], b(local_n)[], c(local_n)[])

...

! perform the calculation repeatedly to get reliable timings
do round = 1, rounds
  do j = 1, rep
    call triad(a,b,c,local_n,scalar)
  end do
  call team_barrier()! synchronous barrier across the default team
end do
...

! perform the calculation with top performance
! assembly code is identical to that for sequential Fortran

subroutine triad(a, b, c, n ,scalar)
  double precision :: a(n), b(n), c(n), scalar
  a = b + scalar * c  ! EP triad as a Fortran 90 vector operation
end subroutine triad
Randomaccess

- A stream of updates to random locations in a distributed table
- Each update consists of xoring a random value into a random location in the table
- Each processor performs a subsequence of the updates
Randomaccess Software Routing

```fortran
event, allocatable :: delivered(:)[*], received(:)[*] ! (stage)
integer(i8), allocatable :: fwd(:,:, :)[*] ! (#, in/out, stage)

! hypercube-based routing: each processor has 1024 updates
do i = world_logsize-1, 0, -1 ! log P stages in a route

... call split(retain(:, last), ret_sizes(last), &
retain(:, current), ret_sizes(current), &
fwd(1:, out, i), fwd(0, out, i), bufsize, dist)

if (i < world_logsize-1) then
  event_wait(delivered(i+1))
  call split(fwd(1:, in, i+1), fwd(0, in, i+1), &
retain(:, current), ret_sizes(current), &
fwd(1:, out, i), fwd(0, out, i), bufsize, dist)
  event_notify(received(i+1)[from]) ! signal buffer is empty
endif

count = fwd(0, out, i)
event_wait(received(i)) ! ensure buffer is empty from last route
fwd(0:count, in, i)[partner] = fwd(0:count, out, i) ! send to partner

event_notify(delivered(i)[partner]) ! notify partner data is there

... end do
```
HPL

- Block-cyclic data distribution
- Team based collective operations along rows and columns
  - synchronous max reduction down columns of processors
  - asynchronous broadcast of panels to all processors

```fortran
(type(paneltype) :: panels(1:NUMPANELS)
  event, allocatable :: delivered(:)[*]

  do j = pp, PROBLEMSIZE - 1, BLKSIZE
    cp = mod(j / BLKSIZE, 2) + 1
    event_wait(delivered(3-cp))
    if (mycol == cproc) then
      if (ncol > 0) ! update part of the trailing matrix
        call fact(m, n, cp) ! factor the next panel
      end if
      call team_broadcast_async(panels(cp)%buff(1:ub), panels(cp)%info(8), &
                               delivered(cp))
      ! update rest of the trailing matrix
      if (nn-ncol>0) call update(m, n, col, nn-ncol, 3 - cp)
    end if
  end do
```
FFT

• Radix 2 1D FFT implementation
• Block distribution of array “c” across all processors
• Computation
  — permute elements: \( c = (c(\text{bitreverse}(i), i = 0, n-1) \)
    - 3 parts: pack data for all-to-all; team collective all-to-all; unpack data locally
  — FFT is log N stages
    - compute (log N - log P) stages of the FFT locally
      - transpose the data so that each processor has elements \( \equiv \text{rank mod P} \)
        - block distribution \( \rightarrow \) cyclic distribution
    - compute the remaining log P stages of the FFT locally
      - transpose the data back to its original order
        - cyclic distribution \( \rightarrow \) block distribution
Experimental Setup

• Coarray Fortran 2.0 by Rice University
  —source to source compilation from CAF 2.0 to Fortran 90
    – generated code compiled with Portland Group’s pgf90
  —CAF 2.0 runtime system built upon GASNet (version 1.14.2)
  —scalable implementation of teams, using O(log P) storage

• Experimental platform: Cray XT
  —systems
    – Franklin at NERSC
      2.3 GHz AMD “Budapest” quad-core Opteron, 2GB DDR2-800/core
    – Jaguar at ORNL
      2.1 GHz AMD “Budapest” quad-core Opteron, 2GB DDR2-800/core
  —network topology
    – 3D Torus based on Seastar2 routers
    – OS provides an arbitrary set of nodes to an application
Scalability: Relative Parallel Efficiency

EP STREAM Triad

HPL

FFT

Randomaccess
## Productivity = Performance / SLOC

### Performance (Cray XT4)

<table>
<thead>
<tr>
<th># of cores</th>
<th>STREAM Triad† (TByte/s)</th>
<th>RandomAccess* (GU P/s)</th>
<th>Global HPL† (TFlop/s)</th>
<th>Global FFT† (GFlop/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.14</td>
<td>0.08</td>
<td>0.36</td>
<td>6.69</td>
</tr>
<tr>
<td>256</td>
<td>0.54</td>
<td>0.24</td>
<td>1.36</td>
<td>22.82</td>
</tr>
<tr>
<td>1024</td>
<td>2.18</td>
<td>0.69</td>
<td>4.99</td>
<td>67.80</td>
</tr>
<tr>
<td>4096</td>
<td>8.73</td>
<td>2.01</td>
<td>18.3</td>
<td>187.04</td>
</tr>
</tbody>
</table>

*Measured on Jaguar †Measured on Franklin

### Source lines of code

<table>
<thead>
<tr>
<th>HPC Challenge Benchmark</th>
<th>Source Lines of Code</th>
<th>Reference SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Randomaccess</td>
<td>409</td>
<td>787</td>
</tr>
<tr>
<td>EP STREAM Triad</td>
<td>58</td>
<td>329</td>
</tr>
<tr>
<td>Global HPL</td>
<td>786</td>
<td>8800</td>
</tr>
<tr>
<td>Global FFT</td>
<td>~390</td>
<td>1130</td>
</tr>
</tbody>
</table>

**Notes**
- EP STREAM: 66% of memory B/W peak
- Randomaccess: high performance without special-purpose runtime
- HPL: 49% of FP peak at @ 4096 cores (uses dgemm)
CAF 2.0 Early Experiences Summary

- A viable programming model for scalable parallel computing
  —expressive
  —easy to use

- Prototype implementation scales to thousands of nodes

- Scalable high performance
  —demonstrated by HPC Challenge Benchmark results
Outline

• Coarray Fortran
  — original 1998 version
  — Fortran 2008 - a standard with coarrays

• Coarray Fortran 2.0 (CAF 2.0)
  — features
  — experiences - HPC challenge benchmarks + performance
  — implementation notes

• Status and plans
CAF 2.0 Team Representation

• Designed for scalability: representation is $O(\log S)$ per node for a team of size $s$

• Based on the concept of pointer jumping

• Pointers to predecessors and successors at distance $i = 2^j$, $j = 0 .. \lfloor \log S \rfloor$
Collective Example: Barrier

Dissemination algorithm

for $k = 0$ to $\lceil \log_2 P \rceil$

processor $i$ signals processor $(i + 2^k) \mod P$ with a PUT

processor $i$ waits for signal from $(i - 2^k) \mod P$
Collective Example: Broadcast

Binomial Tree

round 0
round 1
round 2
Progress Engine

• Tracks and manages state for all outstanding asynchronous operations on an image

• Operations are set up as finite state machines
  —initialize, waiting for a non-blocking write, etc.

• Advance function invoked regularly
  —inside various CAF 2.0 runtime calls
  —(eventually) sprinked through user code by our compiler
  —manually as desired

• Gives each operation a chance to make progress

• Cooperative multitasking

• Research issue
  —scheduling progress engine tasks when there are multiple threads
Implementing Non-blocking Collectives

• State machine for each communication partner
  — each state machine begins in state 0

• Example: long broadcast
  — state machine (1) - for parent communication
    0: provide my buffer location to my parent in the broadcast tree
    set closure variables
      count = number of my children; event = event to signal for completion
    1: test for data for my parent; if no, state = 1; return to progress engine
    enqueue instance of state machine (2) for each child in the broadcast tree
    dequeue myself from the progress engine
  — state machine (2) - for child communication
    0: test if my child provided buffer location for receiving broadcast
      if not, return to progress engine
      provide data to my child
      decrement a count in the closure for (1)
      if count = 0, signal event in parent’s closure, free my parent’s closure
      dequeue myself from the progress engine; free my closure
Outline

• Coarray Fortran
  —original 1998 version
  —Fortran 2008 - a standard with coarrays

• Coarray Fortran 2.0 (CAF 2.0)
  —features
  —experiences - HPC challenge benchmarks + performance
  —implementation notes

• Status and plans
Strengths and Weaknesses of CAF 2.0

• Strengths
  — provides full control over data and computation partitioning
  — admits sophisticated parallelizations
  — compiler and runtime systems are tractable
  — yields scalable high performance today with careful programming

• Weaknesses
  — users code data movement and synchronization
    – significantly harder than HPF
  — optimizing performance can require careful parallel programming
    – overlapping communication and computation may require managing multiple communication buffers
    – hiding latency requires
      using non-blocking primitives for data movement and synchronization
      overlapping latency of communication with computation
      managing the completion of asynchronous operations
Implementation Status & Plans

• Source-to-source translator is a work in progress
  — requires no vendor buy-in
  — delivers node performance of mature vendor compilers
  — ongoing work to improve Fortran coverage in ROSE

• Ongoing work
  — copointers
  — lazy multithreading
  — coarray binding interface for inter-team communication
  — graph topology for managing irregular communication patterns

• Future plans
  — use compiler-based vectorization to target SIMD and accelerators
Planned Application Studies

- LANL’s Parallel Ocean Program
  - block structured, dense matrix
- Sandia’s S3D
  - regular, dense matrix
- LANL’s HEAT
  - cell-by-cell AMR code, sparse matrix
- Community Earth System Model
  - coupled code
  - multiple block-structured dense matrix components