Coarray Fortran: Past, Present, and Future

John Mellor-Crummey
Department of Computer Science
Rice University
johnmc@cs.rice.edu

Rice CAF Team

- Staff
  - Bill Scherer
  - Laksono Adhianto
  - Guohua Jin
  - Dung Nguyen
  - Fengmei Zhao
- Student
  - Chaoran (Jack) Yang
- Alumni
  - Yuri Dotsenko
  - Cristian Coarfa

Outline

- Partitioned Global Address Space (PGAS) languages
- Coarray Fortran, circa 1998 (CAF98)
- Assessment of CAF98
- A look at the emerging Fortran 2008 standard
- A new vision for Coarray Fortran

Partitioned Global Address Space Languages

- Global address space
  - one-sided communication (GET/PUT) simpler than msg passing
- Programmer has control over performance-critical factors
  - data distribution and locality control lacking in OpenMP
  - computation partitioning
  - communication placement

  } HPF & OpenMP compilers must get this right
- Data movement and synchronization as language primitives
  - amenable to compiler-based communication optimization

Outline

- Partitioned Global Address Space (PGAS) languages
- Coarray Fortran, circa 1998 (CAF98)
  - motivation & philosophy
  - execution model
  - co-arrays and remote data accesses
  - allocatable and pointer co-array components
  - processor spaces: co-dimensions and image indexing
  - synchronization
  - other features and intrinsic functions
- Assessment of CAF98
- A look at the emerging Fortran 2008 standard
- A new vision for Coarray Fortran

Coarray Fortran Design Philosophy

- What is the smallest change required to make Fortran 90 an effective parallel language?
- How can this change be expressed so that it is intuitive and natural for Fortran programmers?
- How can it be expressed so that existing compiler technology can implement it easily and efficiently?

Coarray Fortran Overview

- Explicitly-parallel extension of Fortran 95
  - defined by Numrich & Reid
- SPMD parallel programming model
- Global address space with one-sided communication
- Two-level memory model for locality management
  - local vs. remote memory
- Programmer control over performance critical decisions
  - data partitioning
  - communication
  - synchronization
- Suitable for mapping to a range of parallel architectures
  - shared memory, message passing, hybrid

SPMD Execution Model

- The number of images is fixed and each image has its own index, retrievable at run-time:
  - $1 \le$ num_images()
  - $1 \le$ this_image() $\le$ num_images()
- Each image executes the same program independently
- Programmer manages local and global control
  - code may branch based on processor number
  - synchronize with other processes explicitly
- Each image works on its local and shared data
- A shared “object” has the same name in each image
- Images access remote data using explicit syntax

Shared Data – Coarrays

- Syntax is a simple parallel extension to Fortran 90
  - it uses normal rounded brackets ( ) to point to data in local memory
  - it uses square brackets [ ] to point to data in remote memory
- Coarrays can be accessed from any image
- Coarrays are *symmetric*
- Coarrays can be SAVE, COMMON, MODULE, ALLOCATABLE
- Coarrays can be passed as procedure arguments

Examples of Coarray Declarations

```
real :: array(N, M)[*]
integer ::scalar[*]

real :: b(N)[p, *]
real :: c(N, M)[0:p, -7:q, 11:*]

real, allocatable :: w(:, :, :)[:, :]

type(field) :: maxwell[p, *]
```

One-sided Communication with Coarrays

```
integer a(10,20)[*]
```

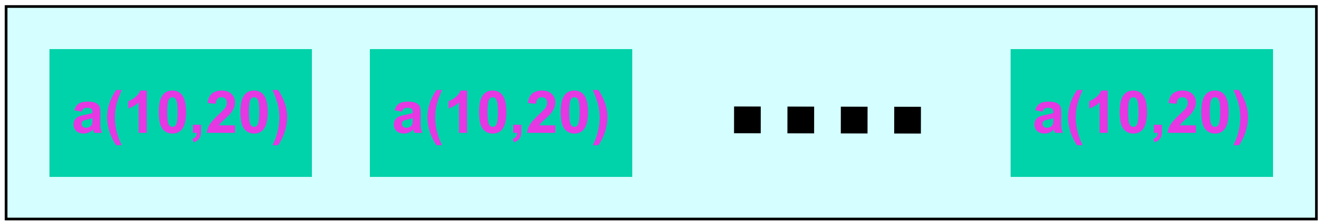


image 1 image 2 image N

```
if (this_image() > 1) then
  a(1:5,1:10) = a(1:5,1:10)[this_image()-1]
endif
```

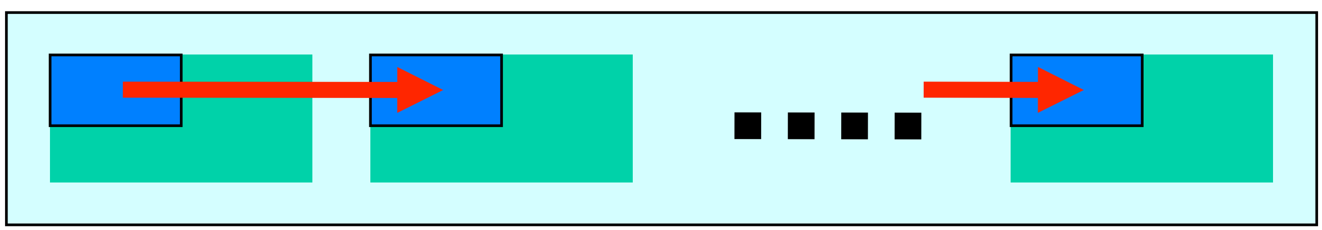


image 1 image 2 image N

Flavors of Remote Accesses

```
y = x[p]                        ! singleton GET
y[p] = x                        ! singleton PUT

y(:) = z(:) + x(:)[p]           ! vector GET
a(:, k)[p] = a(:, 1)            ! vector PUT

a(1:N:2)[p] = c(1:N:2, j)       ! strided PUT
a(1:N:2) = c(1:N:2, j) [p]      ! strided GET

x(prin(k1:k2)) = x(prin(k1:k2)) + x(ghost(k1:k2))[neib(p)] ! gather
x(ghost(k1:k2))[neib(p)] = x(prin(k1:k2))   ! scatter
```

No brackets = local access

Allocatable Coarrays

```
real, allocatable :: a(:)[:], s[:, :]
:

allocate( a(10)[*],  s[-1:34, 0:*])    ! symmetric and collective
```

- Illegal allocations:
  - allocate(a(n))
  - allocate(a(n)[p])
  - allocate(a(this_image())[*])

Allocatable Coarrays and Pointer Components

```
type T
  integer, allocatable :: ptr(:)
end type T
type (T), allocatable :: z[:]

allocate( z[*] )
allocate( z%ptr( this_image()*100 ))    ! asymmetric
allocate( z[p]%ptr(n) )    ! illegal

x = z%ptr(1)
x(:) = z[p]%ptr(i:j:k) + 3
```

Processor Space

	1	2	3	4
1	1	5	9	13
2	2	6	10	14
3	3	7	11	15
4	4	8	12	16

x[4,*]   this_image() = 15   this_image(x) = (/3,4/)

CAF98 Synchronization Primitives

- sync_all()
- sync_team(team, [wait])
- flush_memory()

Source-level Broadcast

```
if (this_image() == 1) then
  x = …
  do i = 2, num_images()
    x[i] = x
  end do
end if
call sync_all()   ! barrier
if (x == …) then
  …
end if
```

Exchange using Barrier Synchronization

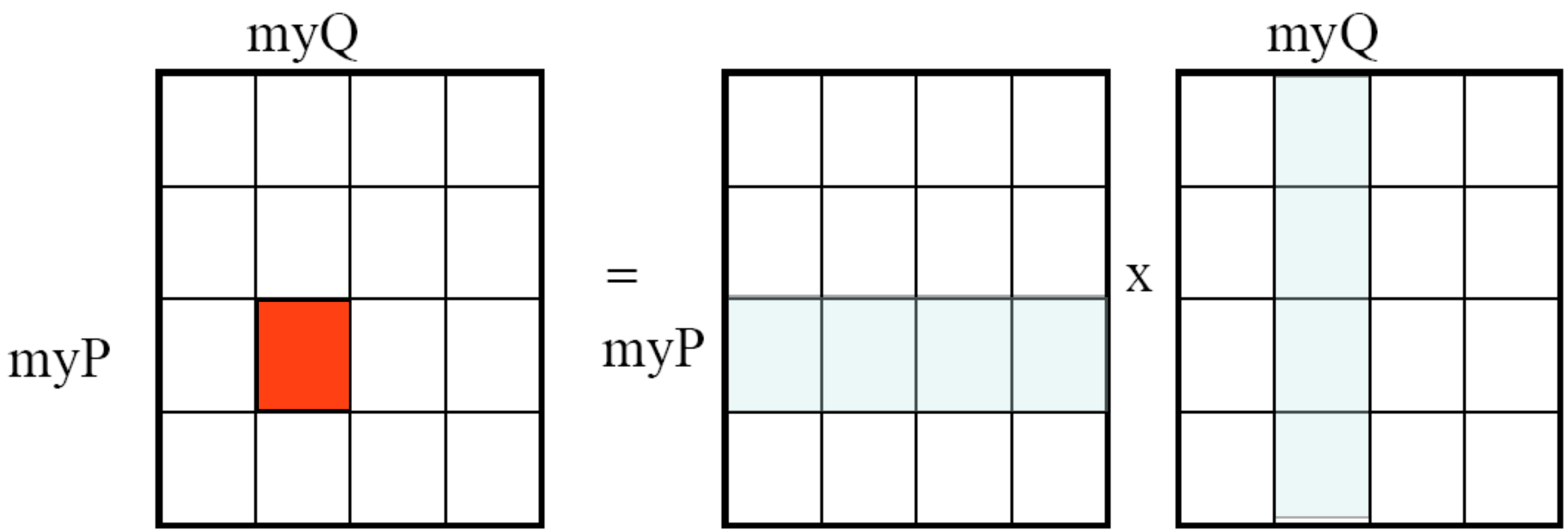
pack SendBuff buffers to exchange with Left and Right

```
RecvBuff(:,1)[Left] = SendBuff(:,-1)
RecvBuff(:,-1)[Right] = SendBuff(:,1)

call sync_all()  ! barrier
```

unpack RecvBuff buffer

Example: Parallel Matrix Multiplication



Parallel Matrix Multiplication 2

```
real, dimension(n, n)[p, *] :: a, b, c

do q = 1, p
  do i = 1, n
    do j = 1, n
      do k = 1, n
         c(i, j)[myP, myQ] = c(i, j)[myP, myQ]
                        + a(i, k)[myP, q]*b(k, j)[q, myQ]
      end do
    end do
  end do
end do
```

Parallel Matrix Multiplication 3

```
real, dimension(n, n)[p, *] :: a, b, c

do q = 1, p
  do i = 1, n
    do j = 1, n
      do k = 1, n
         c(i, j) = c(i, j) + a(i, k)[myP, q]*b(k, j)[q, myQ]
      end do
    end do
  end do
end do
```

A Finite Element Example (Numrich, Reid; 1998)

```
subroutine assemble(start, prin, ghost, neib, x)
  integer :: start(:),prin(:),ghost(:),neib(:),k1, k2, p
  real :: x(:) [*]
  call sync_all(neib)
  do p = 1, size(neib) ! Add contribs. 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)]
  enddo
  call sync_all(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))
  enddo
  call sync_all
end subroutine assemble
```

High-level Assessment of CAF

- Advantages
  - admits sophisticated parallelizations with compact expression
  - doesn’t require as much from compilers
  - yields scalable high performance today with careful programming
    - if you put in the effort, you can get the results
- Disadvantages
  - users code data movement and synchronization
    - tradeoff between the abstraction of HPF vs. control of CAF
  - optimizing performance can require intricate programming
    - buffer management is fully exposed!
  - expressiveness is a concern for CAF
    - insufficient primitives to express a wide range of programs

A Closer Look at CAF98 Details

- Strengths
  - one-sided data access can simplify some programs
  - vectorized access to remote data can be efficient
    - amortizes communication startup costs
    - data streaming can hide communication latency (e.g. on the Cray X1)
- Weaknesses
  - synchronization can be expensive
    - single critical section was very limiting
    - synch_all, synch_team were not sufficient
      - sync_all: barriers are a heavy-handed mechanism
      - sync_team semantics requires $O(n^2)$ pairwise communication
    - rolling your own collectives doesn't lead to portable high performance
  - latency hiding is impossible in important cases
    - procedure calls had implicit barriers to guarantee data consistency
      - communication couldn't be overlapped with a procedure call

Emerging Fortran 2008 Standard

Coarray features being considered for inclusion

- Single and multidimensional coarrays
- Collective allocation of coarrays to support a symmetric heap
- Critical section for structured mutual exclusion

```
critical
  ...
end critical
```

- Sync all(): global barrier
- Sync images(image list)
  - any processor can specify any list of images; * = all
- Sync memory(), atomic_define(x, val), atomic_ref(x,??)
- thisimage(), thisimage(c), and image_index(c, (/3,1,2/))
- Locks, along with lock(L) and unlock(L) statements
- all stop: initiate asynchronous error termination

Are F2008 Coarrays Ready for Prime Time?

Questions worth considering

1. What types of parallel systems are viewed as the important targets for Fortran 2008?

2. Does Fortran 2008 provide the set of features necessary to support parallel scientific libraries that will help catalyze development of parallel software using the language?

3. What types of parallel applications is Fortran 2008 intended to support and is the collection of features proposed sufficiently expressive to meet those needs?

4. Will the collection of coarray features described provide Fortran 2008 facilitate writing portable parallel programs that deliver high performance on systems with a range of characteristics?

1. Target Architectures?

CAF support must be ubiquitous or (almost) no one will use it

- Important targets
  - clusters and leadership class machines
  - multicore processors and SMPs
- Difficulties
  - F2008 CAF lacks flexibility, which makes it a poor choice for multicore
    - features are designed for regular, SPMD
    - multicore will need better one-sided support
      - flexible allocation, extension, manipulation of shared data
        - e.g. linked data structures (requires pointers, dynamic allocation)
  - current scalable parallel systems lack h/w shared memory
    - e.g. clusters, Blue Gene, Cray XT
    - big hurdle for third-party compiler vendors to target scalable systems

2. Adequate Support for Libraries?

Lessons from MPI: Library needs [MPI 1.1 Standard]

- **Safe communication space**: libraries can communicate as they need to, without conflicting with communication outside the library
- **Group scope for collective operations**: allow libraries to avoid unnecessarily synchronizing uninvolved processes
- **Abstract process naming**: allow libraries to describe their communication to suit their own data structures and algorithms
- **Provide a means to extend the message-passing notation:** user-defined attributes, e.g., extra collective operations

All are missing if F2008!

Lack of Support for Process Subsets

A library can't conveniently operate on a process subset

- Multidimensional coarrays
  - must be allocated across all process images
  - can't conveniently employ this abstraction for a process subset
- Image naming
  - all naming of process images is global
  - would make it hard to work within process subsets
    - must be cognizant of their embedding in the whole
- Allocation/deallocation
  - libraries shouldn't unnecessarily synchronize uninvolved processes
  - but ... coarrays in F2008 require
    - global collective allocation/deallocation
  - serious complication for coupled codes on process subsets
    - complete loss of encapsulation

3. Target Application Domains?

- Can F2008 support applications that require one-sided update of mutable shared dynamic data structures?
- No. Two key problems
  - can't add a piece to a partner's data structure
    - F2008 doesn't support remote allocation
    - F2008 doesn't support pointers to remote data
    - F2008 doesn't support remote execution using "function shipping"
  - synchronization is inadequate
    - critical sections are an extreme limit on concurrency
      - only one process active per static name
    - unreasonable to expect users to "roll their own"
    - no support for point-to-point ordering
      - one-sided synchronization, e.g. post(x), wait(x)
    - no support for collectives
- As defined, F2008 useful for halo exchanges on dense arrays

4. Adequate Support for Writing Fast Code?

- Lack of support for hiding synchronization latency
  - sync all, sync images are very synchronous
  - need one-sided synchronization
    - F2008 considered notify/query between images, but tabled for now
    - even that doesn’t suffice
  - no split-phase barrier
- Lack of support for exploiting locality in machine topology
- Lack of a precise memory model
  - developers must use loose orderings where possible
  - must be able to reason about what behaviors one should expect
  - programs must be resilient to reorderings

Lessons from MPI 1.1

What capabilities are needed for parallel libraries?

- Abstraction for a group of processes
  - functions for constructing and manipulating process groups
- Virtual communication topologies
  - e.g. cartesian, graph
  - neighbor operation for indexing
- Multiple communication contexts
  - e.g. parallel linear algebra uses multiple communicators
    - rows of blocks, columns of blocks, all blocks

Recommendations for Moving Forward (Part 1)

- Only one-dimensional co-arrays
  - no collective allocation/deallocation: require users to synchronize
- Team abstraction that represents explicitly ordered process groups
  - deftly supports coupled codes, linear algebra
  - enables renumbering to optimize embedding in physical topology
- Topology abstraction for groups: cartesian and graph topologies
  - cartesian is a better alternative to k-D coarrays
    - supports processor subsets, periodic boundary conditions as well
  - graph is a general abstraction for all purposes
- Multiple communication contexts
  - apply notify/query to semaphore-like variables for multiple contexts
- Add support for function shipping
  - spawn remote functions for latency avoidance
  - spawn local functions to exploit parallelism locally
    - lazy multithreading and work stealing within an image

Recommendations for Moving Forward (Part 2)

- Better mutual exclusion support for coordinating activities
  - short term: critical sections using lock variables, lock sets
  - longer term: conditional ATOMIC operations based on transactional memory?
- Rich support for collectives, including
  - user-defined reduction operators
  - scan reductions
  - all-to-all operations
- Add multiversion variables
  - simplify producer/consumer interactions in a shared memory world
- Add global pointers

A New Vision for Coarray Fortran: CAF 2.0

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

CAF 2.0 Design Principles

Largely borrowed from MPI 1.1 design principles

- **Safe communication spaces** allow for modularization of codes and libraries by preventing unintended message conflicts
- Allowing **group-scoped collective operations** avoids wasting overhead in processes that are otherwise uninvolved (potentially running unrelated code)
- **Abstract process naming** allows for expression of codes in libraries and modules; it is also mandatory for dynamic multithreading
- **User-defined extensions** for message passing and **collective operations** interface support the development of robust libraries and modules

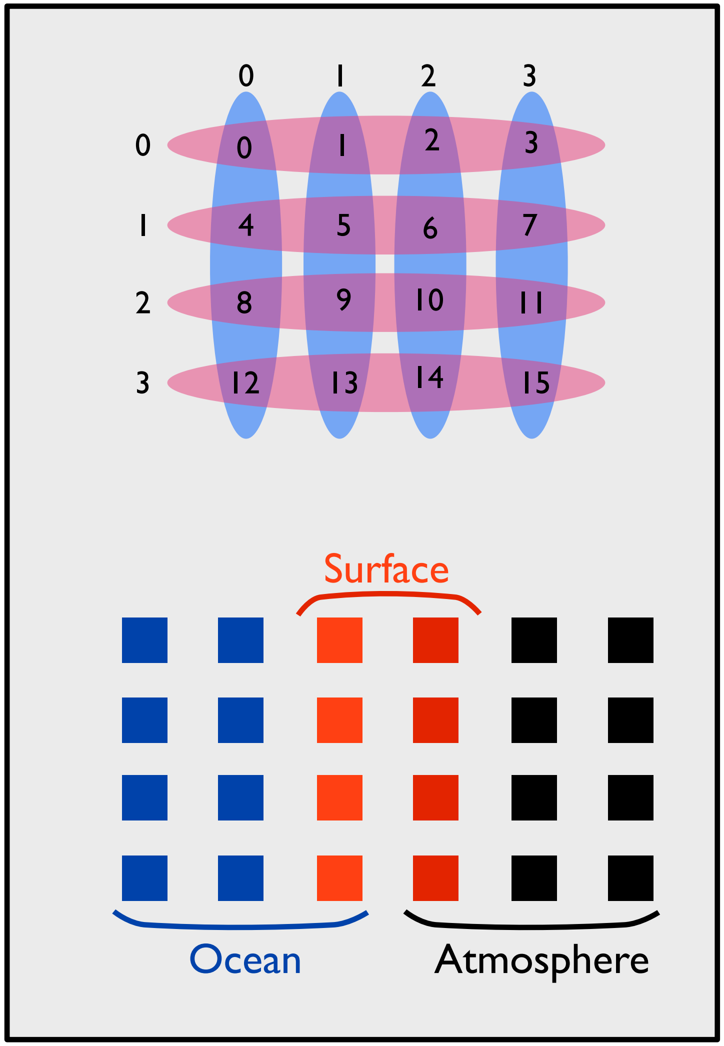
The syntax for language features must be convenient

Coarray Fortran 2.0 Features

- **Process subsets: teams**
- **Topologies**
- **Copointers**
- **Synchronization**
- **Collective communication**

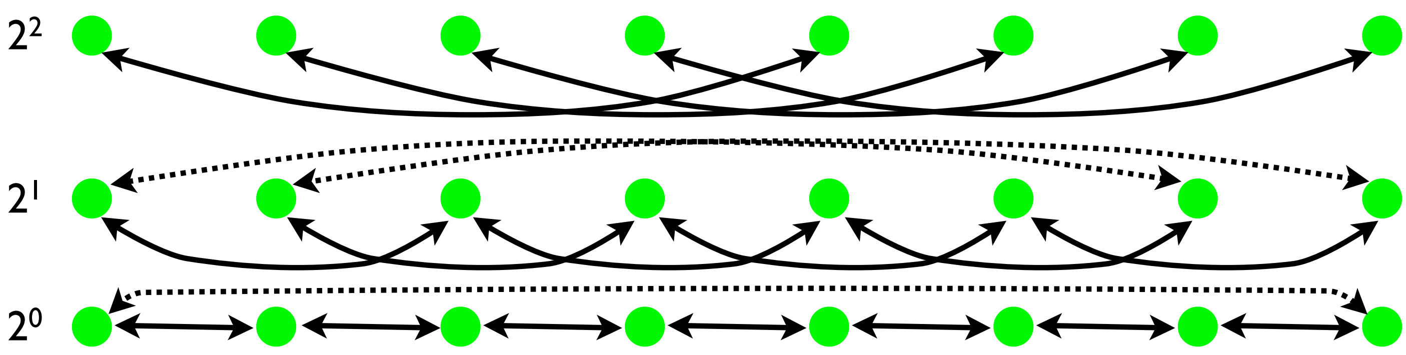
Process Subsets: Teams

- **Teams are first-class entities**
  - —ordered sequences of process images
  - —namespace for indexing images by rank r in team t
    - – r ∈ {0..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 in CAF 2.0

- Predefined teams
  - team_world: contains all images
  - team_default: dynamically scoped, modifiable using “with team”
- Team representation
  - distributed representation; caching of team members
  - our approach:
    - represent teams using multiple bidirectional circular linked lists
      - at distances 1, 2, ..., $2^{\log(\text{team size})}$



Splitting Teams

- team_split (team, color, key, team_out)
  - team: team of images (handle)
  - color: control of subset assignment. Images with the same color are in the same new team
  - key: control of rank assigment (integer)
  - team_out: receives handle for this image’s new team
- Example:
  - Consider p processes organized in a q × q grid
  - Create separate teams each row of the grid

```
team newteam
integer rank, row
rank = this_image(team_world)
row = rank/q
call team_split(team_world, row, rank, newteam)
```

Teams and Coarrays

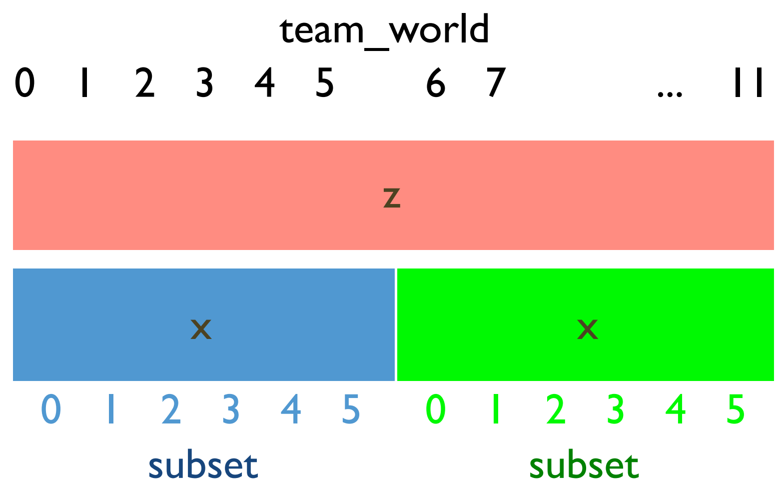
```
real, allocatable :: x(:,:)[*] ! 2D array
real, allocatable :: z(:,:)[*]
team :: subset
integer :: color, rank

! 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

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_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
```

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^{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

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

```
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
```

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

Safe Communication Spaces

- Event object for anonymous pairwise coordination
- Safe synchronization space: can allocate as many events as possible
- Notify: nonblocking, asynchronous signal to an event; a pairwise fence between sender and target image
- Wait: blocking wait for notification on an event or event set
- Trywait: non-blocking consumption of a notify, if available
- Waitany: return the ready event in an event set

Team-based Collective Communication

- Language and compiler support for collectives
- Source and destination variables
  - —may be coarrays or not
  - —scalars, whole arrays, array sections
- Size implicit: computed by the compiler from variables
- Team may be implicit (team_default) or explicit

Team-based Synchronous Collectives

Statement	Description	Syntax
team_broadcast	broadcasts a data from an image to all images in a team	team_broadcast(var, root_rank [, team])
team_gather	collects individual data from each image in a team at one image	team_gather(var_src, var_dest, root_rank [, team])
team_allgather	gathers data from all images and distribute it to all images	team_allgather(var_src, var_dest [, team])
team_reduce	reduces data, the result is stored to an image of the team	team_reduce(var_src, var_dest, root_rank, operator [, team] [, ro])
team_allreduce	reduces data, the result is stored to all images of the team	team_allreduce(var_src, var_dest, operator [, team] [, ro])
team_scan	performs partial reduction (scan), each image store the result of reduction from its neighbor	team_scan(var_src, var_dest [, team])
team_scatter	distributes individual data from an image to each image in a team	team_scatter(var_src, var_dest, root_rank [, team])
team_shift	moves data from another image at an offset within a team	team_shift(var_src, var_dest, image_offset [, team])
team_sort	sorts arrays of the same size and type within a team	team_sort(var_src, var_dest, comparison_function [, team])

For most statements:

typedef::var_src	local source variable
typedef::var_dest[*]	target Coarray Fortran variable
integer::root_rank	the rank of the root image
team::team	process subset (default team if not specified)

Asynchronous Point-to-Point Communication

Design Challenges

- Allow read/write of coarray data to be overlapped with computation
- Allow determination of when asynchronous operations are complete
- Await completion of asynchronous operations in any order
- Make the completion of asynchronous operations orthogonal to scopes
  - —any routine whatsoever can request completion at any point in the future
- Provide a syntactic construct that is easy to use

Asynchronous Point-to-Point

copy_async(var_dest, var_src, ev_after *[, ev_before]*)

var_dest	=	a coarray reference target
var_src	=	a coarray reference source
ev_after	=	an event indicating that the write to dest is complete
ev_before	=	an optional event indicating that the source data is ready

Team-based Asynchronous Collectives

- **Collective communication may be overlapped with**
  - —execution of other asynchronous communication
    - – collective or point-to-point
  - —computation
- **Notifying an event signals that an operation is complete**

Statement	Description
team_barrier_async(event [, team])	barrier synchronization between image processes
team_broadcast_async(var, root, event [, team])	broadcasts data from an image to all images in a team
team_gather_async(var_src, var_dest, root, event [, team])	collects individual data from each image in a team at one image
team_allgather_async(var_src, var_dest, event [, team])	gathers data from all images and distributes it to all images
team_reduce_async(var_src, var_dest, root, operator, event [, team])	reduces data; the result is stored to an image of the team
team_allreduce_async (var_src, var_dest, operator, event [, team])	reduces data; the result is stored to all images of the team
team_scatter_async(var_src, var_dest, root, event [, team])	distributes individual data from an image to each image in a team
team_alltoall_async(var_src, var_dest, event [, team])	sends distinct data from each image to every image in a team
team_sort_async(var_src, var_dest, comparison_fn, event [, team])	sorts arrays of the same size and type within a team

Dynamic Multithreading

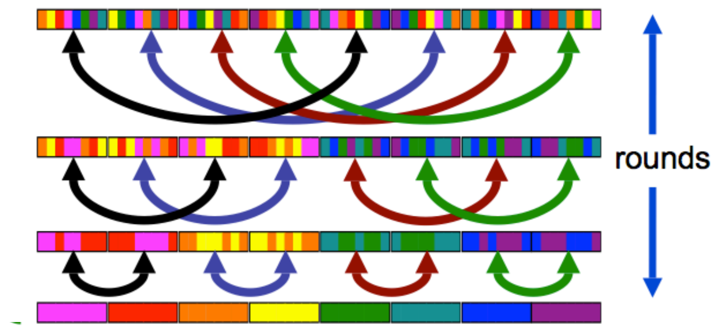
- Spawn
  - Create local or remote asynchronous threads by calling a procedure declared as a co-function
    - Simple interface for function shipping
  - Local threads can exploit multicore parallelism
  - Remote threads can be created to avoid latency when manipulating remote data structures
- Finish
  - Terminally strict synchronization for (nested) spawned sub-images
  - Orthogonal to procedures (like X10 and unlike Cilk)
    - Exiting a procedure does not require waiting on spawned sub-images

CAF 2.0 Examples

CAF 2.0 Randomaccess

```
module module_route
  ...
  event, allocatable, dimension(:) :: delivered[*]
  event, allocatable, dimension(:) :: received[*]
  integer(8), allocatable, dimension(:,:,:) :: fwd[*]
 contains
   ...
   subroutine route()
   ...
   do i = world_logsize-1, 0, -1
     partner = mod(world_rank + distance + world_size, &
                   world_size)
     ...
     call split(..., fwd(1:,out,i), fwd(0,out,i), ...)
     if (i < world_logsize-1) then
       call event_wait(delivered(i+1))
       call split(fwd(1:,in,i+1),..., fwd(1:,out,i), &
                 fwd(0,out,i), ...)
       call event_notify(received(i+1)[from])
     endif

     copy_async(fwd(0:outgoing_size,in,i)[partner], &
           fwd(0:outgoing_size,out,i),  &
           delivered(i)[partner], received(i))
     ...
     from = mod(world_rank - distance + world_size, &
           world_size)
   enddo
   end subroutine route
end module module_route
```

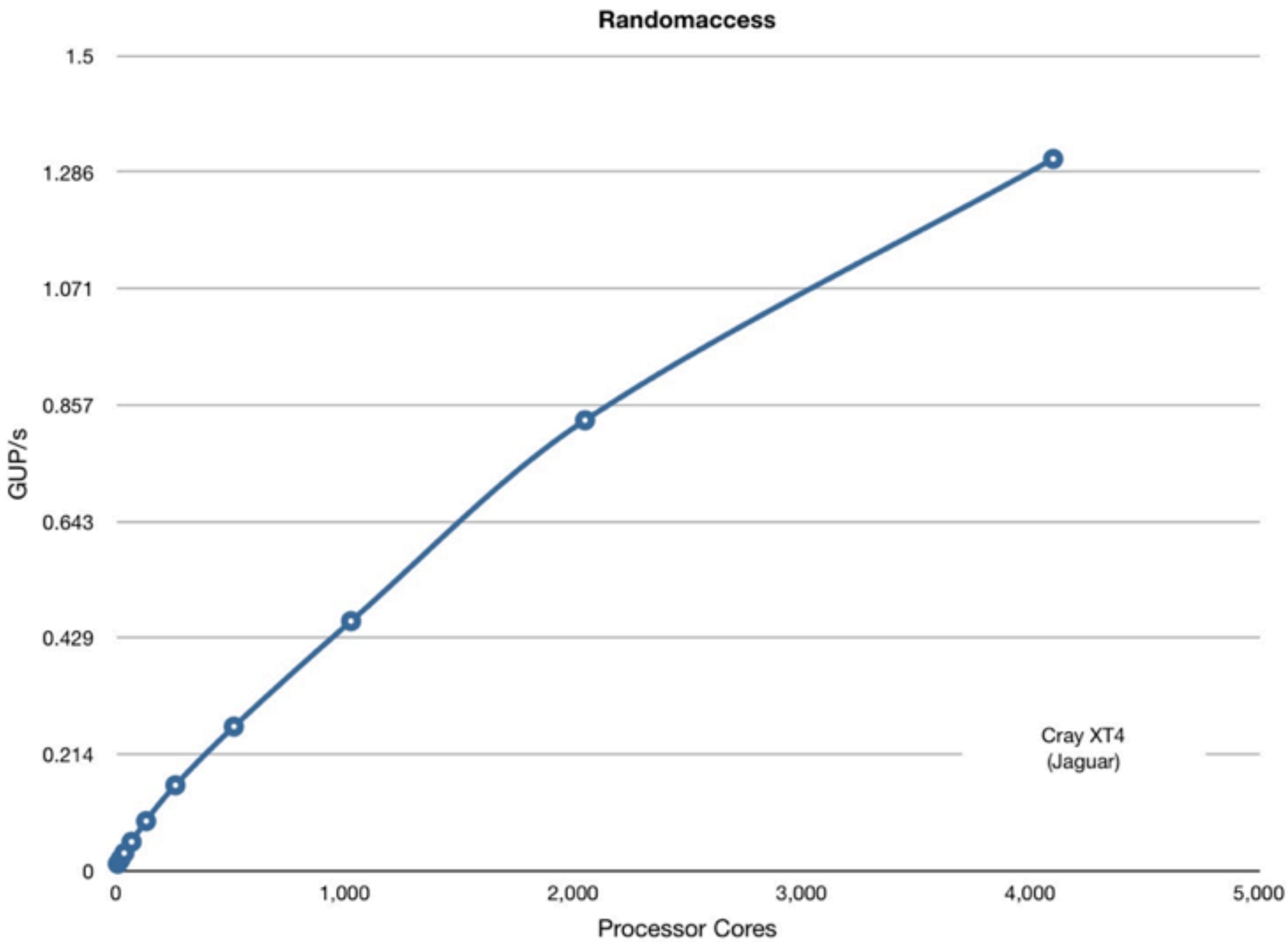


Route groups of updates to their destinations using log P rounds of communication

Randomaccess with Function Shipping

```
module module_table
  integer(8), allocatable :: table(:)[*]
  ...
  subroutine apply_global_updates(buffer, size)
    integer(8) :: buffer(:)
    ...
    finish
      do i = 1, size
        pe = ishft(buffer(i), -local_table_logsize)
        pe = iand(pe, world_size_minus_one)
        index = iand(buffer(i), local_table_size - 1)
        if (pe == world_rank) then
          table(index) = ieor(table(index), buffer(i))
        else
          spawn update(table, index, buffer(i))[pe]
        endif
        update_index = update_index + 1
      end do
    end finish
  end subroutine apply_global_updates
  ...
end module module_table
```

CAF 2.0 Randomaccess Performance

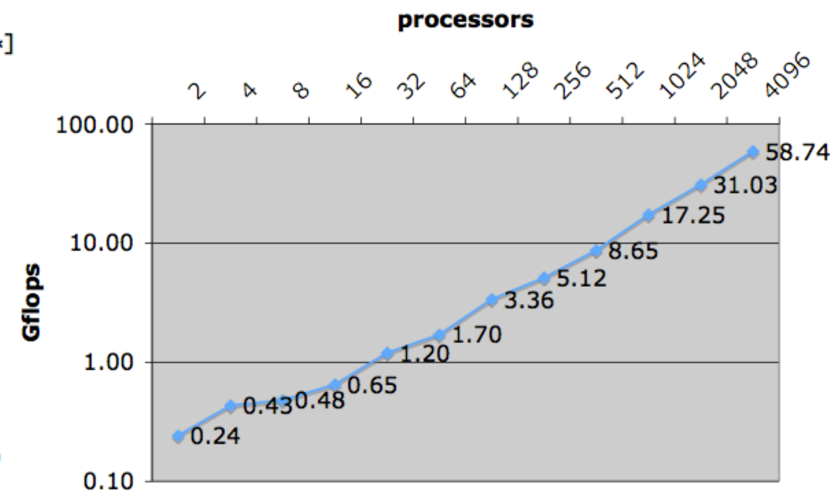


CAF 2.0 FFT

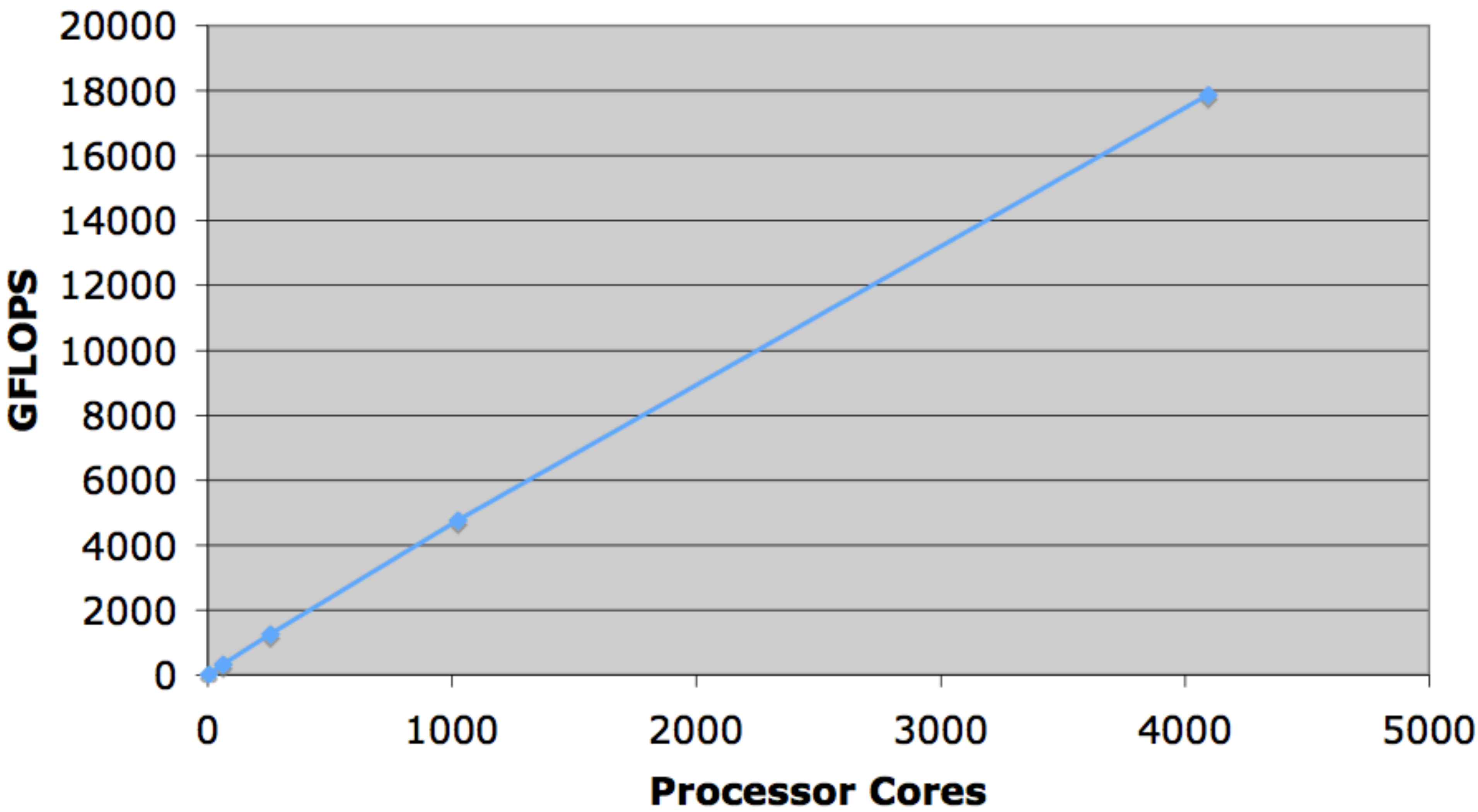
```
module module_fft
    complex, allocatable, dimension (:,2) :: c[*]
    event, allocatable, dimension(:) :: ready_to_copy[*]
    event, allocatable, dimension(:) :: copied[*]
    event, allocatable, dimension(:) :: prefetch[*]
    ...
contains
    ...
  subroutine fft(n_local_size, block_size)
    complex(8), target  :: cbuff(0:n_local_size-1)
    ...
    ! remote communication
    do l = loc_comm, levels
       m  = ishft(1, l)
       partner     = ieor(rank, ishft(1,l-loc_comm))
       ...
       event_notify(ready_to_copy(l-loc_comm)[partner])
       event_wait(ready_to_copy(l-loc_comm))
       ...
       ! prefetch blocks of data
        do outer = 0, (n_local_size/2)-1, block_size
           lo = index_adjustment + outer
           hi = lo + block_size -1
           copy_async(cbuff(lo:hi), c(lo:hi,last)[partner], &
             prefetch(outer/block_size), &
             copied(l-1 - loc_comm))
        end do

       do outer = 0, (n_local_size/2)-1, block_size
          ! Get a chunk of data
          call event_wait(prefetch(outer/block_size))

          ! Process it
          ...
          ! Send result to partner
          copy_async(c(lo:hi,current)[partner], &
            cbuff(lo:hi), copied(l-loc_comm)[partner])
       end do
       ...
  end subroutine fft
end module module_fft
```



CAF 2.0 HPL Performance



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, function shipping, asynchronous operations
- Provides expressiveness, simplicity and orthogonality
- Source-to-source translator and runtime are operational
  - —requires no vendor buy-in
  - —will deliver node performance of mature vendor compilers
- Status
  - —what's there: teams, coarrays, collectives, events, locks, event sets, lock sets, function shipping, asynchronous communication
  - —what's missing: topologies, copointers, multithreading
- Coming attractions:
  - —coarray binding interface for inter-team communication