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
  - Fengmei Zhao
- 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
  - computation partitioning
  - communication placement

lacking in OpenMP

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

Co-array 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?

Co-Array 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
- Co-arrays can be accessed from any image
- Co-arrays are *symmetric*
- Co-arrays can be SAVE, COMMON, MODULE, ALLOCATABLE
- Co-arrays 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)[*]
```

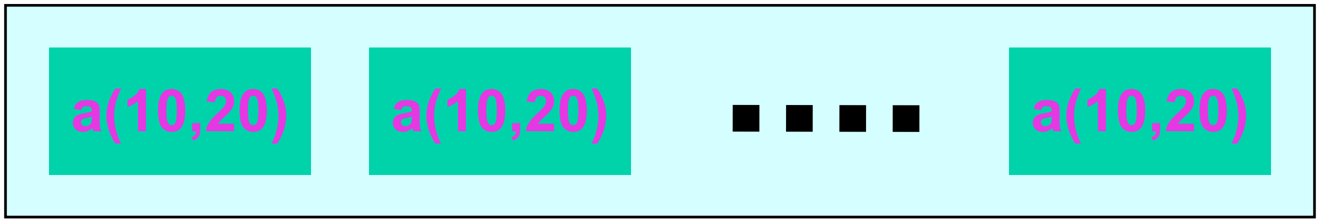


image 1 image 2 image N

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

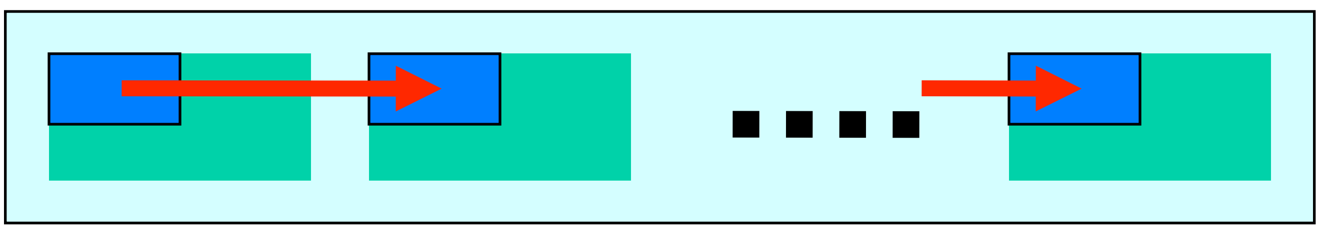


image 1 image 2 image N

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 Co-arrays

```
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

	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/)

Processor Space 2

	0	1	2	3
0	**1**	**5**	**9**	**13**
1	**2**	**6**	**10**	**14**
2	**3**	**7**	**11**	**15**
3	**4**	**8**	**12**	**16**

x[0:3,0:*]   this_image() = 15   this_image(x) = (/2,3/)

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

Exchange using Point-to-point Synchronization

pack SendBuff buffers to exchange with Left and Right

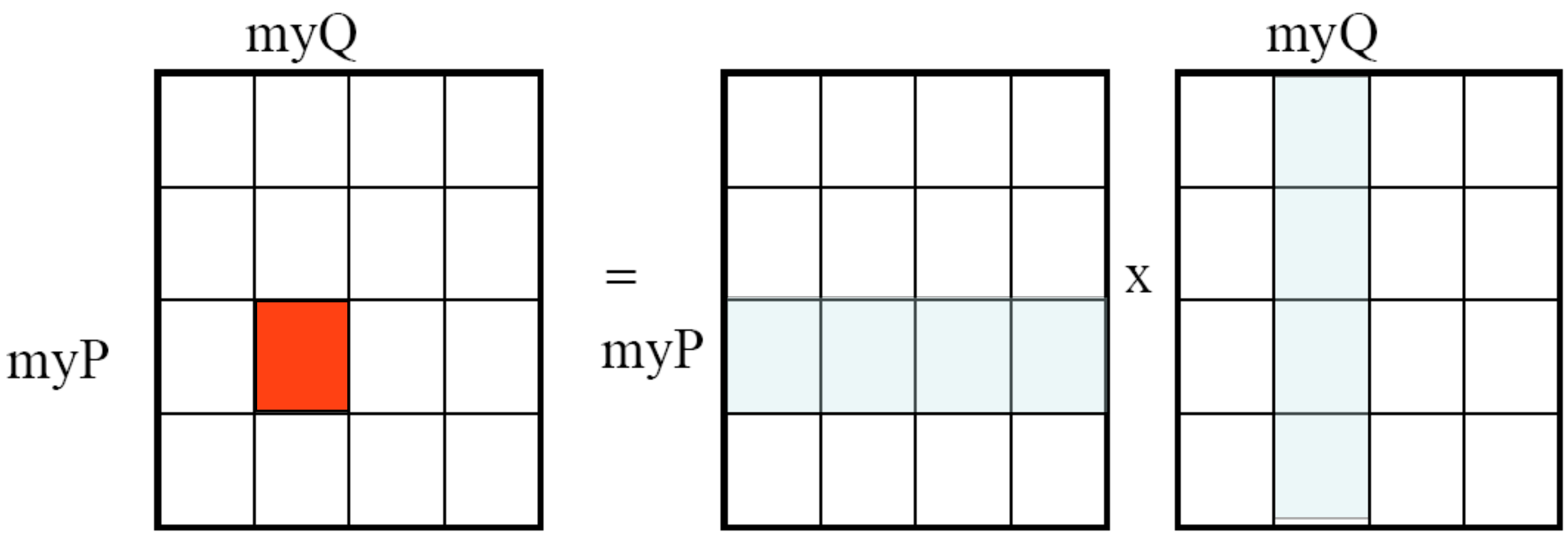
```
RecvBuff(:,1)[Left] = SendBuff(:,-1)
RecvBuff(:,-1)[Right] = SendBuff(:,1)

call notify_team(Left)
call notify_team(Right)
call wait_team(Right)
call wait_team(Left)
```

unpack RecvBuff buffer

Significant performance gain at scale!

- up to 35% for NAS MG class A on 64 processors (RTC)

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)

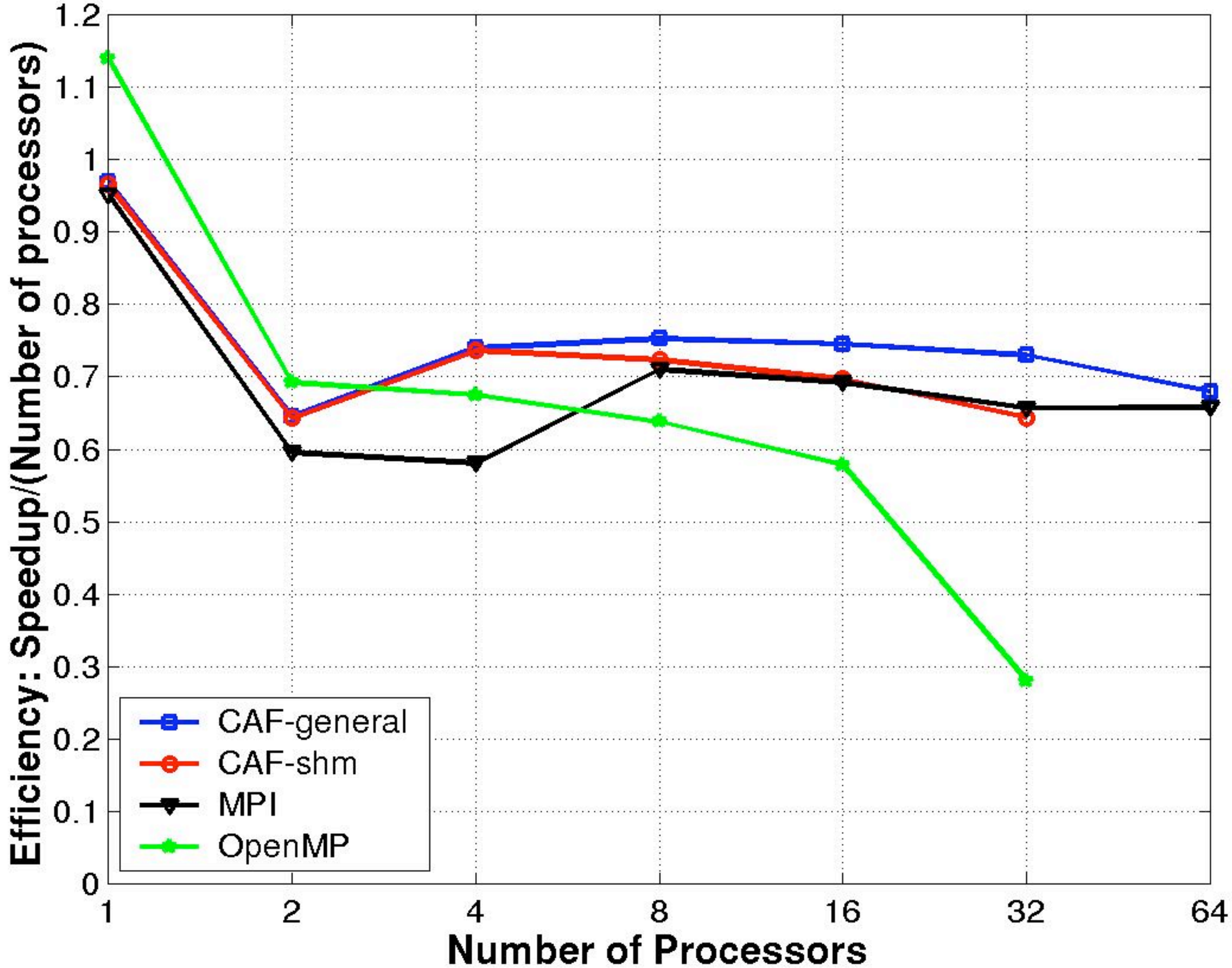
```
subroutine assemble(start, prin, ghost, neib, x)
  integer :: start(:),prin(:),ghost(:),neib(:),k1, k2, p
  real :: x(:) [*]
  call sync_all(neib)
  do p = 1, size(neib) ! Add contribs. from ghost regions
    k1 = start(p); k2 = start(p+1)-1
    x(prin(k1:k2))=x(prin(k1:k2))+x(ghost(k1:k2))[neib(p)]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

Performance Evaluation

- Platforms
  - MPP2: Itanium2+Quadrics QSNet II (Elan4)
  - RTC: Itanium2+Myrinet 2000
  - Lemieux: Alpha+Quadrics (Elan3)
  - Altix 3000

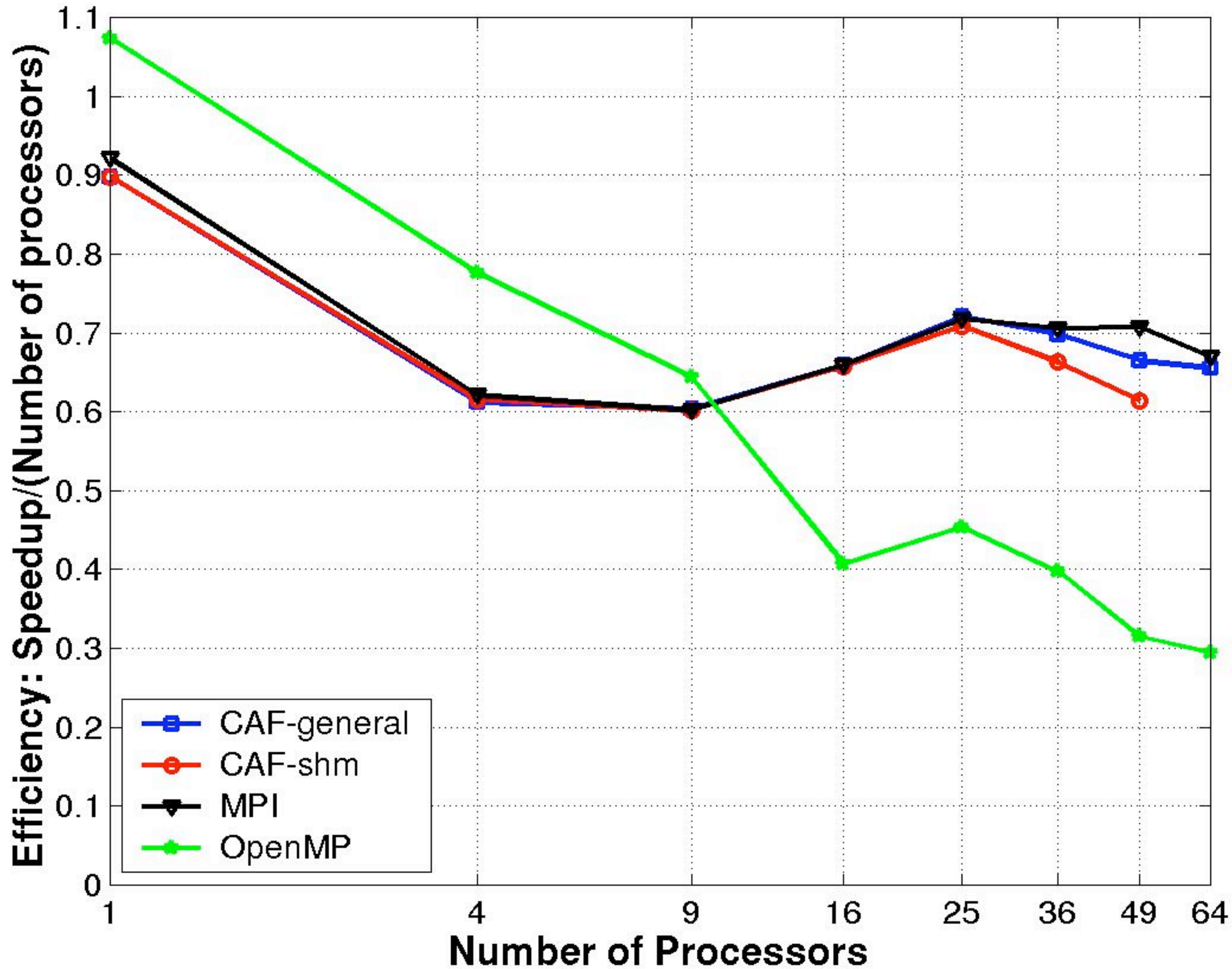
- Parallel Benchmarks (NPB v2.3) from NASA Ames
  - hand-coded MPI versions
  - serial versions
  - CAF implementation, based on the MPI version, compiled with ***`cafc`***
  - UPC implementation, based on the MPI version, compiled with the Berkeley UPC compiler (in collaboration with GWU)
  - Open MP versions (v 3.0)

NAS MG class C (512^3) on an SGI Altix 3000



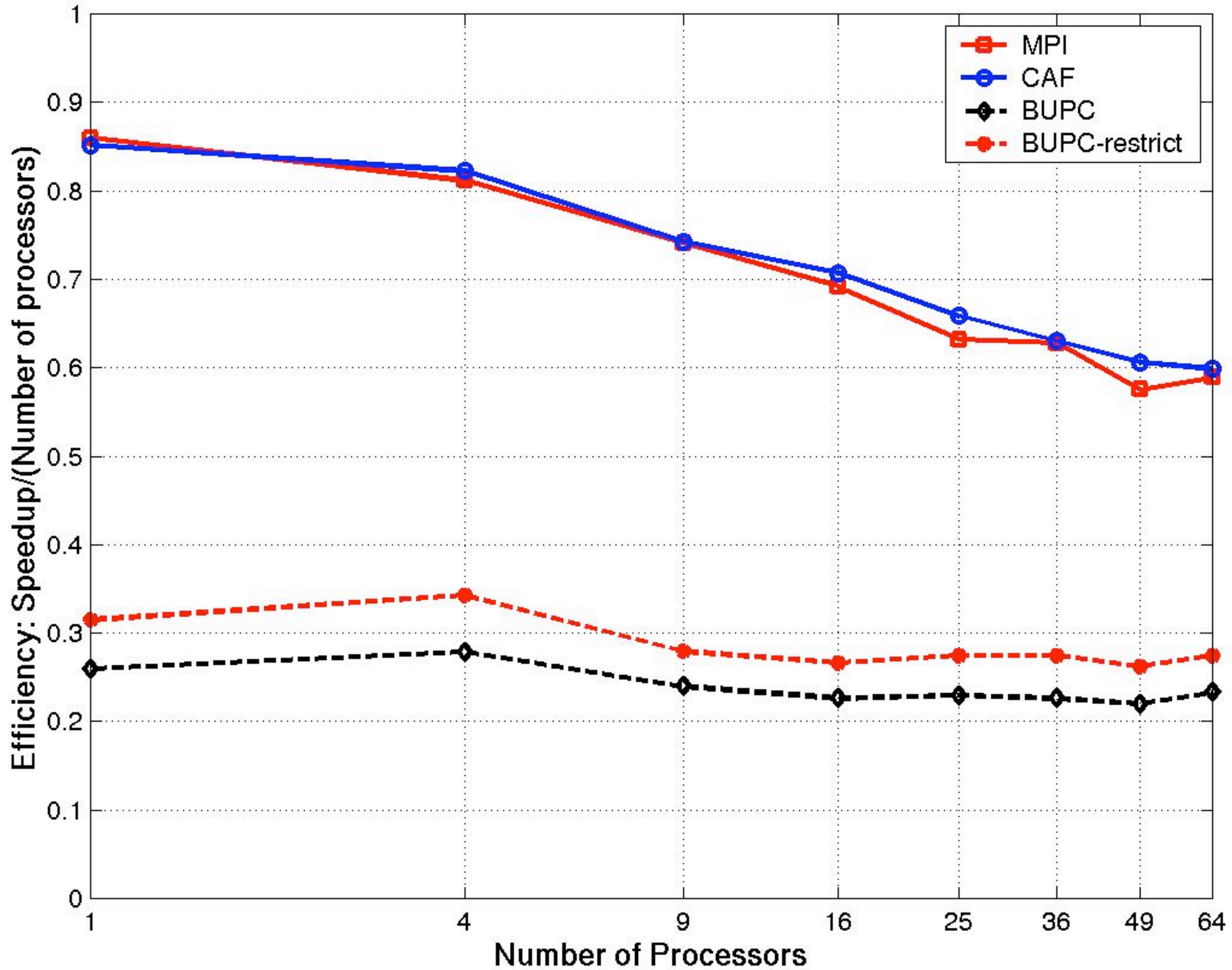
Higher is better

NAS SP class C (162^3) on an SGI Altix 3000

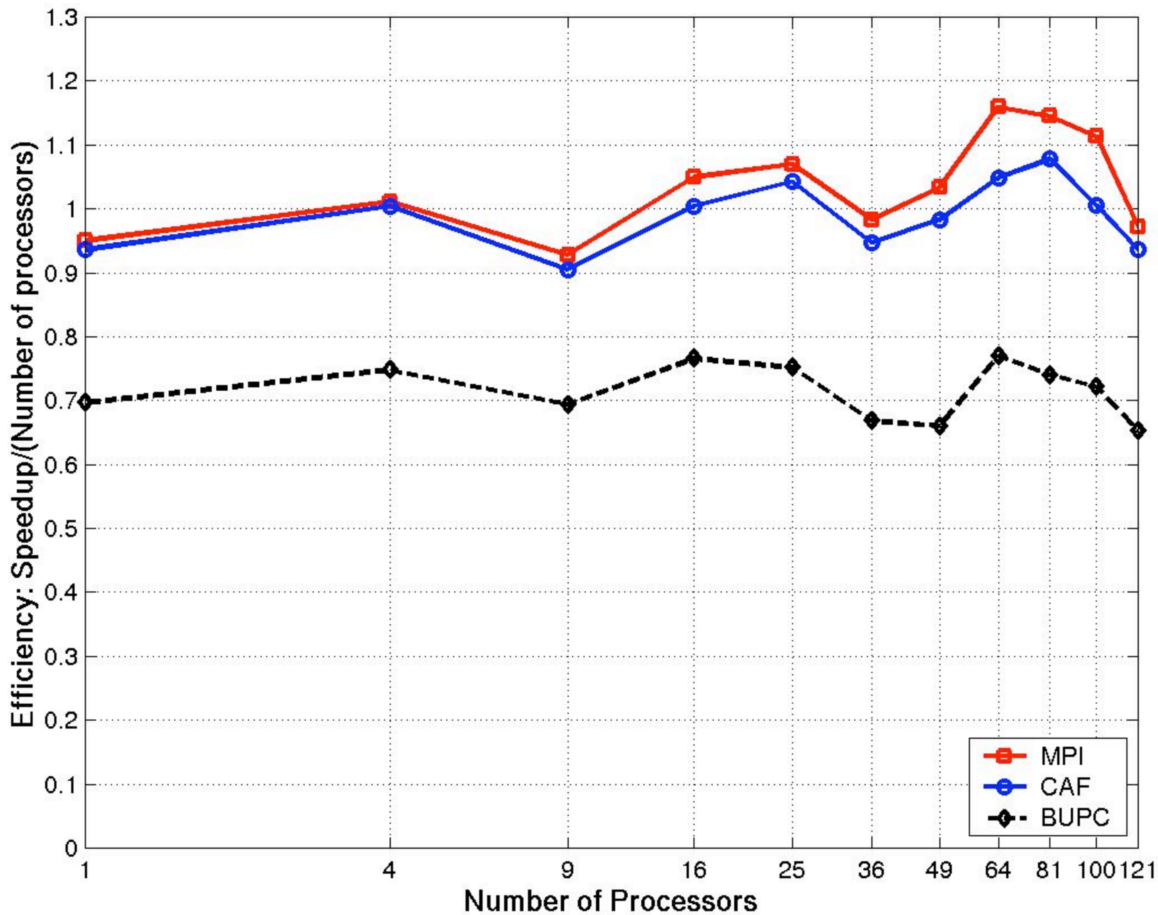


Higher is better

NAS SP class C (162^3) on Itanium2+Myrinet2000



NAS SP class C (162^3) on Alpha+Quadrics



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
      - synch_all: barriers are a heavy-handed mechanism
      - synch_team semantics required $O(n^2)$ 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
- Synch images(image list)
  - any processor can specify any list of images
- 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
  - 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

Open Questions (Part 2)

Synchronization with dynamic threading

- Barriers with dynamic threading: who participates?
- Alternatives
  - Cilk's "SYNC"
    - a SYNC in a procedure blocks until all its spawned work finishes
    - limited to rigid nested fork-join synchronization
  - X10's "FINISH" construct
    - all computation and threads inside a FINISH block must complete
    - more flexible than Cilk's model
      - an entire nested computation can complete to a single FINISH
    - FINISH is global
- Proposed approach
  - support FINISH on processor subsets (CAF teams)

Take Home Points

- CAF uses global address space abstraction
- Global address space programs can be easier than message passing with MPI
- CAF programs can be compiled for high performance on today’s scalable parallel architectures
  – match hand-coded MPI performance on both cluster and shared-memory architectures
- Amenable to compiler optimizations
- CAF language is a work in progress ...

CAF 2.0 Design Goals

- Facilitate the construction of sophisticated parallel applications and parallel libraries
- Scale to emerging petascale architectures
- Exploit multicore processors
- Deliver top performance: enable users to avoid exposing or overlap communication latency
- Support development of portable high-performance programs
- Interoperate with legacy models such as MPI
- Support irregular and adaptive applications

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

Design Features Overview:

- Participation: Teams of processors
- Organization: Topologies
- Communication: Co-dimensions
- Mutual Exclusion: Extended support for locking
- Multithreading: Dynamic processes
- Coordination: Events
- Collective Synchronization: Barriers and team-based reductions

Teams

- Partitioning and organizing images for computation with teams
  - a team is a global concept: all team members agree
  - a team is immutable once created
- Predefined team: CAF_TEAM_WORLD
  - contains all images
- Team representation?
  - want 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

```
IMAGE_TEAM team
integer rank, row
rank = this_image(CAF_TEAM_WORLD)
row = rank/q
call team_split(CAF_TEAM_WORLD, row, rank, team)
```

Topologies

- Provide a mechanism for organizing the members of a team
- Cartesian topology is a useful special case
  - multiple dimensions
  - support periodic and aperiodic boundaries
- Graph topology to support the general case
  - arbitrary connectivity between processor nodes
- Better than multiple codimensions!

Mutual Exclusion

- Built-in LOCK type

  CAF_LOCK L

  LOCK(L)

  !…use data protected by L here…

  UNLOCK(L)
- Drop global critical section

Lock Sets: Safer Multi-locking

- Big problem with locks: Deadlock
  - Results from lock acquisition cycles
- Take a cue from two-phase locking
  - Acquire all locks as one logical processing step
  - Total ordering over locks avoids cycles between processes during acquisition
- Lockset abstraction supports this idiom for programmer convenience
  - Add or remove individual locks to a runtime set
  - Acquire operation on the set acquires individual locks in canonical order

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

Safe Communication Spaces

- Event data object for anonymous pairwise coordination
- Safe synchronization space: can allocate as many events as possible
  - event sync(2)[*]
- Notify: nonblocking, asynchronous signal to an event; a pairwise fence between sender and target image
  - evnotify(sync(LEFT)[p-1))
  - evnotify(sync(RIGHT)[p+1]])
- Wait: blocking wait for notification on an event or event set
  - evwait(sync(LEFT))
- Waitany: return the ready event in an event set
  - perhaps: evwaitany(sync(1:2), readyindex)

Collective Communication

- Sync: barrier within a team
- All the standard collective operations
  - sum, product, maxloc, maxval, minloc, minval
  - any, all, count, alltoall
- Coreduce: collective communication within a team
- User-defined reductions for extensibility

Syntactic Convenience: Critical Sections

- Structured critical section construct for mutual exclusion

```
critical (Lock | Lockset)
    … ! Critical region here
end critical
```

- Impossible to miss releasing a lock
- Does not support hand-over-hand locking
- Names vs. locks: static vs. dynamic
  - Cannot implement dynamic data structures with a lock in each node if the set of locks is static!

Syntactic Convenience: Team Namespaces

- Specify a default team for data access
- Retain ability to override with explicit team specifier

```
with team (air) ! sets default team
    a(:)[1] = b(:)[2@ocean]
    ! Image 1 from the air team gets data
    ! from image 2 of the ocean team
end with
```

Implementation Status

- New source-to-source implementation of CAF 2.0 under way
- New front-end compiler based on Rose
- Compiler and runtime library implementation in progress
  - Thinnest possible runtime for maximal performance
- GasNet substrate for interprocess communication in the runtime
- Clever strategy for source-to-source compilation of coarrays
  - use Cray pointers to initialize Fortran 90 pointers
  - no need to know anything about dope vector format
  - generated code can use any compiler that supports Cray pointers

End Notes

- Our Feb 2008 critique of coarray support in the Fortran 2008 draft standard may be found online at:
  - http://www.j3-fortran.org/doc/meeting/183/08-126.pdf