SciDAC
Scientific Discovery through Advanced Computing

RICE

Programming Models for Scientific Computing on Leadership Computing Platforms:

The Evolution of Coarray Fortran

John Mellor-Crummey
Department of Computer Science
Rice University

Acknowledgments

- CAF
  - Yuri Dotsenko
  - Cristian Coarfa
  - Bill Scherer
  - Laksono Adhianto
  - Guohua Jin
  - Fengmei Zhao

- HPF
  - Guohua Jin
  - Daniel Chavarria-Miranda
  - Vikram Adve

Outline

- Challenges for parallel languages
- A bit of parallel Fortran history
- Coarray Fortran, circa 1998 (CAF98)
- Assessment of CAF98
- A look at the emerging Fortran 2008 standard
- Recommendations for moving forward
- Open issues

Challenges for Parallel Languages

To succeed, a parallel programming language must …

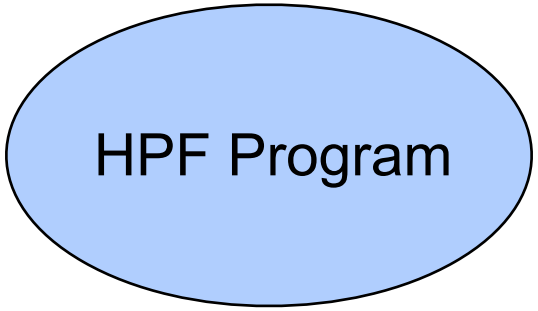
- be ubiquitous
  - multicore laptops
  - clusters on site
  - leadership-class machines at national centers
- be expressive
  - arbitrary algorithms
  - complex data structures
  - sophisticated parallelizations
- be productive
  - easy to write
  - easy to read and maintain
  - easy to reuse
- leverage legacy code: must be interoperable
- have a promise of future availability and longevity
- be supported by tools
- be efficient

1990s Vision: The Compiler was King

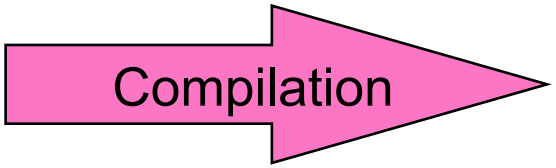
High Performance Fortran

Partitioning of data drives partitioning of computation, communication, and synchronization

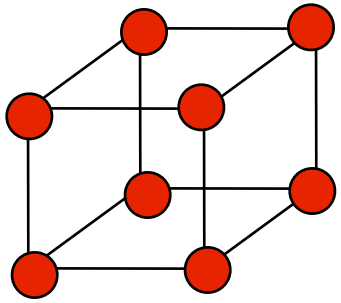
Fortran program
+ data partitioning



Partition computation
Insert communication
Manage storage



Same answers as sequential program

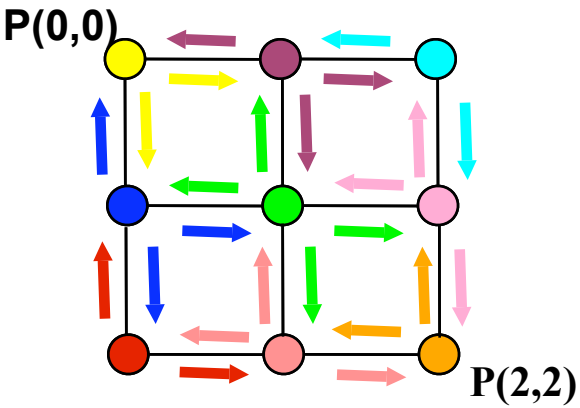


Parallel Machine

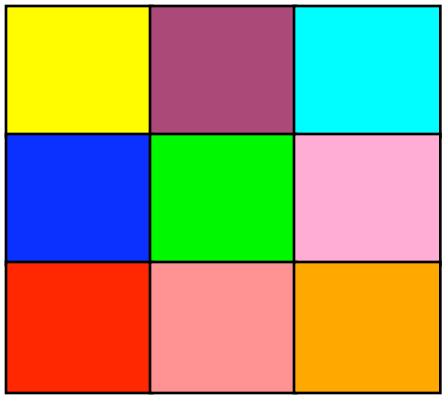
Example HPF Program

```
CHPF$ processors P(3,3)
CHPF$ distribute A(block, block) onto P
CHPF$ distribute B(block, block) onto P
```

```
      DO i = 2, n - 1
        DO j = 2, n - 1
            A(i,j) = .25 *(B(i-1,j) + B(i+1,j)+
                           B(i,j-1) + B(i,j+1))
```



Processors



Data for A, B

(BLOCK,BLOCK)distribution

Compiling HPF

- Partition data
  - follow user directives
- Select mapping of computation to processors
  - co-locate computation with data
- Analyze communication requirements
  - identify references that access off-processor data
- Partition computation by reducing loop bounds
  - schedule each processor to compute on its own data
- Insert communication
  - exchange values as needed by the computation
- Manage storage for non-local data

Productive Parallel 1D FFT ($n = 2^k$)

```
subroutine fft(c, n)
    implicit complex(c)
    dimension c(0:n-1), irev(0:n-1)
!HPF$ processors p(number_of_processors())
!HPF$ template t(0:n-1)
!HPF$ align c(i) with t(i)
!HPF$ align irev(i) with t(i)
!HPF$ distribute t(block) onto p
    two_pi = 2.0d0 * acos(-1.0d0)
    levels = number_of_bits(n) - 1
    irev = (/ (bitreverse(i,levels), i= 0, n-1) /)
    forall (i=0:n-1) c(i) = c(irev(i))
    do l = 1, levels                    ! --- for each level in the FFT
       m = ishft(1, l)
       m2 = ishft(1, l - 1)
       do k = 0, n - 1, m               ! --- for each butterfly in a level
          do j = k, k + m2 - 1          ! --- for each point in a half bfly
             ce = exp(cmplx(0.0,(j - k) * -two_pi/real(m)))
             cr = ce * c(j + m2)
             cl = c(j)
             c(j) = cl + cr
             c(j + m2) = cl - cr
          end do
       end do
    enddo
 end subroutine fft
```

partitioning the k loop is subtle: driven by partitioning of j loop

partitioning the j loop is driven by the data accessed in its iterations

1D FFT Support Code

```
pure function number_of_bits(i)
   number_of_bits = 0
   i_tmp = i
   do while (i_tmp .gt. 0)
      number_of_bits = number_of_bits + 1
      i_tmp = ishft(i_tmp, -1)
   end do
 end function number_of_bits
```

```
pure function bitreverse(i, n)
   interchange_bits(ival,imask,ishift) =  &
       ior(ishft(iand(ival, imask), -ishift),  &
       ishft(iand(ival, not(imask)), ishift))
   parameter (mask32 = Z'FFFFFFFF00000000')
   parameter (mask16 = Z'FFFF0000FFFF0000')
   parameter (mask8   = Z'FF00FF00FF00FF00')
   parameter (mask4   = Z'F0F0F0F0F0F0F0F0')
   parameter (mask2   = Z'CCCCCCCCCCCCCCCC')
   parameter (mask1   = Z'AAAAAAAAAAAAAAAA')
   integer bitreverse

   itmp = interchange_bits(i,      mask32, 32)
   itmp = interchange_bits(itmp, mask16, 16)
   itmp = interchange_bits(itmp, mask8,   8)
   itmp = interchange_bits(itmp, mask4,   4)
   itmp = interchange_bits(itmp, mask2,   2)
   itmp = interchange_bits(itmp, mask1,   1)
   bitreverse = ishft(itmp, n - 64)
end function bitreverse
```

Some Lessons from HPF

Everything matters for good performance!

- Good data partitionings are essential for good parallelizations
  - e.g. BLOCK partitionings inferior to multipartitionings for line-sweeps
- Excess communication undermines scalability
  - both frequency and volume must be right
    - coalesce communication sets for multiple references
      - 41% lower message volume, 35% faster: NAS SP @ 64 procs
    - partially replicate computation to reduce communication
      - 66% lower message volume, 38% faster: NAS BT @ 64 procs
- User guidance is an invaluable supplement to analysis
  - e.g. HPF/JA directives to control communication; parallel loops
- Complex things are possible; realizing them can be challenging
  - RandomAccess, FFT
- If the compiler can't deliver, you're out of luck!

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

- Challenges for parallel languages
- A bit of parallel Fortran history
- 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
- Recommendations for moving forward
- Open issues

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, PIM

SPMD Execution Model

- The number of images is fixed and each image has its own index, retrievable at run-time:
  - 1 ≤ num_images()
  - 1 ≤ this_image() ≤ 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

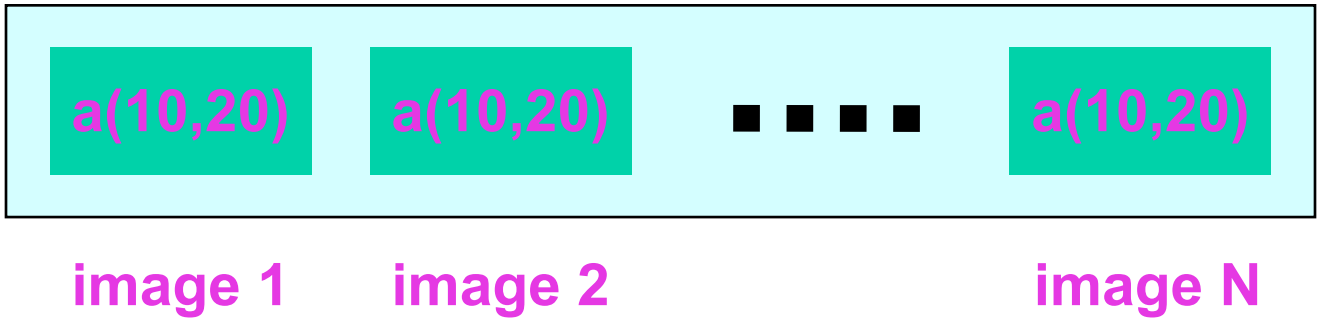
- Syntax is a simple parallel extension to Fortran 90
  - it uses normal rounded brackets ( ) to point to data in local memory
  - it uses square brackets [ ] to point to data in remote memory
- Co-arrays can be accessed from any image
- Co-arrays are *symmetric*
- Co-arrays can be SAVE, COMMON, MODULE,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

Examples of Coarray Declarations

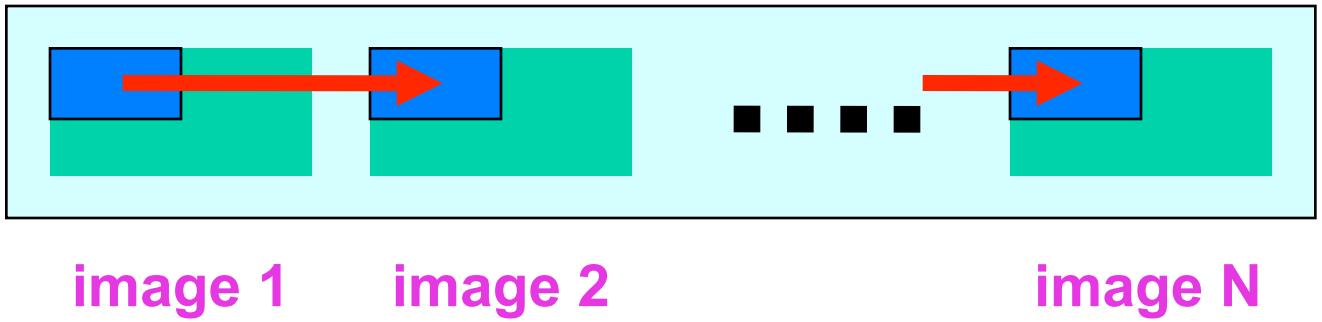
```
real :: array(N, M)[*]
integer ::scalar[*]

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)[*]
```




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



Flavors of Remote accesses

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

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/)

Processor Space 3

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

x[-5:-2,0:*] this_image() = 15 this_image(x) = (/-3, 3/)

Processor Space 4

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

x[0:1,0:*] this_image() = 15 this_image(x) =(/0,7/)

CAF98 Synchronization Primitives

- sync_all()
- sync_team(team, [wait])
- notify_team(team)
- wait_team(team)
- 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)

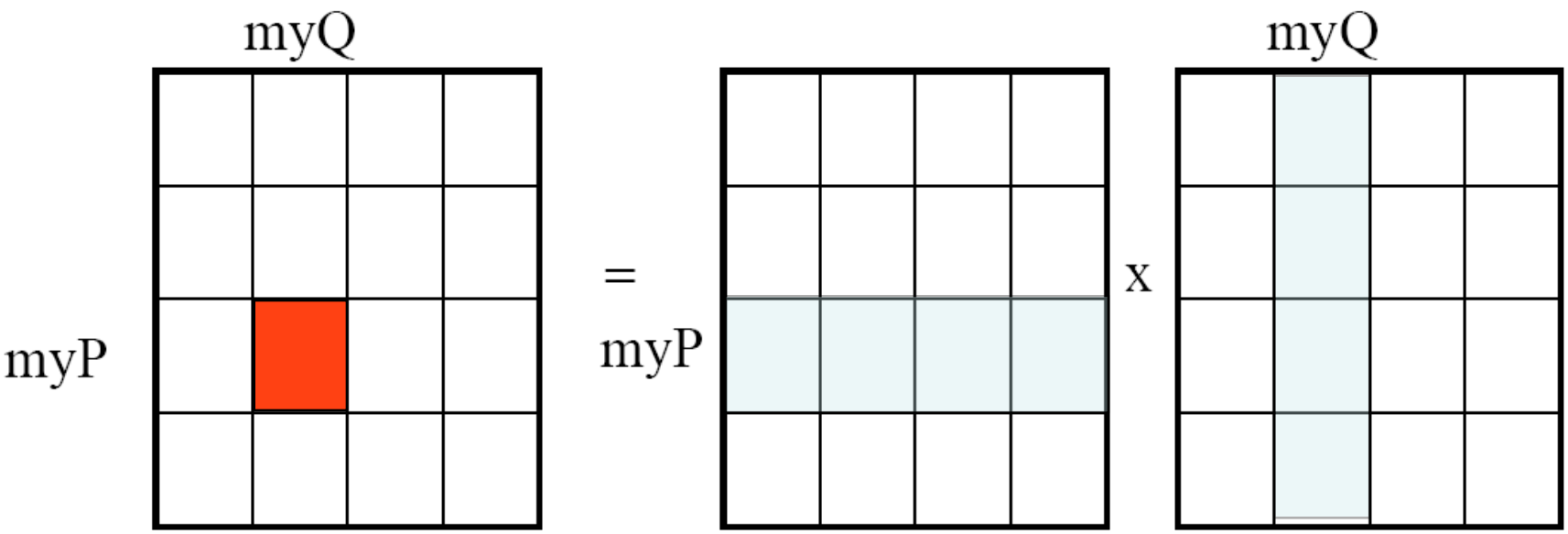
call notify_team(Left)
call notify_team(Right)
call wait_team(Right)
call wait_team(Left)

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)

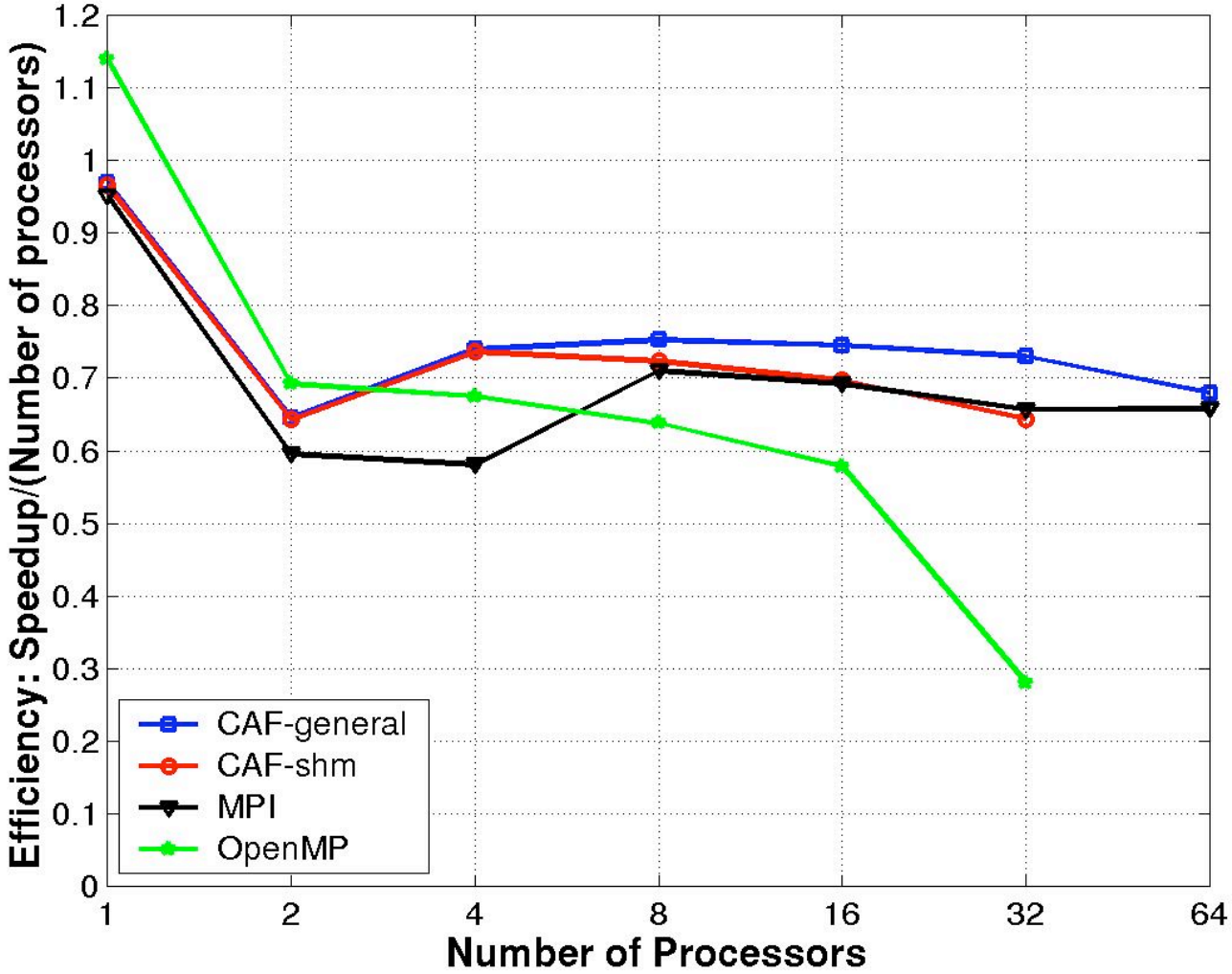
```
subroutine assemble(start, prin, ghost, neib, x)
  integer :: start(:),prin(:),ghost(:),neib(:),k1, k2, p
  real :: x(:) [*]
  call sync_all(neib)
  do p = 1, size(neib) ! Add contribs. from ghost regions
    k1 = start(p); k2 = start(p+1)-1
    x(prin(k1:k2))=x(prin(k1:k2))+x(ghost(k1:k2))[neib(p)]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

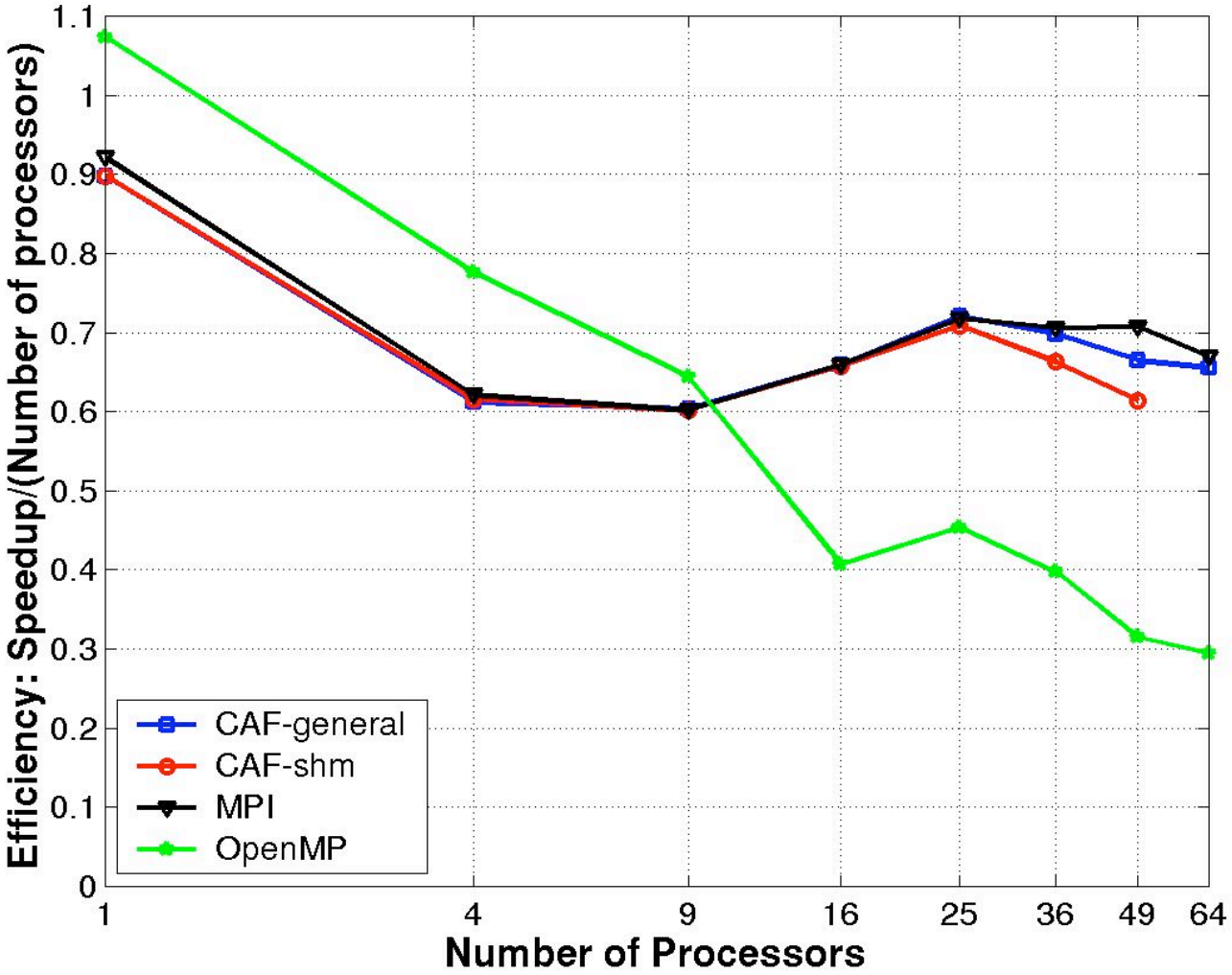
- Parallel Benchmarks (NPB v2.3) from NASA Ames
  - hand-coded MPI versions
  - serial versions
  - CAF implementation, based on the MPI version, compiled with ***`cafc`***
  - UPC implementation, based on the MPI version, compiled with the Berkeley UPC compiler (in collaboration with GWU)
  - Open MP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

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



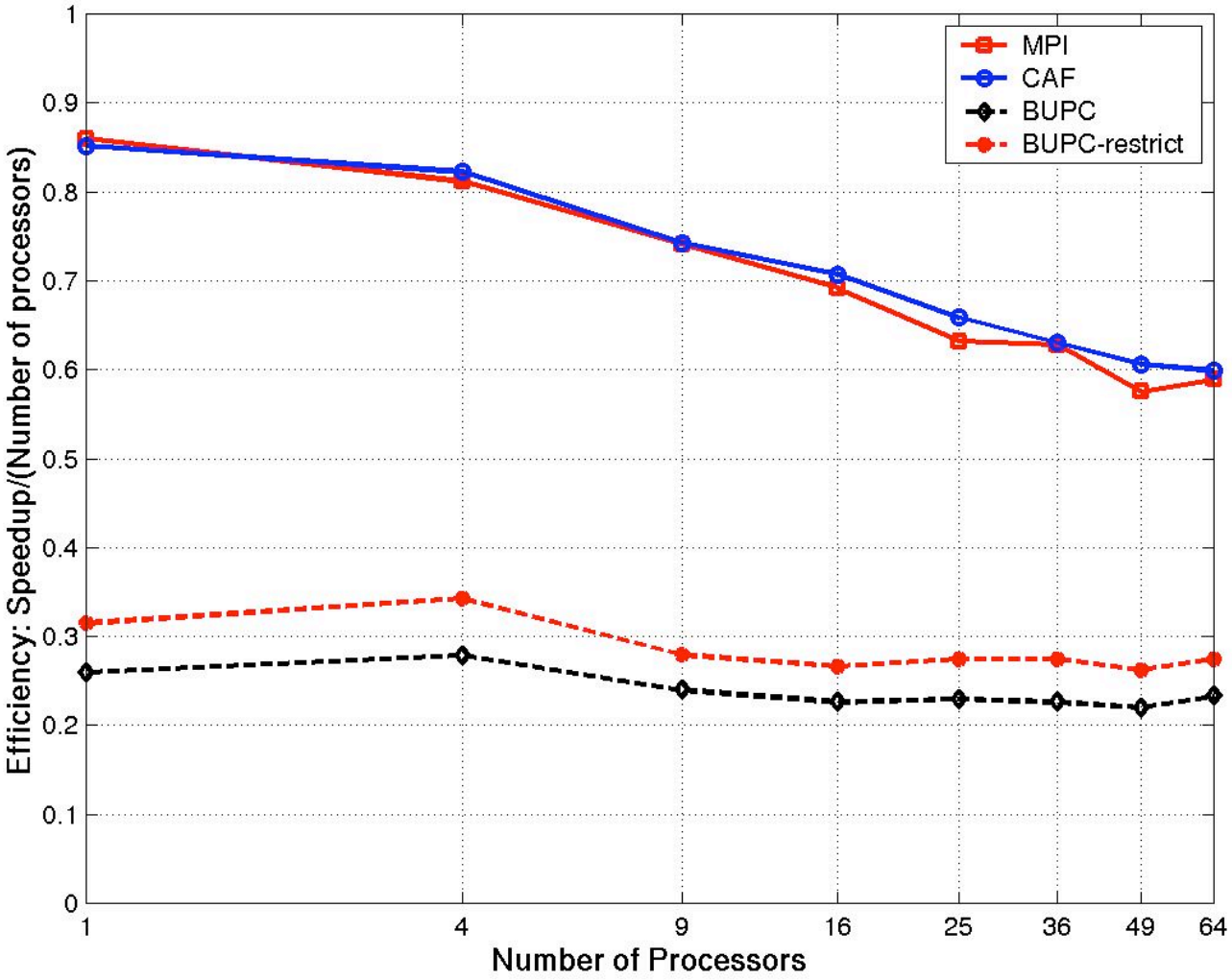
Higher is better

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

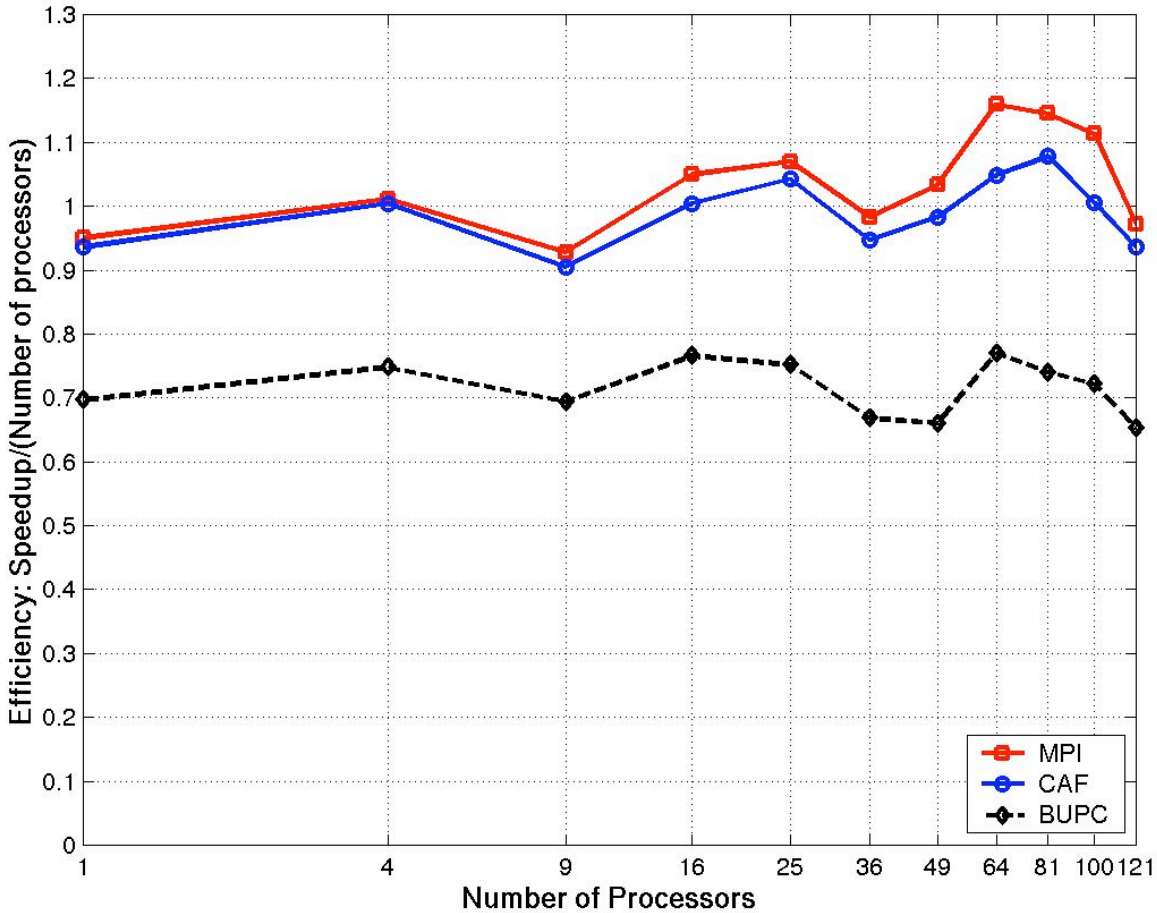


Higher is better

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, Feb. 2008

Coarray features being considered for inclusion

- Single and multidimensional coarrays
- Collective allocation of coarrays to support a symmetric heap
- Named critical sections for structured mutual exclusion
  - improves over single critical section in CAF98
- Synch_all and synch_team
  - pre-arrange teams with form_team for efficient synchronization
- Pairwise non-blocking synchronization with notify/query
  - support communication/computation overlap
- Collective communication intrinsics for convenience & portability

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
  - current scalable parallel systems lack h/w shared memory
    - e.g. clusters, Blue Gene, Cray XT, SiCortex
    - big hurdle for third-party compiler vendors to target scalable systems

2. Adequate Support for Libraries?

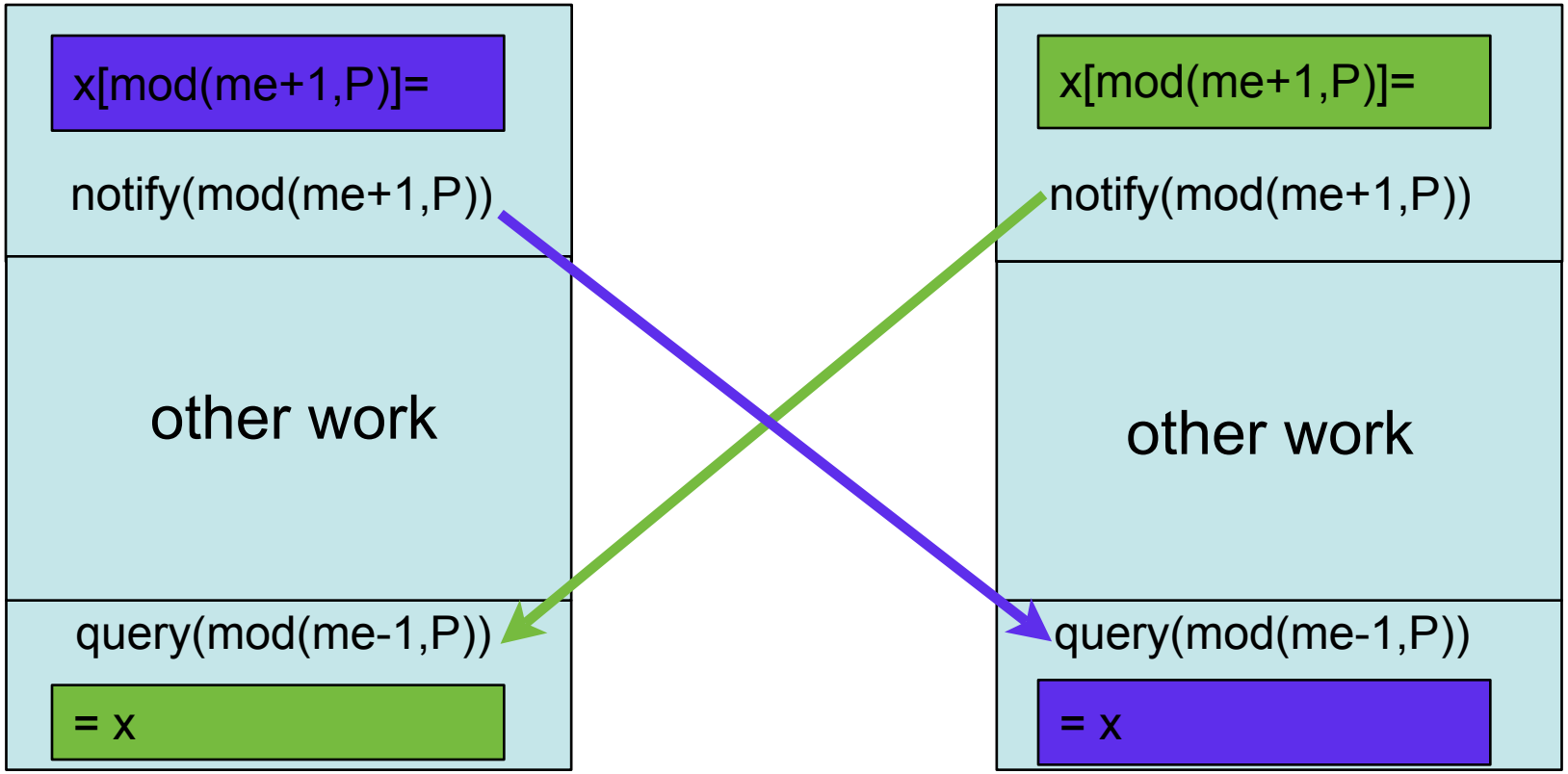
Lessons from MPI: Library needs [MPI 1.1 Standard]

- **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

Lack of a Safe Communication Space (Part 1)

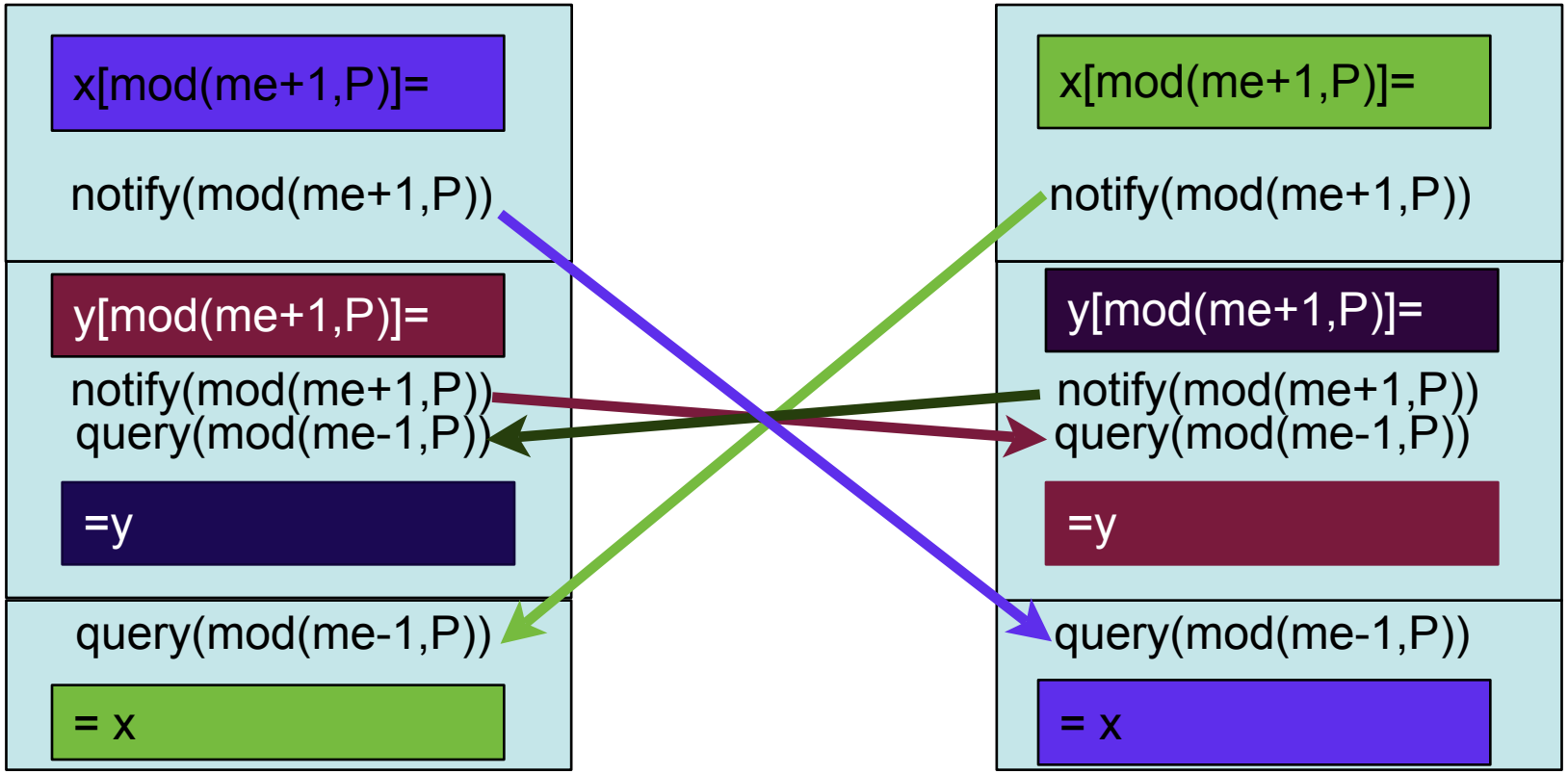
- F2008 has a single synchronization channel between process pairs
- Multiple channels are essential for library encapsulation

Consider the following



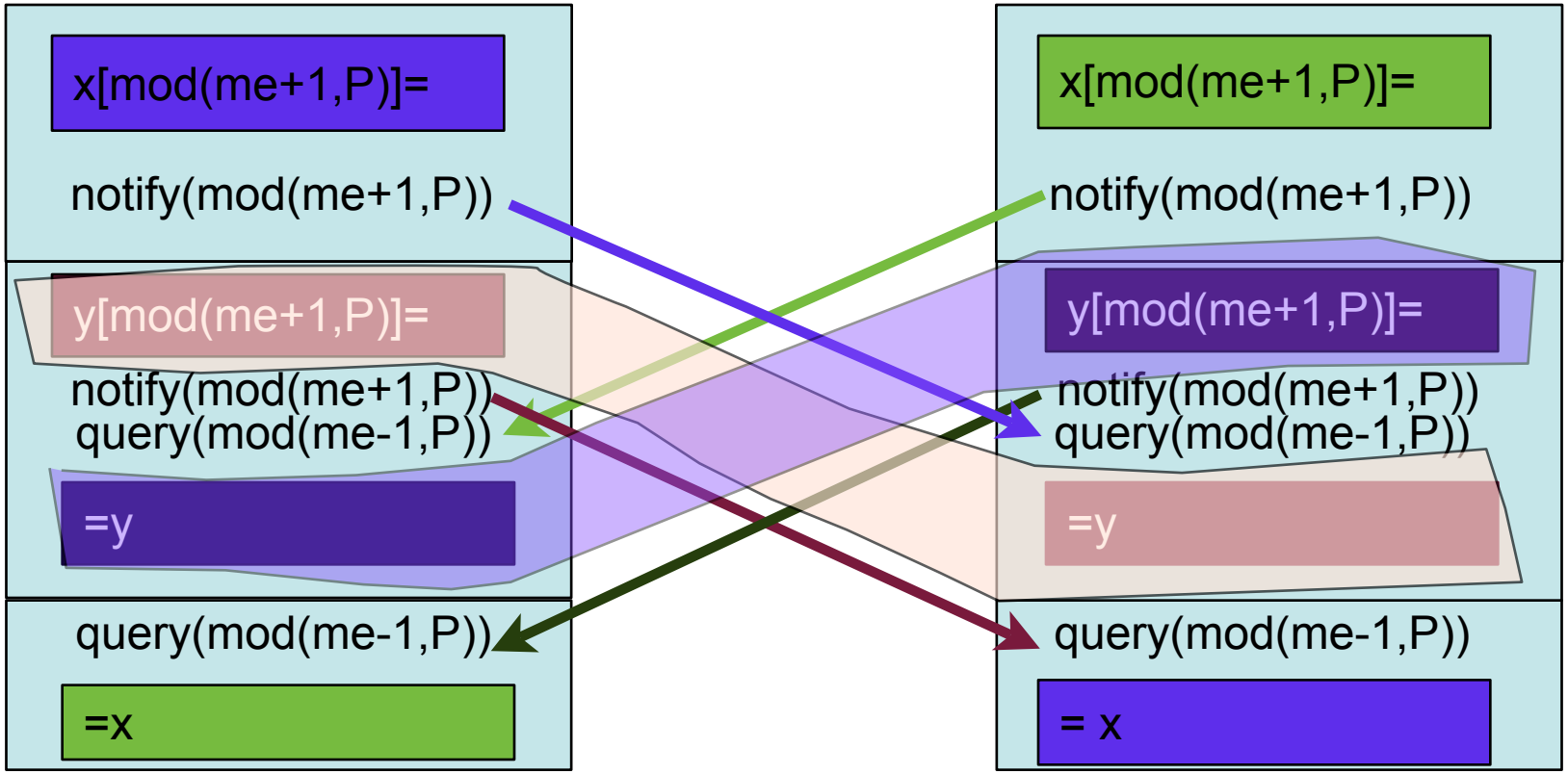
Lack of a Safe Communication Space (Part 2)

What if “other work” was synchronized?



Lack of a Safe Communication Space (Part 3)

Lack of encapsulation leads to data races



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
  - makes it harder 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

Lack of a Means to Extend Collectives

- A variety of collective reduction subroutines
  - sum, product, maxloc, maxval, minloc, minval
  - any, all, count
- Shortcomings
  - no support for user-defined reduction operators
  - no support for scan reductions
  - missing feature: all-to-all

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
    - named critical sections overly limit concurrency
      - only one process active per static name
    - unreasonable to expect users to "roll their own"
- As defined, F2008 useful for halo exchanges on dense arrays

4. Adequate Support for Writing Fast Code?

- Lack of support for hiding synchronization latency
  - F2008 notify/query specification precludes overlap
  - 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
  - longer term: conditional ATOMIC operations based on transactional memory
- Enhanced support for collectives
  - add support for user-defined reduction operators
  - add support for scan reductions
  - add support for all-to-all operations
- Add multiversion variables
  - simplify producer/consumer interactions in a shared memory world
- Add global pointers to enable flat access to data

Open Questions (Part 1)

- Atomic operations: what is the best approach for scalable systems?
  - X10 approach: similar to k-way compare-and-swap?
  - transactional memory?
    - Bocchino, Adve, Chamberlain: Cluster STM [PPoPP '08]
      - weak atomicity; transactional and non-transactional data in a phase
      - flexible distributed atomic operations
      - function shipping for latency avoidance
      - shortcomings: rigid SPMD model (1 task per PE)

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 ...