
Co-Array Fortran and High Performance Fortran

John Mellor-Crummey

**Department of Computer Science
Center for High Performance Software Research
Rice University**

The Problem

- **Petascale systems will be hard to program efficiently**
 - vast numbers of nodes (memory systems)
 - hybrid parallelism
 - exotic communication networks
 - long latencies both to memory and remote nodes
- **Need simpler, more abstract, and convenient ways of conceptualizing, expressing, debugging, and tuning programs**
- **Programming models help harness large-scale systems**
 - enhance programmer productivity through abstraction
 - manage platform resources to deliver performance
 - provide standard interface for platform portability
- **Models trade off convenience, expressivity, and performance**

To Succeed, Parallel Programming Models Must ...

- **Be ubiquitous**
 - laptop
 - cluster at your site
 - leadership-class machines at national centers
- **Be expressive**
- **Be productive**
 - easy to write
 - easy to read and maintain
 - easy to reuse
- **Have a promise of future availability and longevity**
- **Be efficient**
- **Be supported by tools**

A Tale of Two Programming Models

Issues and Ongoing Work

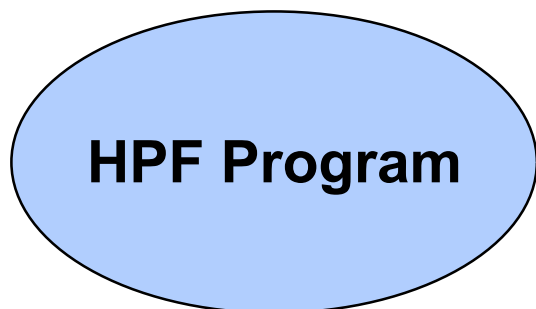


- **High Performance Fortran**
- **Co-array Fortran**

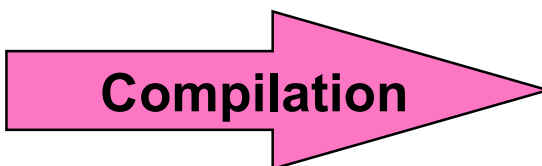
High Performance Fortran

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

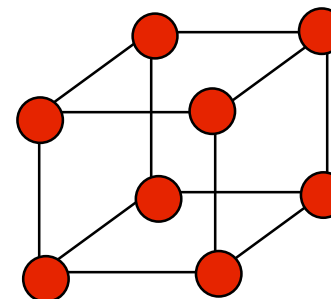
*Fortran program
+ data partitioning*



*Partition computation
Insert communication
Manage storage*



*Same answers as
sequential program*

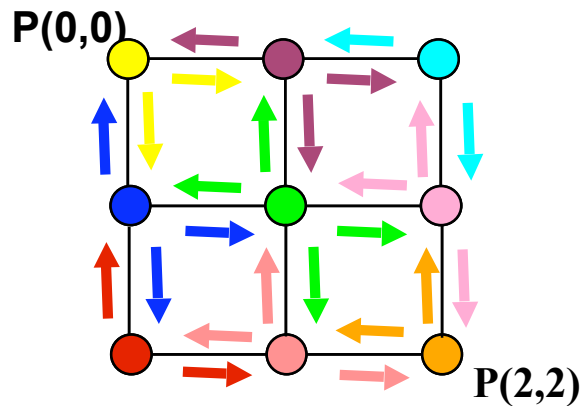


Parallel Machine

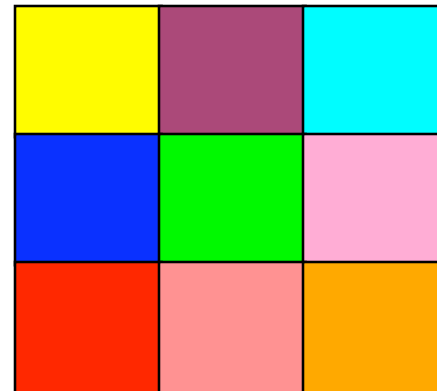
Example HPF Program

```
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
- **Map 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**

Formal Compilation Framework (Rice dHPF)

3 types of Sets

Data
Iterations
Processors

3 types of Mappings

Layout: data \longleftrightarrow processors
Reference: iterations \longleftrightarrow data
CompPart: iterations \longleftrightarrow processors

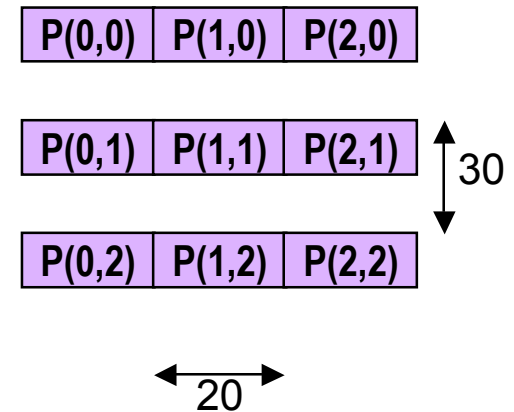
- **Representation**
 - integer tuples with Presburger arithmetic (universal & existential quantifiers + linear inequalities with constant coefficients + logical operators) for constraints
- **Analysis: Use set equations to compute set(s) of interest**
 - iterations allocated to a processor
 - communication sets
- **Code generation: Synthesize loops from set(s), e.g.**
 - parallel (SPMD) loop nests
 - message packing and unpacking

Symbolic Sets?

```

processors P(3,3)
distribute A(block, block) onto P
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) )
  
```

data / loop partitioning

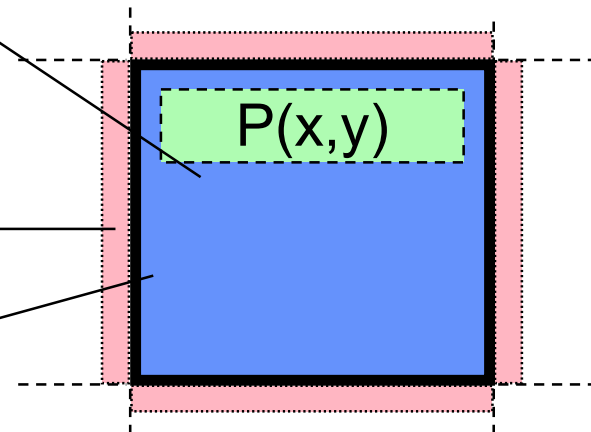


Local section for P(x,y)
(and iterations executed)

$$\{ [i, j]: 20x + 2 \leq i \leq 20x + 19 \\ \& 30y + 2 \leq j \leq 30y + 29 \}$$

Non-local data
accessed

Iterations that access
non-local data



Analyzing Programs with Integer Sets

```
real A(100)
distribute A(BLOCK) on P(4)
do i = 1, N
  ... = A(i-1) + A(i-2) + ...           ! ON_HOME A(i-1)
enddo
```

symbolic N

Layout := { [pid] -> [i] : 25 * pid + 1 ≤ i ≤ 25 * pid + 25 }

Loop := { [i] : 1 ≤ i ≤ N }

CPSubscript := { [i] → [i-1] }

RefSubscript := { [i] → [i-2] }

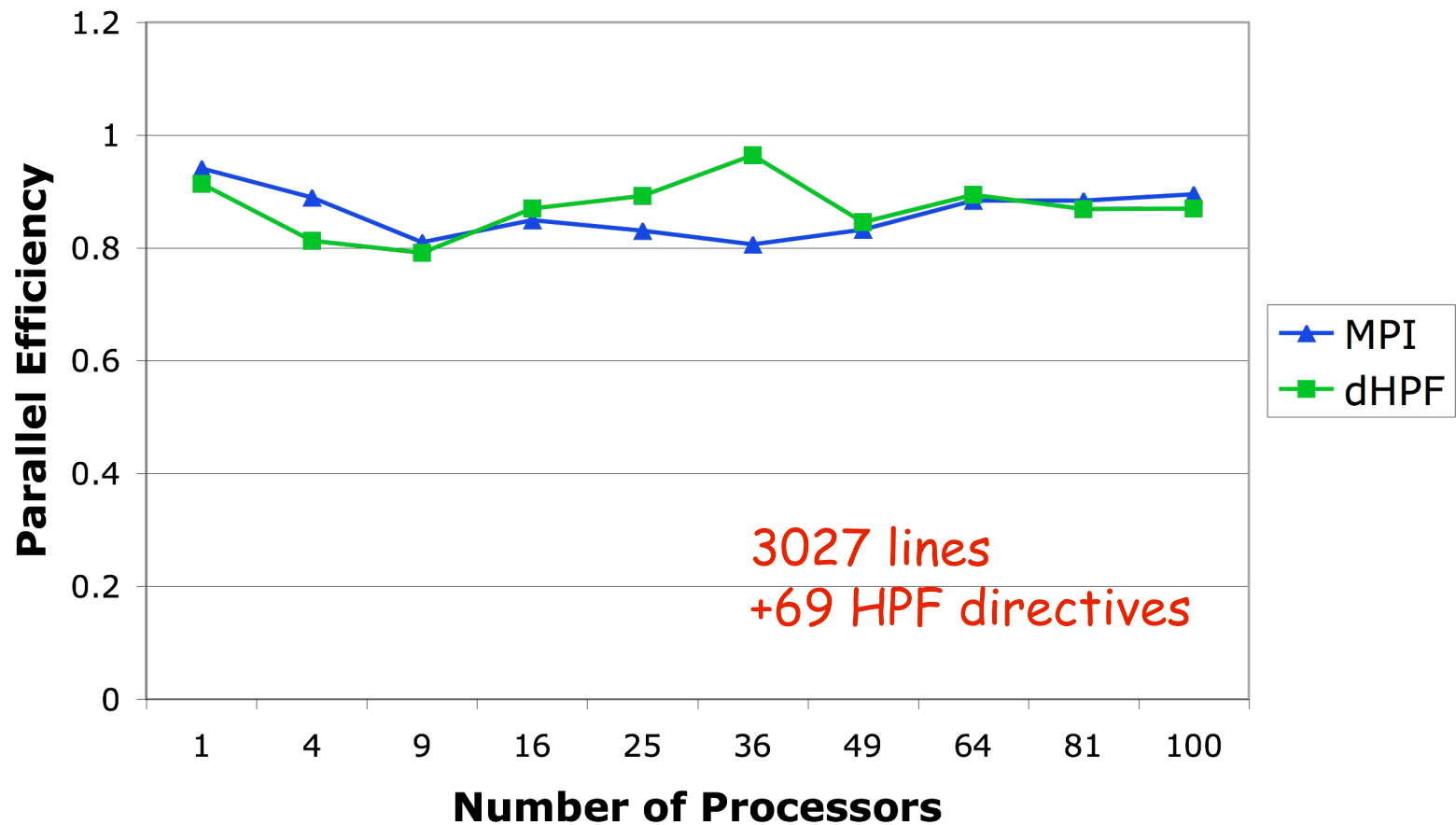
CompPart := (Layout ◦ CPSubscript⁻¹) ∩ Loop

DataAccessed = CompPart ◦ RefSubscript

NonLocal Data Accessed = DataAccessed - Layout

Performance Using the Rice dHPF Compiler

Efficiency NAS SP class 'C'



IMPACT-3D

HPF application: Simulate 3D Rayleigh-Taylor instabilities in plasma fluid dynamics using TVD

- **Problem size: 1024 x 1024 x 2048** 1334 lines
- **Compiled with HPF/ES compiler** +45 HPF directives
 - 7.3 TFLOPS on 2048 ES processors ~ 45% peak
- **Compiled with dHPF on PSC's Lemieux**

# procs	relative speedup	GFLOPS	% peak
128	1.0	46.4	18.1
256	1.94	89.9	17.6
512	3.78	175.5	17.4
1024	7.58	352.0	17.3

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
```

1D FFT: Partitioning Work

```
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

ON HOME c(j)

ON HOME c(j + m2)

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

1D FFT: Generating Communication

```
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
```

All communication can occur here

ON HOME $c(j)$

ON HOME $c(j + m2)$

1D FFT: Eliminating Redundant Work

```
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
```

ripe for space-time tradeoff
as well as strength reduction

Current Code Generation Challenges

- **Symbolically strided iteration spaces**
- **Efficient management of non-local storage**
- **Overlapping communication and computation**
- **Top-quality code for inner loops**

A Tale of Two Programming Models

Issues and Ongoing Work

- High Performance Fortran
- 👉 Co-array Fortran

Co-Array Fortran (CAF)

- **Explicitly-parallel extension of Fortran 95**
 - defined by Numrich & Reid
- **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
 - communication
 - synchronization
- **Suitable for mapping to a range of parallel architectures**
 - shared memory, clusters, hybrid

CAF Programming Model Features

- **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
- **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_mem` – a memory fence
 - `notify_team(team)`, `wait_team(team)`, ... – point-to-point
- **Asymmetric dynamic allocation of shared data**
- **Weak memory consistency**

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_all(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)]
  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
```

Enhancing CAF

- **Expressiveness**
- **Programmability**
- **Latency hiding**
- **Compiler-based optimization**
- **Performance**

Emerging Approaches

- **Communication topologies**
- **Multi-version variables**
- **Distributed multithreading**

CAF Co-shapes

- integer `a(10,10)` `[2,*]`
- **Number of images is 6**

<code>a(10,10)[1, 1]</code>	<code>a(10,10)[1, 2]</code>	<code>a(10,10)[1, 3]</code>
<code>a(10,10)[2, 1]</code>	<code>a(10,10)[2, 2]</code>	<code>a(10,10)[2, 3]</code>

Shortcomings of Multi-dimensional Co-shapes

- **Limited expressiveness**
 - only Cartesian topology without periodic boundaries for all images
 - programmers manually implement other topologies
 - lack processor subsets, e.g. for coupled applications
- **Programming complexity**
 - “global” coordinates; explicit index management by programmers
 - “holes” require special handling
- **Performance**
 - no support for collective communication on subgroups
 - difficult to analyze and optimize communication patterns

CAF co-shape example
`integer a(10,10) [2,*]`

Co-spaces

- **Goals**
 - increase expressiveness
 - beyond Cartesian grids without periodic boundary conditions
 - reduce programming complexity
 - abstract away details of managing communication partners
 - organize images into groups
 - processor subsets and collective operations
 - facilitate compiler analysis and optimization
- **Approach: co-space abstraction**
 - types: Group, Cartesian, Graph
 - default co-space: `CAF_WORLD`

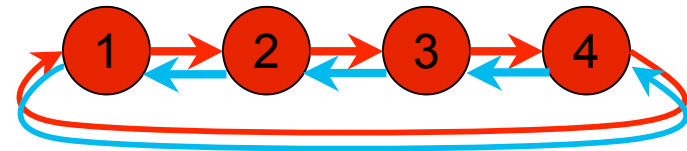
Inspired by MPI communicators and process topologies

Analyzing Communication on Co-spaces

- **Goals**
 - understand data movement and synchronization
 - make communication and synchronization efficient
- **Analysis questions**
 - which images perform communication?
 - which image is the target of each communication?

Exploiting Co-space Structure

```
barrier(cs)
axis = 1
a[successor(cs, axis, 1)] = x
barrier(cs)
```



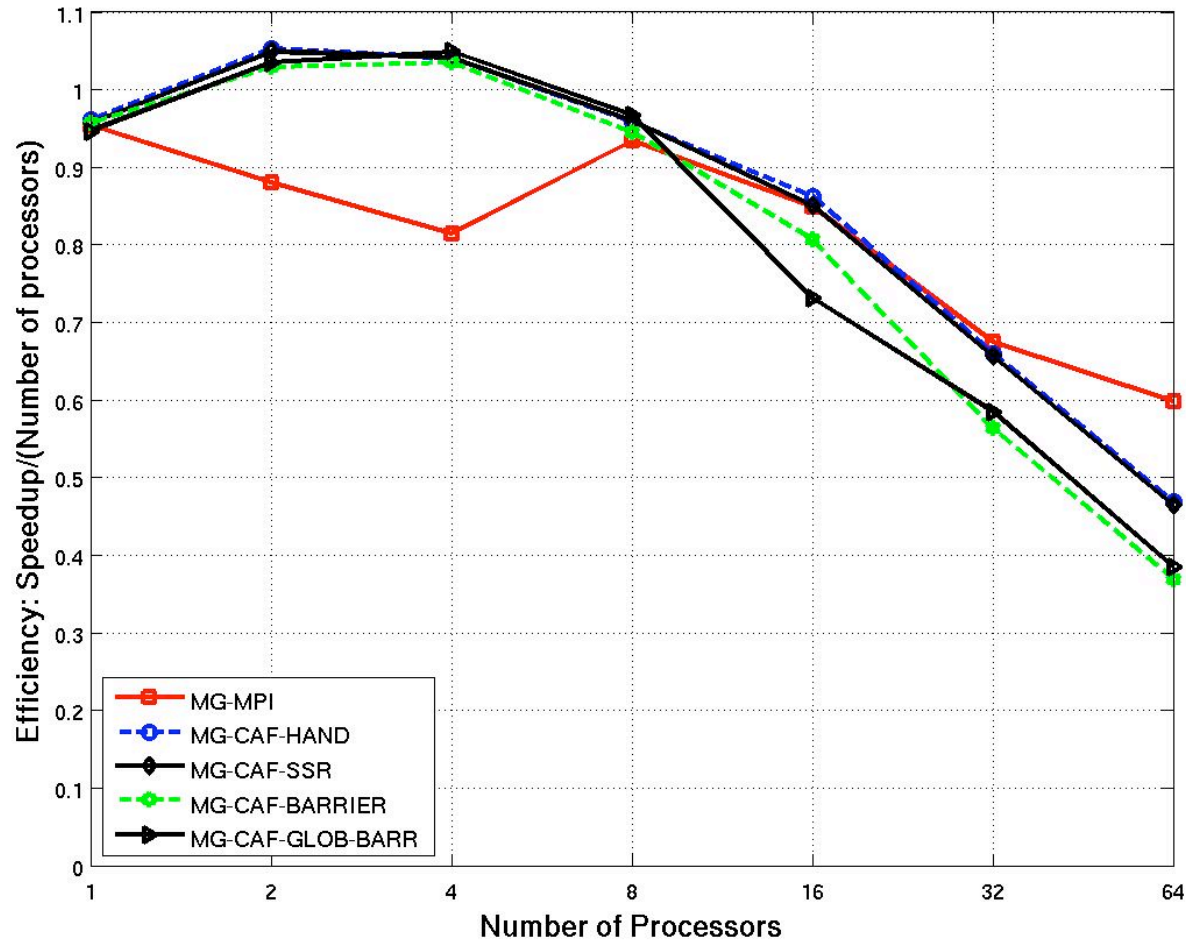
successor(cs, 1, 1)

predecessor(cs, 1, 1)

- **Co-space textual barriers**
 - inspired by Titanium's global textual barriers
- **Co-space single-valued expressions**
 - inspired by Titanium's `single`
 - analyze co-space group control flow
- **Co-space topology structure**
 - identify the source and target of each GET/PUT

SSR optimization: convert barriers into point-to-point synchronization

NAS MG class A (256³) on Itanium2+Myrinet

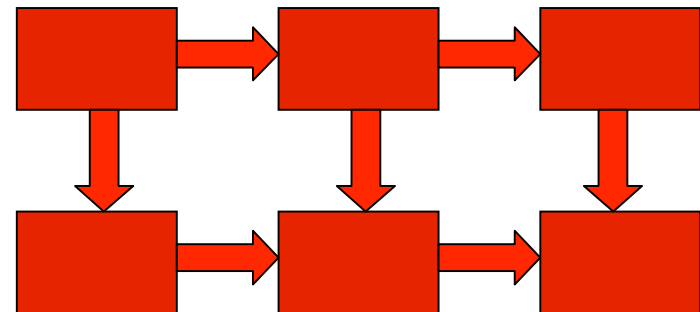


SSR boosts performance & scalability

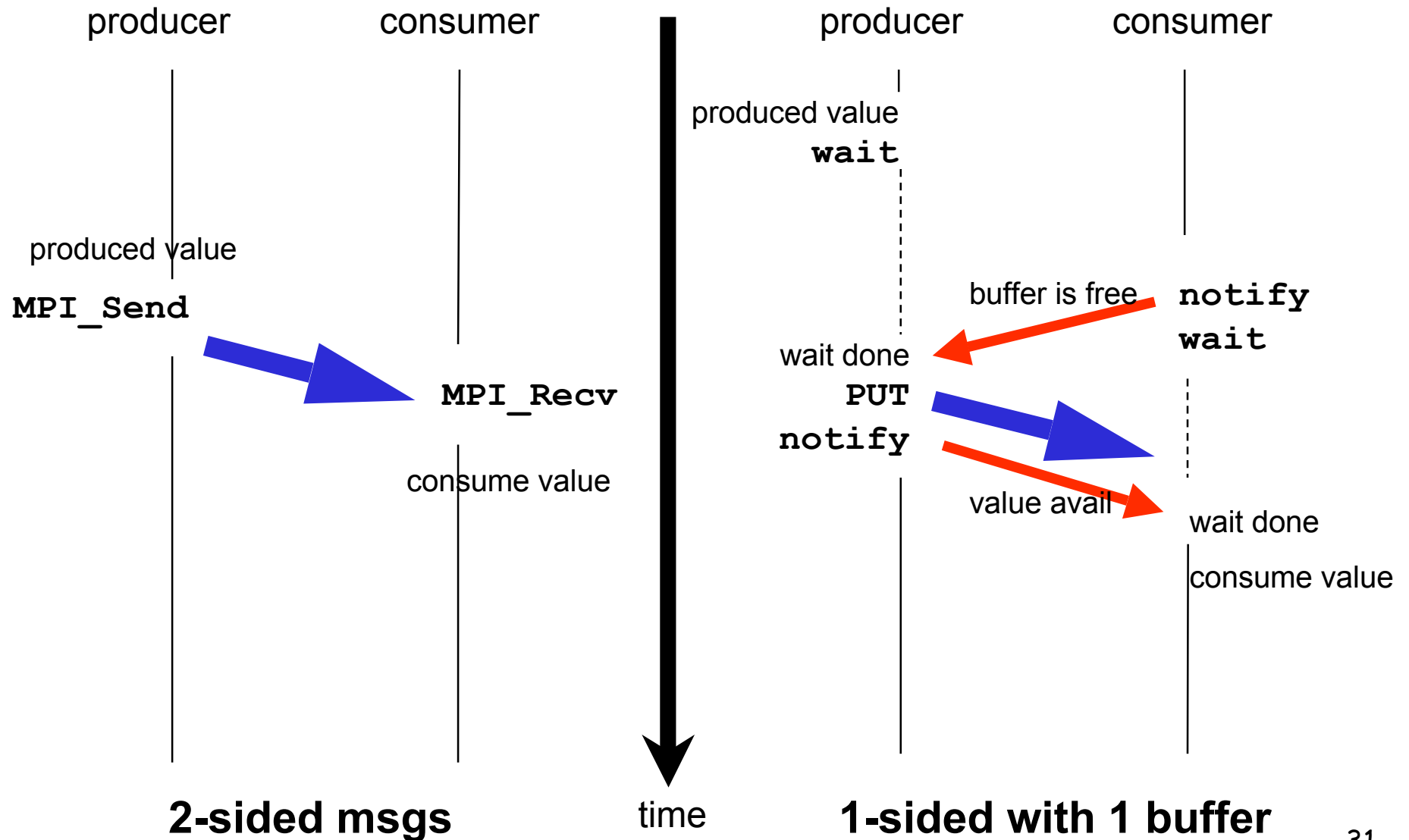
Higher is better

Wavefront Communication

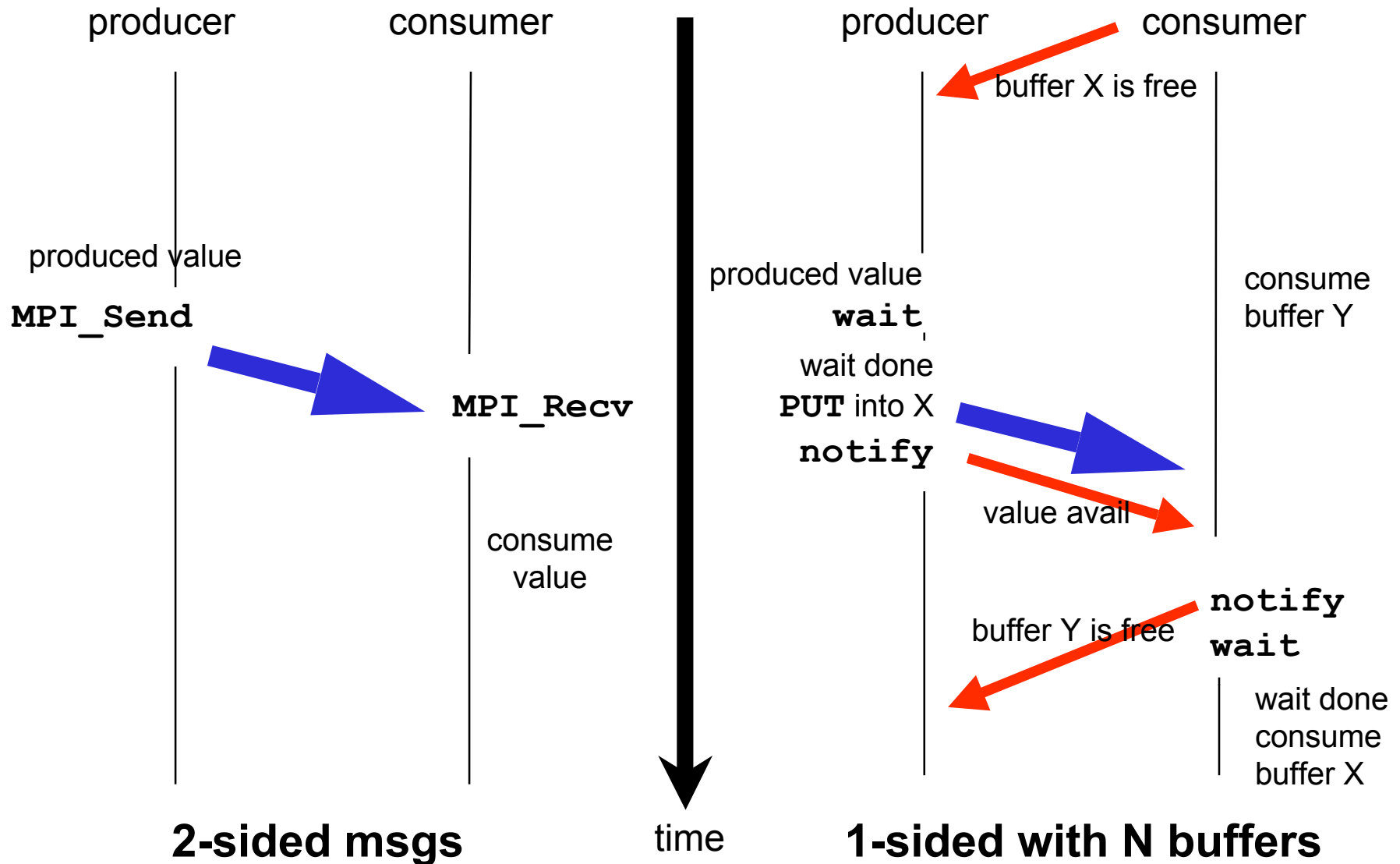
- **Streaming values from producers to consumers**
- **Barrier synchronization is unnatural**
- **Overlap communication with computation**
 - multiple communication buffers
 - pipelined synchronization
 - non-blocking communication



Two-sided vs. One-sided Wavefronts



Hiding Latency with Multiple Buffers



Assessing CAF Support for Wavefronts

- **MPI is simpler!**
 - message = data movement and synchronization
 - no explicit buffer management
 - eager protocol can achieve latency tolerance
- **CAF with one buffer**
 - lacks asynchrony and latency tolerance
- **CAF with multiple buffers**
 - performance better than MPI
 - sender delivers directly to destination memory
 - much harder to program in classic CAF
 - explicit buffer management
 - pipelined non-blocking communication for producer latency tolerance
 - pipelined point-to-point synchronization for consumer latency tolerance

Multi-version variables for CAF

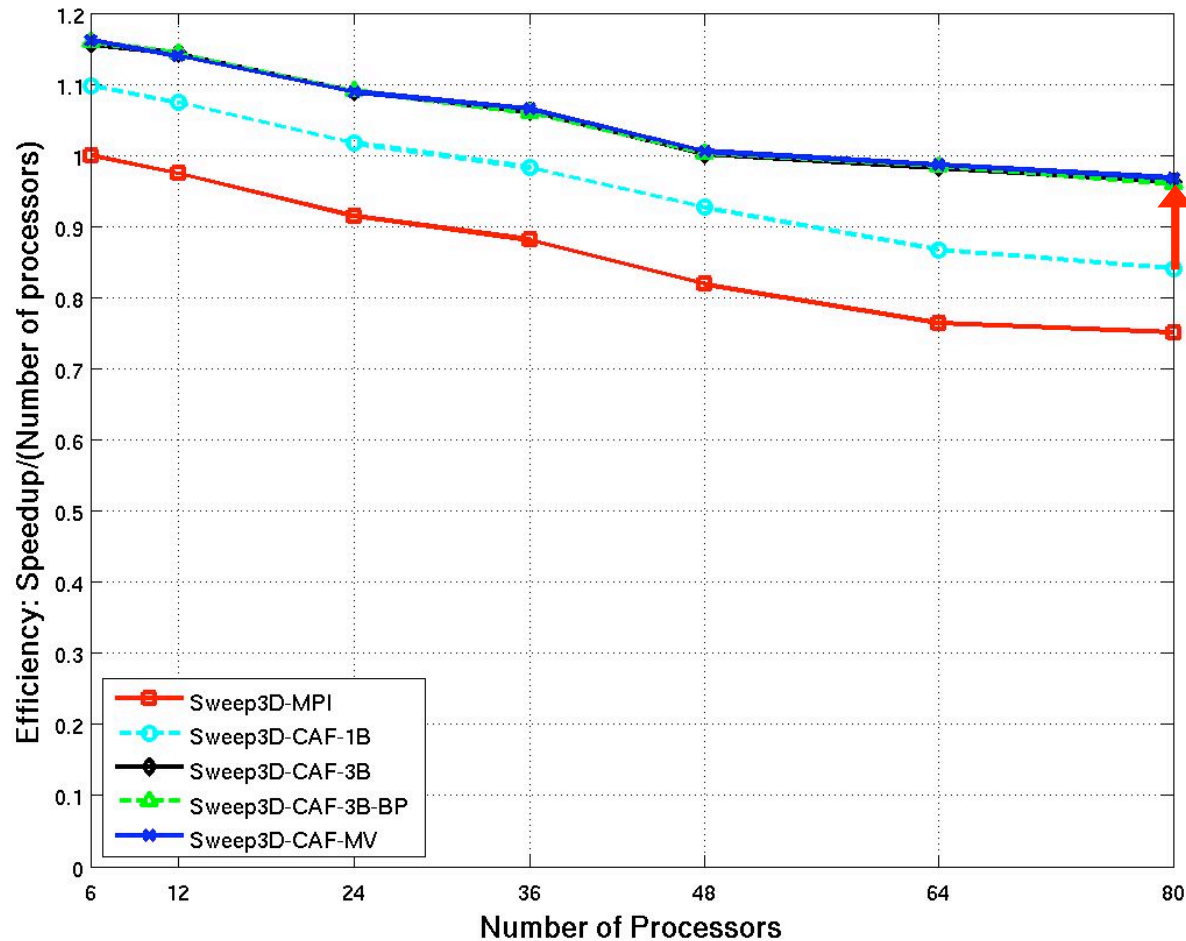
CAF support for wavefront and producer/consumer patterns

- **type(T), multiversion :: a(N, M)[*]**
 - SAVE, allocatable, module, subroutine parameter
- **Producer**
 - commit(a[dest], b, [tag], [live=true])
- **Consumer**
 - retrieve(a, [b], [tag], [image])
 - check(a, [tag])

Multi-version Variable Benefits

- **Well-suited for a variety of communication styles**
 - wave-front, e.g. Sweep3D
 - line-sweep, e.g. NAS BT, SP
 - loosely synchronous
- **Better programmability**
 - transparent buffer mgmt, synchronization, non-blocking communication
- **High performance**
 - hides data movement and synchronization latency
 - exploits non-blocking PUTs
 - removes buffer synchronization from the critical path
 - avoids extra data copies
 - the destination address of communication is known

Itanium2 + Myrinet 2000, Sweep3D Size 300³



Multi-version variables boost performance & scalability

Higher is better

Looking forward with CAF

- **Our experience with CAF has highlighted difficulties**
 - lack of process topologies for collectives and coupled apps
 - exposed latency
 - difficulty of coding high performance wavefront calculations
- **Additional language and library support can help**
 - co-spaces: simplify programming & aid compiler optimization
 - multi-version variables: simplify programming and deliver performance
 - distributed multithreading: avoid exposed latency