From HPF to Coarray Fortran 2.0

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Parallel Programming Language Challenge

To succeed, a language for scalable parallelism must …

- be ubiquitous
  - —multicore processors
  - —cluster in your building
  - —the largest supercomputers available
- be expressive
- be productive
  - —easy to write
  - —easy to read and maintain
  - —easy to reuse

  } significantly simpler than MPI!
- be efficient
- have a promise of future availability and longevity
- be supported by tools
- provide a migration path for users

Outline

- High Performance Fortran
  - —background and motivation
  - —experiences compiling High Performance Fortran (HPF)
- Coarray Fortran
  - —original 1998 version
  - —Fortran 2008 - a standard with coarrays
- Coarray Fortran 2.0 (CAF 2.0)
  - —features
  - —experiences - HPC challenge benchmarks + performance
  - —implementation notes
  - —status
- Looking forward

Context for HPF

Early Models for Parallel Programming

- **Automatic parallelization**
- **Explicitly parallel programming: PCF Fortran and OpenMP**
- **Data-parallel languages**

The Failure of Automatic Parallelization

"Parallelizing compilers are becoming increasingly successful at exploiting coarse-grain parallelism in scientific computations as evidenced by recent results in both ... Polaris ... and ... SUIF ... . While these results are impressive, some of the programs achieved little or no speedup when executed in parallel."

Sungdo Moon, Byoungro So, Mary Hall. Evaluating Automatic Parallelization in the SUIF compiler. IEEE TPDS 11:1, Jan 2000.

Explicitly Parallel Programming Models

- Parallel Computing Forum Fortran and OpenMP
- Features
  - —single-threaded programming
  - —SPMD parallelism within parallel loops, cases, and regions
  - —synchronization mechanisms
    - – locks
    - – support for “ordered/doacross” parallelism
- Limitations
  - —target shared memory platforms

Data Parallel Languages

- Strongly influenced by SIMD programming paradigm and its closeness to the sequential programming model
- Data parallel model
  - —large data structures laid out across memories of a distributed memory parallel machine
  - —elements of these data structures could be operated upon in parallel
  - —key properties
    - – global name space
    - – single thread of control
    - – parallel statements execute in a loosely synchronous fashion
- Examples
  - — Fortran D (Rice), Vienna Fortran, CM Fortran, C*, Data Parallel C, ZPL

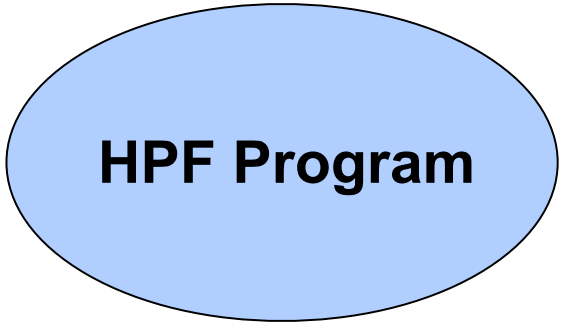
HPF Goals

- Provide a convenient programming model for scalable parallel systems
  - —particular emphasis on data parallelism
- Present an appropriate machine independent programming model
  - —global view: application developer should view memory as a single address space
  - —programs should have a single thread of control
    - – all parallelism should derive from data parallelism
  - —communication should be implicitly generated
- Deliver performance comparable to best hand-coded MPI

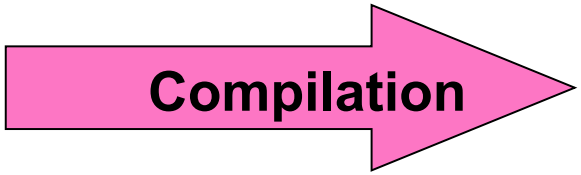
High Performance Fortran

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

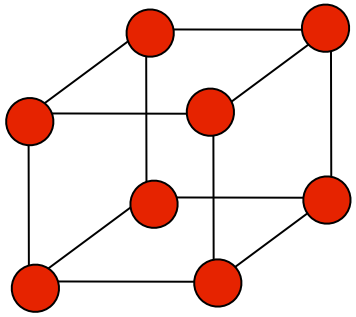
Fortran program
+ data partitioning



Partition computation
Insert communication
Manage storage



Same answers as
sequential program



Parallel Machine

Disclaimer

- This talk doesn't attempt to describe all of the HPF language features and extensions
- Complete coverage language descriptions can be found in the language standard documents
  - —http://www.netlib.org/hpf/hpf-v10-final.ps.gz
  - —http://www.netlib.org/hpf/hpf-v11.ps.gz
  - —http://www.netlib.org/hpf/hpf-v11.ps.gz
- Concurrency and Computation: Practice and Experience. Special Issue: High Performance Fortran. Editors: Ken Kennedy, Yoshiki Seo. Volume 14, Issue 8-9. Pages 551–803, July - 10 August 2002.
  - —Yoshiki Seo, Hidetoshi Iwashita, Hiroshi Ohta, Hitoshi Sakagami. HPF/JA: extensions of High Performance Fortran for accelerating real-world applications, pages 555–573.

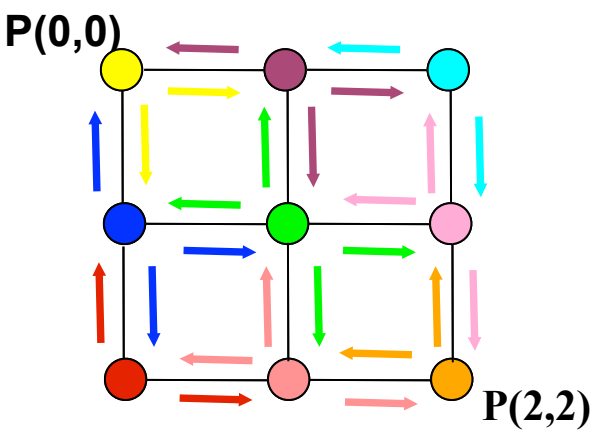
Principal HPF Language Features

- **PROCESSORS**
  - —define a logical k-dimensional grid of virtual processors
  - —specify rank, and extent in each dimension
  - —e.g., PROCESSORS p(8, 64, 16)
- **TEMPLATE**
  - —abstract space of indexed positions
  - —target for alignment mappings
  - —e.g., TEMPLATE t(N,N)
- **ALIGN**
  - —specify that data objects will be mapped in the same way as others
  - —map array dimensions to ranks and positions (affine expressions)
  - —e.g., ALIGN a(j, i) with t(2*i+1,j)
- **DISTRIBUTE**
  - —specify how a dimension of an array or template will be partitioned
  - —e.g., *, block, cyclic, block(256), cyclic(5)

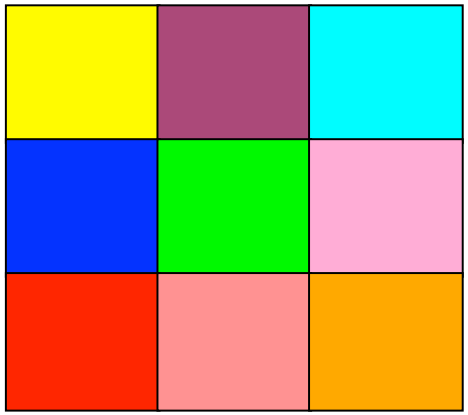
Example HPF Program

```
!HPF$ processors P(3,3)
!HPF$ distribute a(block, block) onto P
!HPF$ 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 with Rice dHPF Compiler

- **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**

dHPF Features for High Performance

- **Program analysis**
  - —integer-set based analysis of iteration spaces, communication
- **Sophisticated computation partitionings**
  - —e.g. partially-replicated computation to reduce communication
- **Sophisticated data partitionings**
  - —skewed cyclic tilings using symbolically-parameterized tiles of uneven size with many-one mappings of tiles to processors
- **Communication optimization**
  - —communication normalization, coalescing
  - —latency hiding: overlap communication and computation
- **Memory hierarchy optimization**
  - —generate clean inner loops
  - —cache optimization (padding, communication buffer mgmt)

Formal Compilation Framework

3 types of Sets	*3 types of Mappings*	
Data	***Layout***:	data ⟷ processors
Iterations	***Reference***:	iterations ⟷ data
Processors	***CompPart***:	iterations ⟷ processors

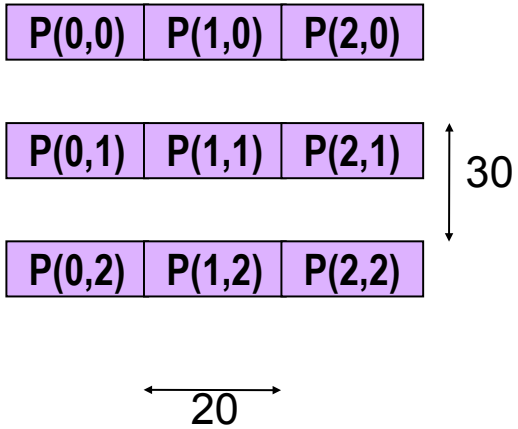
- **Representation**
  - **integer tuples with Presburger arithmetic for constraints**
    - **universal & existential quantifiers**
    - **linear inequalities with constant coefficients**
    - **logical operators**
- **Analysis: use set equations to compute set(s) of interest**
  - **iterations allocated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

[Adve & Mellor-Crummey, PLDI98]

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) )
  ENDDO
ENDDO
```

data / loop partitioning

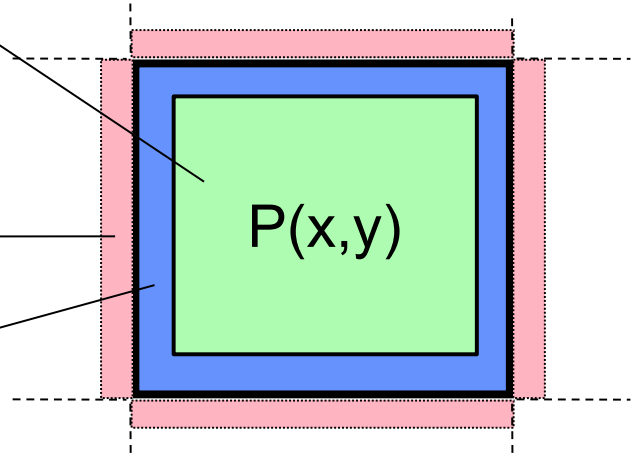


Local section for P(x,y)
(and iterations executed)

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

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

Integer Sets Inside the dHPF Compiler

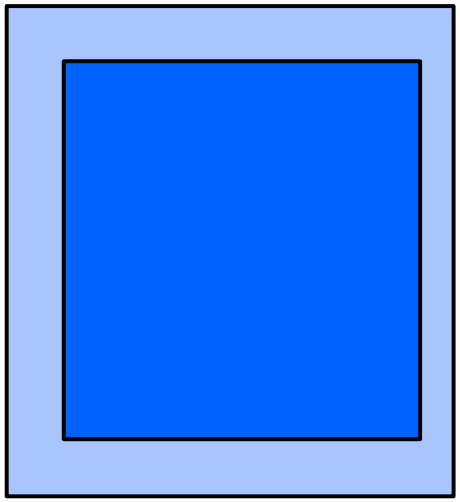
Fragment from CodeGenDisjunctiveIterationSpaces

```
Relation intersection = Relation::True(noutput);
Relation all = Relation::False(noutput);
int numEntries = ispaces.NumberOfEntries();

if (numEntries > 1) {
  for (i = 0; i < numEntries; i++) {
    Relation iterSet = *(ispaces[i]);
    Relation transform = *(transformations[i]);
    Relation transformedIters = (iterSet.is_null() ?
                 Relation::False(iterSet.n_set()) :
                 Composition(copy(transform),copy(iterSet)));

    all = Union(copy(transformedIters), all);
    intersection = Intersection(transformedIters, intersection);
    CompactSet(all);
    CompactSet(intersection);
  }
  intersection = Intersection(intersection, Extend_Set(copy(known), noutput));
  CompactSet(intersection);

  if (intersection.is_satisfiable()) {
    Relation difference = Difference(all, copy(intersection));

  difference = Intersection(difference, Extend_Set(copy(known), noutput));
  CompactSet(difference);
```

dHPF set implementation: Omega [Pugh]

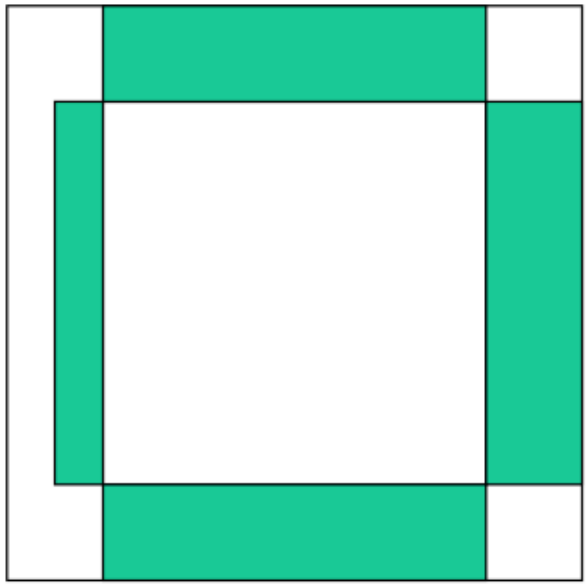
HPF/JA: Explicit Control of Shadow Regions

- SHADOW A(4:2,4:4)
- REFLECT
- ON EXT_HOME
- LOCAL



dHPF Extended ON HOME

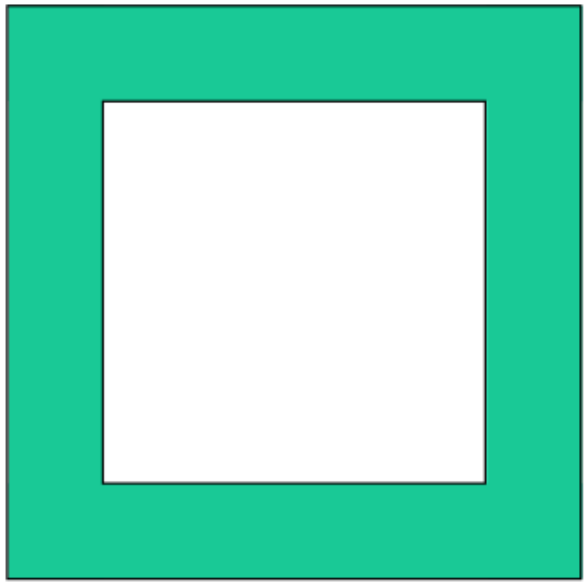
Sophisticated partitionings for partially-replicated computation



SHADOW a(2, 2)

ON_HOME a(i-2, j) ∪ ON_HOME a(i+2, j) ∪

ON_HOME a(i, j-2) ∪ ON_HOME a(i, j+1)



SHADOW a(2, 2)

ON_EXT_HOME a(i, j)

Example of Partial Replication: NAS SP rhs.f

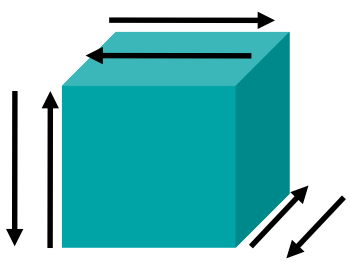
```
    do    k = 0, grid_points(3)-1
       do    j = 0, grid_points(2)-1
          do    i = 0, grid_points(1)-1
             rho_inv = 1.0d0/u(i,j,k,1)
!HPF$ ON HOME (rhs(i, j, k, 1), rhs(i - 1, j, k, 1), rhs(i + 1, j, k, 1), rhs(i, j - 1, k, 1),
rhs(i, j + 1, k, 1), rhs(i, j, k - 1, 1), rhs(i, j, k + 1, 1)) BEGIN
             rho_i(i,j,k) = rho_inv
             us(i,j,k) = u(i,j,k,2) * rho_inv
             vs(i,j,k) = u(i,j,k,3) * rho_inv
             ws(i,j,k) = u(i,j,k,4) * rho_inv
             square(i,j,k)     = 0.5d0* ( u(i,j,k,2)*u(i,j,k,2) + u(i,j,k,3)*u(i,j,k,3) +   &
                                  u(i,j,k,4)*u(i,j,k,4) ) * rho_inv
             qs(i,j,k) = square(i,j,k) * rho_inv
             aux = c1c2*rho_inv* (u(i,j,k,5) - square(i,j,k))
             aux = dsqrt(aux)
             speed(i,j,k) = aux
             ainv(i,j,k)  = 1.0d0/aux
CHPF$ END ON
          end do
       end do
    end do
```

later computation nested in the same enclosing loop used the partially-replicated values

Data Partitioning

- **Good parallel performance requires suitable partitioning**
- **Tightly-coupled computations are problematic**

- **Line-sweep computations: e.g., ADI integration**



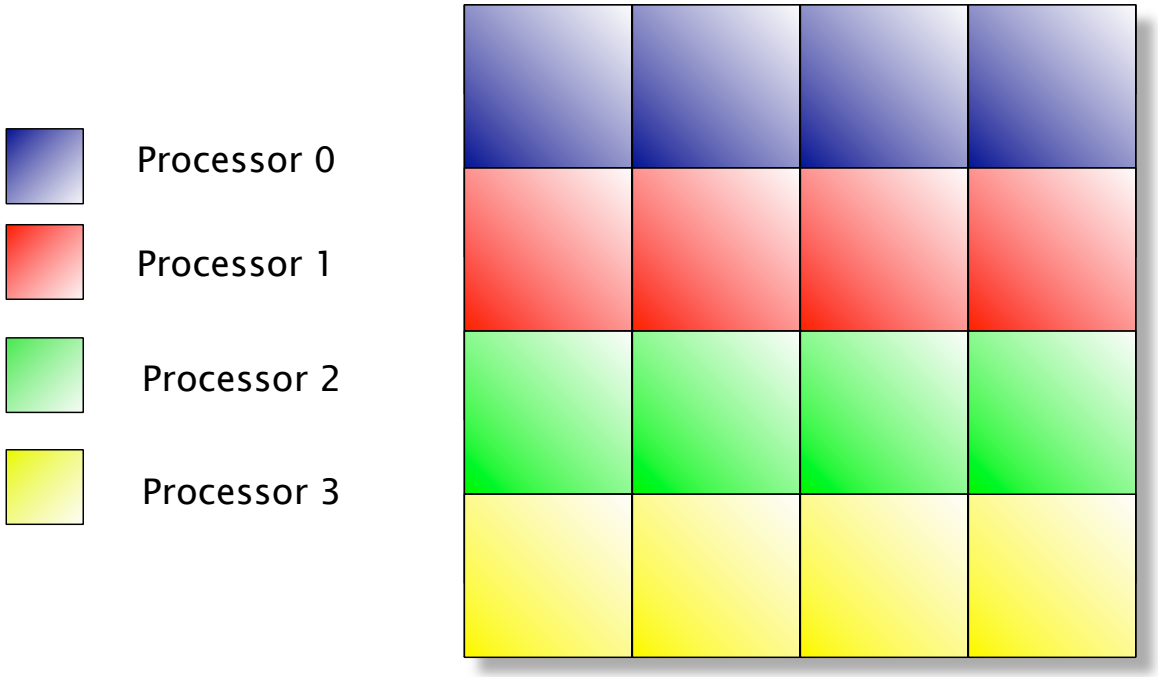
```
do j = 1, n
   do i = 2,n
      a(i,j) = … a(i-1,j)
```

recurrences make parallelization difficult with BLOCK partitionings

Coarse-Grain Pipelining

Compute along partitioned dimensions

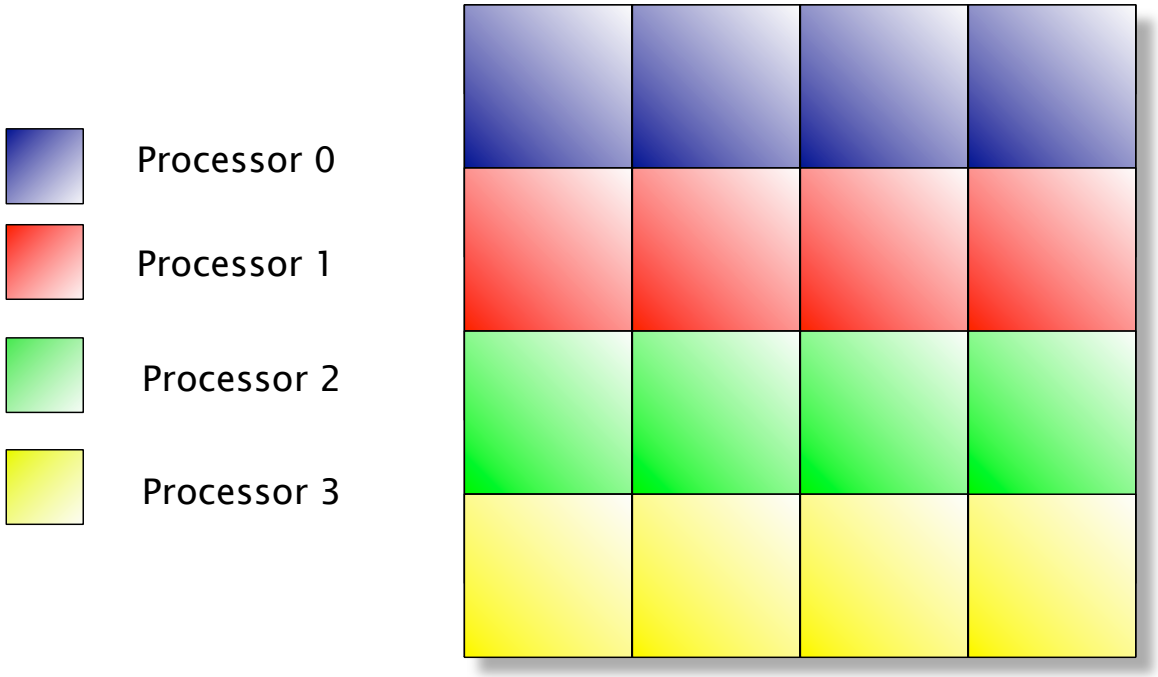
Partial serialization induces wavefront parallelism with block partitioning



Coarse-Grain Pipelining

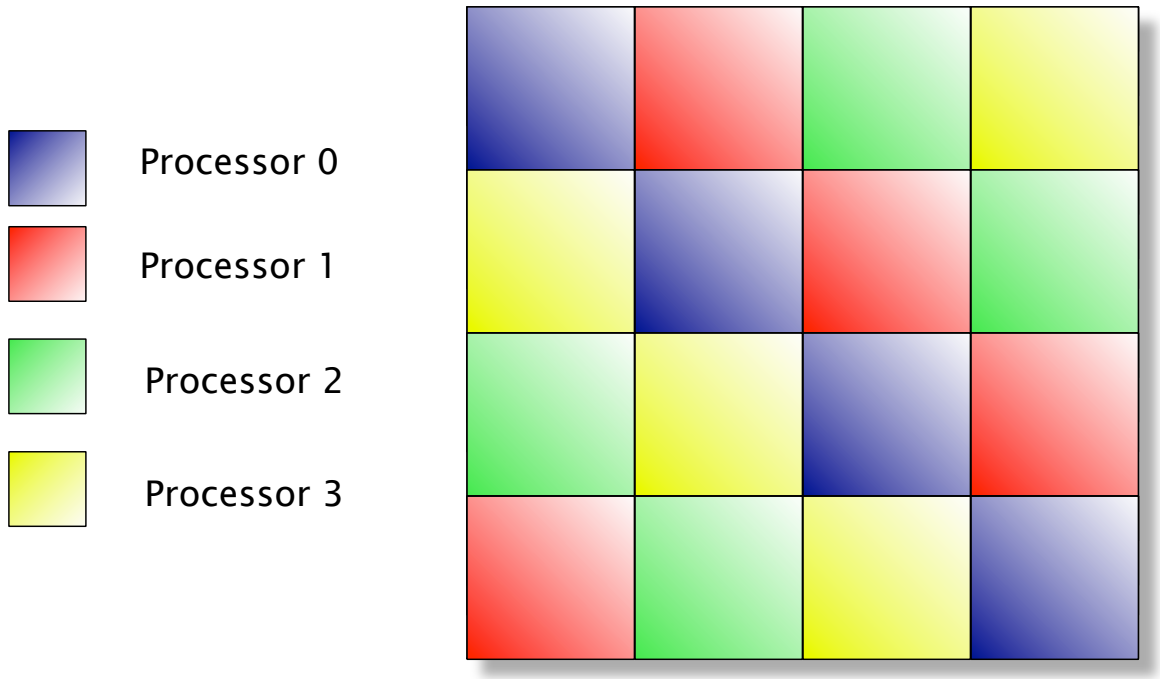
Compute along partitioned dimensions

Partial serialization induces wavefront parallelism with block partitioning



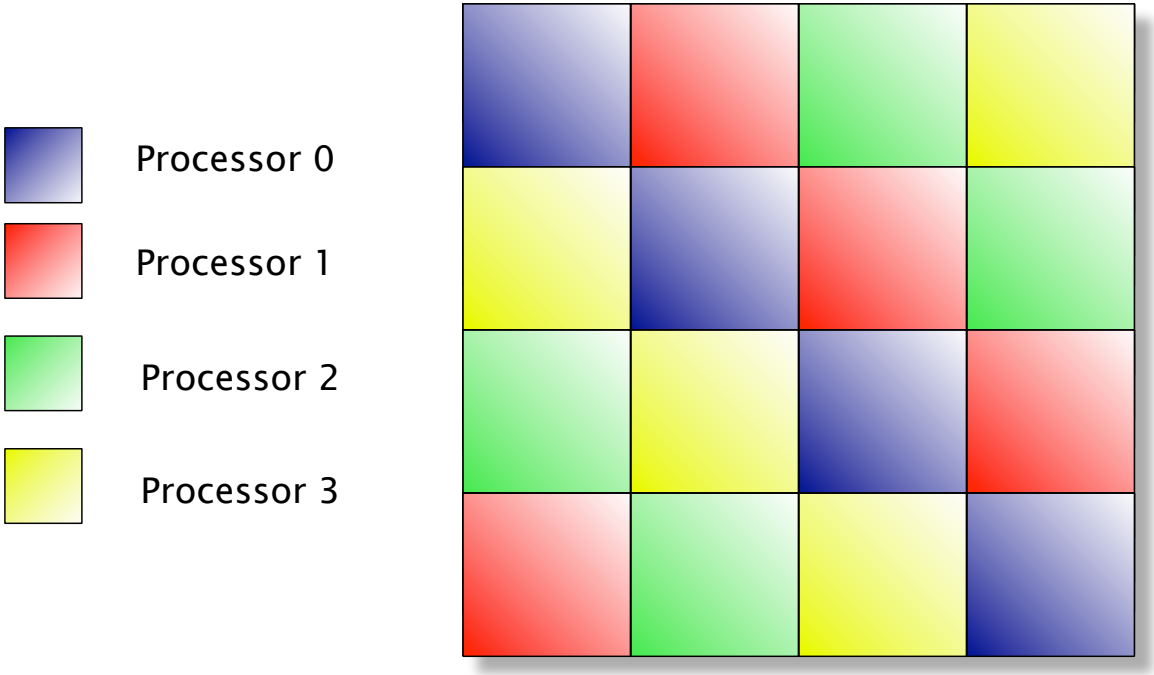
Multipartitioning

- Each processor owns a tile between each pair of cuts along each distributed dimension
- Enables full parallelism for a sweep along any partitioned dimension



Multipartitioning

- Each processor owns a tile between each pair of cuts along each distributed dimension
- Enables full parallelism for a sweep along any partitioned dimension

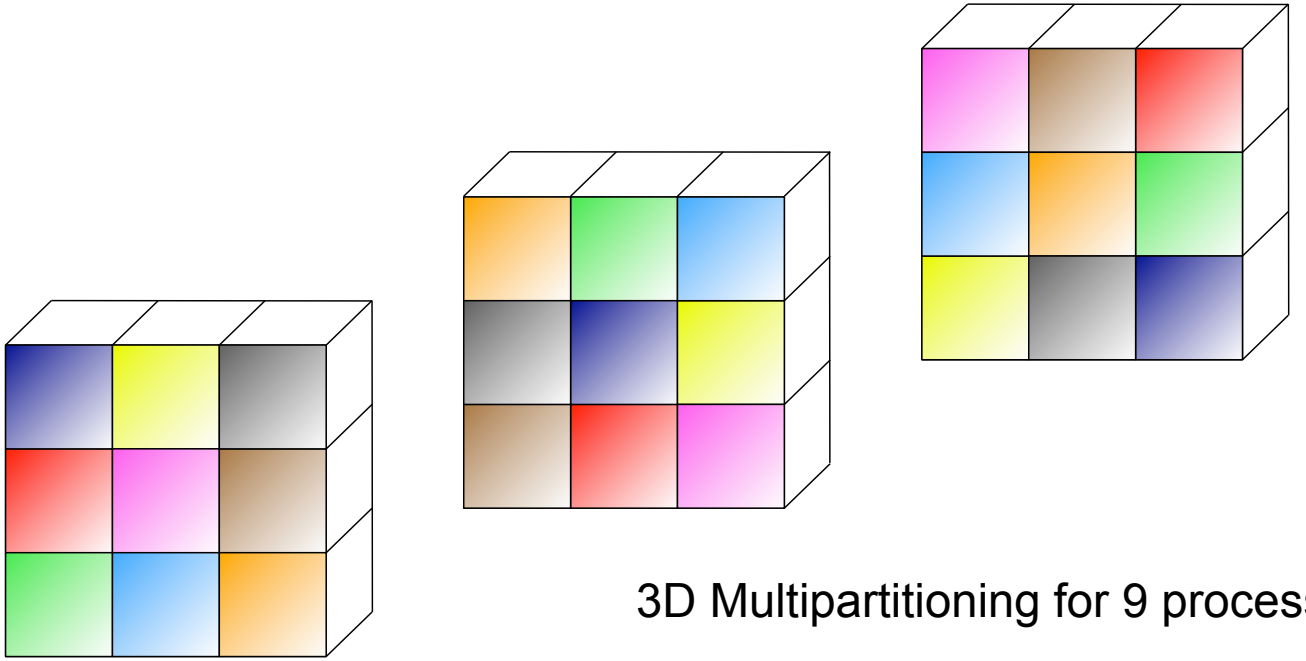


Higher-dimensional Multipartitioning

An array of k > d dimensions can be partitioned into

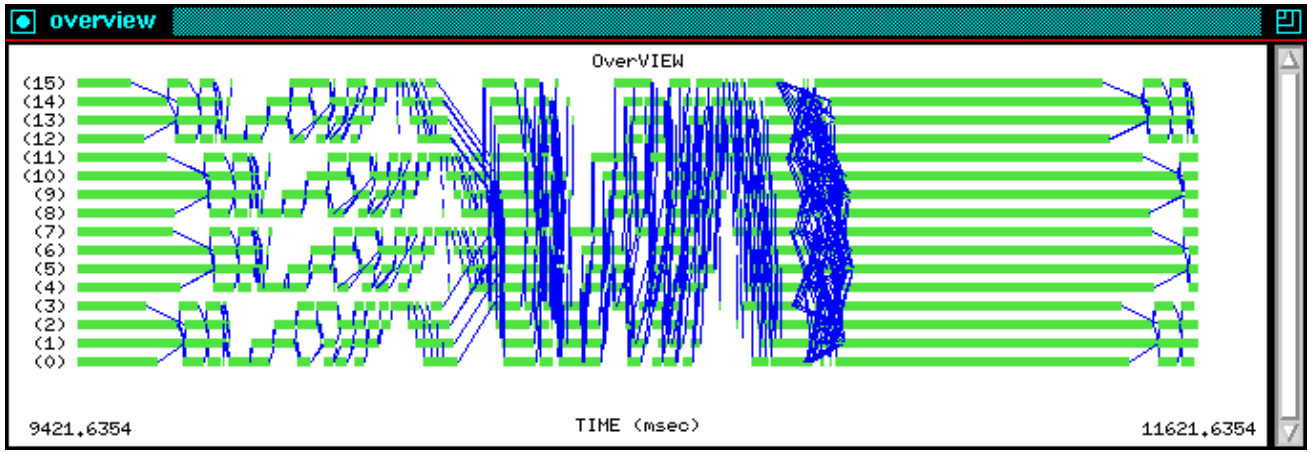
$p^{d/(d-1)}$ tiles (diagonal multipartitioning)

(*p* is the number of processors)

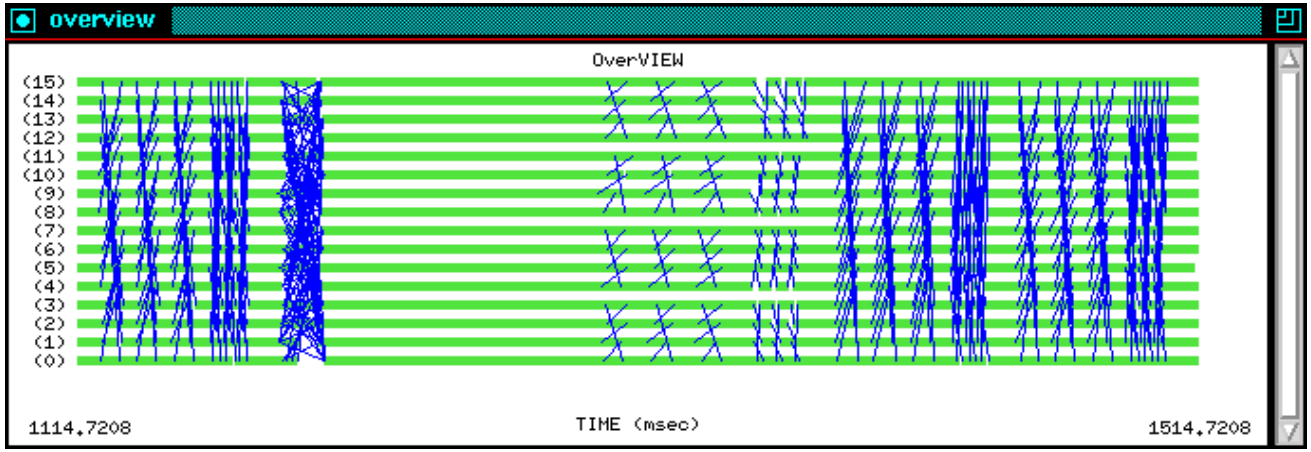


3D Multipartitioning for 9 processors

Comparing Parallelization Strategies



Compiler-generated coarse-grain pipelining

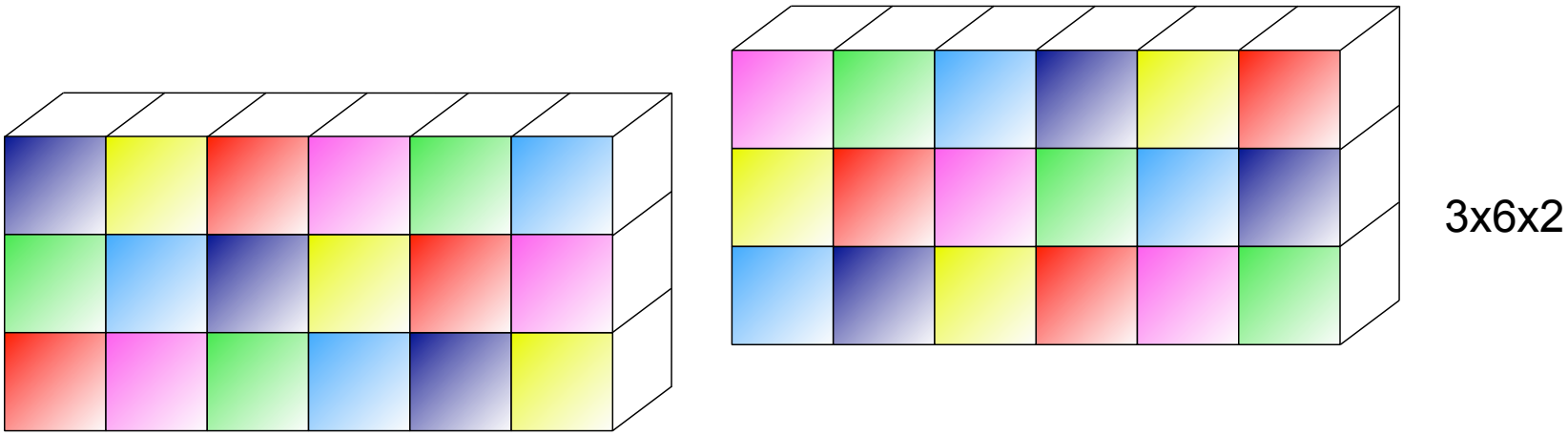


Hand-coded multipartitioning

Generalized Multipartitioning

Higher dimensional multipartitionings for arbitrary numbers of processors

- **Optimal overpartitionings (more than one tile per processor per hyperplane) + modular mappings**
- **Compiler aggregates carried communication for hyperplanes**



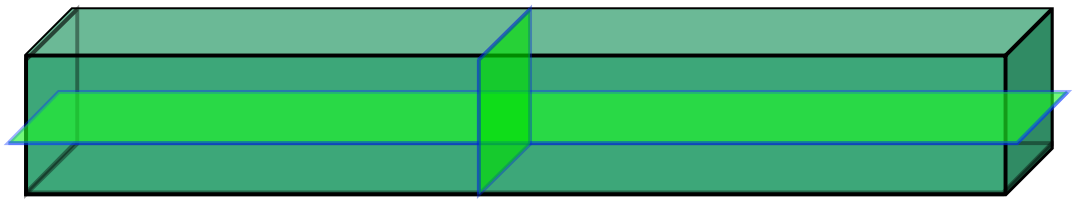
3D Multipartitioning for 6 processors

Generalized Multipartitioning

Given an *n*-dimensional data domain and *p* processors, select

— **which λ dimensions to partition, $2 \leq \lambda \leq n$; how many cuts in each**

- **Partitioning constraints**
  - **# tiles in each λ - 1 dimensional hyperplane is a multiple of *p***
  - **no more cuts than necessary**
- **Objective function: minimize communication volume**
  - **pick the configuration of cuts to minimize total cross section**



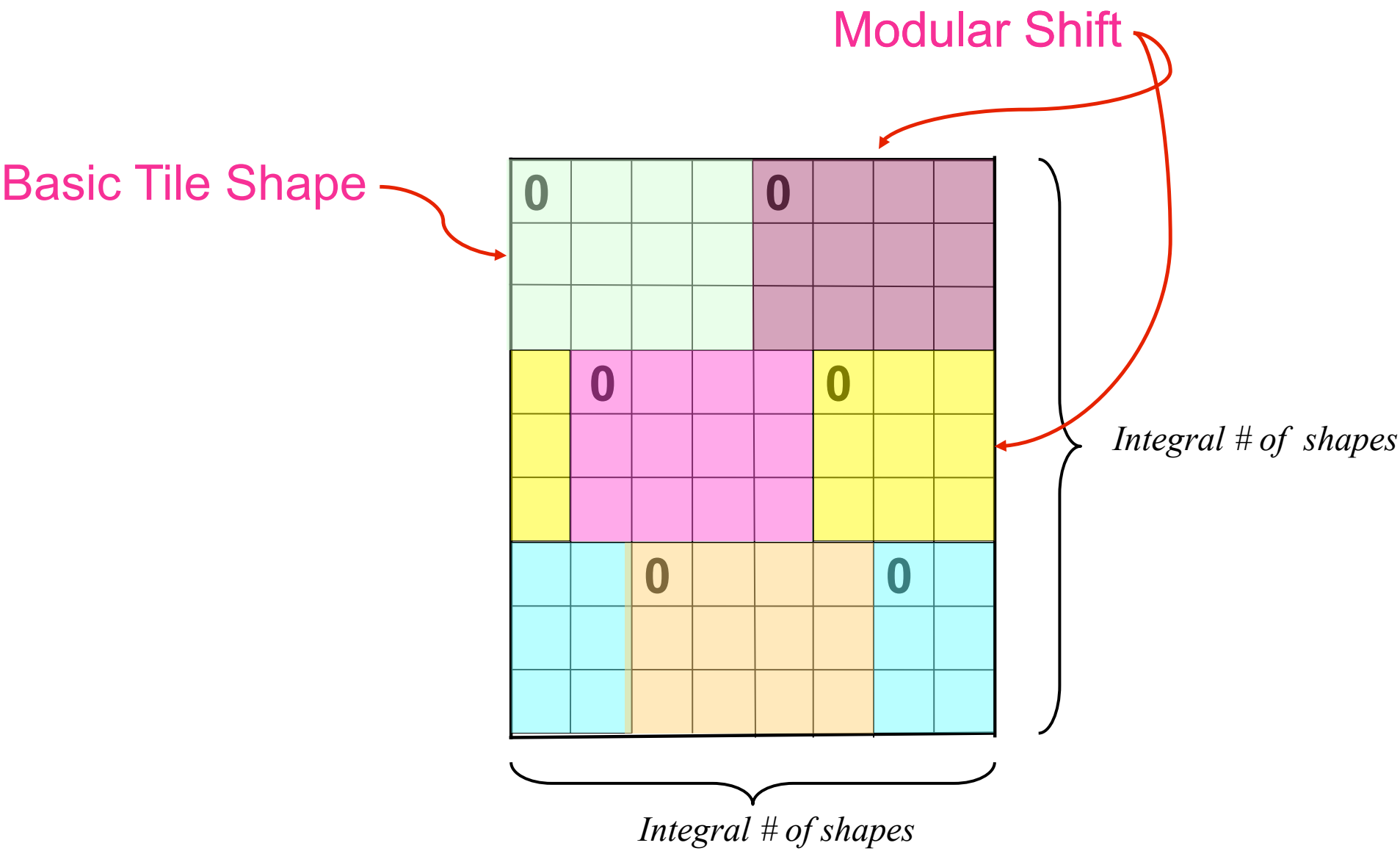
- **Mapping constraints**
  - **load balance:** ***in a hyperplane, each proc has same # tiles***
  - **neighbor: in any particular direction, the neighbor of a given processor is the same**

IPDPS 2002 Best paper in Algorithms; JPDC 2003

Choosing the Best Partitioning

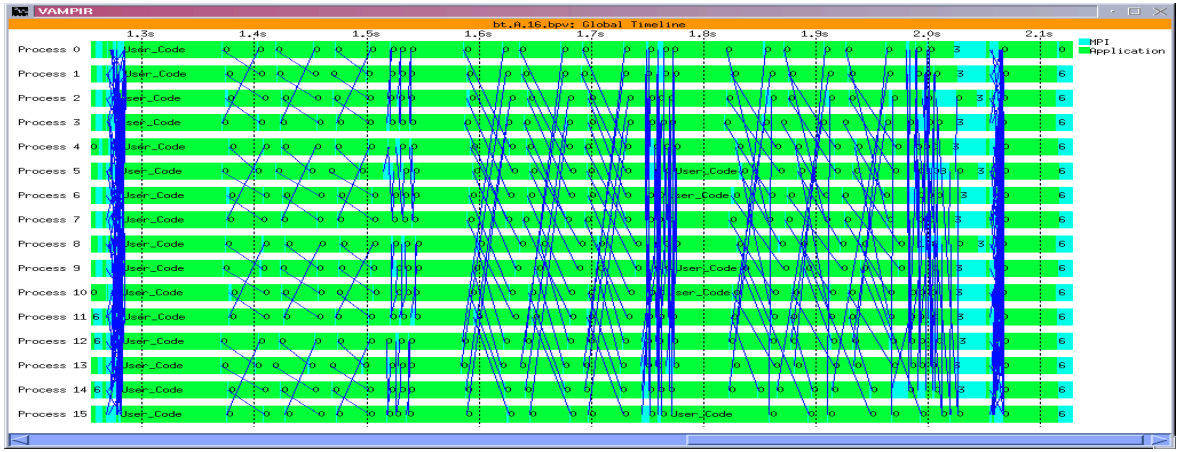
- **Enumerate all elementary partitionings**
  - **—candidates depend on factorization of *p***
- **Evaluate their communication cost**
- **Select the minimum cost partitioning**
- **Modest complexity**
- **Very fast in practice**

Map Tiles with Modular Mappings

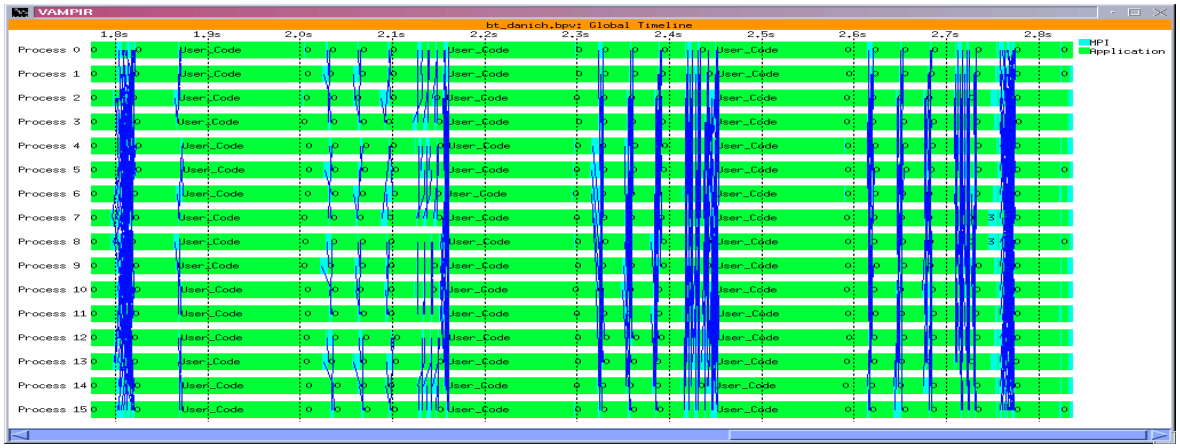


Compiler vs. Hand-coded Parallelization

Hand-written 3D Multipartitioning



Compiler-generated 3D Multipartitioning



Execution Traces for NAS BT Class 'A' - 16 processors, SGI Origin 2000
Compiler parallelization with Rice's dHPF compiler

Communication Coalescing

- **Two kinds optimizations**
  - **—subsumption**
    - **completely eliminate a communication set that is covered by another**
  - **—coalescing**
    - **fuse and eliminate duplicates in partially overlapping sets**
    - **conditions**
      - **same dimension**
      - **same direction**
      - **constant width**
      - **same destination**

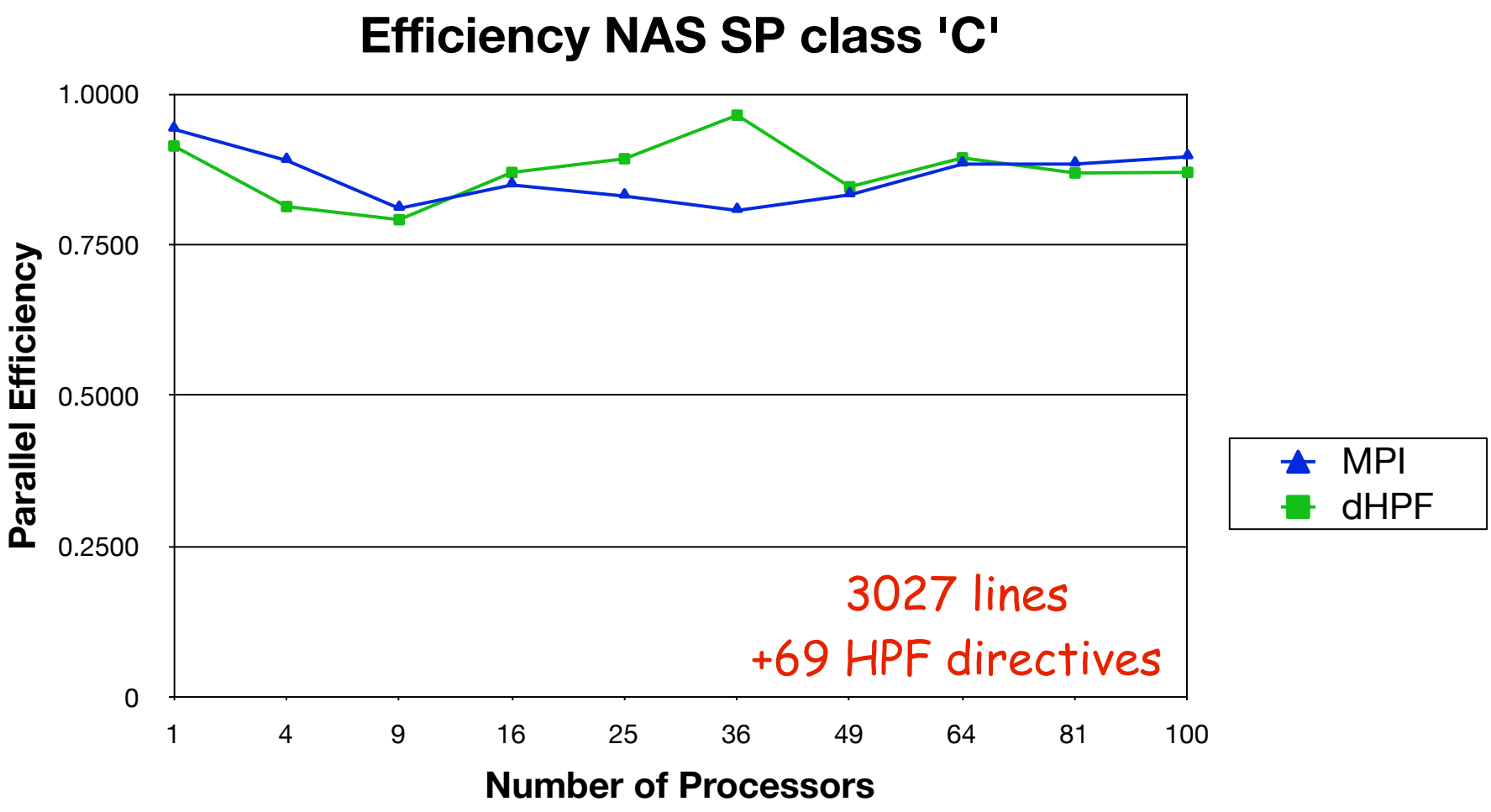
```
do timestep = 1, T
                 Coalesce data exchange at this point
  do j = 1, n
    do i = 3, n
      a(i, j) = a(i + 1, j) + b(i - 1, j) ! ON_HOME a(i,j)
    enddo
  enddo

  do j = 1, n
    do i = 1, n - 2
      a(i + 2, j) = a(i + 3, j) + b(i + 1, j) ! ON_HOME a(i + 2, j)
    enddo
  enddo

  do j = 1, n
    do i = 1, n - 1
      a(i + 1, j) = a(i + 2, j) + b(i + 1, j) ! ON_HOME a(i + 1, j)
    enddo
  enddo
enddo
```

- **How:**
  - **—normalize reference subscripts with respect to on home subscript**
  - **—compare resulting sets using 'integer set framework'**

Memory Hierarchy Management

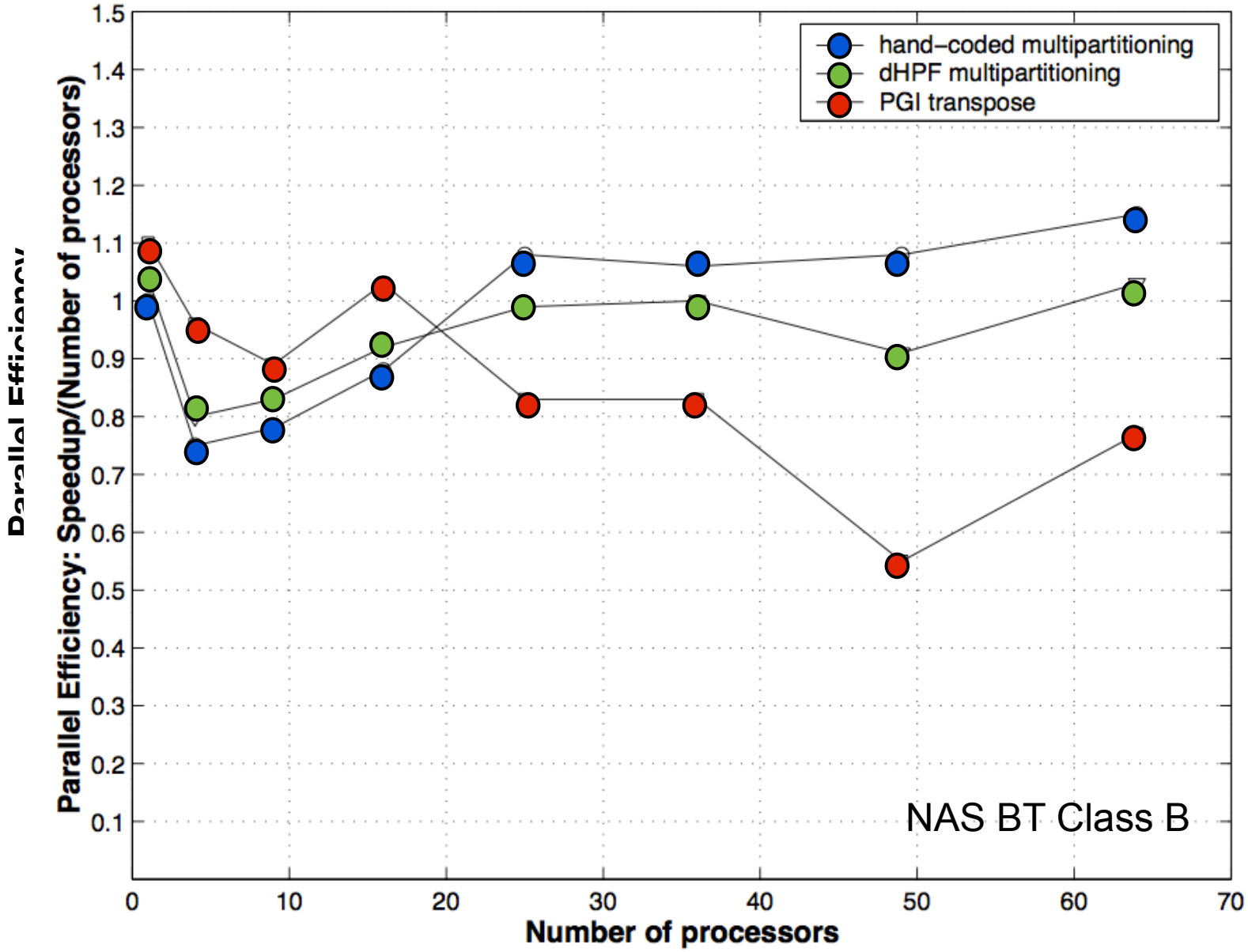
- Array padding to avoid cache conflicts within arrays
- Inter-array padding to avoid conflicts between arrays
- Arena-based buffer management
  - —reduced footprint of communication buffers
- Direct access communication buffers as alternative to overlap regions
  - —avoid unpacking into overlap region to avoid extra “footprint” in the cache

NAS SP Using 3D Multipartitioning



3D multipartitioning
communication coalescing
partially-replicated computation
memory hierarchy optimization

NAS BT: Comparing 3 parallelizations



IMPACT-3D

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

- **Problem size: 1024 x 1024 x 2048**
- **Compiled with HPF/ES compiler**
  - **—7.3 TFLOPS on 2048 ES processors ~ 45% peak**
- **Compiled with dHPF on Alpha Cluster (Lemieux)**

1334 lines
+45 HPF directives

# procs	relative speedup	GFLOPS	% FP peak
128	1.0	47.3	18.5
256	1.88	89.1	17.4
512	3.72	175.9	17.2
1024	7.45	352.0	17.2

3D block partitioning; use REFLECT and LOCAL

Careful Optimization is Required!

- **Excess communication undermines scalability**
  - —both frequency and volume must be right!
  - —examples and impact
    - – 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
  - —embrace HPF/JA-style directives to control communication
- **Single processor efficiency is critical**
  - —must use caches effectively on microprocessors
  - —examples and impact
    - – use constraints about partners to simplify communication code

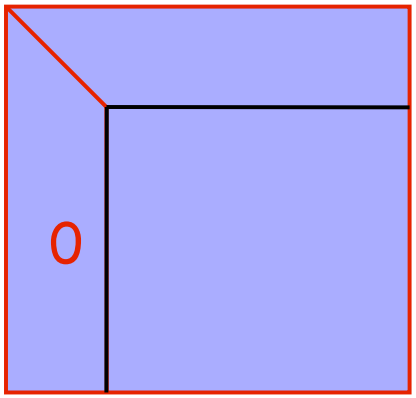
      12% fewer Icache misses, 7% faster: NAS SP @ 64 procs
    - – split loops into “local-only” and “off-processor” loops

      when profitable, don’t unpack into overlap regions

      10% fewer Dcache misses, 9% faster: NAS SP @64 procs

High-level Optimization Challenges

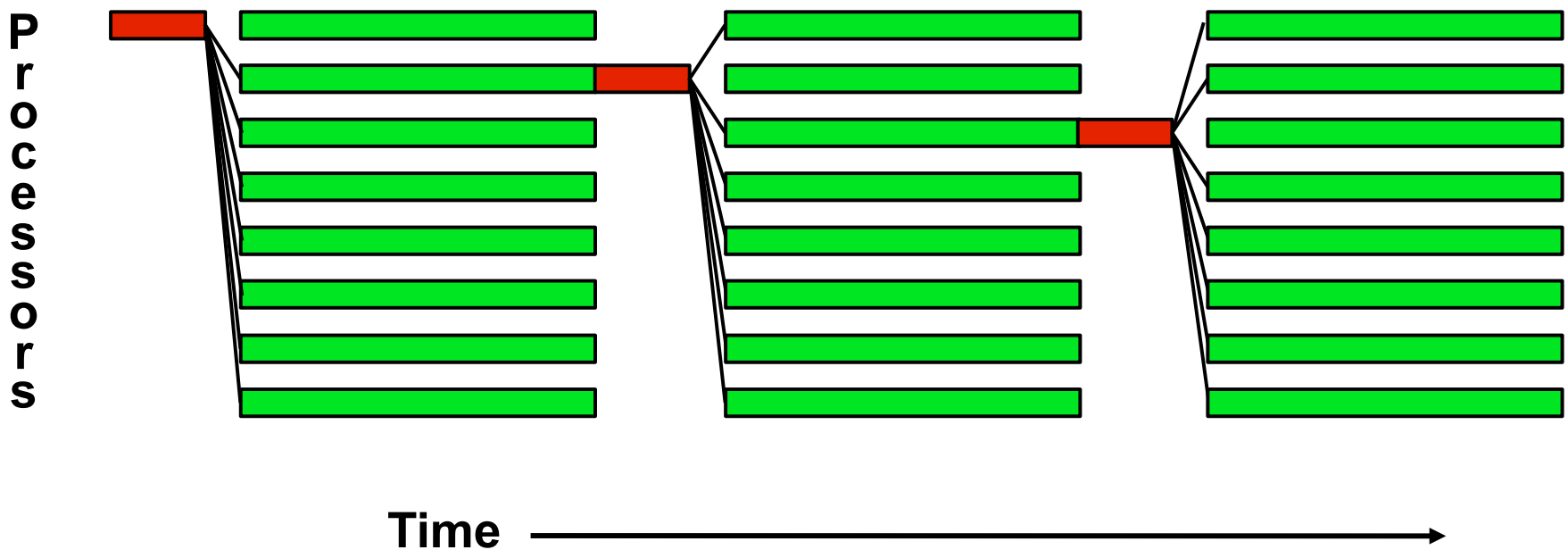
- **Abstract models like HPF rely on compilers to get the parallelism right**
- **Example: Gaussian elimination + partial pivoting**
  - **for each column**
    - **compute the pivot within the column**
    - **compute multipliers to eliminate with pivot**
    - **broadcast pivot and multipliers**
    - **perform elimination on the lower right quadrant**



Gaussian Elimination Parallelism

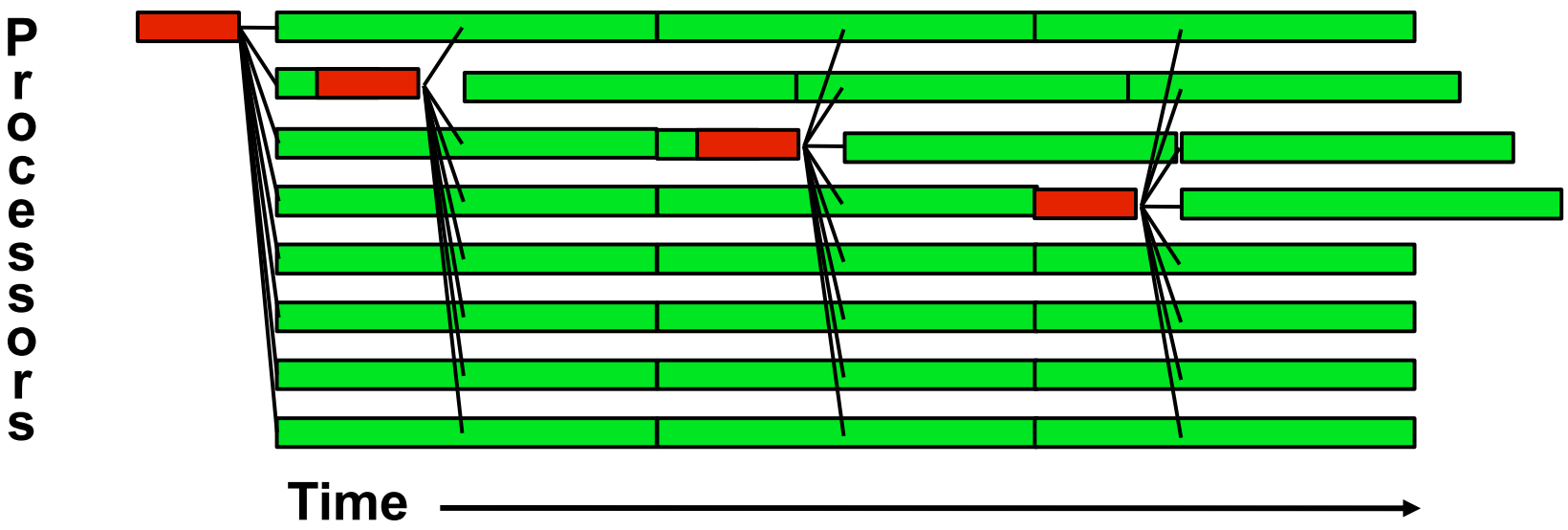
Conventional HPF Compilation

- All processors perform elimination computation with full parallelism, but
- Serialized computation of pivot and multipliers



Getting the Parallelism Right

- Overlap computation of pivot and multipliers with elimination step
- Requires complex optimization of SPMD program
  - —splitting elimination computation
    - – pivot column vs. rest of elimination
  - —software pipelining can avoid impact of serialization



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

ripe for space-time tradeoff as well as strength reduction

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

FFT Challenges

- Efficient code for bit reverse permutation
  - —using the memory hierarchy effectively is challenging alone
    - gather vs. scatter vs. blended approach
- Strided iteration space for k loop
  - —makes Presburger arithmetic representation for sets undecidable
- Effectively partitioning computation
  - —avoid executing loop iterations for which you have no work
- Amortizing communication overhead
  - —avoid element-wise communication
- Efficient access to values received from remote processors
- Overlapping communication and computation
- Efficient code for inner loops

Need for Tools

- **Challenge: substantial gap between a user program and its distributed-memory implementation**
- **dHPF approach**

static tracking

  - **track dependences between input code and generated code**
    - **track the sequence of operations that the compiler applies to the abstract syntax tree**
  - **using mappings collected, can map back and forth between fragments in generated code and fragments in source code**
  - **provide tool for viewing source and generated code**
  - **attribute performance to both generated and source programs**
- **HPCToolkit's global view of performance**

dynamic tracking

  - **X. Liu, J. Mellor-Crummey, M. Fagan. A New Approach to Performance Analysis of OpenMP. ICS 2013.**
  - **A.Eichenberger, J. Mellor-Crummey, et al: OMPT and OMPD: OpenMP tools application programming interfaces for performance analysis and debugging. April 2013. http://openmp.org/mp-documents/ ompt-tr.pdf**

Tools Challenge: Gap Between Source and Implementation

Case study: LLNL's Lulesh in Chapel

- **Use Rice's HPCToolkit to measure, analyze, present performance data**
- **Challenges for Chapel**
  - —**tools can only show local view of performance**
    - **master thread**
    - **worker threads (shown)**
  - —**without runtime help, can't reconstruct relationship between compiler-generated code and user-level application calling context**

hpcviewer: lulesh

lulesh.chpl

```
977
978   CalcHourglassControlForElems(determ);
979 }
980
981
982 proc IntegrateStressForElems(sigxx, sigyy, sigzz, determ) {
983   forall k in Elems {
984     var b_x, b_y, b_z: 8*real;
985     var x_local, y_local, z_local: 8*real;
986     localizeNeighborNodes(k, x, x_local, y, y_local, z, z_local);
987
988     var fx_local, fy_local, fz_local: 8*real;
989
990     //    local {
991       /* Volume calculation involves extra work for numerical consistency. */
992       CalcElemShapeFunctionDerivatives(x_local, y_local, z_local,
993                                        b_x, b_y, b_z, determ[k]);
994
```

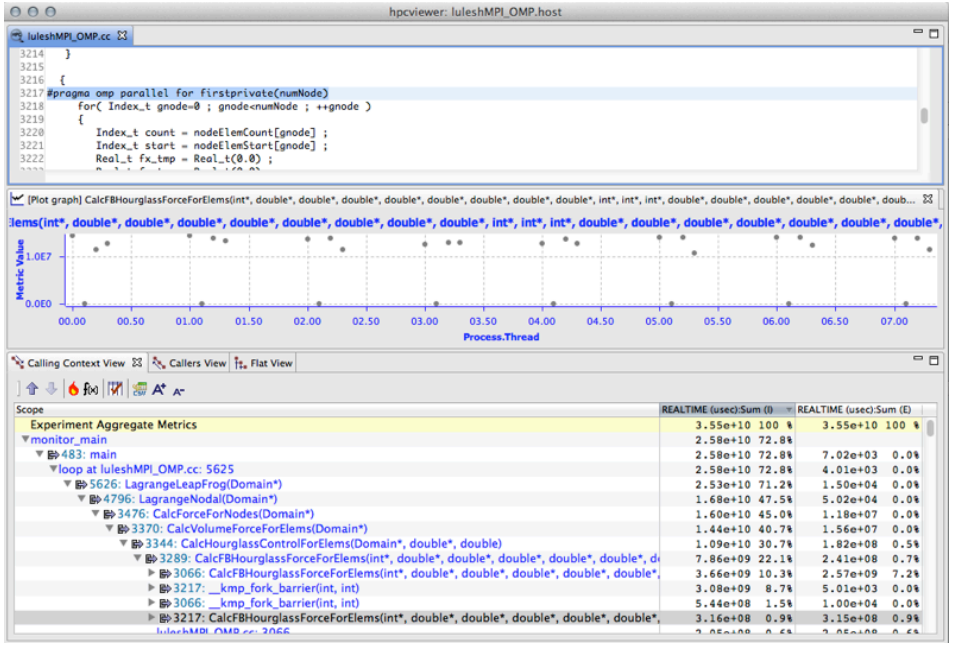
Calling Context View | Callers View | Flat View

Scope	PAPI_TOT_CYC:Sum (I)		PAPI_TOT_CYC:Sum (E)	
Experiment Aggregate Metrics	1.34e+10	100 %	1.34e+10	100 %
thread_begin	1.18e+10	88.1%	2.76e+08	2.1%
loop at tasks-fifo.c: 0	1.18e+10	88.1%		
loop at tasks-fifo.c: 0	1.18e+10	88.1%		
loop at tasks-fifo.c: 0	2.94e+09	22.0%	6.00e+06	0.0%
wrapcoforall_fn28	2.70e+09	20.2%		
wrapcoforall_fn25	2.33e+09	17.5%	2.00e+06	0.0%
inlined from lulesh.chpl: 983	2.33e+09	17.4%		
lulesh.chpl: 982	2.00e+06	0.0%	2.00e+06	0.0%
wrapcoforall_fn37	5.16e+08	3.9%	2.00e+06	0.0%
wrapcoforall_fn26	4.98e+08	3.7%	6.00e+06	0.0%
wrapcoforall_fn20	2.94e+08	2.2%		
wrapcoforall_fn44	2.26e+08	1.7%	4.00e+06	0.0%
wrapcoforall_fn29	1.90e+08	1.4%	2.00e+06	0.0%
wrapcoforall_fn7	1.86e+08	1.4%	3.40e+07	0.3%

Global View Performance via Dynamic Tracking

Case study: Lulesh in MPI+OpenMP

- Use emerging OMPT interface to assemble global view of application performance
- Key OMPT functionality
  - —track runtime states
  - —provide hooks that enable tools to reconstruct application call stacks
- Tool can assemble code-centric, thread-centric, and time-centric performance views correlated with application global view



Some Lessons from dHPF Project

- Good parallelizations require proper partitionings
  - —inferior partitionings will fall short at scale
- Excess communication undermines scalability
  - —both frequency and volume must be right!
- Must exploit what smart users know
  - —allow the power user to hide or avoid latency
- Single processor efficiency is critical
  - —node code must be competitive with serial versions
  - —must use caches effectively on microprocessors
- Compilation challenges can sometimes be daunting
  - —e.g. FFT
- Brittle compilers present a challenge
  - —achieving high performance requires “knowing the secret code”
    - – experiences with HPF randomaccess benchmark

Open Research Issues

- Generalize static analysis and code generation for complex regular cases
  - —design efficient implementations that are robust
- Give more user feedback/tools so that the issues affecting performance can be pinpointed
  - —help user perform source-level tuning
- More directives to enable more programmer control
  - —in some cases, directives must carry semantic meaning for improving performance
- Provide efficient support for user-defined distributions to broaden applicability
  - —combine data structure abstraction with compiler support
  - —support for managing details at run-time associated with implementing complex user-defined partitionings
- Interoperability with other models

Some Reasons Why HPF Failed

- Vendors rushed products to market
- Immature compiler technology led to poor performance
  - —lots learned in dHPF project and others, but too late to save the language
- Lack of flexible data distributions
  - —need user-defined distributions
- Inconsistent compiler and runtime implementations
  - —tailor codes to leverage compiler strengths and avoid idiosyncrasies
  - —undermined creation of codes with portable high performance
- Paucity of good implementations of HPF Library
  - —users could not rely on having a good one
  - —missed opportunity: create a good open source implementation
    - – Thinking Machine's CMSSL might have become been a starting point
- Lack of patience by the user community

Outline

- High Performance Fortran
  - —background and motivation
  - —experiences compiling High Performance Fortran (HPF)
- Coarray Fortran
  - —original 1998 version
  - —Fortran 2008 - a standard with coarrays
- Coarray Fortran 2.0 (CAF 2.0)
  - —features
  - —experiences - HPC challenge benchmarks + performance
  - —implementation notes
  - —status
- Looking forward

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**

- **Examples: UPC, Titanium, Chapel, X10, Coarray Fortran**

Coarray Fortran (CAF)

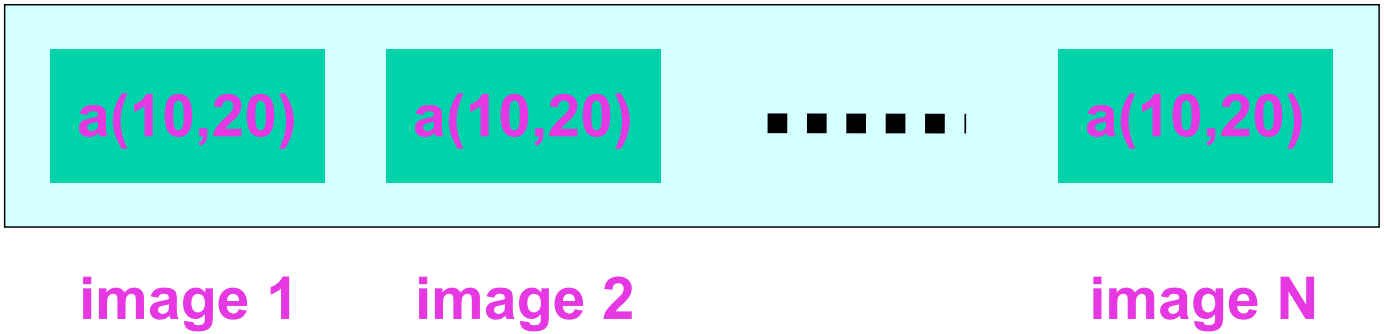
- Explicitly-parallel extension of Fortran 95 (Numrich & Reid 1998)
- Global address space SPMD parallel programming model
  - —one-sided communication
- Simple, two-level memory model for locality management
  - —local vs. remote memory
- Programmer has control over performance critical decisions
  - —data partitioning
  - —computation partitioning
  - —communication
  - —synchronization
- Suitable for mapping to shared and distributed memory systems

Coarray Fortran (1998)

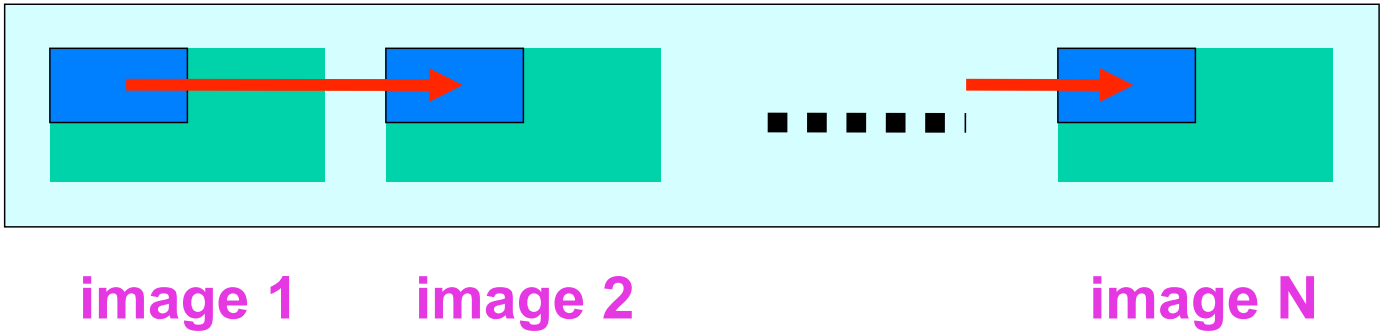
- SPMD process images
  - fixed number of images during execution: num_images()
  - images operate asynchronously: this_image()
- Both private and shared data
  - real x(20, 20) a private 20x20 array in each image
  - real y(20, 20) [*] a shared 20x20 array in each image
- Coarrays with multiple codimensions
  - real y(20, 20) [4,*]
- Simple one-sided shared-memory communication
  - x(:,j:j+2) = y(:,p:p+2) [r] copy columns from p:p+2 into local columns
- Synchronization intrinsic functions
  - sync_all – a barrier and a memory fence
  - sync_team(notify, wait)
    - notify = a vector of process ids to signal
    - wait = a vector of process ids to wait for
  - sync_memory – a memory fence
  - start_critical/end_critical
- Asymmetric dynamic allocation of shared data
- Weak memory consistency

One-sided Communication with Coarrays

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




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



A CAF Finite Element Example (Numrich)

```
subroutine assemble(start, prin, ghost, neib, x)
  integer :: start(:), prin(:), ghost(:), neib(:), k1, k2, p
  real :: x(:) [*]
  call sync_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
```

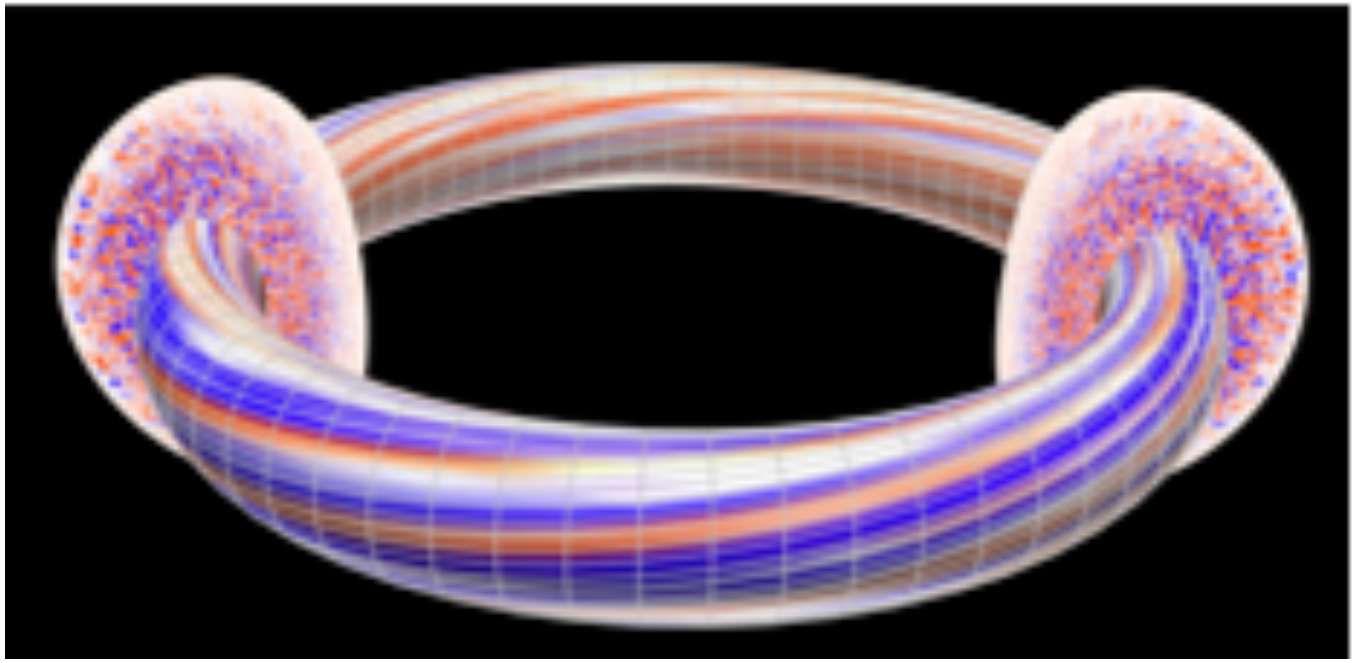
Fortran 2008

- SPMD process images
  - fixed number of images during execution: num_images()
  - images operate asynchronously: this_image()
- Both private and shared data
  - real x(20, 20) a private 20x20 array in each image
  - real y(20, 20) [*] a shared 20x20 array in each image
- Coarrays with multiple codimensions
  - real y(20, 20) [4,*]
- Simple one-sided shared-memory communication
  - x(:,j:j+2) = y(:,p:p+2) [r] copy columns from p:p+2 into local columns
- Synchronization intrinsic functions
  - sync all, sync images(image vector)
  - sync memory
  - critical sections, locks
  - atomic_define, atomic_ref
- Asymmetric dynamic allocation of shared data
- Weak memory consistency

CAF on Cray XE6 in 2011

GTS Particle Shifter (LBNL, Cray, PPPL) [SC11]

Preissl, Wichmann, Long, Shalf, Ethier, Koniges

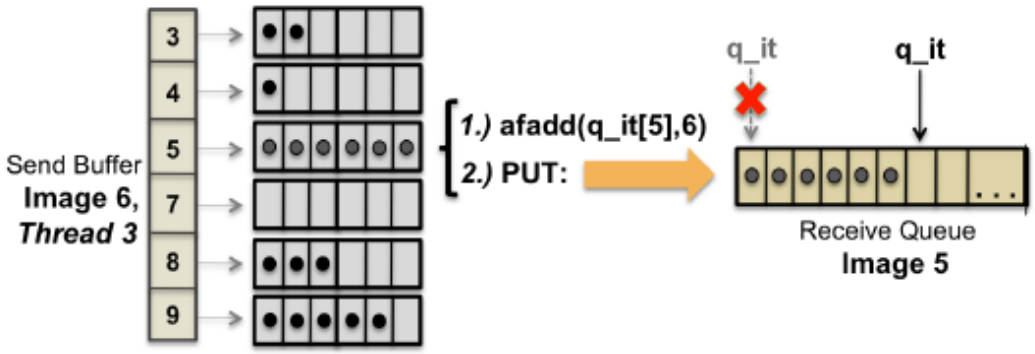


GTS Particle Shifter in MPI

two-sided bulk synchronous send

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

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

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

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

!(5) append remaining particles or fill holes
if(recv_length < shift) {
  append_particles(p_array,recv_buf) }
else { fill_remaining_holes(p_array,holes) }
```

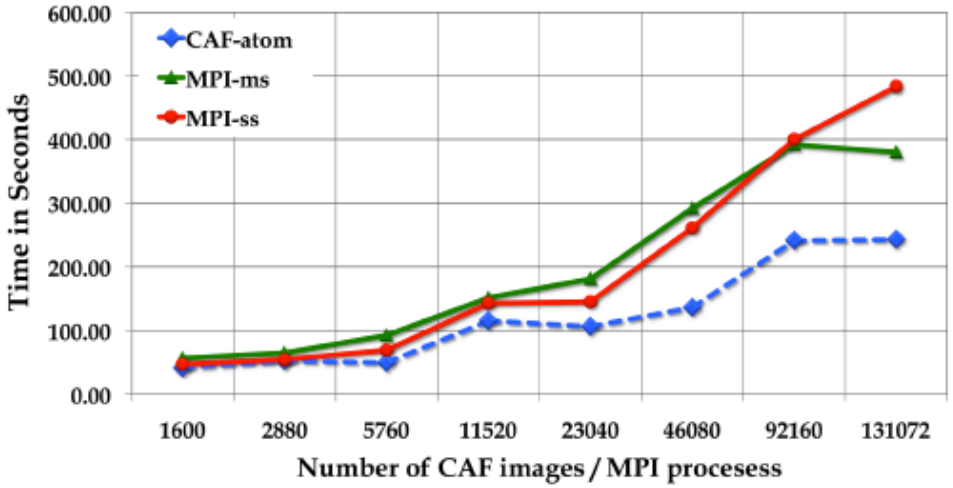
GTS Particle Shifter in CAF



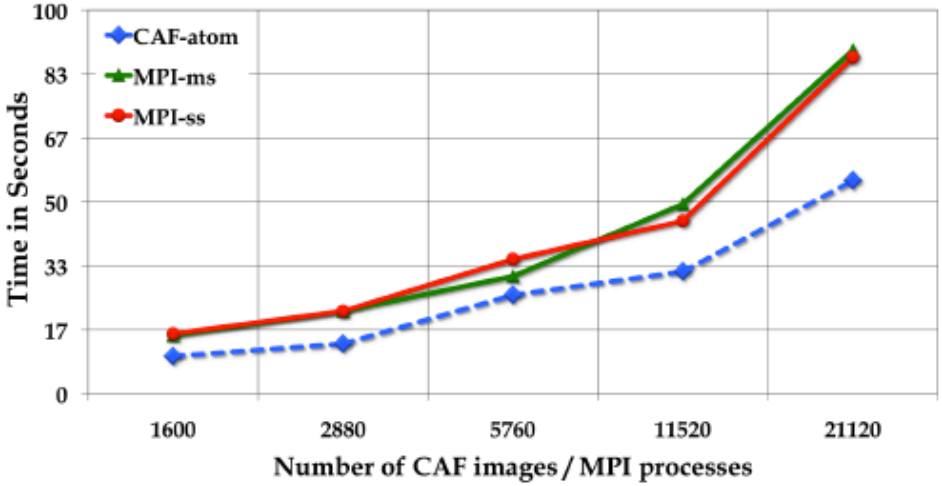
one-sided
asynchronous
push

```
!(1) compute shifted particles and fill the
!   receiving queues on destination images
!$omp parallel do schedule(dynamic,p_size/100)&
!$omp private(s_buf,buf_cnt) shared(recvQ,q_it)
do i=1,p_size
  dest=compute_destination(p_array(i))
  if(dest.ne.local_toroidal_domain) {
    holes(shift++)=i
    s_buf(dest,buf_cnt(dest)++)=p_array(i)
    if(buf_cnt(dest).eq.sb_size) {
      q_start=afadd(q_it[dest],sb_size)
      recvQ(q_start:q_start+sb_size-1)[dest] &
        =s_buf(dest,1:sb_size)
      buf_cnt(dest)=0   }   }
enddo

!(2) shift remaining particles
empty_s_buffers(s_buf)
!$omp end parallel

!(3) sync with images from same toroidal domain
sync images([my_shift_neighbors])

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

!(5) append remaining particles or fill holes
if(length_recvQ-min(length_recvQ,shift).gt.0) {
  append_particles(p_array,recvQ) }
else { fill_remaining_holes(p_array,holes) }
```

GTC Particle Shifter Performance



(a) 1 OpenMP thread per instance



(b) 6 OpenMP threads per instance

GTS Weak Scaling Performance

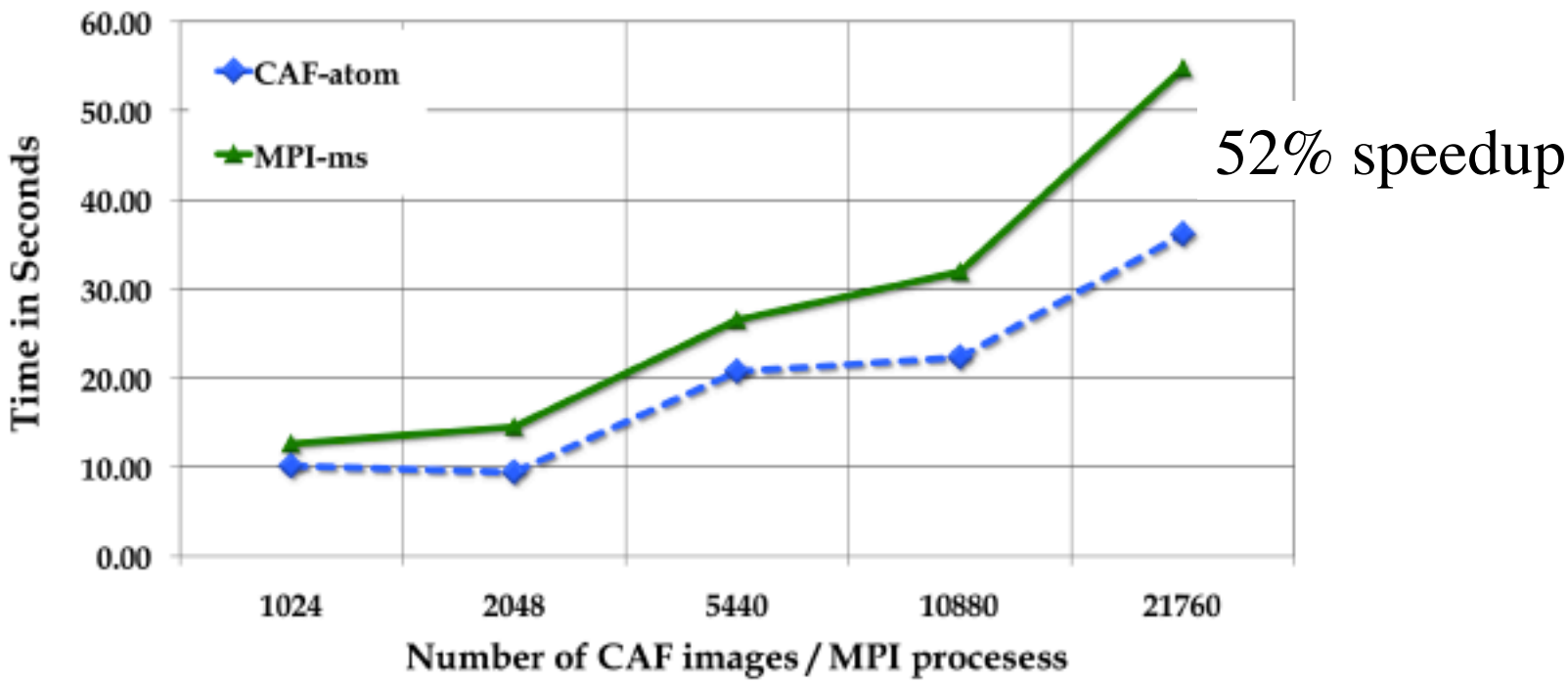


Figure 8: Weak scaling GTS experiments with CAF-atom & MPI-ms as particle shift algorithms (6 OpenMP threads per instance)

Why a New Vision?

Fortran 2008 characteristics

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

Coarray Fortran 2.0 Goals

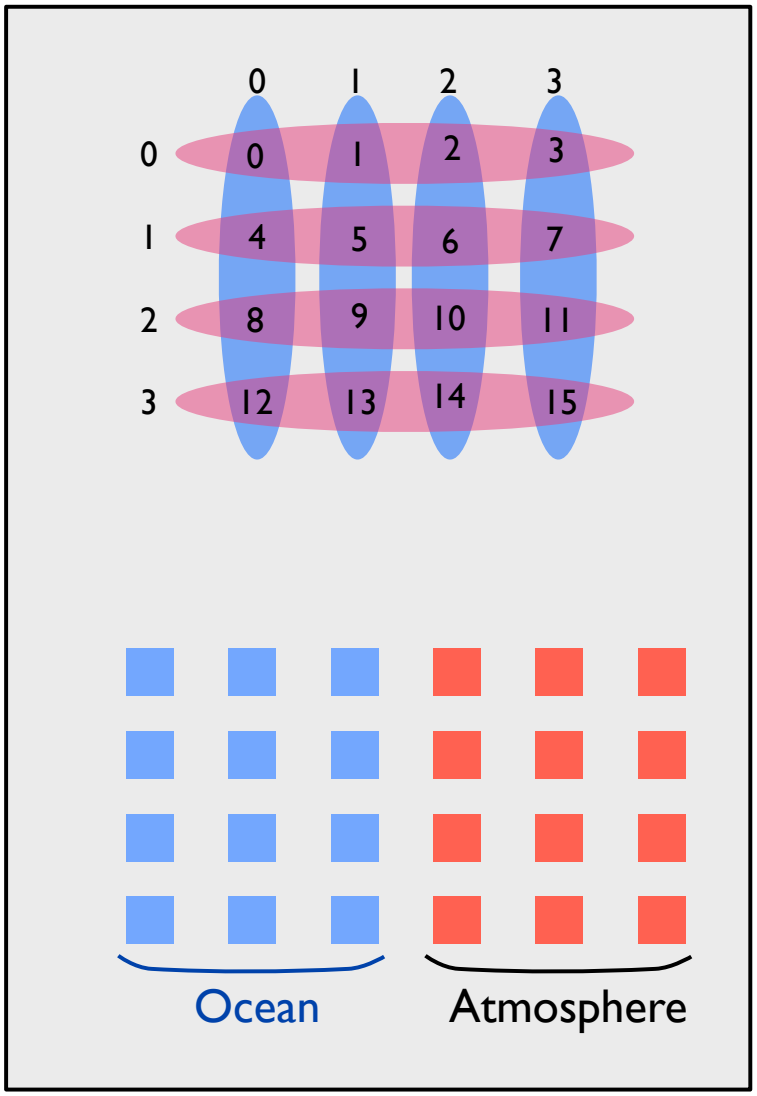
- Exploit multicore processors
- Enable development of portable high-performance programs
- Interoperate with legacy models such as MPI
- Facilitate construction of sophisticated parallel applications and parallel libraries
- Support irregular and adaptive applications
- Hide communication latency
- Colocate computation with remote data
- Scale to world's largest supercomputers

Coarray Fortran 2.0 (CAF 2.0)

- **Teams: process subsets, like MPI communicators**
  - **formation using team_split (like MPI_Comm_split)**
  - **collective communication**
- **Topologies**
- **Coarrays: shared data allocated across processor subsets**
  - **declaration:** `double precision :: a(:,:)[*]`
  - **dynamic allocation:** `allocate( a(n,m)[@row_team] )`
  - **access:** `x(:,n+1) = x(:,0)[mod(team_rank()+1, team_size())]`
- **Latency tolerance**
  - **hide: asynchronous copy, asynchronous collectives**
  - **avoid: function shipping**
- **Synchronization**
  - **event variables: point-to-point sync; async completion**
  - **finish: SPMD construct inspired by X10**
- **Copointers: pointers to remote data**

Process Subsets: Teams

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

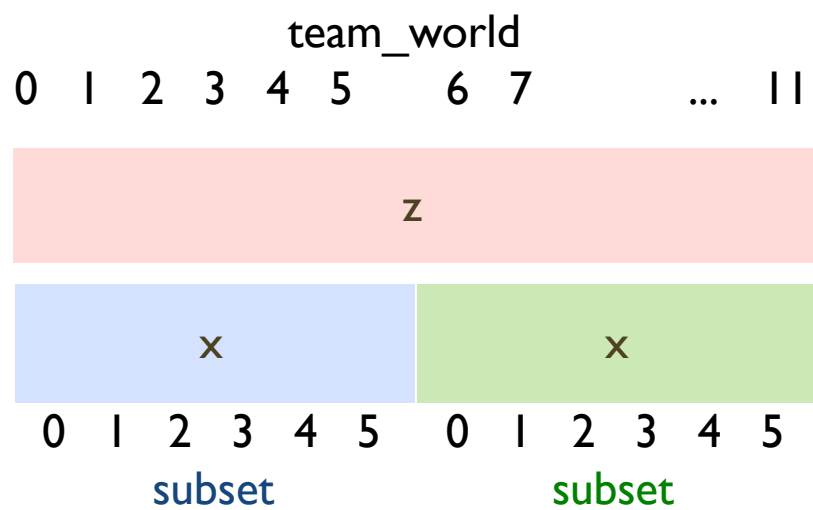
- **Predefined teams**
  - —team_world
  - —team_default
    - – used for any coarray operation that lacks an explicit team specification
- **Operations on teams**
  - —team_rank(team)
    - – returns the relative rank of the current image within a team
  - —team_size(team)
    - – returns the number of images of a given team
  - —team_split (existing_team, color, key, new_team)
    - – images supplying the same color are assigned to the same team
    - – each image's rank in the new team is determined by lexicographic order of (key, parent team rank)

Teams and Coarrays

- Coarray allocation occurs over teams
  - —storage is allocated over each member of the specified team
- Example
  - —integer :: a(:, :)[*]
  - —allocate (a (10, 100)[@team_world])
- Allocation is a collective operation
  - —barrier after an allocation to know that a coarray is available on other team members before accessing their data

Teams and Coarrays

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

! each image allocates a singleton for z
allocate( z(200,200) [@team_world] )

color = floor((2*team_rank(team_world)) / team_size(team_world))

! split into two subsets:
!    top and bottom half of team_world
team_split(team_world, color,  team_rank(team_world), subset)

! members of the two subset teams
! independently allocate their own coarray x
allocate( x(100,n)[@ subset])
```



Accessing Coarrays on Teams

- **Accessing a coarray relative to a team**
  - —x(i,j)[p@ocean] *! p names a rank in team ocean*

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

- **Simplifying processor indexing using "with team"**

```
with team atmosphere ! set team_default to atmosphere within
   ! p is wrt team atmosphere, q is wrt team ocean
   x(:,0)[p] = y(:)[q@ocean]
end with team
```

Communication Topologies

- **Motivation**
  - —a vector of images may not adequately reflect their logical communication structure
  - —multiple co-dimensions only support grid-like logical structures
  - —want a single mechanism for expressing more general structures
- **Topology**
  - —shamelessly patterned after MPI Topologies
  - —logical structure for communication within a team
  - —more expressive than multiple codimensions

Using Topologies

- Creation
  - —Cartesian: topology_cartesian((/e1,e2,.../), (/ w1, w2, ... /))
  - —Graph: topology_graph(e)
    - graph_neighbor_add(g,e,n,nv)
    - graph_neighbor_delete(g,e,n,nv)
- Binding: topology_bind(team,topology)
- Accessing a coarray using a topology
  - —Cartesian
    - array(:) [+(i1, i2, ..., in)@ocean] ! relative index wrt self in team ocean
    - array(:) [(i1, i2, ..., in)@ocean] ! absolute index wrt 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

Synchronization

- Point-to-point synchronization via event variables
  - —like counting semaphores
  - —each variable provides a synchronization context
  - —a program can use as many events as it needs
    - – user program events are distinct from library events
  - —event_notify / event_wait
  - —event_notify is non-blocking
- Lockset: ordered sets of locks
  - —convenient to avoid deadlock when locking/unlocking multiple locks -- uses a canonical ordering

Latency Tolerance

- <u>Hide</u> latency for accessing remote data by overlapping it with computation
- <u>Avoid</u> exposed latency when manipulating remote data structures
- Asynchrony models
  - —explicit: signal an event to indicate when an asynchronous operation has completed
  - —implicit: programmer specifies a point when program must block until outstanding asynchronous operations have completed
  - —interactions between models are subtle!

Predicated Asynchronous Copy

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

- **var_dest**: data target
- **var_src**: data source
- **ev_src**: event to be triggered when the read of var_src is complete
- **ev_dest**: event to be triggered when the write of var_dest is complete
- **ev_pred**: optional event indicating that copy may proceed

Collective Communication

- Why provide collectives?
  - —application programmers want them
  - —avoid having programmers roll their own (non scalable) versions
- Collective operations
  - —alltoall, barrier, broadcast, all/gather, permute, all/reduce, scatter, segmented/scan, shift
- User-defined reduction operators
- Potential flavors
  - —two-sided synchronous
    - – all execute it together
  - —two-sided asynchronous
    - – all team members will execute a call to start it
    - – all will later wait for it to complete
  - —one-sided synchronous: one starts it and blocks until done
  - —one-sided asynchronous: one starts it and later finishes it

Two-sided vs. One-sided Collectives

- **Issues with one-sided collectives**
  - —where does the data get delivered?
    - – does the initiator specify an address for each recipient?
    - – does data get delivered to the same offset in a coarray for each recipient?
  - —how do I know when I can overwrite it?
- **Two-sided collectives address these issues**
  - —each participant receiving a value specifies where to deliver it
  - —each participant can decide how many asynchronous collectives can be outstanding at once
    - – based on the number of buffers available for receiving values
  - —an asynchronous collective initiated before some recipients are ready will have (at least part of) its execution deferred until recipients are ready

Coarray Fortran 2.0 supports two-sided synchronous and asynchronous collectives

Asynchronous Collective Operations

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

Function Shipping

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

```
finish (team)
  spawn f(table(i,j)[p], n)[p]
  ...
end finish
```

CAF 2.0 Finish

- X10 finish

```
finish {
    ...
}
```

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

- CAF 2.0 finish
  - SPMD construct defined over teams

```
finish (team)
  ...
end finish
```

  - all members of a team enter a finish block
  - any functions that team members ship to one another from within a finish block must complete before any node will exit the corresponding finish block

CAF 2.0 Cofence

- **Finish is a heavyweight mechanism**
  - **—manages global completion across a team**
  - **—sometimes only local completion is needed**
    - **e.g. an asynchronous copy has delivered a value locally**
- **Cofence manages local completion**
  - **—asynchronous copies with implicit completion**
  - **—asynchronous collectives with implicit completion**
- **Can use a cofence within a finish block to demand early completion of asynchronous operations**

Local Teams

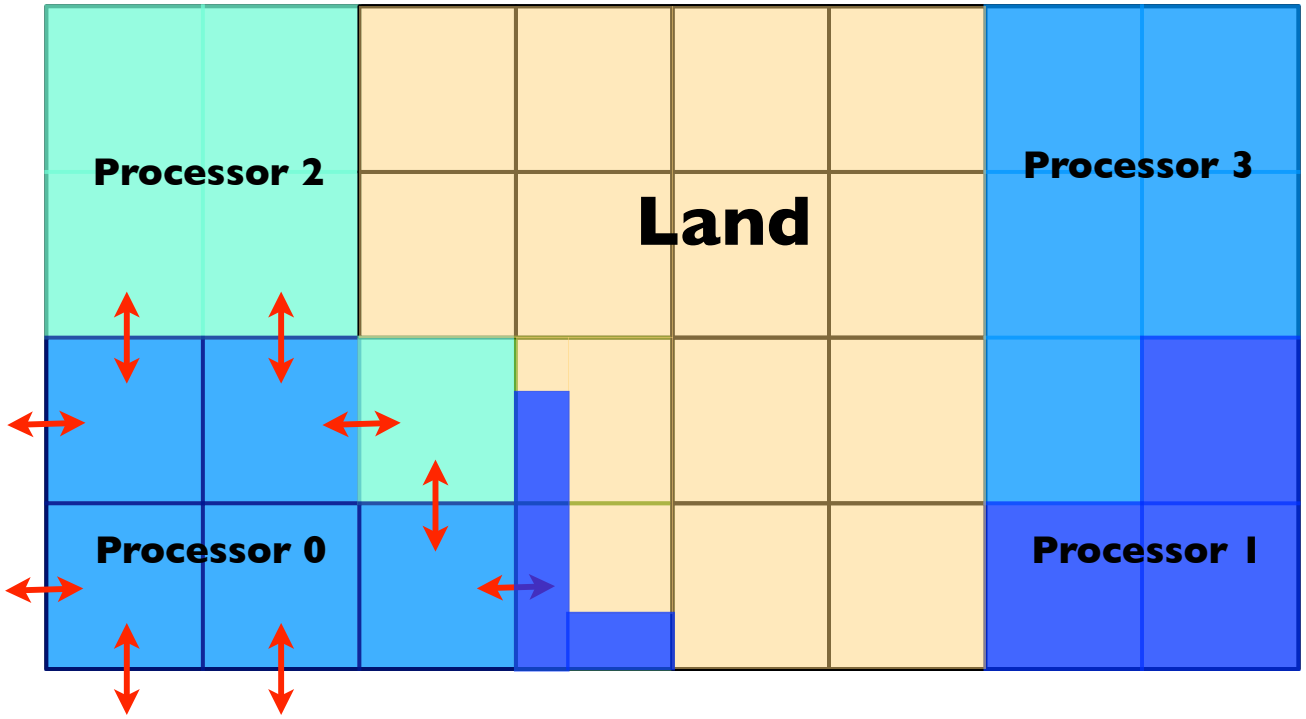
- Useful to have teams within a locality domain
  - —bind processes to locality domains (e.g., sockets)
- Add a keyword to a team declaration if it is a local team
- Automatically generate shared-memory communication within such teams

Copointers: Global Pointers

- Motivation: support linked data structures
- **copointer** attribute enables association with remote shared data
- **imageof(x)**returns the image number for **x**
  - useful to determine whether copointer x is local

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

allocate(a(1:20, 1:30)[@ team_world]

! associate copointer x with a
! remote section of a coarray
x => a(4:20, 2:25)[p]

! imageof intrinsic returns the target
! image for x
prank = imageof(x)

x(7,9) = 4      ! assumes target of x is local
x(7,9)[ ] = 4  ! target of x may be remote
```

LANL's Parallel Ocean Program



- Data partitioning of ocean blocks
  - — cartesian, balanced, space-filling curve distributions
- Data communication
  - — boundary updates between neighboring processors
  - — collective communications (gather, scatter, reduction)
- Different boundary types
  - — cyclic, closed, tripole

MPI

```
! post a receive
do n=1,in_bndy%nmsg_ew_rcv
   bufsize = ny_block*nghost*in_bndy%nblocks_ew_rcv(n)
   call MPI_IRECV(buf_ew_rcv(1,1,1,n), bufsize, mpi_dbl,   &
                  in_bndy%ew_rcv_proc(n)-1,                &
                  mpitag_bndy_2d + in_bndy%ew_rcv_proc(n), &
                  in_bndy%communicator, rcv_request(n), ierr)
end do

! pack data and send data
do n=1,in_bndy%nmsg_ew_snd
   bufsize = ny_block*nghost*in_bndy%nblocks_ew_snd(n)

   partner = in_bndy%ew_snd_proc(n)-1
   do i=1,in_bndy%nblocks_ew_snd(n)
      ib_src    = in_bndy%ew_src_add(1,i,n)
      ie_src    = ib_src + nghost - 1
      src_block = in_bndy%ew_src_block(i,n)
      buf_ew_snd(:,:,i,n) = ARRAY(ib_src:ie_src,:,src_block)
   end do

   call MPI_ISEND(buf_ew_snd(1,1,1,n), bufsize, mpi_dbl, &
                  in_bndy%ew_snd_proc(n)-1, &
                  mpitag_bndy_2d + my_task + 1, &
                  in_bndy%communicator, snd_request(n), ierr)

end do

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

do n=1,in_bndy%nmsg_ew_rcv
   partner = in_bndy%ew_rcv_proc(n) - 1
   do k=1,in_bndy%nblocks_ew_rcv(n)
      dst_block = in_bndy%ew_dst_block(k,n)
      ib_dst = in_bndy%ew_dst_add(1,k,n)
      ie_dst = ib_dst + nghost - 1
      ARRAY(ib_dst:ie_dst,:,dst_block) = buf_ew_rcv(:,:,k,n)
   end do
end do

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

CAF 2.0

```
type :: outgoing_boundary
   double, copointer :: remote(:,:,:)[*]
   double, pointer :: local(:,:,:)
   event :: snd_ready[*]
   event, copointer :: src_done[*]
end type

type :: incoming_boundary
   event, copointer :: dest_ready[*]
   event :: dest_done[*]
end type

type :: boundaries
   integer :: in_faces, out_faces
   type(outgoing_boundary) :: outgoing(:)
   type(incoming_boundary) :: incoming(:)
end type

! initialize outgoing boundary
!    set remote to point to a partner's incoming boundary face
!    set local to point to one of my outgoing boundary faces
!    set snd_done to point to rcv_done of a partner's incoming boundary

! initialize incoming boundary
!    set my face's rcv_ready to point to my partner face's snd_ready

! notify each partner that my face is ready
do face=1,bndy%in_faces
   call event_notify(bndy%incoming(face)%dest_ready[])
end do

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

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

Multithreading

- Where can asynchronous threads of control arise in CAF 2.0?
  - —spawned procedures
  - —parallel loops
    - Fortran 90's "do concurrent"
- Work in progress to employ Cilk-like lazy multithreading
  - —generate continuations when spawning functions
  - —generate a continuation when blocking for synchronization

Outline

- High Performance Fortran
  - —background and motivation
  - —experiences compiling High Performance Fortran (HPF)
- Coarray Fortran
  - —original 1998 version
  - —Fortran 2008 - a standard with coarrays
- Coarray Fortran 2.0 (CAF 2.0)
  - —features
  - —experiences - HPC challenge benchmarks + performance
  - —implementation notes
  - —status
- Looking forward

HPC Challenge Benchmark Goal: Productivity

- Priorities, in order
  - —performance
  - —source code volume
- Productivity = performance / (lines of code)
- Implications
  - —EP STREAM Triad
    - – outlined a loop to assist compiler optimization
  - —Randomaccess
    - – used software routing for higher performance
  - —FFT
    - – blocked packing/unpacking loops for bitreversal (8x gain for packing kernel)
  - —HPL
    - – tuned code to make good use of the memory hierarchy

EP STREAM Triad

```
double precision, allocatable :: a(:)[*], b(:)[*], c(:)[*]
...
! each processor in the default team allocates their own array parts
allocate(a(local_n)[], b(local_n)[], c(local_n)[])
...
! perform the calculation repeatedly to get reliable timings
do round = 1, rounds
  do j = 1, rep
    call triad(a,b,c,local_n,scalar)
  end do
  call team_barrier() ! synchronous barrier across the default team
end do
...
! perform the calculation with top performance
! assembly code is identical to that for sequential Fortran
subroutine triad(a, b, c, n ,scalar)
  double precision :: a(n), b(n), c(n), scalar
  a = b + scalar * c  ! EP triad as a Fortran 90 vector operation
end subroutine triad
```

Randomaccess

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

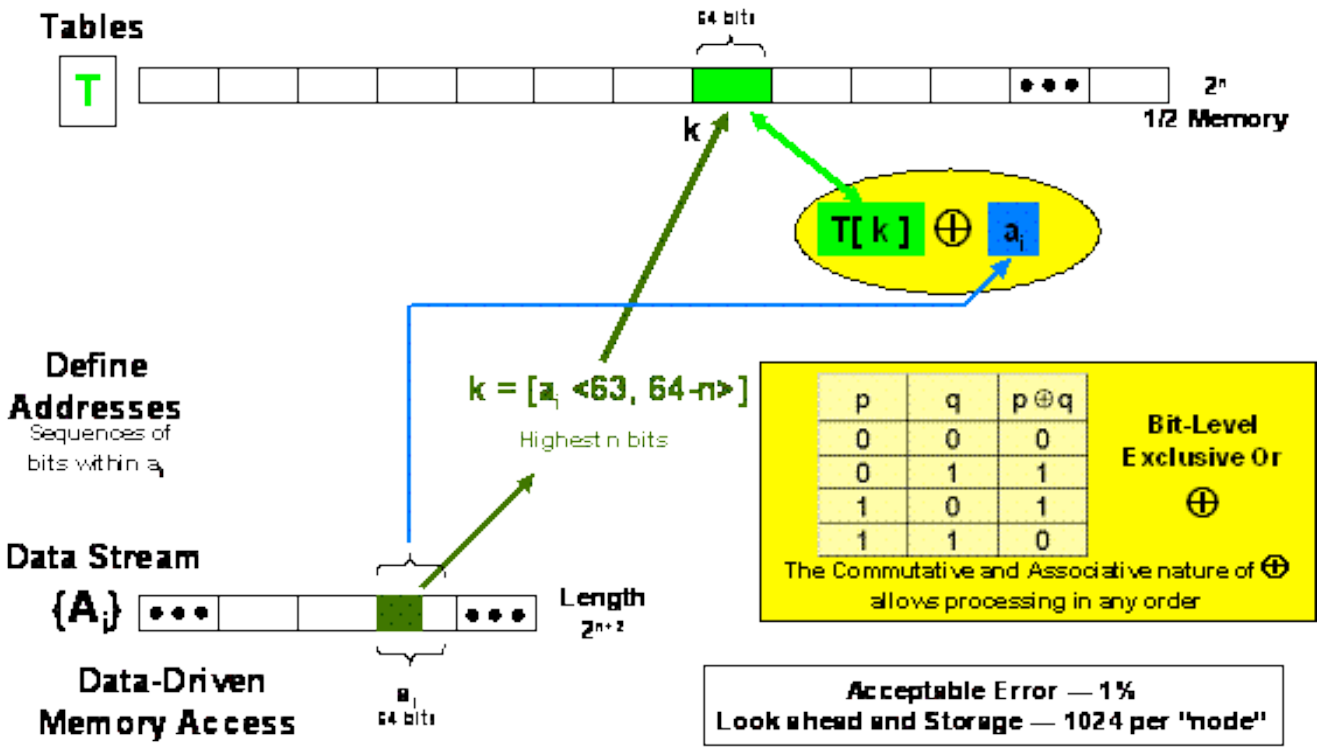


Figure credit: UTK

Randomaccess Software Routing

```
event, allocatable :: delivered(:)[*],received(:)[*] !(stage)
integer(i8), allocatable :: fwd(:,:,:)[*] ! (#,in/out,stage)
...
! hypercube-based routing: each processor has 1024 updates
do i = world_logsize-1, 0, -1  ! log P stages in a route
  ...
  call split(retain(:,last), ret_sizes(last), &
             retain(:,current), ret_sizes(current), &
             fwd(1:,out,i), fwd(0,out,i), bufsize, dist)

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

  count = fwd(0,out,i)
  event_wait(received(i)) ! ensure buffer is empty from last route
  fwd(0:count,in,i)[partner] = fwd(0:count,out,i) ! send to partner
  event_notify(delivered(i)[partner]) ! notify partner data is there
  ...
end do
```

1

2

HPL

- **Block-cyclic data distribution**
- **Team based collective operations along rows and columns**

  —synchronous max reduction down columns of processors

  —asynchronous broadcast of panels to all processors

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

FFT

- Radix 2 1D FFT implementation
- Block distribution of array “c” across all processors
- Computation
  - —permute elements: c = (/ c(bitreverse(i), i = 0, n-1 /)
    - 3 parts: pack data for all-to-all; team collective all-to-all; unpack data locally
  - —FFT is log N stages
    - compute (log N - log P) stages of the FFT locally
    - transpose the data so that each processor has elements ≡ rank mod P

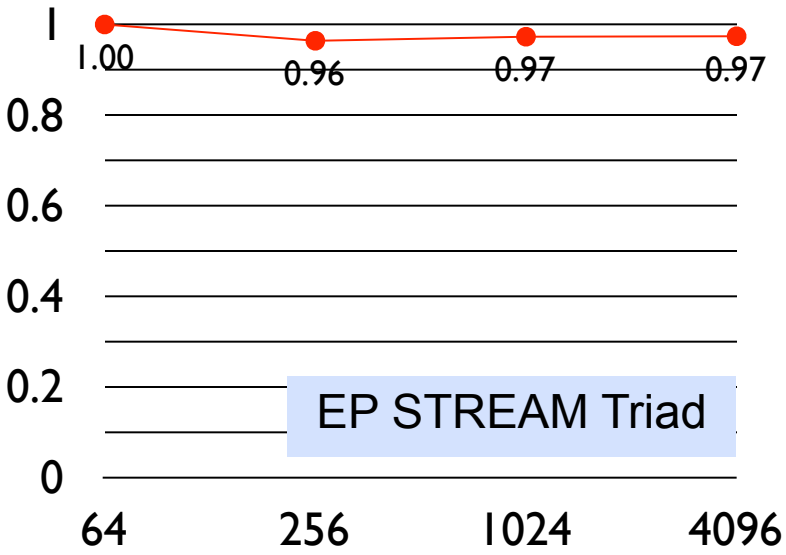
      block distribution → cyclic distribution
    - compute the remaining log P stages of the FFT locally
    - transpose the data back to its original order

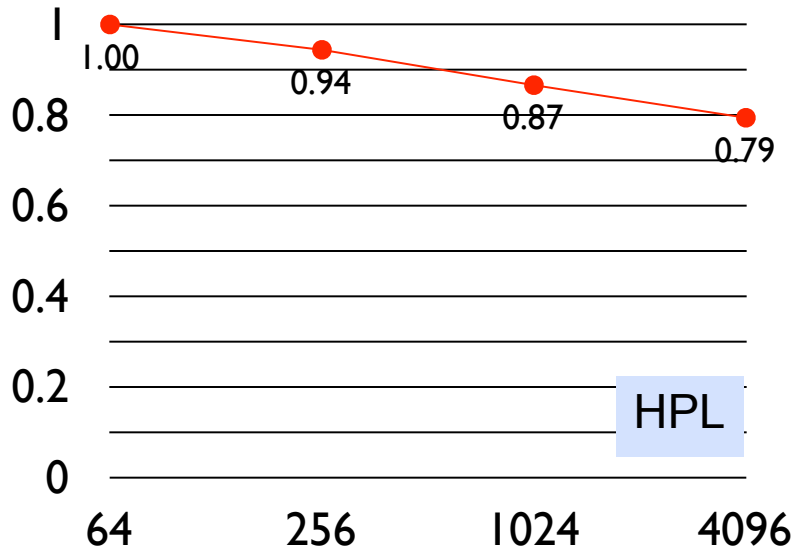
      cyclic distribution → block distribution

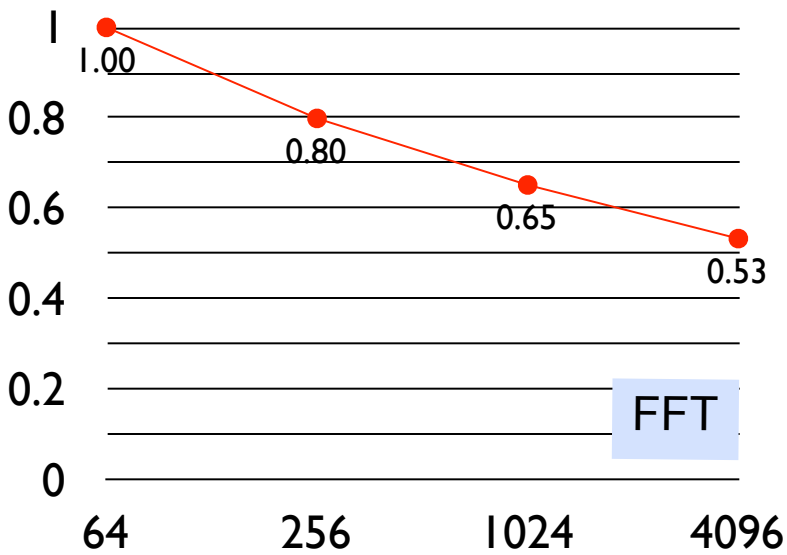
Experimental Setup

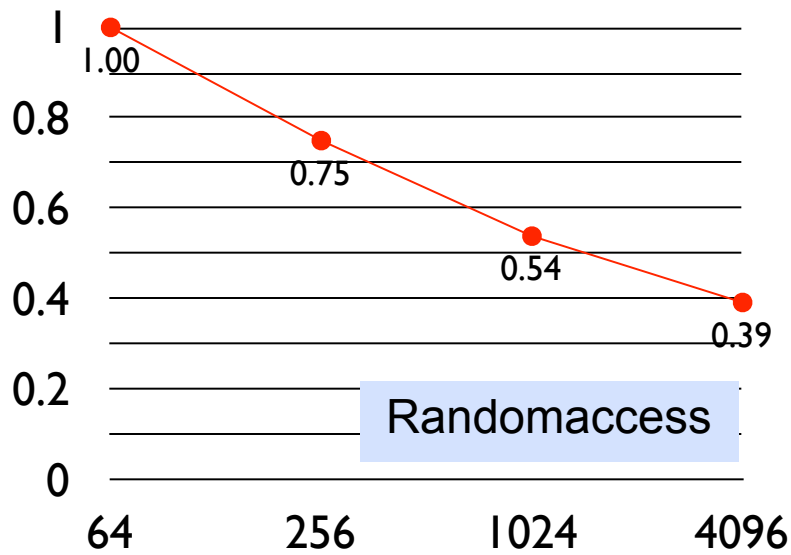
- Coarray Fortran 2.0 by Rice University
  - —source to source compilation from CAF 2.0 to Fortran 90
    - – generated code compiled with Portland Group's pgf90
  - —CAF 2.0 runtime system built upon GASNet (version 1.14.2)
  - —scalable implementation of teams, using O(log P) storage
- Experimental platform: Cray XT
  - —systems
    - – Franklin at NERSC
      - 2.3 GHz AMD "Budapest" quad-core Opteron, 2GB DDR2-800/core
    - – Jaguar at ORNL
      - 2.1 GHz AMD "Budapest" quad-core Opteron, 2GB DDR2-800/core
  - —network topology
    - – 3D Torus based on Seastar2 routers
    - – OS provides an arbitrary set of nodes to an application

Scalability: Relative Parallel Efficiency









Productivity = Performance / SLOC

Performance (Cray XT4)

	HPC Challenge Benchmark			
# of cores	STREAM Triad † (TByte/s)	RandomAccess*(GUP/s)	Global HPL † (TFlop/s)	Global FFT † (GFlop/s)
64	0.14	0.08	0.36	6.69
256	0.54	0.24	1.36	22.82
1024	2.18	0.69	4.99	67.80
4096	8.73	2.01	18.3	187.04

*Measured on Jaguar † Measured on Franklin

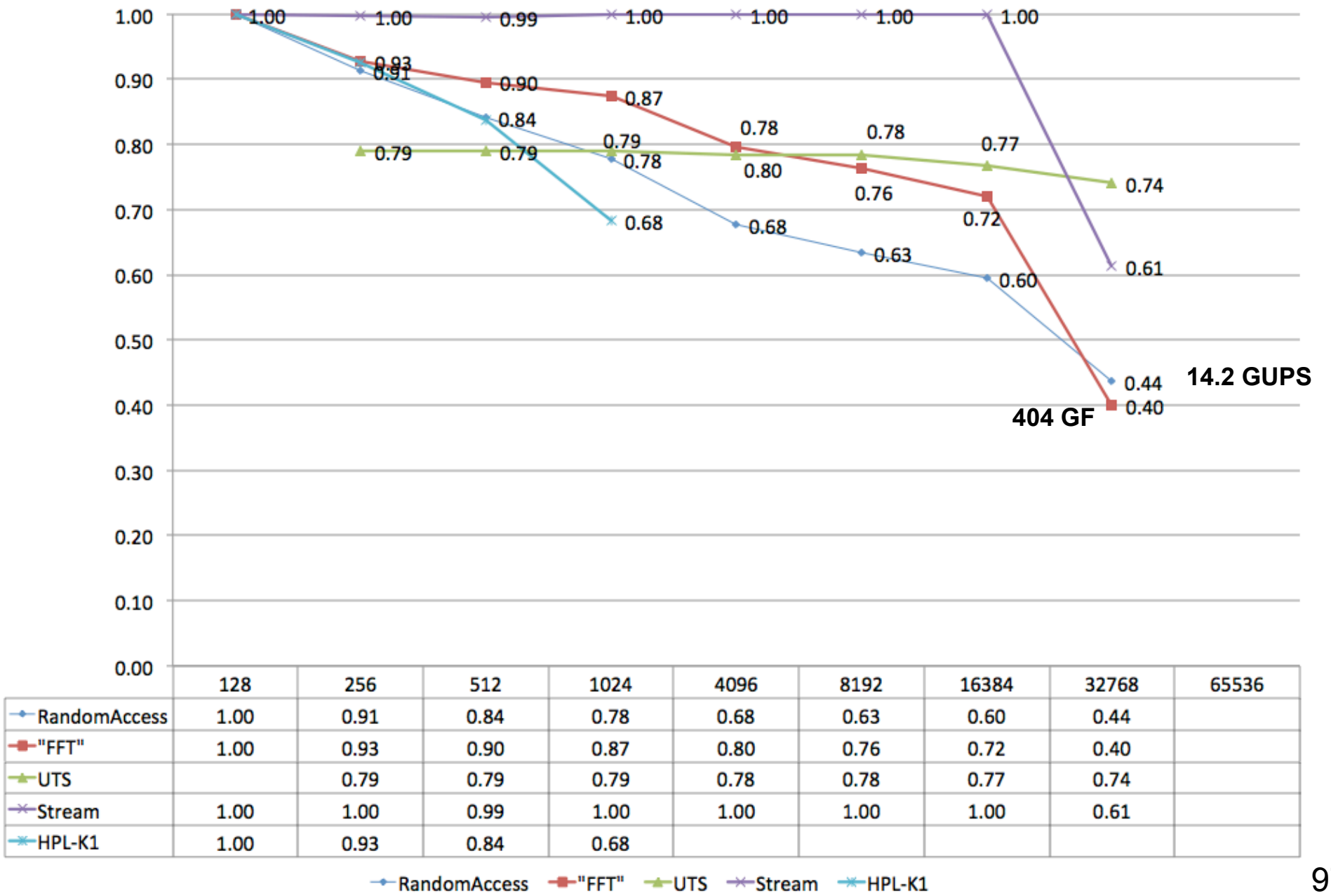
Source lines of code

HPC Challenge Benchmark	Source Lines of Code	Reference SLOC
Randomaccess	409	787
EP STREAM Triad	58	329
Global HPL	786	8800
Global FFT	~390	1130

Notes

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

Scalability: Relative Parallel Efficiency

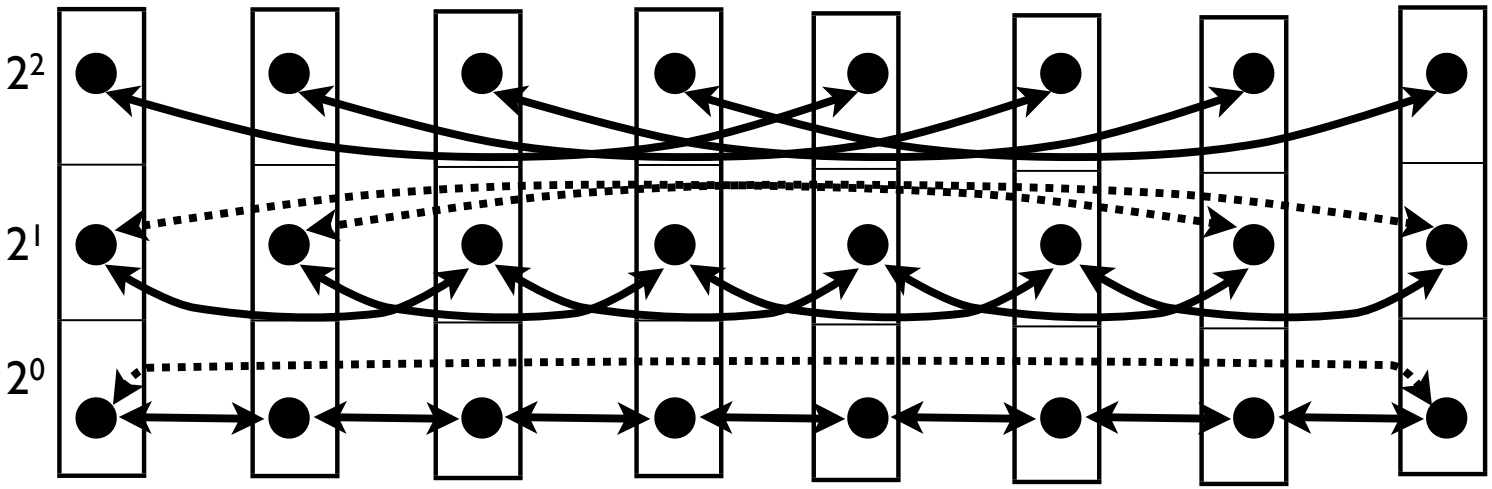


	128	256	512	1024	4096	8192	16384	32768	65536
RandomAccess	1.00	0.91	0.84	0.78	0.68	0.63	0.60	0.44	
"FFT"	1.00	0.93	0.90	0.87	0.80	0.76	0.72	0.40	
UTS		0.79	0.79	0.79	0.78	0.78	0.77	0.74	
Stream	1.00	1.00	0.99	1.00	1.00	1.00	1.00	0.61	
HPL-K1	1.00	0.93	0.84	0.68					

CAF 2.0 Early Experiences Summary

- **A viable programming model for scalable parallel computing**
  - **—expressive**
  - **—easy to use**
- **Significantly smaller code than MPI, yet achieves scalable high performance**
  - **—prototype implementation scales to thousands of nodes**
  - **—scalable high performance, but not *exceptional* performance**
- **Significant increase in productivity measured by performance per line of code**

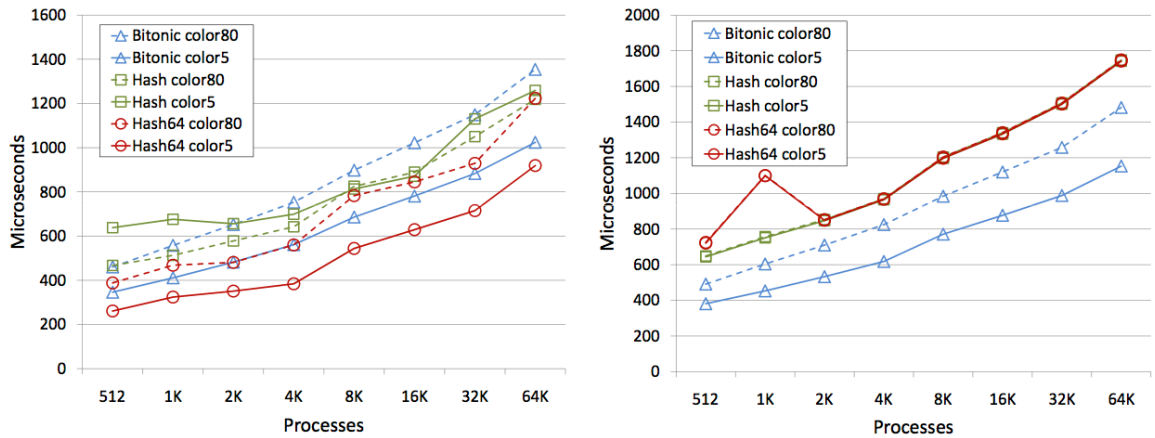
CAF 2.0 Team Representation



- **Designed for scalability: representation is O(log S) per node for a team of size s**
- **Based on the concept of pointer jumping**
- **Pointers to predecessors and successors at distance $i = 2^j$, $j = 0 .. \lceil \log S \rceil$**

CAF Team Split

- Sort (color, key, rank) tuples using parallel bitionic sort
- Left and right shift operations to determine team boundaries
- Segmented scans to compute one's rank within a team
  - compute team size and rank and disseminate first rank with a forward scan
  - segmented broadcast in the reverse direction informs each rank of the size and last member
- Subteams can be assembled
  - its left and right neighbors at distance one in the circular order of its subteam
  - the size of the subteam, and its rank in the subteam.
- Space and time: $O(\log^2 P)$
  - bitonic sort



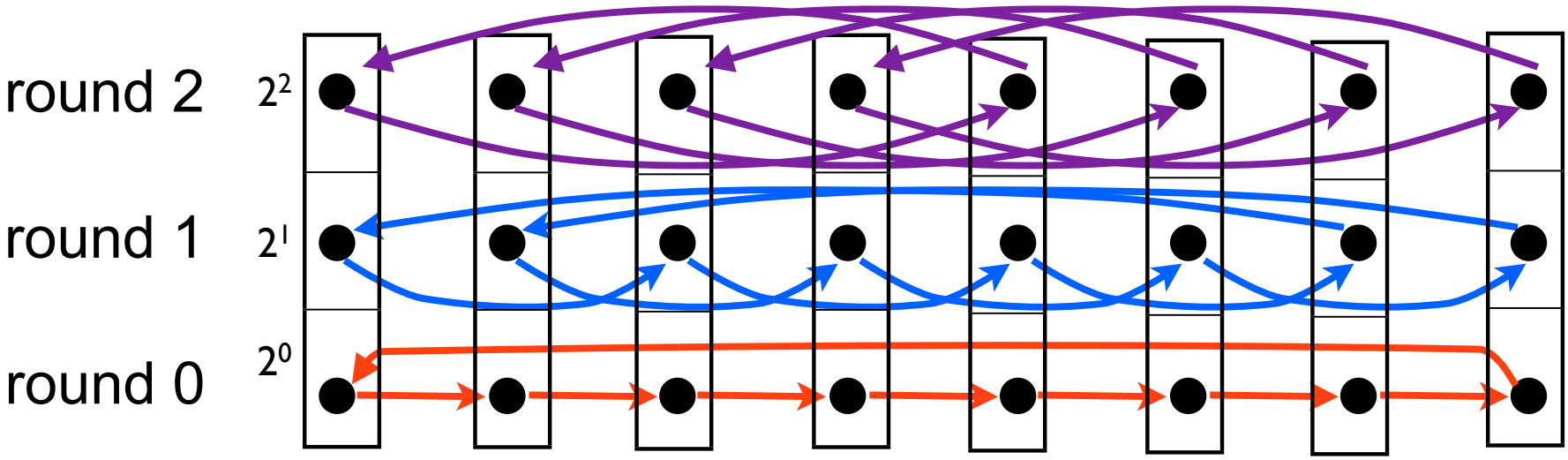
(a) One color per block of 16 processes (b) Two colors total

A. Moody, et al. Exascale Algorithms for Generalized MPI_Comm_split. EuroMPI 2011.

Collective Example: Barrier

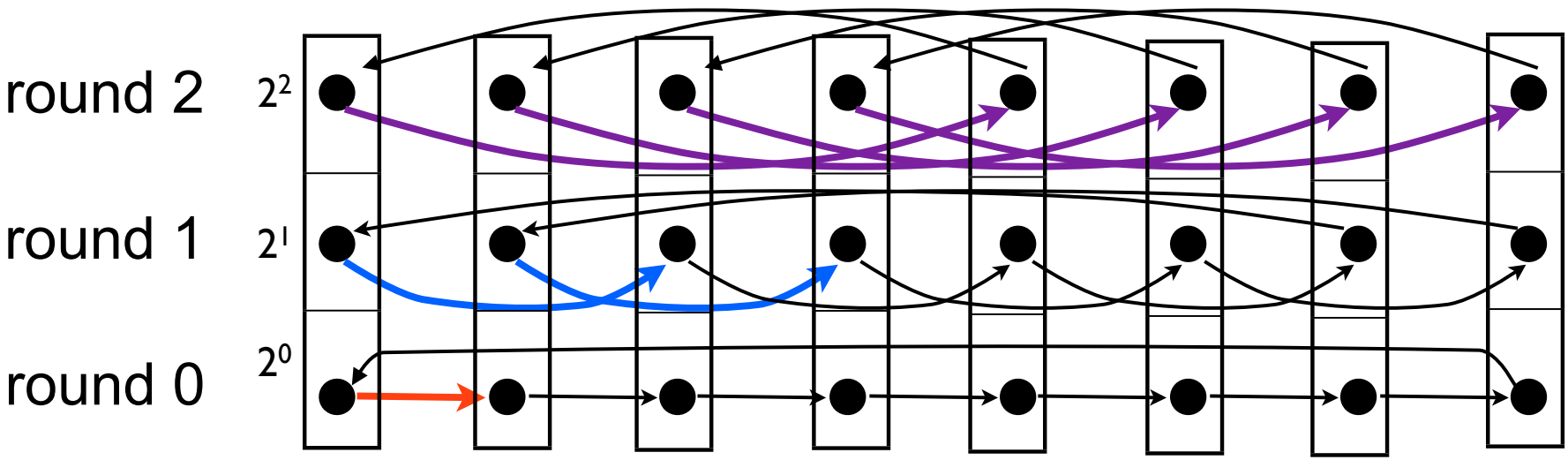
Dissemination algorithm

```
for k = 0 to ⌈log2 P⌉
   processor i signals processor (i + 2^k) mod P with a PUT
   processor i waits for signal from (i - 2^k) mod P
```



Collective Example: Broadcast

Binomial Tree



Strengths and Weaknesses of CAF 2.0

- **Strengths**
  - —provides full control over data and computation partitioning
  - —admits sophisticated parallelizations
  - —compiler and runtime systems are tractable
  - —yields scalable high performance **today** with careful programming
- **Weaknesses**
  - —users code data movement and synchronization
    - – significantly harder than HPF
  - —optimizing performance can require careful parallel programming
    - – overlapping communication and computation may require managing multiple communication buffers
    - – hiding latency requires
      - using non-blocking primitives for data movement and synchronization
      - overlapping latency of communication with computation
      - managing the completion of asynchronous operations

Lessons from Experience with CAF 2.0

- **Need the right communication primitives to support the language implementation**
  - **missing: one-sided "put with notify"**
    - **notify should be an atomic add**
- **Flow control of one-sided communication is an issue for current architectures**
- **Integrated progress engine between language runtime and underlying communication layer is a key to good performance**

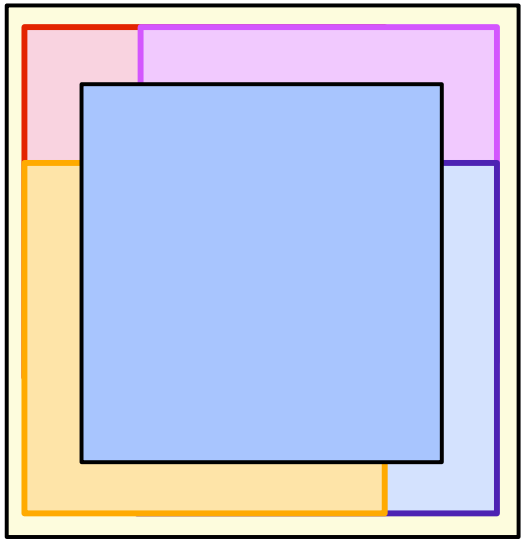
Implementation Status

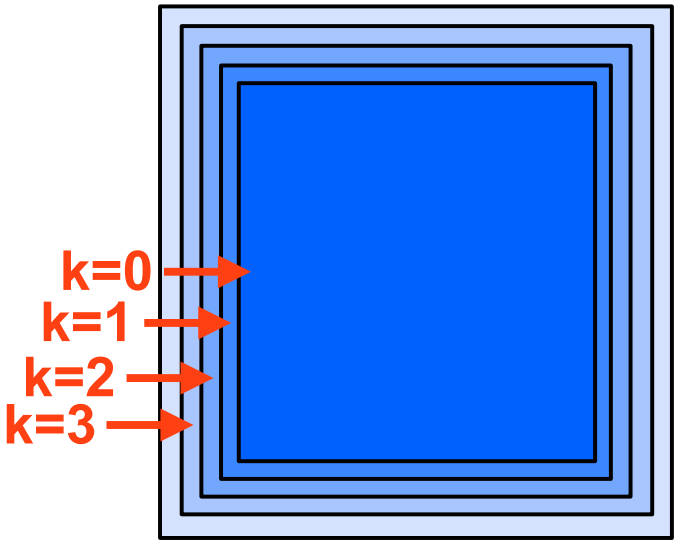
- **Source-to-source translator is a work in progress**
  - —requires no vendor buy-in
  - —delivers node performance of mature vendor compilers
  - —ongoing work to improve Fortran coverage in ROSE
- **Ongoing work**
  - —copointers
  - —lazy multithreading
  - —coarray binding interface for inter-team communication
  - —graph topology for managing irregular communication patterns

Looking Forward

- **Communication avoiding algorithms**
  - —broad class of strategies that communicate asymptotically less than their conventionsl counterparts
- **Examples**
  - —time skewing, e.g., overlapped tiling
  - —new algorithms for linear algebra, e.g., matrix multiply

Vision for Overlapped Tiling in dHPF





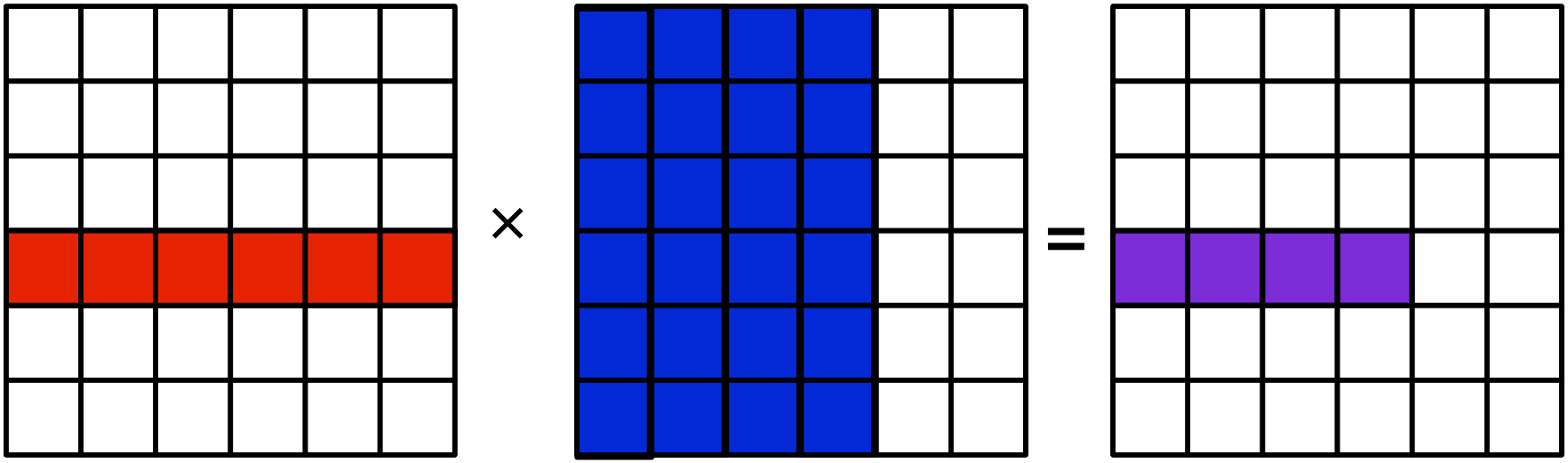
Assumptions: 2D BLOCK distribution for A SHADOW (4:4,4:4)

```
!HPF$ REFLECT
  do k =3, 0, -1
!HPF$ on home a(i+k, j-k), a(i+k,j+k), a(i-k,j+k), a(i-k,j-k) begin
!HPF$ local begin
    do i = 1, n-1
      a(i,j) = a(i, j) + a(i,j-1)+a(i,j+1)+a(i-1,j)+a(i+1,j)
     enddo
!HPF$ local end
!HPF$ on home end
  enddo
```

Overlapped tiling has been used in code generation for GPUs

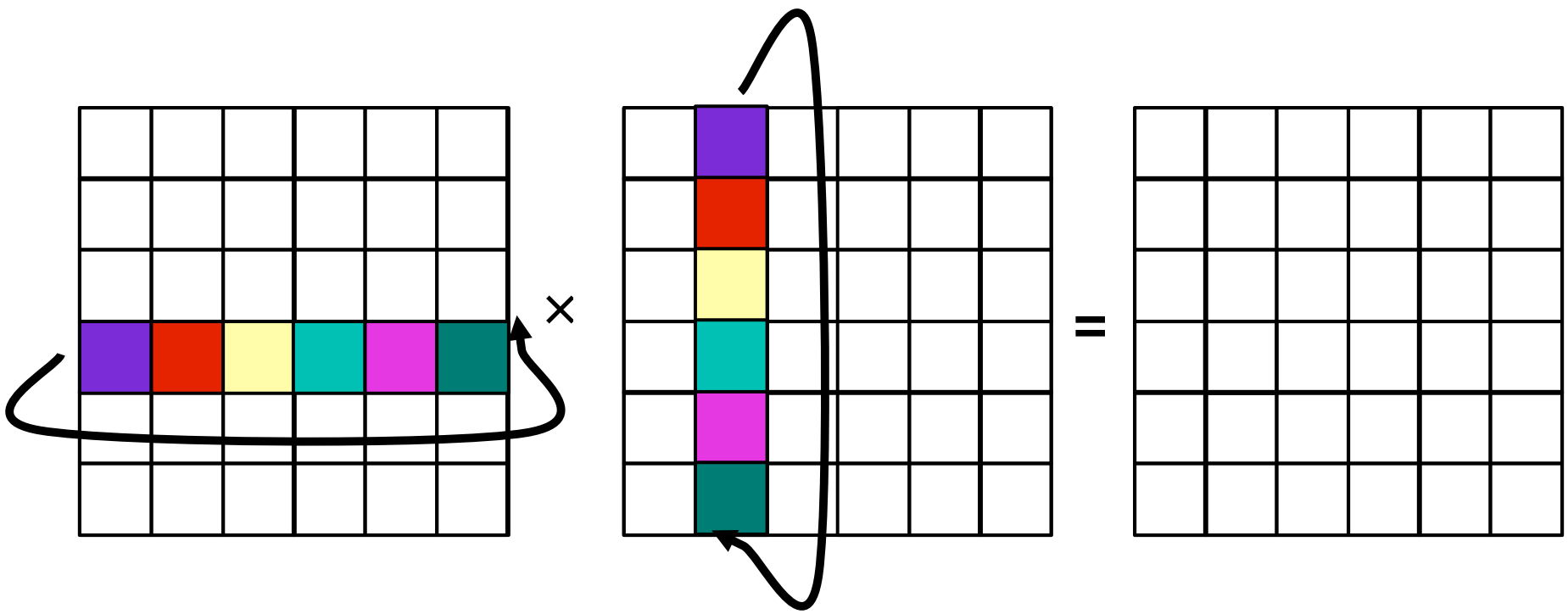
Matrix Multiplication

Consider data needed for output matrix block shown in purple



Cannon's Matrix Multiplication

Initial State



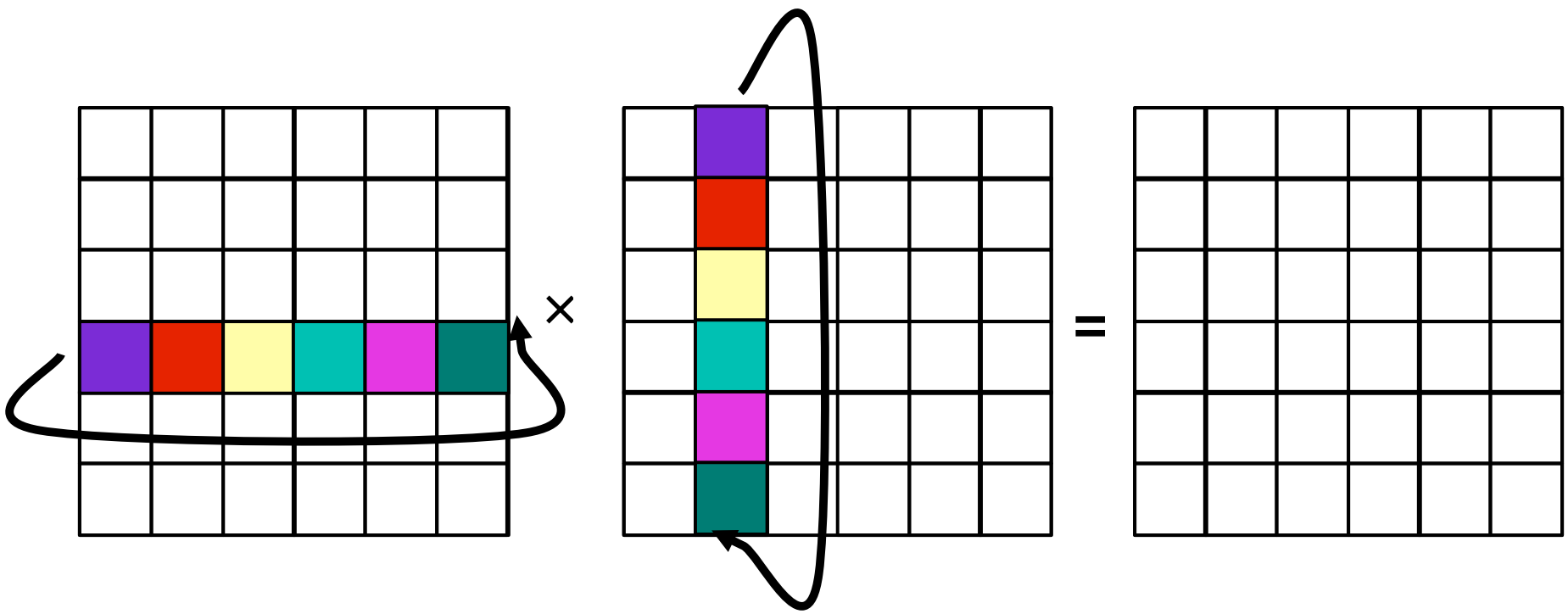
A, B are distributed on $\sqrt{p}$ x$\sqrt{p}$ processor grid

Cannon's Matrix Multiplication

Stage 1: Perform Alignment

shift A_{ij} left by I

shift B_{ij} up by j

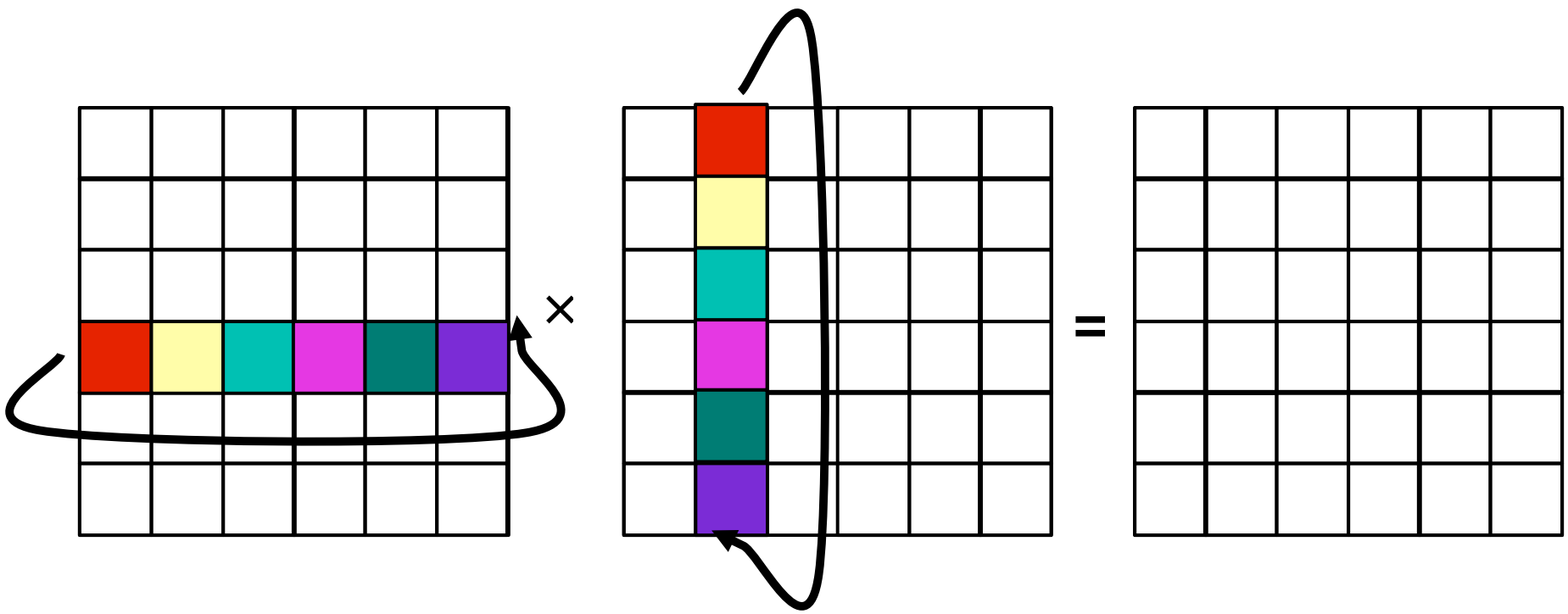


Cannon's Matrix Multiplication

Step 1: Perform Alignment

shift A_{ij} left by I

shift B_{ij} up by j



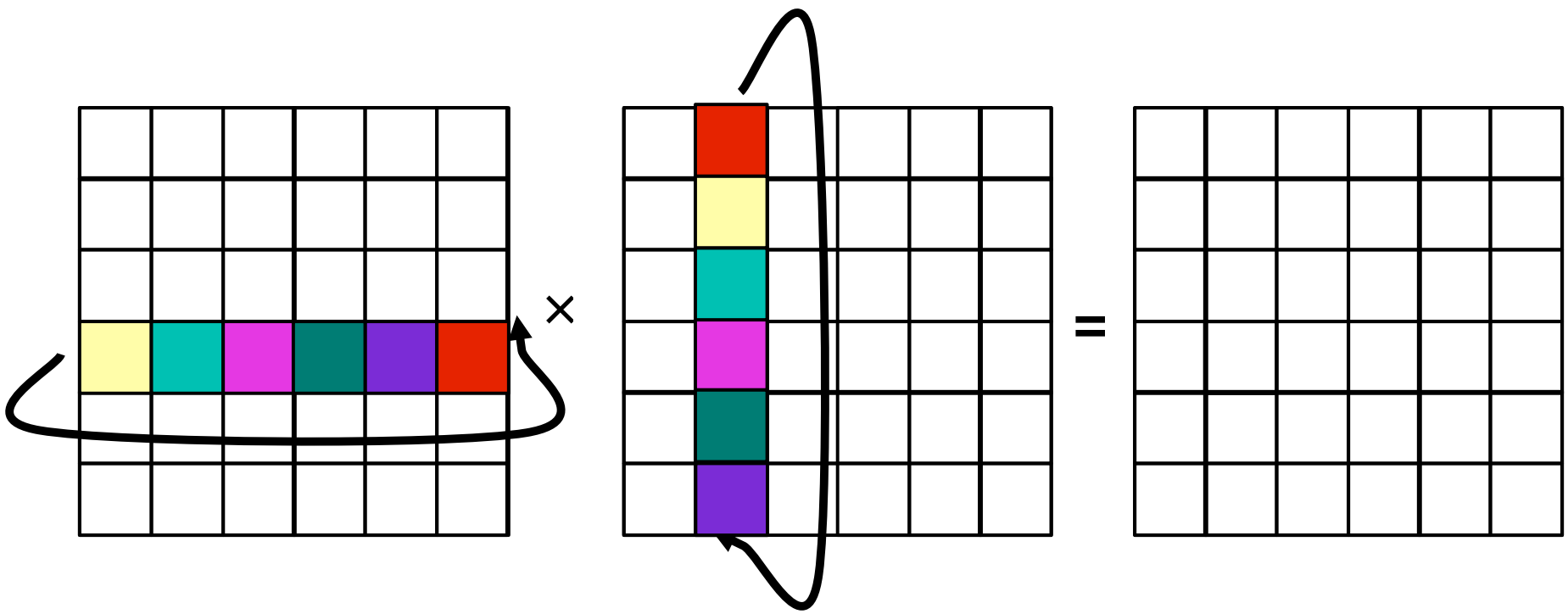
Alignment step 1

Cannon's Matrix Multiplication

Step 1: Perform Alignment

shift A_{ij} left by I

shift B_{ij} up by j



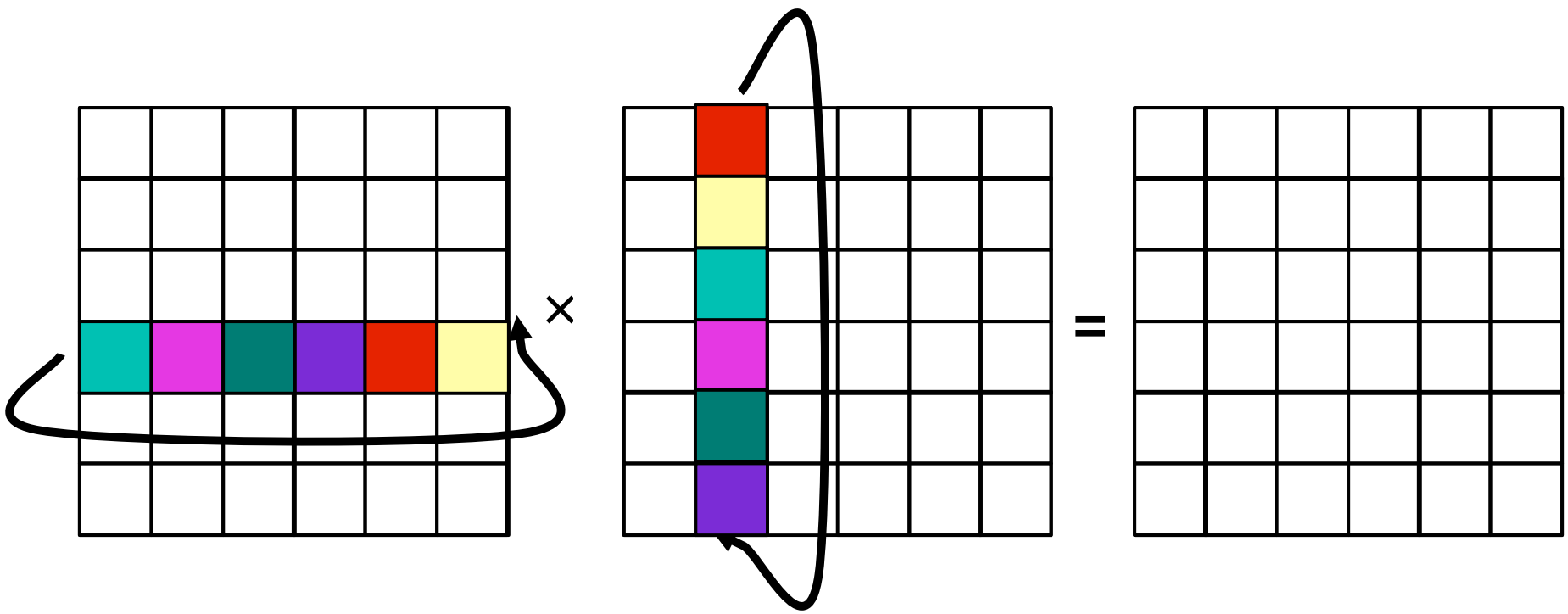
Alignment step 2

Cannon's Matrix Multiplication

Step 1: Align the tiles for the systolic computation

shift A_{ij} left by I

shift B_{ij} up by j



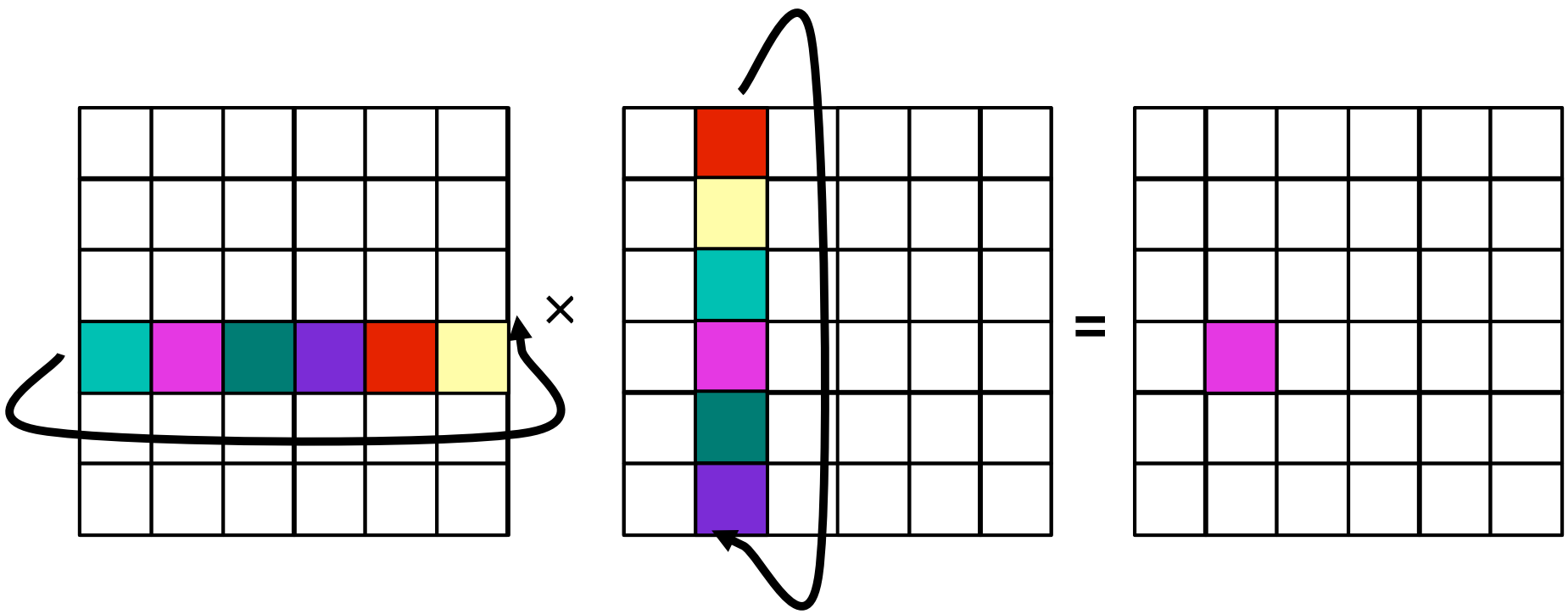
Alignment step 3

Cannon's Matrix Multiplication

Step 1: Perform Multiplication; then

shift A_{ij} left by 1

shift B_{ij} up by 1



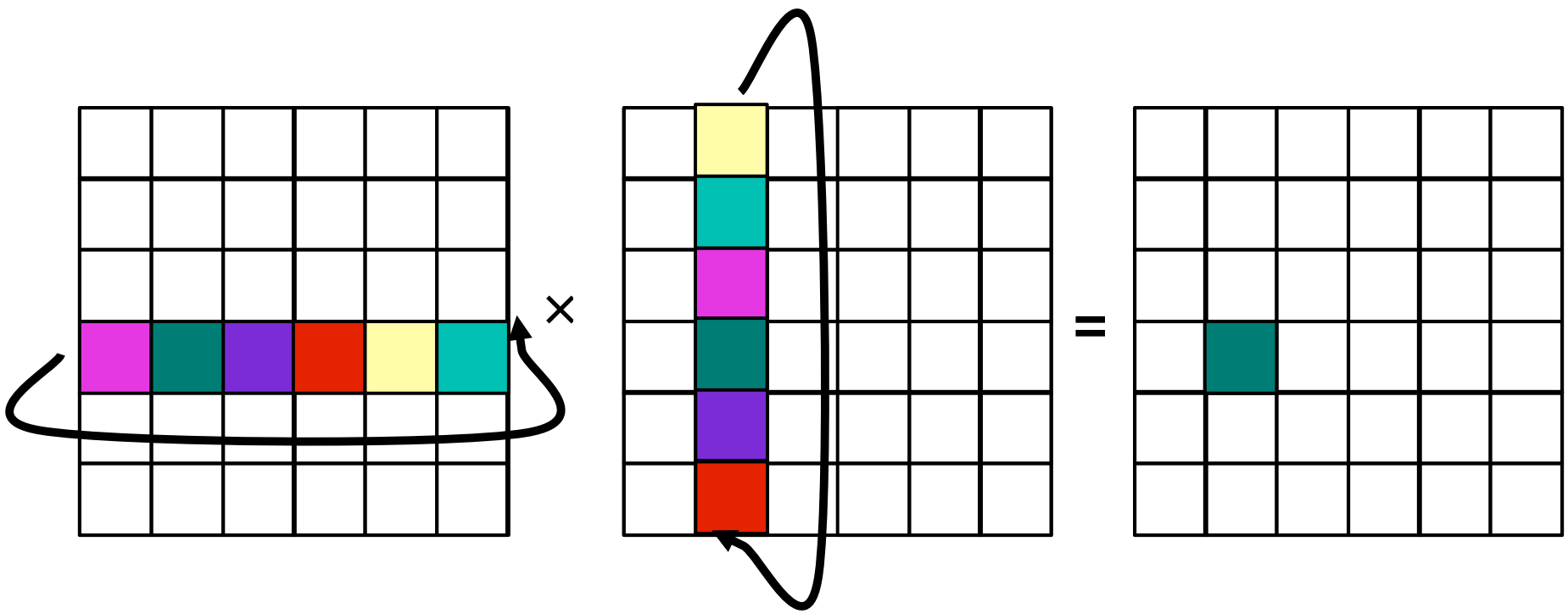
Multiplication step 1

Cannon's Matrix Multiplication

Step 1: Perform Multiplication; then

shift A_{ij} left by 1

shift B_{ij} up by 1



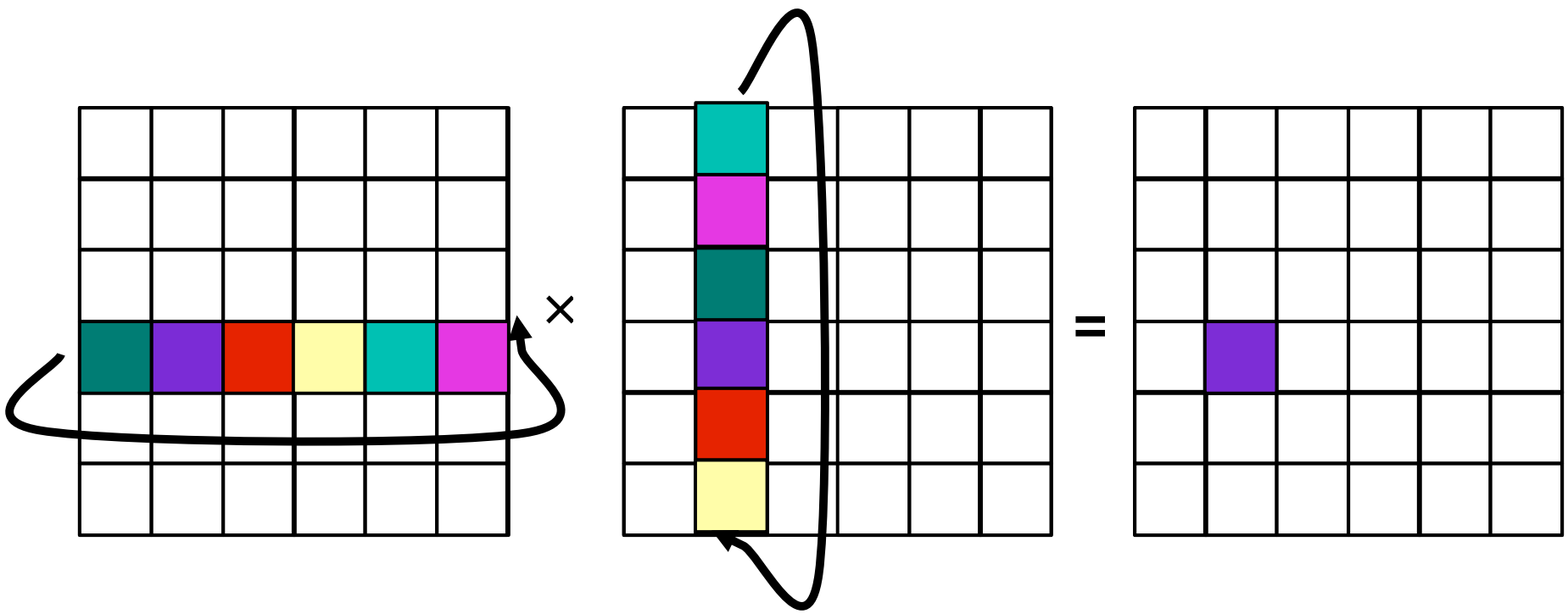
Multiplication step 2

Cannon's Matrix Multiplication

Step 1: Perform Multiplication; then

shift A_{ij} left by 1

shift B_{ij} up by 1



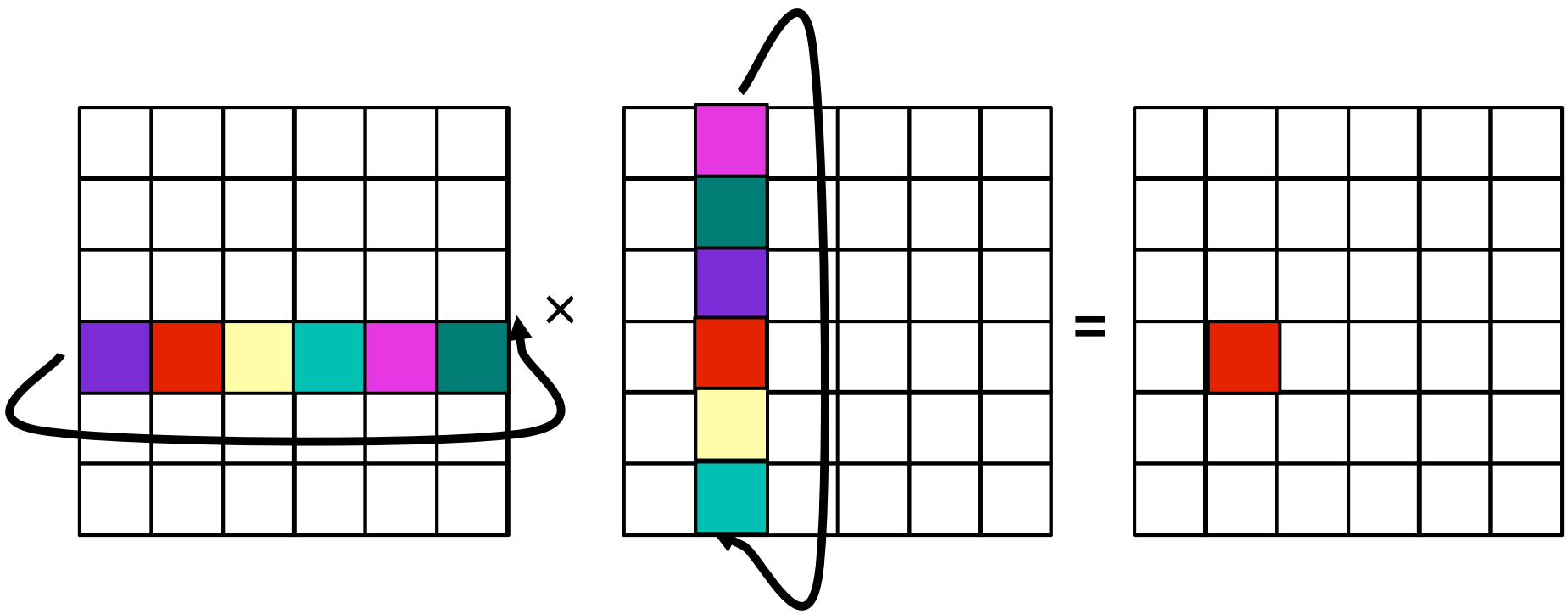
Multiplication step 3

Cannon's Matrix Multiplication

Step 1: Perform Multiplication; then

shift A_{ij} left by 1

shift B_{ij} up by 1



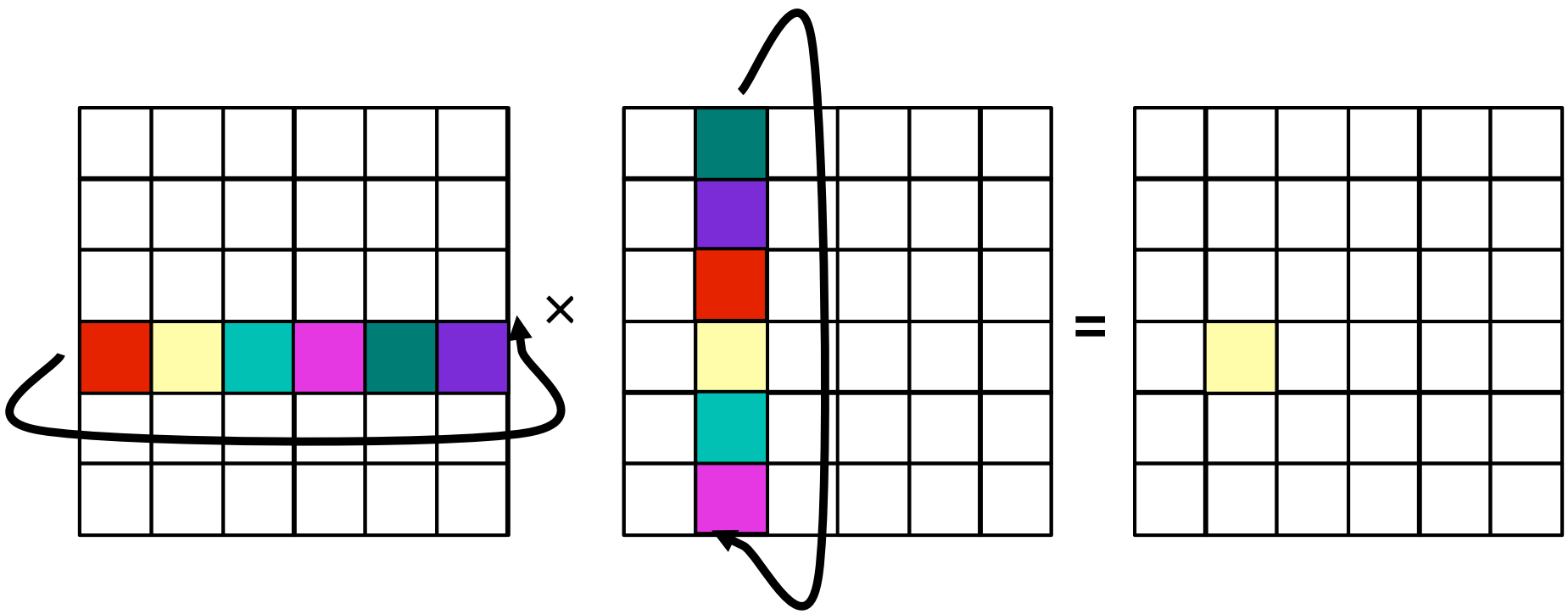
Multiplication step 4

Cannon's Matrix Multiplication

Step 1: Perform Multiplication; then

shift A_{ij} left by 1

shift B_{ij} up by 1



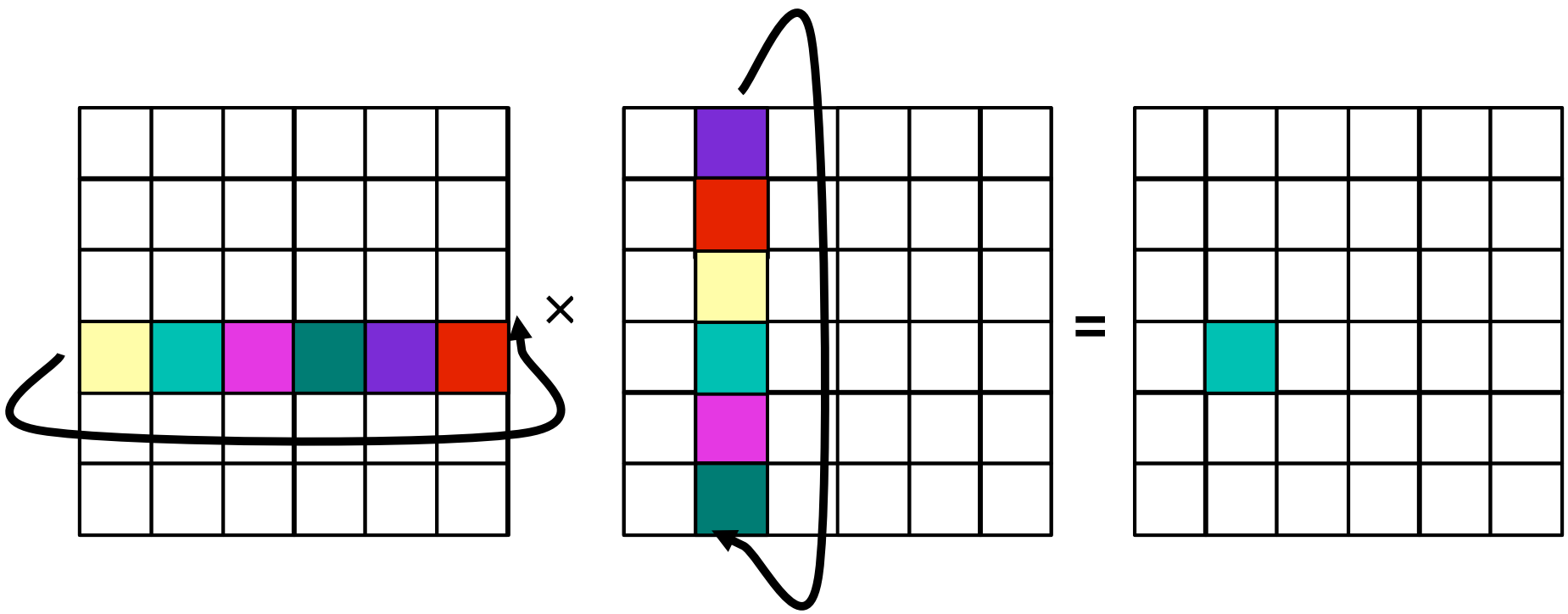
Multiplication step 5

Cannon's Matrix Multiplication

Step 1: Perform Multiplication; then

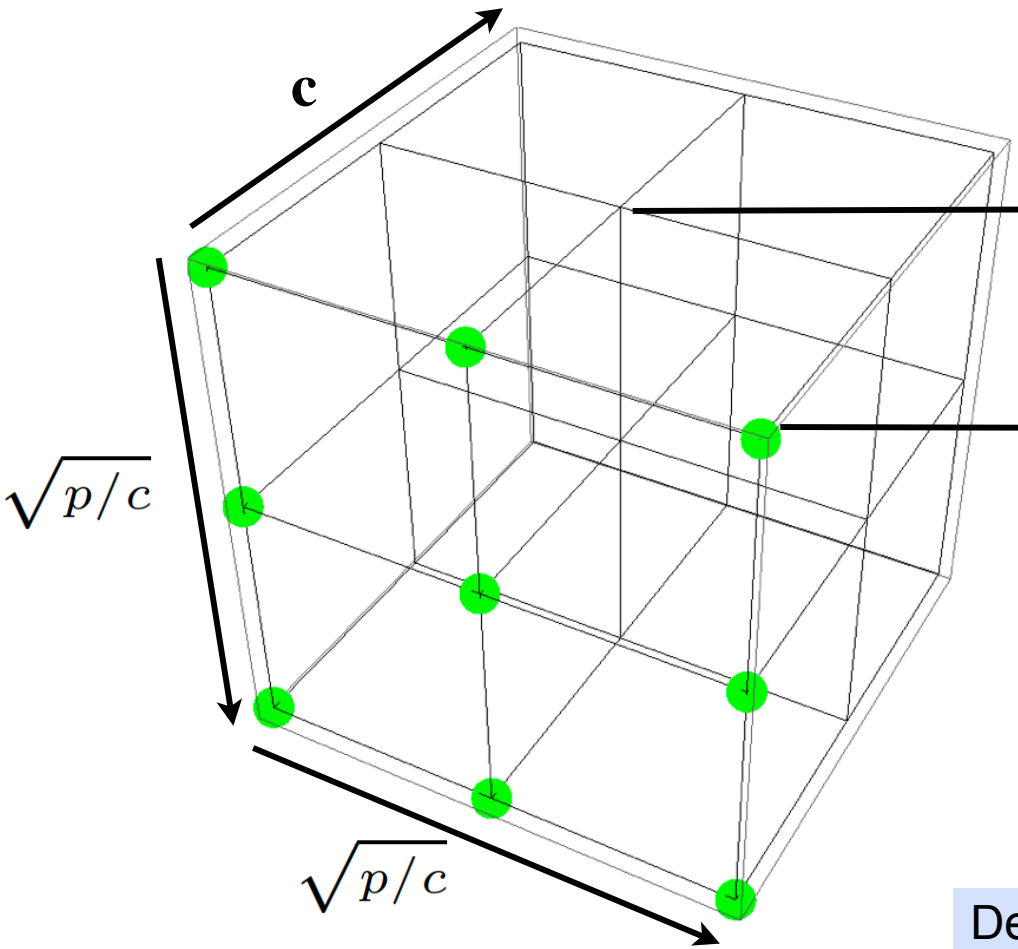
shift A_{ij} left by 1

shift B_{ij} up by 1



Multiplication step 6

2.5D Matrix Multiplication



Each point on the cube is a processor

Each green circle represents tiles of the A and B matrix of size:

$$\frac{n}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}}$$

Demmel and Solomonik, Europar 2011, Distinguished Paper.

2.5D Matrix Multiplication Algorithm

Algorithm 2: $[C]$ = 2.5D-matrix-multiply(A,B,n,p,c)

Input: square n-by-n matrices A, B distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$-by-$\frac{n}{\sqrt{p/c}}$ blocks A_{ij} and B_{ij} for each i, j

Output: square n-by-n matrix $C = A \cdot B$ distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$-by-$\frac{n}{\sqrt{p/c}}$ block C_{ij} for each i, j

```
/* do in parallel with all processors                                  */
```

forall $i, j \in \{0, 1, ..., \sqrt{p/c} - 1\}$, $k \in \{0, 1, ..., c - 1\}$ **do**

- P_{ij0} broadcasts A_{ij} and B_{ij} to all P_{ijk} — `/* replicate input matrices */`
- $s :=$ mod $(j - i + k\sqrt{p/c^3}, \sqrt{p/c})$ — `/* initial circular shift on A */`
- P_{ijk} sends A_{ij} to A_{local} on P_{isk}
- $s' :=$ mod $(i - j + k\sqrt{p/c^3}, \sqrt{p/c})$ — `/* initial circular shift on B */`
- P_{ijk} sends B_{ij} to B_{local} on $P_{s'jk}$
- $C_{ijk} := A_{\text{local}} \cdot B_{\text{local}}$
- $s :=$ mod $(j + 1, \sqrt{p/c})$
- $s' :=$ mod $(i + 1, \sqrt{p/c})$
- **for** $t = 1$ **to** $\sqrt{p/c^3} - 1$ **do**
  - P_{ijk} sends A_{local} to P_{isk} — `/* rightwards circular shift on A */`
  - P_{ijk} sends B_{local} to $P_{s'jk}$ — `/* downwards circular shift on B */`
  - $C_{ijk} := C_{ijk} + A_{\text{local}} \cdot B_{\text{local}}$
- **end**
- P_{ijk} contributes C_{ijk} to a sum-reduction to P_{ij0}

end

2.5D Matrix Multiplication Algorithm

Algorithm 2: $[C] = \text{2.5D-matrix-multiply}(A,B,n,p,c)$

Input: square n-by-n matrices A, B distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$-by-$\frac{n}{\sqrt{p/c}}$ blocks A_{ij} and B_{ij} for each i, j

Output: square n-by-n matrix $C = A \cdot B$ distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$-by-$\frac{n}{\sqrt{p/c}}$ block C_{ij} for each i, j

```
/* do in parallel with all processors                                    */
```

forall $i, j \in \{0, 1, ..., \sqrt{p/c} - 1\}$, $k \in \{0, 1, ..., c - 1\}$ **do**

- P_{ij0} broadcasts A_{ij} and B_{ij} to all P_{ijk}
- $s := \mod (j - i + k\sqrt{p/c^3}, \sqrt{p/c})$
- P_{ijk} sends A_{ij} to A_{local} on P_{isk}
- $s' := \mod (i - j + k\sqrt{p/c^3}, \sqrt{p/c})$
- P_{ijk} sends B_{ij} to B_{local} on $P_{s'jk}$
- $C_{ijk} := A_{\text{local}} \cdot B_{\text{local}}$
- $s := \mod (j + 1, \sqrt{p/c})$
- $s' := \mod (i + 1, \sqrt{p/c})$
- **for** $t = 1$ **to** $\sqrt{p/c^3} - 1$ **do**
  - P_{ijk} sends A_{local} to P_{isk}
  - P_{ijk} sends B_{local} to $P_{s'jk}$
  - $C_{ijk} := C_{ijk} + A_{\text{local}} \cdot B_{\text{local}}$
- **end**
- P_{ijk} contributes C_{ijk} to a sum-reduction to P_{ij0}

end

broadcast front plane's blocks to the back planes of the cube

analogous to Cannon's alignment

analogous to Cannon's multiply and shift step

Sketch Communication-avoiding MM in HPF

Global view/SPMD programming style

```
!HPF$ processors p(p1,p1,c)
!HPF$ template t(p1,p1,c)
!HPF$ align x(*,*,*,:,:,:) with t(:,:,:)
!HPF$ distribute t(block,block,block) onto p
integer x(n,n,3,p1,p1,c)
```

these directives apply to all arrays

```
subroutine bcast(x,n,p1,c)
   integer n, p1, c, k
   do k = 2, c   ! broadcast
     x(:, :, 1, :, :, k) = x(:, :, 1, :, :, 1)
   enddo
   end
```

from front plane to the rest

```
subroutine multiply(r,a,b,n,p1,c,cur)
   integer i,j,k
   do j = 1, n
     do k = 1, n
       do i = 1, n
         r(i, j, cur, :, :, :) = r(i, j, cur, :, :, :)
             +  a(i, k, cur, :, :,:) * b(k, j, cur, :, :,:)
       enddo
     enddo
   enddo
   end
```

local computation on each processor

```
subroutine rowshift(a,n,p1,c,now)
integer n, p1, c, d, src, dest, next
next = mod(now,3) + 1
do dest = 1, p1
  src = mod(dest, p1) + 1
  a(:, :, next, dest, :, :) = a(:, :, now, src, :, :)
enddo
end
```

can be used for systolic row shift; similar for row alignment

```
subroutine reduce(x, n, p1, c)
integer n, p1, c
integer x(n,n,3,p1,p1,c)
integer k
do k = 2, c
  x(:, :, 1, :, :, 1) = x(:, :, 1, :, :, 1) + x(:, :, 1, :, :, k)
enddo
end
```

reduce to front plane from rest

Summary: What We Need For HPC Languages

- Careful design of language features to support separation of data parallel aspects from algorithm
- Explicit high-level control of communication where practical
- Support for user-defined distributions
- Attention to important programming idioms
- Sustained investment in compiler technology
  - —managing iteration spaces, data movement, synchronization, latency tolerance, locality
- Interoperability
- Programming language ecosystem: tools
- High quality open source implementation
- Plan for longevity
- If not, we're doomed to fragmented programming!