From HPF to Coarray Fortran 2.0

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

- be efficient

- have a promise of future availability and longevity

- be supported by tools

- provide a migration path for users

significantly simpler than MPI!
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
“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.”

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

Compilation

HPF 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
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))
  END DO
END DO

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

- Data
- Iterations
- Processors

3 types of Mappings

- Layout: data ↔ processors
- Reference: iterations ↔ data
- 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

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

data / loop partitioning

P(0,0)  P(1,0)  P(2,0)
P(0,1)  P(1,1)  P(2,1)
P(0,2)  P(1,2)  P(2,2)
real A(100)
distribute A(BLOCK) on P(4)
do  i = 1, N
    ... = A(i-1) + A(i-2) + ...  ! ON_HOME A(i-1)
endo

symbolic N

Layout := \{ [pid] -> [i] : 25 *pid + 1 ≤ i ≤ 25 *pid + 25 \}
Loop := \{ [i] : 1 ≤ i ≤ N \}
CPSubscript := \{ [i] \rightarrow [i-1] \}
RefSubscript := \{ [i] \rightarrow [i-2] \}

CompPart := (Layout \circ CPSubscript^{-1}) \cap Loop
DataAccessed = CompPart \circ RefSubscript
NonLocal Data Accessed = DataAccessed - Layout
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 transformedIter = (iterSet.is_null() ?
            Relation::False(iterSet.n_set()) :
            Composition(copy(transform), copy(iterSet)));

        all = Union(copy(transformedIter), all);
        intersection = Intersection(transformedIter, 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

\[
\text{SHADOW } a(2, 2) \\
\text{ON_HOME } a(i-2, j) \cup \text{ON_HOME } a(i+2, j) \cup \\
\text{ON_HOME } a(i, j-2) \cup \text{ON_HOME } a(i, j+1)
\]
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
                      !HPF$ END ON
                      end do
                  end do
              end do
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

Processor 0
Processor 1
Processor 2
Processor 3
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
An array of $k > d$ dimensions can be partitioned into $p^{d/(d-1)}$ tiles (diagonal multipartitioning) ($p$ is the number of 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
Given an $n$-dimensional data domain and $p$ processors, select

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

• Partitioning constraints
  — # tiles in each $\lambda-1$ 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

Basic Tile Shape

Modular Shift

Integral # of shapes

Integral # of shapes
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

- 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

Efficiency NAS SP class 'C'

3027 lines
+69 HPF directives

3D multipartitioning
communication coalescing
partially-replicated computation
memory hierarchy optimization
NAS BT: Comparing 3 parallelizations

Parallel Efficiency: Speedup/
(Number of processors)

Number of processors

NAS BT Class B

hand-coded multipartitioning
dHPF multipartitioning
PGI transpose
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)

<table>
<thead>
<tr>
<th># proc.</th>
<th>relative speedup</th>
<th>GFLOPS</th>
<th>% FP peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.0</td>
<td>47.3</td>
<td>18.5</td>
</tr>
<tr>
<td>256</td>
<td>1.88</td>
<td>89.1</td>
<td>17.4</td>
</tr>
<tr>
<td>512</td>
<td>3.72</td>
<td>175.9</td>
<td>17.2</td>
</tr>
<tr>
<td>1024</td>
<td>7.45</td>
<td>352.0</td>
<td>17.2</td>
</tr>
</tbody>
</table>

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
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
        m = ishft(1, l)
        m2 = ishft(1, l - 1)
        do k = 0, n - 1, m
            do j = k, k + m2 - 1
                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

ripen 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
  — 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
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
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
integer a(10,20)[*]

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

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

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

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

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

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

two-sided bulk synchronous send
GTS Particle Shifter in CAF

one-sided asynchronous push

![Diagram of GTS Particle Shifter in CAF]

```c
!(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)

52% speedup
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
    - $r \in \{0..\text{team}\_size(t) - 1\}$
  — domain for allocating coarrays
  — substrate for collective communication

• Teams need not be disjoint
  — an image may be in multiple teams
• Predefined teams
  —\texttt{team\_world}
  —\texttt{team\_default}
    – used for any coarray operation that lacks an explicit team specification

• Operations on teams
  —\texttt{team\_rank(team)}
    – returns the relative rank of the current image within a team
  —\texttt{team\_size(team)}
    – returns the number of images of a given team
  —\texttt{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 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\textsuperscript{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

- **Hide** latency for accessing remote data by overlapping it with computation
- **Avoid** 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:
  — \texttt{team\_barrier\_async([event] [, team])}

• Communication:
  — \texttt{team\_broadcast\_async(var, root [, event] [, team])}
  — \texttt{team\_gather\_async(var\_src, var\_dest, root [, event] [, team])}
  — \texttt{team\_allgather\_async(var\_src, var\_dest [, event] [, team])}
  — \texttt{team\_reduce\_async(var\_src, var\_dest, root, operator [, event] [, team])}
  — \texttt{team\_allreduce\_async(var\_src, var\_dest, operator [, event] [, team])}
  — \texttt{team\_scatter\_async(var\_src, var\_dest, root [, event] [, team])}
  — \texttt{team\_alltoall\_async(var\_src, var\_dest [, event] [, team])}
  — \texttt{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

```plaintext
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

```fortran
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
! post a receive
do n=1,in_bndy%nmsg_ew_rcv
  bufsize = ny_block*nghost*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*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
end do

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

! 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
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
```fortran
! 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
```
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

```fortran
... 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: \( c = (/ c(\text{bitreverse}(i), i = 0, n-1 /) \)
    - 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 \( \equiv \text{rank mod } P \)
      - block distribution \( \rightarrow \) cyclic distribution
    - compute the remaining \( \log P \) stages of the FFT locally
    - transpose the data back to its original order
      - cyclic distribution \( \rightarrow \) 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
Productivity = Performance / SLOC

Performance (Cray XT4)

<table>
<thead>
<tr>
<th># of cores</th>
<th>STREAM Triad † (TByte/s)</th>
<th>RandomAccess *(GUP/s)</th>
<th>Global HPL † (TFlop/s)</th>
<th>Global FFT † (GFlop/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.14</td>
<td>0.08</td>
<td>0.36</td>
<td>6.69</td>
</tr>
<tr>
<td>256</td>
<td>0.54</td>
<td>0.24</td>
<td>1.36</td>
<td>22.82</td>
</tr>
<tr>
<td>1024</td>
<td>2.18</td>
<td>0.69</td>
<td>4.99</td>
<td>67.80</td>
</tr>
<tr>
<td>4096</td>
<td>8.73</td>
<td>2.01</td>
<td>18.3</td>
<td>187.04</td>
</tr>
</tbody>
</table>

*Measured on Jaguar  † Measured on Franklin

Source lines of code

<table>
<thead>
<tr>
<th>HPC Challenge Benchmark</th>
<th>Source Lines of Code</th>
<th>Reference SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Randomaccess</td>
<td>409</td>
<td>787</td>
</tr>
<tr>
<td>EP STREAM Triad</td>
<td>58</td>
<td>329</td>
</tr>
<tr>
<td>Global HPL</td>
<td>786</td>
<td>8800</td>
</tr>
<tr>
<td>Global FFT</td>
<td>~390</td>
<td>1130</td>
</tr>
</tbody>
</table>

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

![Graph showing relative parallel efficiency for different scalability measures.

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>4096</th>
<th>8192</th>
<th>16384</th>
<th>32768</th>
<th>65536</th>
</tr>
</thead>
<tbody>
<tr>
<td>RandomAccess</td>
<td>1.00</td>
<td>0.91</td>
<td>0.84</td>
<td>0.78</td>
<td>0.68</td>
<td>0.63</td>
<td>0.60</td>
<td>0.44</td>
<td>0.38</td>
</tr>
<tr>
<td>&quot;FFT&quot;</td>
<td>1.00</td>
<td>0.93</td>
<td>0.90</td>
<td>0.87</td>
<td>0.80</td>
<td>0.76</td>
<td>0.72</td>
<td>0.40</td>
<td>0.40</td>
</tr>
<tr>
<td>UTS</td>
<td>0.79</td>
<td>0.79</td>
<td>0.79</td>
<td>0.78</td>
<td>0.78</td>
<td>0.78</td>
<td>0.77</td>
<td>0.74</td>
<td>0.74</td>
</tr>
<tr>
<td>Stream</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.61</td>
<td>0.61</td>
</tr>
<tr>
<td>HPL-K1</td>
<td>1.00</td>
<td>0.93</td>
<td>0.84</td>
<td>0.68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 bitonic 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

Collective Example: Barrier

Dissemination algorithm

for $k = 0$ to $\lceil \log_2 P \rceil$

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

round 0 2^0
round 1 2^1
round 2 2^2
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 conventional counterparts

- Examples
  - time skewing, e.g., overlapped tiling
  - new algorithms for linear algebra, e.g., matrix multiply
Vision for Overlapped Tiling in dHPF

!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)
endo
!HPF$ local end
!HPF$ on home end
enddo

Assumptions: 2D BLOCK distribution for A

SHADOW (4:4,4:4)

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 $\sqrt{p} \times \sqrt{p}$ processor grid
Cannon’s Matrix Multiplication

Stage 1: Perform Alignment

shift $A_{ij}$ left by $l$

shift $B_{ij}$ up by $j$
Cannon’s Matrix Multiplication

Step 1: Perform Alignment

shift $A_{ij}$ left by $l$

shift $B_{ij}$ up by $j$

Alignment step 1
Cannon’s Matrix Multiplication

Step 1: Perform Alignment

- shift $A_{ij}$ left by $l$
- 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 $l$
- 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.
Further, the cost of sacrificing flops for latency is large. Namely, if is best to pick all $P$ by counting the complexity along this path. The latency cost is given this dependency path (shown in Figure 2), we can lower bound the complexity of the algorithm. Factorizations of these blocks are on the critical path and must be done in strict sequence.

We now lower bound the communication cost for any algorithm that follows the above restrictions. Any given a parallel LU factorization algorithm, we assume the algorithm must uphold the following properties.

- The latency lower bound is actually much higher, namely.
- We argue that for Gaussian-elimination style LU algorithms that achieve the bandwidth lower bound, the 4 2.5D LU communication lower bound.

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

**Input:** square $n$-by-$n$ matrices $A$, $B$ distributed so that $P_{ij0}$ owns $n/\sqrt{p/c}$-by-$n/\sqrt{p/c}$ 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 $n/\sqrt{p/c}$-by-$n/\sqrt{p/c}$ block $C_{ij}$ for each $i,j$

/* do in parallel with all processors */

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

  $P_{ij0}$ broadcasts $A_{ij}$ and $B_{ij}$ to all $P_{ijk}$

  $s := \text{mod} (j - i + k\sqrt{p/c^2}, \sqrt{p/c})$

  $P_{ijk}$ sends $A_{ij}$ to $A_{\text{local}}$ on $P_{isk}$

  $s' := \text{mod} (i - j + k\sqrt{p/c^2}, \sqrt{p/c})$

  $P_{ijk}$ sends $B_{ij}$ to $B_{\text{local}}$ on $P_{s'jk}$

  $C_{ijk} := A_{\text{local}} \cdot B_{\text{local}}$

  $s := \text{mod} (j + 1, \sqrt{p/c})$

  $s' := \text{mod} (i + 1, \sqrt{p/c})$

  for $t = 1$ to $\sqrt{p/c^2} - 1$ do

    $P_{ijk}$ sends $A_{\text{local}}$ to $P_{isk}$

    $P_{ijk}$ sends $B_{\text{local}}$ 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
```

/* replicate input matrices */

/* initial circular shift on $A$ */

/* initial circular shift on $B$ */

/* rightwards circular shift on $A$ */

/* downwards circular shift on $B$ */
2.5D Matrix Multiplication Algorithm

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

Input: square $n$-by-$n$ matrices $A$, $B$ distributed so that $P_{ij0}$ owns $\frac{n}{p/c}$-by-$\frac{n}{p/c}$ 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 $\frac{n}{p/c}$-by-$\frac{n}{p/c}$ 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 $P_{ijk}$

$s := \text{mod} \ (j - i + k\sqrt{p/c^3},\sqrt{p/c})$

$P_{ijk}$ sends $A_{ij}$ to $A_{local}$ on $P_{isk}$

$s' := \text{mod} \ (i - j + k\sqrt{p/c^3},\sqrt{p/c})$

$P_{ijk}$ sends $B_{ij}$ to $B_{local}$ on $P_{s'jk}$

$C_{ijk} := A_{local} \cdot B_{local}$

$s := \text{mod} \ (j + 1,\sqrt{p/c})$

$s' := \text{mod} \ (i + 1,\sqrt{p/c})$

for $t = 1$ to $\sqrt{p/c^3} - 1$ do

$P_{ijk}$ sends $A_{local}$ to $P_{isk}$

$P_{ijk}$ sends $B_{local}$ to $P_{s'jk}$

$C_{ijk} := C_{ijk} + A_{local} \cdot B_{local}$

end

$P_{ijk}$ contributes $C_{ijk}$ to a sum-reduction to $P_{ij0}$

end
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)

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

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

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

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

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

123
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!