Co-Array Fortran and High Performance Fortran

John Mellor-Crummey

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

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

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

Issues and Ongoing Work

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

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

Fortran program + data partitioning
Partition computation
Insert communication
Manage storage

Same answers as sequential program

Compilation

Parallel Machine
Example HPF Program

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

DO i = 2, n - 1
  DO j = 2, n - 1
    A(i,j) = 0.25 * (B(i-1,j) + B(i+1,j) + B(i,j-1) + B(i,j+1))
  END DO
END DO

Processors P(0,0) P(2,2)

Data for A, B
(BLOCK,BLOCK) distribution
Compiling HPF

• Partition data
  —follow user directives

• Map computation to processors
  —co-locate computation with data

• Analyze communication requirements
  —identify references that access off-processor data

• Partition computation by reducing loop bounds
  —schedule each processor to compute on its own data

• Insert communication
  —exchange values as needed by the computation

• Manage storage for non-local data
FormalCompilationFramework(RicedHPF)

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 (universal & existential quantifiers + linear inequalities with constant coefficients + logical operators) for constraints

- Analysis: Use set equations to compute set(s) of interest
  - iterations allocated to a processor
  - communication sets

- Code generation: Synthesize loops from set(s), e.g.
  - parallel (SPMD) loop nests
  - message packing and unpacking

[Adve & Mellor-Crummey, PLDI98]
processors $P(3,3)$
distribute $A(\text{block, block})$ onto $P$
distribute $B(\text{block, block})$ onto $P$

DO $i = 2, n - 1$
DO $j = 2, n - 1$

\[
A(i, j) = 0.25 \times (B(i-1, j) + B(i+1, j) + B(i, j-1) + B(i, j+1))
\]

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

Symbolic 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)
endo

symbolic N

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

Loop := \{ [i] : 1 \leq i \leq N \}

CPSubscript := \{ [i] \mapsto [i-1] \}

RefSubscript := \{ [i] \mapsto [i-2] \}

CompPart := (Layout \circ CPSubscript^{\text{-1}}) \cap Loop

DataAccessed = CompPart \circ RefSubscript

NonLocal Data Accessed = DataAccessed - Layout
Performance Using the Rice dHPF Compiler

Efficiency NAS SP class 'C'

Number of Processors

Parallel Efficiency

3027 lines
+69 HPF directives

MPI
dHPF
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 PSC’s Lemieux

<table>
<thead>
<tr>
<th># procs</th>
<th>relative speedup</th>
<th>GFLOPS</th>
<th>% peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.0</td>
<td>46.4</td>
<td>18.1</td>
</tr>
<tr>
<td>256</td>
<td>1.94</td>
<td>89.9</td>
<td>17.6</td>
</tr>
<tr>
<td>512</td>
<td>3.78</td>
<td>175.5</td>
<td>17.4</td>
</tr>
<tr>
<td>1024</td>
<td>7.58</td>
<td>352.0</td>
<td>17.3</td>
</tr>
</tbody>
</table>
subroutine fft(c, n)
    implicit complex(c)
    dimension c(0:n-1), irev(0:n-1)
!HPF$ processors p(number_of_processors())
!HPF$ template t(0:n-1)
!HPF$ align c(i) with t(i)
!HPF$ align irev(i) with t(i)
!HPF$ distribute t(block) onto p
    two_pi = 2.0d0 * acos(-1.0d0)
    levels = number_of_bits(n) - 1
    irev = (/ (bitreverse(i,levels), i= 0, n-1) /)
    forall (i=0:n-1) c(i) = c(irev(i))
    do l = 1, levels ! --- for each level in the FFT
        m = ishft(1, l)
        m2 = ishft(1, l - 1)
        do k = 0, n - 1, m ! --- for each butterfly in a level
            do j = k, k + m2 - 1 ! --- for each point in a half bfly
                ce = exp(cmplx(0.0,(j - k) * -two_pi/real(m)))
                cr = ce * c(j + m2)
                cl = c(j)
                c(j) = cl + cr
                c(j + m2) = cl - cr
            end do
        end do
    enddo
end subroutine fft
1D FFT: Partitioning Work

```fortran
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
      end do
end subroutine fft
```

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

partitioning the j loop is driven by the data accessed in its iterations
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
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(complex(0.0,(j - k) * -two_pi/real(m)))
        cr = ce * c(j + m2)
        cl = c(j)
        c(j) = cl + cr
        c(j + m2) = cl - cr
    end do
end do
enddo
end subroutine fft

ripe for space-time tradeoff as well as strength reduction
Current Code Generation Challenges

• Symbolically strided iteration spaces
• Efficient management of non-local storage
• Overlapping communication and computation
• Top-quality code for inner loops
A Tale of Two Programming Models

Issues and Ongoing Work

- High Performance Fortran
- Co-array Fortran
Co-Array Fortran (CAF)

• Explicitly-parallel extension of Fortran 95
  — defined by Numrich & Reid

• Global address space SPMD parallel programming model
  — one-sided communication

• Simple, two-level memory model for locality management
  — local vs. remote memory

• Programmer has control over performance critical decisions
  — data partitioning
  — computation
  — communication
  — synchronization

• Suitable for mapping to a range of parallel architectures
  — shared memory, clusters, hybrid
CAF Programming Model Features

- **SPMD process images**
  - fixed number of images during execution: `num_images()`
  - images operate asynchronously: `this_image()`

- **Both private and shared data**
  - `real x(20, 20)` a private 20x20 array in each image
  - `real y(20, 20) [*]` a shared 20x20 array in each image

- **Simple one-sided shared-memory communication**
  - `x(:,j:j+2) = y(:,p:p+2) [r]` copy columns from p:p+2 into local columns

- **Synchronization intrinsic functions**
  - `sync_all` – a barrier and a memory fence
  - `sync_mem` – a memory fence
  - `notify_team(team), wait_team(team), …` – point-to-point

- **Asymmetric dynamic allocation of shared data**
- **Weak memory consistency**
A CAF Finite Element Example (Numrich)

subroutine assemble(start, prin, ghost, neib, x)
    integer :: start(:), prin(:), ghost(:), neib(:), k1, k2, p
    real :: x(:) [*]
    call sync_all(neib)
    do p = 1, size(neib) ! Add contributions from ghost regions
        k1 = start(p); k2 = start(p+1)-1
        x(prin(k1:k2)) = x(prin(k1:k2)) + x(ghost(k1:k2)) [neib(p)]
    enddo
    call sync_all(neib)
    do p = 1, size(neib) ! Update the ghosts
        k1 = start(p); k2 = start(p+1)-1
        x(ghost(k1:k2)) [neib(p)] = x(prin(k1:k2))
    enddo
    call sync_all
end subroutine assemble
Enhancing CAF

- Expressiveness
- Programmability
- Latency hiding
- Compiler-based optimization
- Performance
Emerging Approaches

• Communication topologies
• Multi-version variables
• Distributed multithreading
CAF Co-shapes

• integer $a(10,10)[2,*]$

• Number of images is 6
Shortcomings of Multi-dimensional Co-shapes

• Limited expressiveness
  — only Cartesian topology without periodic boundaries for all images
    – programmers manually implement other topologies
  — lack processor subsets, e.g. for coupled applications

• Programming complexity
  — “global” coordinates; explicit index management by programmers
  — “holes” require special handling

• Performance
  — no support for collective communication on subgroups
  — difficult to analyze and optimize communication patterns

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

• Goals
  — increase expressiveness
    – beyond Cartesian grids without periodic boundary conditions
  — reduce programming complexity
    – abstract away details of managing communication partners
  — organize images into groups
    – processor subsets and collective operations
  — facilitate compiler analysis and optimization

• Approach: co-space abstraction
  — types: Group, Cartesian, Graph
  — default co-space: CAF_WORLD

Inspired by MPI communicators and process topologies
Analyzing Communication on Co-spaces

• Goals
  — understand data movement and synchronization
  — make communication and synchronization efficient

• Analysis questions
  — which images perform communication?
  — which image is the target of each communication?
Exploiting Co-space Structure

- **Co-space textual barriers**
  - inspired by Titanium’s global textual barriers

- **Co-space single-valued expressions**
  - inspired by Titanium’s **single**
  - analyze co-space group control flow

- **Co-space topology structure**
  - identify the source and target of each GET/PUT

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

SSR optimization: convert barriers into point-to-point synchronization
NAS MG class A \((256^3)\) on Itanium2+Myrinet

Higher is better

SSR boosts performance & scalability

*Higher is better*
Wavefront Communication

• Streaming values from producers to consumers
• Barrier synchronization is unnatural
• Overlap communication with computation
  — multiple communication buffers
  — pipelined synchronization
  — non-blocking communication
Two-sided vs. One-sided Wavefronts

2-sided msgs

producer

produced value

MPI_Send

consumer

consume value

MPI_Recv

producer

produced value

wait

consumer

wait done

PUT

notify

value avail

consumer

buffer is free

notify

wait

wait done

consume value

1-sided with 1 buffer
Hiding Latency with Multiple Buffers

producer | consumer
---|---
produced value
*MPI_Send*

consumer

produced value
*MPI_Recv*

consume value

producer | consumer
---|---
buffer X is free

consumer

consume buffer Y

2-sided msgs

time

1-sided with N buffers
Assessing CAF Support for Wavefronts

• MPI is simpler!
  — message = data movement and synchronization
  — no explicit buffer management
  — eager protocol can achieve latency tolerance

• CAF with one buffer
  — lacks asynchrony and latency tolerance

• CAF with multiple buffers
  — performance better than MPI
    – sender delivers directly to destination memory
  — much harder to program in classic CAF
    – explicit buffer management
    – pipelined non-blocking communication for producer latency tolerance
    – pipelined point-to-point synchronization for consumer latency tolerance
Multi-version variables for CAF

CAF support for wavefront and producer/consumer patterns

- type(T), multiversion :: a(N, M)[*]
  - SAVE, allocatable, module, subroutine parameter

- Producer
  - commit(a[dest], b, [tag], [live=true])

- Consumer
  - retrieve(a, [b], [tag], [image])
  - check(a, [tag])
Multi-version Variable Benefits

• Well-suited for a variety of communication styles
  — wave-front, e.g. Sweep3D
  — line-sweep, e.g. NAS BT, SP
  — loosely synchronous

• Better programmability
  — transparent buffer mgmt, synchronization, non-blocking communication

• High performance
  — hides data movement and synchronization latency
    – exploits non-blocking PUTs
    – removes buffer synchronization from the critical path
  — avoids extra data copies
    – the destination address of communication is known
Itanium2 + Myrinet 2000, Sweep3D Size 300³

Higher is better

Multi-version variables boost performance & scalability
Looking forward with CAF

• Our experience with CAF has highlighted difficulties
  — lack of process topologies for collectives and coupled apps
  — exposed latency
  — difficulty of coding high performance wavefront calculations

• Additional language and library support can help
  — co-spaces: simplify programming & aid compiler optimization
  — multi-version variables: simplify programming and deliver performance
  — distributed multithreading: avoid exposed latency