Introduction to Co-Array Fortran

Robert W. Numrich

Minnesota Supercomputing Institute
University of Minnesota, Minneapolis
rwn@msi.umn.edu
What is Co-Array Fortran?

- Co-Array Fortran is one of three simple language extensions to support explicit parallel programming.
  - Co-Array Fortran (CAF) Minnesota
  - Unified Parallel C (UPC) GWU-Berkeley-NSA-Michigan Tech
  - Titanium (extension to Java) Berkeley
- Recent additions that are not simple extensions
  - Chapel from Cray
  - X10 from IBM
Programming Models

• Libraries
  – MPI, Shmem, ScaLAPACK, Trilinos, …

• Language extensions
  – CAF, UPC, Titanium, Intel Ct, Microsoft C#…

• Language directives
  – HPF, OpenMP, …

• New languages
  – X10, Chapel, …
Arguments about Programming Models

• Libraries are more portable than language extensions but may not be very flexible.
• Language extensions allow compilers to optimize for specific hardware capabilities but they may not do it well.
• Language directives work well for loop-level parallelism and for simple data decomposition but not for more complicated things.
• New languages allow for higher levels of abstraction, but they are far removed from hardware and people won’t adopt them quickly.
• The significant differences between models usually comes down to three questions:
  – Does the model use a global view of data or a local view of data?
  – Does the model assume a single thread of control or multiple threads of control?
  – How is the affinity between data and work defined?
The Guiding Principle for the Co-Array Model

• What is the smallest change required to make Fortran an effective parallel language?
• How can this change be expressed so that it is intuitive and natural for Fortran programmers?
• How can it be expressed so that existing compiler technology can implement it easily and efficiently?
The Co-Array Programming Model

• Single-Program-Multiple-Data (SPMD)
  – A program is replicated a fixed number of times.
  – Each replication is called an image.
  – The run-time system assigns a physical processor to perform work on the data associated with an image.

• Images execute asynchronously except where explicit synchronization is inserted in the code.
  – All data is local
  – All computation is local
  – One-sided communication thru co-dimensions

• Programmer is responsible for
  – Explicit data decomposition
  – Explicit synchronization
Co-Array Fortran Execution Model

- The number of images is fixed and each image has its own index, retrievable at run-time:
  \[ 1 \leq \text{num\_images}() \]
  \[ 1 \leq \text{this\_image}() \leq \text{num\_images}() \]
- Each image executes the same program independently of the others.
- The programmer inserts explicit synchronization and branching as needed.
- An “object” has the same name in each image.
- Each image works on its own local data.
- An image moves remote data to local data through, and only through, explicit co-array syntax.
What is Co-Array Syntax?

- Co-Array syntax is a simple parallel extension to normal Fortran syntax.
  - It uses normal rounded brackets ( ) to point to data in local memory.
  - It uses square brackets [ ] to point to data in remote memory.
  - Syntactic and semantic rules apply separately but equally to ( ) and [ ].
Declaration of a Co-Array

real :: x(n)[*]
Co-Array Memory Model

1

\[ x(1) \]
\[ x(n) \]

p

\[ x(1) \]
\[ x(n) \]

\[ x(1)[q] \]

q

\[ x(1) \]
\[ x(n) \]

\[ x(n)[p] \]

*
Examples of Co-Array Declarations

- real :: a(n)[*]
- complex :: z[0:*]
- integer :: index(n)[*]
- real :: b(n)[p, *]
- real :: c(n,m)[0:p, -7:q, +11:*]
- real, allocatable :: w(:)[::]
- type(field), allocatable :: maxwell[::]
Communication Using CAF Syntax

\[ y(:) = x(:)[p] \]

\[ x(\text{index}(k)) = y[\text{index}(p)] \]

\[ x(:)[q] = x(:) + x(:)[p] \]

Absent co-dimension defaults to the local object.
One-to-One Execution Model

One Physical Processor
Many-to-One Execution Model

Many Physical Processors

1

x(1) \downarrow \ x(n)

x(1) \downarrow \ x(n)

x(1) \downarrow \ x(n)

x(n) \uparrow \ x(n)

p

x(1)[q]

x(n)[p]

q

x(1) \downarrow \ x(n)

x(1) \downarrow \ x(n)

x(n) \uparrow \ x(n)

*
One-to-Many Execution Model

1

\[ x(1) \rightarrow x(n) \]

p

\[ x(1) \rightarrow x(n) \]

\[ x(1)[q] \]

x(n)[p]

q

\[ x(1) \rightarrow x(n) \]

* 

\[ x(1) \rightarrow x(n) \]

One Physical Processor
Many-to-Many Execution Model

Many Physical Processors
What Do Co-Dimensions Mean?

real :: x(n)[p,q,*]

1. Replicate an real array called x of local length n, one on each image.
2. Build a map so each image knows how to find the array on any other image.
3. Organize images in a logical (not physical) three-dimensional grid.
4. The last co-dimension acts like an assumed size array: * ⇒ num_images()/(pxq)
\[ x[4,\ast] \quad \text{this\_image()} = 15 \quad \text{this\_image}(x) = (3,4) \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
$x[0:3,0:*] \quad \text{this\_image()} = 15 \quad \text{this\_image}(x) = (2,3)$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
\[ x[-5:-2,0:*] \quad \text{this}_\text{image}() = 15 \quad \text{this}_\text{image}(x) = (-3, 3) \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>-4</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>-3</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>-2</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
\[
x[0:1,0:*] \quad \text{this}\_\text{image}() = 15 \quad \text{this}\_\text{image}(x) = (0,7)
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & 1 & 3 & 5 & 7 & 9 & 11 & 13 & 15 \\
1 & 2 & 4 & 6 & 8 & 10 & 12 & 14 & 16 \\
\end{array}
\]
\( x[3,0:*] \) \hspace{1cm} \text{num\_images()} = 13

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>-</td>
</tr>
</tbody>
</table>
Procedure Interfaces

Co-dimensions are interpreted locally.

```fortran
real :: x[*]
call sub(x,p)
...

subroutine sub(x,p)
integer :: p
real :: x[p,*]
...
end subroutine
```
Example 0

program ex0
  implicit none
  real :: z[3,0:*]
  integer :: me(2)
  integer :: iAm
  iAm = this_image()
  me = this_image(z)
  z = iAm
  sync all
  write(*,"(Hello from image ',i5,' (',i5,',',i5,')',f10.3)") iAm, me,z[1,4]
  !write(*,"(Hello from image ',i5,' (',i5,',',i5,')',f10.3)") iAm, me,z[2,4]
end program ex0
Synchronization and Memory Consistency
Synchronization

sync all
   Full barrier; wait for all images before continuing.

sync images(list)
   Partial barrier with images in list(:)

sync memory
   Make local co-arrays visible.

critical
   One image at a time

lock/unlock
   Control access to a co-array variable

spin loops
   Spin on a co-array until it changes
Hidden Sync’s

• Hidden sync all after variable declarations
• Hidden sync all after allocating a co-array
• Hidden sync all before deallocating a co-array
• Hidden sync all before end program
sync images()

if (this_image() == 1) then
    sync images(*)
else
    sync images(1)
end if
Examples

• Global reductions
• Matrix multiplication
• Halo exchange
Example 1: Global sum
Global Sum

subroutine globalSum(x)
real(kind=8),dimension[0:*] :: x
real(kind=8) :: work
integer n,bit,i,mypal,dim,me, m
dim = log2_images()
if(dim .eq. 0) return
m = 2**dim
bit = 1
me = this_image(x)
do i=1,dim
    mypal=xor(me,bit)
    bit=shiftl(bit,1)
    sync all
    work = x[mypal]
    sync all
    x=x+work
end do
end subroutine globalSum
Exercise 1: Global Sum

1. Write the function log2_images().
2. Remove the power-of-two assumption.
3. Convince yourself that two sync’s are necessary.
4. Rewrite with only one sync.
5. Rewrite using sync images.
Example 2: Matrix Multiplication
Matrix Multiplication

real,dimension(n,n) :: a, b, c

do k=1,n

\[ c(i,j) = c(i,j) + a(i,k) \times b(k,j) \]

end do
Matrix Multiplication

\[
\begin{align*}
\text{myP} & \times \text{myQ} \\
\text{myP} & \times \text{myQ} \\
\text{myP} & \times \text{myQ}
\end{align*}
\]
Matrix Multiplication

real,dimension(n,n)[p,*] :: a,b,c

do k=1,n
do q=1,p
    c(i,j)[myP,myQ] = c(i,j)[myP,myQ] + a(i,k)[myP, q]*b(k,j)[q,myQ]
endo
dendo
Matrix Multiplication

real,dimension(n,n)[p,*] :: a, b, c

do k=1,n
  do q=1,p
    c(i,j) = c(i,j) + a(i,k)[myP, q]*b(k,j)[q,myQ]
  enddo
endo
Block Matrix Multiplication

Figure 4: Time as a function of the number of processors $p = q \times r$ for block matrix multiplication. The matrix size is $1000 \times 1000$ with blocks of size $1000/q \times 1000/r$. Time is expressed in dimensionless giga-clock-ticks, $\nu t \times 10^{-9}$, as measured on a CRAY-T3E with frequency $\nu = 300$MHz. The dotted line represents perfect scaling.
program matmul
  implicit none
  real, allocatable,dimension(:,,:), codimension[,:]: :: a,b,c
  integer :: i
  integer :: j
  integer :: k
  integer :: l
  integer,parameter :: n = 10
  integer :: p
  integer :: q
  integer :: iAm
  integer :: myP
  integer :: myQ
  p = num_images()
  q = int(sqrt(float(p)))
  iAm = this_image()
  if (q*q /= p) then
    if (iAm == 1) write (*,"(('num_images must be square: p=',i5)') p
    stop
  end if
  allocate(a(n,n)[q,*])
  allocate(b(n,n)[q,*])
  allocate(c(n,n)[q,*])
  myP = this_image(c,1)
  myQ = this_image(c,2)
  a = 1.0
  b = 1.0
  c = 0.0
  sync all
  do i=1,n
    do j=1,n
      do k=1,n
        do l=1,q
          c(i,j) = c(i,j) + a(i,k)[myP,l]*b(k,j)[l,myQ]
        end do
      end do
    end do
  end do
  if (any(c /= n*q)) write(*,"(('error on image: ',2i5,e20.10)') myP, myQ, c(1,1)
  write(*,"(('check sum[',i5,';',i5,']',e20.10)') myP, myQ, sum(c) - q*n**3
  deallocate(a,b,c)
end program matmul
Exercise 2: Matrix Multiplication

1) Remove the restrictions (n,n) and [q,q].
2) Change element-by-element to a block algorithm.
3) How many of these can you implement?
4) When is one better than another?

\[ C_q = A B_q \]
\[ C_q = A_r B^r_q \]
\[ C^p = A^p B^r \]
\[ C^p = A_r B^r \]
\[ C^p = A^p B \]
\[ C^p = A_r^p B^r \]

Example 3: Halo exchange
Incremental Conversion of the UKMet Climate Model to Co-Array Fortran

• Fields are allocated on the local heap
• One processor knows nothing about another processor’s local memory structure
• But each processor knows how to find co-arrays in another processor’s memory
• Define one supplemental co-array structure
• Create an alias for the local field through the co-array field
• Communicate through the alias
Co-array Alias to Local Fields

type field
    real,pointer :: ptr(:,:,)
end type field

real :: u(0:m+1,0:n+1,lev)
type(field) :: z[p,*]

z%ptr => u
u = z[p,q]%ptr
Irregular and Changing Data Structures
Problem Decomposition and Co-Dimensions

```
  N

 W | [p-1,q] | [p,q]  | [p+1,q]  |
   | [p,q+1] |         |          |
 S | [p,q-1] |         |          |
   | [p,q-1] |         |          |
 E
```
Cyclic Boundary Conditions
East-West Direction

real,dimension [p,*] :: z

myP = this_image(z,1)     !East-West
myQ = this_image(z,2)     !North-South

West = myP - 1
if(West < 1) West = nProcEW     !Cyclic

East = myP + 1
if(East > nProcEW) East = 1     !Cyclic
East-West Halo Swap

• Move last row from west to my first halo

\[ u(0, 1:n, 1:lev) = \text{z[West,myQ]} \% \text{ptr}(m, 1:n, 1:lev) \]

• Move first row from east to my last halo

\[ u(m+1, 1:n, 1:lev) = \text{z[East,myQ]} \% \text{Field}(1, 1:n, 1:lev) \]
Exercises

1. Write code for the North-South exchange.
2. Change the halo width to some value $w \geq 1$.
3. What happens if the sizes of the blocks on different images are not equal?
Where Can I Try CAF?
CRAY Co-Array Fortran

- CAF has been a supported feature of Cray Fortran since release 3.1
- CRAY T3E
  - f90 -Z src.f90
  - mpprun -n7 a.out
- CRAY X1
  - ftn -Z src.f90
  - aprun -n17 a.out
- CRAY XT4/5
  - ftn -hcaf src.f90
  - aprun -n13 a.out
Open Source g95 compiler

- Andy Vaught has produced a co-array compiler.
- Download from
  - [www.g95.org/downloads.shtml](http://www.g95.org/downloads.shtml)
  - [www.g95.org/coarray.shtml](http://www.g95.org/coarray.shtml)
- `ar -r libf95.a coarray.o`
- `g95 src.f90`
- `cocon -i4 a.out`
Other Efforts

- Rice University is developing a compiling system for CAF.
- University of Houston is developing a CAF compiler.
- IBM compiler and run-time system under development.
- Intel compiler under development.
References

## Total Time (s)

<table>
<thead>
<tr>
<th>PxQ</th>
<th>SHMEM</th>
<th>SHMEM w/CAF SWAP</th>
<th>MPI w/CAF SWAP</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x2</td>
<td>191</td>
<td>198</td>
<td>201</td>
<td>205</td>
</tr>
<tr>
<td>2x4</td>
<td>95.0</td>
<td>99.0</td>
<td>100</td>
<td>105</td>
</tr>
<tr>
<td>2x8</td>
<td>49.8</td>
<td>52.2</td>
<td>52.7</td>
<td>55.5</td>
</tr>
<tr>
<td>4x4</td>
<td>50.0</td>
<td>53.7</td>
<td>54.4</td>
<td>55.9</td>
</tr>
<tr>
<td>4x8</td>
<td>27.3</td>
<td>29.8</td>
<td>31.6</td>
<td>32.4</td>
</tr>
</tbody>
</table>
CAF and Object-Oriented Programming Methodology
Object-Oriented Programming combined with Co-Arrays

- Fortran 2003 is an object-oriented language.
  - allocate/deallocate for dynamic memory management
  - Named derived types are similar to classes
  - Type-associated methods.
  - Constructors and destructors can be defined to encapsulate parallel data structures.
  - Generic interfaces can be used to overload procedures based on the named types of the actual arguments.
A Parallel Class Library for CAF

• Combine the object-based features of Fortran 95 with co-array syntax to obtain an efficient parallel numerical class library that scales to large numbers of processors.

• Encapsulate all the hard stuff in modules using named objects, constructors, destructors, generic interfaces, dynamic memory management.
  
  
CAF Parallel Class Libraries

use BlockMatrices
use BlockVectors

type(PivotVector) :: pivot[p,*]
type(BlockMatrix) :: a[p,*]
type(BlockVector) :: x[*]

call newBlockMatrix(a,n,p)
call newPivotVector(pivot,a)
call newBlockVector(x,n)
call luDecomp(a,pivot)
call solve(a,x,pivot)
LU Decomposition

Figure 6: Time as a function of the number of processors $p = q \times r$ for block-cyclic LU decomposition. The matrix size is $1000 \times 1000$ with blocks of size $48 \times 48$. Time is expressed in dimensionless giga-clock-ticks, $\nu t \times 10^{-9}$, as measured on a CRAY-T3E with frequency $\nu = 300$MHz. The dotted line represents perfect scaling. The curve marked with bullets (●) is code written in Co-Array Fortran. The curve marked with triangles (▲) is SCALAPACK code.
Communication for LU Decomposition

- **Row interchange**
  - temp(,:) = a(k,:)
  - a(k,:) = a(j,:) [p,myQ]
  - a(j,:) [p,myQ] = temp(:)

- **Row “Broadcast”**
  - L0(i:n,i) = a(i:,n,i) [p,p] i=1,n

- **Row/Column “Broadcast”**
  - L1(:,:) = a(:,:) [myP,p]
  - U1(:,:) = a(:,:) [p,myQ]
Cyclic-Wrap Distribution

1  2  3  4  5  6  7

1  4  7  2  5  3  6

1  4  7

2  5

3  6
Vector Objects

type vector
  real, allocatable :: vector(:)
  integer :: lowerBound
  integer :: upperBound
  integer :: halo
end type vector
Block Vectors

type BlockVector
  type(VectorMap) :: map
  type(Vector), allocatable :: block(:)
  --other components--
end type BlockVector
Block Matrices

type BlockMatrix
  type(VectorMap) :: rowMap
  type(VectorMap) :: colMap
  type(Matrix), allocatable :: block(:, :)
  --other components--
end type BlockMatrix
CAF I/O for Named Objects

use BlockMatrices
use DiskFiles

type(PivotVector) :: pivot[p,*]
type(BlockMatrix) :: a[p,*]
type(DirectAccessDiskFile) :: file

call newBlockMatrix(a,n,p)
call newPivotVector(pivot,a)
call newDiskFile(file)
call readBlockMatrix(a,file)
call luDecomp(a,pivot)
call writeBlockMatrix(a,file)
Summary
Why Language Extensions?

• Programmer uses a familiar language.
• Syntax gives the programmer control and flexibility.
• Compiler concentrates on local code optimization.
• Compiler evolves as the hardware evolves.
  – Lowest latency and highest bandwidth allowed by the hardware
  – Data ends up in registers or cache not in memory
  – Arbitrary communication patterns
  – Communication along multiple channels
Summary

• Co-dimensions match your logical problem decomposition
  – Run-time system matches them to hardware decomposition
  – Explicit representation of neighbor relationships
  – Flexible communication patterns
• Code simplicity
  – Non-intrusive code conversion
  – Modernize code to Fortran 2003 standard
• Code is always simpler and performance is always better than MPI.
sync images()

me = this_image()
ne = num_images()
if(me == 1) then
    p = 1
else
    sync images(me-1)
    p = p[me-1] + 1
end if
if(me<ne) sync images(me+1)
Proposed Synchronization

\texttt{notify()}/\texttt{query()}

Asynchronous split barrier

\texttt{sync team(teamObject)}

Synchronize within a subset of images.

\texttt{collectives}

co\_sum, co\_max, co\_min, etc.