High Performance Fortran

http://www-jics.cs.utk.edu
jics@cs.utk.edu

Kwai Lam Wong
Overview

• HPF : High Performance FORTRAN
• A language specification standard by High Performance FORTRAN Forum (HPFF), a coalition founded in Jan 1992.
• A portable extension to F90 for writing data parallel applications at a relatively high level
• Version 1.1 of the language specification was published November 1994.
• Version 2 , 1997
• Main goals: Data parallel programming, SPMD paradigm
HPF Features

- HPF concentrates on data parallelism and as such deals with global data structures
- All of FORTRAN 90
- FORALL construct
- INDEPENDENT directive before loops means “parallel safe”
- Provide the high-level data mapping strategy using **directives**
  - Distributive directive for data distribution
  - Align directive for data alignment
- No parallel I/O support
- Little support for irregular data structure computations
- Little support for non-data parallelism
To compile a source code program, use either xlhpf or xlhpf90 command. Both of these command automatically turn on the -qhpf compiler option, which is required to produce a parallel program.

- xlhpf90 compiles program using F77 conventions while xlhpf90 compiles programs using F90 conventions.
- Use xlhpf90 command whenever possible for HPF program.
  - xlhpf90 -o fileexe file.f

Because xlhpf90 on SP is a POE application. To run the fileexe, you may attach any poe variables as the option
  - poe fileexe -nprocs 4 -hostfile host.list

For Portland Group HPF, compiler with pghpf
Example : hello.f

```fortran
program hello
    implicit none
    double precision t1, t2, t3, rtc, x, y, z
    t1 = rtc()
    x = 1.0d0; y = 2.0d0; z = 0.0d0
    z = x + y
    t2 = rtc()
    t3 = t2 - t1
    write(6,*) 'hello world, 1+2 = ', z
    write(6,*) 'Time needed to compute the value is ', t3
    stop
end
```

```
xlhp90 -o hello hello.f
poe hello -procs 4 -labelio yes

0: hello world, 1+2 = 3.0000
0: Time needed to compute the value is 0.9536E-06
```


Example Again

```fortran
program hello
implicit none
double precision t1,t2,t3,rtc.x.y.z
interface
extrinsic(HPF_LOCAL) subroutine sayit (t1,t2,z)
   real*8, intent(in) :: t1,t2,z
   end subroutine sayit
end interface

end interface

! Timing
! Start

   t1 = rtc()
x = 1.0d0 ; y =2.0d0 ; z = 0.0d0 ; z=x+y
   t2 = rtc() ; call sayit(t1,t2,z)

! Stop

stop; end

extrinsic(HPF_LOCAL) subroutine sayit(t1,t2,z)
double precision t1,t2,t3,z
   t3=t2-t1
   write(6,*) 'hello world, 1+2 =',z
   write(6,*) ' Time needed to compute the value is ',t3
   return; end
```

% xlhpf90 -o hello hello.f
% poe hello -procs 4 -labelio yes

0: hello world, 1+2 = 3.0000
2 :hello world, 1+2 = 3.000
3 :hello world, 1+2 = 3.0000
1 : hello world, 1+2 = 3.0000
0: Time needed to compute the vaule is 0.9536E-06
2: Time needed to compute the vaule is 0.9536E-06
3: Time needed to compute the vaule is 0.9536E-06
1: Time needed to compute the vaule is 0.9536E-06
```
F90 Features

- Major extension of F77
- All of FORTRAN 77
- Syntax improvements including free-form source
- Array operations
- New intrinsic procedures (arrays, bit manipulation)
- New and improved control constructs
- Dynamics storage allocation
- User-defined data types
- Pointers
- Procedure interfaces
Syntax Improvements

- Statements may appear anywhere
- Column 1-6 are no longer reserved
- Line continuation -- “&”
- Trailing comments may be used -- “!”
- Multiple statements allowed in one line
- Underline symbol “_” is permitted
- 31 characters for length of variables
- Example:
  
  TMP_VALUE_OF_X = X ; X = Y ; Y &
  = TMP_VALUE_OF_X ! swap X and Y
Language Elements

- Attributes are extra properties of variables in Type specifications
  
  INTEGER, PARAMETER :: n=1000
  REAL, DIMENSION(n,n) :: a, b

- Data Types:
  
  INTEGER, DIMENSION(10) :: m, n
  REAL X(-10 : 20), Y(1:50)
  CHARACTER :: CH
  LOGICAL :: TF

- Do loops:
  
  DO I = 0, 10
  M(I) = I*I + 1
  END DO
Arrays Features

F77

REAL A(50,50), B(50,50), C(50,50)
DO I =1, 50
  DO J = 1,50
    C(I,J) = A(I, J) + B(I, J)
  END DO
END DO

F90

REAL, DIMENSION(50,50) :: A, B, C
C = A + B

REAL, DIMENSION(5,20) :: X, Y
REAL, DIMENSION(-2:2, 1:20) :: Z
!elementwise multiplication
Z = 4.0*Y * X
## Array Allocations

### Allocatable Array
*(creation and destruction are user-controlled)*

```
PROGRAM simulate
IMPLICIT NONE
INTEGER :: n
INTEGER, DIMENSION(:, :), &
ALLOCATABLE :: a
PRINT *, n
......
ALLOCATE( a (n, 2*n) )
 ......
DEALLOCATE ( a )
END
```

### Automatic Arrays
*(created on entry and destroyed on exit from procedure)*

```
PROGRAM auto_array
INTEGER :: n,m
READ *, n,m
CALL sim(n,m)
END

SUBROUTINE sim(n,m)
REAL :: a(n,m), b(m)
 ......
RETURN
END
```
Assumed-Shape Array

- Assume shape of actual argument to which it is associated

```fortran
 subroutine asshape( f, isign, indx)
 !assumed-shape array
 real, dimension( : ) :: f
 integer isign, indx
 .......
 end subroutine asshape
```

- Assumed-shape arrays require an explicit interface (compiler verifies matching arguments)

```fortran
program main
 integer, parameter :: isign=0, indx=10, nx=2**indx
 real, dimension(nx) :: f
 interface
 subroutine asshape(a, j,k)
 real, dimension( : ) :: a
 integer :: j, k
 end subroutine asshape
 end interface
 call asshape(f, isign, indx)
 end
```
Interface Blocks

• Interface blocks provide the compiler with all the information it needs to allow it to make consistency checks and ensure that enough information is communicated to procedure at run-time.

• An interface declaration gives the characteristics (attributes) of both the dummy arguments (e.g. name, kind, type, and rank) and the procedure (e.g. name, class, and type).

```
Interface
  subroutine of function header
  declarations of dummy arguments
  end subroutine or function
end interface
```
# Statement Ordering

<table>
<thead>
<tr>
<th>PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USE</strong></td>
</tr>
<tr>
<td><strong>IMPLICIT NONE</strong></td>
</tr>
<tr>
<td><strong>PARAMETER</strong></td>
</tr>
<tr>
<td><strong>IMPLICIT</strong></td>
</tr>
<tr>
<td>Derived-Type Definition, Interface blocks, Declarations, Statement functions</td>
</tr>
<tr>
<td><strong>DATA</strong></td>
</tr>
<tr>
<td>Executable constructs</td>
</tr>
<tr>
<td><strong>CONTAINS</strong></td>
</tr>
<tr>
<td>Internal or module procedures</td>
</tr>
<tr>
<td><strong>END</strong></td>
</tr>
</tbody>
</table>
Module and Use Statements

- Modules contain declarations, functions, and type definitions that can be conveniently accessed and used by executable program units.

- Modules create interface blocks automatically. It is sometimes advantageous to package INTERFACE definitions into a module.

```fortran
module test
    ........ declarations ....
    contains
        subroutine abc(x,y)
        ........
        end subroutine abc
    end module test
```

- `Use` statement makes modules “available”, like F77 `COMMON` and `INCLUDE`
### Example: Module

**Module swapping**
```fortran
module swapping
    contains
    subroutine swap (x, y)
        real, intent(inout) :: x, y
        tmp = x; x = y; y = tmp
    end subroutine swap
end module swapping
```

**Module Pye**
```fortran
module Pye
    ! save makes pi a global constant
    ! acts like common
    real, save :: pi = 3.1415926
end module Pye
```

**Program Area**
```fortran
program Area
    use Pye
    implicit none
    real :: r
    read *, r
    print* , " Area = ", pi * r * r
end program Area
```

**Program trymodule**
```fortran
program trymodule
    use swapping
    real :: a = 1.0 , b = 2.0
    call swap(a, b)
end program trymodule
```
Derived Types

- User defined type from intrinsic and previously defined types
- Various components can be unified by a derived type

```fortran
type private_complex
    real :: real, imaginary
end type private_complex

type (private_complex) :: a, b, c
a%real = 1.0
b%imaginary = 2.0
c%real = a%real*b%real - a%imaginary*b%imaginary
c%imaginary = a%real*b%imaginary + a%imaginary*b*real
```
Encapsulation in Modules

- Grouping of data and operations into a single well-defined unit

```fortran
module private_complex_module
  type private_complex  ! define type
      real :: real, imaginary
  end type private_complex
  contains
    type (private_complex) function pc_mult(a,b)  ! function def. type (private_complex), intent (in) : : a, b
    pc_mult%real = a%real*b%real - a%imaginary*b%imaginary
    pc_mult%imaginary = a%real*b%imaginary + &
                       a%imaginary*b%real
    end function pc_mult
  end contains
end module private_complex_module
```
Encapsulation (Cont’d)

• A main program to multiple two private_complex numbers

```fortran
program main
  use private_complex_module  ! bring in the module
  type (private_complex) :: a, b, c
    a%real = 1.0
    a%imaginary = -1.0
    b%real = -1.0
    b%imaginary = 2.0
    c = pc_mult(a, b)
    print *, 'c=', c%real, c%imaginary
  stop
end program main
```
FORTRAN 90 Pointers

- Pointers variable do not hold data, they point to scalar or array variables which themselves may contain data
- Target: the space to which a pointer variable points
- Pointer and target Declarations:

  ```fortran
  real, pointer :: ptor
  real, dimension(:,,:), pointer :: ptoareal
  real, target :: x, y ! may associate with ptor
  real dimension(5,3), target :: a, b ! may associate with ptoareal
  x = 3.1416
  ptor => y ! pointer assignment (aliasing)
  ptor = x ! “normal” assignment, y = x
  nullify(ptrr) ! disassociate pointer from y
  ptoa => a(3:5:2, ::2)
  ```
Intrinsic Functions

- Array construction functions
  - SPREAD, PACK, RESHAPE,…
- Vector and matrix multiplication
  - DOT_PRODUCT, MATMUL
- Reduction functions
  - SUM, PRODUCT, COUNT, MAXCAL, ANY, ALL...
- Geometric location functions
  - MAXLOC, MINLOC
- Array manipulation functions
  - CSHIFT, EOSHIFT, TRANSPOSE…..
Examples

REAL :: a(100), b(4,100)
scalar = SUM(a) ! sum of all elements
a = PRODUCT( b, DIM=1) ! product of elements in first dim
scalar = COUNT( a = 0) ! gives number of zero elements
scalar = MAXVAL ( a , MASK = a .LT. 0)

LOGICAL a(n)
REAL, DIMENSION(n) :: b, c
IF ( ALL(a) ) ....... ! global AND
IF ( ALL(b = c) ....... ! true if all elements equal
IF ( ANY(a) ) ....... ! global OR
IF ( ANY( b < 0.0 ) ....... ! true if any elements < 0.0

XOUT = CC* X + CN* CSHIFT(X, +1, 2) + CS * CSHIFT( X, -1, 2) 
       + CE* CSHIFT( X, +1, 1) + CW * CSHIFT( X, -1, 1)
Data-Parallel Statements

- Data parallelism emphasizes having many fine-grain operations, such as computations on every element of an array
- HPF has several ways to exploit data parallelism:
  - Array expressions: Taken from F90
  - FORALL: Tightly-coupled parallel execution based on the structure of an indexed space
  - INDEPENDENT: Assertion that iterations do not interface with each other
  - HPF library and intrinsics: Extended from F90
FORALL

- A = [0, 1, 2, 3, 4], B = [0, 10, 20, 30, 40], C = [0, 0, 0, 0, 0]
- DO LOOP:

<table>
<thead>
<tr>
<th>DO I = 2, 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(I) = A(I-1) + A(I+1)</td>
</tr>
<tr>
<td>C(I) = B(I) * A(I + 1)</td>
</tr>
</tbody>
</table>

END DO

A = [0, 2, 5, 9, 4], C = [0, 20, 60, 120, 0]

- FORALL:

<table>
<thead>
<tr>
<th>FORALL (I = 2 : 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(I) = A(I - 1) + A(I + 1)</td>
</tr>
<tr>
<td>C(I) = B(I) * A(I + 1)</td>
</tr>
</tbody>
</table>

END FORALL

A = [0, 2, 4, 6, 4], C = [0, 40, 120, 120, 0]
**DO LOOP**

- **I = 2**
  
  \[
  A(2) = A(1) + A(3) = 0 + 2 = 2 \\
  C(2) = B(2) \cdot A(3) = 10 \cdot 2 = 20
  \]

- **I = 3**
  
  \[
  A(3) = A(2) + A(4) = 2 + 3 = 5 \\
  C(3) = B(3) + A(4) = 20 \cdot 3 = 60
  \]

- **I = 4**
  
  \[
  A(4) = A(3) + A(5) = 5 + 4 = 9 \\
  C(4) = B(4) \cdot A(5) = 30 \cdot 4 = 120
  \]

**FORALL**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A(2)</td>
<td>A(1) + A(3)</td>
<td>A(3) = A(2) + A(4)</td>
</tr>
<tr>
<td></td>
<td>= 0 + 2 = 2</td>
<td>= 1 + 3 = 4</td>
</tr>
<tr>
<td>C(2)</td>
<td>B(2) \cdot A(3)</td>
<td>C(3) = B(3) \cdot A(4)</td>
</tr>
<tr>
<td></td>
<td>= 10 \cdot 4 = 40</td>
<td>= 20 \cdot 6 = 120</td>
</tr>
</tbody>
</table>
Syntax of FORALL

- Syntax:

  ```fortran
  FORALL (index-spec-list, mask-expr)
  assignments
  END FORALL
  ```

- Semantics:
  - Equivalent to array assignment in F90
  - For each value of indices, check the mask
  - Compute RHS for unmasked values
  - Assign values to LHS for unmasked values
  - Each assignment statement is completed before the next one starts
INDEPENDENT Directive

• Syntax:
  \[
  \text{!HPF$ INDEPENDENT, NEW(variables)}
  \]

• Semantics:
  – INDEPENDENT is an assertion that no iteration affects any other iteration in any way
  – NEW variables create new instances of variables for each iteration

• Applied to a DO LOOP:
  – No loop carried dependencies (except for NEW variables)

• Applied to a FORALL
  – No index point assigns to any location that another uses
The INDEPENDENT directive is used to provide the compiler with additional information about the execution of a FORALL construct or a DO loop. This is intended to allow the compiler to generate a parallel loop in situations where the parallelism is not obvious, but require knowledge of the values of the data. It is a promise by the user to the compiler that the result of the loop statements will be the same whether executed in serial or in parallel.

```
!HPF$ INDEPENDENT
DO J = 1, 3
   A(J) = A( B (J) )
   C(A(J)) = A( J ) * B( A ( J ) )
END DO

!HPF$ INDEPENDENT
FORALL ( J = 1 : 3 )
   A(J) = A( B (J) )
   C(A(J)) = A( J ) * B( A ( J ) )
END FORALL
```
Data Mapping

- Data mapping complements data parallelism by placing data for parallel access
- HPF uses a two-phase data mapping:
  - ALIGN:
    - Creates a relationship between objects
  - DISTRIBUTED:
    - Partitions an object between processors
  - Vendors may define further levels of mapping

```
ARRAYS ➔ ARRAYS ➔ VIRTUAL PROCESSORS ➔ PHYSICAL PROCESSORS
ALIGN ➔ DISTRIBUTED ➔ VENDOR EXTENSIONS
```
PROCESSORS Directive

- Specified the arrangement of abstract processors
- Syntax:

  \[
  \text{!HPF$ PROCESSORS processors-name(shape)}
  \]
  \[
  \text{!HPF$ PROCESSORS procs(3,2) ! assumes 6 physical \& processors, and constructs a 3 x 2 array of abstract processors}
  \]
- Rules:
  - The shape must contain the extents of all dimensions of the abstract processor array.
  - If no shape is specified, then the processor arrangement is scalar
DISTRIBUTE Directive

- Syntax:
  \[ !\text{HPF}\$ \text{DISTRIBUTE } \text{array (dist-format-list) ONTO procs} \]
  \[ !\text{HPF}\$ \text{DISTRIBUTE (dist-format-list) ONTO procs :: array-list} \]

- Semantics:
  - Elements of array are divided according to the corresponding pattern in dist-format-list

- Options for dist-format-list:
  - \text{BLOCK}
  - \text{CYCLIC}
  - \text{CYCLIC(K)}
  - \text{* : Dimension not distributed}
**Examples**

```fortran
!HPF$ PROCESSORS procs(2,2)
INTEGER, DIMENSION(100,100) : : A
!HPF$ DISTRIBUTE ( BLOCK, BLOCK) ONTO procs : : A
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1,1) A(1,50)</td>
<td>A(1,51) A(1,100)</td>
</tr>
<tr>
<td>proc(1,1)</td>
<td>proc(1,2)</td>
</tr>
<tr>
<td>A(50,1) A(50,50)</td>
<td>A(50,51) A(50,100)</td>
</tr>
<tr>
<td>A(51,1) A(51,50)</td>
<td>A(51,51) A(51,100)</td>
</tr>
<tr>
<td>proc(2,1)</td>
<td>proc(2,2)</td>
</tr>
<tr>
<td>A(100,1) A(100,50)</td>
<td>A(100,51) A(100,100)</td>
</tr>
</tbody>
</table>
```
Examples of Distribute

> !HPFS$ DISTRIBUT W(BLOCK, *)
> !HPFS$ DISTRIBUT X(*, CYCLIC)
> !HPFS$ DISTRIBUT Y(BLOCK, BLOCK)
> !HPFS$ DISTRIBUT Z(CYCLIC(2), CYCLIC(3))
ALIGN Directive

- Syntax:
  
  ```fortran
  !HPF$ ALIGN array(source-list) WITH target(subscript-list)
  !HPF$ ALIGN (source-list) WITH target (subscript-list) :: array-list
  ```

- Semantics:
  
  - Creates a relationship between array and target so that for all values of the source-list variables, array(source-list) and target (subscript-list) stored on the same processor

- Advantages:
  
  - Enhances locality
  
  - Replication ( * in subscript-list )
Examples of Align

> !HPFS ALIGN X(I, J) WITH W(I, I)
> !HPFS ALIGN Y(K) WITH W(K, *)
> !HPFS ALIGN Z(L) WITH X(3, 2*L-1)

Using `DISTRIBUTE W(BLOCK, *)`

Using `DISTRIBUTE W(*, BLOCK)`
**Example : MM**

- Multiple matrix A and B to form a matrix C
- Keep rows of A and columns of B on the same processor

```fortran
PROGRAM MM
  IMPLICIT NONE
  INTEGER, PARAMETER :: N = 100
  INTEGER, DIMENSION (N, N) :: A, B, C
  INTEGER :: I, j
  REAL*8 rtc, t1, t2, t3
  !HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO procs :: C
  !HPF$ ALIGN A(:, :) WITH C(:, :)
  !HPF$ ALIGN B(:, :) WITH C(:, :)
  t1 = rtc()
  A = 1 ; B = 2 ; C = 0
  c = matmul (A, B)
  t2 = rtc()
  t3 = t2 - t1
  write(6,*) 'Timing for matrix multiple : ', t3
END
```

9/22/97

*High Performance Fortran*
Example : MM

- Multiple matrix A and B to form a matrix C
- Keep rows of A and columns of B on the same processor

```fortran
PROGRAM MM
    IMPLICIT NONE
    INTEGER, PARAMETER :: N = 100
    INTEGER, DIMENSION (N, N) :: A, B, C
    INTEGER :: I, j
    !HPF$ PROCESSORS square(2,2)
    !HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO procs :: C
    !HPF$ ALIGN A(I,*) WITH C(I, j)
    !HPF$ ALIGN B(*,j) WITH C(I,j)
    A = 1 ; B = 2 ; C = 0
    DO I = 1 , N
        DO j = 1, N
            C(I,j) = DOT_PRODUCT( A(I, :) , B( :, j) )
        END DO
    END DO
END
```
**TEMPLATE Directive**

- Create conceptual objects for clearer and cleaner alignment without actually allocating memory for the array.

```fortran
PROGRAM TMP
IMPLICIT NONE
REAL  C( 1:4 ), D( 1:8 ), E (1:2)
!HPF$ TEMPLATE, DIMENSION ( 8 ) : : T
!HPF$ ALIGN  D( : ) WITH T (: )
!HPF$ ALIGN  C( : ) WITH  T( : : 2)
!HPF$ ALIGN   E( : ) WITH  T( : : 4)
!HPF$ DISTRIBUTE (BLOCK) : : T
    C = 1 ;   D = 2
    E = D ( : : 4) +  C ( : : 2)
END PROGRAM TMP
```
Dynamic Data Mapping

• One data mapping is not always appropriate for an entire program, therefore HPF needs executable DISTRIBUTE and ALIGN : REDISTRIBUTE and REALIGN
• Both REDISTRIBUTE and REALIGN require that the argument must have been declared with the DYNAMIC attribute

!HPF$ DISTRIBUTE D1(CYCLIC)
!HPF$ DISTRIBUTE D2(BLOCK), DYNAMIC
!HPF$ DYNAMIC, ALIGN WITH D2 :: A1, A2

........
!HPF$ REDISTRIBUTE D2( CYCLIC(2))
....
!HPF$ REALIGN A2 WITH D1
Rules of Thumb for Communication

- Computations will execute in parallel IF:
  - They are done in array operations
  - The data is partitioned (by `DISTRIBUTE`)
- BLOCK:
  - Good for local (nearest-neighbor) communication
  - Reduces surface-to-volume ratio
- CYCLIC:
  - Improve load balance
- CYCLIC (K):
  - Has some advantage of each
- Strides and broadcasts are expensive on any distribution
- Communication between different distributions is very expensive
Examples From CRPC

Solving a 2D Poisson equation
Jacobi Iteration: The Algorithm

• The Problem
  - Given a partial differential equation & boundary conditions
  - Find the solution

\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -2x^2 + 2x - 2y^2 + 2y \]

\[ u = 0 \text{ if } x = 0, x = 1, y = 0, \text{ or } y = 1 \]

• The Approach
  - Divide (continuous) space into a (discrete) grid
  - Guess a solution on the grid
  - Update the solution at every grid point
  - Repeat update until solution doesn't change
Jacobi Iteration: Equations and Pictures

Discretized Equations

\[ 4u_{1,1} - u_{2,1} - u_{1,2} = -0.00220 \]
\[ 4u_{1,2} - u_{2,2} - u_{1,3} - u_{1,1} = -0.00293 \]
\[ 4u_{1,3} - u_{2,3} - u_{1,2} - u_{1,3} = -0.00220 \]
\[ 4u_{2,1} - u_{1,1} - u_{2,2} - u_{2,3} = -0.00293 \]
\[ 4u_{2,2} - u_{1,2} - u_{2,3} = -0.00391 \]
\[ 4u_{2,3} - u_{1,3} - u_{2,2} = -0.00293 \]
\[ 4u_{3,1} - u_{2,1} - u_{3,2} = -0.00220 \]
\[ 4u_{3,2} - u_{2,2} - u_{3,3} = -0.00220 \]
\[ 4u_{3,3} - u_{2,3} = -0.00220 \]

Reordered Equations

\[ u_{1,1} = \frac{u_{2,1} + u_{1,2} - 0.00220}{4} \]
\[ u_{1,2} = \frac{u_{2,2} + u_{1,1} + u_{1,3} - 0.00293}{4} \]
\[ u_{1,3} = \frac{u_{2,3} + u_{1,2} - 0.00220}{4} \]
\[ u_{2,1} = \frac{u_{1,1} + u_{3,1} + u_{2,2} - 0.00293}{4} \]
\[ u_{2,2} = \frac{u_{1,2} + u_{3,2} + u_{2,3} - 0.00391}{4} \]
\[ u_{2,3} = \frac{u_{1,3} + u_{3,3} + u_{2,2} - 0.00293}{4} \]
\[ u_{3,1} = \frac{u_{2,1} + u_{3,2} - 0.00220}{4} \]
\[ u_{3,2} = \frac{u_{2,2} + u_{3,3} - 0.00220}{4} \]
\[ u_{3,3} = \frac{u_{2,3} + u_{3,2} - 0.00220}{4} \]
Jacobi Iteration: Parallelism

- Each Jacobi iteration uses all the data computed in the previous step
  - No parallelism at this level
  - (We won’t try other iterative schemes to avoid this)

- All updated elements within an iteration can be updated in parallel
  - \( u_{\text{new}}(i,j) = \frac{u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) + f(i,j)}{4} \)
  - These are independent because \( u_{\text{new}} \neq u \) and \( u_{\text{new}} \neq f \)
  - This is a classic data-parallel operation

- Testing for convergence can be done in parallel
  - Convergence criteria: Largest element in array
  - Searching for the maximum is a data-parallel reduction
Jacobi Iteration: Data Mapping

- Convergence test requires a whole-array reduction
  - Any distribution ⇒ parallel, with communication

- Element updates require local value and nearest neighbors
  - BLOCK ⇒ least communication volume
  - CYCLIC ⇒ communicate entire array
  - (BLOCK,*) ⇒ move u(i-1,j), u(i+1,j) ∀ j
  - (*,BLOCK) ⇒ move u(i,j-1), u(i,j+1) ∀ i
  - (BLOCK,BLOCK) ⇒ move u(ILOW-1,j), u(ITHIGH+1,j) ∀ j,
    u(i,JLOW-1), u(i,JHIGH+1) ∀ i

- Computation is static and homogenous
  - No load balancing issues

- The bottom line
  - (BLOCK,*) or (*,BLOCK) on high-latency machines or small problem sizes
  - (BLOCK,BLOCK) on low-latency machines
Jacobi Iteration: HPF Program

REAL u(0:nx,0:ny), unew(0:nx,0:ny), f(0:nx,0:ny)
!HPF$ DISTRIBUTED u(BLOCK,*)
!HPF$ ALIGN BY u(:,): unew(:,), f(:,)

dx = 1.0/nx; dy = 1.0/ny; err = tol * 1e6
FORALL ( i=0:nx, j=0:ny )
    f(i,j) = -2*(dx*i)**2+2*dx*i-2*(dy*j)**2+2*dy*j
END FORALL
u = 0.0; unew = 0.0

DO WHILE (err > tol)
    DO ALL ( i=1:nx-1, j=1:ny-1 ) &
        unew(i,j) = (u(i-1,j)+u(i+1,j)+u(i,j-1)+ &
                    u(i,j+1)+f(i,j))/4
    err = MAXVAL( ABS(unew-u) )
    u = unew
END DO
Conjugate Gradient: The Algorithm

• The Problem
  - Given a partial differential equation & boundary conditions
  - Find the solution

• The Approach
  - Divide (continuous) space into a (discrete) grid
  - Guess a solution on the grid
  - Estimate how the solution should change
  - Move in that direction
  - Repeat estimate and move until solution doesn’t change

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -2x^2 + 2x - 2y^2 + 2y
\]

\[
u = 0 \text{ if } x = 0, x = 1, y = 0, \text{ or } y = 1
\]
Conjugate Gradient: Equations and Pictures

Minimize

$r = (Au - f)^T (Au - f)$

Discretized Equations

\[
\begin{align*}
4u_{1,1} - u_{2,1} - u_{1,2} &= -0.00220 \\
4u_{1,2} - u_{2,2} - u_{1,3} &= -0.00293 \\
4u_{1,3} - u_{2,3} - u_{1,1} &= -0.00220 \\
4u_{2,1} - u_{2,2} - u_{3,1} &= -0.00293 \\
4u_{2,2} - u_{2,3} - u_{2,1} &= -0.00391 \\
4u_{2,3} - u_{2,1} - u_{2,2} &= -0.00293 \\
4u_{3,1} - u_{3,2} - u_{3,3} &= -0.00220 \\
4u_{3,2} - u_{3,3} - u_{3,1} &= -0.00293 \\
4u_{3,3} - u_{3,2} - u_{3,1} &= -0.00293 \\
4u_{3,3} - u_{3,2} - u_{3,1} &= -0.00220
\end{align*}
\]

$A = \begin{bmatrix}
4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 4 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 4 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 4 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 4 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 4 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 4 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 4
\end{bmatrix}$

$f = \begin{bmatrix}
-0.00220 \\
-0.00293 \\
-0.00220 \\
-0.00293 \\
-0.00391 \\
-0.00293 \\
-0.00220 \\
-0.00293 \\
-0.00220
\end{bmatrix}$
Conjugate Gradient: More Equations

\[ u = \langle \text{initial guess} \rangle \]
\[ r = f - A \ast u \]
\[ \delta = \max( |r| ) \]
\[ i = 0; \, \rho = 0 \]
\[ \text{WHILE (} \delta > \varepsilon \text{) DO} \]
\[ i = i + 1; \, \rho_{\text{old}} = \rho \]
\[ \rho = r \cdot r \]
\[ \text{IF (} i=1 \text{) THEN } p = r \text{ ELSE } p = r + \rho / \rho_{\text{old}} \cdot p \]
\[ q = A \ast p \]
\[ \alpha = \rho / ( p \cdot q ) \]
\[ u = u + \alpha \cdot p \]
\[ r = r - \alpha \cdot q \]
\[ \delta = \max( |r| ) \]
\[ \text{END WHILE} \]
Conjugate Gradient: Parallelism

- Each CG iteration uses all the data computed in the previous step, plus data computed in the current step
  - No parallelism at this level
  - (We won't try to overlap computation within a step)

- Each matrix operation can compute elements in parallel
  - \( r(i,j) = f(i,j) - 4 \times u(i,j) + u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) \)
  - \( p(i,j) = r(i,j) + \rho \times \rho_{old} \times p(i,j) \)
  - These are classic data-parallel operations
  - \( \rho = \sum_{i,j} r(i,j)^2 \)
  - This is a data-parallel reduction

- Testing for convergence can be done in parallel
  - As in Jacobi iteration
Conjugate Gradient: Data Mapping

• Convergence test requires a whole-array reduction
  – Any distribution ⇒ parallel, with communication

• Inner products require whole-array reductions
  – Any distribution ⇒ parallel, with communication

• Array operations require local value and nearest neighbors
  – \texttt{BLOCK} ⇒ least communication volume
  – \texttt{(BLOCK,*)} ⇒ move \(u(i-1,j), u(i+1,j) \forall j\)
  – \texttt{(BLOCK,BLOCK)} ⇒ move \(u(ILOW-1,j), u(IHIGH+1,j) \forall j, u(i,JLOW-1), u(i,JHIGH+1) \forall i\)

• Computation is static, homogenous, and over full array
  – No load balancing issues

• The bottom line
  – \texttt{(BLOCK,*)} or \texttt{(*,BLOCK)} on high-latency machines or small problem sizes
  – \texttt{(BLOCK,BLOCK)} on low-latency machines
REAL u(0:n,0:n), r(0:n,0:n), p(0:n,0:n)
REAL q(0:n,0:n), f(0:n,0:n)
//HPF$ DISTRIBUT u(BLOCK,*)
//HPF$ ALIGN WITH u(:,:) :: r(:,:), p(:,:)
//HPF$ ALIGN WITH u(:,:) :: q(:,:), f(:,:)
INTERFACE
SUBROUTINE a_times_vector( x, y )
REAL, INTENT(IN) :: x(:,:)
REAL, INTENT(OUT) :: y(:,:)
//HPF$ DISTRIBUT x *(BLOCK,*)
//HPF$ ALIGN y(:,:) WITH *x(:,:)
END INTERFACE

u = 0.0
r = f
err = MAXVAL( ABS(r(1:n-1,1:n-1)) )
i = 0; rho = 0

DO WHILE (err > tol)
i = i + 1; rho_old = rho
rho = SUM( r(1:n-1,1:n-1)**2 )
IF (i=1) THEN
  p = r
ELSE
  p = r + rho/rho_old * p
END IF
CALL a_times_vector(p, q)
alpha = rho / SUM(p*q)
u = u + alpha * p
r = r - alpha * q
err = MAXVAL(ABS(r(1:n-1,1:n-1)))
END DO