Outline

1. Introduction to Data-Parallelism
2. Fortran 90 Features
3. HPF Parallel Features
4. HPF Data Mapping Features
5. Parallel Programming in HPF
6. HPF Version 2.0

Plan of Attack

- **Four algorithms**
  - Gaussian Elimination
  - Jacobi Iteration
  - Conjugate Gradient
  - Irregular Mesh Relaxation
- **For each one, look at**
  - The algorithm
  - Parallelism
  - Data distribution
  - Resulting HPF program

A Simple Model for Programs

$$T_{total} = T_{seq} + \frac{T_{par}}{P} + T_{comm}$$

T_{total} = Total execution time

T_{seq} = Sequential execution time

T_{par} = Total parallel computation time

P = Number of processors

T_{comm} = Communication / synchronization time

Gaussian Elimination: The Algorithm

- **The Problem**
  - Given N linear equations in N unknowns x_i
  - Find values of all x_i to satisfy the equations
- **The Approach**
  - Use Eq. 1 to eliminate x_1 from Eq. 2, 3, … N
  - Use Eq. 2 to eliminate x_2 from Eq. 3, … N
  - …
  - Eq. N only involves $x_N \Rightarrow$ Solve it!
  - Work backwards to find x_{N-1}, …, x_1

$$2.00\,x_1 + 1.00\,x_2 + 0.50\,x_3 + 0.25\,x_4 = 3.75$$
$$2.00\,x_1 + 2.00\,x_2 + 2.00\,x_3 + 2.00\,x_4 = 8.00$$
$$2.00\,x_1 + 3.00\,x_2 + 4.50\,x_3 + 6.75\,x_4 = 16.5$$
$$2.00\,x_1 + 4.00\,x_2 + 8.00\,x_3 + 16.0\,x_4 = 30.0$$

$$\Downarrow$$

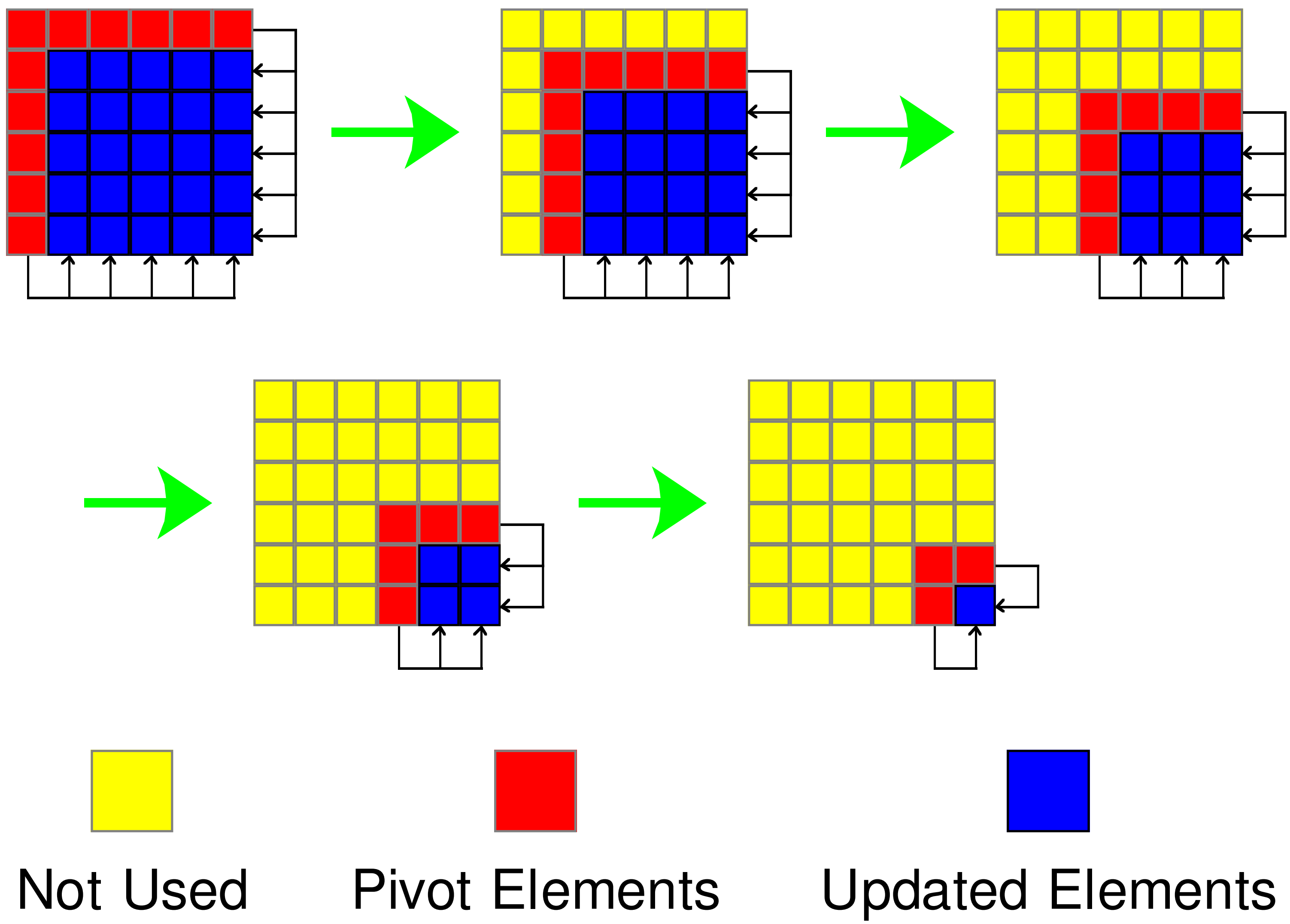
$x_1 = 1.00$

$x_2 = 1.00$

$x_3 = 1.00$

$x_4 = 1.00$

Gaussian Elimination: Pictures



Gaussian Elimination: Parallelism

- **Each step of Gaussian Elimination uses all the data computed in the previous step**
  - No parallelism at this level
  - (We won't try to reorder computations to avoid this)
- **All updated elements within a step can be done in parallel**
  - Basic formula: `a(i,j)=a(i,j)-a(i,K)/a(K,K)*a(K,j)`
  - These are independent because $i \neq K$, $j \neq K$
  - This is a classic data-parallel operation
- **Choosing the pivot row can be done in parallel**
  - Pivot criteria: Row with the largest element
  - Searching for the maximum is a data-parallel reduction

Gaussian Elimination: Data Mapping

- **Pivot selection requires a 1-D reduction**
  - Distribute rows ⇒ parallel, with communication
  - Distribute columns ⇒ sequential, but no communication
- **Element updates require the old value and elements from the pivot row and column**
  - Distribute rows ⇒ parallel, but broadcast the pivot row
  - Distribute columns ⇒ parallel, but broadcast the pivot column
- **Each stage works on a smaller contiguous region of the array**
  - `BLOCK` ⇒ processors drop out of the computation
  - `CYCLIC` ⇒ work stays (fairly) evenly distributed until the end
  - `CYCLIC(K)` ⇒ granularity will be at least K elements
- **The bottom line**
  - `(*,CYCLIC)` if broadcast > pivoting one column
  - `(CYCLIC,*)` if broadcast < one column, synchronous comm.
  - `(CYCLIC,CYCLIC)` if broadcast < one col., overlapped comm.

Gaussian Elimination: HPF Program

```
REAL a(n,n), tmp(n)
!HPF$ DISTRIBUTE a(CYCLIC,CYCLIC)
!HPF$ ALIGN tmp(i) WITH a(*,i)

DO k = 1, n-1

   ! Select the pivot
   ipivot = MAXLOC( ABS(a(k:n,k)) ) + k - 1

   ! Swap the rows
   tmp(k:n) = a(ipivot,k:n)
   a(ipivot,k:n) = a(k,k:n)
   a(k,k:n) = tmp(k:n)

   ! Update the submatrix
   FORALL ( i=k+1:n, j=k+1:n ) &
      & a(i,j) = a(i,j) - a(i,k)/tmp(k)*tmp(j)

END DO
```

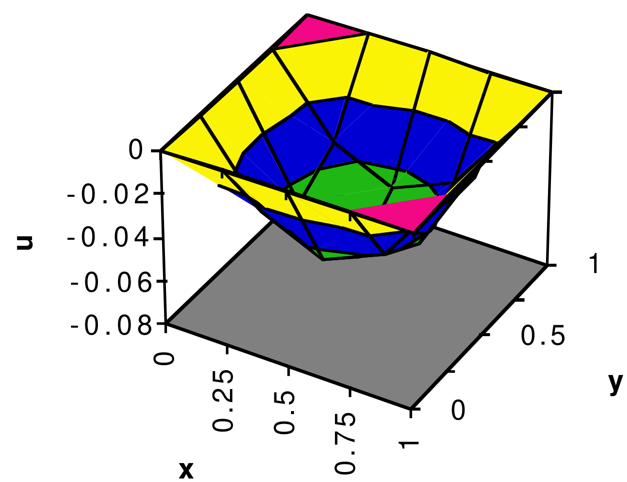
Jacobi Iteration: The Algorithm

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

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

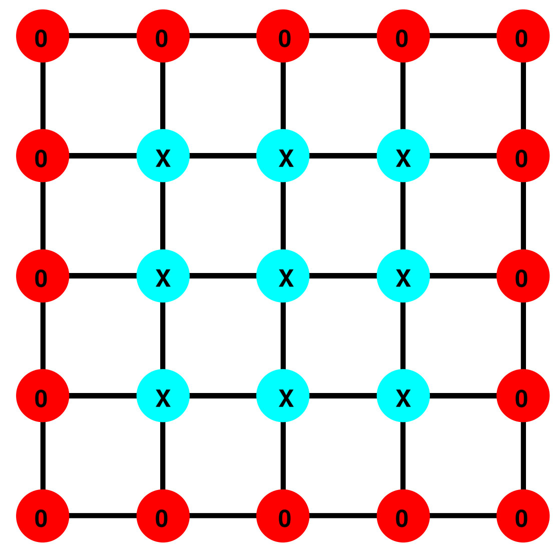
$$u = 0 \text{ if } x = 0, x = 1, y = 0, \text{ or } y = 1$$

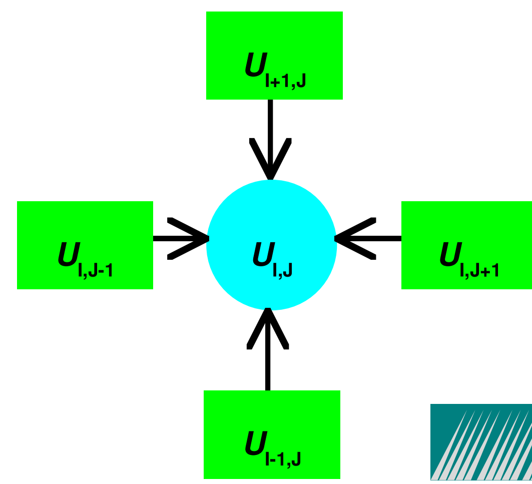
- **The Approach**
  - Divide (continuous) space into a (discrete) grid
  - Guess a solution on the grid
  - Update the solution at every grid point
  - Repeat update until solution doesn't change



Jacobi Iteration: Equations and Pictures

Discretized Equations	Reordered Equations
$4u_{1,1}-u_{2,1}-u_{1,2}=-0.00220$	$u_{1,1}=(u_{2,1}+u_{1,2}-0.00220)/4$
$4u_{1,2}-u_{2,2}-u_{1,1}-u_{1,3}=-0.00293$	$u_{1,2}=(u_{2,2}+u_{1,1}+u_{1,3}-0.00293)/4$
$4u_{1,3}-u_{2,3}-u_{1,2}=-0.00220$	$u_{1,3}=(u_{2,3}+u_{1,2}-0.00220)/4$
$4u_{2,1}-u_{1,1}-u_{3,1}-u_{2,2}=-0.00293$	$u_{2,1}=(u_{1,1}+u_{3,1}+u_{2,2}-0.00293)/4$
$4u_{2,2}-u_{1,2}-u_{3,2}-u_{2,1}-u_{2,3}=-0.00391$	$u_{2,2}=(u_{1,2}+u_{3,2}+u_{2,1}+u_{2,3}-0.00391)/4$
$4u_{2,3}-u_{1,3}-u_{3,3}-u_{2,2}=-0.00293$	$u_{2,3}=(u_{1,3}+u_{3,3}+u_{2,2}-0.00293)/4$
$4u_{3,1}-u_{2,1}-u_{3,2}=-0.00220$	$u_{3,1}=(u_{2,1}+u_{3,2}-0.00220)/4$
$4u_{3,2}-u_{2,2}-u_{3,1}-u_{3,3}=-0.00293$	$u_{3,2}=(u_{2,2}+u_{3,1}+u_{3,3}-0.00293)/4$
$4u_{3,3}-u_{2,3}-u_{3,2}=-0.00220$	$u_{3,3}=(u_{2,3}+u_{3,2}-0.00220)/4$





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 interation can be updated in parallel**
  - `unew(i,j)=(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)+f(i,j))/4`
  - These are independent because `unew` ≠ `u` and `unew` ≠ `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

CRPC

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(IHIGH+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

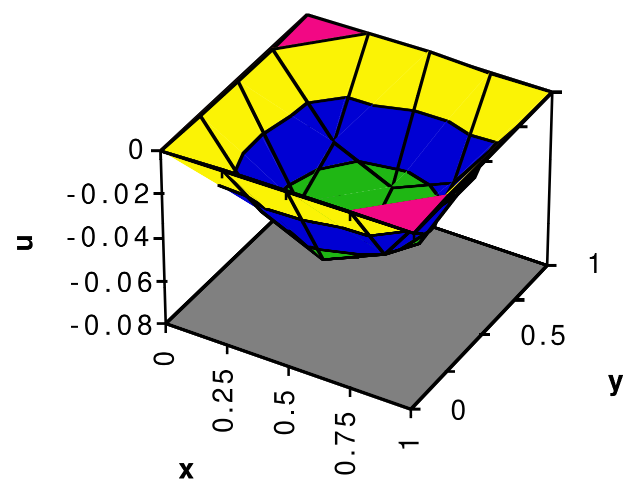
```
REAL u(0:nx,0:ny), unew(0:nx,0:ny), f(0:nx,0:ny)
!HPF$ DISTRIBUTE u(BLOCK,*)
!HPF$ ALIGN WITH u(:,:) :: unew(:,:), f(:,:)

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)
    FORALL ( 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

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

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



CRPC

Conjugate Gradient: Equations and Pictures

Discretized Equations

$4u_{1,1}-u_{2,1}-u_{1,2}=-0.00220$

$4u_{1,2}-u_{2,2}-u_{1,1}-u_{1,3}=-0.00293$

$4u_{1,3}-u_{2,3}-u_{1,2}=-0.00220$

$4u_{2,1}-u_{1,1}-u_{3,1}-u_{2,2}=-0.00293$

$4u_{2,2}-u_{1,2}-u_{3,2}-u_{2,1}-u_{2,3}=-0.00391$

$4u_{2,3}-u_{1,3}-u_{3,3}-u_{2,2}=-0.00293$

$4u_{3,1}-u_{2,1}-u_{3,2}=-0.00220$

$4u_{3,2}-u_{2,2}-u_{3,1}-u_{3,3}=-0.00293$

$4u_{3,3}-u_{2,3}-u_{3,2}=-0.00220$

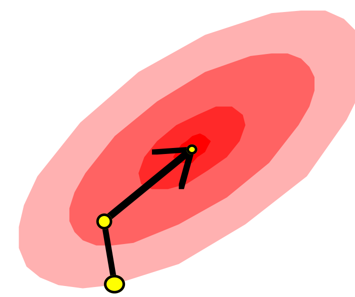
Minimize

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

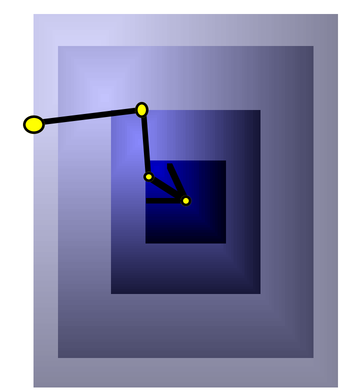
$$A=\begin{bmatrix}4&1&0&1&0&0&0&0&0\\1&4&1&0&1&0&0&0&0\\0&1&4&0&0&1&0&0&0\\1&0&0&4&1&0&1&0&0\\0&1&0&1&4&1&0&1&0\\0&0&1&0&1&4&0&0&1\\0&0&0&1&0&0&4&1&0\\0&0&0&0&1&0&1&4&1\\0&0&0&0&0&1&0&1&4\end{bmatrix}$$

$$f=\begin{bmatrix}-.00220\\-.00293\\-.00220\\-.00293\\-.00391\\-.00293\\-.00220\\-.00293\\-.00220\end{bmatrix}$$

2-Dimensional Search



3-Dimensional Search



9-Dimensional Search

???

CRPC

Conjugate Gradient: More Equations

```
u = ⟨ initial guess ⟩
r = f – A ∗ u
δ = max( |r| )
ι = 0; ρ = 0
WHILE (δ > ε) DO
    ι = ι + 1; ρ_old = ρ
    ρ = r · r
    IF  (ι=1)  THEN  p = r   ELSE  p = r + ρ/ρ_old p
    q = A ∗ p
    α = ρ / (p · q)
    u = u + α p
    r = r – α q
    δ = max( |r| )
END WHILE
```

CRPC

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*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/rho_old * p(i,j)`
  - These are classic data-parallel operations
  - $\mathtt{rho} = \sum_{i,j} \mathtt{r(i,j)}^2$
  - 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**
  - `BLOCK` ⇒ least communication volume
  - `(BLOCK,*)` ⇒ move `u(i-1,j)`, `u(i+1,j)` ∀ `j`
  - `(BLOCK,BLOCK)` ⇒ move `u(ILOW-1,j)`, `u(IHIGH+1,j)` ∀ `j`, `u(i,JLOW-1)`, `u(i,JHIGH+1)` ∀ `i`
- **Computation is static, homogenous, and over full array**
  - No load balancing issues
- **The bottom line**
  - `(BLOCK,*)` or `(*,BLOCK)` on high-latency machines or small problem sizes
  - `(BLOCK,BLOCK)` on low-latency machines

Conjugate Gradient: HPF Program

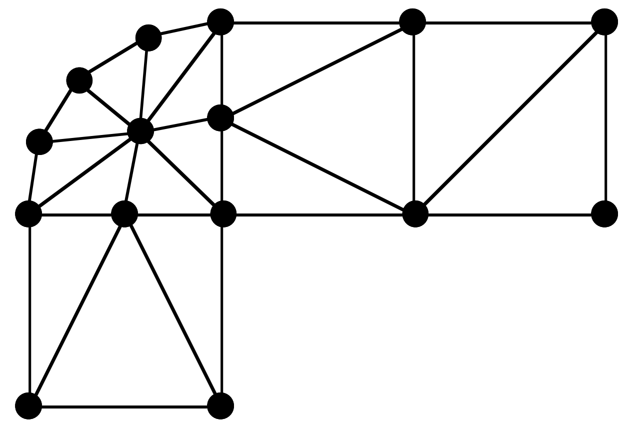
```
REAL u(0:n,0:n), r(0:n,0:n), p(0:n,0:n)
REAL q(0:n,0:n), f(0:n,0:n)
!HPF$ DISTRIBUTE u(BLOCK,*)
!HPF$ ALIGN WITH u(:,:) :: r(:,:), p(:,:)
!HPF$ ALIGN WITH u(:,:) :: q(:,:), f(:,:)
INTERFACE
    SUBROUTINE a_times_vector( x, y )
    REAL, INTENT(IN) :: x(:,:)
    REAL, INTENT(OUT) :: y(:,:)
    !HPF$ DISTRIBUTE 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
```

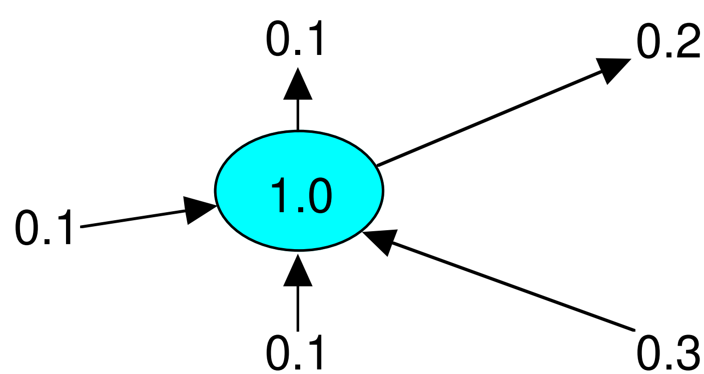
```
DO WHILE (err > tol)
    i = i + 1; rho_old = rho
    rho = SUM( r(1:n-1,1:n-1)**2 )
    IF  (i=1)  THEN
        p = r
    ELSE
        p = r + rho/rho_old * p
    END IF
    CALL a_times_vector(p, q)
    alpha = rho / SUM(p*q)
    u = u + alpha * p
    r = r - alpha * q
    err = MAXVAL(ABS(r(1:n-1,1:n-1)))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

CRPC

Irregular Mesh Relaxation: The Algorithm

- **The Problem**
  - Given an irregular mesh of values
  - Update each value using its neighbors in the mesh
- **The Approach**
  - Store the mesh as a list of edges
  - Process all edges in parallel





Irregular Mesh: Parallelism

- **Each iteration of the relaxation uses all the data computed in the previous step, and the edge array**
  - No parallelism at this level
  - (Does this sound familiar?)
- **All edge values can be computed in parallel**
  - `flux(i)=(x(iedge(i,1))-x(iedge(i,2)))/2`
  - These are independent because `edge_val` ≠ `node_val`
- **Updating the node values is *not trivially* parallel**
  - `x(iedge(i,1)) = x(iedge(i,1)) - flux(i)`
  - `x(iedge(i,2)) = x(iedge(i,2)) + flux(i)`
  - They are dependent because sometimes `iedge(`i_Y`,1)` = `iedge(`i_Z`,2)`
  - Fortunately, HPF provides the `XXX_SCATTER` functions

CRPC

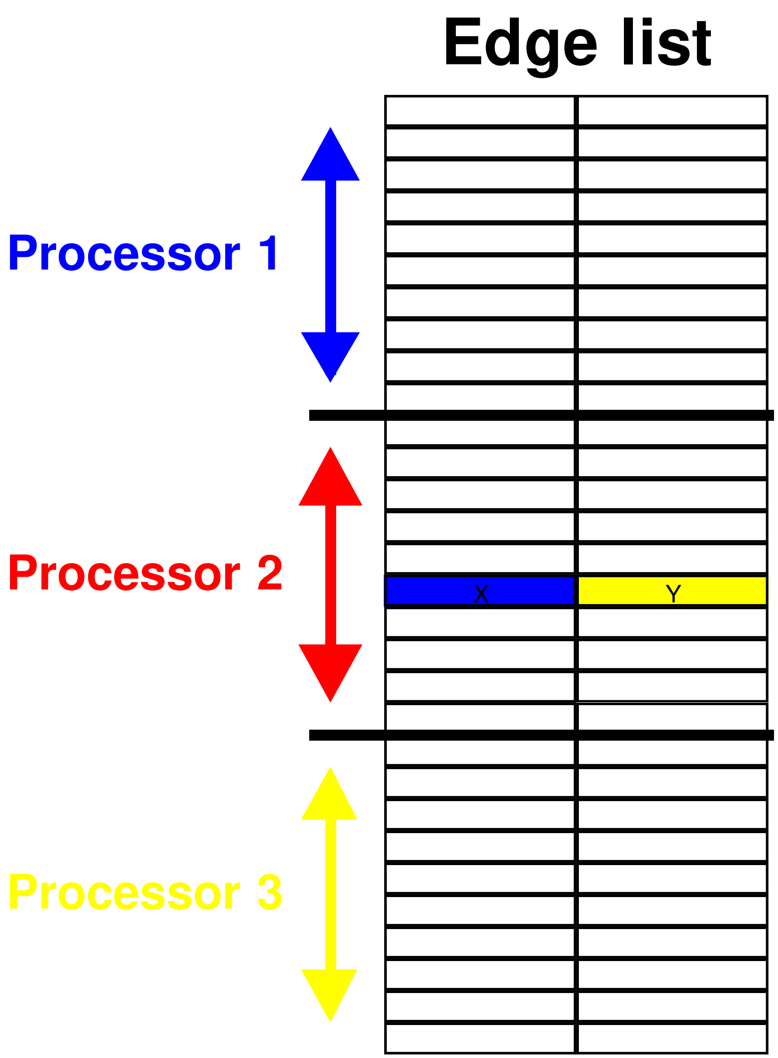
Irregular Mesh: Data Mapping I

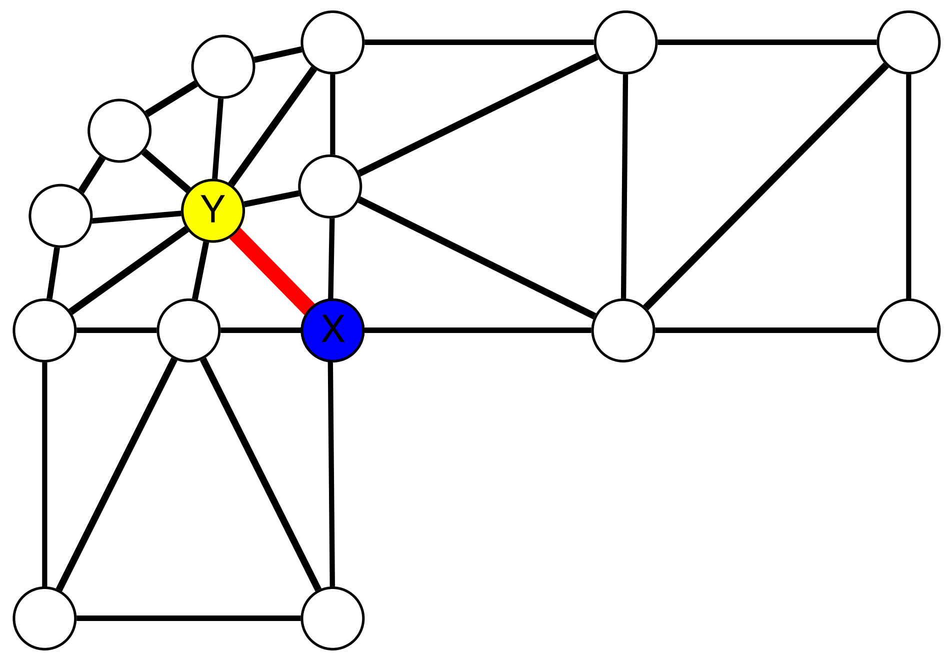
- **Warning: Your compiler may do things differently!**
- **Computing edge values requires edge list and node values**
  - Distribute edges ⇒ parallel, no communication for edges
  - Replicate edges ⇒ sequential, broadcast edge values
  - Distribute nodes ⇒ move "shared" endpoints
  - Replicate nodes ⇒ no movement for endpoints
- **Updating node values requires edge list, edge values, and node values**
  - Distribute edges ⇒ parallel, no communication for edges
  - Replicate edges ⇒ sequential, no communication for edges
  - Distribute nodes ⇒ move "shared" endpoints
  - Replicate nodes ⇒ move all endpoints
- **The bottom line, part I**
  - Always distribute edges
  - Distribute nodes unless the problem is very small

Irregular Mesh: Data Mapping II

- **Warning: Your compiler may do things differently!**
- **Computation is static, heterogeneous, and over full array (with respect to the edges)**
  - No load balancing issues
- **Accesses to node array are "nearest neighbor" in the mesh**
  - This is not reflected in the index order!
  - ∴ This does not favor either `BLOCK` or `CYCLIC`
- **To minimize communication, edge and node distributions must fit the mesh topology**
  - HPF's regular distributions are not ideal for this
  - Future indirect distributions may be better, but require careful construction
- **The bottom line, part II**
  - No silver bullet
  - Order the nodes and edges to bring "close" entities together, then use `BLOCK`

Irregular Mesh: Pictures





Irregular Mesh: Pictures

Edge list

Processor 1

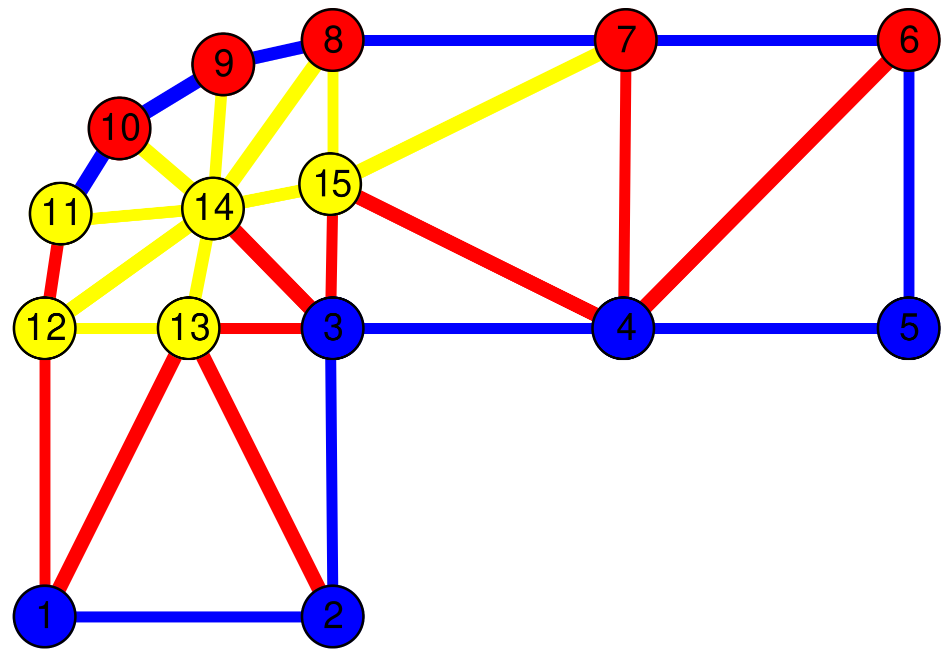
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10
10	11

Processor 2

11	12
12	1
1	13
2	13
3	13
3	14
3	15
4	6
4	7
4	15

Processor 3

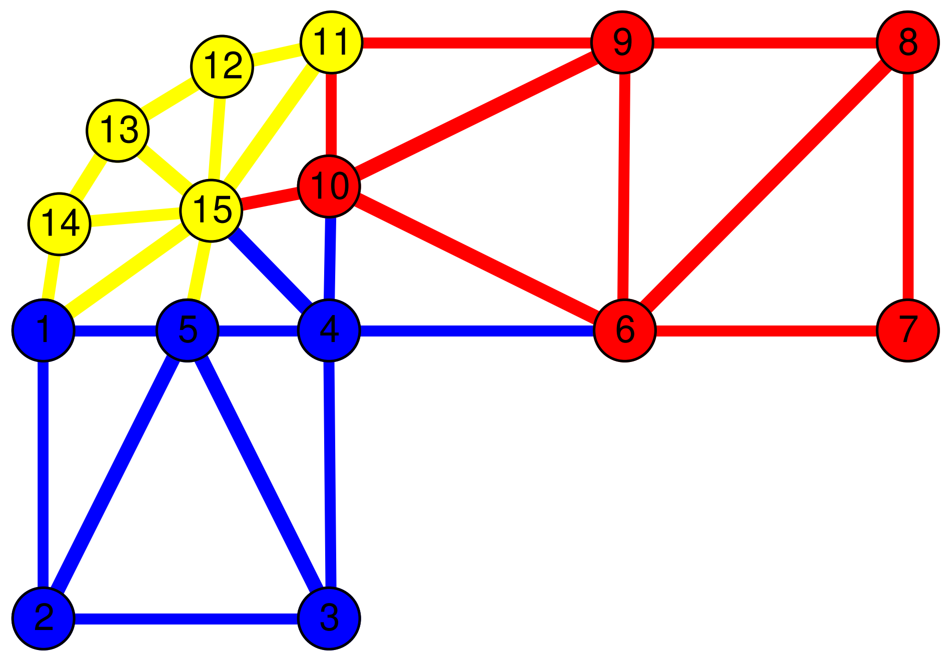
7	15
8	14
8	15
9	14
10	14
11	14
12	13
12	14
13	14
14	15



Irregular Mesh: Pictures

Edge list

Processor		
Processor 1	1	2
	1	5
	2	3
	2	5
	3	4
	3	5
	4	5
	4	6
	4	10
	4	15
Processor 2	6	7
	6	8
	6	9
	6	10
	7	8
	8	9
	9	10
	9	11
	10	11
	10	15
Processor 3	1	14
	1	15
	5	15
	11	12
	11	15
	12	13
	12	15
	13	14
	13	15
	14	15



Irregular Mesh: HPF Program

```
USE HPF_LIBRARY
REAL x(nnode), flux(nedge)
INTEGER iedge(nedge,2)
INTEGER permute_node(nnode), permute_edge(nedge)
!HPF$ DISTRIBUTE x(BLOCK)
!HPF$ DISTRIBUTE flux(BLOCK)
!HPF$ ALIGN iedge(i,*) WITH flux(i)
!HPF$ ALIGN permute_edge(i) WITH flux(i)
!HPF$ ALIGN permute_node(i) WITH x(i)
CALL renumber_nodes( iedge, permute_node )
x( permute_node(:) ) = x
FORALL (i=1:nedge) iedge(i,:) = permute_node(iedge(i,:))
permute_edge = GRADE_UP( iedge(:,1) )
FORALL (i=1:nedge) iedge(i,:) = iedge(permute_edge(i),:)
err = tol * 1e6
DO WHILE (err > tol)
   flux=(x(iedge(1:nedge,1))-x(iedge(1:nedge,2)))/2
   x=SUM_SCATTER(-flux(1:nedge),x,iedge(1:nedge,1))
   x=SUM_SCATTER( flux(1:nedge),x,iedge(1:nedge,2))
   err = SUM( flux*flux**2 ) / nedge
END DO
```