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**
  - Jacobi Iteration
  - Gaussian Elimination
  - 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

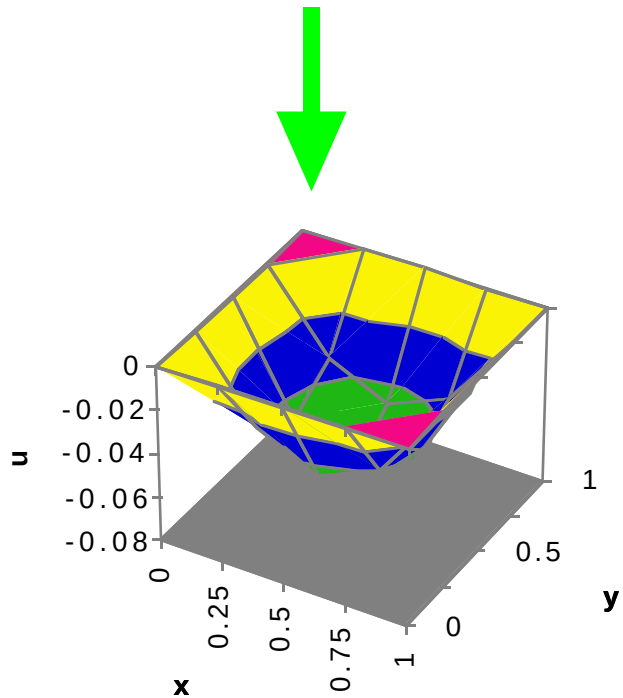
T_{comm} = Communication / synchronization time

Jacobi Iteration: 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
  - Update the solution at every grid point
  - Repeat update 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

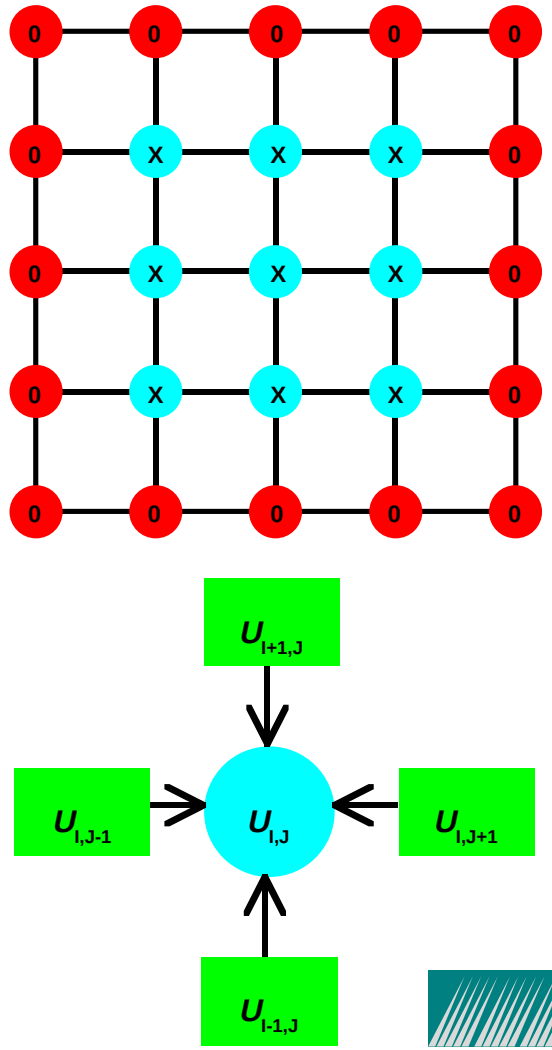
Jacobi Iteration: Equations and Pictures

Discretized Equations

$$4u_{1,1}-u_{2,1}-u_{1,2}=-0.00220$$
$$4u_{1,2}-u_{2,2}-u_{1,1}-u_{1,3}=-0.00293$$
$$4u_{1,3}-u_{2,3}-u_{1,2}=-0.00220$$
$$4u_{2,1}-u_{1,1}-u_{3,1}-u_{2,2}=-0.00293$$
$$4u_{2,2}-u_{1,2}-u_{3,2}-u_{2,1}-u_{2,3}=-0.00391$$
$$4u_{2,3}-u_{1,3}-u_{3,3}-u_{2,2}=-0.00293$$
$$4u_{3,1}-u_{2,1}-u_{3,2}=-0.00220$$
$$4u_{3,2}-u_{2,2}-u_{3,1}-u_{3,3}=-0.00293$$
$$4u_{3,3}-u_{2,3}-u_{3,2}=-0.00220$$

Reordered Equations

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



Jacobi Iteration: Parallelism

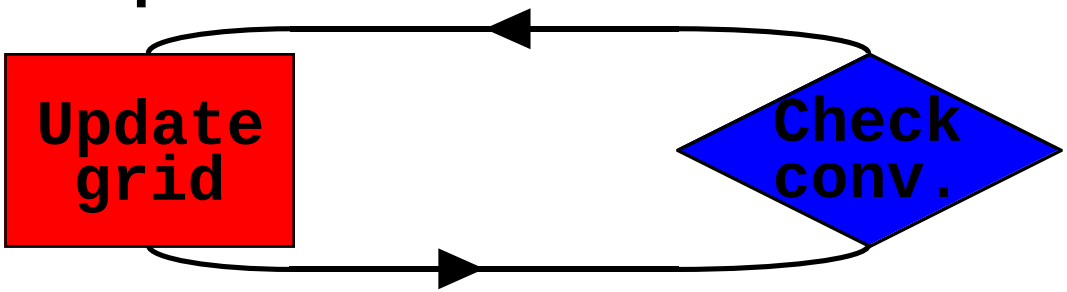
- **Each Jacobi iteration uses all the data computed in the previous step**
  - No parallelism at this level
  - (We won't try other iterative schemes to avoid this)
- **All updated elements within an iteration can be updated in parallel**
  - `unew(i,j)=(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)+f(i,j))/4`
  - 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

Jacobi Iteration: Data Mapping

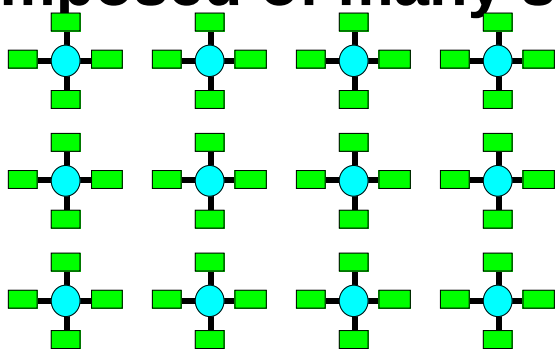
- **Convergence test requires a whole-array reduction**
  - Any distribution ⇒ parallel, with communication
- **Element updates require local value and nearest neighbors**
  - **`BLOCK`** ⇒ least communication volume
  - **`CYCLIC`** ⇒ communicate entire array
  - **`(BLOCK,*)`** ⇒ move **`u(i-1,j)`**, **`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

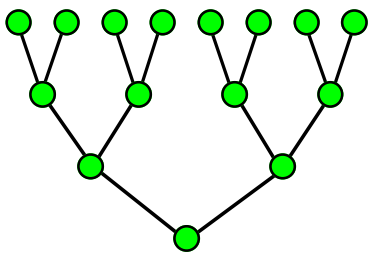
Jacobi Iteration: Partitioning

- At the most abstract level, Jacobi iteration is a sequential process.



- However, each step in the process is itself composed of many smaller operations.





- Conclusion: Jacobi iteration has plenty of data parallelism

Jacobi Iteration: Communication

- **Element updates:**
  - Each requires 4 values from previous update step
  - Static, local communication
    - Generally, the second-fastest kind (after no communication)
- **Convergence test:**
  - Uses all values from latest update step
  - Static, global communication
    - Reduction operation
    - Efficient methods known, encapsulated in libraries

Jacobi Iteration: Agglomeration

- **Element updates require nearest neighbors**
  - **`CYCLIC`** ⇒ communicate entire array
  - **`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`**
- **Convergence test requires a whole-array reduction**
  - Any distribution ⇒ static, structured communication
- **The bottom line**
  - **`(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
```

Jacobi Iteration: Mapping

- **This program is a piece of cake for the compiler.**
  - Allocate portion of array on each processor based on **`DISTRIBUTE`**
  - Apply owner-computes rule analytically based on left-hand side
  - Detect shift communication from dependence analysis of subscripts or pattern matching
  - Recognize **`MAXVAL`** intrinsic as reduction communication
  - Place all communication directly outside parallel construct where it occurs
  - Number processors so that shifts do not cause contention

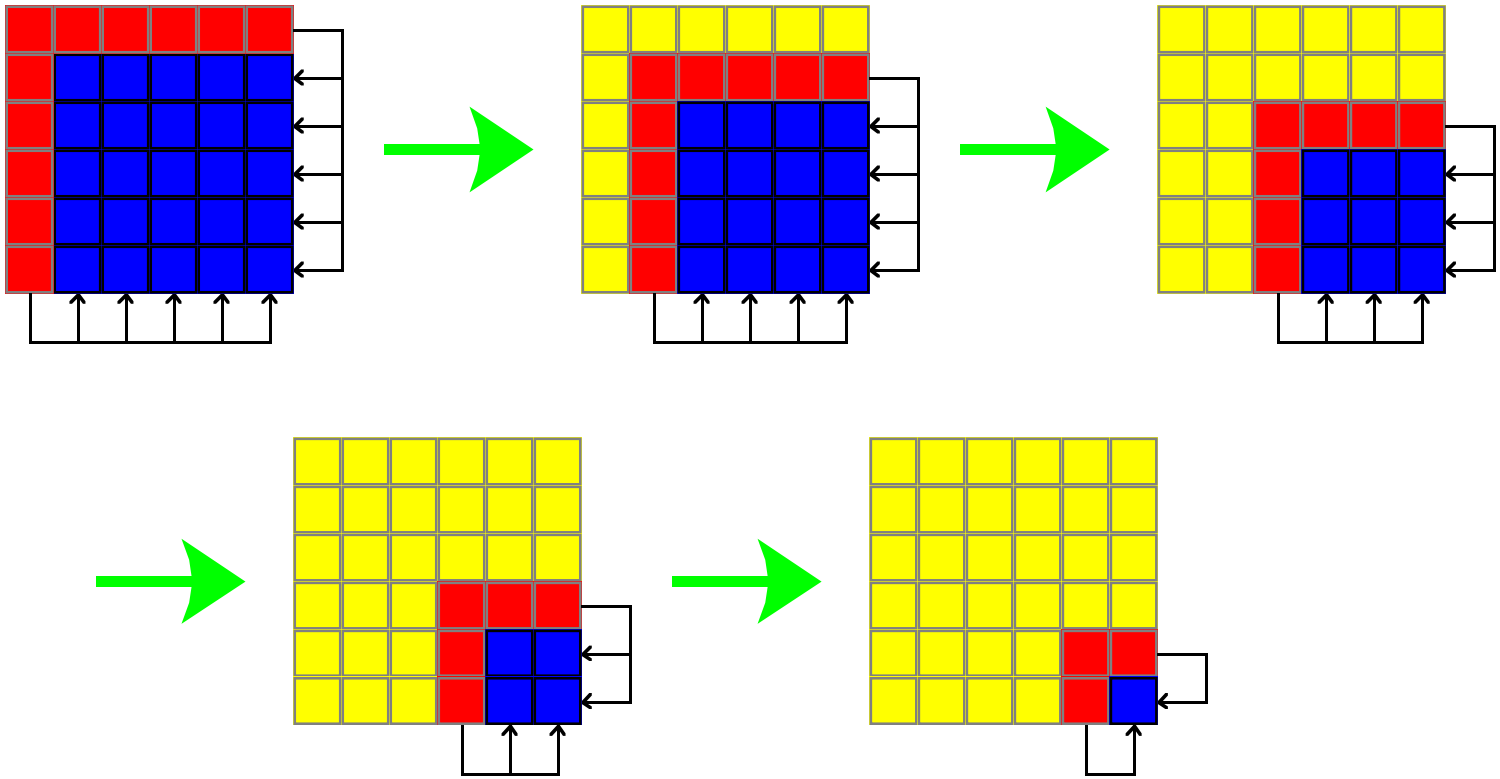
Gaussian Elimination: The Algorithm

- **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

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

$$x_1 = 1.00$$
$$x_2 = 1.00$$
$$x_3 = 1.00$$
$$x_4 = 1.00$$

Gaussian Elimination: Pictures



Not Used Pivot Elements Updated Elements

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 K`**, **`j 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: Partitioning

- **At the most abstract level, Gaussian elimination is a sequential process.**
  - Need all elements in column **k** to find pivot row
  - Need all elements in column **k** and pivot row to perform pivoting
- **However, each step in the process is itself composed of many smaller operations.**
  - Perform all element updates independently
- **Conclusion: Gaussian elimination has plenty of data parallelism**
  - Until the last few stages, anyway…

Gaussian Elimination: Communication

- **Pivot search:**
  - Reduction along column
  - Static, global communication
- **Element updates:**
  - Each requires itself, elements from pivot column & row
  - Static, global communication
    - Broadcast

Gaussian Elimination: Agglomeration

- **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
- **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
```

Gaussian Elimination: Mapping

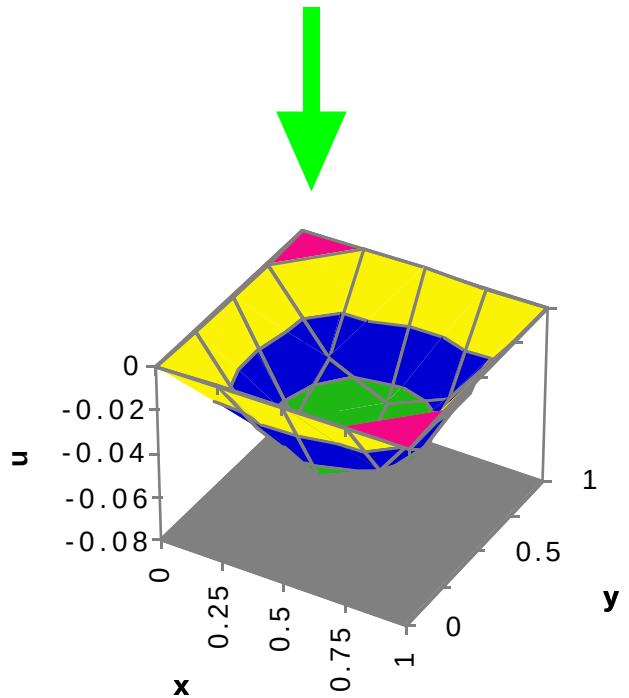
- **This program is harder for the compiler.**
  - Allocate portion of array on each processor based on **`DISTRIBUTE`**
  - Apply owner-computes rule analytically based on left-hand side
  - Detect communication from dependence analysis & intrinsics
  - Here, it really pays to transform the program!
    - Reorder computation to always precompute the next pivot column
    - Rearrange communication to pipeline the series of updates
    - Do broadcasts asynchronously
    - Net result: 2× speedup
  - Use standard numbering for processors

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

$$4u_{1,1}-u_{2,1}-u_{1,2}=-0.00220$$
$$4u_{1,2}-u_{2,2}-u_{1,1}-u_{1,3}=-0.00293$$
$$4u_{1,3}-u_{2,3}-u_{1,2}=-0.00220$$
$$4u_{2,1}-u_{1,1}-u_{3,1}-u_{2,2}=-0.00293$$
$$4u_{2,2}-u_{1,2}-u_{3,2}-u_{2,1}-u_{2,3}=-0.00391$$
$$4u_{2,3}-u_{1,3}-u_{3,3}-u_{2,2}=-0.00293$$
$$4u_{3,1}-u_{2,1}-u_{3,2}=-0.00220$$
$$4u_{3,2}-u_{2,2}-u_{3,1}-u_{3,3}=-0.00293$$
$$4u_{3,3}-u_{2,3}-u_{3,2}=-0.00220$$

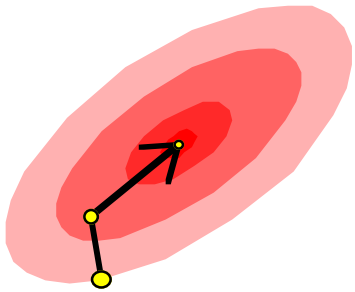
Minimize

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

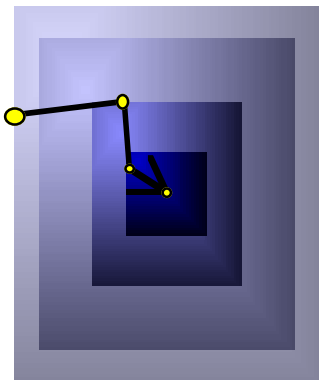
$$A=\begin{bmatrix}4&1&0&1&0&0&0&0&0\\1&4&1&0&1&0&0&0&0\\0&1&4&0&0&1&0&0&0\\1&0&0&4&1&0&1&0&0\\0&1&0&1&4&1&0&1&0\\0&0&1&0&1&4&0&0&1\\0&0&0&1&0&0&4&1&0\\0&0&0&0&1&0&1&4&1\\0&0&0&0&0&1&0&1&4\end{bmatrix}$$

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

2-Dimensional Search



3-Dimensional Search



9-Dimensional Search

???

Conjugate Gradient: More Equations

u = ⟨ initial guess ⟩
$r = f - A * u$
δ = max(|r|)
$\iota = 0$; $\rho = 0$
WHILE ($\delta > \varepsilon$) DO
    $\iota = \iota + 1$; $\rho_{old} = \rho$
    $\rho = r \cdot r$
    IF (ι=1) THEN $p = r$ ELSE $p = r + \rho/\rho_{old}\ p$
    $q = A * p$
    $\alpha = \rho / (p \cdot q)$
    $u = u + \alpha\ p$
    $r = r - \alpha\ q$
    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
  - **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**
  - **`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: Partitioning

- **At the most abstract level, conjugate gradient has minimal parallelism**
  - ***u***, ***r***, α can be updated independently
- **The real potential parallelism is in the matrix and vector operations, however.**
  - $r \cdot r$ is a reduction of size N
  - $u + \alpha p$ is a vector update of size N
  - $A * p$ is a (sparse) matrix-vector multiply, in this case of size O(N)
    - It looks a lot like the operator in Jacobi
- **Conclusions:**
  - Stick to the matrix/vector operators
  - Task parallelism (and pipelining) may improve some

Conjugate Gradient: Communication

- **Convergence test:**
  - Global reduction
- **Dot products:**
  - Global reductions
- **Vector updates:**
  - Elementwise scaling and addition of two vectors
  - No communication if vectors are aligned
- **Matrix-vector multiplies:**
  - Depends on the matrix (PDE operator) in the problem
  - For the model problem, equivalent to Jacobi iteration grid update
  - Static, local communication

Conjugate Gradient: Agglomeration

- **Dot products and convergence test always require global communication**
  - No reason to pick one **`DISTRIBUTE`** over another
- **Vector updates require no communication**
  - *Really* no reason to choose a particular **`DISTRIBUTE`**
- **Matrix-vector multiply *does* care where its data come from**
  - In this case, same advantages/disadvantages as Jacobi iteration
- **The bottom line**
  - **`(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, 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
```

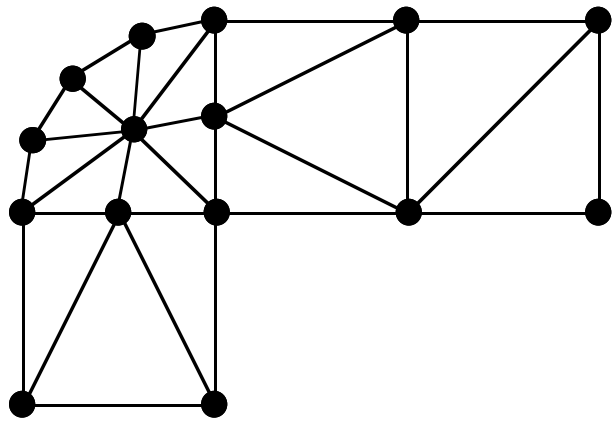
```
DO WHILE (err > tol)
     i = i + 1; rho_old = rho
     rho = SUM( r(1:n-1,1:n-1)**2 )
     IF  (i=1)  THEN
          p = r
     ELSE
          p = r + rho/rho_old * p
     END IF
     CALL a_times_vector(p, q)
     alpha = rho / SUM(p*q)
     u = u + alpha * p
     r = r – alpha * q
     err = MAXVAL(ABS(r(1:n-1,1:n-1)))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

Conjugate Gradient: Mapping

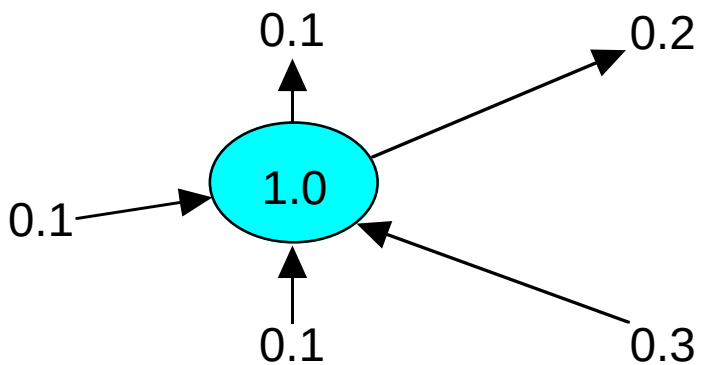
- **This program looks like a more complicated version of Jacobi iteration to the compiler.**
  - Allocate arrays based on **`DISTRIBUTE`**
  - Apply owner-computes rule
  - Detect communication from dependence analysis and intrinsics
  - Useful optimizations include aggregating communication, overlapping communication with computation
  - All of this becomes more *interesting* if the program encapsulates operations in a subroutine
    - The programmer must trade off efficiency for flexibility and maintainability

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 paralle
    - Compute contribution of edge
    - Add to one endpoint, subtract from the otherl



Irregular Mesh: Sequential Program

```
REAL x(nnode), flux(nedge)
INTEGER iedge(nedge,2)
err = tol * 1e6
DO WHILE (err > tol)
    DO i = 1, nedge
       flux(i) = (x(iedge(i,1))-x(iedge(i,2))) / 2
       err = err + flux(i)*flux(i)
    END DO
    DO i = 1, nedge
       x(iedge(i,1)) = x(iedge(i,1)) - flux(i)
       x(iedge(i,2)) = x(iedge(i,2)) + flux(i)
    END DO
    err = err / nedge
END DO
```

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 **`SUM_SCATTER`** function

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 *or* 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, homogeneous, 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
  - HPF 2.0 indirect distributions are 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: Partitioning & Communication

- **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?)
- **Instead, use the data-parallel edge and node updates**
  - `flux(i)=(x(iedge(i,1))-x(iedge(i,2)))/2`
    - Independent because `edge_val` `node_val`
  - `x(iedge(i,1)) = x(iedge(i,1)) - flux(i);`
    `x(iedge(i,2)) = x(iedge(i,2)) + flux(i)`
    - Not independent because sometimes $\texttt{iedge}(i_y,1) = \texttt{iedge}(i_x,2)$
    - Fortunately, HPF provides the **SUM_SCATTER** function
- **Communication needed in both stages**
  - Between edges and nodes to compute **flux**
  - Edge-node and node-node to compute **x**
  - All communication is static, local *with respect to grid*, but unstructured *with respect to array indices*

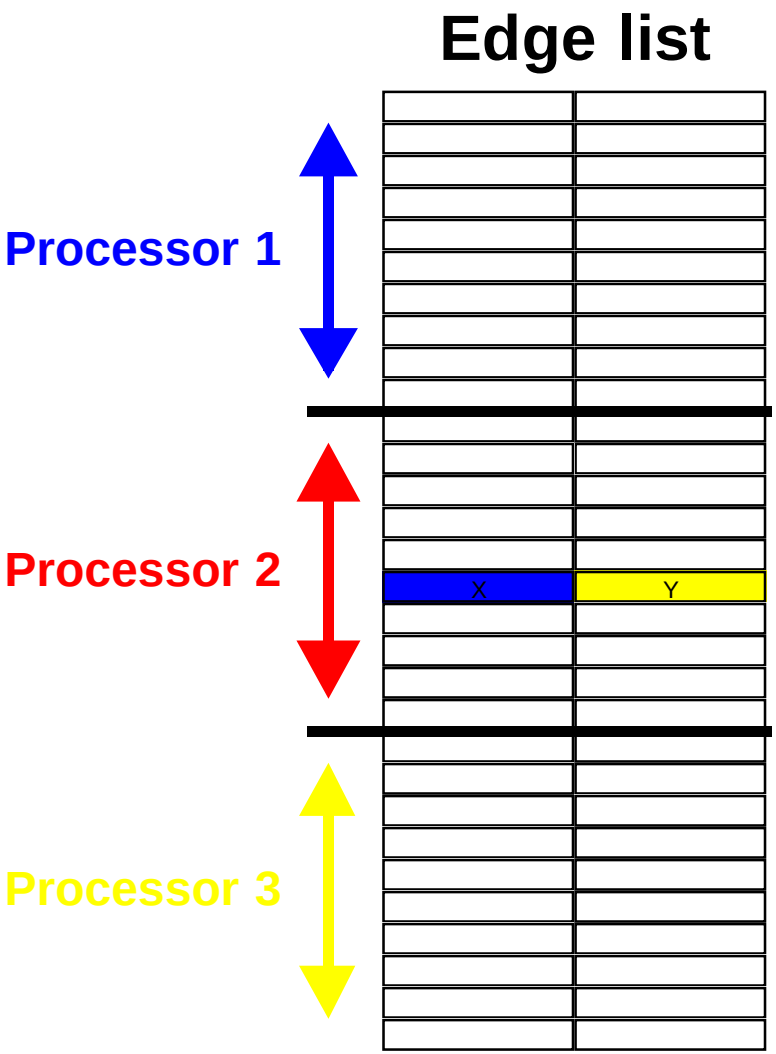
Irregular Mesh: Agglomeration I

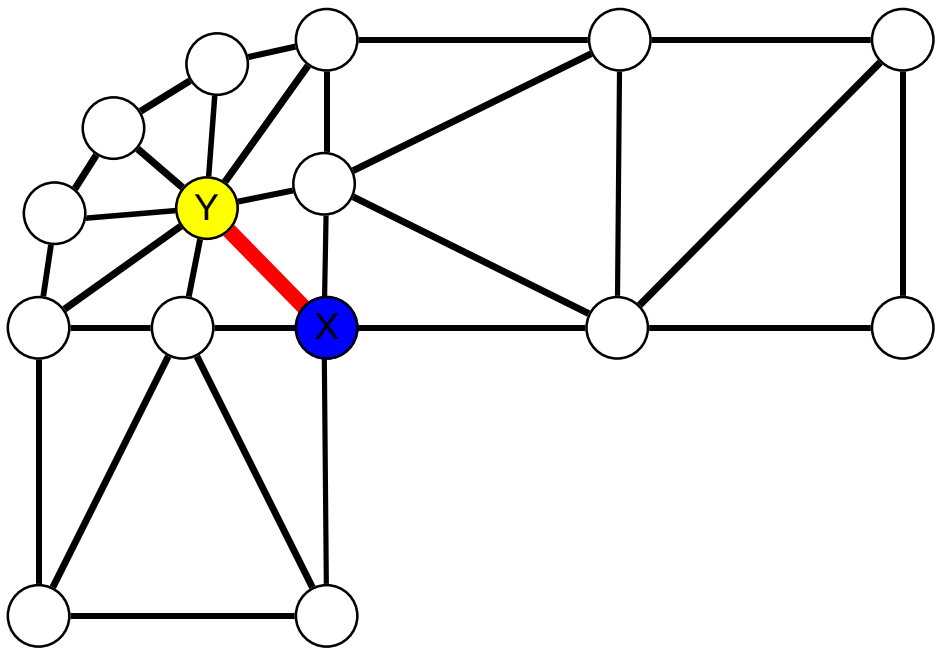
- **Warning: Your compiler may do things differently!**
- **Computing edge values requires edge list and node values**
  - Distribute edges ⇒ parallel, no communication for edges
  - Replicate edges ⇒ sequential *or* broadcast edge values
  - Distribute nodes ⇒ move “shared” endpoints
  - Replicate nodes ⇒ 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: Agglomeration II

- **Warning: Your compiler may do things differently!**
- **Computation is static, homogeneous, 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
  - HPF 2.0 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: Bad Data Distribution

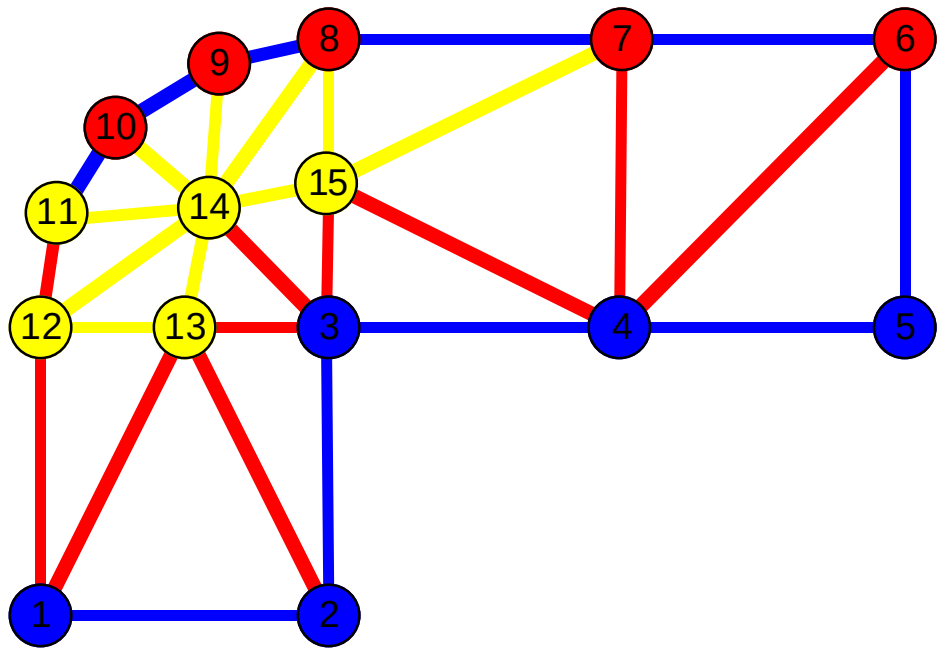
Edge list

Processor 1

Processor 2

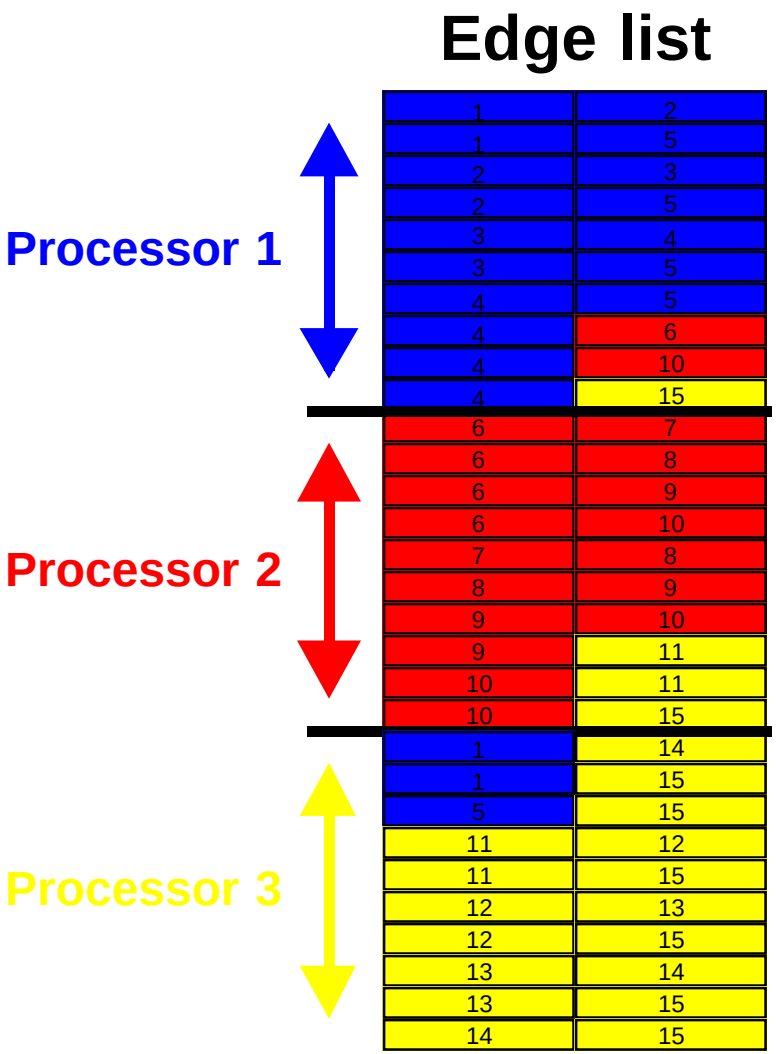
Processor 3

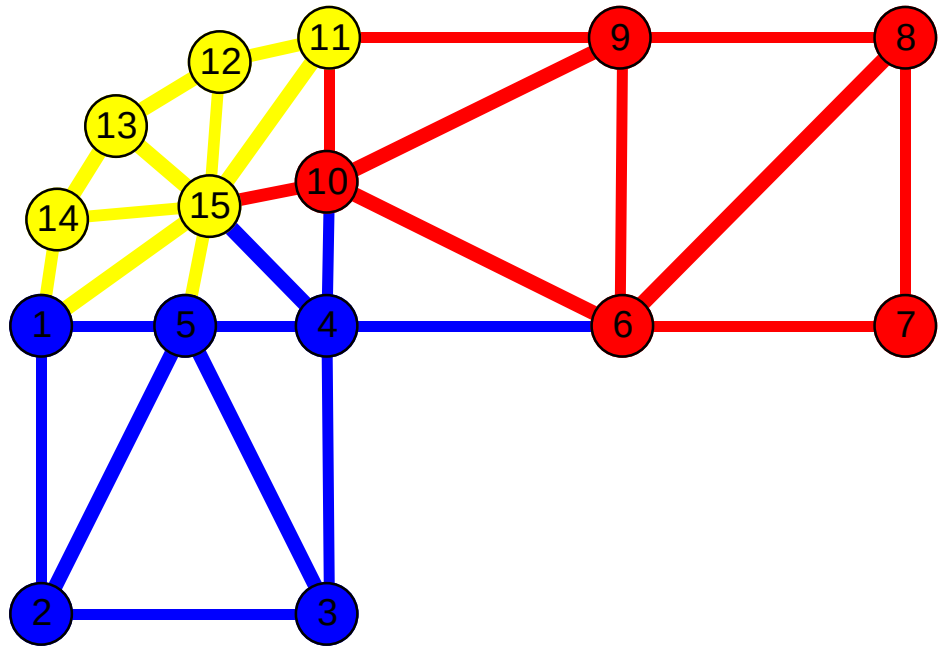
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10
10	11
11	12
12	1
1	13
2	13
3	13
3	14
3	15
4	6
4	7
4	15
7	15
8	14
8	15
9	14
10	14
11	14
12	13
12	14
13	14
14	15



CRPC

Irregular Mesh: Good Data Distribution





CRPC

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 ) / nedge
END DO
```

Irregular Mesh: Mapping

- **This program is going to be *challenging*…**
  - Indexing of arrays will be difficult
  - Owner-computes rule difficult to apply
  - Key technique: inspector-executor communication
    - First time the code is executed, generate a table of required communication at run time (*inspector*)
      - Problem: How big does that table get?
      - Problem: How do you efficiently distribute that table to all processors?
    - Use this table to manage unstructured communication until the communication pattern changes (*executor*)
      - Problem: How do you know the pattern has changed?
  - Commercial compilers are attacking these problems, but there is a long way to go