Data-parallel languages

George Bosilca
bosilca@eecs.utk.edu

Data-parallelism

• Abstract, machine independent model of parallelism.
  – Fine-grain parallel operations, such as element-wise operations on array
  – Shared data in large, global arrays with mapping "hints"
  – Implicit synchronization between operations
  – Partially explicit communication from operation definitions
• Advantages:
  – Global operations conceptually simple
  – Easy to program
• Disadvantages
  – Difficult to find "perfect" compilers

Properties of parallelism

• Determinacy: is the behavior of a program repeatable?
• Compositionality: can independently created subprograms be combined?
• Expressiveness: can all sort of parallelism be expressed?
• Implementability: can a compiler generate efficient code for a variety of architectures?
Matrix Multiply (easy)

- Simple sequential algorithm.

\[ C = A \times B \Rightarrow C_{i,j} = \sum_k A_{i,k} B_{k,j} \]

All multiplications can be done in parallel!!

Parallelizing Compilers

- After 20 years of intensive research:
  - Limited success in parallelism detection
    - Instruction-level parallelism at the basic-block level can be detected
    - Parallelism in nested-loops with arrays with simple index expression can be discovered
    - Analysis technique such as data dependence analysis, pointer analysis, flow sensitive analysis, abstract interpretation, still can not be applied across procedure boundaries

- Result: instead of training compiler to parallelize code, people have been trained to write parallel algorithms.
Data parallel Programming model

- All data structures distributed across a grid of virtual processors.
- The owner processor computes the data elements assigned to it.
- Global communication primitives allow processors to exchange data.
- Implicit global barrier after each communication
- All processors execute the same program!

Data parallel Languages

- Work distributed between
  - The programmer (concentrate on solution):
    - High level structure and concept
    - Aggregate operations on large data structures
    - Data in global array with mapping information
  - The compiler (map conceptual “massive” parallelism to the physical “finite” machine):
    - Complete the details
    - Distribute data guided by the user hints
    - Optimize computations and communications

HPF details

- Analysis
  - Traditional dataflow and dependence analysis
  - Data mapping analysis
- Computation partitioning
  - Use data mapping to create locality
  - Transform the code to enhance this locality
- Communication
  - Move data if data mapping and computation partitioning don’t agree
  - Minimize/package communications (!)
- Code generation
  - All others optimizations
What compilation means for programmers

- Help analysis with assertions
  - ALIGN and DISTRIBUTE
  - INDEPENDENT
- Distribute arrays dimensions that exhibit parallelism
  - Conflicts require complex compilers, REDISTRIBUTE or new algorithm
- Consider communication patterns
  - BLOCK generally good for local stencils and fully-filled arrays
  - CYCLIC and CYCLIC(k) generally good for load-balancing and triangular loops

High Performance Fortran

- Defined by the High Performance Fortran Forum (HPFF) as a portable language for data-parallel computation
- History:
  - Proposed at SuperComputing 91 and HPFF Kickoff Meeting
  - Final draft of HPF, version 1.0, June 1993
  - New meeting 1994 & 1995 to make corrections, define further requirements.
- Influences:
  - Industry: C*, MasPar Fortran, Connection Machine Fortran
  - Academia: Fortran D, Vienna Fortran, ADAPT

HPF Features

- Data-parallel oriented
- Based on Fortran 90
- It’s more a compiler specification
- Few language extensions (not anymore)
  - FORALL and PURE
- Compiler directives:
  - INDEPENDENT, ALIGN, DISTRIBUTE
- Data alignment and Distribution left to the compiler
- Miscellaneous Support Operations (HPF library)
- Nothing about:
  - I/O, Explicit message passing, Irregular applications, Non-data parallelism
HPF: few words about performance

- Highly dependent on the compiler and the nature of the code
- Close to MPI performance FOR regular problems with simple subscript expression (at least some compilers).
- Research continue (?) on task-parallelism and irregular problems.
  - Branch-and-bound (tree algorithm)
  - Subscript : array dependent ( a(index(1,1) )

HPF: Data mapping

- THE feature of HPF
- Everything is done using the compiler directive HPF$
- User hints at 2 levels:
  - Distribute the data on the processors (DISTRIBUTE)
  - Create relationship between arrays (ALIGN)

HPF: Data mapping

- Goals:
  - Create locality: a processor should have direct access to all data it needs.
  - Avoiding contention: Data which are written in parallel should reside on different processors.
**HPF: PROCESSORS**

- Abstract a processors arrangement
- Declaration of virtual processors
  - The compiler will map these virtual processors to the physical one.
- `!HPFS PROCESSORS` line(4) square(2,2)

```
  p1  p2  p3  p4
  p5  p6
```

**HPF: DISTRIBUTE**

- Declare the data distribution on the processor arrangement.
- `!HPFS DISTRIBUTE [array] [how-to] [ONTO proc]`
- `!HPFS DISTRIBUTE [how-to] [ONTO proc] :: [array]`
- The distribution should be done on each dimension of the array
  - `BLOCK`, `CYCLIC`, `CYCLOC(k)`, *
- `ONTO` specifies a processor arrangement.

**HPF: DISTRIBUTE**

- How-to distribute the data:
- `BLOCK[size]`
  - Equal size blocks of consecutive elements distributed among processors
    - `N/p` on each processor
    - Maybe less on the last one
  - No wrap around (i.e. size >= `N/p`)
HPF: DISTRIBUTE

• BLOCK

REAL a(4,4)
HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: a

REAL b(7)
HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO line :: b

HPF: DISTRIBUTE

• CYCLIC [blocksize]
  – Blocks with equal number of elements are cyclically distributed among the processors.
  – Default blocksize is 1.
  – For cyclic(m) element with index i will be mapped on processor
    \[ i \mod m \]
  – No specific distribution
  – All of them, overlapping …

HPF: DISTRIBUTE

(BLOCK,*) (*, CYCLIC)

(BLOCK, BLOCK) (CYCLIC(2), CYCLIC(3))
HPF: DISTRIBUTE

(BLOCK,*) (*, CYCLIC)

ONTΟ line

ONTΟ square

(BLOCK, BLOCK) (CYCLIC(2), CYCLIC(3))

HPF: DISTRIBUTE

(BLOCK,*) (*, CYCLIC)

ONTΟ line

ONTΟ square

(BLOCK, BLOCK) (CYCLIC(2), CYCLIC(3))

HPF: DISTRIBUTE

(BLOCK,*) (*, CYCLIC)

ONTΟ line

ONTΟ square

(BLOCK, BLOCK) (CYCLIC(2), CYCLIC(3))
HPF: DISTRIBUTE

- Communication point of view:
  - BLOCK usually good for local communication
  - Locality of cyclic not easy to see
  - CYCLIC(m) has advantages of both
- Problems:
  - Access with stride very expensive
  - Data redistribution can be very expensive
  - DYNAMIC, REDISTRIBUTE

HPF: ALIGN

- Specify that some objects have to be mapped in the same way as certain other object.
  - Computation will be more efficient if data are mapped on the same virtual processor.
- Only the <target> can have a DISTRIBUTE
- Definitions:
  - Dummy variable: scalar variable used locally to distribute the data with values depending on the valid initial axis values
  - Subscript-triplet: similar to an implicit loop in FORTRAN
    - I:E:S values starting from I until E with a step of S
    - "*" is a kind of "dummy variable that cannot interfere in subsequent distribution."
HPF: ALIGN

• \(!HPF\) ALIGN <what> <source-list> WITH <target> <subscript-list>
  – <what> and <target> can be any dimensional array
  – <source-list> can be:
    • ‘*’ axis will be spread out across the matching axis of <what>
    • ‘**’ axis is collapsed: position across this axis make no difference in determining the corresponding position.
    • ‘dummy variable’: scalar range over all valid index values
  – <subscript-list> can be:
    • ‘*’ or ‘**’ or integer value or linear (affine) expression (depending on dummy variables) or subscript-triple or ‘dummy variable’

\[
\begin{align*}
\text{HPF: ALIGN} \\
\text{• } \text{Equivalent:} \\
\text{!HPF$ ALIGN A(:,*,K,:,:,* \text{ WITH B(31,:,K+3,20:100:3)}} \\
\text{is equivalent to} \\
\text{!HPF$ ALIGN A(J,K,L,M,N \text{ WITH B(L-BOUND(A,1)+31,}} \\
\text{L-BOUND(A,4)+BOUND(B,2), K+3,} \\
\text{M-BOUND(A,5))/3 + 20)}} \\
\text{with the conditions:} \\
\text{SIZE(A,1) EQ. UBOUND(B,1)-31} \\
\text{SIZE(A,4) EQ. SIZE(B,2)} \\
\text{SIZE(A,5) EQ. (100-20)/3/3}
\end{align*}
\]
HPF: ALIGN

- Not all notation are equivalent
  - \texttt{!HPF$ ALIGN A(:,*) \text{ WITH } D(:)}
  - \texttt{!HPF$ ALIGN A(:,j) \text{ WITH } D(:)}
  - \texttt{!HPF$ ALIGN A(:) \text{ WITH } D(:,*)}
  - \texttt{!HPF$ ALIGN A(:) \text{ WITH } D(:,j)}

- Only a variable appearing in the source-list is understood to be a align-dummy

---

HPF: ALIGN

- \texttt{REAL A(4), B(8)}
- \texttt{!HPF$ ALIGN A(:) \text{ WITH } B(1:4)}
- \texttt{!HPF$ ALIGN A(:) \text{ WITH } B(2:8:2)}
- \texttt{!HPF$ ALIGN A(:) \text{ WITH } B(8:5:-1)}
- \texttt{!HPF$ ALIGN A(j) \text{ WITH } B(8-2*j+1)}

---

HPF: ALIGN

- "*":
  - In source-list = collapsing
    - Multiple elements of source array will be aligned with a single element of the target
  - In subscript-list = replication
    - A single element of source array will be copied and aligned to multiple element of the target.
    - BUT update at runtime require a \textit{global communication}
**HPF: TEMPLATE**

- Allow to define a virtual array that can be distributed and used as a ALIGN target.

```hpf
!HPF$ TEMPLATE DISTRIBUTE(BLOCK, BLOCK) :: EARTH(N,N)
REAL, DIMENSION(N,N) :: NW, NE, SW, SE
!HPF$ ALIGN NW(I,J) WITH EARTH(I,J)
!HPF$ ALIGN NE(I,J) WITH EARTH(I,J+1)
!HPF$ ALIGN SW(I,J) WITH EARTH(I+1,J)
!HPF$ ALIGN SE(I,J) WITH EARTH(I+1,J+1)
```

**HPF: data alignment**

- How about the procedure call?
- Some functions require different data alignments
- A subprogram can INHERIT data distributions from the calling routine

```hpf
!HPF$ INHERIT array
```

**HPF: data alignment**

- Explicit interface: subprogram define his own data alignments in the interface block
- Variables present in this interface will be copied into temporary variables for the duration of the subprogram prior the entry in the subprogram
- They will be copied back in the original distribution on return from the subprogram

This approach might generate communication
HPF: data alignment

- Prescriptive: describe the mapping of the dummy argument. 
  !HPF$ DISTRIBUTE A(BLOCK, CYCLIC)
  • If the actual argument does not have this mapping the 
    compiler will generate the code to correctly remap the data 
  • Information available at compile time
- Descriptive: weak assertion by the programmer that there is no 
  need to remap
  !HPF$ DISTRIBUTE A *(BLOCK, CYCLIC)
  • Compiler check and generate warning
- Transcriptive: no specified mapping
  • The caller pass the mapping information at runtime
  • !HPF$ DISTRIBUTE A* [ONTO *]

HPF: data alignment

!HPF$ DYNAMIC array
• Such arrays can/will be realigned or 
  redistributed at runtime
!HPF$ REALIGN <what> <source-list> WITH 
<target><subscript-list>
• Redistribute <what> and all arrays aligned 
  to it !!!

HPF: high level parallelism

• FORALL: tightly-coupled parallel execution 
  based on the structure of the index space
• PURE: procedure without side effects (to 
  be used with FORALL)
• INDEPENDENT: assertion that iterations 
  do not interfere with each other.
HPF: FORALL

- Has the semantic of array assignment but it’s more clear and concise
- More general array regions
- More general access patterns
  
  ```fortran
  FORALL (index-spec-list[,mask-expr]) forall-assignment
  FORALL (I = 1,10) a(I) = b(I) + c(I+1)
  FORALL (I = 1,10) a(index(I)) = b(I)
  ```

- Does not support multiple assignments
  
  ```fortran
  FORALL (I = 1, 10, a(I) > 1.0) a(I) = 1.0/a(I)
  FORALL (I = 2, 9) a(I) = a(I-1) + a(I+1)
  a(2:9) = a(1:8) + a(3,10)
  ```

HPF: FORALL

- The execution of a single statement FORALL has four steps:
  - Compute the valid index set.
    - Based on `index-spec-list`.
  - Compute the active index set.
    - Based on `mask-expr`.
  - Compute the right hand side.
    - Any order - perhaps parallel!
  - Assign to the left hand side.
    - Any order - perhaps parallel!

HPF: FORALL

- Multi-statement FORALL
  
  ```fortran
  FORALL (index-spec-list[,mask-expr]) forall-body-statements
  END FORALL
  ```

- Equivalent to multiple single FORALL
- Nested: the inner FORALL modifies the active index set.
  
  ```fortran
  FORALL (I=1:3, J=1:3, I>J)
  FORALL (K=1:3, L=1:J, K+L > J)
  A(I,J,K,L) = J*K + L
  END FORALL
  END FORALL
  ```
HPF: FORALL

Index sets for this example:
Outer valid set : {(1,1),(2,1),(3,1),(1,2)...(3,3)}
Outer active set : {(2,1),(3,1),(3,2)}
Inner valid set: {(2,1,1,1),(2,1,2,1),(2,1,3,1)...(3,2,2,2),(3,2,3,1),(3,2,3,2)}
Inner active set: {(2,1,2,1),(2,1,3,1),(3,1,3,1), (3,2,2,2),(3,2,3,1),(3,2,3,2)}

HPF: Pure attribute

- is guaranteed side-effect free
- returns a value and does not modify global data or pointer associations or perform input/output

```fortran
PURE INTEGER FUNCTION f(x, y)  
  IMPLICIT NONE  
  INTEGER, INTENT(IN) :: x, y  
  f = x*x + y*y * 2*x*y  
END FUNCTION f
```

- Can be used in FORALL statements

```fortran
FORALL (i=1 : n, j=1 : n, l .NE. j, a(i, j) = f(i, j))  
```

- Most of the intrinsic Fortran90 functions

HPF: INDEPENDENT

- no iteration of the loop statement to which it applies affects any other iteration

```fortran
!HPF$ INDEPENDENT [, NEW(private-list), REDUCTION(reduct-list)]
```

- DO loop
  iterations or assignments can be performed in any order

```fortran
!HPF$ INDEPENDENT
DO i = 1, n
  j(i) = i * i
END DO
```

- FORALL loop
  the whole right-hand side does not have to be evaluated before assignment to the left-hand side can begin

```fortran
!HPF$ INDEPENDENT
FORALL (i=1 : n) j(i) = i * i
```
HPF: INDEPENDENT

• Support temporary variables which will be local to every virtual processor via NEW
  \texttt{HPF$\text{INDEPENDENT}, \text{NEW}(\text{tmp})}
  \begin{verbatim}
  DO i=1,n
    tmp = SUM(B(i,:))
    A = tmp*tmp
  END DO
\end{verbatim}

• The temporary variable is uninitialized on each loop!

---

HPF: INDEPENDENT

• Support global variables update using commutative global functions via REDUCTION
  \texttt{HPF$\text{INDEPENDENT, REDUCTION(Z)}$}
  \begin{verbatim}
  Z = 5.
  DO I = 1, 10
    Z = Z + I
  ENDDO
\end{verbatim}

• The reduction function should be commutative & associative (as MPI reduction functions): +, -, *
  , /, OR, AND, IEOR, IAND, MIN, MAX

• Some operations can be used together like (+,-) and (\texttt{*,/})

---

HPF: DO vs. FORALL

\begin{verbatim}
DO i=1, 3
  tmp(i) = tmp(i) + tmp(i)
END DO
\end{verbatim}

\begin{verbatim}
FORALL (i=1:3)
  tmp(i) = tmp(i) + tmp(i)
END FORALL
\end{verbatim}
HPF: Intrinsic Functions

- System inquiry: NUMBER_OF_PROCESSOR, PROCESSOR_SHAPE
- Mapping inquiry:
  - Bit manipulation: LEADZ, POPCNT, POPPAR
  - Array Reduction: IALL, IANY, IPARITY, PARITY
- Array sorting functions: SORT_UP, SORT_DOWN, GRADE_UP, GRADE_DOWN (index sort depending on the array value at index)

HPF: Intrinsic Functions

- Array Combining: XXX_SCATTER(ARRAY, BASE, INDEX1, ..., INDEXN, MASK)
  - Where XXX : ALL, ANY, COPY, COUNT, IALL, IANY, MAXVAL, MINVAL, PRODUCT & SUM
  - An arbitrary subset of the elements of an array can be combined to produce an element of the result.

Example:

```python
if A = [10 20 30 40 -10],
B = [1 2 3 4] and
IND = [3 2 2 1 1] then
SUM_SCATTER(A, B, IND, MASK=A.GT.0) is
[41 52 13 4]
```
HPF: Intrinsic Functions

• Array prefix and suffix
  – Each element of the result is a function of the elements that precede it or that follow it.
  Example:
  If \( A = [3 \ 5 \ -2 \ -1 \ 7 \ 4 \ 8] \) then
  \[ \text{SUM\_PREFIX}(A, \text{MASK}=A.\text{LT}.6) \] is
  \[ [3 \ 8 \ 6 \ 5 \ 5 \ 9 \ 9] \]

HPF through classical examples

• Jacobi
• Gauss Elimination
• Conjugate Gradient
• Irregular mesh relaxation

HPF: Jacobi example

• Numerical solution to Laplace’s equation

\[
U_{i,j}^{n+1} = \frac{1}{4} (U_{i,j-1}^n + U_{i,j+1}^n + U_{i-1,j}^n + U_{i+1,j}^n)
\]

for \( j = 1 \) to \( j_{\text{max}} \)
for \( i = 1 \) to \( i_{\text{max}} \)
  \[
  U_{\text{new}}(i,j) = 0.25 \times (U(i-1,j) + U(i+1,j) + U(i,j-1) + U(i,j+1))
  \]
end for
end for

Plenty of data parallelism.
HPF: Jacobi example

- **Element updates:**
  - 4 elements from the previous step
  - Generate static communication (compile time)

- **Convergence test**
  - Use all values from the last step
  - Global communication
    - Reduction operation
    - Efficient operation (encapsulated in HPF library).

- **Element updates require 4 neighbors**
  - CYCLIC: worst case as all elements will be exchanged between processors
  - BLOCK: less communications as some data will reside on the same processor
    - (BLOCK, *): move U(i-1,j) & U(i+1,j) for all J
    - (BLOCK, BLOCK): move
      - U(iLOW-1, J) & U(iHIGH+1, J) for all J
      - U(i, JLOW-1) & U(i, JHIGH+1) for all I

- **Convergence require the array reduction**
  - Any distribution is OK: static, structured communications.

```hpf
REAL u(0:nx), unew(0:nx)
!HPF$ DISTRIBUTE u(BLOCK, BLOCK)
!HPF$ ALIGN (:,:), WITH u(:,:): unew
DO WHILE (err .GT. epsilon)
  FORALL (i=1:nx-1, j = 1:nx-1)
    unew(i,j) = (u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j-1)) / 4
  Err = MAXVAL_SCATTER(ABS(unew-u))
  u = unew
ENDDO
```
HPF: Jacobi example

- The compiler get the easy job
  - Allocate data on processors based on DISTRIBUTE
  - Apply owner-compute rule based on left-hand side
  - Detect SHIFT communications from dependencies analysis
  - Identify MAXVAL as a global reduction function
  - Place all communications outside the parallel loop (optimization)
  - Number processors so that shift do not cause contention

HPF: Gauss Elimination

- Linear solver: Given N linear equations in N unknown $x_i$, find all $x_i$ which satisfy the equations.
- Algorithm:
  - Use eq.1 to eliminate $x_1$ from equations (2..N)
  - Use eq.2 to eliminate $x_2$ from equations (3..N)
  - Use eq.N to compute $x_n$
  - Backward compute $x_{n-1}$, ..., $x_2$, $x_1$
HPF: Gauss Elimination

• Gaussian elimination look like a sequential algorithm:
  – Need whole column k to find pivot row
  – Need the whole column k and pivot row to perform pivoting
• Each step in the process it’s composed by many smaller operations: they can be updated independently
• Still enough data parallelism do make a parallel algorithm (except the last stages).

HPF: Gauss Elimination

• How about communications?
  – Pivot search:
    • Reduction among column
    • Static, global communication
  – Element updates:
    • Each element from itself, pivot column and row
    • Static, global communication (broadcast)

HPF: Gauss Elimination

• Pivot requires a 1-D reduction
  – Distribute rows => parallel with communications
  – Distribute columns => parallel without communications
• Element updates requires old value, value from pivot and value from column
  – Distribute rows => parallel + broadcast pivot column
  – Distribute column => parallel + broadcast pivot row
• Which distribution?
  – BLOCK: processors drop out of computation
  – CYCLIC: work stays distributed until the end, each time all processors have less work to do.
• Best distribution:
  – (*, CYCLIC) if broadcast > pivoting one column
  – (CYCLIC,*) if broadcast < one column, synchronous communications
  – (CYCLIC, CYCLIC) if broadcast < one column, overlapped communications
HPF: Gauss Elimination

REAL A(n,n), tmp(n)
!HPF$ DISTRIBUTE a(CYCLIC, CYCLIC)
!HPF$ ALIGN tmp(i) with a(*,j)
DO k = 1, n – 1
   ipivot = MAXLOC( ABS( A(k:n, k)) ) + k – 1
   tmp(k:n) = a(ipivot, k:n)
   a(ipivot, k:n) = a(k, k:n)
   a(k, k:n) = tmp(k:n)
   FORALL ( i=k+1:n, j=k+1:n )
      a(i,j) = a(i,j) – a(i,k) / tmp(k) * tmp(j)
ENDDO

Select the pivot
Static global reduction

Swap the rows
Local copies
REAL A(n,n), tmp(n)
!HPF$ DISTRIBUTE a(CYCLIC, CYCLIC)
!HPF$ ALIGN tmp(i) with a(i,j)
DO k = 1, n - 1
  ipivot = MAXLOC(ABS(A(k:n, k))) + k - 1
  tmp(k:n) = a(ipivot, k:n)
  a(ipivot, k:n) = a(k, k:n)
  a(k, k:n) = tmp(k:n)
  FORALL (i=k+1:n, j=k+1:n) &
    a(i,j) = a(i,j) - a(i,k) / tmp(k) * tmp(j)
ENDDO

Select the pivot
Static global reduction
Swap the rows
Local copies
Broadcast pivot row
and column

Update the sub-matrix
HPF: Gauss Elimination

- More difficult for the compiler
  - Allocate portion of array on each processor based on `DISTRIBUTE`
  - Owner-compute applied based on left-hand side
  - Detect communications from dependencies & intrinsic
- Transform the program:
  - Reorder computation to always pre-compute the next pivot column
  - Rearrange the communications to pipeline the updates
  - Broadcast values asynchronously
  - Net result: a 2x speedup
- Use standard numbering for processor

HPF: Irregular mesh relaxation

- Given a irregular mesh of values
- Update each value using its neighbors in the mesh
- Airflow simulation, car crash …
- Algorithm
  - Store the mesh as a list of edges
  - Process all edges in parallel
    - Compute contribution of the edge
    - Add to the endpoint, subtracts from the other

HPF: Irregular mesh relaxation

```fortran
REAL x(nnodes), flux(nedges)
INTEGER iedges(nedges,2)
err = tol * 1e6
DO WHILE( err .GT. Tol )
  DO i=1, nedges
    flux(e) = (x(iedges(e,1)) – x(iedges(e,2))/2
    err = err + flux(i) * flux(i)
  ENDDO
  DO I = 1, nedges
    x(iedges(i,1)) = x(iedges(i,1)) – flux(i)
    x(iedges(i,2)) = x(iedges(i,2)) + flux(i)
  ENDDO
  err = err / nedges
ENDDO
```
HPF: Irregular mesh relaxation

• Each iteration use all data computed in the previous step and the edge array.
  – No parallelism at all
  – Instead use data-parallel edge and node updates
    • flux(i) = (x(iedge(i,1)) – x(iedge(i,2)))/2
    • x(iedges(i,1)) = x(iedges(i,1)) – flux(i)
    • x(iedges(i,2)) = x(iedges(i,2)) + flux(i)
    • Not independent as sometimes iedges(i,1) ≠ iedges(i,2)
    • But we can use a SUM_SCATTER
  – 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

HPF: Irregular mesh relaxation

• Computing edge values require 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 of endpoints
• Updating node values require edge list, edge values and node values
  – Distribute edges => parallel, no communications
  – Replicate edges => sequential, no communication
  – Distribute nodes => move “shared” endpoints
  – Replicate nodes => move all endpoints
• The bottom line
  – Always distribute edges
  – Distribute nodes unless the problem is very small

HPF: Irregular mesh relaxation

• Computation is static, and over full array with respect to edges
  – No load balancing issues
• Access to node array are “nearest neighbor” in the mesh
  – Not reflected in the index order
  – Does not favor either BLOCK or CYCLIC
• To minimize communications, edge and node distribution must fit the topology
  – Difficult (impossible) with HPF regular distributions
  – HPF 2.0 indirect distributions may be better, but require careful construction
• Idea: order the nodes and edges to keep “close” entities together and then use BLOCK
HPF: Irregular mesh relaxation

- BAD data distribution

HPF: Irregular mesh relaxation

- Good data distribution
HPF: Irregular mesh relaxation

• Challenging …
  – Indexing of array will be difficult
  – How to apply the owner-compute rule?
• Key technique: inspector-executor communication
  – First time the code is executed, generate a table of required communication at runtime (inspector)
    • How big is it? How we can efficiently distribute the table to all processors?
  – Use this table to manage unstructured communication until the communication pattern change (executor)
    • How do we know that the pattern change?