Data Parallelism for Engineering Applications: HPF, HPF-2, and JaHPF

Rolf Rabenseifner, Alfred Geiger
University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

Outline

- Introduction
- Data distribution
- Independent loops
- Pure, Reduction
- Shadows, Intrinsic functions
- Kinds of distribution
- Features of HPF 2.0
- Summary
- JaHPF
Multicomputer

Node or PE

Memory-Segment

CPU

Memory-Segment

CPU

Memory-Segment

CPU

Memory-Segment

CPU

Node-Interconnect

Multiprocessor

Memory-Segment

CPU

Memory-Segment

CPU

Memory-Segment

CPU

Memory-Segment

CPU

Memory - Interconnect
Goals of HPF

- Definition of a parallel programming standard
- Easy to program
- Few local chances of serial code
- Hardware independency

HPF - History

- Defined by the High Performance Fortran Forum (HPFF)
- Proposed at Supercomputing’91
- Final Draft in June 1993
- Definition of HPF-2 in 1994 and 1995
- Influences from CM-Fortran, MasPar, DEC, Fortran-D, Vienna Fortran
- Goal: Portable Language for all platforms

- JaHPF in January 1999
Outline

• Introduction
• Data distribution
  • Independent loops
  • Pure, Reduction
  • Shadows, Intrinsic functions
  • Kinds of distribution
  • Features of HPF 2.0
• Summary
• JaHPF

Goals of Data-Mapping

• Load-Balancing: Same load on all nodes
• Locality: Data used together on same node
• Minimal Contention: Data used in parallel on different processors
• Potential Conflicts:
  – Maximal Locality: All data on same processor
  – Minimal Contention: All data on different processors
Concept of Data-Mapping

Two-step Mapping-Concept:

**Problem-Mapping**

- **ALIGN**
- **REALIGN**

**Processor-Mapping**

- **DISTRIBUTE**
- **REDISTRIBUTE**

Problem-Mapping: **TEMPLATE**, **ALIGN** and **REALIGN**

- **!HPF$ TEMPLATE t(1:n,1:n)**
  - Abstract space of indexed positions
  - Declaration

- **!HPF$ ALIGN a WITH t**
  - Mapping of arrays relative to templates or other arrays
  - Declaration
  - e.g. transposed **!HPF$ ALIGN a(j,i) WITH t(i,j)**
  - e.g. multigrid **!HPF$ ALIGN c(i,j) WITH f(2*i,2*j)**

- **!HPF$ REALIGN a(j,i) WITH t(i,j)**
  - Realignment during runtime
  - Executable Statement
  - Arrays must have DYNAMIC attribute (f90)
Processor-Mapping: PROCESSORS, DISTRIBUTE and REDISTRIBUTE

- !HPF$ PROCESSORS p(1:n,1:n)
  - Virtual processor-topology
  - Only grid-topologies possible (syntax of FORTRAN-arrays)
  - Declaration

- !HPF$ DISTRIBUTE t(BLOCK,CYCLIC) ONTO p
  - Mapping of templates to processor arrays
  - BLOCK
  - CYCLIC
  - CYCLIC(n)
  - BLOCK(n)
  - Declaration

- !HPF$ REDISTRIBUTE t ONTO p2
  - Redistribution during runtime
  - Executable Statement
  - Arrays must have DYNAMIC attribute (f90)

Example: Serial code

Multiplication matrix * vector

INTEGER, PARAMETER :: N = 10000
INTEGER, DIMENSION(N, N) :: A
INTEGER, DIMENSION(N) :: B, C
INTEGER :: i, j

C=0
DO i=1,N
  DO j=1,N
    C(i) =C(i) + A(i, j) * B(j)
  END DO
END DO
Example: Data distribution

```fortran
INTEGER, DIMENSION(N, N) :: A
INTEGER, DIMENSION(N) :: B, C

!HPF$ PROCESSORS Procs(4)
!HPF$ DISTRIBUTE(BLOCK,*) ONTO Procs :: A
!HPF$ ALIGN C(i) WITH A(i,*)

C=0
DO i=1,N
  DO j=1,N
    C(i) = C(i) + A(i, j) * B(j)
  END DO
END DO
```

Distribution matrix * vector

```
C = A * B
```
Limitations of Data Distribution

- Global Name-Space
  - Distributed arrays may not be used when storage- or sequence-association is needed (e.g. collapsing dimensions).

  \[
  \text{REAL } A(1:N,1:N) \\
  \text{global address-space: } A(N+1) = A(1,2) \\
  \text{global name-space: } A(N+1) \ll A(1,2)
  \]

- Single or multiple threads of control (implementation dependent)
  - Less synchronisation in multithreaded case
  - Replication of Scalars in multithreaded case

Compile & Run

- Login at T3E:
  \[\text{ssh hwwt3e}\]
- Compile
  \[\text{hpf -o } \text{<Executable_parallel> } \text{<Fortran_File>}\]
- Run
  \[\text{mpirun -np } \text{<Number_Of_Processors> } ./\text{<Executable_parallel>}\]

  \textbf{Exercises}: Set \text{<Number_Of_Processors>} to 4.

- For serial compilation:
  \[\text{f90 -o } \text{<Executable_serial> } \text{<Fortran_File>}\]
  \[./\text{<Executable_serial>}\]

- Time measurement:
  \[\text{time mpirun -np } \text{<Number_Of_Processors> } ./\text{<Executable_parallel>}\]
  \[\text{time } ./\text{<Executable_serial>}\]
PGHPF Compiler-Syntax

```
pghpf -Mhpfoption -f90option prog.hpf
```

Important Options:
- Mautopar  Automatic Parallelisation of Loops
- Mg       Debug-Option
- Minline   Inlineing of independent Loops
- Msequence Create all variables in sequence

The PGI HPF-Compilation System

- Developed by Portland Group
- Available on Cray T3E, Intel Paragon, IBM SP/2 and all Workstation/PC-Clusters

```
pghpf
gphpf -c
f90
asm
ld
```

HPF-Source

FORTRAN 77/90 + Communication

Assembly-Language

Object-File

Executable
Exercise 1: Matrix multiplication (i)

- Calculates the product of two squared matrices
- $C(i, j) = \text{sum } (A(i,k) \times B(k, j))_{k=1:N}$

---

Exercise 1: Matrix multiplication (i) — continued

- Change to your working directory
  ```bash
  cd ~/HPF/#nr with #nr=number of your PC
  ```

- Open exa1_matrix.f90
  ```bash
  cp ../course/exa1_matrix.f90 .
  ```

- Use data distribution to parallelize the algorithm
- Compile on T3E
- Execute program
- Compare output with serial version
- Compare your solution with exa1_matrix.hpf.f90
Exercise 1: Matrix multiplication (i) — Results

- 3 different solutions for C
  - BLOCK,* = 4x1
  - *,BLOCK = 1x4
  - BLOCK,BLOCK = 2x2
- Bad alignments
  - Hugh execution time
  - But nevertheless correct

Possible distributions

- $C := A * B$
- $C := A * B$
- $C := A * B$

37. — Data Parallelism: HPF, HPF-2 and JaHPF — 37-11
Outline

- Introduction
- Data distribution
- **Independent loops**
  - Pure, Reduction
  - Shadows, Intrinsic functions
  - Kinds of distribution
  - Features of HPF 2.0
- Summary
- JaHPF

Motivation: Independent loops

Work

Compilation

- W
- O
- R
- K

Do it parallel!
### Expression of Parallelism: INDEPENDENT

The INDEPENDENT directive guarantees that sequence of execution doesn't matter ➔ possibility of parallelisation.

```hpfs```
INDEPENDENT
DO i = 1, N
    DO j = 1, N
        C(i) = C(i) + A(i, j) * B(j)
    END DO
END DO
```

Note that only the outer loop is independent!

### Expression of Parallelism: FORALL

- Simultaneous assignment to a group of array-elements
- *Generalized Loop, but without predefined order of the operations*
- Example
  ```hpfs```
  FORALL (i = 1:N, j = 1:M, Y(i, j) .NE. 0.0)
  X(i, j) = 1.0 / Y(i, j)
  END FORALL
  ```

```fortran```
DO i = 1, N
    DO j = 1, M
        IF (Y(i, j) .NE. 0.0) THEN
            X(i, j) = 1.0 / Y(i, j)
        END IF
    END DO
END DO
```

---

---
Expression of Parallelism: FORALL (continued)

- Simultaneous assignment to a group of array-elements
- More correct: Generalized Array Assignment
- Example

\[
\begin{align*}
\text{FORALL} & \ (i = 1:N, \ j = 1:M, \ Y(i, j) \ .NE. \ 0.0) \\
& \ X(i, j) = 1.0 \ / \ Y(i, j) \\
\text{END FORALL}
\end{align*}
\]

- Computation sequence:
  - Valid set of index values
  - Active set of index values
  - Compute all expressions (pointers, right-hand-side)
  - Assign right-hand-side values to the left-hand-side

WHERE-statement

- Describes a loop with a single conditioned statement (or a group of statements).
- Execution in dependency of a logical array expression.

\[
\begin{align*}
\text{REAL, DIMENSION}(1000) & :: \ A, \ B \\
\text{WHERE} & \ (A /= 0.0) \\
& \ B = 1.0 \ / \ A \\
\text{ELSEWHERE} & \\
& \ B = 1.0 \\
\text{END WHERE}
\end{align*}
\]
Exercise 2: Matrix multiplication (ii)

- Calculates the product of two squared matrices
- \[ C(i, j) = \sum_{k=1}^{N} (A(i,k) \times B(k, j)) \]

Exercise 2: Matrix multiplication (ii) — continued

- Open exa2_matrix.f90
  - cp ../course/exa2_matrix.f90 .
- Use as many as possible INDEPENDENT-directives to parallelize the algorithm
- Compile on T3E
- Execute the program
- Compare output with serial version
- Compare your solution with exa2_matrix.hpf.f90
The PURE attributed

The PURE attribute is an assertion to the compiler, that a function has no side effects, i.e. a loop containing this function can be parallelised.

A function has to be PURE, if it is used in one of the following contexts:

- The mask or body of a FORALL statement
- Within the body of a pure procedure
- As an actual argument in a pure procedure reference

Note that all intrinsic functions are PURE.
Expression of Parallelism: PURE Procedures

- No change of global data which is not in parameter-list
- No change of global pointer-associations and data-mappings
- Dummy arguments should have attribute
  - INTENT(IN) on functions
  - INTENT(IN) or INTENT(OUT) on subroutines
- Example:

  ```fortran
  FUNCTION f(x)
  !HPF$ PURE f
  ......
  ......
  END FUNCTION
  ```

- Recommendation: Make all functions and subroutines PURE in the first step of parallelisation with HPF

HPF-2: Parallelism: Loop-Reductions

```
!HPF$ INDEPENDENT, NEW(c), REDUCTION(x)
DO i = 1, N
  c = f(i)
  x = x + g(c)
END DO
```

Reduction-Treatment:
- sequential (reproducible)
- parallel (tree)
Example: Reduction

! Standard deviation

REAL, DIMENSION(N) :: Statistic

Median = SUM(Statistic) / N
! Intrinsic SUM, implicit reduction

! HPF$, INDEPENDENT, REDUCTION(X)
! Userdefined reduction
DO i = 1, N
    X = X + (Statistic(i) – Median) ** 2
END DO

Standard_Deviation = SQRT(X / N)

The Set-Compute-Rule

• Worksharing according to the Set-Compute-Rule
  – The node which has the data in the LHS of a statement does the work
  – Influencable with ON clause
  – General worksharing with LOCAL subroutines
Communication-Rules

- Communication is implicit and hidden from the user.
- Communication takes place (default) when operands are on different processors (indicated by different colors)
  - \( a(i) = b(j) + b(j+1) \) Communicate \( b(j), b(j+1) \)
  - \( x = a(i) + b(j) \) Communicate \( a(i), b(j) \)
- By default, operations are performed on the PE which owns the LHS.
- In implementations following the SPMD-model, scalars are replicated and synchronized.

Expression of Parallelism: Execute ON HOME

Suggest where iterations are performed by influencing the set-compute-rule

```
!HPFS INDEPENDENT           DO K = 1, 6!HPFS ON HOME (I(K))
Y(K) = X(I(K)) + X(J(K))
END DO
```

Standard

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

ON HOME

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

```
Y(1) = X(1) + X(1)   Y(1) = X(1) + X(1)
Y(2) = X(1) + X(1)   Y(2) = X(1) + X(1)
Y(3) = X(3) + X(4)   Y(3) = X(3) + X(4)
Y(4) = X(4) + X(3)   Y(4) = X(4) + X(3)
Y(5) = X(6) + X(6)   Y(5) = X(6) + X(6)
Y(6) = X(6) + X(6)   Y(6) = X(6) + X(6)
```
Exercise 3: Derivation

• This exercise solves the differential equation (linear equation):

\[ f(0) = 0; \quad f(x) = f(x-1) + dx \]

• The saved results are derivated and should be constant

\[ f'(x) = dx \quad \text{(constant)} \]

• Then we integrate the derivation:

\[ \text{Area} = F(f'(x)) = f(x) \]

Exercise 3: Derivation (continued)

• Open exa3_diff.f90

• distribute the data via DISTRIBUTE clause
• use INDEPENDENT DOs to parallelize loops wherever possible
• Use REDUCTION to determine the area bounded by derivation
• Output “area”

• Compare output with serial version
• Compare your solution with exa3_diff.hpf.f90
Outline

- Introduction
- Data distribution
- Independent loops
- Pure, Reduction
- **Shadows, Intrinsic functions**
  - Kinds of distribution
  - Features of HPF 2.0
  - Summary
  - JaHPF

Intrinsics

**what are intrinsics?**

- predefined functions
- HPF offers F77 / F90 intrinsics optimized for multiprocessing
- handle basic tasks and array operations
**Intrinsic types**

**array manipulation**
- SORT_DOWN, SORT_UP to sort an array
- CSHIFT, TRANPOSE to reorganize array data
- MERGE, SPREAD, PACK to operate on arrays

**reduction operations**
- logicals like MAXVAL, MINVAL
- boolean functions like ALL, ANY
Intrinsic types

inquiry functions

needed to handle HPF parallelization, e.g.

- NUMBER_OF_PROCESSORS
- PROCESSORS_SHAPE (returns shape of processor array)

HPF-2: Distribution with Shadowing

!HPF$ DISTRIBUTE (*,BLOCK) :: A
!HPF$ SHADOW (0:0, 1:1) :: A
Exercise 4: Second derivation

• A function \( f(x) = x^3 / 6 \) is being calculated and its values are stored.

• In the next step, the second derivation is determined directly:

\[
f''(x) = f(x+1) + f(x-1) - 2.0 \cdot f(x)
\]

Exercise 4: Second derivation (continued)

• Open exa4_diff.f90

• (parallelize like exercise 3)
• Use shadows to optimize communication
• Instead of INDEPENDENT DO – loop use intrinsic CSHIFT

\[
\text{CSHIFT(array, shift=\ldots , dim=\ldots )}
\]

returns the ring-shifted array, shifted in dimension “dim”

Example: array =
\[
\begin{bmatrix}
1, & 2, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 0
\end{bmatrix}
\]

cshift(array,3,1) =
\[
\begin{bmatrix}
4, & 5, & 6, & 7, & 8, & 9, & 0, & 1, & 2, & 3
\end{bmatrix}
\]

• Compile on T3E
• Compare output with serial version
• Compare your solution with exa4_diff.hpf.f90
Outline

- Introduction
- Data distribution
- Independent loops
- Pure, Reduction
- Shadows, Intrinsic functions
- Kinds of distribution
- Features of HPF 2.0
- Summary
- JaHPF

Communication-Requirements

Communicated array-elements (Data-Volume):
All elements that are combined residing on different nodes

Latencies:
Data-Volume (sequential case)
Number of distinct ordered pairs of PEs (packed case)
1 (parallel case)

Example:
REAL X(1:9), Y(1:9)
!HPF$ DISTRIBUTE X(BLOCK)
!HPF$ DISTRIBUTE Y(CYCLIC)
DO I = 1,9,1
Y(I) = X(I)
END DO
Communication: BLOCK

Constant Offset

REAL X(1:9), Y(1:9)
!HPF$ DISTRIBUTE X(BLOCK)
!HPF$ DISTRIBUTE Y(BLOCK)
DO I = 1,8,1
  Y(I) = X(I+1)
END DO

Linear Offset

REAL X(1:9), Y(1:9)
!HPF$ DISTRIBUTE X(BLOCK)
!HPF$ DISTRIBUTE Y(BLOCK)
DO I = 1,4,1
  Y(I) = X(2*I)
END DO

Communication: BLOCK, Non-Matching Dimensions

Constant Offset

REAL X(1:9), Y(1:6)
!HPF$ DISTRIBUTE X(BLOCK)
!HPF$ DISTRIBUTE Y(BLOCK)
DO I = 1,6,1
  Y(I) = X(I+1)
END DO

Linear Offset

REAL X(1:9), Y(1:6)
!HPF$ DISTRIBUTE X(BLOCK)
!HPF$ DISTRIBUTE Y(BLOCK)
DO I = 1,4,1
  Y(I) = X(2*I)
END DO
Communication: CYCLIC

**Constant Offset**

```
REAL X(1:9), Y(1:9)
!HPF$ DISTRIBUTE X(CYCLIC)
!HPF$ DISTRIBUTE Y(CYCLIC)
DO I = 1,8,1
    Y(I) = X(I+1)
END DO
```

Volume: 8
Latencies: 5(3)

**Linear Offset**

```
REAL X(1:9), Y(1:9)
!HPF$ DISTRIBUTE X(CYCLIC)
!HPF$ DISTRIBUTE Y(CYCLIC)
DO I = 1,4,1
    Y(I) = X(2*I)
END DO
```

Volume: 3
Latencies: 3(2)

---

Communication: CYCLIC, Non-Matching Dimensions

**Constant Offset**

```
REAL X(1:9), Y(1:6)
!HPF$ DISTRIBUTE X(CYCLIC)
!HPF$ DISTRIBUTE Y(CYCLIC)
DO I = 1,6,1
    Y(I) = X(I+1)
END DO
```

Volume: 6
Latencies: 6(3)

**Linear Offset**

```
REAL X(1:9), Y(1:6)
!HPF$ DISTRIBUTE X(CYCLIC)
!HPF$ DISTRIBUTE Y(CYCLIC)
DO I = 1,4,1
    Y(I) = X(2*I)
END DO
```

Volume: 3
Latencies: 3(2)
Effort of communication

(BLOCK, BLOCK) → (*) BLO CK

Distribution onto P (here P = 9) processors

Communication C:

\[ C = (\sqrt{P} - 1) \times 2 \times L \]

\[ C = (P - 1) \times L \]

In general, P>4, (BLOCK, BLOCK) reduces communication!

Common blocks

- Define a local view to ONE global object
- Are declared in one or more subroutines, for example in subroutine Alpha
  - common /Data/ A(8, 8), B(20, 5)
- in subroutine Beta
  - common /Data/ R(8, 8), S(20, 5)
- Should ALWAYS consist of the same set of variables, i.e.
  - same type
  - same size
  - same shadow everywhere it appears.
- Names may differ
- Other use is DANGEROUS!
Example: common blocks

subroutine Alpha
common /SharedData/ A(16, 16), B(256), C(32, 4, 4)
common /Dangerous/ P(8, 16), R(8, 64), S(8, 64)
common /Pedigro/ W(4, 4), X(4, 16)
...  
end subroutine Alpha

subroutine Beta
common /SharedData/ D(16, 16), E(256), F(32, 4, 4)
common /Dangerous/ T(8, 80), U(84, 8)
common /Pedigro/ Y(5, 5), Z(3, 7)
...  
end subroutine Beta

Problems in HPF

In Subr. Alpha: Common /Data/ A(20), B(40)
In Subr. Beta: Common /Data/ X(30), Y(30)

What about distributions of /Data/ ?
What about dynamic redistributions of A or B?
Example: common blocks

- Variables in common blocks may be explicitly distributed, if, and only if they have
  - same size
  - same type
  - same shadow
  - same mapping
everywhere they appear.

- Otherwise they have to be declared as sequential
  \texttt{HPF$ sequential /Dangerous/}
to force the compiler associating storage linear.

- Redistribution is forbidden to guarantee the compiler having the same mapping everywhere!

HPF-2: Parallelism: Loop-Reductions

\texttt{HPF$ INDEPENDENT, NEW(c), REDUCTION(x)}
\begin{verbatim}
DO i = 1, N
  c = f(i)
  x = x + g(c)
END DO
\end{verbatim}

Reduction-Treatment:

- sequential (reproducible)
- parallel (tree)
Exercise 5: Block Distribution

an array is initialized with random numbers (1.00 / 0.00)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

in the next step the arithmetic average of upper/lower/left/right neighbour is calculated

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00</td>
<td>0.50</td>
<td>0.50</td>
<td>0.75</td>
<td>1.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.25</td>
<td>0.75</td>
<td>0.25</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>0.25</td>
<td>0.75</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Exercise 5: Block Distribution (continued)

- Open exa5_blockdistribution.f90
- Use BLOCK distributions, INDEPENDENT DO’s and NEW clause to parallelize the algorithm
- Compile on T3E
- Execute the program
- Compare output with serial version
- Compare your solution with exa5_blockdistribution.hpf.f90

- Additional exercise:
  - Replace nested do-loops by one FORALL
  - Compare with exa5_blockdistribution_forall.hpf.f90
Outline

- Introduction
- Data distribution
- Independent loops
- Pure, Reduction
- Shadows, Intrinsic functions
- Kinds of distribution
- **Features of HPF 2.0**
  - Summary
  - JaHPF

HPF, HPF-2 and Approved Extensions

- Irregular Distributions
- Asynchronous I/O
- Loop-based Reductions
- HPF-2 with Approved Extensions
**HPF-2: Distribution to Processor-Subsets**

```
!HPF$ PROCESSORS P(1:5)
!HPF$ DISTRIBUTE A(*,BLOCK) ONTO P(3:5)
```

```
3 4 5
```

**HPF-2: Irregular Distributions**

```
INTEGER map(1:15) = (/ 1,3,2,2,2,3,1,1,3,2,1,3,3,2,2 /)
!HPF$ DISTRIBUTE A(*, INDIRECT(map))
```
HPF-2: Distribution with Flexible Blocksize

```
!HPF$ DISTRIBUTE A(*,BLOCK((/3,7,5/))
```

HPF-2: Distribution of Derived Type Components

HPF2 allows to define a distribution for a user-defined type. Variables of that typ are distributed automatically.

```
TYPE Grid_Variables
   REAL, ARRAY(1:50,1:100,1:50) :: rho, p
   !HPF$ DISTRIBUTE (BLOCK, BLOCK, *) :: rho, p
END TYPE

TYPE(Grid_Variables) :: A, X
```
Communication: Remarks

- **Different Distributions:**
  - Almost all elements are communicated
  - Global communication-pattern
- **Different Array-Size:**
  - Cyclic may help depending on access-pattern
- **Indirect Distributions:**
  - Workaround by using preprocessors and intrinsics
  - Can’t be handled by HPF-1 (⇒ Extension in HPF2)

Minimising Communication

- **BLOCK-Distributions** are optimal for algorithms with nearest-neighbour type locality. Problems can occur from arrays of different size.
- **CYCLIC-Distributions** have optimal load-balancing characteristics but rarely good communication-behaviour.
- **Strides** produce high communication-overhead.
- **Broadcast** (e.g. transpose) is the worst case.
- **Dynamic change** of distributions is expensive.
**HPF-2: Asynchronous I/O**

**Standard**

\[ \text{DO } I = 1, 10, 1 \]

\[ \text{READ (FILE) } A(I,1:1000) \]

\[ \text{CALL COMPUTE ( } A(I,1:1000) ) \]

END DO

**Serial**

\[ \begin{align*}
1 & 1 2 3 4 5 6 7 8 9 10 \\
2 & 3 4 5 6 7 8 9 10
\end{align*} \]

**Async**

\[ \text{READ (FILE, ID=f1) } A(1,1:1000) \]

\[ \text{DO } I = 2, 10, 1 \]

\[ \text{WAIT (ID=f1) } \quad \text{! For Read(A(I-1))} \]

\[ \text{READ (FILE, ID=f1) } A(I,1:1000) \]

\[ \text{CALL COMPUTE ( } A(I-1,1:1000) ) \]

END DO

**Pipelining**

\[ \begin{align*}
1 & 2 3 4 5 6 7 8 9 10 \\
2 & 3 4 5 6 7 8 9 10
\end{align*} \]

**HPF-2: Approved Extensions, Deletions & Changes**

**Extensions**

- DYNAMIC, REALIGN, REDISTRIBUTE
  - Removed from baseline HPF-2 due to complexity of implementation
- HPF_SPMD (HPF_CRAFT)
  - Worksharing-model based on Cray’s CRAFT programming model

**Deletions & Changes**

- Mapping together with sequence-association
- Mapping-change at subroutine-boundaries only allowed with explicit interface
- Other changes in the Spirit of FORTRAN-90 and FORTRAN-2000
Exercise 6: Multiplication: sparse matrix * vector

- Multiplication of a sparsely populated matrix A with a vector B.
- N = 500000 rows and columns
- M = 4 Elements per row
- Data structure:
  \[
  \text{INTEGER}(N, M) :: A \\
  \text{INTEGER}(N, M) :: \text{Column}_\text{Idx}
  \]
  !Column indices of each element
- \[ C(I) = \sum_{J=1}^{M} (A(I,J) \times B(\text{Column}_\text{Idx}(I,J))) \]

Exercise 6: Multiplication: sparse matrix * vector (continued)

- Open exa6_sparse_matrix.f90
- Use irregular distributions to parallelize the algorithm
- Compile on T3E
- Execute the program
- Compare output with serial version
- Compare your solution with exa6_sparse_matrix.hpf.f90
Outline

- Introduction
- Data distribution
- Independent loops
- Pure, Reduction
- Shadows, Intrinsic functions
- Kinds of distribution
- Features of HPF 2.0

Summary
- JaHPF

Granularity, Architecture and Comfort

- Granularity: coarse
- hidden Architecture: complex
- Programming: simple
- Efficiency: high
- Comfort: high
### Usability of HPF in CFD

<table>
<thead>
<tr>
<th>Numerical Method</th>
<th>HPF-1</th>
<th>HPF-2</th>
<th>HPF-2 + Ext.</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>structured</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>High overhead in HPF-1: Missing Shadows</td>
</tr>
<tr>
<td>block-structured</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>HPF-1: Missing mapping of derived types and general block-distributions</td>
</tr>
<tr>
<td>explicit, unstructured</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>High overhead in HPF-1: Missing shadows and indirect mapping</td>
</tr>
<tr>
<td>implicit, unstructured</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>HPF-1: Missing indirect mapping</td>
</tr>
<tr>
<td>FEM</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>HPF-1: Missing indirect mapping</td>
</tr>
<tr>
<td>Adaptive, structured</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>Realign and Redistribute abandoned in HPF-2</td>
</tr>
<tr>
<td>Adaptive, unstructured</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>Missing indirect mapping in HPF-1, Realign and Redistribute abandoned in HPF-2</td>
</tr>
</tbody>
</table>

### Programming-Models for NUMA

- **Message-Passing**
  - Explicit Data-Distribution / Explicit Communication
  - The Standard: MPI
- **Data-/Worksharing**
  - Explicit Data-Distribution / Implicit Communication
  - The Standard: HPF
- **Distributed Shared-Memory**
  - Implicit Data-Distribution / Implicit Communication
  - Actually no Standard
Exercise 7: A heat transfer example

1.) initialization of heat array
only borders contain valid data at startup!

2.) array after n iterations
inner fields conduct the temperature gap only slowly. After many iterations, the deviation is less than EPS

(iMax, kMax)
Exercise 7: A heat transfer example  (continued)

- Open exa7_heat.f90
- Use block distributions and shadow directive to parallelize the algorithm
- Use independent loops and REDUCE statement for “dPhiMax”
- Use NEW clause for temporary variables and ON HOME to define where the inner loop must be executed
- Compile on T3E
- Execute the program on 1, 2, 3, 4 nodes
- Compare output with serial version
- Compare your solution with exa7_heat.hpf.f90

Outline

- Introduction
- Data distribution
- Independent loops
- Pure, Reduction
- Shadows, Intrinsic functions
- Kinds of distribution
- Features of HPF 2.0
- Summary
- JaHPF
HPF/JA 1.0

The Japanese effort to promote high performance Fortran

HPF/JA (Japan Association) 1.0

- extended language specification for HPF
- developed by NEC, Fujitsu, Hitachi + 30 Users
- language specification published in January 1999

HPF problems

Problems in HPF programming

- applicability not sufficient
  - irregular computations
- programmer’s explicit parallelization control limited
  - too much implementation dependent stuff
- programming techniques not established
  - i.e. sharing distributed arrays between procedures
  - parallelizing loops containing procedure calls
HPF extensions overview

HPF/JA

Approved Extensions

• Reduction kind
• Asy. Comm.
• Explicit SHADOW
• FULL SHADOW
• Comm. Sched. Reuse
• ……..

HPF 2.0 (Core)

• Mapped Pointer
• Mapped Derived Type Compo.
• Indirect Distribution

HPF/JA 1.0 - overview

HPF/JA extensions classified into two major purposes

• enlargement of description capability for parallel processing
  – specification of REDUCTION kind

• optimization of communication
  – asynchronous transfer
  – extension of SHADOW directive
  – REFLECT direct
  – extension of HOME clause in ON directive
  – extension of LOCAL clause or directive
  – reuse of communication schedule
HPF/JA 1.0 - reduction kind

- **Reduction in HPF**
  - Reduction variables may appear only in "reduction statements." (i.e. \( S = S + X \))

- **Reduction in HPF/JA**
  - Reduction variables may appear in any statements as long as the user knows "It's reduction."
  - Users specify reduction kind in the REDUCTION clause.
  - FIRSTLOC/LASTLOC for MAX/MIN is supported.

```hpf
!HPF$ INDEPENDENT, REDUCTION:SUM(S), &
!HPF$ REDUCTION:MAXVAL(QMAX:FIRSTLOC(ILOC))
DO I=...
   CALL SUB(SUM)
   IF(QMAX.LT.Q(I)) THEN
   QMAX = Q(I)
   ILOC = I
ENDIF
ENDDO
```

HPF/JA 1.0 - asynchronous communication

- **Overlap communication and computation**
  - A block of assignments (Array assignments, WHERE constructs, FORALL constructs) can be treated as a packet of one-sided communication.
  - Asynchronous communication + WAIT

**Example:**

```hpfj
!HPFJ ASYNCHRONOUS (ID=X) BEGIN
    FORALL (J=1:N) S(I) = T(N-J+1)
!HPFJ END ASYNC
Computation independent of the above communication
!HPFJ WAIT ASYNC (ID=X)
```

---

---
**HPF/JA 1.0 - shadow extensions**

**FULL SHADOW**
- Memory area for a whole array is allocated on all abstract processors.
- Strong point
  - No global to local address translation required
  - No buffer area for remote memory access needed
- Weak point
  - Requires huge memory area
- Useful when memory consumption is not a problem

Example:
```hpf
!HPF$ !HPFJ SHADOW A(*)
```

---

**Explicit shadow**
- Shadow in HPF
  - Shadow elements are not really visible to the programmer, and are managed completely by the compiler.
- Explicit shadow in JAHPF
  - The communication for the SHADOW area can be controlled by REFLECT, EXT_HOME and LOCAL.

Example:
```hpf
!HPF$ DISTRIBUTE A(BLOCK)
!HPF$ SHADOW A(1)
!HPF$ INDEPENDENT
DO J=...
!HPFJ ON EXT_HOME(A(J)) ! Two processors may own
A(J) = func(J) ! Redundant computation
ENDDO
```
HPF/JA 1.0 - REFLECT / LOCAL directive

- REFLECT: Set values into the SHADOW area from its original.
- LOCAL: Assert no communication required
  - RESIDENT: Communication among active processors may be required

Example

```hpf
REAL A(N), B(N)
!HPF$ DISTRIBUTE (BLOCK)::A, B
!HPF$ SHADOW(1)::A
DO I=1,N
  A(I) = ...
ENDDO
!HPF$ REFLECT <object-list>
DO I=2,N-1
  !HPF$ ON HOME(B(I)), LOCAL(A)
  B(I) = A(I-1) + A(I) + A(I+1)
ENDDO
```

HPF/JA 1.0 - ON_EXT Clause

- Remove communications by overlapped execution

Example:

```hpf
REAL A(1000)
!HPF$ DISTRIBUTE (BLOCK) :: A
!HPF$ SHADOW (1) :: A
!HPF$ INDEPENDENT
DO I=1,1000
  !HPF$ ON EXT_HOME(A(I)), LOCAL(A(I))
  A(I) = ...
END DO
```

All the data access can be performed locally.
HPF/JA 1.0 - communication pattern reuse

- Objective
  - Reuse a communication pattern generated in INSPECTOR phase by specifying array access index is unchanged.
  - Original idea from CSCS/Vienna.

Example

```
!HPF$ DISTRIBUTE A(BLOCK) ONTO PROC
DO ...
!HPF$ REUSE_INDEX (.true.) A
  DO I=1,N
     ... = A(IDX(I))
  ENDDO
... ENDDO
```

HPF/JA 1.0 - parallelization patterns and HPF capabilities

<table>
<thead>
<tr>
<th>Patterns</th>
<th>MPI</th>
<th>HPF1.1</th>
<th>HPF2.0 Core</th>
<th>HPF2.0 A/E</th>
<th>HPF/JA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nearest Neighbor</td>
<td>++</td>
<td>-</td>
<td>-</td>
<td>±</td>
<td>++</td>
</tr>
<tr>
<td>Broadcast</td>
<td>++</td>
<td>±</td>
<td>±</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Reduction</td>
<td>++</td>
<td>±</td>
<td>±</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Array Transposition</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Pipelining</td>
<td>++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Irregular (locality)</td>
<td>++</td>
<td></td>
<td></td>
<td>±</td>
<td></td>
</tr>
<tr>
<td>Domain Decompo.</td>
<td>++</td>
<td></td>
<td></td>
<td>++</td>
<td></td>
</tr>
<tr>
<td>Comm. Optimization</td>
<td>++</td>
<td></td>
<td></td>
<td>±</td>
<td></td>
</tr>
<tr>
<td>Easy to Program</td>
<td>++</td>
<td>++</td>
<td>±</td>
<td>±</td>
<td></td>
</tr>
</tbody>
</table>

Blank: Impossible, -: A little support, ±: not sufficient, ++: Good
HPF-Info

- http://www.epcc.ed.ac.uk/epcc-tec/hpf/ (HPF-2 standard, exercises)
- http://www.crpc.rice.edu/HPFF/home.html (HPF-2 misc)
- http://www.tokyo.rist.or.jp/jahpf/present/index.html (JaHPF presentations)
- http://www.pgroup.com/ (Infos about PGHPF compiler)
- http://www.hlrs.de/organization/par/services/models/ (Programming models at HLRS)

HPF-Info

- Books and Standards

  - „High Performance Fortran Language Specification“ by High Performance Fortran Forum
    http://www.epcc.ed.ac.uk/epcc-tec/hpf/

  - „HPF/Ja Language Specification“ by JAHPF
    http://www.tokyo.rist.or.jp/jahpf/