Introduction to Data Parallel Programming

Contents

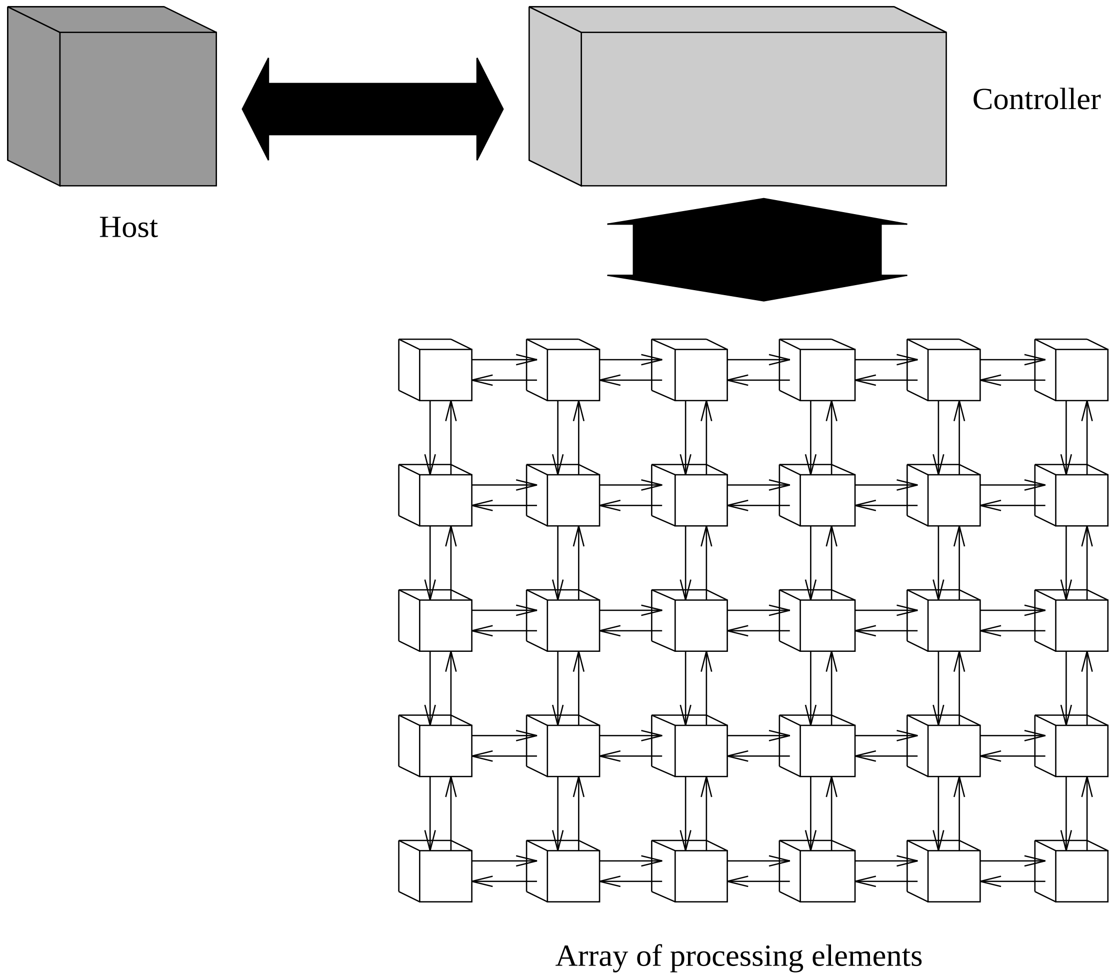
Parallel Computers

- Low cost / high-performance computers
- Serial processors limited by,
  - economics
  - fundamental physical laws
- *Parallel processing:* divide the problem up and solve on many processors
- Aim to keep all processors busy, while minimising communication
- An issue is the difficulty in writing efficient parallel code
- Need a portable and widely accepted standard: HPF?

Parallel Architectures

- Important features:
  - processor
  - memory
  - interconnect
- Typical classification of architectures using Flynn's Taxonomy
- Labels architectures according to instruction stream and data stream e.g.
  - Idealised serial machine: Single Instruction Single Data (SISD)
  - Single Instruction Multiple Data (SIMD)
  - Multiple Instruction Multiple Data (MIMD)

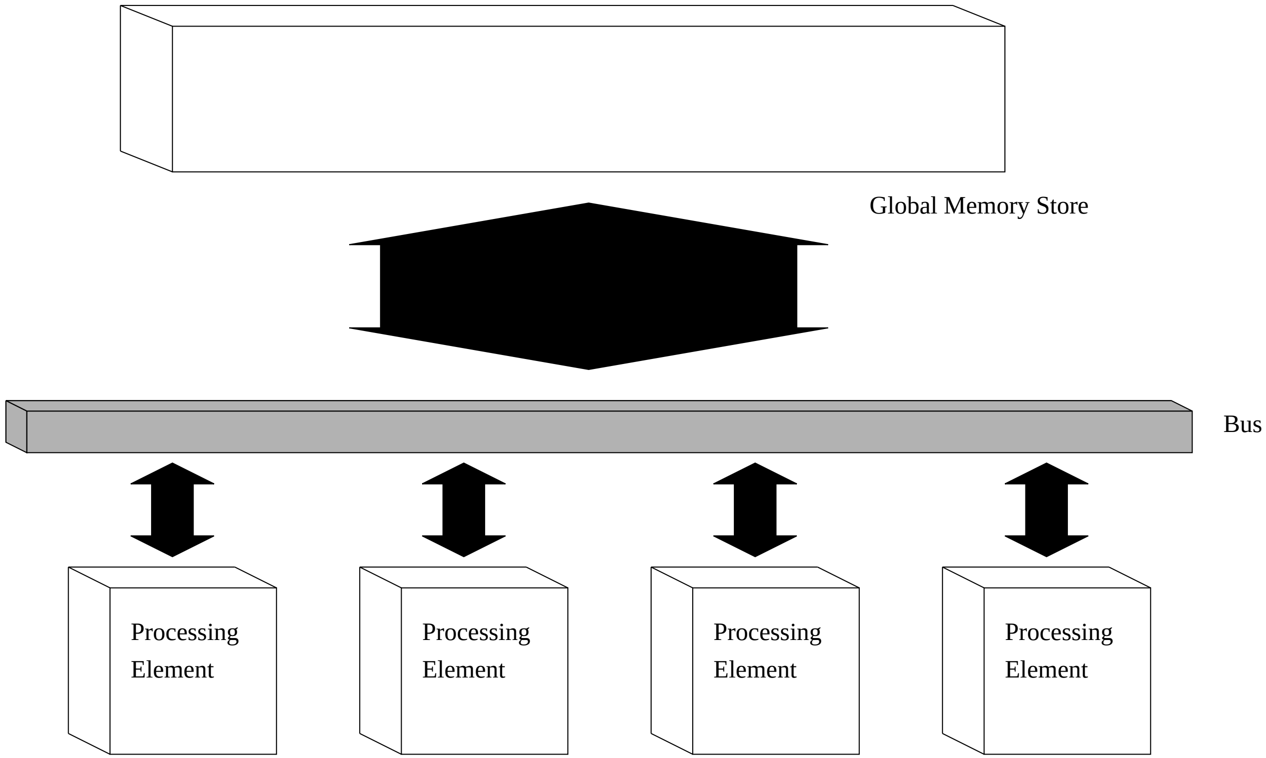
SIMD Architecture



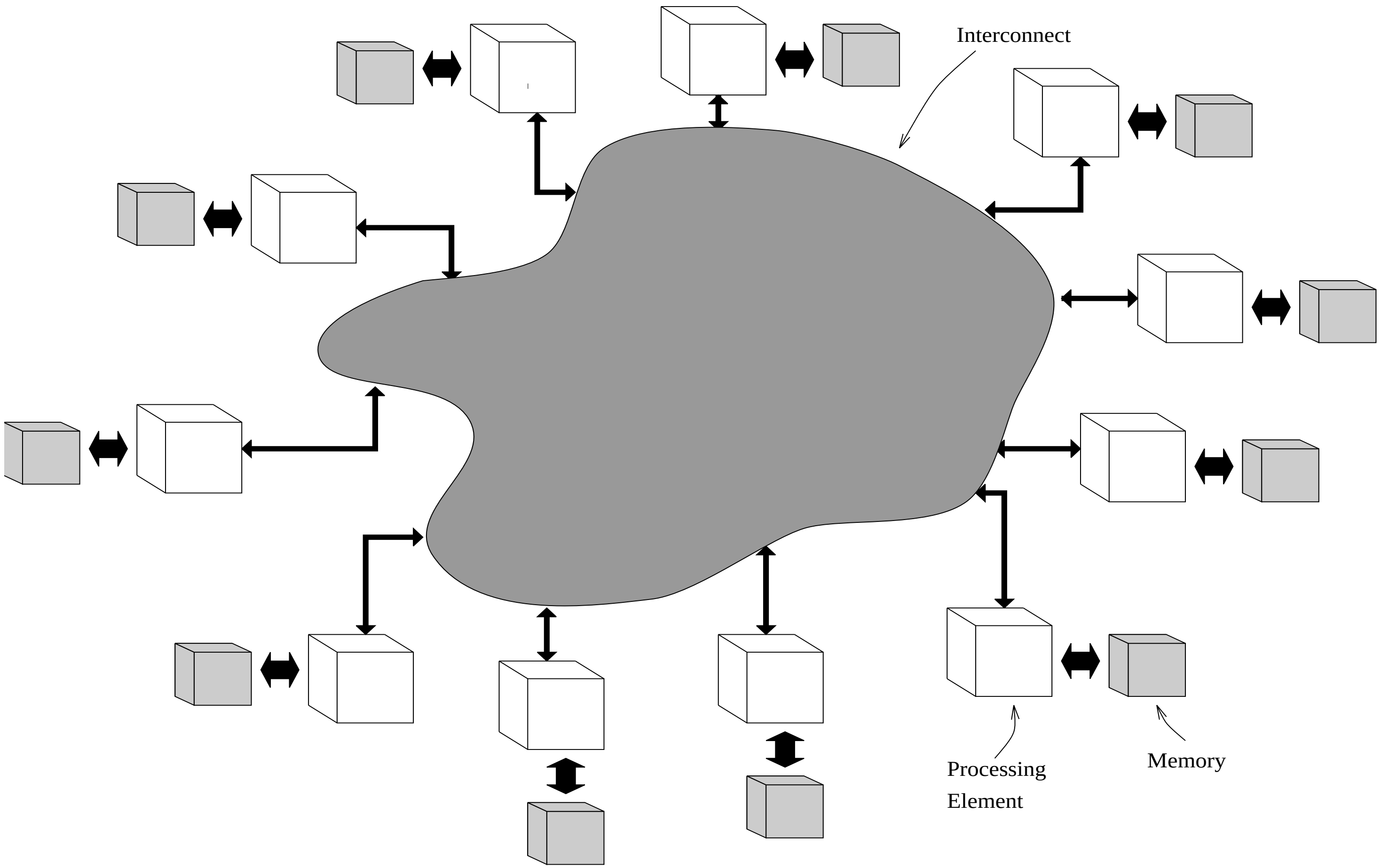
MIMD Architecture

- Multiple Instructions Multiple Data
- Several independent processors capable of executing separate programs
- Typically the processors are not synchronised
- Subdivision of MIMD architectures by relationship between processors and memory
  - SM: SGI Power Challenge, Sequence Balance, Symmetry
  - DM: Meiko Computing Surface, CM-5, workstation clusters
  - VSM: Cray T3D

Shared Memory Architecture



Distributed Memory Architectures



Virtual Shared Memory

- Each processor has some local memory
- Direct access to remote memory by global address space
- Hardware includes support circuitry to deal with remote accesses, allowing very fast communications

Message-Passing Paradigm

- Typically, same program runs on every node: SPMD
  - written in a conventional sequential language
  - all variables are private
  - communicate via special subroutine calls
- Complete control: data distribution and communications
- Trade-off between parallelism and communication
- Very flexible but ...
  - writing and debugging code is difficult
  - programmer needs to keep processors busy while minimising communications
  - avoid “dead-lock”

Data-Parallel Paradigm

- The same transformation is applied to many elements of a data object; *array operations in parallel*
- Program is written in terms of “parallel” constructs added to a sequential language e.g. Fortran or C
- Advisory compiler directives are often provided by the programmer
- Actual distribution of data and communication done by compiler

Data Parallel Programming

- A programming style characterised by:
  - Single-threaded control
  - Global name space
  - Loosely synchronous processes
  - Parallelism implied by operations applied to data
  - Compiler directives
- Manufacturers produce efficient routines for host
- *Transfer low level details from programmer to compiler*
- Encourage wider use of parallelism

Pros and Cons

Advantages: Ease of Use

- Simple to write (no explicit message passing), debug (single thread of control) and port (standard languages + set of intrinsic functions)

Disadvantages: Flexibilty and Control

- Not as flexible: forced to write in DP style, only suitable for certain applications
- Restricted control: distribution of data and work
- Harder to get top performance
- *Reliant on good compilers*

Applications

- Requirements
  - Data must be in large arrays
  - Similar operations on each element
  - Independent operations
  - Problem should be naturally load-balanced
- Examples:
  - lattice physics, e.g. QCD,
  - atmospheric modelling,
  - cellular automata,
  - image processing

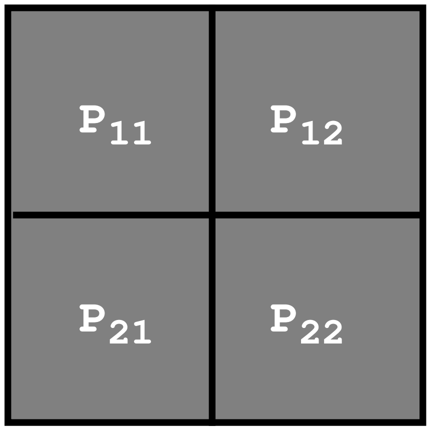
Data Parallel Building Blocks

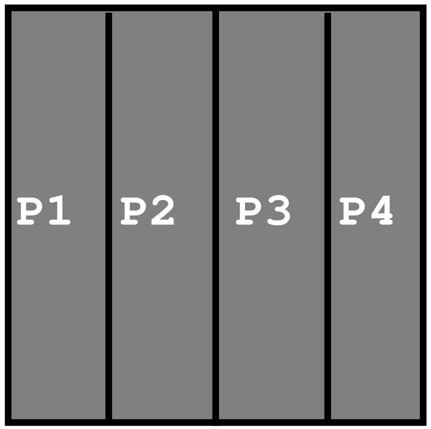
Extend serial languages, such as C or Fortran:

- Data distribution
- Array syntax
- Conditional operations
- Intrinsic functions
  - Shift operations
  - Reduction operations
  - Scan operations
  - Generalised communications
- Libraries

Data Distribution

- Keep processors busy
- Minimise communications
- Data distributions controlled by language constructs/ directives





Elemental Operations

Whole arrays used in expressions

$$\begin{bmatrix} 2 & 12 \\ 8 & 25 \\ 18 & 42 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} * \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}$$

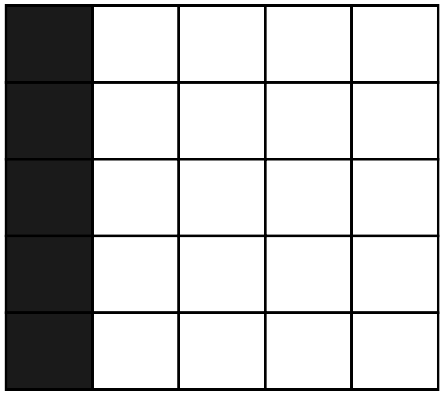
$$c = a * b$$

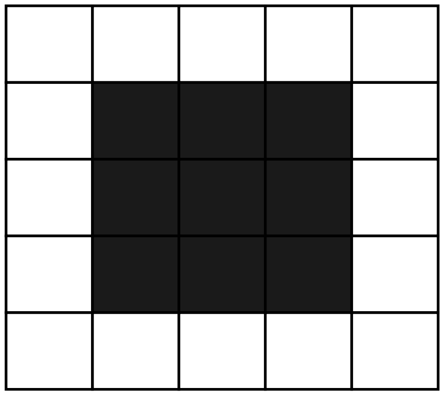
- Mathematical functions on array-valued arguments
- Makes code more compact and readable
- Array operations in parallel

Array Sections

Features to allow some operations to act on only a subset of array elements

Regular patterns specified

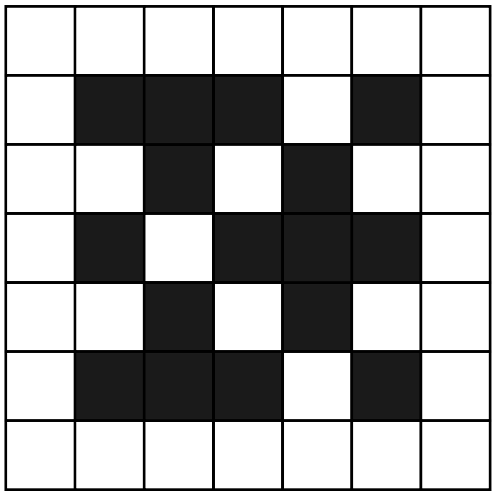




Conditional Operations

Operations on subset of array, subject to some conditional mask

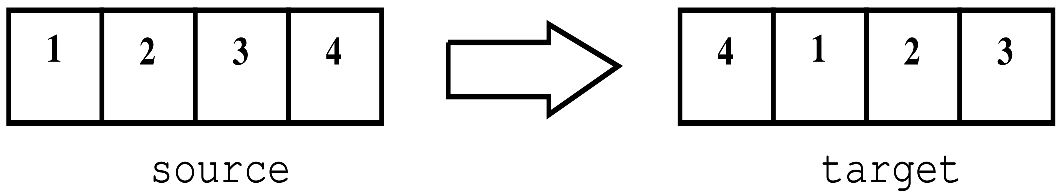
Arbitary patterns specified



Shift Operations

Move whole array in particular direction

- For example, a cyclic shift one place to the right would produce the following result in one dimension



- Also “end-off” shift; provide boundary conditions
- Many efficient parallel algorithms can be implemented in terms of shifts
- Useful for, e.g. image processing, cellular automata

Reduction Operations

Produce one result from many array elements, some global property of the array

- Provided to enable the programmer to, e.g.
  - Find the largest/smallest value in an array
  - Calculate the sum/product of all the elements of an array
  - logical AND, OR, EOR
  - count number of true elements in a logical array
- Very important in data parallel programming and often implemented efficiently

Other Operations

- Useful for constructing parallel logarithms; efficient implementations exist
- Scan Operations: scan through arrays and combine elements to produce some cumulative result (operations include: sum, product, maximum, minimum)
- Generalised Communications: specify how array elements are mapped and combined, to form a result

Data Parallel Languages

- Many data parallel languages implemented:
  - CM Fortran, CRAFT, C*, Fortran D, Fortran-Plus, MP Fortran Vienna Fortran
- Each language expressed data parallel operations differently
- Problems:
  - Too many languages and dialects
  - Languages were machine specific
  - Hence lack of portability (users tied to one manufacturer and prospective users unwilling to learn)
- Need portable standard ... HPF

HPF Forum: a Brief History

- November 1991: DEC organised a discussion meeting at Supercomputing ‘91 --> leads to the formation of the High Performance Fortran Forum (HPFF)
- January 1992: first HPFF meeting in Houston (130)
- During 1992: Working group (40) met to flesh out a standard
- May 1993: HPF Specification v1.0 published
- HPFF consists of an informal working group

Who was involved?

- Representatives of: industry, universities, laboratories
- Not sponsored by official standards bodies, but informal contacts with ANSI X3J3

Alliant Computer Systems Corporation
Amoco
Applied Parallel Research
Archipel
Convex
Cornell Theory Center
Cray Research Inc.
Digital Equipment Corporation
Fujitsu
GMD
Hewlett Packard
IBM
ICASE
Intel Supercomputer
Lahey Computer
Lawrence Livermore National Laboratory
Los Almos National Laboratory
Louisiana State University
MasPar Computer Corporation
Meiko, Inc.
nCUBE inc.
Ohio State University
Oregon Graduate Institute
The Portland Group, Inc.
RIACS
Meiko, Inc.
nCUBE inc.
Ohio State University
Oregon Graduate Institute
The Portland Group, Inc.
RIACS
Rice University
Schlumberger
Shell
SUNY Buffalo
Sun Pro, Sun Microsystems
Syracuse University
Technical University, Delft
Thinking Machines
Unified Technologies
University of Stuttgart
University of Southampton
University of Vienna
Yale University

HPFF Goals

- Define a language that offers:
  - Data parallel programming (Single threaded, global name space, loosely synchronous parallel computation)
  - Top performance with SIMD and MIMD machines with non-uniform memory access costs
  - Code tuning for various architectures
- These goals are addressed in the form of
  - new intrinsics
  - first class language constructs (FORALL)
  - Compiler directives

Subsidiary HPF Goals

- ❑ Portability of existing code
- ❑ Efficient portability of new code
- ❑ Compatibility with existing standards: Fortran 90
- ❑ Simplicity
- ❑ Ease of compiler implementation
- ❑ Availability of compilers in the near future

The HPF Language

- Fortran 90
- Data alignment and distribution features
- Data parallel constructs
- Extended set of intrinsic functions and standard library
- Standard interface with other languages or styles

Subset HPF

- Subset standard defined to promote early implementations
- Fortran 90 features *not* in Subset HPF:
  - free source form, **`POINTER`** and **`TARGET`** attributes,
  - derived data types and generic operators, **`MODULE`**s,
  - new I/O features
- Some features in full HPF which are *not* in Subset HPF:
  - dynamic data mapping directives,
  - the HPF library module and
  - transcriptive data mapping in subroutines

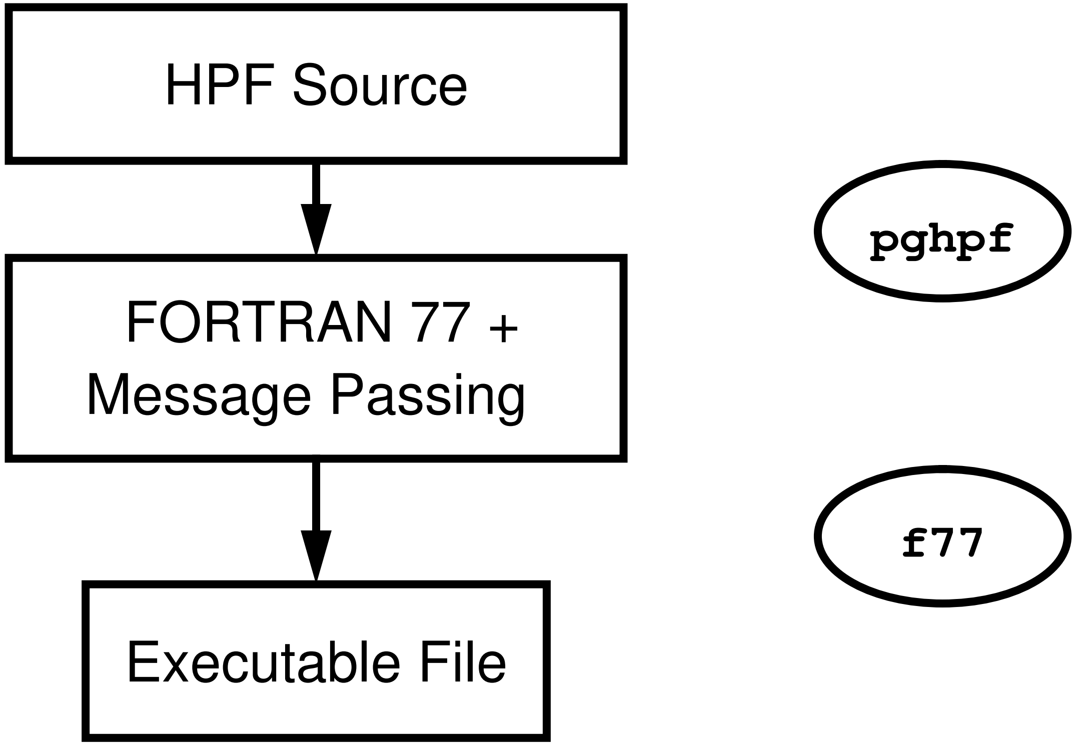
Current Implementations

- Announced Products
  - Applied Parallel Research
  - Digital
  - Kuck and Associates
  - Intel
  - Meiko
  - NA Software
  - The Portland Group
- Announced Effort:
  - ACE, Archipel, Convex Cray Computer, Fujitsu, Hitachi, IBM, Lahey, MasPar, NEC, nCUBE, PSR, TMC

The Portland Group Compiler: pghpf

- Parallelising translator
- Compiles to SPMD FORTRAN 77 + message passing
- Works on a variety of hosts: including Sun4, Cray T3D, SGI
- Supports various communication protocols: PVM, MPI, RPM (optimised PVM)
- *ONLY SUPPORTS SUBSET HPF !*

Compilation Stages



Executable file for the target system

Compiling and Running

- All commands collected in a script

```
hpf [options] root
```

- Options
  - `-help, -h`
  - `-compile`
  - `-run, -nodes n`
  - `-profile`

Running on a Single Node

- Example program: **`hello.hpf`**
- Compile, run commands:

```
hpf -compile hello
hpf -run hello
```

- Profiling:

```
hpf -compile -profile hello
hpf -run -profile hello
```

Running on a Multiple Nodes

- Create file `./MYHOSTS` e.g. working on `young`,

```
young
sutherland
stewart
blair
```

- Run on multiple nodes:

```
hpf -compile -run -nodes 4 hello
```

Summary

- Parallel Architectures: SIMD, MIMD
- Programming Paradigms:
  - Message-Passing
  - Data-Parallel
- Data-parallel building blocks
- High Performance Fortran:
  - History
  - High Performance Fortran Forum (HPFF)
  - HPF Language, Subset HPF
- The Portland Group Compiler: pghpf

Exercise 1: Discontinuous Function

Introduction the the PGHPF compiler

- Write a Fortran 90 code to evaluate the following:

$$0 \leq x < 400: \quad y = x * x * x + C1$$

$$400 \leq x < 1000: \quad y = x * x + C2$$

$$1000 \leq x \leq 10000: \quad y = x$$

with integer vector arrays, x and y, of size [10000] and $C1 = 5$ and $C2 = 10$.Initialise x(i) = i for all i = 1, 10000.

Fortran 90 Features

Overview

- Full HPF includes all of the Fortran 90 language
- Fortran 90 itself is significantly more powerful than FORTRAN 77
- We will concentrate on the Fortran 90 features which are part of subset HPF

What is Fortran 90?

It is FORTRAN 77 with many new features:

- ❑ Array operations
- ❑ Improved facilities for numeric computation
- ❑ User-defined data types
- ❑ New control constructs
- ❑ Facilities for modular data and procedure definitions
- ❑ New source form
- ❑ Optional procedure arguments and recursion
- ❑ Dynamic storage allocation
- ❑ Pointers
- ❑ New intrinsic procedures

Subset HPF

Some F90 features in the subset are:

- FORTRAN 77 (except sequence and storage association)
- Syntax improvements (__, !, lower case, long names)
- **`DO WHILE/END DO, IMPLICIT NONE, INCLUDE`**
- Attributed type declarations
- Array syntax
- New intrinsics
- Interface blocks, optional and keyword arguments

Subset HPF

Some F90 features *not* in the subset are:

- Free source form
- **`POINTER, TARGET`**
- Derived types and operators
- **`MODULE`**s
- **`KIND`** support
- New I/O features

Syntax Improvements

```
      PROGRAM Goodbye_World
!   Say Bye
      print *,'Bye World' ! Say goodbye
      END
```

- Free source format
- _ in names
- 31 character names
- ! comments
- Lower case

New Statements

```
PROGRAM mandelbrot
IMPLICIT none
INCLUDE '/usr/local/include/graphicsf.h'
INTEGER pixels(1000,1000)

do while (.true.)
   pixels = mandel() ! Untyped - compile error
   call X_draw(pixels)
end do
END
```

- New form of `DO`
- `IMPLICIT NONE`
- `INCLUDE`

Attributed Type Specifications

```
FUNCTION mandel(pixels)
IMPLICIT none
INTEGER, PARAMETER :: n=1000
INTEGER, DIMENSION(n,n) :: pixels,mandel
REAL, DIMENSION(n) :: x,y
INTEGER :: call=0

call = call + 1
...
END
```

Array Features

- Array syntax
- **`WHERE`**
- Array-valued functions (intrinsics and user-defined)
- Automatic arrays
- Allocatable arrays
- Assumed shape arrays

Elemental Operations

- Whole arrays may be used in expressions:

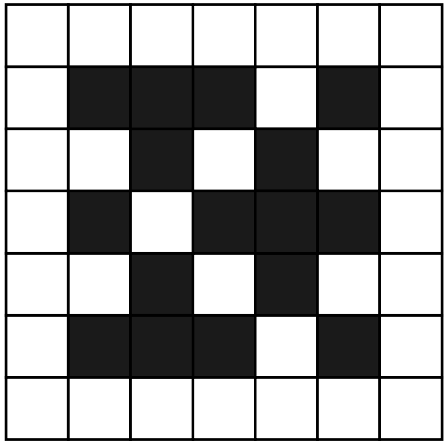
FORTRAN 77	Fortran 90
`REAL A(100), B(100)`	`real a(100), b(100)`
`DO 1 i = 1,100`	
`A(i) = 2.0`	`a = 2.0`
`1 B(i) = B(i)*A(i)`	`b = b * a`

- This tends to make Fortran 90 code more compact

WHERE

- Equivalent to **`'if'`** on each element:

```
real, dimension(7,7) :: a,b
...
where (a > 2.0)
    a = a * 9.0
    b = a / 3.0
end where
```



- There is also an **`else where`** clause

Subscript Triplets and Array Constants

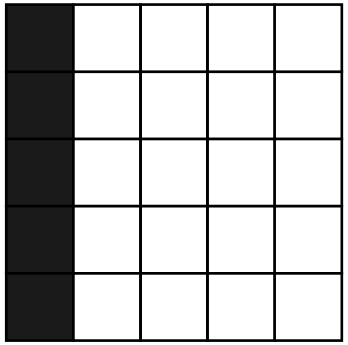
- Array sections and array constants can be defined through triplet notation:

  start index : end index : stride

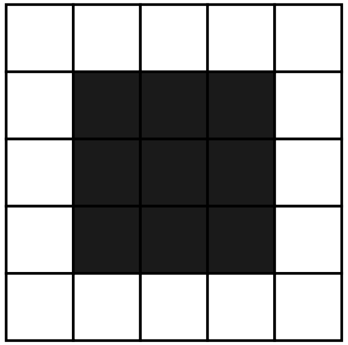
- *For example:*

```
real, dimension(1024) :: a
a(1:1024) = 1.0          ! same as a = 1.0
a(:1024:2) = 2.0         ! a(1)=a(3)=...= 2.0
a = (/ (i,i=1,1024) /)  ! a(1)=1,a(2)=2,...
a = (/ (i,i=1,4096,4) /)! a(1)=1,a(2)=5,...
```

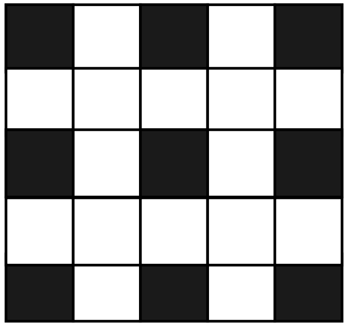
Examples of Array Sections



`a(1,:)`



`a(2:4,2:4)`



`a(1::2,1::2)`

Conformable Arrays

- Arrays used in F90 expressions must be *conformable*
- That is they must have
  - the same rank (number of dimensions), and
  - the same extents in every dimension
- The sequence of extents defines the *shape*

```
real :: a(1024),b(512)
a(1:1024:4) = (/ (i,i=1,256) /) ! This is OK
b = a                           ! This won't compile
```

Declaring Dummy Arrays

Arrays passed to a subroutine can be declared in various ways:

- Explicit Shape
  - Static extents
  - Adjustable extents
- Assumed size
- Assumed shape

Explicit Shape / Assumed Size

Explicit Shape

```
subroutine s1(a,b,c, n,m)
real :: a(100,100)          ! static
real :: b(n,m)              ! adjustable
real :: c(-10:20,n:m)       ! adjustable
```

- The shape of the array is declared explicitly

Assumed Size

```
subroutine flop(a,b,n)
real :: a(*)
real :: b(0:n,0:n,0:*)
```

- This relies on the assumption of a linear memory model

Assumed Shape

```
subroutine s2(a,b,n)
implicit none
real :: a(:,:,:)
real :: b(0:,n:)
```

- The shape of the array is not specified

Local Arrays

Local arrays can be made to conform with dummy arguments that are assumed shape, or have explicit shape with adjustable extents, as well as with static extents

```
subroutine e(a,n,m)
implicit none
integer :: n, m
real :: a(n,m),local(n,m)
```

This is an example of an *automatic* array

Benefits of Automatic Arrays

A program can be made to scale with the size of the data set using automatic arrays

```
program any_size
read *,n,m
call sim( n,m )
end

subroutine sim(n,m)
real a(n,m),b(m)
...
end
```

Array Inquiry Functions

Useful in writing generic routines

- ❑ **`SHAPE(SOURCE):`** Returns a default integer array holding the shape of **`SOURCE`**
- ❑ **`LBOUND(ARRAY [,DIM] ):`** Returns a default integer array holding the lower bounds of **`ARRAY`** or the particular bound if **`DIM`** is present
- ❑ **`UBOUND(ARRAY [,DIM] )`**: Similar to **`LBOUND`** except it returns the upper bound
- ❑ **`SIZE(ARRAY [,DIM] ):`** Returns a default integer holding the size of **`ARRAY`** or the extent of the specified axis if **`DIM`** is present

Array Inquiry Functions: Example

```
subroutine swap(a,b)
implicit none
real, dimension(:) :: a,b
real, dimension(size(a)) :: work
work = a
a = b
b = work
end
```

Interface Block

Allows compiler to know the expected shape, type any number of subroutine arguments when compiling call

Provides:

- A debugging aid
- Better efficiency

What goes into an Interface Block?

- **`FUNCTION/SUBROUTINE`** header
- Dummy argument declarations
- Compiler Directives (if any)
- Intent statements
- Local declarations

Interface Block Example

```
program stress
interface
   subroutine squash(a,n)
      real :: a(n)
   end subroutine
end interface
integer, parameter :: m = 100
real :: q(m)
q=71
call squash(q,m)
```

The compiler can now check calls to **`squash`** for consistency

Intent Attributes

An **`intent`** attribute can reduce unnecessary copying

- if array arguments are not altered in a routine,
- or their initial values are not read

```
subroutine neuron(a,b,c,m,n)
real, dimension(n,n), intent(in) :: a
real, dimension(m,m) :: b,c
intent(out) :: b
intent(inout) :: c
c = sqrt(c)
b = c + sum(a)
end
```

Must use an INTERFACE block in this case

Keyword/Optional Arguments

Procedure arguments may be specified with a keyword and may also be optional:

```
call set_corners(a)
call set_corners(a,value=2)
...
subroutine set_corners(a,value)
integer, intent(in), optional :: value
integer, dimension(:,:) :: a
if (present(value)) then
   a(::size(a,dim=1)-1,::size(a,dim=2)-1)=value
else
   a(::size(a,dim=1)-1,::size(a,dim=2)-1) = 0
end if
end
```

Allocatable Arrays

Arrays can be allocated and deallocated when required:

```
      program simulate
      implicit none
      integer :: n
      integer, dimension(:,:), allocatable :: a
      print *,'Enter n:'
      read *,n
      allocate( a(n,2*n) )

   ! do some work

      deallocate(a)

      end
```

F90 Intrinsic Procedures

- Elemental functions
  - Scalar, F77, functions also accept array arguments:
    `DBLE`, **`SIN`**, **`MAX`**, **`ABS`**, **`LOG`**, ...
  - Each element of the array has the function applied to it
    --- possibly in parallel on a parallel machine
- Bit manipulation functions
  - **`IBITS`**, ...
- Array inquiry functions
  - **`SHAPE`**, **`LBOUND`**, **`SIZE`**, ...*(see earlier)*
- Array transformation functions

Array Transformation Functions

- Array construction functions
  - **SPREAD**, **PACK**, **RESHAPE**, ...
- Vector and matrix multiplication
  - **DOT_PRODUCT**, **MATMUL**
- Reduction functions
  - **SUM**, **MAXVAL**, **ANY**, ...
- Geometric location functions
  - **MAXLOC**, **MINLOC**
- Array manipulation functions
  - **CSHIFT**, **EOSHIFT**, **TRANSPOSE**

Array Construction: SPREAD

```
result = SPREAD(source,dim,ncopies)
```

A:

2	0	9	1

B:

2	2	2
0	0	0
9	9	9
1	1	1

```
B = SPREAD(A,dim=2,ncopies=3)
```

The resulting array has the same shape as **`source`** but extended in dimension **`dim`** to have an extent, **`ncopies`**

Reduction Operations

These typically involve every element of an array $\Rightarrow$ global operation

Finding the sum of all the elements in an array is a global reduction operation

```
real :: a(1024),b(4,1024)
...
scalar = sum(a)
scalar = count(a==0)
a = product(b,dim=1)
scalar = maxval(a,mask=a.LT.0)
```

Logical Reductions

- **`ANY`**: is *any* element in the array true (global **`OR`**)
- **`ALL`**: is *all* the elements in the array true (global **`AND`**)

```
logical a(n)
real :: b(n),c(n)
...
if ( all(a) )        ...
if ( all(b == c) )   ...
if ( any(b < 0.) )   ...
```

Geometric Location: MINLOC, MAXLOC

```
result = minloc(source[,mask])
```

A

1	5	0	9
3	1	7	2
8	9	4	1

P

1	3

```
P = minloc(A)
```

- ❑ Resulting array has a size equal to the rank of `source`
- ❑ If there is more than one extreme value, the first position in array element order is chosen

Circular Shifts

src

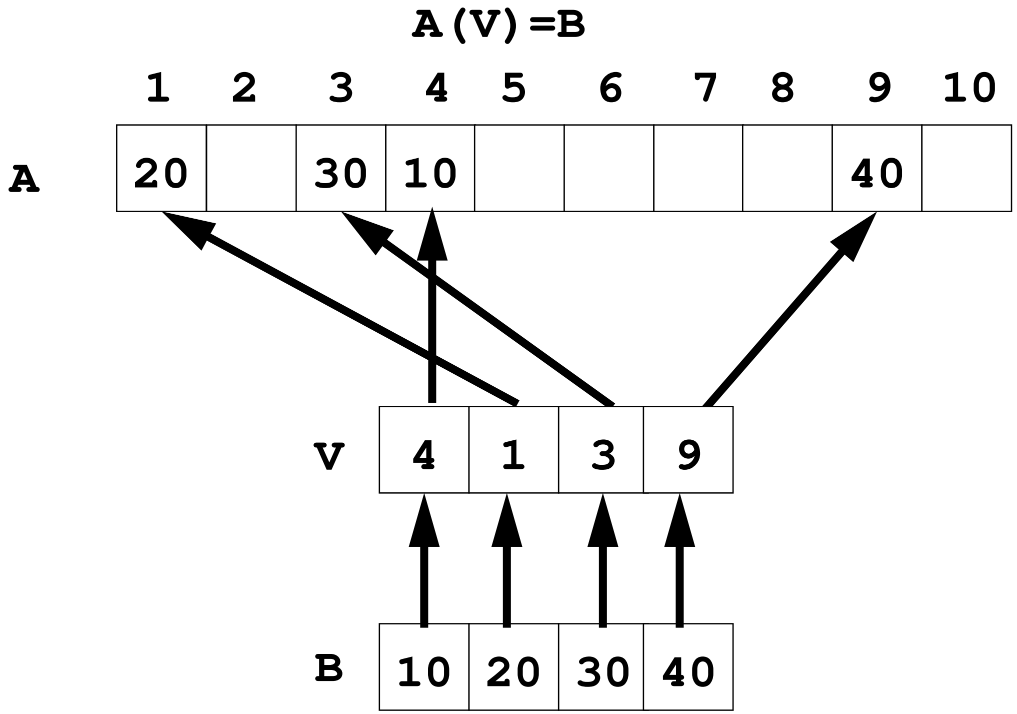
1	2	3
4	5	6
7	8	9

res1

3	1	2
6	4	5
9	7	8

```
res1 = cshift(src,shift=-1,dim=2)
```

Vector Valued Subscripts



- **B(I)** is sent to **A(V(I))** for all **I**
- Collisions are not allowed

Exercise 2: The Game of Life

The Game of Life is a well-known cellular automaton

- In this exercise the world is a 2D grid of cells which have two states: *alive* or *dead*
- Evolution of the system is determined by a set of simple rules, dependent on the state of the neighbouring cells
  - Exactly two alive neighbours; cell maintains its state
  - Three alive neighbours; cell is alive
  - Otherwise; cell is dead
- Write a Fortran 90 code using array syntax to simulate this system

Data Mapping I:

Distributing Data

Data Distribution

- Data Parallel programming involves the processing and manipulation of large arrays
- Processors should (simultaneously) perform similar operations on different array elements
- Each processor has a local memory $\Rightarrow$ large arrays must be distributed across many different processors
- Arrays can be distributed in different ways depending on how they are to be used
- Want to chose distribution which maximises the ratio of local work to communication

DISTRIBUTE Directive

- Compiler directive to specify type of distribution to use
- Specifies the distribution for each dimension of array
- Formats:

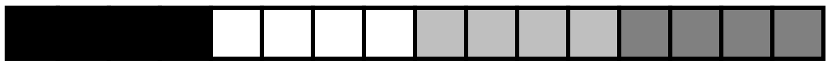
```
!HPF$ DISTRIBUTE a(distribution)

!HPF$ DISTRIBUTE (distribution) :: a, b
```

- **`!HPF$`** is used for all HPF compiler directives --- this is a comment to non-HPF compilers
- *distribution* is a comma-separated list of the distributions for each array dimension

BLOCK Distribution

```
      REAL, DIMENSION(16) :: a
!HPF$ DISTRIBUTE (BLOCK) :: a
```

P1	P2	P3	P4
1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16



- *Example:* regular domain decomposition (Game of Life)

CYCLIC Distribution

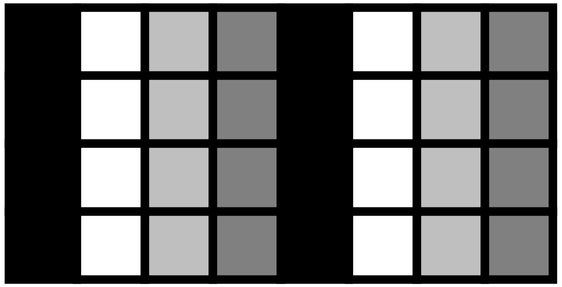
```
      REAL, DIMENSION(16) :: a
!HPF$ DISTRIBUTE (CYCLIC) :: a
```

P1	P2	P3	P4
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

- *Example:* ray tracing; Mandelbrot

Degenerate Distribution

```
      REAL, DIMENSION(8, 4) :: a
!HPF$ DISTRIBUTE (CYCLIC, *) :: a
```



- Second dimension can be called *degenerate*, *serial*, *collapsed*,...
- *Example:* serial dimension is an array of attributes associated with a point on the grid

More General Distributions

- ***CYCLIC(n):*** array split into contiguous chunks of size n and then distributed cyclically

```
        REAL, DIMENSION(16) :: a
!HPF$ DISTRIBUTE (CYCLIC(2)) :: a
```

- ***BLOCK(n):*** similar to ***CYCLIC(n)*** except only 1 chunk per (abstract) processor
  - For above example of 16 elements on 4 processors, $n \geq 16/4$
  - This means **BLOCK(4)**, **BLOCK(5)**, etc. would be OK, while **BLOCK(3)** would be *non-conforming*

Exercise 1: Distributed

```
      PROGRAM disfunc
      IMPLICIT NONE
      INTEGER i
      INTEGER, PARAMETER :: c1=5, c2=10
      INTEGER, DIMENSION(0:16383) :: x, y
! Distribute data and hence work
!HPF$ DISTRIBUTE (CYCLIC) :: x, y

      x = (/ (i, i = 0, 16383) /)

      y(   0:  399)= x(0:399)*x(0:399)*x(0:399)+ c1
      y( 400:  999)= x(400:999)*x(400:999)     + c2
      y(1000:16383)= x(1000:16383)

      END
```

Summary

- HPF has a more sophisticated data distribution model than is outlined in this talk $\rightarrow$ see later
- This is one of the major extensions to Fortran 90
- **`DISTRIBUTE`** directive indicates type of distribution appropriate for each array dimension
- Data distribution helps compiler with work sharing
- Choosing the appropriate distribution can make a significant difference to performance

Exercise 3: Life Revisited

Distributing the Arrays

- Return to the Game of Life example
- Include the data mapping directives required to distribute the arrays
- Compile and run the code on four nodes
- Compare the results when the distribution is changed

HPF Parallel Features

Contents

- The **`FORALL`** Statement
- The **`INDEPENDENT`** Directive
- The **`NEW`** Clause
- The **`PURE`** Attribute

Data Parallel Statements

- HPF contains several ways to express data-parallelism
  - Array operations: from Fortran 90
  - First-class language constructs: **`FORALL`**, **`PURE`**
  - Compiler directives: **`INDEPENDENT`**
  - New intrinsics
- All focus on fine grain parallelism

FORALL Statement

- Generalisation of Fortran 90 array assignment
- Useful for:
  - assignments based on array index

```
FORALL(I=1:N,J=1:N) A(I,J)=1.0/REAL(I+J)
```

  - expression of irregular data motion

```
FORALL(I=1:N) B(I) = A(I,I)
```

  - whole array operations with user defined functions

```
FORALL(I=1:N,J=1:N) F(I,J)=FORCE(R(I,J),M(I,J))
```

- HPF does not specify order of assignment
- Must give the same result from parallel execution

FORALL Syntax

- the **`FORALL`** statement:

  `FORALL (`*forall-triplet-list* `[,` *scalar-mask*`] )` *assignment*

- the **`FORALL`** construct: *NOT IN SUBSET HPF* !!

  `FORALL (`*forall-triplet-list* `[,` *scalar-mask*`] )`
      forall-body
  `END FORALL`

- Example:

```
FORALL(I=1:N,J=1:N,A(I,J).NE.0.0) B(I,J)=1.0/A(I,J)
```

Example FORALL Statements

- ❑ Assignment based on array index:

HPF:

```
FORALL (I=1:N,J=1:N) A(I,J)=1/REAL(I+J)
```

Fortran 90: array syntax:

```
A=1/REAL(SPREAD((/(I,I=1,N)/),DIM=2,NCOPIES=N) &
    + SPREAD((/(J,J=1,N)/),DIM=1,NCOPIES=N))
```

- ❑ Masked assignment:

HPF:

```
FORALL(I=1:N,X(I).NE.0.0) Y(I) = 1.0 / X(I)
```

Fortran 90:

```
WHERE(X(1:N) .NE. 0.0) Y(1:N) = 1.0/X(1:N)
```

❑ Irregular data motion

HPF:

```
FORALL(I=1:N) A(i,i) = Y(i)
```

Fortran 90:

```
A=RESHAPE(MERGE((/Y(i),(0,j=1,N),i=1,N-1),Y(N),&
PACK(A,.TRUE.),(/((i==j,j=1,N),i=1,N)/)),(/N,N/))
```

❑ Array assignment

HPF:

```
FORALL(I=2:N-1) X(I) = X(I-1) + X(I) + X(I+1)
```

Fortran 90:

```
X(2:N-1) = X(1:N-2) + X(2:N-1) + X(3:N)
```

Fortran 77:

```
DO 1, I = 2, N-1 !!! not the same !!!
    X(I) = X(I-1) + X(I) + X(I+1)
END DO
```

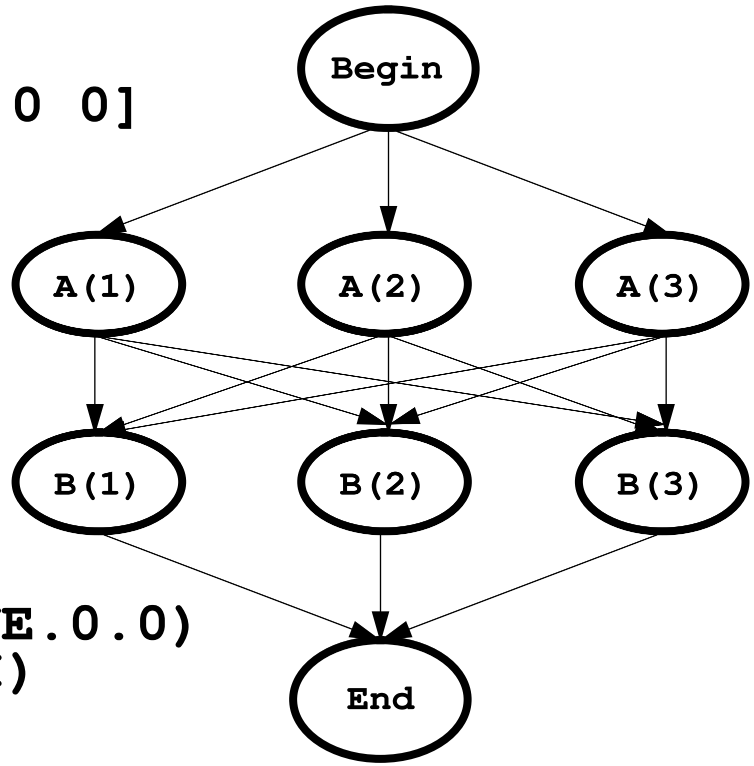
Execution of a FORALL

- Executed in four stages:
  - Compute valid index set
  - Define active index set
  - Evaluate right hand sides
  - Assign to left hand sides
- Semantics same as for F90 array assignment; `FORALL` is not a generalised parallel loop
- Elements at each stage can be evaluated/assigned in any order
- Each stage must be fully completed before the next
- Index is undefined outside the `FORALL` statement

Visualising a FORALL

- Precedence Graphs: arrows indicate order of execution stages; no defined order of iterations; synchronisation between steps

A has value **[1 4 2 0 0 0]**




```
 FORALL(I=1:6, A(I).NE.0.0)
&    B(I) = 1.0 / A(I)
```

Restrictions on a FORALL

- Can't do multiple assignments to same element

```
FORALL(I=1:N) S = X(I)
```

- Expressions in forall-header cannot be affected by the evaluation of other expressions in the header
  - range of index variables can't depend on other index variables, e.g.

```
FORALL(I=1:N,J=1:I) A(I,J)=A(I,J)/A(I,I)
! NON-CONFORMING
! use the following instead,
FORALL(I=1:N,J=1:N,J.LT.I) A(I,J)=A(I,J)/A(I,I)
```

INDEPENDENT Directive

Additional information about execution of a FORALL statement or DO loop

- Promise made by programmer to compiler:
  - `DO` loop: no iteration affects any other
  - `FORALL` statement: index operations do not interfere
- Should not alter meaning of the code; advisory nature only
- Order of computations is not significant; can be safely done in parallel
- If assertion is false, results are not defined by HPF

Examples of INDEPENDENT

- Must precede a **DO** loop or **FORALL** statement
- Express obvious parallelism

```
!HPF$  INDEPENDENT
       DO I = 2, N-1
          X(I) = Y(I-1) + Y(I) + Y(I+1)
       END DO
```

- Inform the compiler of non-obvious parallelism

```
!HPF$  INDEPENDENT
       DO I = 1, N
          X(INDX(I)) = Y(I)
       END DO
! NON_CONFORMING if INDX has repeated values
```

Restrictions on INDEPENDENT

- Read / Write restrictions:
  - if a data object is written in one iteration, it cannot be written or read in another (read/writes also occur in I/O and data remapping)
  - examples where **`INDEPENDENT`** *cannot* be used:

```
DO I = 2, N
   A(I) = A(I-1)
END DO
```

```
DO I = 1, N
   S = S + A(i) * B(I)
END DO
```

- No control dependence or branching
  - if a statement starts, it continues to completion; whether an iterations executes should not depend on the execution of another

```
SEARCH:   DO I = 1, N ! Not INDEPENDENT
             IF( A(i) .EQ. 10 ) EXIT SEARCH
          END DO
```

The NEW Clause

Allow “temporary” variables in an INDEPENDENT loop

- Modifies meaning of **`INDEPENDENT`** directive
- Overcomes some of the restrictions
- Allows for temporary variables, “private” to a loop
- Asserts that the value of the **`NEW`** variable not needed at the start and end of each iteration (the value is undefined outside the loop)
- Syntax (only applies to **`DO`** loops)

`!HPF$ INDPENDENT, NEW (`*variable-list*`)`

NEW Clause: Examples

- Temporary variables:

```
!HPF$  INDEPENDENT, NEW (T)
       DO I = 1, N
          T = A(I) * B(I)
          A(I) = SQRT(T)
       END DO
```

- Nested Loops:

```
!HPF$  INDEPENDENT, NEW (I)
       DO J = 1, N
          DO I = 1, N
             A(J) = A(J) + B(I,J)
          END DO
       END DO
```

The PURE Attribute

Safe functions for use in FORALL statements

- Not a parallel construct, used to generalise **`FORALL`** for use with user defined functions
- **`PURE`** guarantees that a function
  - does not alter the input data (**`INTENT(IN)`** input data is implicit)
  - has no other side effects (e.g. modification of global data, data mapping of arguments, external I/O)
- All intrinsic functions are **`PURE`**
- Many constraints on a **`PURE`** function
- *NOT IN SUBSET HPF !!*

Example of a PURE Function

Mandelbrot Set Example / Exercise

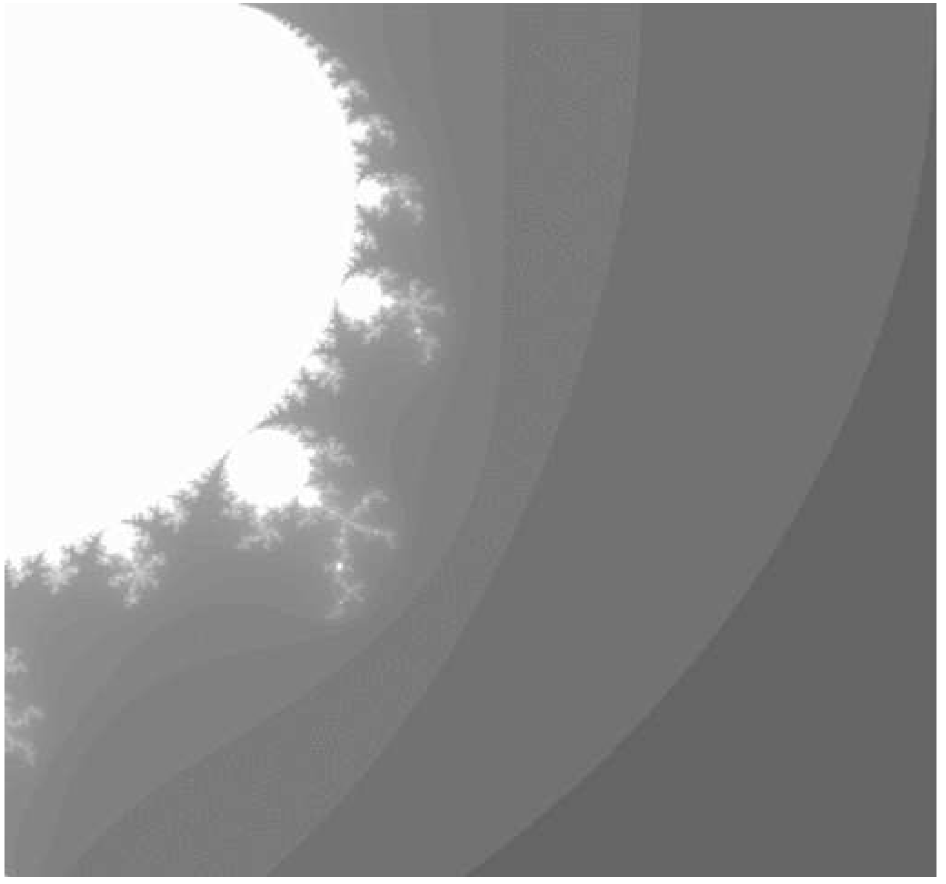
```
      FORALL(i = 1:N, j= 1:N)
    &     COLOUR(i,j) = MANDELBROT(Z_INITIAL(i,j))
      ...
      PURE INTEGER FUNCTION MANDELBROT(Z)
          COMPLEX, INTENT(IN) :: Z
          COMPLEX             :: TMP
          INTEGER             :: k, COLOUR
          TMP = Z
ITERATE:  DO k = 0, 256
              IF (ABS(TMP) < 2.0) EXIT ITERATE
              TMP = TMP*TMP - Z
          END DO
          MANDELBROT = k
      END FUNCTION
```

Summary

- HPF contains new constructs for fine grain parallelism
- **`FORALL`** statement: generalised array assignment
- **`INDEPENDENT`** directive: guarantee of parallel execution without interference between iterations
- **`NEW`** clause: allow for temporary variables in **`INDEPENDENT`** loops
- **`PURE`** attribute: produce user defined functions which are safe to use in **`FORALL`** statements

Exercise 4: The Mandelbrot Set

Iteration of a complex function on a 2D grid



Data Mapping II:

Model used by HPF

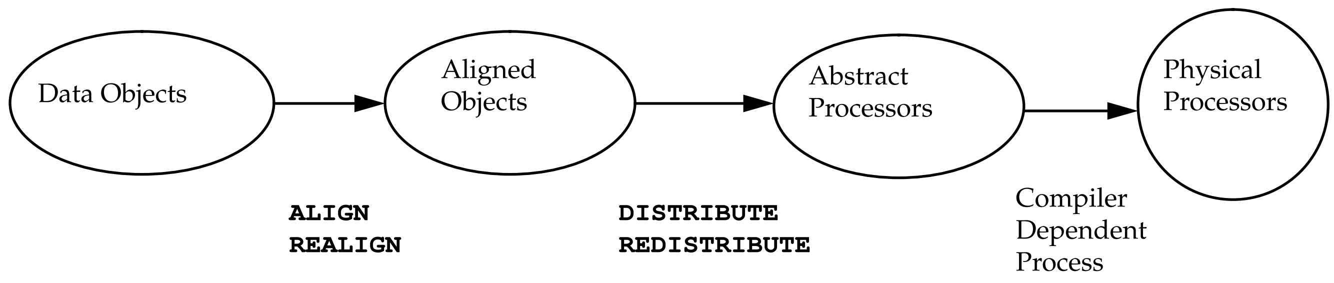
Contents

- Overview of data mapping model
- Data mapping directives
  - **`PROCESSORS`**
  - **`DISTRIBUTE`**
  - **`ALIGN`**
  - **`TEMPLATE`**
- Dynamic data mapping

Motivation

- Current compilers are not sophisticated enough to automatically distribute arrays over processors
- HPF contains various directives to control data mapping and help compiler
- $\Rightarrow$ can reduce communication overhead and produce more effective code
- Data mapping is key feature of HPF

Data Mapping Model



- Two stage process
  - *Alignment:* attribute of data set describing interactions between objects
  - *Distribution:* partitioning of index space onto abstract processors
- Alignment + distribution = mapping
- Describe HPF model from *right to left*

PROCESSORS Directive

- The **`PROCESSORS`** directive specifies the shape of the grid of abstract processors, eg,

```
!HPF$ PROCESSORS example(6,2)
```

  specifies a 6x2 array of 12 abstract processors labelled **`example`**

- If architecture has a master plus slave processors, it can be useful to specify a scalar processor arrangement

```
!HPF$ PROCESSORS scalar
```

  Also useful for indicating data which should *not* be distributed

Rules for PROCESSORS

- Defines arrangement of set of *abstract* processors; mapping to *physical* processors is not specified in HPF
- Processor arrangement name cannot be same as variable name, named constant or procedure name
- Corresponding elements of arrangements with same shape refer to same abstract processor, eg,

```
!HPF$ PROCESSORS P1(2)
!HPF$ PROCESSORS P2(2)
```

  guarantees that **`P1(1)`** and **`P2(1)`** refer to same abstract processor

- If two objects are mapped to same abstract processor, then objects are mapped to same physical processor

Processor Inquiry Functions

- ***NUMBER_OF_PROCESSORS:*** intrinsic function which, at run time, returns actual number of physical processors
- ***PROCESSORS_SHAPE:*** returns arrangement of physical processors
- These can be used in specification of abstract processor arrangement, eg,

```
!HPF$ PROCESSORS grid(NUMBER_OF_PROCESSORS()/3,3)
```

- HPF compilers are *required* to accept any declaration where the product of extents in the processor arrangement is equal to ***NUMBER_OF_PROCESSORS()***

DISTRIBUTE Directive

- Partitions index space onto abstract processors
- Indicates type of distribution appropriate for each array dimension
- Want to chose distribution which maximises the ratio of local work to communication
- **`ONTO`** clause indicates abstract processor arrangement
- Syntax Examples:

```
!HPF$ DISTRIBUTE a(BLOCK,BLOCK) ONTO procs

!HPF$ DISTRIBUTE (CYCLIC,*) ONTO grid :: b
```

Formal syntax is in notes

DISTRIBUTE ONTO

- Rank of abstract processor grid must be equal to number of distributed axes of array
- For example, with

```
!HPF PROCESSORS line(8)

!HPF DISTRIBUTE (CYCLIC) ONTO line :: vector
!HPF DISTRIBUTE (*,BLOCK) ONTO line :: matrix
```

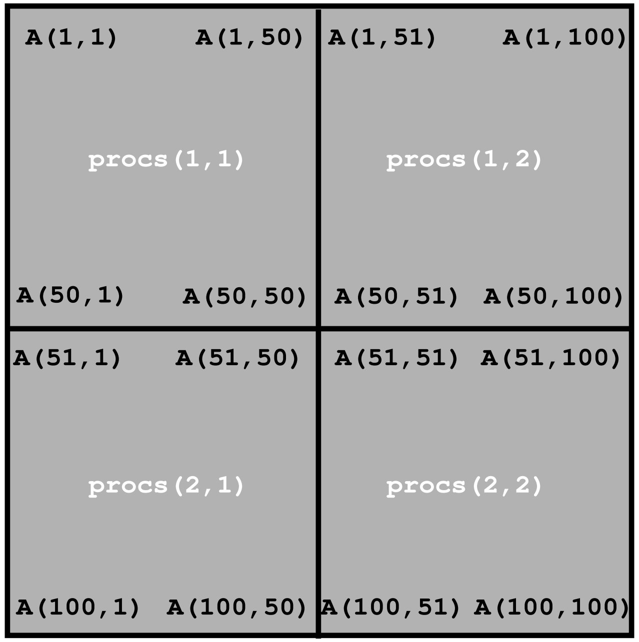
both **`DISTRIBUTE`** directives are OK, but

```
!HPF DISTRIBUTE (CYCLIC,BLOCK) ONTO line :: matrix
```

would be HPF *non-conforming*

DISTRIBUTE Example

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




```
procs(1,1):A(1:50,1:50)
procs(1,2):A(1:50,51:100)
procs(2,1):A(51:100,1:50)
procs(2,2):A(51:100,51:100)
```

Other examples are in the notes and a previous talk

ALIGN Directive

- Used to specify relationship between data objects, eg, arrays $\rightarrow$ reduced communication
- When 2 arrays are used together, could simply specify the same distribution for each, eg,

```
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO procs::board
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO procs::neighbours
```

- While this may be OK for a simple example, in general, it is clearer to use,

```
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO procs :: board
!HPF$ ALIGN WITH board :: neighbours
```

- Guarantees required alignment

Image: F77 Code

```
  REAL centre(N,N), image(N+2,N+2)
  ...
  DO i = 1, N
    DO j = 1, N
      centre(i,j) =
& -image(i  ,j)-image(i  ,j+1)    -image(i  ,j+2)
& -image(i+1,j)-image(i+1,j+1)*8.0-image(i+1,j+2)
& -image(i+2,j)-image(i+2,j+1)    -image(i+2,j+2)
     END DO
   END DO
```

- F77 update algorithm on regular grid using *halo*
- Note that (i,j)th component of `centre` is obtained from 3x3 square of index pairs around $(i+1, j+1)$ in `image`

Image: Data Mapping

- Can distribute **image** over, say, 4 processors

```
!HPF$ PROCESSORS square(2,2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square::image
```

- Appropriate alignment for **centre** is

```
!HPF$ ALIGN WITH image(i+1,j+1) :: centre(i,j)
```

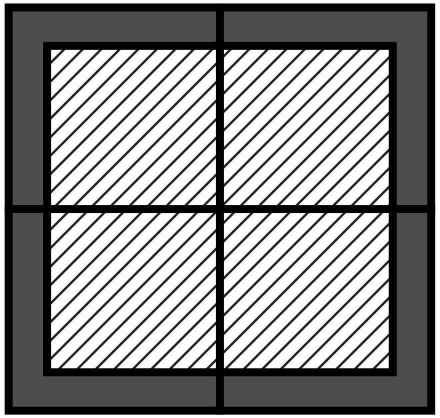


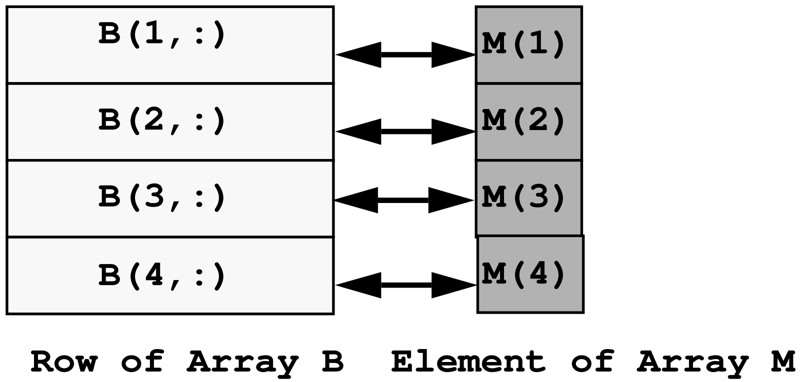
Image: HPF Code

```
      REAL, DIMENSION(N,N) :: centre
      REAL, DIMENSION(N+2,N+2) :: image
!HPF$ PROCESSORS square(2,2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: image
!HPF$ ALIGN WITH image(i+1,j+1) :: centre(i,j)
      ...
      centre(:,:) =

    &  -image( :N,  :N)-image( :N,  2:N+1)    -image( :N,  3:N+2)
    &  -image(2:N+1.:N)-image(2:N+1,2:N+1)*8.0-image(2:N+1,3:N+2)
    &  -image(3:N+2,:N)-image(3:N+2,2:N+1)    -image(3:N+2,3:N+2)
```

Collapsing Axes using ALIGN

```
REAL B(4,3), M(4)
!HPF$ ALIGN B(i,*) WITH M(i)
```

- Will result in the following alignment,



- ...where double arrows indicate that objects are stored on the same processor, i.e., are aligned in memory

Collapsing II

- Each row of **B** is aligned with a single element from **M**
- This means that

```
M = B(:, 1) + B(:, 2) + B(:, 3)
```

...requires no communication

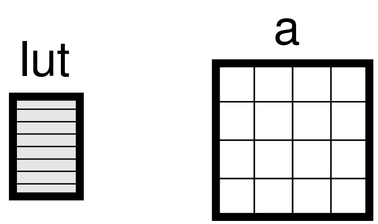
- Equivalent syntax

```
!HPF$ ALIGN B(:, *) WITH M(:)
!HPF$ ALIGN B(i, k) WITH M(i)
```

Formal syntax of ALIGN is in notes

Alignment for Replication

- `ALIGN` can be used to effectively replicate heavily used arrays, such as lookup tables
- Any actual replication will depend on distribution of target
- Trade off between extra storage requirements and possible savings in communication
- Example:

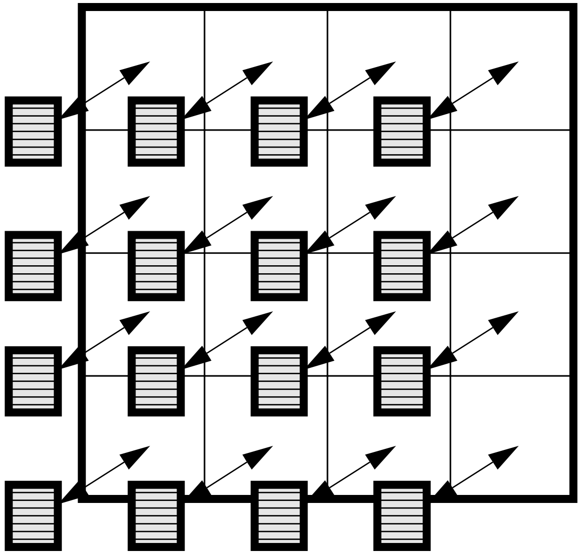



```
REAL a(4,4)
REAL lut(8)
...
FORALL (i=1:4, j=1:4) a(i,j) = a(i,j) + lut(i+j)
```

Replicated Lookup Table

- Align all of lookup table with each element of *a*

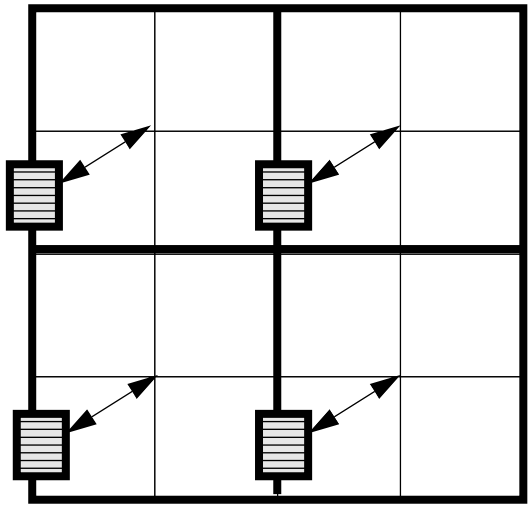
```
!HPF$ ALIGN WITH a(*,*) :: lut(*)
```



Replicated Lookup Table II

- Presumed implementation with distribution

```
!HPF$ PROCESSORS square(2,2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: a
```



Example: Matrix Multiplication

- Assume **A**, **B** and **C** are 2-dimensional NxN arrays
- $C = AB \Rightarrow C(i,j)$ is produced by multiplying the *i*th row of **A** and the *j*th column of **B**
- $\Rightarrow$ store each row of **A** on single processor
  and each column of **B** on single processor
- Could use DISTRIBUTE

```
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: C
!HPF$ DISTRIBUTE (*,BLOCK) ONTO line :: A
!HPF$ DISTRIBUTE (BLOCK,*) ONTO line :: B
```

Matrix Multiplication II

- Only using **DISTRIBUTE** can cause problems:
  - No specific relationship between **A**(**B**) and **C**
  - Cannot indicate replication
  - Specifying all the distributions of a set of arrays $\rightarrow$ clumsy code
- Better to use **ALIGN**, ***eg***,

```
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: C
!HPF$ ALIGN A(i,*) WITH C(i,j)
!HPF$ ALIGN B(*,j) WITH C(i,j)
```

- Guarantees that rows of **A** and columns of **B** are stored on all processors where they are required

Matrix Multiplication Code

```
      PROGRAM ABmult
      IMPLICIT NONE
      INTEGER, PARAMETER :: N = 100
      INTEGER, DIMENSION (N,N) :: A, B, C
      INTEGER :: i, j

!HPF$ PROCESSORS square(2,2)
!HPF$ DISTRIBUTE(BLOCK,BLOCK) ONTO square :: C

!HPF$ ALIGN A(i,*) WITH C(i,j)
!    replicate copies of row A(i,*)
!    onto processors which compute C(i,j)

!HPF$ ALIGN B(*,j) WITH C(i,j)
!    replicate copies of column B(*,j))
!    onto processors which compute C(i,j)
```

```
      A = 1
      B = 2
      C = 0


      DO i = 1, N
         DO j = 1, N
!           All the work is local due to ALIGNs
            C(i,j) = DOT_PRODUCT(A(i,:), B(:,j))
         END DO
      END DO

      WRITE (*,*) C

      END
```

TEMPLATE Directive

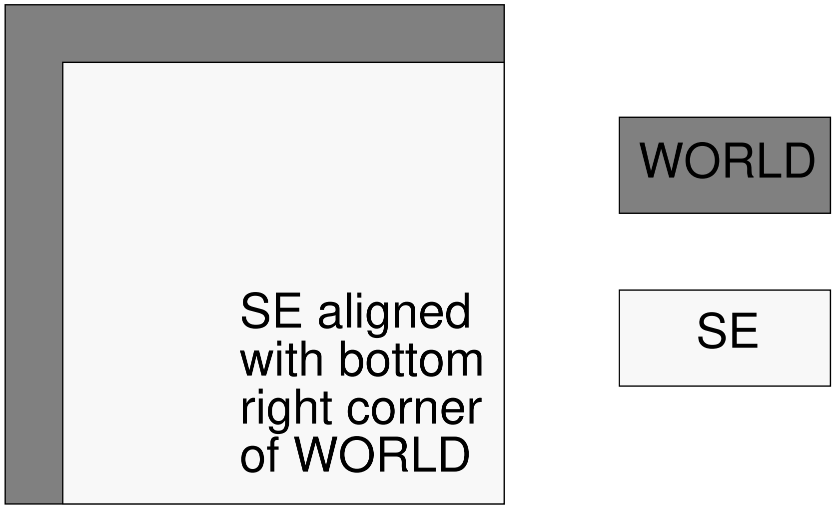
- Provides index space for alignment without actually allocating memory

```
!HPF$ TEMPLATE t(100)
```

- Can be useful to express more complicated alignments
- *Example:* assume we wish to align 4 smaller arrays (**`NW`**, **`NE`**, **`SW`**, **`SE`**) of size $N \times N$ with the four corners of a larger array (**`WORLD`**) of size $(N+1) \times (N+1)$, *BUT* without allocating the larger space

Example: WORLD

```
!HPF$ TEMPLATE, DIMENSION(N+1,N+1) :: WORLD
        REAL, DIMENSION (N,N) :: NW, NE, SW, SE
!HPF$ ALIGN NW(i,j) WITH WORLD(i  , j  )
!HPF$ ALIGN NE(i,j) WITH WORLD(i  , j+1)
!HPF$ ALIGN SW(i,j) WITH WORLD(i+1, j  )
!HPF$ ALIGN SE(i,j) WITH WORLD(i+1, j+1)
```



Dynamic Data Mapping

- Previous directives are all declarations and must appear in specification part of program module
- HPF also contains directives which allow data mapping to change during the course of program execution
  - ***REALIGN***: changes target to which array is aligned
  - ***REDISTRIBUTE***: changes distribution of array and all arrays aligned with it
- Dynamic forms are similar to static equivalents, except
  - They appear in executable part of program module
  - Arrays which appear in these directives must be declared with the **DYNAMIC** attribute
- These are expensive and are *not* in the HPF Subset

DYNAMIC Example

```
!HPF$ DISTRIBUTE (CYCLIC) :: D1
!HPF$ DYNAMIC, DISTRIBUTE (BLOCK) :: D2
!HPF$ DYNAMIC, ALIGN WITH D2 :: A1, A2
       ...
!HPF$ REDISTRIBUTE (CYCLIC(7)) :: D2
       ...
!HPF$ REALIGN A2 WITH D1
```

- The **`REDISTRIBUTE`** directive changes the distribution of ***`D2`*** and hence of ***`A1`*** and ***`A2`***
- The **`REALIGN`** directive changes the alignment of ***`A2`*** from ***`D2`*** to ***`D1`*** and therefore alters ***`A2`***'s mapping

Summary

- HPF has a two stage process for mapping data:
  - **`ALIGN`**: indicates relationships between data objects
  - **`DISTRIBUTE`**: partitions index space onto abstract processors
- This process provides a powerful and flexible method for describing an efficient data distribution
- The mapping to physical processors is compiler dependent
- HPF also includes directives for dynamic mapping of data, but these are *not* supported in the subset

Exercise 5: Birthdays

Using replicated look-up tables in HPF

- Find the day of the year on which any given individual's birthday falls within a particular (non-leap) year
- Begin with the code segment in the notes which defines the data structures and data initialisation
- Complete the program to calculate the number of days from the begining of the year
- HINT: Firstly decide which arrays should be distributed and how (use both `DISTRIBUTE` and `ALIGN`)

Data Mapping and Procedure Arguments

Contents

Introduction

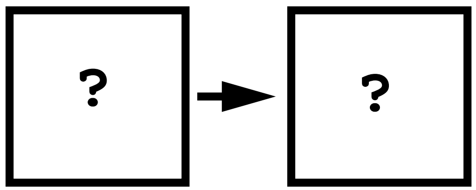
- Subroutine calls require a correspondence between actual arguments and dummy arguments.
- Why is data mapping of dummy arguments tricky?
  - sometimes subroutine needs data with a specific distribution (remapping required)
  - sometimes subroutine does not need to know the distribution of the data (remapping not required)
  - sometimes programmer knows the distribution of the incoming data, sometimes not
- No way to pass mapping information to subroutine
- HPF requires that subroutine calls do not permanently remap the data (c.f. dynamic data remapping)
- Large communication cost in remapping data!!

Directives for Data Mapping

- *Prescriptive*: incoming data could have any distribution, but may be remapped
- *Descriptive*: incoming data *will* have a certain distribution
- *Transcriptive*: any distribution accepted and no remapping applied. *NOT IN SUBSET HPF !!*
- Implicit remapping only if distribution directives do not match up
- Remapping not visible to the programmer, except via performance analysis

Directives for Data Mapping

- Four possibilities for dummy argument distribution
  - **`DISTRIBUTE`** directives match
  - **`DISTRIBUTE`** directives do not match
  - **`DISTRIBUTE`** directives do not match, but an **`INTERFACE`** block is included
  - **`DISTRIBUTE`** directive match and an **`INTERFACE`** block is included



Interface Blocks

- Fortran 90 construct to provide an explicit interface for external procedures.
- Contains subroutine header, without the executable statements, (c.f. ANSI C prototypes) e.g.

```
      INTERFACE
         SUBROUTINE MULTIPLY(A, B, C)
            REAL, DIMENSION(:,:) :: A, B, C
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: A
!HPF$ ALIGN WITH A :: B, C
         END SUBROUTINE MULTIPLY
      END INTERFACE
```

- Not always needed: module and internal procedures automatically have explicit interfaces

Prescriptive Data Mapping

- Subroutine requires data with a certain distribution.
- **`DISTRIBUTE`** directive in subroutine tells the compiler to remap the data if necessary

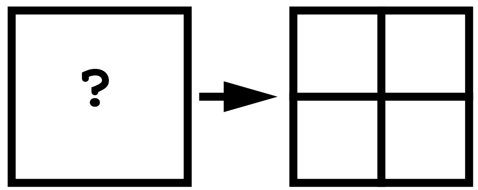
```
!HPF$ DISTRIBUTE C(BLOCK,BLOCK)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: C
```

- Prescriptive subroutine checks the incoming data's distribution (unless **`INTERFACE`** specified) and remaps or not as necessary.

Prescriptive Subroutine

- Subroutine **IT**: performs some calculation which needs the data in **(BLOCK,BLOCK)** distribution.

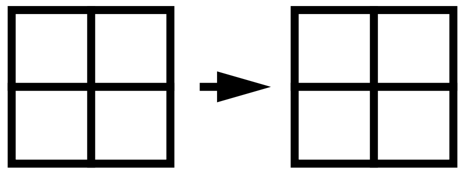
```
      SUBROUTINE IT(C)
      REAL C(10,10)
!HPF$ DISTRIBUTE C(BLOCK,BLOCK)
      ...
      END
```



Prescriptive: Example Program1

- Program **EANY:** no remapping of array **B**
- May be a time penalty for testing whether remapping is required.

```
      PROGRAM EANY
      REAL B(10,10)
!HPF$ DISTRIBUTE B(BLOCK,BLOCK)
      ...
      CALL IT(B)
      ...
      END
```

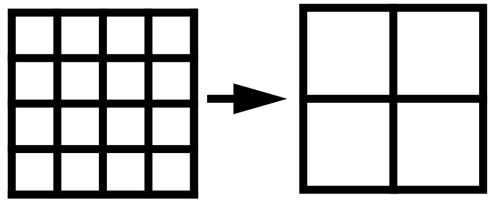


No remapping required

Prescriptive: Example Program2

- Program **MEANY**: array **A** must be implicitly remapped
- On return from **IT**, **A** is remapped again to regain the **(CYCLIC,CYCLIC)** distribution

```
      PROGRAM MEANY
      REAL A(10,10)
!HPF$ DISTRIBUTE A(CYCLIC,CYCLIC)
      ...
      CALL IT(A)
      ...
      END
```

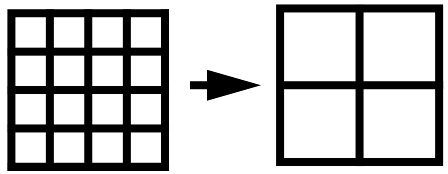


Implicit Remapping

Prescriptive: Example Program3

- Program **MINY:** array **C** is implicitly remapped
- **INTERFACE** block included: remapping required; known at compiler time; attempt to do efficiently

```
      PROGRAM MINY
      REAL C(10,10)
!HPF$ DISTRIBUTE C(CYCLIC, CYCLIC)
      INTERFACE
        SUBROUTINE IT(P)
          REAL P(10,10)
!HPF$     DISTRIBUTE P(BLOCK,BLOCK)
        END SUBROUTINE
      END INTERFACE
      ...
      CALL IT(C)
      ...
      END
```

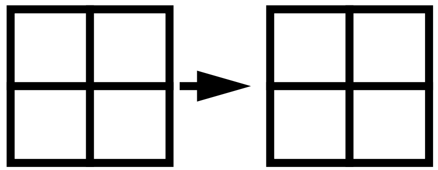


Remapping required
(Known at compile time)

Prescriptive: Example Program4

❑ Program **MO**: same distribution as in **IT** with **INTERFACE** block; known at compile time; no remapping

```
      PROGRAM MO
      REAL D(10,10)
!HPF$ DISTRIBUTE D(BLOCK,BLOCK)
      INTERFACE
        SUBROUTINE IT(P)
          REAL P(10,10)
!HPF$     DISTRIBUTE P(BLOCK,BLOCK)
        END SUBROUTINE
      END INTERFACE
      ...
      CALL IT(D)
      ...
      END
```



No remapping required
(Known at compile time)

Descriptive Argument Mapping

- Assurance that data will only be passed with a particular mapping
- No remapping required
- No test made for a particular data mapping
- **`DISTRIBUTE`** directive for descriptive mapping of a dummy argument,

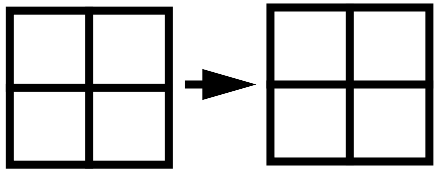
```
!HPF$ DISTRIBUTE S *(BLOCK,BLOCK)
!HPF$ DISTRIBUTE *(BLOCK,BLOCK) :: S
```

- `*` here indicates a “trusted” argument

Descriptive Subroutine

- Subroutine **STONES**: dummy argument **S** and actual argument are required to have a **(BLOCK,BLOCK)** distribution

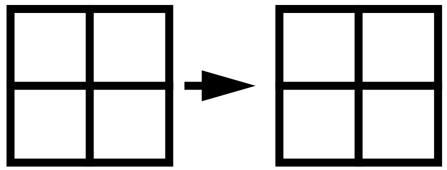
```
      SUBROUTINE STONES(S)
      REAL S(10,10)
!HPF$ DISTRIBUTE S *(BLOCK,BLOCK)
      ...
      END
```



Descriptive: Example Program 1

- Program **`EMERALD`**: match between the distribution of the actual and dummy arguments
- No check made on mapping of actual and dummy arguments
- Nothing need be done in passing array to **`STONES`**

```
      PROGRAM EMERALD
      REAL GREEN(10,10)
!HPF$ DISTRIBUTE GREEN(BLOCK,BLOCK)
      ...
      CALL STONES(GREEN)
      ...
      END
```

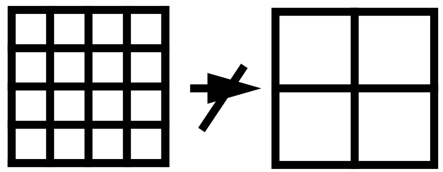


No remapping required

Descriptive: Example Program 2

- Program **`RUBY`**: array **`RED`** distributed in a **`(CYCLIC,CYCLIC)`** form.
- This *does not match* the distribution specified in the callee subroutine for the mapping of dummy argument **`D`**
- Call is HPF *nonconforming* (program behaviour is undefined in HPF)

```
      PROGRAM RUBY
      REAL RED(10,10)
!HPF$ DISTRIBUTE RED(CYCLIC,CYCLIC)
      ...
      CALL STONES(RED)
      ...
      END
```

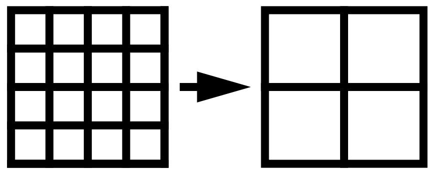


Nonconforming call

Descriptive: Example Program 3

- Program **AMBER**: array **YELLOW** must be remapped
- HPF conforming because of use of **INTERFACE** block
- A descriptive mapping contained in **INTERFACE** block treated as prescriptive, (remapping done before entry)

```
       PROGRAM AMBER
       REAL YELLOW(10,10)
!HPF$ DISTRIBUTE YELLOW(CYCLIC,CYCLIC)
       INTERFACE
         SUBROUTINE STONES(S)
           REAL S(10,10)
!HPF$      DISTRIBUTE S *(BLOCK,BLOCK)
         END SUBROUTINE
       END INTERFACE
       ...
       CALL STONES(YELLOW)
       ...
       END
```



Implicit Remapping
(Known at compile time)

Transcriptive Argument Mapping

- Avoid remapping of dummy arguments; accept data mapped in any distribution (more general)
- Inherit the distribution of the actual argument

```
!HPF$ INHERIT D
```

- Implies a **`DISTRIBUTE`** directive of the form, inherit both data distribution and processor arrangement

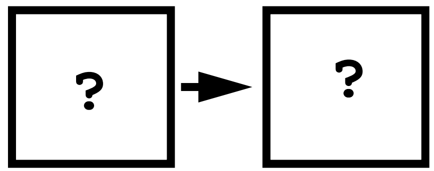
```
!HPF$ DISTRIBUTE * ONTO *
```

- *NOT IN SUBSET HPF !!*

Transcriptive Subroutine

- Subroutine **DOBEEDO**; any distribution of the dummy argument can be accepted without remapping

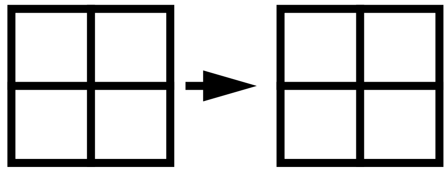
```
      SUBROUTINE DOBEEDO(D)
      REAL D(10,10)
!HPF$ INHERIT D
!HPF$ DISTRIBUTE D * ONTO *
      ...
      END
```



Transcriptive:Example Program1

- Program **SHAGGY:** passes array **B** to **DOBEEDO** with distribution **(BLOCK,BLOCK)** and no remapping, but ...
- **DOBEEDO** must be prepared to accept dummy argument in **(BLOCK,BLOCK)** distribution

```
       PROGRAM SHAGGY
       REAL B(10,10)
 !HPF$ DISTRIBUTE B(BLOCK,BLOCK)
       ...
       CALL DOBEEDO(B)
       ...
       END
```

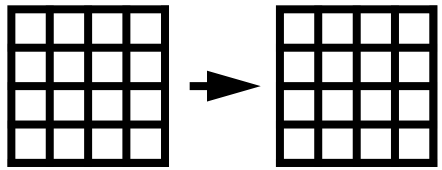


No remapping occurs

Transcriptive: Example Program2

- Program **SCOOBY**: no remapping is required, but ...
- **DOBEEDO** must be prepared to accept dummy argument with distribution **(CYCLIC,CYCLIC)**

```
      PROGRAM SCOOBY
      REAL A(10,10)
!HPF$ DISTRIBUTE A(CYCLIC,CYCLIC)
      ...
      CALL DOBEEDO(A)
      ...
      END
```



No remapping occurs

Problems with Transcriptive Mapping

- Remapping not required; saving in communication cost
- More flexible, but could generate inefficient code:
  - subroutine must be able to handle all possible distributions
  - distribution inherited may not be suitable for algorithm
- May be more efficient to remap on entry and exit to sub-routine to reduce the communications overhead within the subroutine itself
- *NOT IN SUBSET HPF !!*

A Possible Use of Transcriptive Mapping

- Write a general subroutine for any data distribution, but use particular algorithms depending on the distribution, determined by *user defined* functions **CYCLIC** and **BLOCK** e.g.

```
SUBROUTINE GAUSSIAN(A)
   REAL, DIMENSION(:,:) :: A
   INHERIT A
   IF (CYCLIC(A)) CALL CYCLIC_GAUSSIAN(A)
   IF (BLOCK(A)) CALL BLOCK_GAUSSIAN(A)
END SUBROUTINE
```

- where **CYCLIC_GAUSSIAN** and **BLOCK_GAUSSIAN** can include *descriptive* mapping directives

Distribution and Processor Arrangement

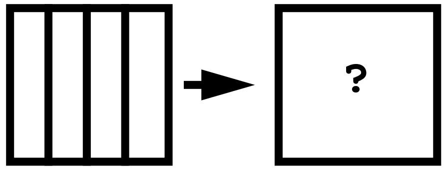
- Prescriptive, descriptive and transcriptive mappings also exist for the processor arrangement (after the **`ONTO`** clause)
- *Prescriptive*: require that processor arrangement is **`SQUARE`** with remapping if necessary

  ```
  !HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO SQUARE
  ```
- *Descriptive*: guarantee that processor arrangement is **`SQUARE`**

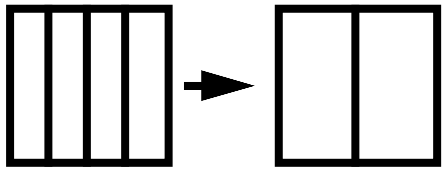
  ```
  !HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO *SQUARE
  ```
- *Transcriptive*: using the **`INHERIT`** directive: *NOT IN SUBSET HPF!!*

Processor Arrangement Example

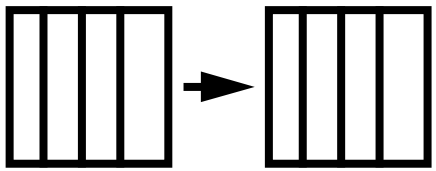
```
      PROGRAM
!HPF$ PROCESSORS four(4)
!HPF$ DISTRIBUTE A(*,CYCLIC) ONTO four
      ...
```




```
      SUBROUTINE HALF(A)
! Prescriptive processor arrangement
!HPF$ PROCESSORS two(2)
!HPF$ DISTRIBUTE * ONTO two
      ...
```



```
      SUBROUTINE QUARTER(A)
! Descriptive processor arrangement
!HPF$ PROCESSORS four(4)
!HPF$ DISTRIBUTE * ONTO *four
      ...
```



Alignment

- **ALIGN** directives also have prescriptive ...

```
      SUBROUTINE SIZE(LONG,SHORT)
      REAL, DIMENSION(1000) :: LONG,SHORT
!HPF$ DISTRIBUTE (BLOCK) :: SHORT
!HPF$ ALIGN WITH SHORT :: LONG
```

- and descriptive forms

```
      SUBROUTINE WEIGHT(HEAVY,LIGHT)
      REAL, DIMENSION(1000) :: HEAVY, LIGHT
!HPF$ DISTRIBUTE (BLOCK) :: HEAVY
!HPF$ ALIGN WITH *HEAVY :: LIGHT
```

- Transcriptive alignment using the **INHERIT** directive:
  NOT IN SUBSET HPF!!

Passing Array Sections

- What happens when passing sections of distributed arrays to subroutines? e.g.

```
CALL ODD(A(1:50:2))
```

- To explicitly specify the distribution of the dummy argument, need to determine the template of the dummy argument
- Template of a dummy argument determined by several rules; see Notes/Handbook for details
- *PASS WHOLE ARRAYS!!*

Summary

- Data mapping of procedure arguments
- Three forms of mappings
  - Prescriptive: requires data with a certain distribution and will remap

    `DISTRIBUTE (BLOCK,BLOCK) :: A`

  - Descriptive: asserts incoming data has a particular distribution

    `DISTRIBUTE *(BLOCK,BLOCK) :: A`

  - Transcriptive: accepts data with any distribution

    `DISTRIBUTE * :: A`

- Use **`INTERFACE`** blocks
- Pass whole arrays

Exercise 6: Life in a Subroutine

Data Mapping of Dummy Arguments

- Return to the distributed Game of Life from Exercise 3
- Rewrite the program with the iterations of the update of the system done in a subroutine
- Include the data mapping directives for the dummy arguments
  - Start with a descriptive mapping (distribute the actual and dummy arguments in the same way)
  - Generalise to a prescriptive mapping (distribute the actual arguments with a different mapping than the dummy arguments)

Intrinsic Functions and HPF Library

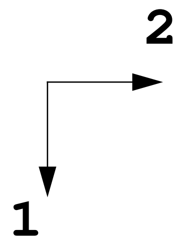
HPF Additions to F90

- HPF adds to the extensive set of intrinsics in F90:
  - Extensions to **`MAXLOC`**, **`MINLOC`**
  - Two system enquiry intrinsic functions
  - New **`ILEN`** intrinsic
  - The HPF Library Module

Extensions to MAXLOC, MINLOC

- **`MAXLOC`** and **`MINLOC`** are extended by the addition of an optional **`DIM`** argument

`MINLOC(ARRAY[,DIM][,MASK])`



A(3,4)

1	5	0	9
3	1	7	2
8	9	4	1

P

1	3

Q

3	2	4

```
P = minloc(A)
Q = minloc(A,dim=2)
```

New Intrinsics

System enquiry Functions

- `NUMBER_OF_PROCESSORS([DIM])` Returns either the total number of physical processors available or the number along a particular dimension of the processor array
- `PROCESSORS_SHAPE()` Returns the shape of the physical processor array

`ILEN(I)`

- Returns number of bits required to store the integer `I`
  - *Example:* `ILEN(4)=3`.

The HPF Library

- Mapping enquiry functions
- New array reduction functions
- Bit manipulation functions
- Array combining scatter functions
- Array sorting functions
- Prefix and suffix functions (unsegmented and segmented)

To make these routines available, a "`USE HPF_library`" statement is required at start of relevant program unit

Why More Library Routines?

- Particular routines are needed to support the data parallel style
- Routines needed to query alignment and distribution at run-time
- Library routines tend to be much more efficient than hand-coded equivalents

Mapping Enquiry Functions

- The actual mapping of an array can be determined at run time
  - **`HPF_ALIGNMENT`** returns information on array alignment
  - **`HPF_TEMPLATE`** returns information on the ultimate alignee (array or template)
  - **`HPF_DISTRIBUTION`** returns information on the distribution of the ultimate alignee (array or template)
- Information is returned in optional **`INTENT(OUT)`** arguments

HPF_ALIGNMENT

- **`HPF_ALIGNMENT(ALIGNEE[,...])`** returns information on array alignment with the align-target:
  - Alignments of elements within axes of **`ALIGNEE`** to corresponding align-target axes
  - Stride of alignment (0 for collapsed)
  - Axis correspondence
  - **`DYNAMIC`** align-target?
  - Number of copies aligned (1 for non-replicated)

HPF_TEMPLATE

- **`HPF_TEMPLATE(ALIGNEE[,...]`**
- Returns information on ultimate align-target
- Information is similar to that from **`HPF_ALIGNMENT`** but is presented from the template's point of view

HPF_DISTRIBUTION

- **`HPF_DISTRIBUTION(DISTRIBUTEE[,...])`**
- Returns information on the distribution of the ultimate alignee
  - Type of each axis ('**BLOCK**','**CYCLIC**','**COLLAPSED**')
  - Block size for distribution of each axis
  - Rank and shape of processor arrangement onto which **DISTRIBUTEE** is mapped

Array Reduction Functions

Similar to F90 **SUM** and **ANY** reduction functions.

- **IALL** corresponds to IAND
  `IALL( (/ 1,3,5,7 /) ) = 1`
- **IANY** corresponds to IOR
  `IANY( (/ 1,2,4,8 /) ) = 15`
- **IPARITY** corresponds to IEOR
  `IPARITY( (/ 3, 6, 12, 24 /) ) = 17`
- **PARITY** corresponds to .NEQV.
  `PARITY( (/ f, t, t, t /) ) is true`

Optional **DIM** and **MASK** arguments are accepted

Bit Manipulation Functions

- **`LEADZ`**
  - Returns the number of leading zero bits
  - **`LEADZ(3)`** is 30 on machine with 32 bit default integers
- **`POPCNT`**
  - Returns the number of one bits. **`POPCNT(15)`** is 4
- **`POPPAR`**
  - Returns the parity of an integer. **`POPPAR(49)`** is 1

Array Combining Scatter Functions

- Generalises the assignment

```
Y(V) = X
```

to allow repeated values in the vector **V**

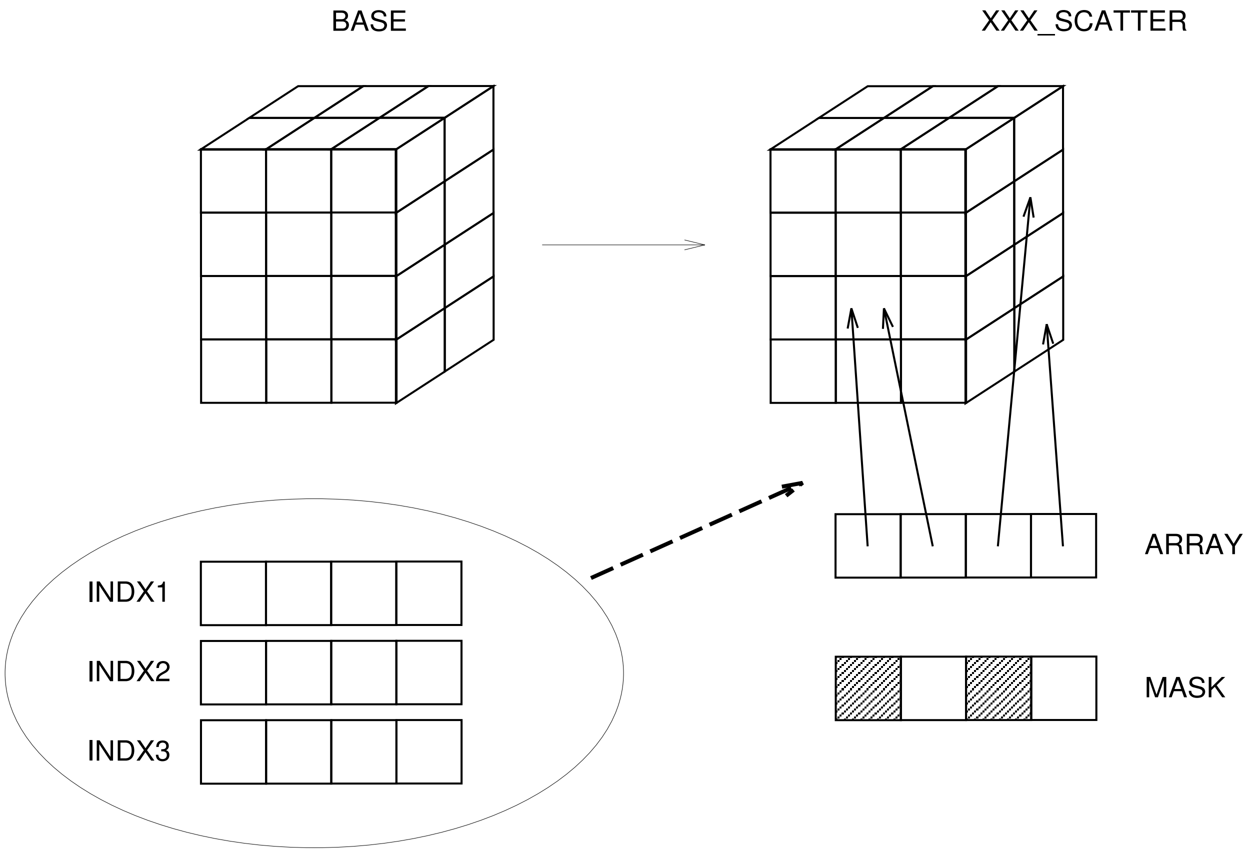
- Potential collisions are handled by a combiner **XXX**

XXX $\in$ **(ALL,ANY,COPY,COUNT,IALL,IANY,**
IPARITY,MAXVAL,MINVAL,PARITY,
PRODUCT,SUM)

- The functions have the form:

```
XXX_SCATTER(ARRAY,BASE,INDX1,...,INDXn[,MASK])
```

Array Combining Scatter Functions: Example



Array Sorting Functions

- Two functions are available to produce grading permutations of arrays

```
GRADE_DOWN(ARRAY[,DIM])
GRADE_UP(ARRAY[,DIM])
```

- These can operate on multidimensional arrays or along particular dimensions of arrays

Prefix and Suffix functions

- A prefix function — or scan — applies an operation along a vector such that any element is the result of operations on preceding elements
- For example:

```
SUM_PREFIX( (/1,2,0,1/) ) = (/1,3,3,4/)
```

- For the suffix operation:

```
SUM_SUFFIX( (/1,2,0,1/) ) = (/4,3,1,1/)
```

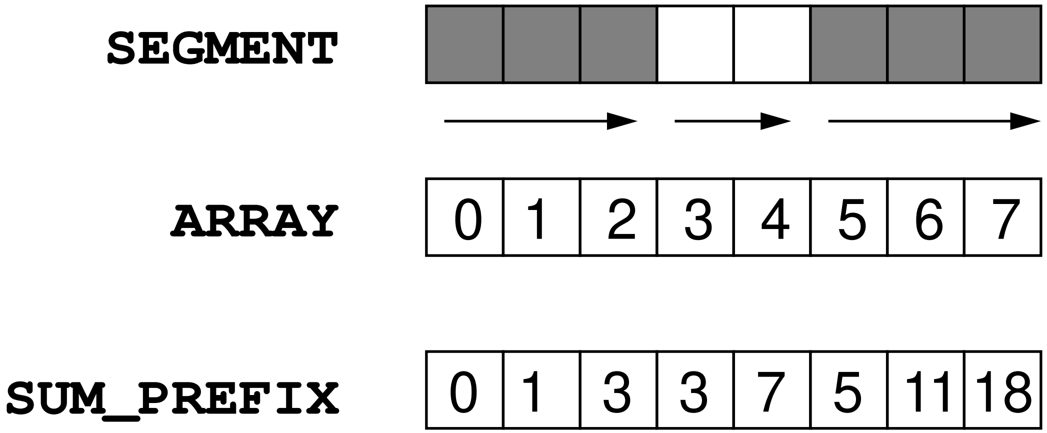
- Same combiners are available as for scatter functions:

```
XXX_PREFIX(ARRAY[,DIM,MASK,SEGMENT,EXCLUSIVE])
XXX_SUFFIX(ARRAY[,DIM,MASK,SEGMENT,EXCLUSIVE])
```

- Optional arguments modify behaviour

Segmentation

- The operation can be segmented with use of the **`SEGMENT`** argument

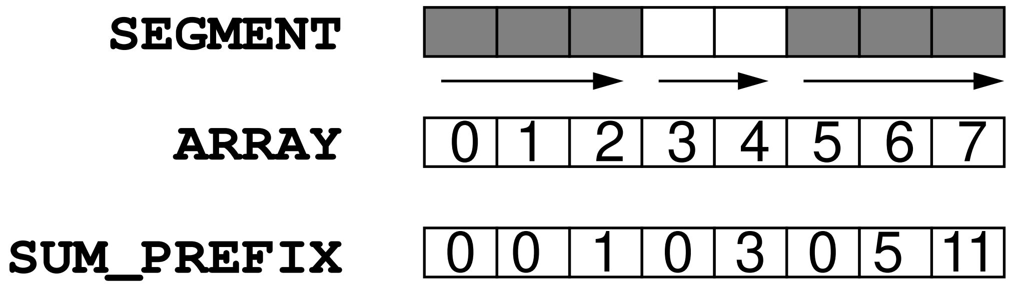


Inclusive and Exclusive Operations

- *Inclusive:* an element participates in the operation for its position
- *Exclusive:* an element does *not* participate in the operation for its position --- elements with no preceding elements receive the identity value
- Prefix and suffix operations have a logical scalar parameter (**`EXCLUSIVE`**) to control their behaviour
- Default is **`.FALSE.`**, *ie*, inclusive

EXCLUSIVE Examples

- An example with **`EXCLUSIVE`** set to **`.TRUE.`**

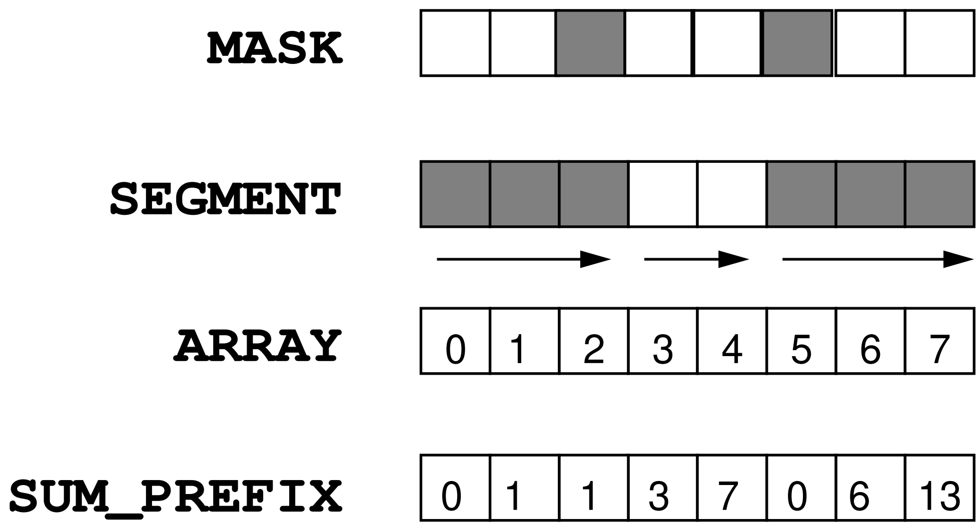


- Useful in “Birthdays” exercise:

```
  daysToMonth = SUM_PREFIX(daysInMonth,
&                                   EXCLUSIVE=.TRUE.)
```

Masking

- Operation can be masked with use of the **`MASK`** argument



- Only elements corresponding to true elements of the mask contribute

PREFIX and SUFFIX usage

- Typically an axis along which to do the scan is chosen with the `DIM` argument
  - If not, then elements are chosen in array-element order
- *Examples* of the use of prefix/suffix or scan routines are:
  - image expansion
  - percolation
  - discrete spatial population simulation

Exercise 7: Golf Scores

Analysing a golf score card using HPF instrinsic functions

- Answer the following questions about the score card in the notes:
  - Find the running total scores for the round
  - Find the running totals for the first and second nine holes
  - Find running totals for Par 3, 4 and 5 holes separately (use three separate arrays; only set the running totals at the holes concerned, set the others to zero)
  - Enumerate the holes on which the player scored a birdie (ie. one less than par)

Advanced Topics and Current Developments

Contents

- Sequence and storage association
- Extrinsic Procedures
- Current Developments in HPF: HPF II

Sequence Association

- Fortran developed on hardware with linear memory architecture
- E.g. elements of array **`A`** of size **`(2,3)`**: stored in memory as **`A(1,1)`**, **`A(2,1)`**, **`A(1,2)`**, **`A(2,2)`**, **`A(1,3)`** **`... A(2,3)`**
- Arrays mapped to linear order: first axis varies fastest
- Sequence association is correspondence between two arrays mapped to a linear order

The rank and shape of the actual argument need not agree with the rank and shape of the dummy argument

Fortran Standard (12.4.1.4)

Example: Sequence Association

```
      PARAMETER (n=100)
      INTEGER A(n,n)
         CALL zero(A,n*n) ! zero multi-dim array
      END

      SUBROUTINE zero(B,m)
      INTEGER m, B(m)     ! 1-D array declared
      DO 1, i = 1, m
1        B(i) = 0
      END
```

- Sequence association defines correspondence between 2-D actual argument and 1-D dummy argument

Storage Association

- Important in **`COMMON`** blocks and **`EQUIVALENCE`** statements
- Storage units reused for different sets of variables in different parts of the code

Storage association is the association of two or more data objects that occurs when two or more storage sequences share or are aligned with one or more storage units

Fortran Standard (14.6.3.1)

Example: Storage Association

```
INTEGER A,C
DOUBLE PRECISION B
COMMON /red/A(10,10),B(50),C(200)
  ...
INTEGER P,R
REAL Q
COMMON /red/P(5,20),Q(50),R(250)
```

- The **COMMON** blocks share storage units and are associated by storage association
  - **P** shares storage with **A**, **Q** shares storage with the first half of **B** and **R** shares storage with the second half of **B** and **C**.

Sequence and Storage Association in HPF

Making old codes work

- Sequence and storage association causes problems for parallel implementation
- Linear memory model no longer valid for distributed arrays
- HPF uses **`SEQUENCE`** directive
- Tells compiler explicitly that:
  - variable/**COMMON** block has sequential property
  - sequence/storage association should be supported

Sequence Association in HPF

```
      PARAMETER (n=100)
      INTEGER, DIMENSION(n,n) :: A, B, C
!HPF$ SEQUENCE A, B, C          ! need HPF$ SEQUENCE if
      CALL zero(A,n*n)          !  array passed by name
      CALL zero(B(1,5),n)       ! pass an array section
      CALL zero(C(1,10),1)      ! pass an array element
      END

      SUBROUTINE zero(D,m)
      INTEGER D(m),m! 1-D array declared
!HPF$ SEQUENCE D
      DO 1, i = 1, m
 1       B(i) = 0
      END
```

Storage Association in HPF

```
      SUBROUTINE ENO
!HPF$ SEQUENCE /MADNESS/
      COMMON /GRIDS/ X(100,100),Y(100,100)
      COMMON /MADNESS/MAD(50,50),NESS(10,10)
      END SUBROUTINE

      SUBROUTINE OWT
!HPF$ SEQUENCE /MADNESS/
      COMMON /GRIDS/ X(100,100),Y(100,100)
      COMMON /MADNESS/M(500),ADNE(2,2,2),SS(10,10)
      END SUBROUTINE
```

- It is safe to distribute the variables in **`GRIDS`**, as all variables have same size, type and shape
- **`!HPF$ SEQUENCE /MADNESS/`** guarantees explicit mapping is same in every subroutine

Extrinsic Procedures

Allow programming in other languages/styles

- EXTRINSIC defines the interface between HPF and a non-HPF environment
- Local code may be in another language
- Local code may be in another style (message-passing)
- Example: interface to a C routine

```
INTERFACE
   EXTRINSIC(C) FUNCTION SYSTEM(COMMAND)
      INTEGER :: SYSTEM
      CHARACTER *(*), INTENT(IN) :: COMMAND
   END FUNCTION SYSTEM
END INTERFACE
```

HPF_LOCAL Extrinsic Kind

- Only other extrinsic kind defined by HPF
- Copy of single HPF procedure executes on each node
- Each node has access to data only through procedure arguments; piece of distributed array
- Nodes synchronise after return from local routine
- `HPF_LOCAL_LIBRARY` module contains routines to support this style

HPFF II

- First meeting: January 1994, Texas
- Clarifications and interpretations of HPF 1.0
- Encourage “industrial strength” implementations
- Identify requirements for HPF 2.0
  - enhanced mapping features
  - parallel I/O
  - task parallelism
- Produce HPF 1.x document (November 1994 HPF v1.1)

Correction, Clarifications and Interpretations (CCI)

- HPF produced in a short time --> errors and ambiguities still remain
- Group is working to:
  - define corrections to HPF standard
  - clarify points not explicitly covered
  - provide explicit interpretations of standard
- CCI document available from HPFF Web pages

Motivating Applications

- Gather applications which are useful motivations for extensions to HPF 1.0
- Provide set of mini-applications for which an HPF implementation should produce optimised code
- Examples include:
  - N-Body simulation
  - Molecular Dynamics
  - 3-D Hydrodynamics
  - Fast Fourier Transform

Tasking Requirements Group

- HPF 1.0 focusses on parallelism at the level of data elements; no control over coarse-grained parallelism
- Tasking involves control over the mapping or scheduling of these fine grained units of computation
- Tasking directives include possibilities for:
  - control of distribution of work over processors (c.f. data)
  - allowing multiple HPF processes

Irregular Data Working Group

- Little support for irregular data distribution in HPF 1.0
- Define more general data distributions/alignments to deal with:
  - unbalanced loads
  - unstructured data access
  - dynamically built data
- Enhanced mapping features include:
  - arbitrary mapping of data to processors
  - mapping of linked data structures
  - mapping to processor subsets

Parallel I/O Working Group

- Parallel I/O would provide scalable access to data storage
- Significant work was done for HPF 1.0 but not included
- Operations of interest include:
  - Parallel access to files
  - check-pointing and restart facilities
  - ability to overlap computation with I/O
- Include recommendations made to X3J3

Summary

- Sequence and Storage Association: HPF **`SEQUENCE`** directive
- Extrinsic functions: a “back door” to other programming languages and styles
- HPFF II
- Current Developments in HPF 1.0

Further Information

- HPF Standard: available by ftp from **`titan.cs.rice.edu`** in directory /**`public/HPFF`**
- "*The High Performance Fortran Handbook*", C.H. Koebel et al., MIT Press, 1994.
- World Wide Web: URL

  `http://www.erc.msstate.edu/hpff/home.html`
- Mailing list: send message "**`add hpff`**" to

  `hpf-request@cs.rice.edu`
- EPCC-TEC WWW pages

  `http://www.epcc.ed.ac.uk/epcc-tec`