THE UNIVERSITY *of* LIVERPOOL

HPF Programming 5 Day Course Slides

—

Dr. A C Marshall (funded by JISC/NTI)

with acknowledgements to Steve Morgan, Dave Watson and Mike Delves.

Lecture 1:
Overview, Objects and Expressions

Course Philosophy

The course:

- assumes a familiarity with a high level language;
- stresses modern scientific programming syntax, for example, array language, modules, defined types, recursion and overloaded operators;
- gives many examples;

Fortran Evolution

History:

- FORmula TRANslation.
- first compiler: 1957.
- first official standard 1972: 'Fortran 66'.
- updated in 1980 to Fortran 77.
- updated further in 1991 to Fortran 90.
- next upgrade due in 1996 - remove obsolescent features, correct mistakes and add limited basket of new facilities such as `ELEMENTAL` and `PURE` user-defined procedures and the `FORALL` statement.
- Fortran is now an ISO/IEC and ANSI standard.

Drawbacks of Fortran 77

Fortran 77 was limited in the following areas,

1. awkward 'punched card' or 'fixed form' source format;

2. inability to represent intrinsically parallel operations;

3. lack of dynamic storage;

4. non-portability;

5. no user-defined data types;

6. lack of explicit recursion;

7. reliance on unsafe storage and sequence association features.

Fortran 90 New features

Fortran 90 supports,

1. free source form;
2. array syntax and many more (array) intrinsics;
3. dynamic storage and pointers;
4. portable data types (`KIND`s);
5. derived data types and operators;
6. recursion;
7. `MODULE`s

- procedure interfaces;
- enhanced control structures;
- user defined generic procedures;
- enhanced I/O.

Language Obsolescence

Fortran 90 has a number of features marked as obsolescent, this means,

- they are already redundant in Fortran 77;
- better methods of programming already existed in the Fortran 77 standard;
- programmers should stop using them;
- the standards committee's intention is that many of these features will be removed from the next revision of the language, Fortran 95;

Obsolescent Features

The following features are labelled as obsolescent and will be removed from the next revision of Fortran, Fortran 95,

- the arithmetic `IF` statement;
- `ASSIGN` statement;
- `ASSIGN`ed `GOTO` statements;
- `ASSIGN`ed `FORMAT` statements;
- Hollerith format strings;
- the `PAUSE` statement;
- `REAL` and `DOUBLE PRECISION` `DO`-loop control expressions and index variables;
- shared `DO`-loop termination;
- alternate `RETURN`;
- branching to an `ENDIF` from outside the `IF` block;

Undesirable Features

- fixed source form layout - use free form;
- implicit declaration of variables - use `IMPLICIT NONE`;
- `COMMON` blocks - use `MODULE`;
- assumed size arrays - use assumed shape;
- `EQUIVALENCE` statements;
- `ENTRY` statements;
- the computed `GOTO` statement - use `IF` statement;

Object Oriented Facilities

Fortran 90 has some Object Oriented facilities such as:

- *data abstraction* — user-defined types;
- *data hiding* — `PRIVATE` and `PUBLIC` attributes;
- *encapsulation* — Modules and data hiding facilities;
- *inheritance* and *extensibility* — super-types, operator overloading and generic procedures;
- *polymorphism* — user can program his / her own polymorphism by generic overloading;
- *reusability* — Modules;

Example

Example Fortran 90 program:

```
MODULE Triangle_Operations
 IMPLICIT NONE
CONTAINS
 FUNCTION Area(x,y,z)
  REAL :: Area       ! function type
  REAL, INTENT( IN ) :: x, y, z
  REAL :: theta, height
  theta=ACOS((x**2+y**2-z**2)/(2.0*x*y))
  height=x*SIN(theta); Area=0.5*y*height
 END FUNCTION Area
END MODULE Triangle_Operations

PROGRAM Triangle
 USE Triangle_Operations
 IMPLICIT NONE
  REAL :: a, b, c, Area
  PRINT*, 'Welcome, please enter the&
         &lengths of the 3 sides.'
  READ*, a, b, c
  PRINT*,'Triangle''s area: ',Area(a,b,c)
END PROGRAM Triangle
```

Source Form

Free source form:

- 132 characters per line;
- '`!`' comment initiator;
- '`&`' line continuation character;
- '`;`' statement separator;
- significant blanks.

Example,

```
PRINT*, "This line is continued &
        &On the next line"; END ! of program
```

Character Set

The following are valid in a Fortran 90 program:

- alphanumeric:

  `a-z`, `A-Z`, `0-9`, and `_` (the underscore)

- symbolic:

Symbol	Description	Symbol	Description
	space	=	equal
+	plus	-	minus
*	asterisk	/	slash
(	left paren	)	right paren
,	comma	.	period
'	single quote	"	double quote
:	colon	;	semicolon
!	shriek	&	ampersand
%	percent	<	less than
>	greater than	$	dollar
?	question mark		

Significance of Blanks

In *free form* source code blanks must not appear:

- within keywords

```
INTEGER :: wizzy        ! is a valid keyword
INT EGER :: wizzy       ! is not
```

- within names

```
REAL :: running_total   ! is a valid name
REAL :: running total   ! is not
```

Blanks must appear:

- between two separate keywords
- between keywords and names not otherwise separated by punctuation or other special characters.

```
INTEGER FUNCTION fit(i) ! is valid
INTEGERFUNCTION fit(i)  ! is not
INTEGER FUNCTIONfit(i)  ! is not
```

Blanks are optional between some keywords mainly '`END` < *construct* >' and a few others; *if in doubt add a blank* (it looks better too).

Names

In Fortran 90 variable names (and procedure names etc.)

- must start with a letter

```
REAL :: a1 ! valid name
REAL :: 1a ! not valid name
```

- may use only letters, digits and the underscore

```
CHARACTER :: atoz ! valid name
CHARACTER :: a-z  ! not valid name
CHARACTER :: a_z  ! OK
```

- underscore should be used to separate words in long names

```
CHARACTER(LEN=8) :: user_name ! valid name
CHARACTER(LEN=8) :: username  ! different name
```

- may not be longer than 31 characters

Comments

It is good practise to use lots of comments, for example,

```
    PROGRAM Saddo
!
! Program to evaluate marriage potential
!
    LOGICAL :: TrainSpotter ! Do we spot trains?
    LOGICAL :: SmellySocks  ! Have we smelly socks?
    INTEGER :: i, j         ! Loop variables
```

- everything after the `!` is a comment;
- the `!` in a character context does **not** begin a comment, for example,

```
    PRINT*, "No chance of ever marrying!!!"
```

Statement Ordering

The following table details the prescribed ordering:

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, MODULE or BLOCK DATA statement</td></tr>
<tr><td colspan="3">USE statement</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY statements</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER statement</td><td>IMPLICIT statements</td></tr>
<tr><td>PARAMETER and DATA statements</td><td>Derived-Type Definition, Interface blocks, Type declaration statements, Statement function statements and specification statements</td></tr>
<tr><td>DATA statements</td><td>Executable constructs</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
<tr><td colspan="3">Internal or module procedures</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

Intrinsic Types

Fortran 90 has three broad classes of object type,

- character;
- boolean;
- numeric.

these give rise to six simple intrinsic types, known as default types,

```
CHARACTER          :: sex   ! letter
CHARACTER(LEN=12)  :: name  ! string
LOGICAL            :: wed   ! married?
REAL               :: height
DOUBLE PRECISION   :: pi    ! 3.14...
INTEGER            :: age   ! whole No.
COMPLEX            :: val   ! x + iy
```

Literal Constants

A literal constant is an entity with a fixed value:

```
12345       ! INTEGER
1.0         ! REAL
-6.6E-06    ! REAL: -6.6*10**(-6)
.FALSE.     ! LOGICAL
.TRUE.      ! LOGICAL
"Mau'dib"   ! CHARACTER
'Mau''dib'  ! CHARACTER
```

Note,

- there are only two `LOGICAL` values;
- `REAL`s contain a decimal point, `INTEGER`s do not,
- `REAL`s have an exponential form
- character literals delimited by " and ';
- two occurrences of the delimiter inside a string produce one occurrence on output;
- there is only a finite range of values that numeric literals can take.

Implicit Typing

Undeclared variables have an implicit type,

- if first letter is `I, J, K, L, M` or `N` then type is `INTEGER`;
- any other letter then type is `REAL`s.

Implicit typing is potentially very dangerous and should **always** be turned off by adding:

```
IMPLICIT NONE
```

as the first line after any `USE` statements.

Consider,

```
    DO 30 I = 1.1000
     ...
30 CONTINUE
```

in fixed format with implicit typing this declares a `REAL` variable `DO30I` and sets it to `1.1000` instead of performing a loop 1000 times!

Numeric and Logical Declarations

With `IMPLICIT NONE` variables must be declared. A simplified syntax follows,

< *type* > [,< *attribute*-list >] ::< *variable*-list >&
[=< *value* >]

The following are all valid declarations,

```
REAL                    :: x
INTEGER                 :: i, j
LOGICAL, POINTER        :: ptr
REAL, DIMENSION(10,10)  :: y, z
INTEGER                 :: k = 4
```

The `DIMENSION` attribute declares an array (10×10).

Character Declarations

Character variables are declared in a similar way to numeric types. `CHARACTER` variables can

- refer to one character;
- refer to a string of characters which is achieved by adding a length specifier to the object declaration.

The following are all valid declarations,

```
CHARACTER(LEN=10)  :: name
CHARACTER          :: sex
CHARACTER(LEN=32)  :: str
CHARACTER(LEN=10), DIMENSION(10,10) :: Harray
CHARACTER(LEN=32), POINTER :: Pstr
```

Constants (Parameters)

Symbolic constants, oddly known as *parameters* in Fortran, can easily be set up either in an attributed declaration or parameter statement,

```
REAL, PARAMETER :: pi = 3.14159
CHARACTER(LEN=*), PARAMETER :: &
                  son = 'bart', dad = "Homer"
```

`CHARACTER` constants can assume their length from the associated literal (`LEN=*`).

Parameters should be used:

- if it is known that a variable will only take one value;
- for legibility where a 'magic value' occurs in a program such as π;
- for maintainability when a 'constant' value could feasibly be changed in the future.

Initialisation

Variables can be given initial values:

- can use *initialisation expressions*,
- may only contain PARAMETERs or literals.

```
REAL              :: x, y =1.0D5
INTEGER           :: i = 5, j = 100
CHARACTER(LEN=5)  :: light = 'Amber'
CHARACTER(LEN=9)  :: gumboot = 'Wellie'
LOGICAL  :: on = .TRUE., off = .FALSE.
REAL, PARAMETER :: pi = 3.141592
REAL, PARAMETER :: radius = 3.5
REAL :: circum = 2 * pi * radius
```

gumboot will be padded, to the right, with blanks.

In general, intrinsic functions *cannot* be used in initialisation expressions, the following can be: REPEAT, RESHAPE, SELECTED_INT_KIND, SELECTED_REAL_KIND, TRANSFER, TRIM, LBOUND, UBOUND, SHAPE, SIZE, KIND, LEN, BIT_SIZE and numeric inquiry intrinsics, for, example, HUGE, TINY, EPSILON.

Expressions

Each of the three broad type classes has its own set of intrinsic (in-built) operators, for example, `+`, `//` and `.AND.`,

The following are valid expressions,

- `NumBabiesBorn+1` — numeric valued
- `"Ward "//Ward` — character valued
- `TimeSinceLastBirth .GT. MaxTimeTwixtBirths` — logical valued

Expressions can be used in many contexts and can be of any intrinsic type.

Assignment

Assignment is defined between all expressions of the same type:

Examples,

```
a = b
c = SIN(.7)*12.7  ! SIN in radians
name = initials//surname
bool = (a.EQ.b.OR.c.NE.d)
```

The LHS is an object and the RHS is an expression.

Intrinsic Numeric Operations

The following operators are valid for numeric expressions,

- `**` exponentiation, dyadic operator, for example, `10**2`, (evaluated right to left);
- `*` and `/` multiply and divide, dyadic operators, for example, `10*7/4`;
- `+` and `–` plus and minus or add and subtract, monadic and dyadic operators, for example, `10+7–4` and `–3`;

Can be applied to literals, constants, scalar and array objects. The only restriction is that the RHS of `**` must be scalar.

Example,

```
a = b - c
f = -3*6/5
```

Intrinsic Character Operations

Consider,

```
CHARACTER(LEN=*), PARAMETER :: str1 = "abcdef"
CHARACTER(LEN=*), PARAMETER :: str2 = "xyz"
```

substrings can be taken,

- `str1` is 'abcdef'
- `str1(1:1)` is 'a' (**not** `str1(1)` — illegal)
- `str1(2:4)` is 'bcd'

The concatenation operator, //, is used to join two strings.

```
PRINT*, str1//str2
PRINT*, str1(4:5)//str2(1:2)
```

would produce

```
abcdefxyz
dexy
```

Relational Operators

The following *relational operators* deliver a `LOGICAL` result when combined with numeric operands,

`.GT.`	`>`	greater than
`.GE.`	`>=`	greater than or equal to
`.LE.`	`<=`	less than or equal to
`.LT.`	`<`	less than
`.NE.`	`/=`	not equal to
`.EQ.`	`==`	equal to

For example,

```
bool = i .GT. j
boule = i > j
IF (i .EQ. j) c = D
IF (i == j)   c = D
```

When using real-valued expressions (which are approximate) `.EQ.` and `.NE.` have no real meaning.

```
REAL :: Tol = 0.0001
IF (ABS(a-b) .LT. Tol) same = .TRUE.
```

Intrinsic Logical Operations

A `LOGICAL` or boolean expression returns a `.TRUE.` / `.FALSE.` result. The following are valid with `LOGICAL` operands,

- `.NOT.` — `.TRUE.` if operand is `.FALSE.`.
- `.AND.` — `.TRUE.` if both operands are `.TRUE.`;
- `.OR.` — `.TRUE.` if at least one operand is `.TRUE.`;
- `.EQV.` — `.TRUE.` if both operands are the same;
- `.NEQV.` — `.TRUE.` if both operands are different.

For example, if `T` is `.TRUE.` and `F` is `.FALSE.`

- `.NOT. T` is `.FALSE.`, `.NOT. F` is `.TRUE.`.
- `T .AND. F` is `.FALSE.`, `T .AND. T` is `.TRUE.`.
- `T .OR. F` is `.TRUE.`, `F .OR. F` is `.FALSE.`.
- `T .EQV. F` is `.FALSE.`, `F .EQV. F` is `.TRUE.`.
- `T .NEQV. F` is `.TRUE.`, `F .NEQV. F` is `.FALSE.`.

Operator Precedence

Operator	Precedence	Example
user-defined monadic	Highest	`.INVERSE.A`
`**`	·	`10**4`
`*` or `/`	·	`89*55`
monadic `+` or `-`	·	`-4`
dyadic `+` or `-`	·	`5+4`
`//`	·	`str1//str2`
`.GT.`, `>`, `.LE.`, `<=`, etc	·	`A > B`
`.NOT.`	·	`.NOT.Bool`
`.AND.`	·	`A.AND.B`
`.OR.`	·	`A.OR.B`
`.EQV.` or `.NEQV.`	·	`A.EQV.B`
user-defined dyadic	Lowest	`X.DOT.Y`

Note:

- in an expression with no parentheses, the highest precedence operator is combined with its operands first;
- in contexts of equal precedence left to right evaluation is performed except for `**`.

Precedence Example

The following expression,

```
x = a+b/5.0-c**d+1*e
```

is equivalent to

```
x = a+b/5.0-(c**d)+1*e
```

as ** is highest precedence. This is equivalent to

```
x = a+(b/5.0)-(c**d)+(1*e)
```

as / and * are next highest. The remaining operators' precedences are equal, so we evaluate from left to right.

Lecture 2:

Control Constructs

Control Flow

Control constructs allow the normal sequential order of execution to be changed.

Fortran 90 supports:

- conditional execution statements and constructs, (`IF` ... and `IF` ... `THEN` ... `ELSE` ... `END IF`);
- loops, (`DO` ... `END DO`);
- multi-way choice construct, (`SELECT CASE`);

`IF` Statement

Example,

```
IF (bool_val) A = 3
```

The basic syntax is,

`IF`(< *logical-expression* >)< *exec-stmt* >

If < *logical-expression* > evaluates to `.TRUE.` then execute < *exec-stmt* > otherwise do not.

For example,

```
IF (x .GT. y) Maxi = x
```

means 'if `x` is greater than `y` then set `Maxi` to be equal to the value of `x`'.

More examples,

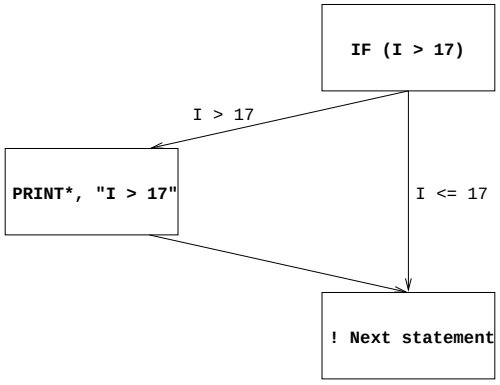
```
IF (a*b+c <= 47) Boolie = .TRUE.
IF (i .NE. 0 .AND. j .NE. 0) k = 1/(i*j)
IF (i /= 0 .AND. j /= 0) k = 1/(i*j) ! same
```

Visualisation of the IF Statement

Consider the IF statement

```
IF (I > 17) Print*, "I > 17"
```

this maps onto the following control flow structure,



IF ... THEN ... ELSE Construct

The block-IF is a more flexible version of the single line IF. A simple example,

```
IF (i .EQ. 0) THEN
 PRINT*, "I is Zero"
ELSE
 PRINT*, "I is NOT Zero"
ENDIF
```

note the how indentation helps.

Can also have one or more ELSEIF branches:

```
IF (i .EQ. 0) THEN
 PRINT*, "I is Zero"
ELSE IF (i .GT. 0) THEN
 PRINT*, "I is greater than Zero"
ELSE
 PRINT*, "I must be less than Zero"
ENDIF
```

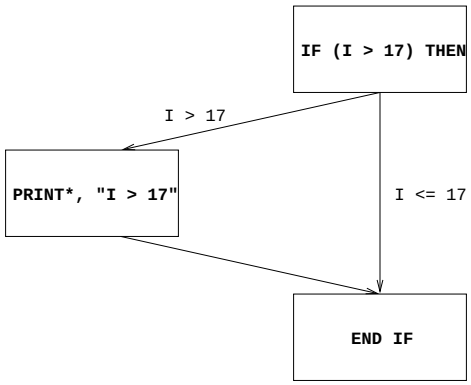
Both ELSE and ELSEIF are optional.

Visualisation of the IF ... THEN Construct

Consider the IF ... THEN construct

```
IF (I > 17) THEN
  Print*, "I > 17"
END IF
```

this maps onto the following control flow structure,

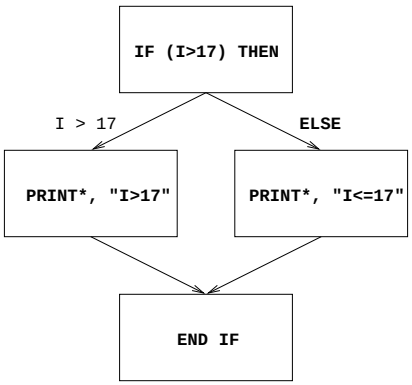


Visualisation of the IF ... THEN ... ELSE Construct

Consider the IF ... THEN ... ELSE construct

```
IF (I > 17) THEN
  Print*, "I > 17"
ELSE
  Print*, "I <= 17"
END IF
```

this maps onto the following control flow structure,



IF ... THEN .... ELSEIF Construct

The IF construct has the following syntax,

IF(< *logical-expression* >)THEN
        < *then-block* >
[ELSEIF(< *logical-expression* >)THEN
        < *elseif-block* >
...]
[ELSE
        < *else-block* >]
END IF

The first branch to have a true < *logical-expression* > is the one that is executed. If none are found then the < *else-block* >, if present, is executed.

For example,

```
IF (x .GT. 3) THEN
 CALL SUB1
ELSEIF (x .EQ. 3) THEN
 A = B*C-D
ELSEIF (x .EQ. 2) THEN
 A = B*B
ELSE
 IF (y .NE. 0) A=B
ENDIF
```

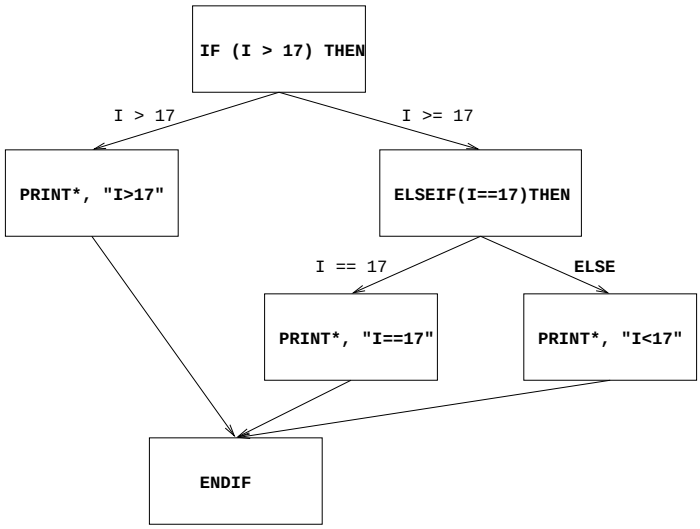
IF blocks may also be nested.

Visualisation of IF ... THEN .. ELSEIF Construct

Consider the IF ... THEN ... ELSEIF construct

```
IF (I > 17) THEN
  Print*, "I > 17"
ELSEIF (I == 17)
  Print*, "I == 17"
ELSE
  Print*, "I < 17"
END IF
```

this maps onto the following control flow structure,



Nested and Named IF Constructs

All control constructs can be both named and nested.

```
outa: IF (a .NE. 0) THEN
        PRINT*, "a /= 0"
        IF (c .NE. 0) THEN
          PRINT*, "a /= 0 AND c /= 0"
        ELSE
          PRINT*, "a /= 0 BUT c == 0"
        ENDIF
      ELSEIF (a .GT. 0) THEN outa
        PRINT*, "a > 0"
      ELSE outa
        PRINT*, "a must be < 0"
      ENDIF outa
```

The names may only be used *once* per program unit and are only intended to make the code clearer.

Conditional Exit Loops

Can set up a DO loop which is terminated by simply jumping out of it. Consider,

```
i = 0
DO
  i = i + 1
  IF (i .GT. 100) EXIT
  PRINT*, "I is", i
END DO
! if i>100 control jumps here
PRINT*, "Loop finished. I now equals", i
```

this will generate

```
I is   1
I is   2
I is   3
  ....
I is   100
Loop finished. I now equals   101
```

The EXIT statement tells control to jump out of the current DO loop.

Conditional Cycle Loops

Can set up a `DO` loop which, on some iterations, only executes a subset of its statements. Consider,

```
i = 0
DO
  i = i + 1
  IF (i >= 50 .AND. i <= 59) CYCLE
  IF (i > 100) EXIT
  PRINT*, "I is", i
END DO
PRINT*, "Loop finished. I now equals", i
```

this will generate

```
I is   1
I is   2
  ....
I is   49
I is   60
  ....
I is   100
Loop finished. I now equals   101
```

`CYCLE` forces control to the **innermost** active `DO` statement and the loop begins a new iteration.

Named and Nested Loops

Loops can be given names and an `EXIT` or `CYCLE` statement can be made to refer to a particular loop.

```
0|      outa: DO
1|       inna: DO
2|         ...
3|         IF (a.GT.b) EXIT outa  ! jump to line 9
4|         IF (a.EQ.b) CYCLE outa ! jump to line 0
5|         IF (c.GT.d) EXIT inna  ! jump to line 8
6|         IF (c.EQ.a) CYCLE      ! jump to line 1
7|       END DO inna
8|      END DO outa
9|      ...
```

The (optional) name following the `EXIT` or `CYCLE` highlights which loop the statement refers to.

Loop names can only be used once per program unit.

Indexed DO Loops

Loops can be written which cycle a fixed number of times. For example,

```
DO i1 = 1, 100, 1
  ... ! i is 1,2,3,...,100
  ... ! 100 iterations
END DO
```

The formal syntax is as follows,

DO < DO-*var* >=< *expr1* >,< *expr2* > [,< *expr3* >]
< *exec-stmts* >
END DO

The number of iterations, which is evaluated **before** execution of the loop begins, is calculated as

MAX(INT((< *expr2* >-< *expr1* >+< *expr3* >)/< *expr3* >),0)

If this is zero or negative then the loop is not executed.

If < *expr3* > is absent it is assumed to be equal to 1.

Examples of Loop Counts

A few examples of different loops,

1. upper bound not exact,

```
loopy: DO i = 1, 30, 2
 ... ! i is 1,3,5,7,...,29
 ... ! 15 iterations
END DO loopy
```

2. negative stride,

```
DO j = 30, 1, -2
 ... ! j is 30,28,26,...,2
 ... ! 15 iterations
END DO
```

3. a zero-trip loop,

```
DO k = 30, 1, 2
 ... ! 0 iterations
 ... ! loop skipped
END DO
```

4. missing stride — assume it is 1,

```
DO l = 1,30
 ... ! i = 1,2,3,...,30
 ... ! 30 iterations
END DO
```

Scope of DO Variables

1. I is recalculated at the top of the loop and then compared with < *expr2* >,

2. if the loop has finished, execution jumps to the statement after the corresponding END DO,

3. I retains the value that it had just been assigned.

For example,

```
DO i = 4, 45, 17
 PRINT*, "I in loop = ",i
END DO
PRINT*, "I after loop = ",i
```

will produce

```
I in loop =  4
I in loop =  21
I in loop =  38
I after loop =  55
```

An index variable may not have its value changed in a loop.

SELECT CASE Construct I

Simple example

```
    SELECT CASE (i)
      CASE (3,5,7)
        PRINT*,"i is prime"
      CASE (10:)
        PRINT*,"i is > 10"
      CASE DEFAULT
        PRINT*, "i is not prime and is < 10"
    END SELECT
```

An IF .. ENDIF construct could have been used but a SELECT CASE is neater and more efficient. Another example,

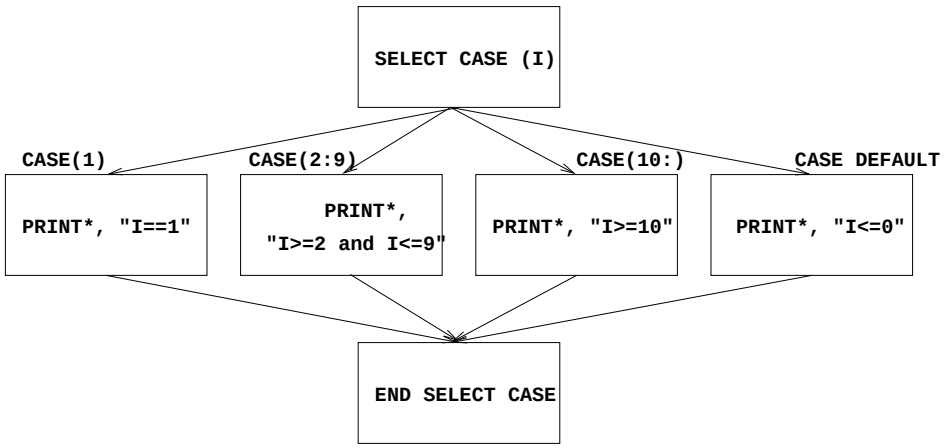
```
    SELECT CASE (num)
      CASE (6,9,99,66)
!     IF(num==6.OR. .. .OR.num==66) THEN
        PRINT*, "Woof woof"
      CASE (10:65,67:98)
!     ELSEIF((num >= 10 .AND. num <= 65) .OR. ...
        PRINT*, "Bow wow"
      CASE DEFAULT
!     ELSE
        PRINT*, "Meeeoow"
     END SELECT
!    ENDIF
```

Visualisation of SELECT CASE

Consider the SELECT CASE construct

```
SELECT CASE (I)
 CASE(1);   Print*, "I==1"
 CASE(2:9); Print*, "I>=2 and I<=9"
 CASE(10);  Print*, "I>=10"
 CASE DEFAULT; Print*, "I<=0"
END SELECT CASE
```

this maps onto the following control flow structure,



SELECT CASE Construct II

This is useful if one of several paths must be chosen based on the value of a single expression.

The syntax is as follows,

```
[ <name>:] SELECT CASE (<case-expr>)
       [ CASE (<case-selector>)[ <name> ]
              <exec-stmts> ] ...
       [ CASE DEFAULT [ <name> ]
              <exec-stmts> ]
END SELECT [ <name> ]
```

Note,

- the <*case-expr*> must be scalar and `INTEGER`, `LOGICAL` or `CHARACTER` valued;
- the <*case-selector*> is a parenthesised single value or range, for example, `(.TRUE.)`, `(1)` or `(99:101)`;
- there can only be one `CASE DEFAULT` branch;
- control cannot jump into a `CASE` construct.

PRINT **Statement**

This is the simplest form of directing unformatted data to the standard output channel, for example,

```
PROGRAM Owt
 IMPLICIT NONE
  CHARACTER(LEN=*), PARAMETER :: &
     long_name = "Llanfair...gogogoch"
  REAL :: x, y, z
  LOGICAL :: lacigol
   x = 1; y = 2; z = 3
   lacigol = (y .eq. x)
   PRINT*, long_name
   PRINT*, "Spock says ""illogical&
            &Captain"" "
   PRINT*, "X = ",x," Y = ",y," Z = ",z
   PRINT*, "Logical val: ",lacigol
END PROGRAM Owt
```

produces on the screen,

```
Llanfair...gogogoch
Spock says "illogical Captain"
X =  1.000  Y =  2.000  Z =  3.000
Logical val:  F
```

`PRINT` **Statement**

Note,

- each `PRINT` statement begins a new line;
- the `PRINT` statement can transfer any object of intrinsic type to the standard output;
- strings may be delimited by the double or single quote symbols, " and ’;
- two occurrences of the string delimiter inside a string produce one occurrence on output;

`READ` Statement

`READ` accepts unformatted data from the standard input channel, for example, if the type declarations are the same as for the `PRINT` example,

```
READ*, long_name
READ*, x, y, z
READ*, lacigol
```

accepts

```
Llanphairphwyll...gogogoch
0.4 5. 1.0e12
T
```

Note,

- each `READ` statement reads from a newline;
- the `READ` statement can transfer any object of intrinsic type from the standard input;

Mixed Type Numeric Expressions

In the CPU calculations must be performed between objects of the *same* type, so if an expression mixes type some objects must change type.

Default types have an implied ordering:

1. `INTEGER` — lowest
2. `REAL`
3. `DOUBLE PRECISION`
4. `COMPLEX` — highest

The result of an expression is always of the highest type, for example,

- `INTEGER * REAL` gives a `REAL`, (`3*2.0` is `6.0`)
- `REAL * INTEGER` gives a `REAL`, (`3.0*2` is `6.0`)
- `DOUBLE PRECISION * REAL` gives `DOUBLE PRECISION`,
- `COMPLEX *` < *anytype* > gives `COMPLEX`,
- `DOUBLE PRECISION * REAL * INTEGER` gives `DOUBLE PRECISION`.

The actual operator is unimportant.

Mixed Type Assignment

Problems often occur with mixed-type arithmetic; the rules for type conversion are given below.

- `INTEGER` = `REAL` (or `DOUBLE PRECISION`)

  The RHS is evaluated, truncated (all the decimal places lopped off) then assigned to the LHS.

- `REAL` (or `DOUBLE PRECISION`) = `INTEGER`

  The RHS is promoted to be `REAL` and stored (approximately) in the LHS.

For example,

```
REAL :: a = 1.1, b = 0.1
INTEGER :: i, j, k
i = 3.9          ! i will be 3
j = -0.9         ! j will be 0
k = a - b        ! k will be 1 or 0
```

Note: as `a` and `b` stored approximately, the value of `k` is uncertain.

Integer Division

Confusion often arises about integer division; in short, division of two integers produces an integer result by truncation (towards zero). Consider,

```
REAL :: a, b, c, d, e
a = 1999/1000              ! LHS is 1
b = -1999/1000             ! LHS is -1
c = (1999+1)/1000          ! LHS is 2
d = 1999.0/1000            ! LHS is 1.999
e = 1999/1000.0            ! LHS is 1.999
```

- `a` is (about) `1.000`
- `b` is (about) `-1.000`
- `c` is (about) `2.000`
- `d` is (about) `1.999`
- `e` is (about) `1.999`

Great care must be taken when using mixed type arithmetic.

Intrinsic Procedures

Fortran 90 has 113 in-built or *intrinsic* procedures to perform common tasks efficiently, they belong to a number of classes:

- elemental such as:
  - mathematical, for example, `SIN` or `LOG`.
  - numeric, for example, `SUM` or `CEILING`;
  - character, for example, `INDEX` and `TRIM`;
  - bit, for example, `IAND` and `IOR`;
- inquiry, for example, `ALLOCATED` and `SIZE`;
- transformational, for example, `REAL` and `TRANSPOSE`;
- miscellaneous (non-elemental `SUBROUTINE`s), for example, `SYSTEM_CLOCK` and `DATE_AND_TIME`.

Note all intrinsics which take `REAL` valued arguments also accept `DOUBLE PRECISION` arguments.

Type Conversion Functions

It is easy to transform the type of an entity,

- `REAL(i)` converts `i` to a real approximation,
- `INT(x)` truncates `x` to the integer equivalent,
- `DBLE(a)` converts `a` to `DOUBLE PRECISION`,
- `IACHAR(c)` returns the position of `CHARACTER c` in the ASCII collating sequence,
- `ACHAR(i)` returns the i^{th} character in the ASCII collating sequence.

All above are intrinsic functions. For example,

```
PRINT*, REAL(1), INT(1.7), INT(-0.9999)
PRINT*, IACHAR('C'), ACHAR(67)
```

are equal to

```
1.000000 1 0
67 C
```

Mathematical Intrinsic Functions

Summary,

`ACOS(x)`	arccosine
`ASIN(x)`	arcsine
`ATAN(x)`	arctangent
`ATAN2(y,x)`	arctangent of complex number (x, y)
`COS(x)`	cosine where x is in radians
`COSH(x)`	hyperbolic cosine where x is in radians
`EXP(x)`	e raised to the power x
`LOG(x)`	natural logarithm of x
`LOG10(x)`	logarithm base 10 of x
`SIN(x)`	sine where x is in radians
`SINH(x)`	hyperbolic sine where x is in radians
`SQRT(x)`	the square root of x
`TAN(x)`	tangent where x is in radians
`TANH(x)`	tangent where x is in radians

Numeric Intrinsic Functions

Summary,

`ABS(a)`	absolute value
`AINT(a)`	truncates `a` to whole `REAL` number
`ANINT(a)`	nearest whole `REAL` number
`CEILING(a)`	smallest `INTEGER` greater than or equal to `REAL` number
`CMPLX(x,y)`	convert to `COMPLEX`
`DBLE(x)`	convert to `DOUBLE PRECISION`
`DIM(x,y)`	positive difference
`FLOOR(a)`	biggest `INTEGER` less than or equal to real number
`INT(a)`	truncates a into an `INTEGER`
`MAX(a1,a2,a3,...)`	the maximum value of the arguments
`MIN(a1,a2,a3,...)`	the minimum value of the arguments
`MOD(a,p)`	remainder function
`MODULO(a,p)`	modulo function
`NINT(x)`	nearest `INTEGER` to a `REAL` number
`REAL(a)`	converts to the equivalent `REAL` value
`SIGN(a,b)`	transfer of sign — `ABS(a)*(b/ABS(b))`

Lecture 3:

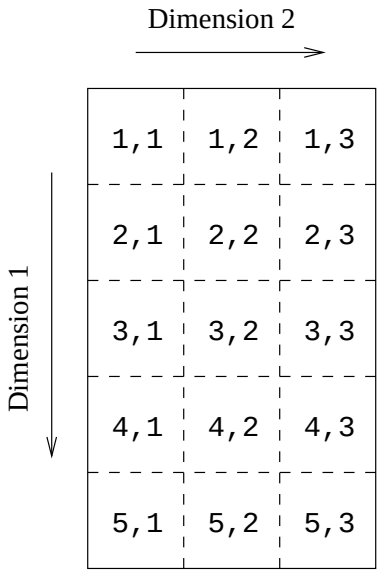
Arrays

Arrays

Arrays (or matrices) hold a collection of different values at the same time. Individual elements are accessed by **subscripting** the array.

A 15 element array can be visualised as:

And a 5 $\times$ 3 array as:



Every array has a type and each element holds a value of that type.

Array Terminology

Examples of declarations:

```
REAL, DIMENSION(15)       :: X
REAL, DIMENSION(1:5,1:3) :: Y, Z
```

The above are *explicit-shape* arrays.

Terminology:

- **rank** — number of dimensions.

  Rank of `X` is 1; rank of `Y` and `Z` is 2.

- **bounds** — upper and lower limits of indices.

  Bounds of `X` are 1 and 15; Bound of `Y` and `Z` are 1 and 5 and 1 and 3.

- **extent** — number of elements in dimension;

  Extent of `X` is 15; extents of `Y` and `Z` are 5 and 3.

- **size** — total number of elements.

  Size of `X`, `Y` and `Z` is 15.

- **shape** — rank and extents;

  Shape of `X` is 15; shape of `Y` and `Z` is 5,3.

- **conformable** — same shape.

  `Y` and `Z` are conformable.

Declarations

Literals and constants can be used in array declarations,

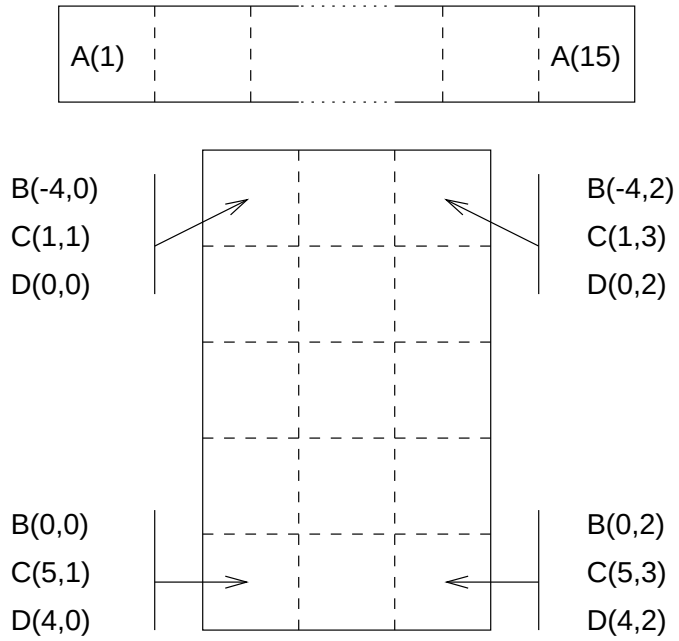
```
REAL, DIMENSION(100)         :: R
REAL, DIMENSION(1:10,1:10)   :: S
REAL                         :: T(10,10)
REAL, DIMENSION(-10:-1)      :: X
INTEGER, PARAMETER              :: lda = 5
REAL, DIMENSION(0:lda-1)        :: Y
REAL, DIMENSION(1+lda*lda,10) :: Z
```

- default lower bound is 1,
- bounds can begin and end anywhere,
- arrays can be zero-sized (if `lda = 0`),

Visualisation of Arrays

```
REAL, DIMENSION(15)        :: A
REAL, DIMENSION(-4:0,0:2)  :: B
REAL, DIMENSION(5,3)       :: C
REAL, DIMENSION(0:4,0:2)   :: D
```

Individual array elements are denoted by *subscripting* the array name by an `INTEGER`, for example, `A(7)` 7^{th} element of `A`, or `C(3,2)`, 3 elements down, 2 across.



Array Conformance

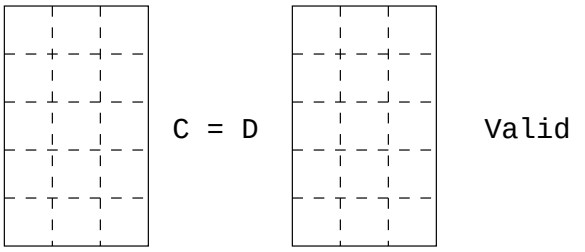
Arrays or sub-arrays must conform with all other objects in an expression:

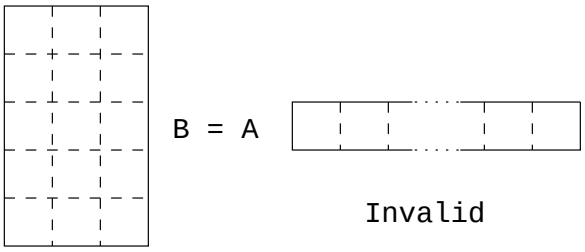
- a scalar conforms to an array of any shape with the same value for every element:

```
C = 1.0   ! is valid
```

- two array references must conform in their shape.

  Using the declarations from before:





  `A` and `B` have the same size but have different shapes so cannot be directly equated.

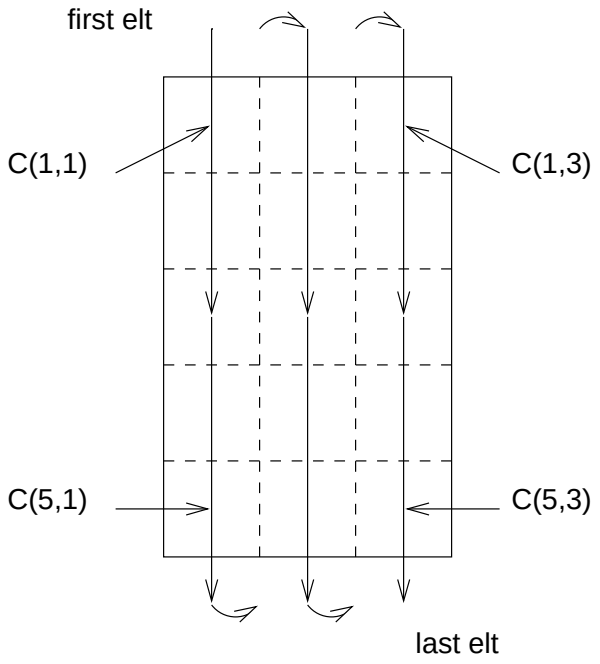
Array Element Ordering

Organisation in memory:

- Fortran 90 does not specify anything about how arrays should be located in memory. **It has no storage association.**

- Fortran 90 does define an array element ordering for certain situations which is of column major form,

The array is conceptually ordered as:



`C(1,1),C(2,1),..,C(5,1),C(1,2),C(2,2),..,C(5,3)`

Array Syntax

Can reference:

- whole arrays
  - `A = 0.0`
    sets whole array `A` to zero.
  - `B = C + D`
    adds `C` and `D` then assigns result to `B`.
- elements
  - `A(1) = 0.0`
    sets one element to zero,
  - `B(0,0) = A(3) + C(5,1)`
    sets an element of `B` to the sum of two other elements.
- array sections
  - `A(2:4) = 0.0`
    sets `A(2)`, `A(3)` and `A(4)` to zero,
  - `B(-1:0,1:2) = C(1:2,2:3) + 1.0`
    adds one to the subsection of `C` and assigns to the subsection of `B`.

Whole Array Expressions

Arrays can be treated like a single variable in that:

- can use intrinsic operators between conformable arrays (or sections),

```
B = C * D - B**2
```

this is equivalent to concurrent execution of:

```
B(-4,0) = C(1,1)*D(0,0)-B(-4,0)**2 ! in ||
B(-3,0) = C(2,1)*D(1,0)-B(-3,0)**2 ! in ||
 ...
B(-4,1) = C(1,2)*D(0,1)-B(-4,1)**2 ! in ||
 ...
B(0,2)  = C(5,3)*D(4,2)-B(0,2)**2  ! in ||
```

- elemental intrinsic functions can be used,

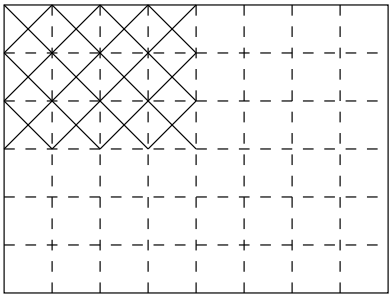
```
B = SIN(C)+COS(D)
```

the function is applied element by element.

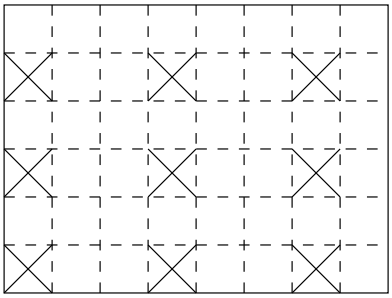
Array Sections — Visualisation

Given,

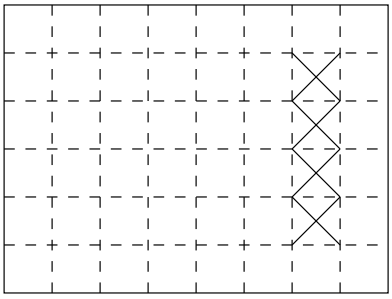
```
REAL, DIMENSION(1:6,1:8) :: P
```



P(1:3,1:4)

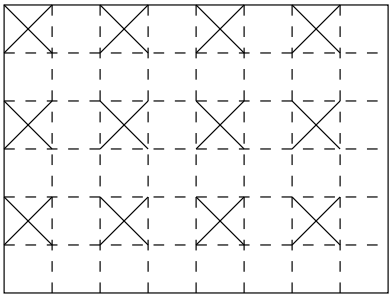


P(2:6:2,1:7:3)



P(2:5,7)

P(2:5,7:7)



P(1:6:2,1:8:2)

Consider the following assignments,

- `P(1:3,1:4) = P(1:6:2,1:8:2)` and `P(1:3,1:4) = 1.0` are valid.
- `P(2:8:2,1:7:3) = P(1:3,1:4)` and `P(2:6:2,1:7:3) = P(2:5,7)` are not.
- `P(2:5,7)` is a 1D section (scalar in dimension 2) whereas `P(2:5,7:7)` is a 2D section.

Array Sections

subscript-triplets specify sub-arrays. The general form is:

[< *bound1* >]:[< *bound2* >][:< *stride* >]

The section starts at < *bound1* > and ends at or before < *bound2* >. < *stride* > is the increment by which the locations are selected.

< *bound1* >, < *bound2* > and < *stride* > must all be scalar integer expressions. Thus

```
A(:)          ! the whole array
A(3:9)        ! A(m) to A(n) in steps of 1
A(3:9:1)      ! as above
A(m:n)        ! A(m) to A(n)
A(m:n:k)      ! A(m) to A(n) in steps of k
A(8:3:-1)     ! A(8) to A(3) in steps of -1
A(8:3)        ! A(8) to A(3) step 1 => Zero size
A(m:)         ! from A(m) to default UPB
A(:n)         ! from default LWB to A(n)
A(::2)        ! from default LWB to UPB step 2
A(m:m)        ! 1 element section
A(m)          ! scalar element - not a section
```

are all valid sections.

Array I/O

The conceptual ordering of array elements is useful for defining the order in which array elements are output. If `A` is a 2D array then:

```
PRINT*, A
```

would produce output in the order:

```
A(1,1),A(2,1),A(3,1),..,A(1,2),A(2,2),..
```

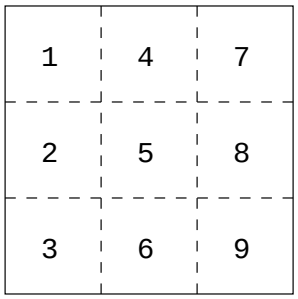
```
READ*, A
```

would assign to the elements in the above order.

This order could be changed by using intrinsic functions such as `RESHAPE`, `TRANSPOSE` or `CSHIFT`.

Array I/O Example

Consider the matrix `A`:



The following `PRINT` statements

```
    ...
     PRINT*, 'Array element   =',a(3,2)
     PRINT*, 'Array section   =',a(:,1)
     PRINT*, 'Sub-array       =',a(:2,:2)
     PRINT*, 'Whole Array     =',a
     PRINT*, 'Array Transp''d =',TRANSPOSE(a)
   END PROGRAM Owt
```

produce on the screen,

```
 Array element    = 6
 Array section    = 1 2 3
 Sub-array        = 1 2 4 5
 Whole Array      = 1 2 3 4 5 6 7 8 9
 Array Transposed = 1 4 7 2 5 8 3 6 9
```

Array Inquiry Intrinsics

These are often useful in procedures, consider the declaration:

```
REAL, DIMENSION(-10:10,23,14:28) :: A
```

- `LBOUND(SOURCE[,DIM])` — lower bounds of an array (or bound in an optionally specified dimension).
  - `LBOUND(A)` is `(/-10,1,14/)` (array);
  - `LBOUND(A,1)` is `-10` (scalar).
- `UBOUND(SOURCE[,DIM])` — upper bounds of an array (or bound in an optionally specified dimension).
- `SHAPE(SOURCE)` — shape of an array,
  - `SHAPE(A)` is `(/21,23,15/)` (array);
  - `SHAPE((/4/))` is `(/1/)` (array).
- `SIZE(SOURCE[,DIM])` — total number of array elements (in an optionally specified dimension),
  - `SIZE(A,1)` is 21;
  - `SIZE(A)` is 7245.
- `ALLOCATED(SOURCE)` — array allocation status;

Array Constructors

Used to give arrays or sections of arrays specific values. For example,

```
IMPLICIT NONE
INTEGER                           :: i
INTEGER, DIMENSION(10)            :: ints
CHARACTER(len=5), DIMENSION(3) :: colours
REAL, DIMENSION(4)                :: heights
heights = (/5.10, 5.6, 4.0, 3.6/)
colours = (/'RED  ','GREEN','BLUE '/)
! note padding so strings are 5 chars
ints    = (/ 100, (i, i=1,8), 100 /)
 ...
```

- constructors and array sections must conform.
- must be 1D.
- for higher rank arrays use `RESHAPE` intrinsic.
- `(i, i=1,8)` is an *implied* `DO` and is 1,2,..,8, it is possible to specify a stride.

The RESHAPE Intrinsic Function

RESHAPE is a general intrinsic function which delivers an array of a specific shape:

```
RESHAPE(SOURCE, SHAPE)
```

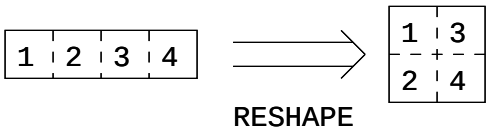
For example,

```
A = RESHAPE((/1,2,3,4/),(/2,2/))
```

A is filled in array element order and looks like:

```
1   3
2   4
```

Visualisation,



Array Constructors in Initialisation Statements

Arrays can be initialised

```
INTEGER, DIMENSION(4) :: solution = (/2,3,4,5/)
CHARACTER(LEN=*), DIMENSION(3) :: &
         lights = (/'RED  ','BLUE ','GREEN'/)
```

In the second statement all strings must be same length.

Named array constants may also be created:

```
INTEGER, DIMENSION(3), PARAMETER :: &
         Unit_vec = (/1,1,1/)
REAL, DIMENSION(3,3), PARAMETER :: &
         unit_matrix = &
            RESHAPE((/1,0,0,0,1,0,0,0,1/),(/3,3/))
```

Allocatable Arrays

Fortran 90 allows arrays to be created on-the-fly; these are known as *deferred-shape* arrays:

- Declaration:

```
INTEGER, DIMENSION(:), ALLOCATABLE :: ages   ! 1D
REAL, DIMENSION(:,:), ALLOCATABLE :: speed   ! 2D
```

  Note `ALLOCATABLE` attribute and fixed rank.

- Allocation:

```
READ*, isize
ALLOCATE(ages(isize), STAT=ierr)
IF (ierr /= 0) PRINT*, "ages : Allocation failed"

ALLOCATE(speed(0:isize-1,10),STAT=ierr)
IF (ierr /= 0) PRINT*, "speed : Allocation failed"
```

- the optional `STAT=` field reports on the success of the storage request. If the `INTEGER` variable `ierr` is zero the request was successful otherwise it failed.

Deallocating Arrays

Heap storage can be reclaimed using the `DEALLOCATE` statement:

```
IF (ALLOCATED(ages)) DEALLOCATE(ages,STAT=ierr)
```

- it is an error to deallocate an array without the `ALLOCATE` attribute or one that has not been previously allocated space,
- there is an intrinsic function, `ALLOCATED`, which returns a scalar `LOGICAL` values reporting on the status of an array,
- the `STAT=` field is optional but its use is recommended,
- if a procedure containing an allocatable array which does not have the `SAVE` attribute is exited without the array being `DEALLOCATE`d then this storage becomes inaccessible.

Masked Array Assignment — Where Statement

This is achieved using `WHERE`:

```
WHERE (I .NE. 0) A = B/I
```

the LHS of the assignment must be array valued and the mask, (the logical expression,) and the RHS of the assignment must all conform;

For example, if

$$\mathtt{B} = \begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}$$

and,

$$\mathtt{I} = \begin{pmatrix} \boxed{2} & 0 \\ 0 & \boxed{2} \end{pmatrix}$$

then

$$\mathtt{A} = \begin{pmatrix} \boxed{0.5} & . \\ . & \boxed{2.0} \end{pmatrix}$$

Only the indicated elements, corresponding to the non-zero elements of `I`, have been assigned to.

Where Construct

- there is a block form of masked assignment:

```
WHERE(A > 0.0)
 B = LOG(A)
 C = SQRT(A)
ELSEWHERE
 B = 0.0 ! C is NOT changed
ENDWHERE
```

- the mask must conform to the RHS of each assignment; `A, B` and `C` must conform;

- `WHERE ... END WHERE` is *not* a control construct and cannot currently be nested;

- the execution sequence is as follows: evaluate the mask, execute the `WHERE` block (in full) then execute the `ELSEWHERE` block;

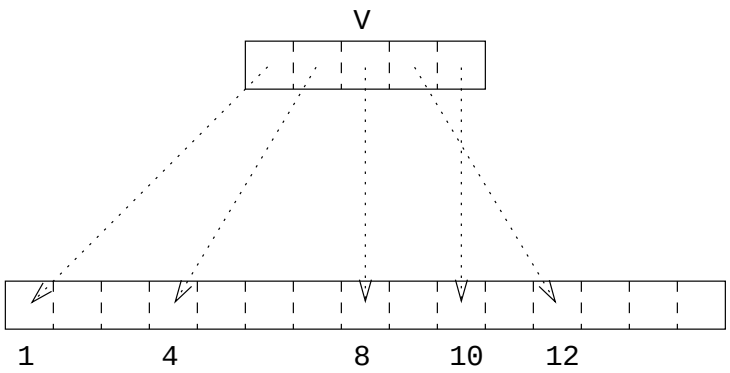
- the separate assignment statements are executed sequentially but the individual elemental assignments within each statement are (conceptually) executed in parallel.

Vector-valued Subscripts

A 1D array can be used to subscript an array in a dimension. Consider:

```
INTEGER, DIMENSION(5) :: V = (/1,4,8,12,10/)
INTEGER, DIMENSION(3) :: W = (/1,2,2/)
```

- `A(V)` is `A(1)`, `A(4)`, `A(8)`, `A(12)`, and `A(10)`.



- the following are valid assignments:

```
A(V) = 3.5
C(1:3,1) = A(W)
```

- it would be invalid to assign values to `A(W)` as `A(2)` is referred to twice.

- only 1D vector subscripts are allowed, for example,

```
A(1) = SUM(C(V,W))
```

Random Number Intrinsic

- `RANDOM_NUMBER(HARVEST)` will return a scalar (or array of) pseudorandom number(s) in the range $0 \leq x < 1$.

  For example,

```
REAL                      :: HARVEST
REAL, DIMENSION(10,10)    :: HARVEYS
CALL RANDOM_NUMBER(HARVEST)
CALL RANDOM_NUMBER(HARVEYS)
```

- `RANDOM_SEED([SIZE=`< *int* >`])` finds the size of the seed.

- `RANDOM_SEED([PUT=`< *array* >`])` seeds the random number generator.

```
CALL RANDOM_SEED(SIZE=isze)
CALL RANDOM_SEED(PUT=IArr(1:isze))
```

Vector and Matrix Multiply Intrinsics

There are two types of intrinsic matrix multiplication:

- `DOT_PRODUCT(VEC1, VEC2)` — inner (dot) product of two rank 1 arrays.

  For example,

  ```
  DP = DOT_PRODUCT(A,B)
  ```

  is equivalent to:

  ```
  DP = A(1)*B(1) + A(2)*B(2) + ...
  ```

  For `LOGICAL` arrays, the corresponding operation is a logical `.AND.`.

  ```
  DP = LA(1) .AND. LB(1) .OR. &
       LA(2) .AND. LB(2) .OR. ...
  ```

- `MATMUL(MAT1, MAT2)` — 'traditional' matrix-matrix multiplication:

  - if `MAT1` has shape (n, m) and `MAT2` shape (m, k) then the result has shape (n, k);
  - if `MAT1` has shape (m) and `MAT2` shape (m, k) then the result has shape (k);
  - if `MAT1` has shape (n, m) and `MAT2` shape (m) then the result has shape (n);

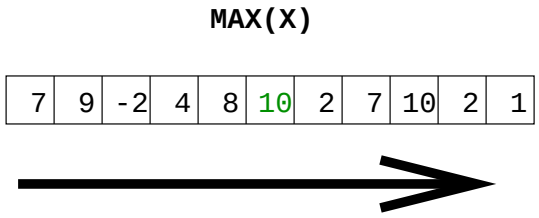
  For `LOGICAL` arrays, the corresponding operation is a logical `.AND.`.

Maximum and Minimum Intrinsics

There are two intrinsics in this class:

- `MAX(SOURCE1,SOURCE2[,SOURCE3[,...]])`— maximum values over all source objects
- `MIN(SOURCE1,SOURCE2[,SOURCE3[,...]])`— minimum values over all source objects

Scan from left to right, choose **first** occurrence if there are duplicates



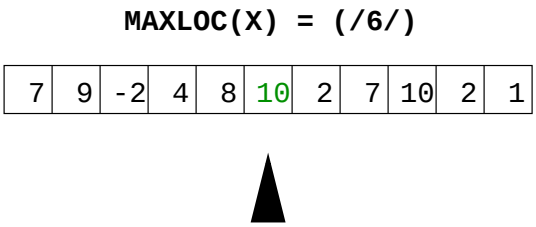
- `MAX(1,2,3)` is 3
- `MIN((/1,2/),(/-3,4/))` is `(/-3,2/)`
- `MAX((/1,2/),(/-3,4/))` is `(/1,4/)`

Array Location Intrinsics

There are two intrinsics in this class:

- `MINLOC(SOURCE[,MASK])`— Location of a minimum value in an array under an optional mask.
- `MAXLOC(SOURCE[,MASK])`— Location of a maximum value in an array under an optional mask.

A 1D example,



A 2D example. If

$$\texttt{Array} = \begin{pmatrix} 0 & -1 & 1 & 6 & -4 \\ 1 & -2 & 5 & 4 & -3 \\ 3 & 8 & 3 & -7 & 0 \end{pmatrix}$$

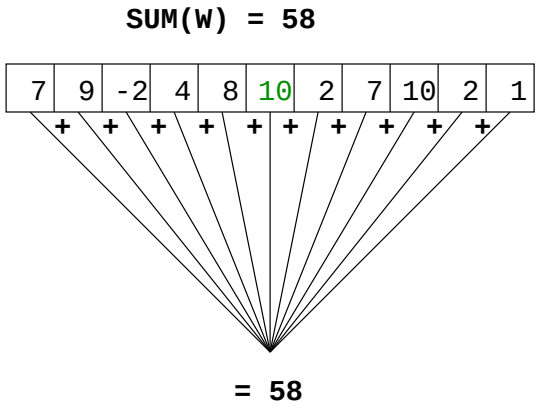
then

- `MINLOC(Array)` is `(/3,4/)`
- `MAXLOC(Array,Array.LE.7)` is `(/1,4/)`
- `MAXLOC(MAXLOC(Array,Array.LE.7))` is `(/2/)` (array valued).

Array Reduction Intrinsics

- `PRODUCT(SOURCE[,DIM][,MASK])`— product of array elements (in an optionally specified dimension under an optional mask);
- `SUM(SOURCE[,DIM][,MASK])`— sum of array elements (in an optionally specified dimension under an optional mask).

The following 1D example demonstrates how the 11 values are reduced to just one by the `SUM` reduction:



Consider this 2D example, if

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

- `PRODUCT(A)` is 720
- `PRODUCT(A,DIM=1)` is `(/2, 12, 30/)`
- `PRODUCT(A,DIM=2)` is `(/15, 48/)`

Array Reduction Intrinsics (Cont'd)

These functions operate on arrays and produce a result with less dimensions that the source object:

- `ALL(MASK[,DIM])`— `.TRUE.` if *all* values are `.TRUE.`, (in an optionally specified dimension);
- `ANY(MASK[,DIM])`— `.TRUE.` if *any* values are `.TRUE.`, (in an optionally specified dimension);
- `COUNT(MASK[,DIM])`— number of `.TRUE.` elements in an array, (in an optionally specified dimension);
- `MAXVAL(SOURCE[,DIM][,MASK])`— maximum Value in an array (in an optionally specified dimension under an optional mask);
- `MINVAL(SOURCE[,DIM][,MASK])`— minimum value in an array (in an optionally specified dimension under an optional mask);

If `DIM` is absent or the source array is of rank 1 then the result is scalar, otherwise the result is of rank $n - 1$.

Lecture 4:

Program Units and Interfaces

Program Units

Fortran 90 has two main program units

- main `PROGRAM`,

  the place where execution begins and where control should eventually return before the program terminates. May contain procedures.

- `MODULE`.

  a program unit which can contain procedures and declarations. It is intended to be attached to any other program unit where the entities defined within it become accessible.

There are two types of procedures:

- `SUBROUTINE`,

  a parameterised named sequence of code which performs a specific task and can be invoked from within other program units.

- `FUNCTION`,

  as a `SUBROUTINE` but returns a result in the function name (of any specified type and kind).

Main Program Syntax

```
PROGRAM Main
 ! ...
CONTAINS ! Internal Procs
   SUBROUTINE Sub1(..)
    ! Executable stmts
   END SUBROUTINE Sub1
    ! etc.
   FUNCTION Funkyn(...)
    ! Executable stmts
   END FUNCTION Funkyn
 END PROGRAM Main
```

[PROGRAM [< *main program name* >]]
        < *declaration of local objects* >
                . . .
        < *executable stmts* >
                . . .
[CONTAINS
        < *internal procedure definitions* >]
END [PROGRAM [< *main program name* >]]

Program Example

```
PROGRAM Main
 IMPLICIT NONE
 REAL :: x
  READ*, x
  PRINT*, FLOOR(x) ! Intrinsic
  PRINT*, Negative(x)
CONTAINS
 REAL FUNCTION Negative(a)
  REAL, INTENT(IN) :: a
   Negative = -a
 END FUNCTION Negative
END PROGRAM Main
```

Introduction to Procedures

The first question should be: "Do we really need to write a procedure?"

Functionality often exists,

- intrinsics, Fortran 90 has 113,
- libraries, for example, NAg `f190` Numerical Library has 300+, BLAS, IMSL, LaPACK, Uniras
- modules, number growing, many free! See WWW.

Library routines are usually *very fast*, sometimes even faster than Intrinsics!

Subroutines

Consider the following example,

```
PROGRAM Thingy
 IMPLICIT NONE
 .....
  CALL OutputFigures(NumberSet)
 .....
CONTAINS
 SUBROUTINE OutputFigures(Numbers)
  REAL, DIMENSION(:), INTENT(IN) :: Numbers
   PRINT*, "Here are the figures", Numbers
 END SUBROUTINE OutputFigures
END PROGRAM Thingy
```

Internal subroutines lie between `CONTAINS` and `END PROGRAM` statements and have the following syntax

`SUBROUTINE` < *procname* >[(< *dummy args* >)]
< *declaration of dummy args* >
< *declaration of local objects* >
. . .
< *executable stmts* >
`END` [`SUBROUTINE` [< *procname* >]]

Note that, in the example, the `IMPLICIT NONE` statement applies to the whole program including the `SUBROUTINE`.

Functions

Consider the following example,

```
PROGRAM Thingy
 IMPLICIT NONE
 .....
  PRINT*, F(a,b)
 .....
CONTAINS
 REAL FUNCTION F(x,y)
  REAL, INTENT(IN) :: x,y
   F = SQRT(x*x + y*y)
 END FUNCTION F
END PROGRAM Thingy
```

Functions also lie between `CONTAINS` and `END PROGRAM` statements. They have the following syntax:

[< *prefix* >] `FUNCTION` < *procname* >([< *dummyargs* >])
< *declaration of dummy args* >
< *declaration of local objects* >
. . .
< *executable stmts, assignment of result* >
`END` [`FUNCTION` [< *procname* >]]

It is also possible to declare the function type in the declarations area instead of in the header.

Argument Association

Recall, on the `SUBROUTINE` slide we had an invocation:

```
CALL OutputFigures(NumberSet)
```

and a declaration,

```
SUBROUTINE OutputFigures(Numbers)
```

`NumberSet` is an *actual argument* and is *argument associated* with the *dummy argument* `Numbers`.

For the above call, in `OutputFigures`, the name `Numbers` is an **alias** for `NumberSet`. Likewise, consider,

```
PRINT*, F(a,b)
```

and

```
REAL FUNCTION F(x,y)
```

The actual arguments `a` and `b` are associated with the dummy arguments `x` and `y`.

If the value of a dummy argument changes then so does the value of the actual argument.

Local Objects

In the following procedure

```
SUBROUTINE Madras(i,j)
 INTEGER, INTENT(IN) :: i, j
 REAL                :: a
 REAL, DIMENSION(i,j):: x
```

`a`, and `x` are know as *local objects*. They:

- are created each time a procedure is invoked,
- are destroyed when the procedure completes,
- *do not* retain their values between calls,
- do not exist in the programs memory between calls.

`x` will probably have a different size and shape on each call.

The space usually comes from the programs stack.

External Procedures

Fortran 90 allows a class of procedure that is not contained within a `PROGRAM` or a `MODULE` — an `EXTERNAL` procedure.

This is the old Fortran 77-style of programming and is more clumsy than the Fortran 90 way.

Differences:

- they may be compiled separately,
- may need an explicit `INTERFACE` to be supplied to the calling program,
- can be used as arguments (in addition to intrinsics),
- should contain the `IMPLICIT NONE` specifier.

Subroutine Syntax

Syntax of a (non-recursive) subroutine declaration:

```
SUBROUTINE Ext_1(...)
 ! ...
CONTAINS ! Internal Procs
   SUBROUTINE Int_1(...)
    ! Executable stmts
   END SUBROUTINE Int_1
    ! etc.
   FUNCTION Int_n(...)
    ! Executable stmts
   END FUNCTION Int_n
END SUBROUTINE Ext_1
```

```
SUBROUTINE Ext_2(...)
 ! etc
END SUBROUTINE Ext_2
```

SUBROUTINE < *procname* >[(< *dummy args* >)]
< *declaration of dummy args* >
< *declaration of local objects* >
. . .
< *executable stmts* >
[CONTAINS
< *internal procedure definitions* >]
END [SUBROUTINE [< *procname* >]]

External Subroutine Example

An external procedure may invoke a further external procedure,

```
SUBROUTINE sub1(a,b,c)
 IMPLICIT NONE
 EXTERNAL sum_sq  ! Should declare or use an INTERFACE
 REAL :: a, b, c, s
   ...
  CALL sum_sq(a,b,c,s)
  ...
END SUBROUTINE sub1
```

calls,

```
SUBROUTINE sum_sq(aa,bb,cc,ss)
 IMPLICIT NONE
 REAL, INTENT(IN) :: aa, bb, cc
 REAL, INTENT(OUT) :: ss
  ss = aa*aa + bb*bb + cc*cc
END SUBROUTINE sum_sq
```

Function Syntax

Syntax of a (non-recursive) function:

[< *prefix* >] `FUNCTION` < *procname* >([< *dummy args* >])
< *declaration of dummy args* >
< *declaration of local objects* >
. . .
< *executable stmts, assignment of result* >
[`CONTAINS`
< *internal procedure definitions* >]
`END` [`FUNCTION` [< *procname* >]]

here < *prefix* >, specifies the result type. or,

`FUNCTION` < *procname* >([< *dummy args* >])
< *declaration of dummy args* >
< *declaration of result type* >
< *declaration of local objects* >
. . .
< *executable stmts, assignment of result* >
[`CONTAINS`
< *internal procedure definitions* >]
`END` [`FUNCTION` [< *procname* >]]

here, < *procname* > must be declared.

External Function Example

- A function is invoked by its appearance in an expression at the place where its result value is needed,

```
total = total + largest(a,b,c)
```

- external functions should be declared as `EXTERNAL` or else the `INTERFACE` should be given,

```
INTEGER, EXTERNAL :: largest
```

- The function is defined as follows,

```
INTEGER FUNCTION largest(i,j,k)
 IMPLICIT NONE
 INTEGER :: i, j, k
 largest = i
 IF (j .GT. largest) largest = j
 IF (k .GT. largest) largest = k
END FUNCTION largest
```

or equivalently as,

```
FUNCTION largest(i,j,k)
 IMPLICIT NONE
 INTEGER :: i, j, k
 INTEGER :: largest
  ...
END FUNCTION largest
```

Argument Intent

Hints to the compiler can be given as to whether a dummy argument will:

- only be referenced — `INTENT(IN)`;
- be assigned to before use — `INTENT(OUT)`;
- be referenced and assigned to — `INTENT(INOUT)`;

```
SUBROUTINE example(arg1,arg2,arg3)
 REAL, INTENT(IN) :: arg1
 INTEGER, INTENT(OUT) :: arg2
 CHARACTER, INTENT(INOUT) :: arg3
 REAL :: r
  r = arg1*ICHAR(arg3)
  arg2 = ANINT(r)
  arg3 = CHAR(MOD(127,arg2))
END SUBROUTINE example
```

The use of `INTENT` attributes is recommended as it:

- allows good compilers to check for coding errors,
- facilitates efficient compilation and optimisation.

Note: if an actual argument is ever a literal, then the corresponding dummy must be `INTENT(IN)`.

Scoping Rules

Fortran 90 is *not* a traditional block-structured language:

- the *scope* of an entity is the range of program unit where it is visible and accessible;
- internal procedures can inherit entities by *host association*.
- objects declared in modules can be made visible by *use-association* (the `USE` statement) — useful for global data;

Host Association — Global Data

Consider,

```
PROGRAM CalculatePay
 IMPLICIT NONE
 REAL :: Pay, Tax, Delta
 INTEGER :: NumberCalcsDone = 0
 Pay = ...;  Tax = ... ; Delta = ...
 CALL PrintPay(Pay,Tax)
 Tax = NewTax(Tax,Delta)
  ....
CONTAINS
 SUBROUTINE PrintPay(Pay,Tax)
  REAL, INTENT(IN) :: Pay, Tax
  REAL :: TaxPaid
   TaxPaid = Pay * Tax
   PRINT*, TaxPaid
   NumberCalcsDone = NumberCalcsDone + 1
 END SUBROUTINE PrintPay
 REAL FUNCTION NewTax(Tax,Delta)
  REAL, INTENT(IN) :: Tax, Delta
   NewTax =  Tax + Delta*Tax
   NumberCalcsDone = NumberCalcsDone + 1
 END FUNCTION NewTax
END PROGRAM CalculatePay
```

Here, `NumberCalcsDone` is a *global* variable. It is available in all procedures by *host association*.

Scope of Names

Consider the following example,

```
PROGRAM Proggie
 IMPLICIT NONE
 REAL ::  A, B, C
 CALL sub(A)
CONTAINS
 SUBROUTINE Sub(D)
  REAL :: D       ! D is dummy (alias for A)
  REAL :: C       ! local C (diff from Proggie's C)
   C = A**3       ! A cannot be changed
   D = D**3 + C   ! D can be changed
   B = C          ! B from Proggie gets new value
 END SUBROUTINE Sub
END PROGRAM Proggie
```

In `Sub`, as `A` is argument associated it may not be have its value changed but may be referenced.

`C` in `Sub` is totally separate from `C` in `Proggie`, changing its value in `Sub` does **not** alter the value of `C` in `Proggie`.

SAVE Attribute and the SAVE Statement

SAVE attribute can be:

- applied to a specified variable. NumInvocations is initialised on **first** call and retains its new value between calls,

```
SUBROUTINE Barmy(arg1,arg2)
 INTEGER, SAVE :: NumInvocations = 0
  NumInvocations = NumInvocations + 1
```

- applied to the whole procedure, and applies to *all* local objects.

```
SUBROUTINE Mad(arg1,arg2)
 REAL :: saved
 SAVE
 REAL :: saved_an_all
```

Variables with the SAVE attribute are *static* objects.

Clearly, SAVE has no meaning in the main program.

Procedure Interfaces

For EXTERNAL procedures it is possible to provide an *explicit interface* for a procedure. Consider:

```
SUBROUTINE expsum( n, k, x, sum )! in interface
 USE KIND_VALS:ONLY long
 IMPLICIT NONE
 INTEGER, INTENT(IN) :: n         ! in interface
 REAL(long), INTENT(IN) :: k,x    ! in interface
 REAL(long), INTENT(OUT) :: sum   ! in interface
 REAL(long), SAVE :: cool_time
  sum = 0.0
  DO i = 1, n
   sum = sum + exp(-i*k*x)
  END DO
END SUBROUTINE expsum             ! in interface
```

The explicit INTERFACE for this routine is given by the statements which appear in the declarations part of any program unit that calls expsum:

```
INTERFACE            ! for EXTERNAL procedures
 SUBROUTINE expsum( n, k, x, sum )
  USE KIND_VALS:ONLY long
  INTEGER, INTENT(IN) :: n
  REAL(long), INTENT(IN) :: k,x
  REAL(long), INTENT(OUT)  :: sum
 END SUBROUTINE expsum
END INTERFACE
```

Interfaces replace any EXTERNAL statements and are *not* needed for internal (or module) procedures.

What Appears in an Interface?

An interface only contains:

- the `SUBROUTINE` or `FUNCTION` header,
- (if not included in the header) the `FUNCTION` type,
- declarations of the dummy arguments (including attributes),
- the `END SUBROUTINE` or `END FUNCTION` statement

Interfaces are only ever needed for `EXTERNAL` procedures and remove the need for any other form of declaration of that procedure.

Interface Example

The following program includes an explicit interface,

```
PROGRAM interface_example
 IMPLICIT NONE

 INTERFACE
  SUBROUTINE expsum(N,K,X,sum)
   INTEGER, INTENT(IN) :: N
   REAL, INTENT(IN) :: K,X
   REAL, INTENT(OUT) :: sum
  END SUBROUTINE expsum
 END INTERFACE

 REAL :: sum
  ...
 CALL expsum(10,0.5,0.1,sum)
  ...
END PROGRAM interface_example
```

Explicit interfaces permit separate compilation, optimisation and type checking.

Required Interfaces

Explicit interfaces are mandatory if an `EXTERNAL` procedure has:

- dummy arguments that are assumed-shape arrays, pointers or targets;
- `OPTIONAL` arguments;
- an array valued or pointer result (functions);
- a result that has an inherited `LEN=*` length specifier (character functions);

and when the reference:

- has a keyword argument;
- is a defined assignment;
- is a call to the generic name;
- is a call to a defined operator (functions).

Lecture 5:

Array Arguments, Intrinsics and Modules

Dummy Array Arguments

There are two main types of dummy array argument:

- *explicit-shape* — all bounds specified;

  ```
  REAL, DIMENSION(8,8), INTENT(IN) :: expl_shape
  ```

  The actual argument that becomes associated with an explicit-shape dummy must conform in size and shape.

- *assumed-shape* — no bounds specified, all inherited from the actual argument;

  ```
  REAL, DIMENSION(:,:), INTENT(IN) :: ass_shape
  ```

  An explicit interface *must* be provided.

- dummy arguments cannot be (unallocated) `ALLOCATABLE` arrays.

Assumed-shape Arrays

Should declare dummy arrays as assumed-shape arrays:

```
PROGRAM Main
 IMPLICIT NONE
  REAL, DIMENSION(40)    :: X
  REAL, DIMENSION(40,40) :: Y
   ...
  CALL gimlet(X,Y)
  CALL gimlet(X(1:39:2),Y(2:4,4:4))
  CALL gimlet(X(1:39:2),Y(2:4,4)) ! invalid
CONTAINS
 SUBROUTINE gimlet(a,b)
  REAL, INTENT(IN)   :: a(:), b(:,:)
    ...
 END SUBROUTINE gimlet
END PROGRAM
```

Note:

- the actual arguments cannot be a vector subscripted array,
- the actual argument cannot be an assumed-size array.
- in the procedure, bounds begin at 1.

Automatic Arrays

Other arrays can depend on dummy arguments, these are called *automatic* arrays and:

- their size is determined by dummy arguments,
- they cannot have the `SAVE` attribute (or be initialised);

Consider,

```
PROGRAM Main
 IMPLICIT NONE
  INTEGER :: IX, IY
 .....
  CALL une_bus_riot(IX,2,3)
  CALL une_bus_riot(IY,7,2)
CONTAINS
 SUBROUTINE une_bus_riot(A,M,N)
  INTEGER, INTENT(IN) :: M, N
  INTEGER, INTENT(INOUT) :: A(:,:)
  REAL :: A1(M,N)                    ! auto
  REAL :: A2(SIZE(A,1),SIZE(A,2)) ! auto
   ...
 END SUBROUTINE
END PROGRAM
```

The `SIZE` intrinsic or dummy arguments can be used to declare automatic arrays. `A1` and `A2` may have different sizes for different calls.

SAVE Attribute and Arrays

Consider,

```
SUBROUTINE sub1(dim)
 INTEGER, INTENT(IN)                          :: dim
 REAL, ALLOCATABLE, DIMENSION(:,:), SAVE :: X
 REAL, DIMENSION(dim)                         :: Y
  ...
  IF (.NOT.ALLOCATED(X)) ALLOCATE(X(20,20))
```

As X has the SAVE attribute it will retain its allocation status between calls otherwise it would disappear.

As Y depends on a dummy argument it cannot be given SAVE attribute.

Array-valued Functions

Functions can return arrays, for example,

```
PROGRAM Maian
 IMPLICIT NONE
  INTEGER, PARAMETER       :: m = 6
  INTEGER, DIMENSION(M,M) :: im1, im2
   ...
  IM2 = funnie(IM1,1) ! invoke
CONTAINS
 FUNCTION funnie(ima,scal)
  INTEGER, INTENT(IN) :: ima(:,:)
  INTEGER, INTENT(IN) :: scal
  INTEGER, DIMENSION(SIZE(ima,1),SIZE(ima,2)) &
                        :: funnie
  funnie(:,:) = ima(:,:)*scal
 END FUNCTION funnie
END PROGRAM
```

Note how the `DIMENSION` attribute **cannot** appear in the function header.

Modules — An Overview

The `MODULE` program unit provides the following facilities:

- global object declaration;
- procedure declaration (includes operator definition);
- semantic extension;
- ability to control accessibility of above to different programs and program units;
- ability to package together whole sets of facilities;

Module - General Form

```
MODULE Nodule
 ! TYPE Definitions
 ! Global data
  ! ..
 ! etc ..
CONTAINS
  SUBROUTINE Sub(..)
   ! Executable stmts
  CONTAINS
    SUBROUTINE Int1(..)
    END SUBROUTINE Int1
     ! etc.
    SUBROUTINE Intn(..)
    END SUBROUTINE Int2n
  END SUBROUTINE Sub
   ! etc.
  FUNCTION Funky(..)
   ! Executable stmts
  CONTAINS
     ! etc
  END FUNCTION Funky
END MODULE Nodule
```

MODULE < *module name* >
< *declarations and specifications statements* >
[CONTAINS
< *definitions of module procedures* >]
END [MODULE [< *module name* >]]

Modules — Global Data

Fortran 90 implements a new mechanism to implement global data:

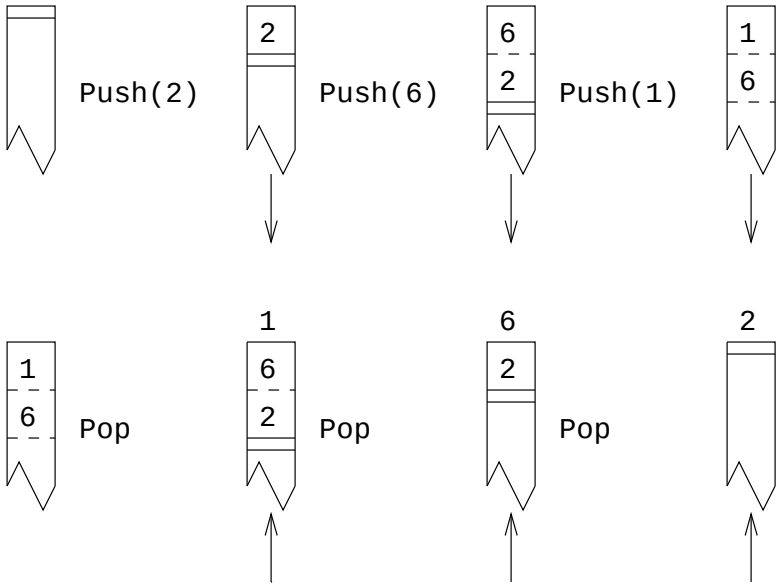
- declare the required objects within a module;
- give them the `SAVE` attribute;
- `USE` the module when global data is needed.

For example, to declare `pi` as a global constant

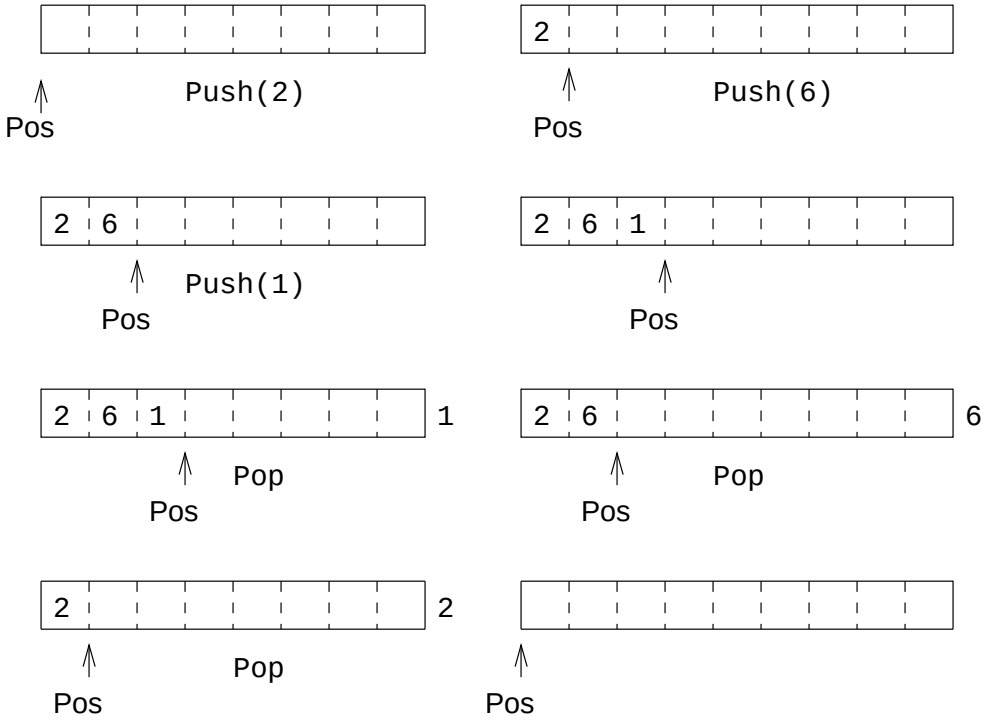
```
MODULE  Pye
 REAL, SAVE :: pi = 3.142
END MODULE Pye

PROGRAM Area
 USE Pye
 IMPLICIT NONE
 REAL :: r
  READ*, r
  PRINT*, "Area= ",pi*r*r
END PROGRAM Area
```

`MODULE`s should be placed *before* the program.

Stack Simulation



Implementation of a Stack



Module Global Data Example

For example, the following defines a very simple 100 element integer stack

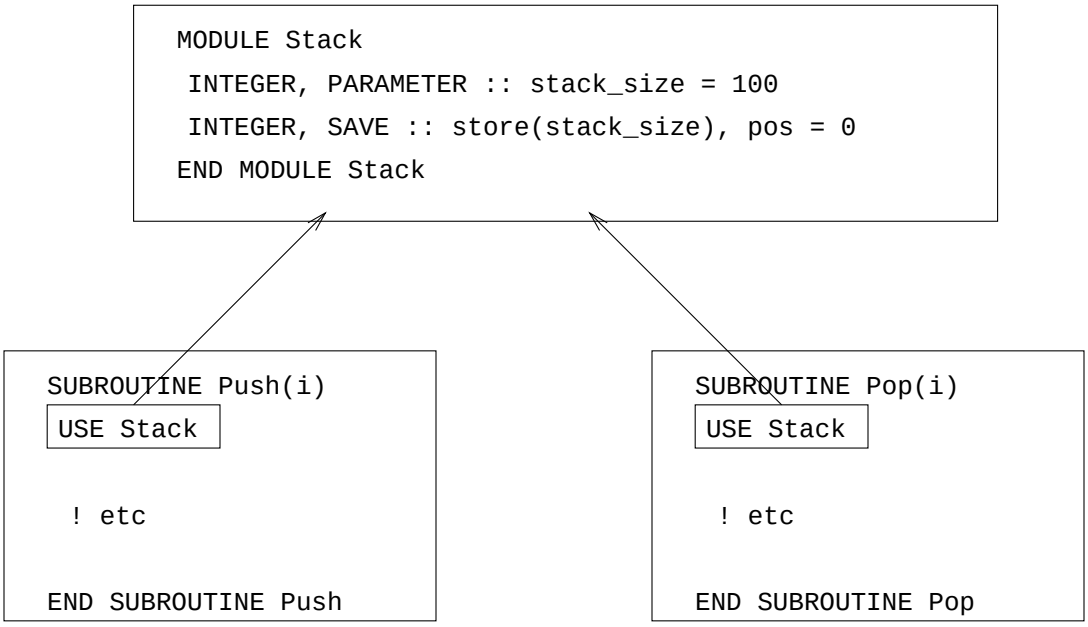
```
MODULE stack
 INTEGER, PARAMETER :: stack_size = 100
 INTEGER, SAVE :: store(stack_size), pos=0
END MODULE stack
```

and two access functions,

```
SUBROUTINE push(i)
 USE stack
 IMPLICIT NONE
  ...
END SUBROUTINE push
SUBROUTINE pop(i)
 USE stack
 IMPLICIT NONE
  ...
END SUBROUTINE pop
```

A main program can now call `push` and `pop` which simulate a 100 element `INTEGER` stack — this is much neater than using `COMMON` block.

Visualisation of Global Storage



Both procedures access the same (global) data in the MODULE.

Modules — Interface Declaration

It is good practice (and often mandatory) to explicitly declare procedure interfaces, however, in a single program there will be:

- a large number of procedures;
- a large amount of duplicated source with great opportunity for text mismatch;

can put *all* interfaces in a `MODULE`, all interfaces are visible when the module is used.

Modules Interface Declaration Example

Consider the following module containing procedure interfaces:

```
MODULE my_interfaces
 INTERFACE
  SUBROUTINE sub1(A,B,C)
   ... ! etc
  END SUBROUTINE sub1
  SUBROUTINE sub2(time,dist)
   ... ! etc
  END SUBROUTINE sub2
 END INTERFACE
END MODULE my_interfaces

PROGRAM use_of_module
 USE my_interfaces
 CALL sub1((/1.0,3.14,0.57/),2,'Yobot')
 CALL sub2(t,d)
END PROGRAM use_of_module
SUBROUTINE sub1(A,B,C)
  ...
END SUBROUTINE sub1
SUBROUTINE sub2(time,dist)
  ...
END SUBROUTINE sub2
```

The module containing the interfaces is used in the main program.

Modules — Procedure Declaration

Placing interfaces in modules for visibility purposes has the following problems,

- procedures may be inter-related and hence call one another;
- the procedure interfaces need to be explicit to one another;
- using the module means that each procedure would contain its *own* interface declaration which is an error.

```
SUBROUTINE Sub1(A,B,C)
 USE my_interfaces ! contains Sub1 stuff
  ...
```

The solution to this is to package the actual procedures in the module; the interfaces become visible and the above problems are solved.

These are now called *module procedures* and are *encapsulated* into the module.

Modules — Procedure Encapsulation

Module procedures are specified after the `CONTAINS` separator,

```
MODULE related_procedures
 IMPLICIT NONE
 ! INTERFACEs of MODULE PROCEDURES do
 ! not need to be specified they are
 ! 'already present'
CONTAINS
 SUBROUTINE sub1(A,B,C)
  ! Can see Sub2's INTERFACE
  ...
 END SUBROUTINE sub1
 SUBROUTINE sub2(time,dist)
  ! Can see Sub1's INTERFACE
  ...
 END SUBROUTINE sub2
END MODULE related_procedures
```

The main program attaches the procedures by *use-association*

```
PROGRAM use_of_module
 USE related_procedures ! includes INTERFACES
 CALL sub1((/1.0,3.14,0.57/),2,'Yobot')
 CALL sub2(t,d)
END PROGRAM use_of_module
```

`sub1` can call `sub2` or vice versa.

Encapsulation - Stack example

We can also encapsulate the `stack` program,

```
MODULE stack
 IMPLICIT NONE
 INTEGER, PARAMETER :: stack_size = 100
 INTEGER, SAVE :: store(stack_size), pos=0
CONTAINS
 SUBROUTINE push(i)
  INTEGER, INTENT(IN) :: i
   ...
 END SUBROUTINE push
 SUBROUTINE pop(i)
  INTEGER, INTENT(OUT) :: i
   ...
 END SUBROUTINE pop
END MODULE stack
```

Any program unit that includes the line:

```
USE stack
CALL push(2); CALL push(6); ..
CALL pop(i); ....
```

can access `pop` and `push` therefore use the 100 element global integer stack.

The USE Renames Facility

The USE statement names a module whose public definitions are to be made accessible.

Syntax:

USE < *module-name* > &
[,< *new-name* > => < *use-name* >...]

module entities can be renamed,

```
USE Stack, IntegerPop => Pop
```

The module object Pop is renamed to IntegerPop when used locally.

USE ONLY Statement

Another way to avoid name clashes is to only use those objects which are necessary. It has the following form:

USE < *module-name* > [ONLY:< *only*-list >...]

The < *only*-list > can also contain renames (=>).

For example,

```
USE Stack, ONLY:pos, &
       IntegerPop => Pop
```

Only pos and Pop are made accessible. Pop is renamed to IntegerPop.

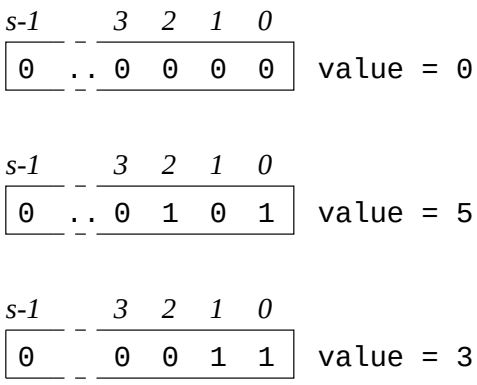
The ONLY statement gives the compiler the option of including only those entities specifically named.

Bit Manipulation Intrinsic Functions

Summary,

`BTEST(i,pos)`	bit testing
`IAND(i,j)`	AND
`IBCLR(i,pos)`	clear bit
`IBITS(i,pos,len)`	bit extraction
`IBSET(i,pos)`	set bit
`IEOR(i,j)`	exclusive OR
`IOR(i,j)`	inclusive OR
`ISHFT(i,shft)`	logical shift
`ISHFTC(i,shft)`	circular shift
`NOT(i)`	complement
`MVBITS(ifr,ifrpos, len,ito,itopos)`	move bits (`SUBROUTINE`)

Variables used as bit arguments must be `INTEGER` valued. The model for bit representation is that of an unsigned integer, for example,



The number of bits in a single variable depends on the compiler

Array Construction Intrinsics

There are four intrinsics in this class:

- `MERGE(TSOURCE,FSOURCE,MASK)`— merge two arrays under a mask,
- `SPREAD(SOURCE,DIM,NCOPIES)`— replicates an array by adding `NCOPIES` of a dimension,
- `PACK(SOURCE,MASK[,VECTOR])`— pack array into a one-dimensional array under a mask.
- `UNPACK(VECTOR,MASK,FIELD)`— unpack a vector into an array under a mask.

MERGE Intrinsic

MERGE(TSOURCE,FSOURCE,MASK)— merge two arrays under mask control. TSOURCE, FSOURCE and MASK must all conform and the result is TSOURCE where MASK is .TRUE. and FSOURCE where it is .FALSE..

If,

$$\texttt{MASK} = \begin{pmatrix} T & T & F \\ F & F & T \end{pmatrix}$$

and

$$\texttt{TSOURCE} = \begin{pmatrix} 1 & 5 & 9 \\ 3 & 7 & 11 \end{pmatrix}$$

and

$$\texttt{FSOURCE} = \begin{pmatrix} 0 & 4 & 8 \\ 2 & 6 & 10 \end{pmatrix}$$

we find

$$\texttt{MERGE(TSOURCE, FSOURCE, MASK)} = \begin{pmatrix} \boxed{1} & \boxed{5} & 8 \\ 2 & 6 & \boxed{11} \end{pmatrix}$$

SPREAD Intrinsic

SPREAD(SOURCE,DIM,NCOPIES)— replicates an array by adding NCOPIES of in the direction of a stated dimension.

If A is (/5, 7/), then

$$\texttt{SPREAD}(\texttt{A}, 2, 4) = \begin{pmatrix} 5 & 5 & 5 & 5 \\ 7 & 7 & 7 & 7 \end{pmatrix}$$

and

$$\texttt{SPREAD}(\texttt{A}, 1, 4) = \begin{pmatrix} 5 & 7 \\ 5 & 7 \\ 5 & 7 \\ 5 & 7 \end{pmatrix}$$

PACK Intrinsic

PACK(SOURCE,MASK[,VECTOR])— pack a arbitrary shaped array into a one-dimensional array under a mask. VECTOR, if present, must be 1-D and must be of same type and kind as SOURCE.

If

$$\texttt{MASK} = \begin{pmatrix} T & T & F \\ F & F & T \end{pmatrix}$$

and

$$\texttt{A} = \begin{pmatrix} \boxed{1} & \boxed{5} & 9 \\ 3 & 7 & \boxed{11} \end{pmatrix}$$

then

- PACK(A,MASK) is (/1, 5, 11/);
- PACK(A,MASK,(/3,4,5,6/)) is (/1, 5, 11, 6/).
- PACK(A,.TRUE.,(/1,2,3,4,5,6,7,8,9/)) is (/1,3,5,7,9,11,7,8,9/).

UNPACK Intrinsic

UNPACK(VECTOR,MASK,FIELD)— unpack a vector into an array under a mask. FIELD, must conform to MASK and and must be of same type and kind as VECTOR. The result is the same shape as MASK.

If

$$\texttt{FIELD} = \begin{pmatrix} 9 & 5 & 1 \\ 7 & 7 & 3 \end{pmatrix}$$

and

$$\texttt{MASK} = \begin{pmatrix} T & T & F \\ F & F & T \end{pmatrix}$$

then

$$\texttt{UNPACK}((/6,5,4/),\texttt{MASK},\texttt{FIELD}) = \begin{pmatrix} \boxed{6} & \boxed{5} & 1 \\ 7 & 7 & \boxed{4} \end{pmatrix}$$

and

$$\texttt{UNPACK}((/3,2,1/),\texttt{.NOT.MASK},\texttt{FIELD}) = \begin{pmatrix} 9 & 5 & \boxed{1} \\ \boxed{3} & \boxed{2} & 3 \end{pmatrix}$$

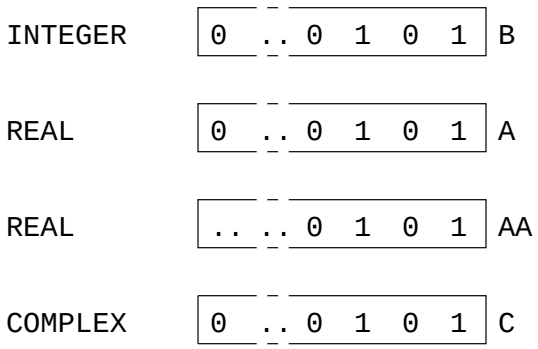
TRANSFER Intrinsic

TRANSFER converts (not coerces) physical representation between data types; it is a retyping facility. Syntax:

TRANSFER(SOURCE,MOLD)

- SOURCE is the object to be retyped,
- MOLD is an object of the target type.

```
REAL, DIMENSION(10)    :: A, AA
INTEGER, DIMENSION(20) :: B
COMPLEX, DIMENSION(5)  :: C
 ...
A  = TRANSFER(B, (/ 0.0 /))
AA = TRANSFER(B, 0.0)
C  = TRANSFER(B, (/ (0.0,0.0) /))
 ...
```



Lecture 6:

Data Parallelism

Parallel Processing

Parallelism:

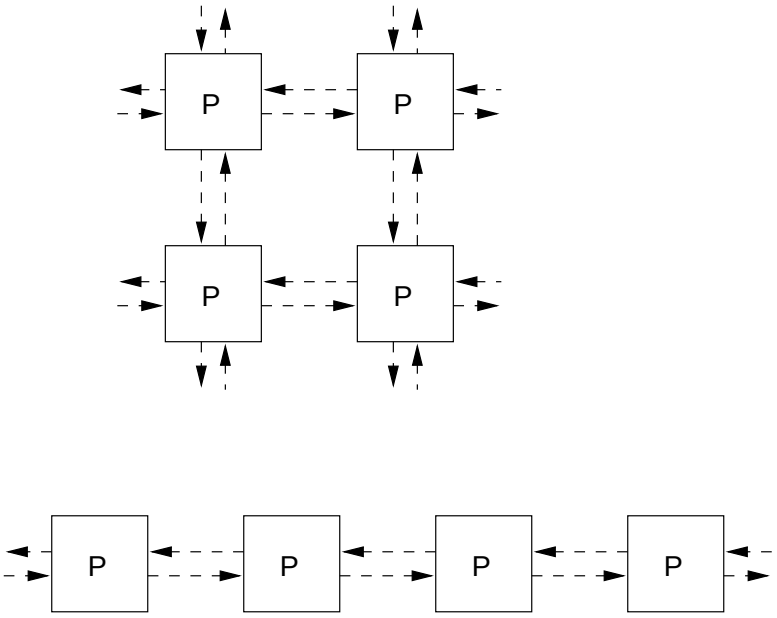
- at least 2 processors working together,
- work is partitioned — use idle machines,
- more processors can give faster execution,

To make parallelism effective, need to:

- minimise (communication time)/(computation time),
- balance load over processors,

Processor Configurations

Processors can be interconnected in a number of ways:



Data is distributed (or 'overlaid') onto the processor grids.

Parallel Programs

Three main classes,

- Data Parallelism,
- Task Parallelism,
- Master-Slave (subtasks),

High Performance Fortran (HPF) concentrates on Data Parallelism.

History of High Performance Fortran (HPF)

The High Performance Fortran Forum (HPFF)

- is an informal group formed at Supercomputing '91,
- produced a draft standard was circulated in Nov '92.
- communicated mainly by Email.

Their objectives were to design a language which:

1. supports data parallel programming,
2. obtains top performance on MIMD and SIMD,
3. supports code tuning.

The project was considered to be a success.

The Concept of HPF

HPF is a set of extensions to the Fortran 90 language. Fortran 90 considered a better platform than C or C++.

HPF targets the Data Parallel programming paradigm which allows systems of interconnected processors to be easily programmed,

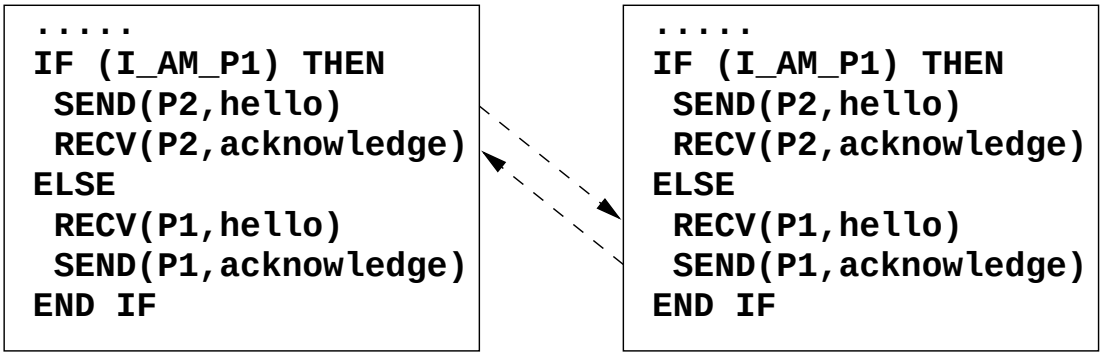
- each processor runs same program (SPMD)
- each processor operates on part of the overall data
- HPF directives say which processor gets what data
- executable statements define parallelism

Much simpler than writing message passing code - HPF brings parallel programming to the masses!

SPMD Model

SPMD model relies on:

- arrays of processors,
- distributed data,
- some global data,
- message passing between processors,
- loose synchronisation,



Processor Communications

Portable message passing systems,

- Message Passing Interface (MPI),
- Parallel Virtual Machine (PVM),

can operate on heterogeneous networks.

For example,

```
Send(to_pid,buffer,length)   ! Send buffer
Recv(from_pid,buffer,length) ! Receive buffer
Bcst(from_pid,buffer,length) ! Broadcast buffer
Barrier()                    ! Synchronise
```

HPF and Data Parallelism

In an HPF program we:

- define conceptual processor grids,
- distribute data onto processors,
- perform calculations,
  - Fortran 90,
  - `FORALL`,
  - `INDEPENDENT` loops,
  - `PURE` procedures,
  - `EXTRINSIC` (foreign) procedures,
  - HPF intrinsics and library (`MODULE`),

Fortran 90 features in High Performance Fortran

Fortran 90 features:

- Fortran 77,
- array syntax (including `WHERE`),
- allocatable arrays,
- most Fortran 90 intrinsics,
- take care with storage and sequence association and pointers!

Language Covered

HPF features:

- processor arrangements,
- static alignment,
- static distribution,
- `FORALL` statement,
- `INDEPENDENT` loops,
- `PURE` and `EXTRINSIC` procedures,

Many HPF features are now in Fortran 95.

HPF Directives

Can give 'hints' to the compiler:

`!HPF$` < *hpf-directive* >

Examples of declarative statements:

```
!HPF$ PROCESSORS, DIMENSION(16)  :: P
!HPF$ TEMPLATE, &
!HPF$           DIMENSION(4,4)   :: T1, T2
!HPF$ DISTRIBUTE                 :: A
!HPF$ DISTRIBUTE X(BLOCK)
!HPF$ DISTRIBUTE (CYCLIC)        :: Y1, Y2
!HPF$ DISTRIBUTE (BLOCK,*) ONTO P :: A
```

Note: all HPF names must be different to Fortran 90 names.

Examples of executable statements,

```
!HPF$ INDEPENDENT, NEW(i)
```

Compiler is at liberty to ignore any (or all) directives.

PROCESSORS Declaration

Can declare a conceptual processor grid.

```
!HPF$ PROCESSORS, DIMENSION(4)     :: P1
!HPF$ PROCESSORS, DIMENSION(2,2)   :: P2
!HPF$ PROCESSORS, DIMENSION(2,1,2) :: P3
```

All processor grids in the same program must have same SIZE.

Conceptual grids do not have to be the same shape as the underlying hardware.

DISTRIBUTE Directive

Distribute objects ONTO processor grids:

```
      REAL, DIMENSION(50)    :: A
      REAL, DIMENSION(10,10) :: B, C, D
!HPF$ DISTRIBUTE (BLOCK) ONTO P1          :: A   !1-D
!HPF$ DISTRIBUTE (CYCLIC,CYCLIC) ONTO P2 :: B,C !2-D
!HPF$ DISTRIBUTE D(BLOCK,*) ONTO P1    ! alt. syntax
```

There must be the same number of non-* distributed dimensions as the rank of the grid.

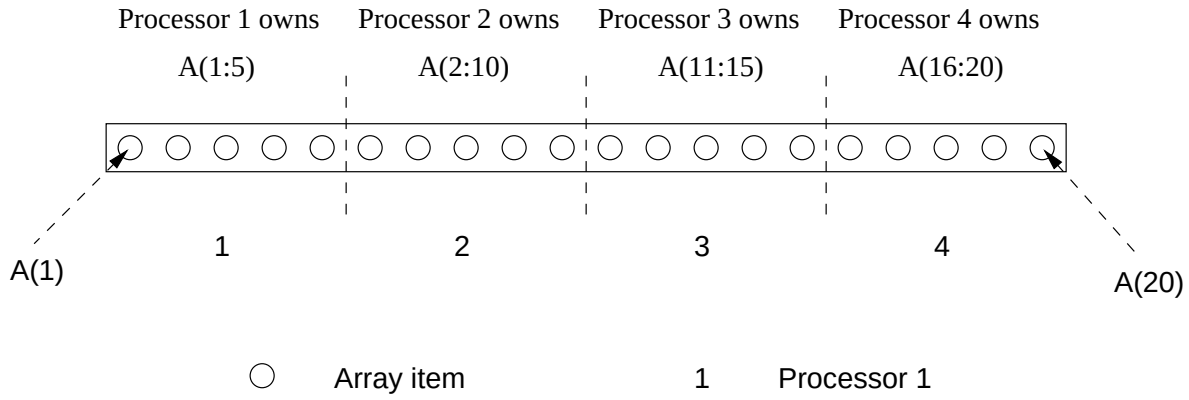
- BLOCK means give a continuous and, as far as possible, equal sized block of elements to each processor,
- CYCLIC means deal the elements out one at a time in a round-robin fashion,
- * means 'give this whole dimension to the processor'.

If an object is distributed then it is said to be **mapped** (or have a mapping).

Block Distribution

Give equal sized chunks of an array to each processor. For example,

```
      PROGRAM Chunks
      REAL, DIMENSION(20)        :: A
!HPF$ PROCESSORS, DIMENSION(4)   :: P
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A
  ....
```



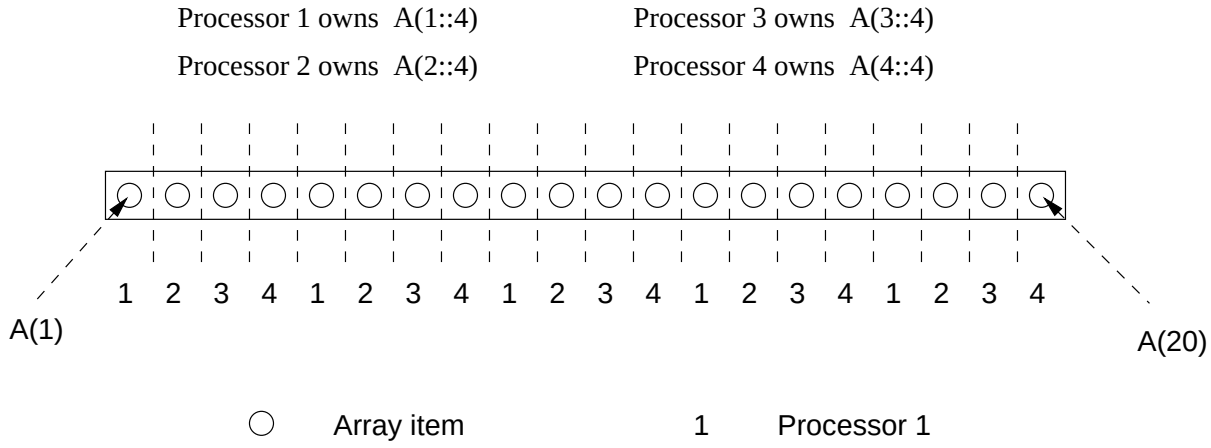
If an array, `A` has $\#A$ elements and is mapped onto $\#P$ processors each processor gets a *block* of (a maximum) of $\lceil \#A/\#P \rceil$ elements.

In this case each processor gets **five** elements.

Cyclic Distribution

Deal out elements of an array to processors in a round robin fashion

```
        PROGRAM Round_Robin
         REAL, DIMENSION(20)       :: A
!HPF$ PROCESSORS, DIMENSION(4)     :: P
!HPF$ DISTRIBUTE (CYCLIC) ONTO P :: A
  ....
```

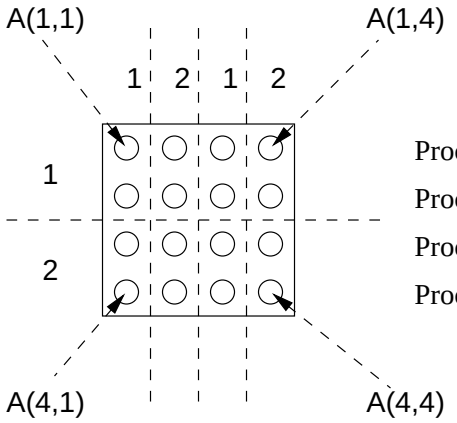


If an array, `A` has $\#A$ elements and is mapped onto $\#P$ processors each processor gets (a maximum) total of $\lceil \#A/\#P \rceil$ separate elements.

In this case each processor gets **five** elements.

2D Distribution Example

```
    PROGRAM Skwiffy
     IMPLICIT NONE
     REAL, DIMENSION(4,4)                :: A, B, C
!HPF$ PROCESSORS, DIMENSION(2,2)         :: P
!HPF$ DISTRIBUTE (BLOCK,CYCLIC) ONTO P :: A, B, C
      B = 1; C = 1; A = B + C
    END PROGRAM Skwiffy
```



Processor P(1,1) owns P(1:2,1::2)

Processor P(2,1) owns P(3:4,1::2)

Processor P(1,2) owns P(1:2,3::2)

Processor P(2,2) owns P(3:4,3::2)

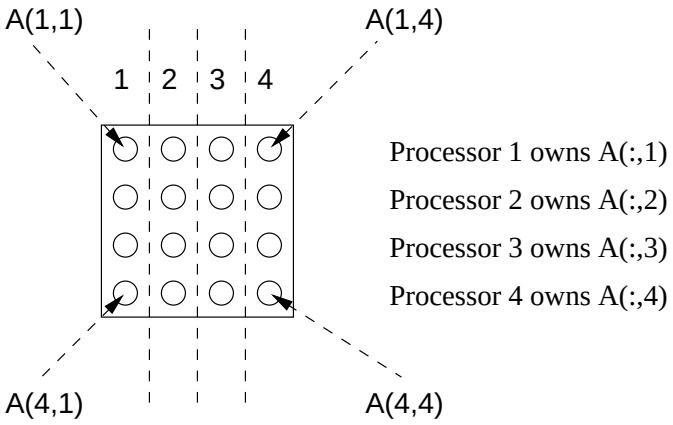
○ Array item

1 Processor 1

Visualisation of * Distribution

A * instead of a distribution method means: "do not distribute this dimension".

```
    PROGRAM Skwiffy
     IMPLICIT NONE
     REAL, DIMENSION(4,4)          :: A, B, C
!HPF$ PROCESSORS, DIMENSION(4)     :: Q
!HPF$ DISTRIBUTE (*,BLOCK) ONTO Q :: A, B, C
      B = 1; C = 1; A = B + C; PRINT*, A
    END PROGRAM Skwiffy
```



○ Array item

1 Processor 1

Each processor gets the whole dimension.

Commentary

- `ONTO` clause can be absent,
- `BLOCK` distribution is good when a computation accesses a group of 'close neighbours',
- `CYCLIC` distribution is good for balancing the load,
- collapsing a dimension is useful when a whole column or row is used in a single computation,
- in general, all scalars are *replicated* (one copy on each processor),

Distribution of Allocatables

Directives take effect at allocation, for example,

```
    REAL, ALLOCATABLE, DIMENSION(:,:) :: A
    INTEGER                           :: ierr
!HPF$ PROCESSORS, DIMENSION(10,10)    :: P
!HPF$ DISTRIBUTE (BLOCK,CYCLIC)       :: A
      ...
     ALLOCATE(A(100,20),stat=ierr)
!---> A automatically distributed here
!     block size in dim=1 is 10 elements
!     ...
     DEALLOCATE(A)
    END
```

The blocksize is determined immediately after allocation.

Once allocated, these arrays behave in the same way as regular arrays.

The Owner-Computes Rule

This is a rule oft-used in HPF Compilation Systems. It says that:

the processor that owns the left-hand side element will perform the calculation.

For example, in

```
DO i = 1,n
 a(i-1) = b(i*6)/c(i+j)-a(i**i)
END DO
```

the processor that owns `a(i-1)` will perform the assignment. The components of the RHS expression may have to be communicated to this processor before the assignment is made.

Scalar Variables

Unless explicitly mapped, scalar variables are generally *replicated*, in other words, *every* processor has a copy of the variable.

These copies must be kept up-to date (by the compiler). Consider,

```
     REAL, DIMENSION(100,100) :: X
     REAL                     :: Scal
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: X
       ....
      Scal = X(i,j)
       ....
```

The processor that owns `X(i,j)` updates its copy of `Scal` and then broadcasts its new value to *all* other processors.

Examples of Good Distributions

Given the following assignments:

```
A(2:99) = (A(:98)+A(3:))/2 ! neighbour calculations
B(22:56)= 4.0*ATAN(1.0)    ! section of B calculated
C(:) = SUM(D,DIM=1)        ! Sum down a column
```

Assuming the 'owner-computes' rule, the following distributions would be examples of good HPF programming,

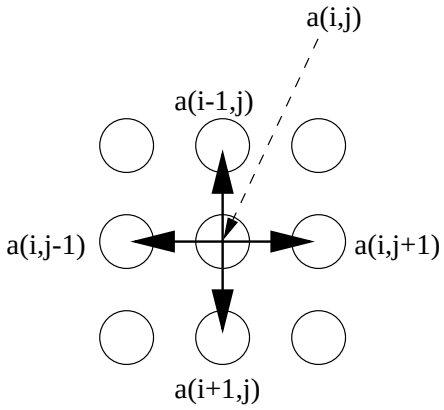
```
!HPF$ DISTRIBUTE (BLOCK) ONTO P   :: A
!HPF$ DISTRIBUTE (CYCLIC) ONTO P  :: B
!HPF$ DISTRIBUTE (BLOCK) ONTO P   :: C  ! or (CYCLIC)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO P :: D ! or (*,CYCLIC)
```

Successive Over Relaxation Example

Successive over relaxation is used in many HPF kernels:

```
DO j = 2,n-1
  DO i = 2,n-1
     a(i,j)=(omega/4)*(a(i,j-1)+a(i,j+1)+ &
              a(i-1,j)+a(i+1,j))+(1-omega)*a(i,j)
  END DO
END DO
```

The calculation of `a(i,j)` uses its 4 neighbours.



`BLOCK` distribution in both dimensions will be the most effective distribution here.

Other Mappings

```
!HPF$ DISTRIBUTE ONTO P         :: A
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: B
```

- distribution method missing — compiler dependent distribution, usually `BLOCK`,
- processor array missing – compiler dependent grid (determined at runtime),
- all directives missing — compiler dependent mapping, could be replicated.

HPF Programming Issues

HPF is always a trade-off between parallelism and communication,

- more processors, more communications,
- try to load balance, assume owner-computes rule,
- try to ensure data locality,
- use array syntax or array intrinsics,
- avoid storage and sequence association (and assumed-size arrays).

Need a good parallel algorithm.

HPF Programming Issues II

The following will slow down an HPF program,

- complicated subscript expressions,
- indirect addressing (vector subscripting),
- sequential (non-parallelisable) `DO`-loops,
- remapping objects,
- ill-chosen mappings,
- poor load balancing.

HPFacts

- Liverpools HPF Home Page

  `http://www.liv.ac.uk/HPC/HPCpage.html`

  has links to this course and other UK HPF material, eg, EPCC and MCC.

- Interactive HTML-based course

  `http://www.liv.ac.uk/HPC/HTMLHPFCourse/HTMLFrontPageHPF.html`

- HPFF Home Page

  `http://www.erc.msstate.edu/hpff/home.html`

- `HPF_LIBRARY`: Public Domain Fortran 90 version

  `http://www.lpac.ac.uk/SEL-HPC/Materials/HPFlibrary/HPFlibrary`

- **The** HPF book: *HPF Handbook*, by Koelbal *et al*. ISBN 0-262-61094-9.

Lecture 7:

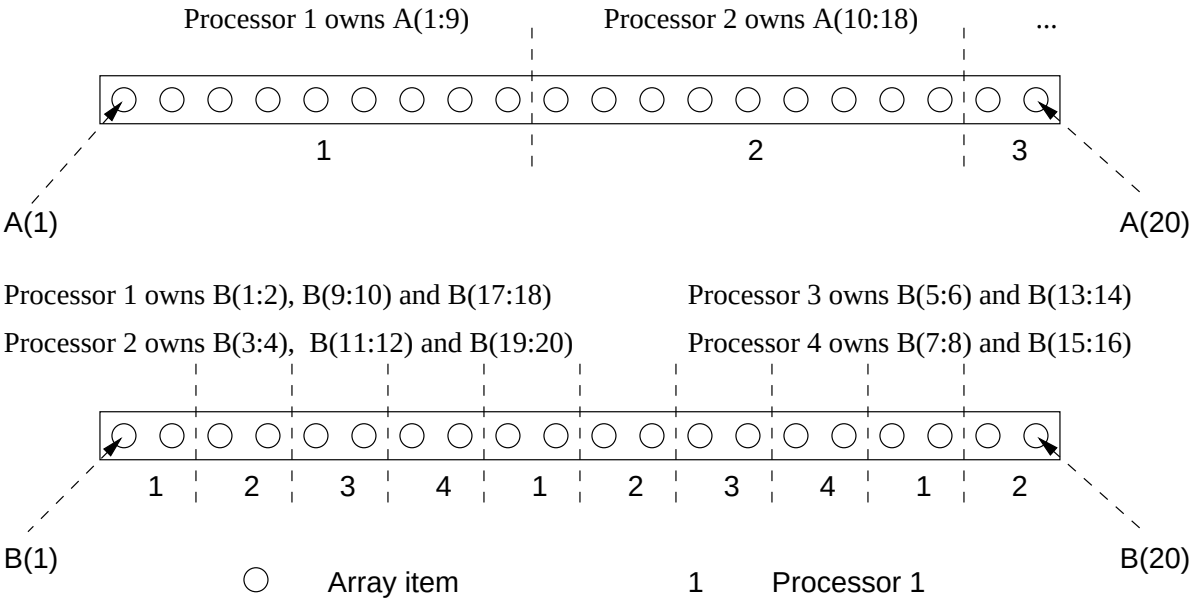
Alignment and Distribution

More Distributions

- `BLOCK(`m`)` — give processors m elements,
- `CYCLIC(`m`)` — distribute m elements in a round-robin fashion, will be inefficient.

For example,

```
     REAL, DIMENSION(20) :: A, B
!HPF$ PROCESSORS, DIMENSION(4) :: P
!HPF$ DISTRIBUTE A(BLOCK(9)) ONTO P
!HPF$ DISTRIBUTE B(CYCLIC(2)) ONTO P
```



2D Example

Consider the following 2D array `A`,

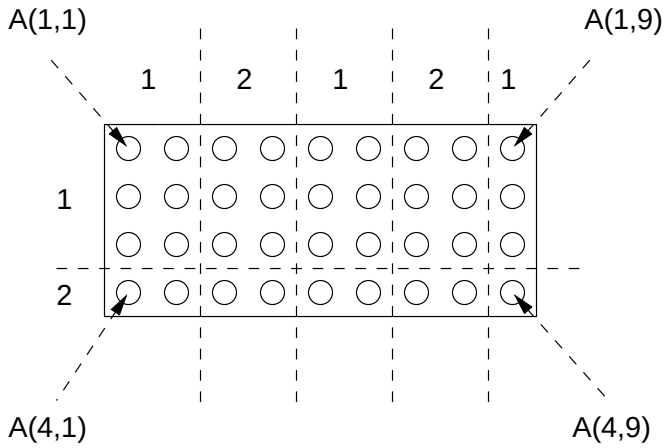
```
     REAL, DIMENSION(4,9)                          :: A
!HPF$ PROCESSORS, DIMENSION(2)                     :: P
!HPF$ DISTRIBUTE (BLOCK(3),CYCLIC(2)) ONTO P :: A
```

Processor P(1,1) owns A(1:3,1:2), A(1:3,5:6) and A(1:3,9:9)

Processor P(2,1) owns A(4:4,1:2), A(4:4,5:6) and A(4:4,9:9)

Processor P(1,2) owns A(1:3,3:4) and A(1:3,7:8)

Processor P(2,2) owns A(4:4,3:4) and A(4:4,7:8)



○ Array item

1 Processor 1

`BLOCK`(m) must 'use up' all the elements.

Alignment

Arrays can be positioned relative to each other,

- enhances data locality,
- minimises communication,
- distributes workload,
- allows replicated or collapsed dimensions,

Two aligned elements will reside on the same physical processor when distributed.

Alignment Syntax

The `ALIGN` statement can be written in two ways.

The attributed form

```
!HPF$ ALIGN (:,:) WITH T(:,:) :: A, B, C
```

is equivalent to

```
!HPF$ ALIGN A(:,:) WITH T(:,:)
!HPF$ ALIGN B(:,:) WITH T(:,:)
!HPF$ ALIGN C(:,:) WITH T(:,:)
```

which is more long-winded.

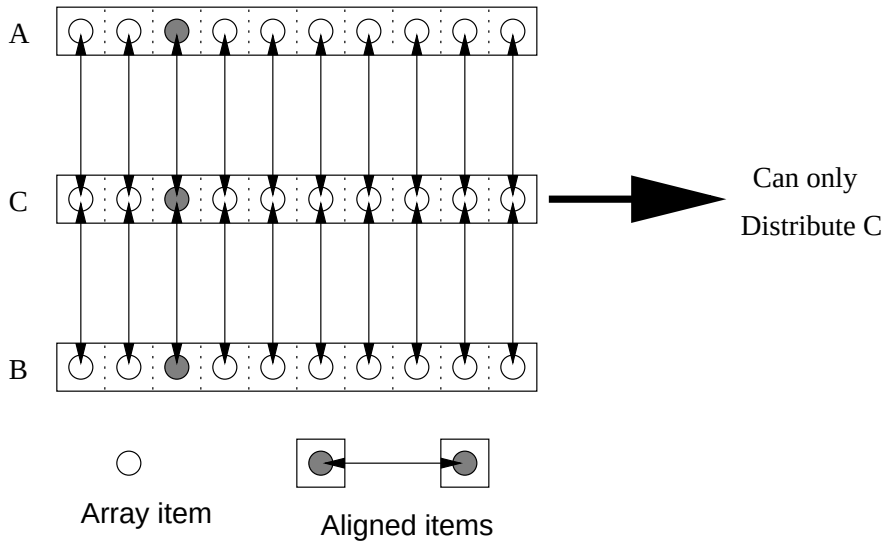
The effect here is that `A(i,j)`, `B(i,j)` and `C(i,j)` will reside on the same processor as `T(i,j)`.

Can only `DISTRIBUTE T`; `A`, `B` and `C` are linked to `T` and will follow.

Example and Visualisation

Simple example,

```
      REAL, DIMENSION(10) :: A, B, C
!HPF$ ALIGN (:) WITH C(:) :: A, B
```



The alignment says: `A(i)` and `B(i)` reside on same processor as `C(i)`. Because of the ':'s, `A`, `B` and `C` must conform. If we have

```
!HPF$ ALIGN (j) WITH C(j) :: A, B
```

then there is no requirement that the arrays conform.

Simple 2D Alignment Example

Given,

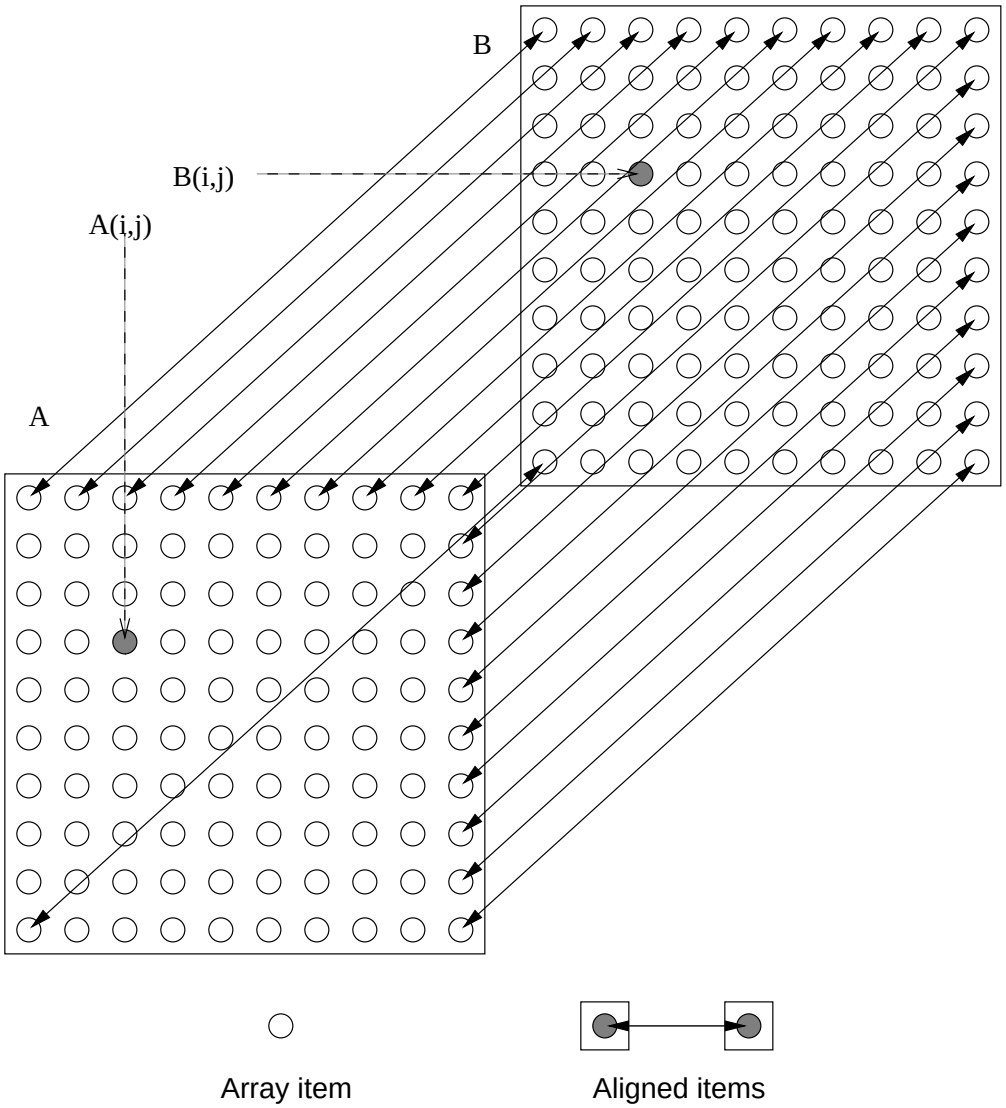
```
    REAL, DIMENSION(10,10) :: A, B
!HPF$ ALIGN A(:,:) WITH B(:,:)
```

This says: $\forall$ `i, j`, elements `A(i,j)` and `B(i,j)` are local.

The following align statement is equivalent but does not imply shape conformance:

```
!HPF$ ALIGN A(i,j) WITH B(i,j)
```

Visualisation of Simple Alignment Example



This alignment is suitable for,

```
A = A + B + A*B ! all local
```

Transposed Alignment Example

Align the first dimension of `A` with the second dimension of `B` (and *vice-versa*):

```
    REAL, DIMENSION(10,10) :: A, B
!HPF$ ALIGN A(i,:) WITH B(:,i)
```

This says: $\forall$ `i, j`, elements `A(i,j)` and `B(j,i)` are local. Could also be written:

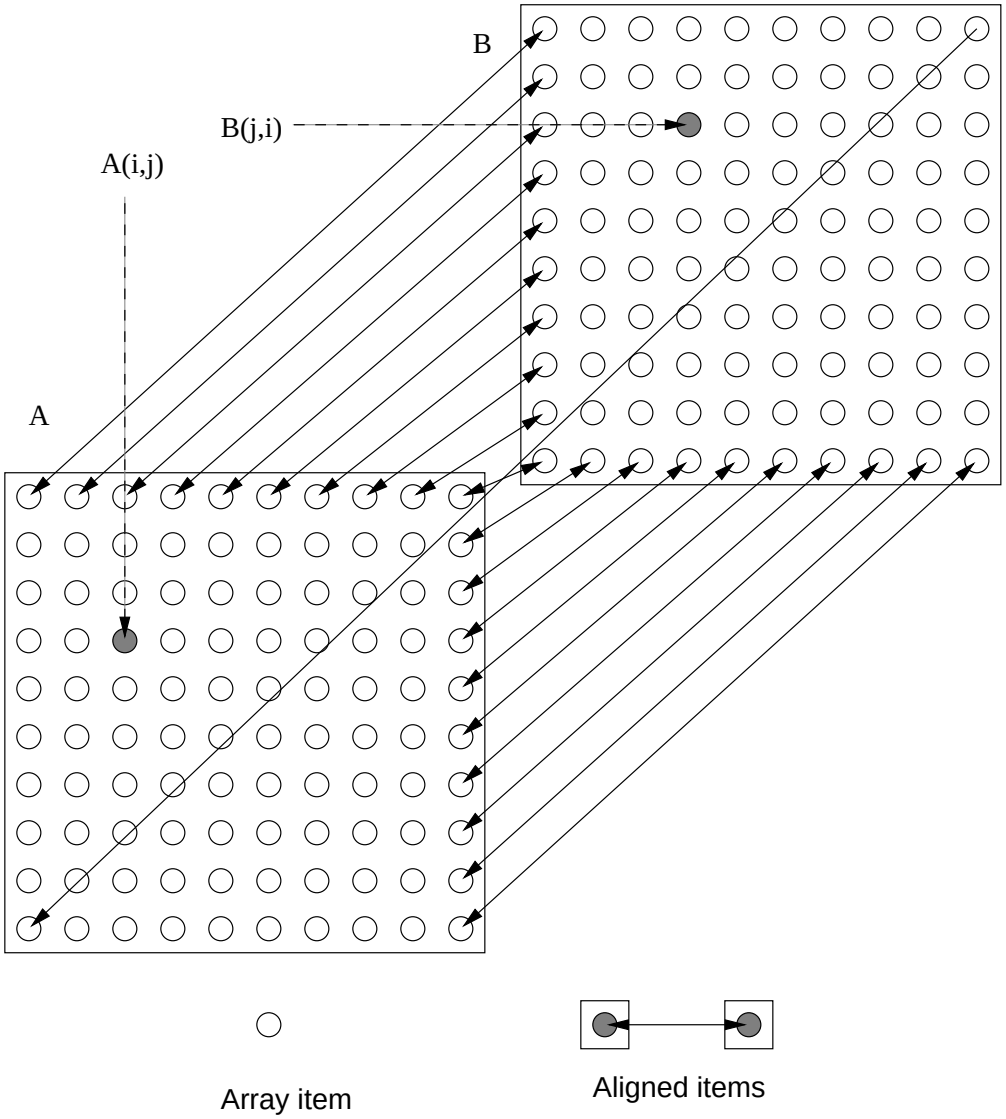
```
!HPF$ ALIGN A(:,j) WITH B(j,:)
```

or

```
!HPF$ ALIGN A(i,j) WITH B(j,i)
```

Here `i` and `j` are "symbols" not variables and are used to match dimensions their value (if any) is unimportant.

Visualisation of Transposed Alignment Example



This alignment is suitable for,

```
A = A + TRANSPOSE(B)*A ! all local
```

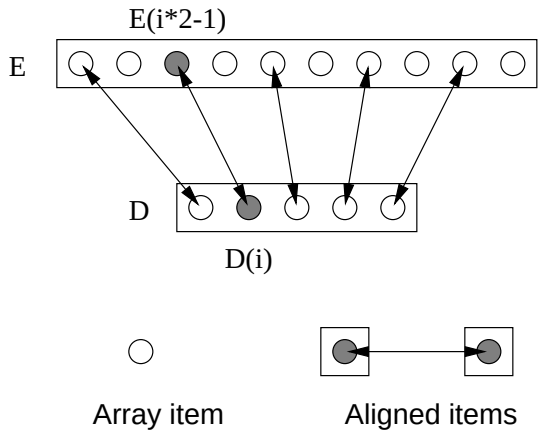
Strided Alignment Example

Align each element of `D` with every second element of `E`:

```
    REAL, DIMENSION(5)  :: D
    REAL, DIMENSION(10) :: E
!HPF$ ALIGN D(:) WITH E(1::2)
```

This says: $\forall$ `i`, elements `D(i)` and `E(i*2-1)` are aligned. For example, `D(3)` and `E(5)`. Alignment could also be written:

```
!HPF$ ALIGN D(i) WITH E(i*2-1)
```



This alignment is suitable for,

```
 D = D + E(::2) ! All local
```

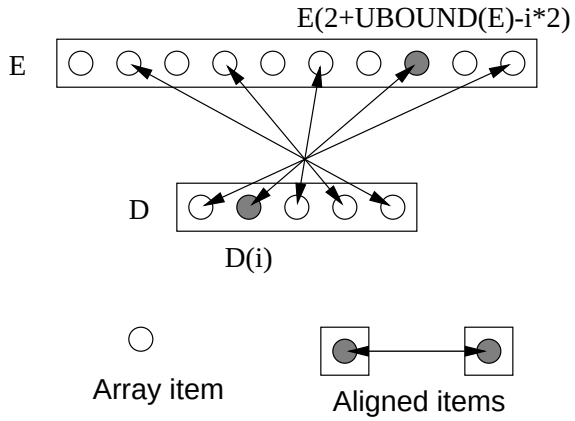
Reverse Strided Alignment Example

Can reverse an array before alignment:

```
    REAL, DIMENSION(5)  :: D
    REAL, DIMENSION(10) :: E
!HPF$ ALIGN D(:) WITH E(UBOUND(E)::-2)
```

This says: $\forall$ `i`, elements `E(2+UBOUND(E)-i*2)` and `D(i)` are local, for example, `D(1)` and `E(10)`. Alignment could also be written:

```
!HPF$ ALIGN D(i) WITH E(2+UBOUND(E)-i*2)
```



This alignment is suitable for,

```
 D = D + E(10:1:-2) ! All local
```

Practical Example of Alignment

The following Fortran 90 program:

```
PROGRAM Warty
 IMPLICIT NONE
 REAL, DIMENSION(4) :: C
 REAL, DIMENSION(8) :: D
 REAL, DIMENSION(2) :: E
  C = 1; D = 2
  E = D(::4) + C(::2)
END PROGRAM Warty
```

should be given these HPF directives to ensure minimal (zero) communications:

```
!HPF$ ALIGN C(:) WITH D(::2)
!HPF$ ALIGN E(:) WITH D(::4)
!HPF$ DISTRIBUTE (BLOCK) :: D
```

Note, cannot distribute `C` or `E`. Only distribute align targets.

Aligning Allocatable Arrays

Allocatable arrays may appear in `ALIGN` statements but

- the alignment takes place at allocation,
- an existing object may not be aligned with an unallocated object,

This means the array on the RHS, the *align-target*, of the `WITH` must be allocated before the array on the LHS is aligned to it.

Example

Given,

```
      REAL, DIMENSION(:), ALLOCATABLE :: A,B
!HPF$ ALIGN A(:) WITH B(:)
```

then,

```
      ALLOCATE (B(100),stat=ierr)
      ALLOCATE (A(100),stat=ierr)
```

is OK, as is

```
      ALLOCATE (B(100),A(100),stat=ierr)
```

because the *align-target*, `B`, exists before `A`, however,

```
      ALLOCATE (A(100),stat=ierr)
      ALLOCATE (B(100),stat=ierr)
```

is not, and neither is,

```
      ALLOCATE (A(100),B(100),stat=ierr)
```

because here the allocations take places from left to right.

Other Pitfalls

Clearly one cannot `ALIGN` a regular array `WITH` an allocatable:

```
      REAL, DIMENSION(:)                :: X
      REAL, DIMENSION(:), ALLOCATABLE :: A
!HPF$ ALIGN X(:) WITH A(:)                    ! WRONG
```

Another pitfall,

```
      REAL, DIMENSION(:), ALLOCATABLE :: A, B
!HPF$ ALIGN A(:) WITH B(:)
     ALLOCATE(B(100),stat=ierr)
     ALLOCATE(A(50),stat=ierr)
```

because, `A` and `B` are not conformable as suggested by `ALIGN` statement, however,

```
      REAL, DIMENSION(:), ALLOCATABLE :: A, B
!HPF$ ALIGN A(i) WITH B(i)
     ALLOCATE(B(100),stat=ierr)
     ALLOCATE(A(50),stat=ierr)
```

would be OK as the `ALIGN` statement does not imply conformance (no ':'s).

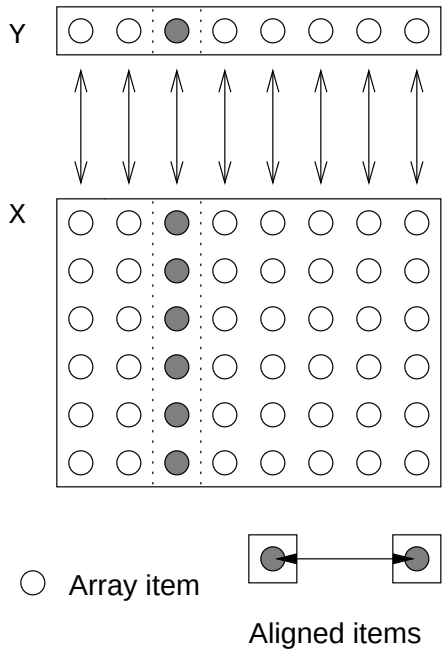
Here `A` cannot be larger than `B`.

Collapsing Dimensions

Can align one or more dimensions with a single element.

```
!HPF$ ALIGN (*,:) WITH Y(:) :: X
```

The `*` on the LHS of the `WITH` keyword, means that columns of `X` are not distributed. Each element of `Y` is aligned with a column of `X`.



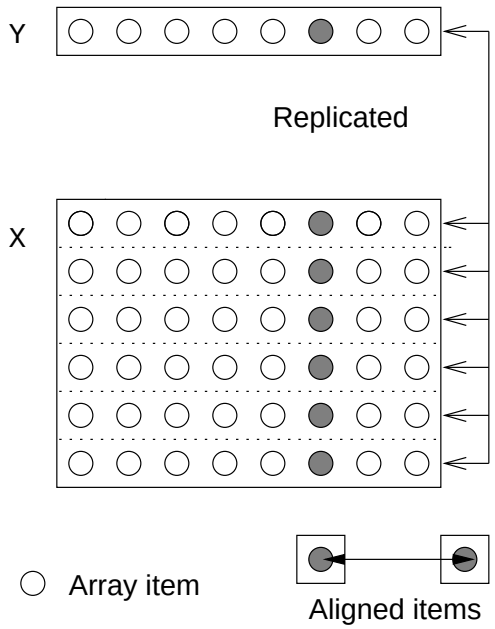
$\forall$ `i`, `X(:,i)` is local to `Y(i)`.

Replicating Dimensions

Can align elements with one or more dimensions.

```
!HPF$ ALIGN Y(:) WITH X(*,:)
```

The `*` on the RHS of the `WITH` keyword means that a copy of `Y` is aligned with every row of `X`.



$\forall$ `i`, `Y(i)` is local to `X(:,i)`.

Gaussian Elimination — 2D Grid

Consider kernel:

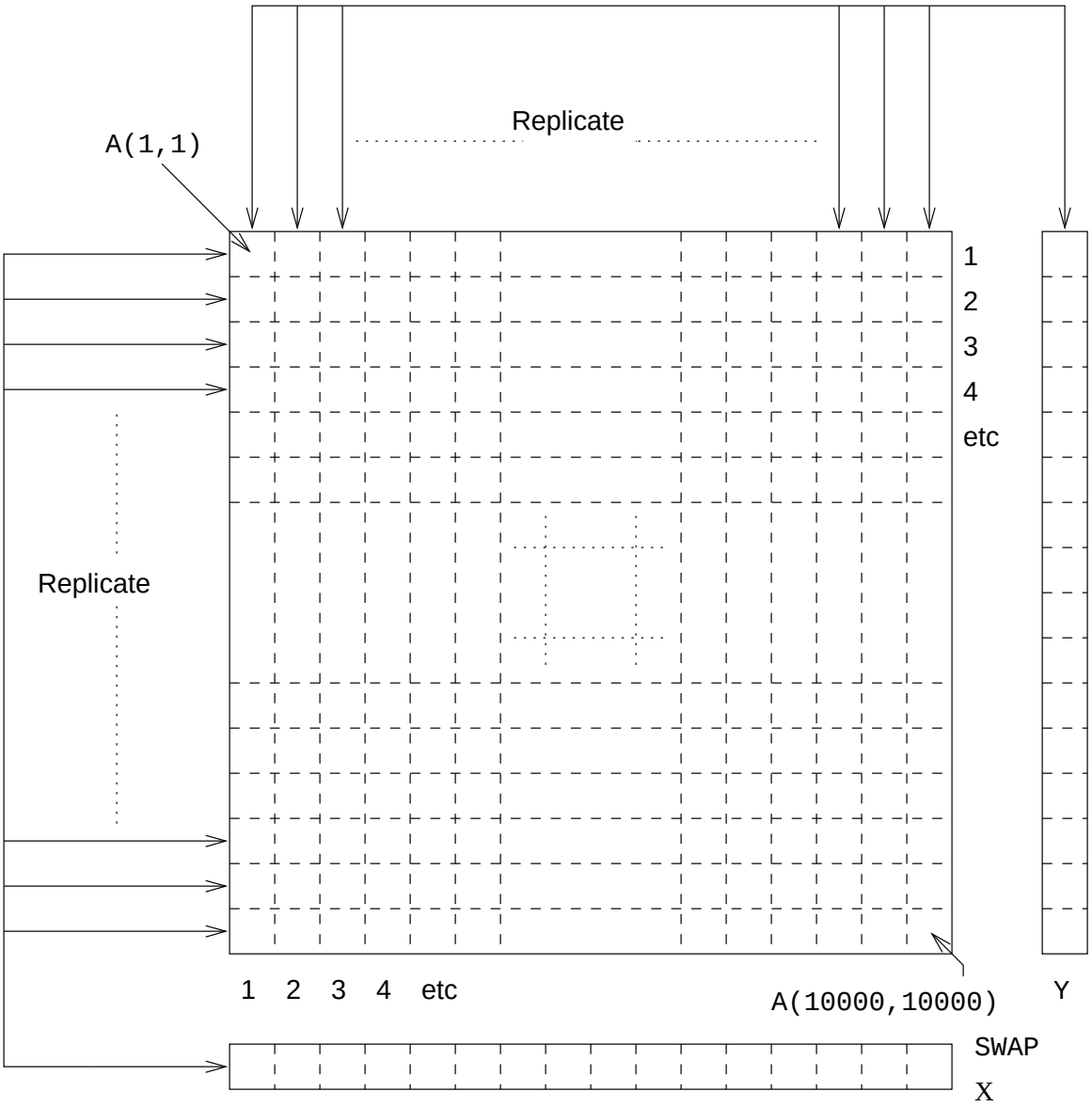
```
 ...
DO j = i+1, n
 A(j,i) = A(j,i)/Swap(i)
 A(j,i+1:n) = A(j,i+1:n) - A(j,i)*Swap(i+1:n)
 Y(j) = Y(j) - A(j,i)*Temp
END DO
```

Want to minimise communications in loop:

```
!HPF$ ALIGN Y(:) WITH A(:,*)
      ! Y aligned with each col of A
!HPF$ ALIGN Swap(:) WITH A(*,:)
      ! Swap aligned with each row of A
!HPF$ DISTRIBUTE A(CYCLIC,CYCLIC) ! onto default grid
```

CYCLIC gives a good load balance.

Visualisation of 2D Gaussian Elimination



New HPF Intrinsics

Can use NUMBER_OF_PROCESSORS intrinsic in initialisation expressions for portability,

```
!HPF$ PROCESSORS P1(NUMBER_OF_PROCESSORS())
!HPF$ PROCESSORS P2(4,4,NUMBER_OF_PROCESSORS()/16)
!HPF$ PROCESSORS P3(0:NUMBER_OF_PROCESSORS(1)-1, &
!HPF$                0:NUMBER_OF_PROCESSORS(2)-1)
```

NUMBER_OF_PROCESSORS returns information about physical processors.

Can obtain physical shape using PROCESSORS_SHAPE intrinsic, for example,

```
    PRINT*, PROCESSORS_SHAPE()
```

on a 2048 processor hypercube gives

```
 2 2 2 2 2 2 2 2 2 2 2
```

Template Syntax

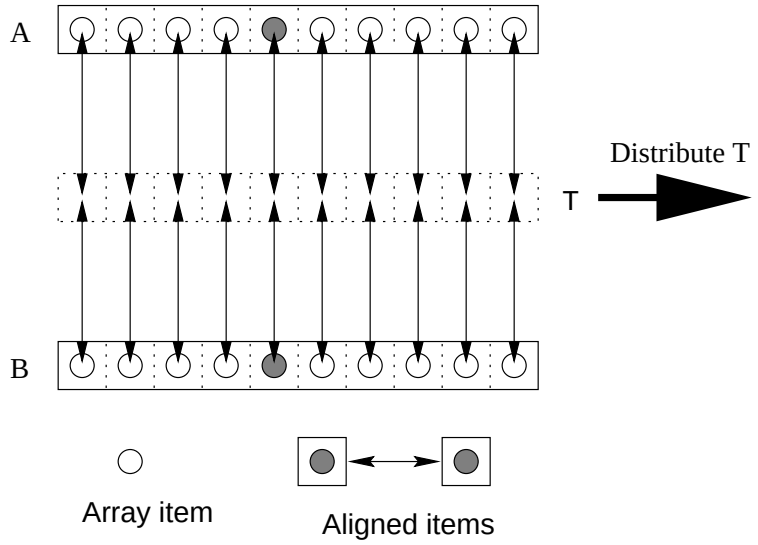
Templates are intended to make alignments easier and clearer.

- a `TEMPLATE` is declared,
- the `TEMPLATE` is distributed,
- arrays are aligned to it,

Simple Template Example

For example,

```
      REAL, DIMENSION(10)      :: A, B
!HPF$ TEMPLATE, DIMENSION(10) :: T
!HPF$ DISTRIBUTE (BLOCK)      :: T
!HPF$ ALIGN (:) WITH T(:)     :: A, B
```



Alternative Template Syntax

A TEMPLATE declaration has a combined form:

```
!HPF$ TEMPLATE, DIMENSION(100,100), &
!HPF$   DISTRIBUTE(BLOCK,CYCLIC) ONTO P :: T
!HPF$ ALIGN A(:,:) WITH T(:,:)
```

this is equivalent to

```
!HPF$ TEMPLATE, DIMENSION(100,100) :: T
!HPF$ ALIGN A(:,:) WITH T(:,:)
!HPF$ DISTRIBUTE T(BLOCK,CYCLIC) ONTO P
```

Thus, distribution is an attribute of a TEMPLATE.

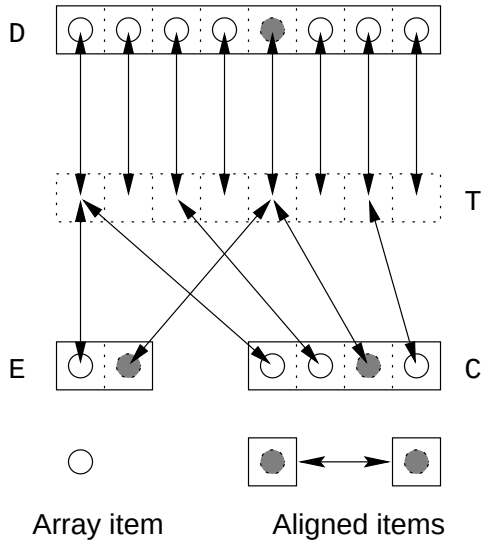
Another Template Example

Recall an earlier example,

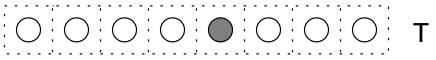
```
    PROGRAM Warty
     IMPLICIT NONE
     REAL, DIMENSION(4) :: C
     REAL, DIMENSION(8) :: D
     REAL, DIMENSION(2) :: E
!HPF$ TEMPLATE, DIMENSION(8) :: T
!HPF$ ALIGN D(:) WITH T(:)
!HPF$ ALIGN C(:) WITH T(::2)
!HPF$ ALIGN E(:) WITH T(::4)
!HPF$ DISTRIBUTE (BLOCK) :: T
      C = 1; D = 2
      E = D(::4) + C(::2)
    END PROGRAM Warty
```

this time the directives use an intermediate template.

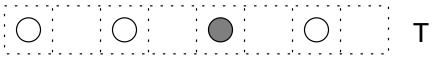
Visualisation



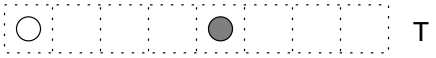
D is aligned with T as follows



C is aligned with T as follows



E is aligned with T as follows



Alignment and Distribution of Templates

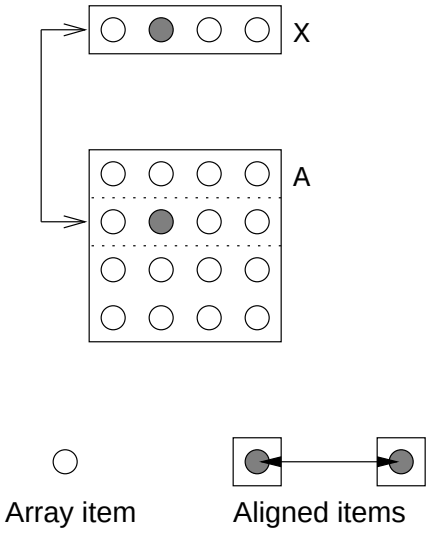
- `ALIGN A(:) WITH T1(:,*)` — ∀ `i`, element `A(i)` is replicated along row `T1(i,:)`.
- `ALIGN C(i,j) WITH T2(j,i)` — the transpose of `C` is aligned to `T1`,
- `ALIGN B(:,*) WITH T3(:)` — ∀ `i`, row `B(i,:)` is collapsed onto `TEMPLATE` element `T2(i)`,
- `DISTRIBUTE (BLOCK,CYCLIC) :: T1, T2`
- `DISTRIBUTE T1(CYCLIC,*) ONTO P` — rows of `T1` are distributed in a round-robin fashion.

Aligning to a Template Section

Is is possible to embed an array in a template:

```
    INTEGER :: i= 2
    REAL, DIMENSION(4) :: X
    REAL, DIMENSION(4,4) :: A
!HPF$ TEMPLATE, DIMENSION(4,4) :: T
!HPF$ ALIGN A(:,:) WITH T(:,:)
!HPF$ ALIGN X(:) WITH T(i,:) ! i used as variable
```

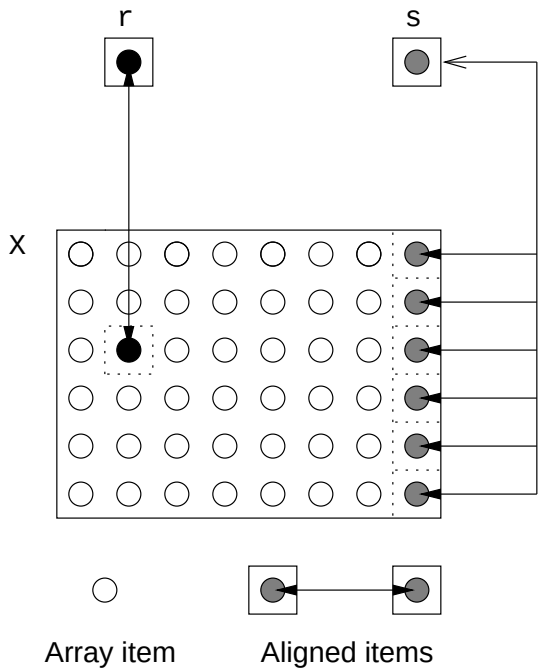
In this case `i` is *not* used as a symbol it needs a value. Since `i = 2` the section is aligned with row 2.



Aligning Scalars

HPF does not forbid mapping scalars [yet]:

```
      REAL, DIMENSION(6,8) :: X
      REAL r, s
!HPF$ ALIGN r WITH X(3,2)
!HPF$ ALIGN s WITH X(*,8)
```



Explicit Replication Using Templates

If an array is not explicitly distributed it is given the default mapping - often, but not always, replication.

To force replication of an array, `A`:

- declare a template to be the same size as the processor grid,
- align `A` with each template element,
- distribute the template

For example,

```
      REAL, DIMENSION(100,100)   :: A
!HPF$ PROCESSORS, DIMENSION(NUMBER_OF_PROCESSORS()) :: P
!HPF$ TEMPLATE, DIMENSION(NUMBER_OF_PROCESSORS())   :: T
!HPF$ ALIGN A(*,*) WITH T(*)
!HPF$ DISTRIBUTE (BLOCK) :: T
```

Lecture 8:

Forall and Independent Loops

Data Parallel Execution

Parallel execution is expressed via Fortran 90 array syntax and intrinsics. HPF adds:

- `FORALL` statement,

  flexible data parallel assignment statement.

- `PURE` procedures.

  'side-effect free' user procedures for parallel execution.

- `INDEPENDENT` directive,

  perform loop iterations in parallel,

- `NEW` variables,

  achieve independence using new variables.

Forall Statement

`FORALL` statement is in Fortran 95, syntax:

`FORALL(`< *forall-triplet*-list >[,< *scalar-mask* >])`&`
< *assignment-stmt* >

- expressive and concise parallel assignment,
- can be masked (cf. `WHERE` statement),
- can invoke `PURE` functions.

For example,

```
FORALL (i=1:n,j=1:m,A(i,j).NE.0) &
       A(i,j) = 1/A(i,j)
```

The stated assignment is performed *in parallel* for all specified values of `i` and `j` for which the mask expression is `.TRUE.`.

Forall Examples

FORALL is more versatile than array assignment:

- can access unusual sections,

```
FORALL (i=1:n) A(i,i) = B(i)  ! diagonal
DO j = 1, n
 FORALL (i=1:j) A(i,j) = B(i) ! triangular
END DO
```

- can use indices in RHS expression,

```
FORALL (i=1:n,j=1:n,i/=j) A(i,j) = REAL(i+j)
```

- can call PURE procedures,

```
FORALL (i=1:n:3,j=1:n:5) A(i,j) = SIN(A(j,i))
```

- can use indirection (vector subscripting),

```
FORALL (i=1:n,j=1:n) A(VS(i),j) = i+VS(j)
```

The above are *very difficult* to express in Fortran 90.

Execution Process

Execution is as follows,

1. evaluate subscript expressions (< *forall-triplet*-list >),

2. evaluate mask for all indices,

3. for all `.TRUE.` mask elements, evaluate whole of RHS of assignment,

4. assign RHSs to corresponding LHSs

Note, as always, parallel integrity must be maintained.

Do-loops and Forall Statements

Take care, FORALL semantics are different to DO-loop semantics,

```
DO i = 2, n-1
 a(i) = a(i-1) + a(i) + a(i+1)
END DO
```

is different to,

```
FORALL (i=2:n-1) &
 a(i) = a(i-1) + a(i) + a(i+1)
```

which is the same as,

```
 a(2:n-1) = a(1:n-2) + a(2:n-1) + a(3:n)
```

Forall Construct

`FORALL` construct is also in Fortran 95, syntax:

`FORALL(`< *forall-triplet*-list >[,< *scalar-mask* >])
< *assignment-stmt* >
....
`END FORALL`

For example,

```
FORALL (i=1:n:2, j=n:1:-2, A(i,j).NE.0)
       A(i,j) = 1/A(i,j) ! s1
       A(i,i) = B(i)     ! s2
END FORALL
```

`s1` executed first followed by `s2`

Can also nest `FORALL`s,

```
FORALL (i=1:3, j=1:3, i>j)
 WHERE  (ABS(A(i,i,j,j)) .LT. 0.1) A(i,i,j,j) = 0.0
 FORALL (k=1:3, l=1:j, k+l>i) A(i,j,k,l) = j*k+l
END FORALL
```

Pure Procedures

For example (in Fortran 95 and Full HPF),

```
PURE REAL FUNCTION F(x,y)
PURE SUBROUTINE G(x,y,z)
```

Side effect free:

- no external I/O or `ALLOCATE`,
- don't change global program state (global data),
- have `PURE` attribute,
- intrinsic / `ELEMENTAL` functions are pure,
- allowed in `FORALL` and pure procedures,

`PURE` procedures can be executed in parallel.

Pure Procedures

Must follow certain rules:

- `FUNCTION` dummy arguments must possess the `INTENT(IN)` attribute, `SUBROUTINE` dummys not restricted,
- local objects cannot be `SAVE`d,
- dummy arguments cannot be aligned to global objects,
- no `PAUSE` or `STOP` statement,
- other procedure invocations must be `PURE`.

Pure Function Example

Consider,

```
PURE REAL FUNCTION F(x,y)
 IMPLICIT NONE
 REAL, INTENT(IN) ::  x, y
  F = x*x + y*y + 2*x*y + ASIN(MIN(x/y,y/x))
END FUNCTION F
```

Here,

- arguments are unchanged,
- intrinsics are pure so can be used.

Example of use:

```
FORALL (i=1:n,j=1:n) &
 A(i,j) = b(i) + F(1.0*i,1.0*j)
```

Pure Subroutine Example

```
    PURE SUBROUTINE G(x,y,z)
     IMPLICIT NONE
     REAL, INTENT(OUT), DIMENSION(:) :: z
     REAL, INTENT(IN),  DIMENSION(:) ::  x, y
     INTEGER i
     INTERFACE
      REAL FUNCTION F(x,y)
       REAL, INTENT(IN) ::  x, y
      END FUNCTION F
     END INTERFACE
!      ...
      FORALL(i=1:SIZE(z)) z(i) = F(x(i),y(i))
    END SUBROUTINE G
```

Note:

- invokes pure procedure,
- interface *not* mandatory but *is* a very good idea.

Example of use,

```
    CALL G(x,y,res)
```

MIMD Example

Multiple Instructions Multiple Data,

```
REAL FUNCTION F(x,i) ! PURE
 IMPLICIT NONE
 REAL, INTENT(IN) :: x    ! element
 INTEGER, INTENT(IN) :: i ! index
  IF (x > 0.0) THEN
   F = x*x
  ELSEIF (i==1 .OR. i==n) THEN
   F = 0.0
  ELSE
   F = x
  END IF
END FUNCTION F
```

- different processors perform different tasks,
- used as alternative to `WHERE` or `FORALL`.

The INDEPENDENT Directive

The INDEPENDENT directive:

- can be applied to DO loops and FORALL assignments.
- asserts that no iteration affects any other iteration either directly or indirectly.

For DO-loops INDEPENDENT means the iterations or assignments can be performed *in any order*:

```
!HPF$ INDEPENDENT
DO i = 1,n
  x(i) = i**2
END DO
```

For FORALL statements INDEPENDENT means the whole RHS does not have to be evaluated before assignment to the LHS can begin,

```
!HPF$ INDEPENDENT
FORALL (i = 1:n) x(i) = i**2
```

Independent Loops — Conditions

`INDEPENDENT` loops cannot:

- assign to same element twice,
- contain `EXIT`, `STOP` or `PAUSE`,
- contain jumps out of loop,
- perform external I/O,
- prefix statements other than `DO` or `FORALL`.

Independent Example 1

This is independent,

```
!HPF$ INDEPENDENT
DO i = 1, n
 b(i) = b(i) + b(i)
END DO
```

this is not, (dependence on order of execution),

```
DO i = 1, n
 b(i) = b(i+1) + b(i)
END DO
```

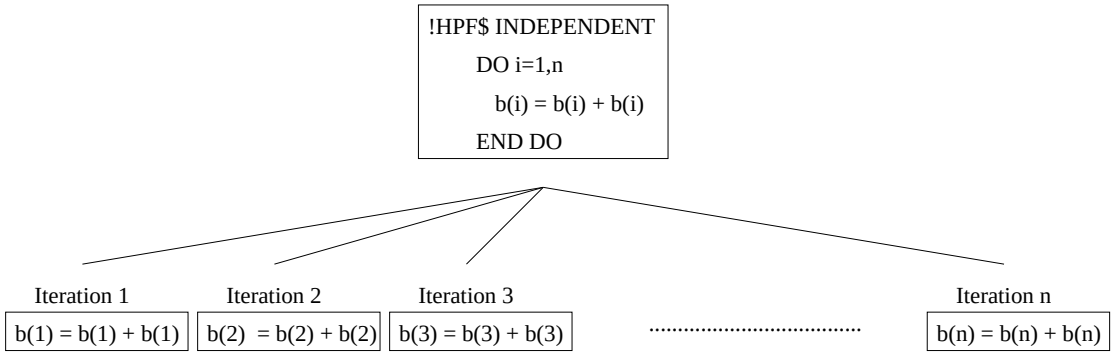
nor is this,

```
DO i = 1, n
 b(i) = b(i-1) + b(i)
END DO
```

however, this is

```
!HPF$ INDEPENDENT
    DO i = 1, n
     a(i) = b(i-1) + b(i)
    END DO
```

Visualisation of Independent Loop



All the iterations are performed at the same time.

Independent Example 2

Consider,

```
!HPF$ INDEPENDENT
DO i = 1, n
 a(i) = b(i-1) + b(i) + b(i+1)
END DO
```

Can perform all iterations in parallel. Also,

```
!HPF$ INDEPENDENT
FORALL (i=1:n) &
 a(i) = b(i-1) + b(i) + b(i+1)
```

don't have to calculate whole RHS before assignment.

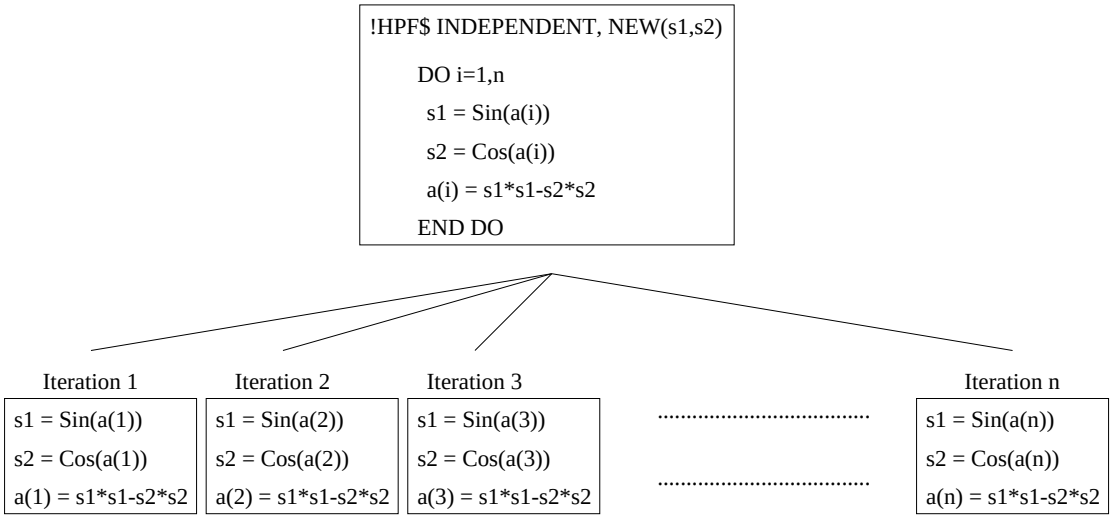
Can also use with vector subscripts,

```
!HPF$ INDEPENDENT
DO i = 1, n
 a(index(i)) =  b(i-1) + b(i) + b(i+1)
END DO
```

Says each element of `index(1:n)` is unique.

INDEPENDENT NEW Loops

In order to parallelise DO-loops NEW instances of s1 and s2 may be needed for each iteration of the loop:



Iteration 1 has its own versions of s1 and s2, as does iteration 2 and so on up to iteration n.

Cannot apply NEW clause to FORALL.

New Variables — Conditions

`NEW` variables cannot:

- be used outside of the loop before being redefined,
- be used with `FORALL`,
- be dummy arguments or pointers,
- be storage associated,
- have `SAVE` attribute.

New Variables Example 1

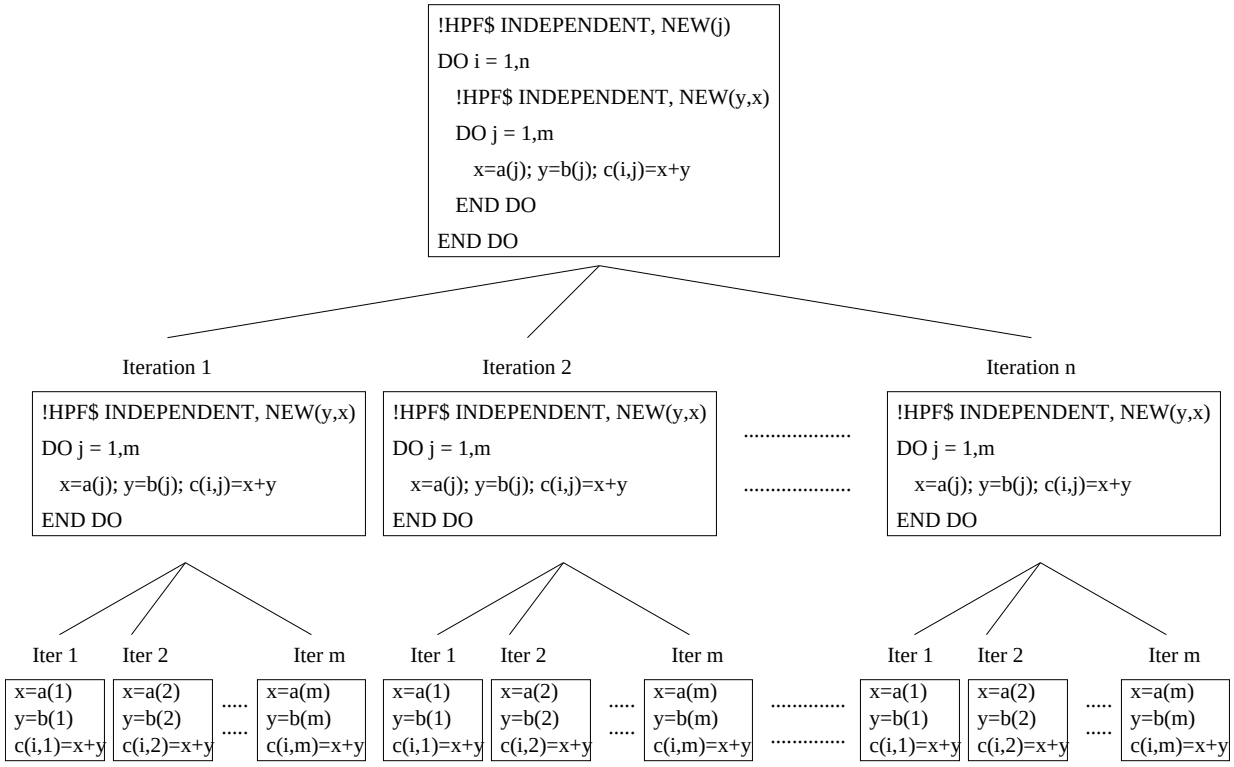
`INDEPENDENT` loops can be formed by creating copies of `x` and `y` for each inner iteration.

Copies of `j` can be made for further independence.

```
!HPF$ INDEPENDENT, NEW (j)
DO i = 1, n
 !HPF$ INDEPENDENT, NEW (x,y)
 DO j = 1, m
  x = A(j)
  y = B(j)
  C(i,j) = x+y
 END DO
END DO
```

After the loop `x`, `y`, `i` and `j` will have an undetermined value so **cannot** be used before being assigned a new value. (In regular Fortran they could be.)

Visualisation of New Variables Example 1



New Variables Example 2

Variable list specifies temporaries to use in `INDEPENDENT` loops, for example,

```
!HPF$ INDEPENDENT, NEW (i2)
DO i1 = 1, n1
 !HPF$ INDEPENDENT, NEW (i3)
 DO i2 = 1, n2
  !HPF$ INDEPENDENT, NEW (i4)
  DO i3 = 1, n3
   DO i4 = 1, n4
    a(i1,i2,i3) = a(i1,i2,i3) &
      + b(i1,i2,i4)*c(i2,i3,i4)
   END DO
  END DO
 END DO
END DO
```

Inner loop not `INDEPENDENT` as `a(i1,i2,i3)` is assigned to repeatedly.

Input and Output

HPF currently has **no** provision for parallel I/O. In general, one processor performs all I/O so `PRINT` and `READ` statements are very expensive.

Upon encountering `PRINT*, A(:,:)`

- each processor must send its part of `A` to the I/O processor
- the I/O processor must service messages from every processor in turn
- rebuild the array
- and print out

this could be a lengthy process!

Lecture 9:

Procedures

HPF Procedure Interfaces

Cannot pass distribution information as actual argument, how to communicate information?

There are 3 different methods in Full HPF:

- Prescriptive — "align the data as follows".
- Descriptive — "the data is already aligned as follows". Should give an `INTERFACE`.
- Transcriptive (`INHERIT` (not covered)) — "inherit the distribution from the dummy arguments". This is *very* inefficient and should be avoided.

Mapping and Procedures

In a procedure one can specify:

- `PROCESSORS`,
- `TEMPLATE`,
- alignment,
- distribution.

Use assumed-shape arrays. Interfaces should also contain mapping information.

Must avoid non-essential remapping.

Prescriptive Distribution

Can prescribe alignment and distribution,

- may cause remapping,
- mapping is restored on procedure exit.

For example,

```
    SUBROUTINE Subby(A,B,RES)
     IMPLICIT NONE
     REAL, DIMENSION(:,:), INTENT(IN)  :: A, B
     REAL, DIMENSION(:,:), INTENT(OUT) :: RES
!HPF$ PROCESSORS, DIMENSION(2,2)       :: P
!HPF$ TEMPLATE, DIMENSION(4,6)         :: T
!HPF$ ALIGN (:,:) WITH T(:,:)          :: A, B, RES
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P  :: T
       ...
    END SUBROUTINE Subby
```

Inside `Subby` the data will be mapped as specified.

Descriptive Distribution

Assert that alignment and distribution is as specified, minimises communications. For example,

```
    SUBROUTINE Subby(A,B,RES)
     IMPLICIT NONE
     REAL, DIMENSION(:,:), INTENT(IN)    :: A, B
     REAL, DIMENSION(:,:), INTENT(OUT)   :: RES
!HPF$ PROCESSORS, DIMENSION(2,2)         :: P
!HPF$ TEMPLATE, DIMENSION(4,6)           :: T
!HPF$ DISTRIBUTE *(BLOCK,BLOCK) ONTO *P  :: T
!HPF$ ALIGN (:,:) WITH *T(:,:)           :: A, B, RES
       ...
    END SUBROUTINE Subby
```

Asserts that `A`, `B` and `RES` are aligned and distributed as shown.

An `INTERFACE` should be given.

Examples of Dummy Distributions

Consider,

- `DISTRIBUTE (CYCLIC) ONTO P :: A` — distribute `A` cyclically onto `P`.
- `DISTRIBUTE *(CYCLIC) ONTO P :: A` — `A` already has cyclic distribution but may not be distributed over `P`.
- `DISTRIBUTE (CYCLIC) ONTO *P :: A` — `A` is distributed over `P` but may not have `CYCLIC` distribution.
- `DISTRIBUTE *(CYCLIC) ONTO *P :: A` — `A` already has cyclic distribution over `P`.

Consequences

Descriptive and prescriptive distributions have limited capabilities,

- describing mapping can be difficult,
- cannot inherit distributions,
- inflexible approach.

Transcriptive distributions are easier to use, but

- compiling inherited mappings is very complex,
- will be less efficient,
- but simple for user.

Templates and Modules

Modules are now supported by most compilers. Should make TEMPLATEs, PROCESSORS *and* DISTRIBUTE statements global:

```
 MODULE Global_Mapping_Info
  !HPF$ PROCESSORS, DIMENSION(2,2)      :: P
  !HPF$ TEMPLATE, DIMENSION(4,6)        :: T
  !HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P :: T
 END MODULE Global_Mapping_Info
```

Makes things easier,

```
    SUBROUTINE Subby(A,B,RES)
     USE Global_Mapping_Info
     IMPLICIT NONE
     REAL, DIMENSION(:,:), INTENT(IN) :: A, B
     REAL, DIMENSION(:,:), INTENT(OUT) :: RES
!HPF$ ALIGN WITH *T :: A, B, RES
        ...
    END SUBROUTINE Subby
```

Note: not for PURE procedures.

Problems with Modules

HPF objects cannot appear in `ONLY` or renames lists in a `USE` statement, for example,

```
SUBROUTINE Subby(A,B,RES)
 USE Global_Mapping_Info, ONLY:ProcArr => P
```

is invalid Fortran 90.

Interfaces

Good practise to declare explicit interfaces for procedures containing mapped dummies.

```
    INTERFACE
     SUBROUTINE Soobie(A,B,Res)
      USE Global_Mapping_Info
      REAL, DIMENSION(:,:), INTENT(IN) :: A, B
      REAL, DIMENSION(:,:), INTENT(OUT) :: Res
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P :: A, B, Res
     END SUBROUTINE Soobie
    END INTERFACE
```

Should always use assumed-shape arrays so interfaces will be mandatory anyway.

Aligning to Dummy Arguments

Can align locals to dummies,

```
    SUBROUTINE Soobie(A,B,Res)
     USE Global_Mapping_Info
     IMPLICIT NONE
     REAL, DIMENSION(:,:), INTENT(IN)          :: A, B
     REAL, DIMENSION(:,:), INTENT(OUT)         :: Res
     REAL, DIMENSION(SIZE(A,1),SIZE(A,2))      :: C
     REAL, DIMENSION(SIZE(A,1)/2,SIZE(A,2)/2) :: D
!HPF$ PROCESSORS, DIMENSION(2,2)               :: P
!HPF$ ALIGN (:,:) WITH A (:,:)                 :: C
!HPF$ ALIGN (:,J) WITH A(J*2-1,::2)            :: D
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P :: A, B, Res
      ...
    END SUBROUTINE Soobie
```

Could also use descriptive distributions, more efficient, less flexible.

Mapping Function Results

Clearly, must be able to map array-valued FUNCTION results,

```
MODULE Block_Dist_1D_Template_Onto_P
  !HPF$ PROCESSORS, DIMENSION(2)  :: P
  !HPF$ TEMPLATE, DIMENSION(4)    :: T
  !HPF$ DISTRIBUTE (BLOCK) ONTO P :: T
END MODULE Block_Dist_1D_Template_Onto_p

FUNCTION ArF(A,B)
 USE Block_Dist_1D_Template_Onto_P
 IMPLICIT NONE
  REAL, INTENT(IN) :: A(:), B(:)
  REAL, DIMENSION(SIZE(A)) :: ArF
  !HPF$ ALIGN A(:) WITH *T(:)
  !HPF$ ALIGN B(:) WITH *T(:)
  !HPF$ ALIGN ArF(:) WITH T(:)
    ...
END FUNCTION ArF
```

An explicit interface should *always* be given containing all mapping information relating to dummy arguments and the function result.

Argument Remapping

Should not remap across a procedure boundary unless *absolutely essential*. The implied communications can be very time consuming. Consider,

```
      INTEGER, DIMENSION(512,512) :: ia, ib
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: ia, ib
       DO icnt = 1, 10
        CALL ReMapSub(ia,ib)
       END DO
      END
      SUBROUTINE ReMapSub(iarg1, iarg2)
      INTEGER, DIMENSION(512,512):: iarg1, iarg2
!HPF directive goes here
       iarg2 = 2*iarg1
      END SUBROUTINE ReMapSub
```

With NA Software Compiler, if `iarg1` and `iarg2` are distributed as,

- `(BLOCK,BLOCK)` — execution time is 0.25,
- `(CYCLIC,CYCLIC)` — execution time is 25.00s,

Explicit Intent

A general procedure call can generate *two* remappings per argument:

- on procedure entry,
- on procedure exit.

If remapping is essential then give the `INTENT` of the arguments:

```
INTEGER, DIMENSION(512,512), INTENT(IN)  :: iarg1
INTEGER, DIMENSION(512,512), INTENT(OUT) :: iarg2
```

Now each dummy would only be remapped once. NA Software execution time is now 14.7s compared to 25.00s without the `INTENT`.

Motto: Always specify `INTENT`.

Passing Array Sections

Without using `INHERIT`, this is very complex. Consider

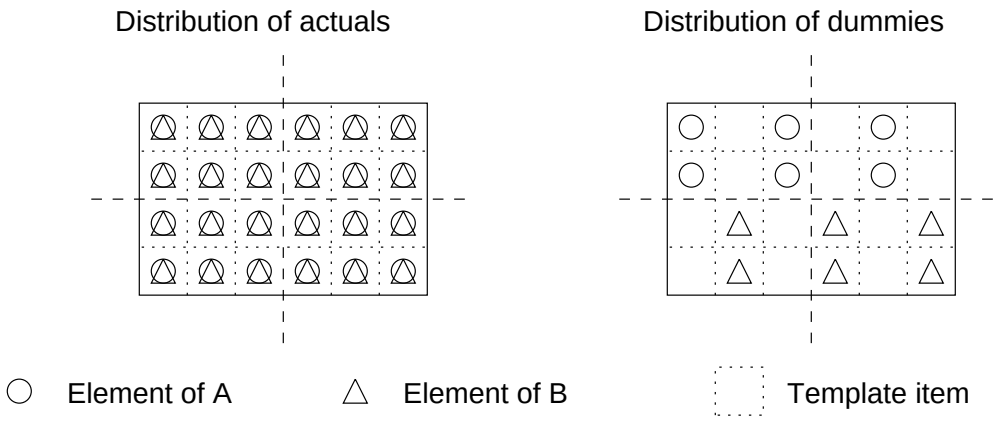
```
     REAL, DIMENSION(4,6) :: A, B
     REAL, DIMENSION(2,3) :: Res
!HPF$ PROCESSORS, DIMENSION(2,2) :: P
!HPF$ ALIGN B(:,:)   WITH A(:,:)
!HPF$ ALIGN Res(:,:) WITH A(::2,::2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P :: A, B, Res
      ...
      CALL Zubbie(A(1:2,1::2),B(3:4,2::2),Res)
```

For descriptive distributions what should interface look like?

- cannot describe mapping using `DISTRIBUTE`,
- cannot simply describe relative alignment,
- must construct intermediate `TEMPLATE` in order to specify distribution.

Array Arguments Example 1

Using descriptive distribution we can reconstruct original layout,




```
      SUBROUTINE Zubbie(x,y,z)
!HPF$ TEMPLATE, DIMENSION(4,6)              :: T
!HPF$ PROCESSORS, DIMENSION(2,2)            :: P
      REAL, INTENT(INOUT), DIMENSION(:,:) :: x, y
      REAL, INTENT(INOUT), DIMENSION(:,:) :: z
!HPF$ ALIGN (:,:) WITH *T(:,::2)            :: x, z
!HPF$ ALIGN (:,:) WITH *T(3:,2::2)          :: y
!HPF$ DISTRIBUTE *(BLOCK,BLOCK) ONTO *P     :: T
      ...
```

asserts distribution of cited case but not for,

```
      CALL Zubbie(A(3:4,::2),B(1:2,2::2),Res)
```

Array Arguments Example 2

It is much simpler but less efficient to use prescriptive distributions.

```
     SUBROUTINE Zubbie(x,y,z)
!HPF$ TEMPLATE, DIMENSION(4,6)              :: T
!HPF$ PROCESSORS, DIMENSION(2,2)            :: P
      REAL, INTENT(INOUT), DIMENSION(:,:) :: x, y
      REAL, INTENT(INOUT), DIMENSION(:,:) :: z
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P       :: x, y, z
      ...
```

may generate,

- two remappings,
  - on entry,
  - on exit.
- (probably) communications due to non-specific alignment.

Collapsing Dimensions

If a dimension has a scalar index it is collapsed. Consider,

```
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: A, B
      ...
     CALL Xubbie(A(i,:),b(i,:))
      ...
     SUBROUTINE Xubbie(x,y)
      REAL, DIMENSION(:) :: x, y
      !HPF$ DISTRIBUTE (BLOCK) :: x, y
```

Will cause remapping. With 16 processors:

- actual arguments are distributed over 4 processors,
- on entry `x` and `y` will be redistributed over 16 processors,
- on exit `x` and `y` will be mapped in same way as actual argument,

Scalar Arguments

This slide demonstrates what happens if a single array item is used as an actual argument. Consider,

```
    REAL, DIMENSION(100,100)    :: A, B
    REAL                        :: z
!HPF$ DISTRIBUTE (BLOCK,BLOCK)  :: A, B
    INTERFACE
     SUBROUTINE Schmubbie(r,t,X)
      REAL, INTENT(OUT)         :: r
      REAL, INTENT(IN)          :: t
      REAL, INTENT(IN)          :: X(:,:)
!HPF$ DISTRIBUTE *(BLOCK,BLOCK) :: X
     END SUBROUTINE Schmubbie
    END INTERFACE
     ...
    CALL Schmubbie(A(1,1),z,B)
```

`r` will be replicated, `t` already is.

Processors Problem

HPF contains the following text:

"An HPF compiler is required to accept any `PROCESSORS` *declaration in which the product of the extents of each declared dimension is equal to the number of physical processors that would be returned by* `NUMBER_OF_PROCESSORS()`*."*

- gives handle on available resources,
- aids portability,
- all processor arrangements have same size,
- problems with procedure interfaces when passing array sections.

The standard contains a fudge:

"Other cases may be handled as well."

which gives a potential portability problem.

A Possible Solution

Problem occurs when descriptive mappings are used to reduce communications. Consider,

```
    REAL, DIMENSION(100,100) :: A, B
!HPF$ PROCESSORS, DIMENSION(10,10) :: P
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: A, B
    INTERFACE
     SUBROUTINE KerXubbie(x,y)
      REAL, DIMENSION(:) :: x, y
      !HPF$ PROCESSORS,DIMENSION(10) :: P ! non-HPF
      !HPF$ ALIGN y WITH *x
      !HPF$ DISTRIBUTE *(BLOCK) ONTO *P :: x
     END SUBROUTINE KerXubbie
    END INTERFACE
     ...
    CALL KerXubbie(A(i,:),b(i,:))
```

Asserts that `x` and `y` are co-mapped and distributed blockwise over 10 processor subset! Not portable but semantics work!

The Next Problem

In HPF, given,

```
!HPF$ PROCESSORS, DIMENSION(4) :: P1
!HPF$ PROCESSORS, DIMENSION(4) :: P2
```

`P1` and `P2` are same processor array, but consider,

```
     REAL, DIMENSION(100,100) :: A
!HPF$ PROCESSORS, DIMENSION(4,4) :: P
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P :: A
      ...
      CALL Grubbie(A(1,:),A(100,:))
       ...
      SUBROUTINE Grubbie(x,y)
       REAL, DIMENSION(:) :: x, y
!HPF$ PROCESSORS, DIMENSION(4) :: P1, P2
!HPF$ DISTRIBUTE *(BLOCK) ONTO *P1 :: x
!HPF$ DISTRIBUTE *(BLOCK) ONTO *P2 :: y
       ...
```

but `P1` and `P2` are *not* same processor array. Oh lordy!

Motto: Do not pass array sections

Lecture 10:

Extrinsics, HPF Library and HPF in the Future

Extrinsic Procedures

HPF can call procedures written in other languages or other parallel programming styles. These are called `EXTRINSIC` procedures.

An `INTERFACE` including mapping information *must* be given:

```
INTERFACE
 EXTRINSIC (C) SUBROUTINE Job(a)
  REAL, DIMENSION(:) :: a
  !HPF$ DISTRIBUTE a(BLOCK)
 END SUBROUTINE Job
END INTERFACE
```

this would correspond to a C `void` function with a single array argument.

It is up to the compiler to decide which languages are supported.

Extrinsic Procedure Example

Specifying the mapping information is *very* important. If it is absent then dummy arguments may be remapped and given the default mapping.

```
INTERFACE
 EXTRINSIC(F77_LOCAL) &
  SUBROUTINE Calc_u_like(My_P_No,Siz,Tot_Proc,a,b,c)
  INTEGER, DIMENSION(:), INTENT(IN)  :: B, C
  INTEGER, DIMENSION(:), INTENT(OUT) :: A
  INTEGER, DIMENSION(:), INTENT(IN)  :: My_P_No
  INTEGER, INTENT(IN)                :: Siz, Tot_Proc
  !HPF$ PROCESSORS, &
  !HPF$  DIMENSION(NUMBER_OF_PROCESSORS()) :: P
  !HPF$ DISTRIBUTE (BLOCK) ONTO P          :: A, B, C
  !HPF$ DISTRIBUTE (BLOCK) ONTO P          :: My_P_No
 END SUBROUTINE output ! EXTRINSIC(F77_LOCAL)
END INTERFACE
```

This is merely an example the keyword `F77_LOCAL` is not defined in HPF. It *may* be defined by the local compiler.

Extrinsic Example Continued

The `EXTRINSIC` is outside of HPF. In general, every `EXTRINSIC` must:

- have an explicit `INTERFACE` *including* mapping information,
- work out which, if any, array elements are local

The `INTERFACE` is expressed using HPF concepts of `INTENT`, distribution and assumed-shape arrays; the `EXTRINSIC` is not.

Extrinsic Example Continued

An F77_LOCAL EXTRINSIC:

```
 SUBROUTINE Calc_u_like(My_P_No,Siz,Tot_Proc,a,b,c)
  INTEGER A(*), B(*), C(*), My_P_No(1), Siz, Tot_Proc
C Find blocksize
  Blk_Siz    = NINT((DBLE(Siz)/DBLE(Tot_Proc))+0.5D0)
C How many elements have I got
  My_Blk_Siz = MIN(Blk_Siz,Siz-(My_P_No(1)-1)*Blk_Siz)
  My_Blk_Siz = MAX(My_Blk_Siz,0)
C Do the Calculation
  DO 100 i = 1,My_Blk_Siz
   a(i) = b(i) + c(i)
  END DO
 END
```

and in the calling program unit:

```
  REAL, DIMENSION(Siz) :: A, B = 0, C = 0
  INTEGER, DIMENSION(NUMBER_OF_PROCESSORS()) :: P_Nos
  !HPF$ PROCESSORS, &
  !HPF$  DIMENSION(NUMBER_OF_PROCESSORS()) :: P
  !HPF$ DISTRIBUTE (BLOCK) ONTO P :: A, B, C, P_Nos
    ... Interface from before goes here
  NOP   = NUMBER_OF_PROCESSORS()
  P_Nos = (/ (i, i=1,NOP) /)
  CALL Calc_u_like(P_Nos,SIZE(A),NOP,A,B,C)
 END
```

Extrinsic Example Continued

Matters become easier with a `F90_LOCAL EXTRINSIC`:

```
SUBROUTINE Calc_u_like(A,B,C)
 INTEGER, DIMENSION(:), INTENT(IN)  :: B, C
 INTEGER, DIMENSION(:), INTENT(OUT) :: A
  A = B+C
END
```

Can use assumed-shape arrays to avoid explicitly calculating block-sizes. Fortran 90 allows zero-sized sections which is also useful.

Rules for Extrinsic Procedures

Extrinsic procedures have a number of constraints placed on them so they behave in the same way as 'regular' HPF procedures. An `EXTRINSIC` must:

- fully terminate before control returns,
- not permanently change the mapping of an object,
- obey any `INTENT`,
- ensure that non-local replicated variables have a consistent value,
- not permanently modify the number of available processors.

Uses of Extrinsics

`EXTRINSIC`s may contain:

- a super-efficient message passing kernel,
- calls to library functions,
- an interface to a package,
- calls to 'trusty' old code,
- calls to a different language,
- code which requires no synchronisation.

Extrinsic instead of INDEPENDENT

The INDEPENDENT directive is currently not implemented by most compilers. Assuming the loop implies no communication an EXTRINSIC can be used to achieve the same functionality:

```
 !HPF$ DISTRIBUTE A(*,CYCLIC)
....
 !HPF$ INDEPENDENT, NEW(i)
 DO j = 1, n
  DO i = 1, m
   ! ... stuff missing
   A(i,j) =
   ! ... stuff missing
  END DO
 END DO
....
```

the loop can be replaced by a call to the EXTRINSIC Ext_Loop:

```
....
 CALL Ext_Loop(A,...)
....
```

The EXTRINSIC contains the loop with n and m modified.

Calls to the NAG Library

It is very easy to call the NAg Fortran 77 library from within an `F77_LOCAL` extrinsic. To use π from the NAg library:

```
      DOUBLE PRECISION FUNCTION Pi()
       DOUBLE PRECISION X01AAF, x
        Pi = X01AAF(x)
      END
```

and the calling program

```
PROGRAM Using_NAG_4_Pi
  !HPF$ PROCESSORS, DIMENSION(4)   :: P
  DOUBLE PRECISION, DIMENSION(100) :: A
  !HPF$ DISTRIBUTE (BLOCK) ONTO P  :: A
  INTERFACE
   EXTRINSIC(F77_LOCAL) DOUBLE PRECISION FUNCTION Pi()
   END FUNCTION Pi
  END INTERFACE
  A   = Pi()
END PROGRAM Using_NAG_4_Pi
```

All scalars (ie `Pi`) must be coherent.

New Intrinsics in HPF

HPF alters two classes of Fortran 90 intrinsics:

- system inquiry intrinsics: adds `NUMBER_OF_PROCESSORS` and `PROCESSORS_SHAPE`,
- computational intrinsics: adds `ILEN` and extends `MINLOC` and `MAXLOC`.

`ILEN` returns the number of bits needed to store an `INTEGER` values and is used to round an integer to the nearest power of 2.

HPF adds a `DIM=` specifier to the location intrinsics.

HPF Library Module

The HPF Library `MODULE` contains a number of functions:

- Mapping inquiry functions
- Array reduction functions
- Bit manipulation functions
- Array combining scatter functions
- Prefix and suffix functions
- Array sorting functions

Must include `USE HPF_LIBRARY` to attach module.

Mapping Inquiry Subroutines

There are three SUBROUTINEs in this class:

- HPF_ALIGNMENT,
- HPF_DISTRIBUTION,
- HPF_TEMPLATE,

All above procedures must be supplied with an object name and up to 7 optional arguments of INTENT(OUT) which return mapping information.

Example of Mapping Inquiry Procedures

For example,

```
REAL, DIMENSION(100,100)           :: A
CHARACTER(LEN=9), DIMENSION(2)     :: DISTS
INTEGER, DIMENSION(2)              :: BLK_SIZE, PSHAPE
INTEGER                            :: PRANK
!HPF$ PROCESSORS, DIMENSION(4)     :: P
!HPF$ DISTRIBUTE (BLOCK,*) ONTO P :: A

CALL HPF_DISTRIBUTION(A,AXIS_TYPE        = DISTS,   &
                        AXIS_INFO        = BLK_SIZE,&
                        PROCESSORS_RANK  = PRANK,   &
                        PROCESSORS_SHAPE = PSHAPE)
```

Here DISTS is equal to (/'BLOCK','COLLAPSED'/).

BLK_SIZE(1) is equal to 50. BLK_SIZE(2) is compiler dependent.

PRANK is 1 and PSHAPE(1) is 4. PSHAPE(2) has not been assigned a value.

The other procedures follow a similar pattern.

New HPF Reduction Functions

There are four `FUNCTION`s in this class:

- `IALL`, corresponds to `IAND` reduction,
- `IANY`, corresponds to `IOR` reduction,
- `IPARITY`, corresponds to `IEOR` reduction,
- `PARITY`, corresponds to `.NEQV.` reduction,

The first three procedures operate on the bit pattern.

Example of New Reduction Functions

All functions operate on arrays, for example, `IALL(A)` is the same as

```
...(IAND(IAND(IAND(A(1),A(2)),A(3)),A(4)),...)
```

`PARITY` is not bitwise and is used with `LOGICAL` valued expressions. `PARITY(A)` is

```
A(1).NEQV.A(2).NEQV.A(3).NEQV.A(4). ...
```

For example,

```
PARITY((/F,T,F,T,F/))
```

is `.FALSE.` whereas,

```
PARITY((/F,T,F,T,T/))
```

is `.TRUE.`.

Bit Manipulation Functions

There are three new functions in this class plus the new intrinsic `ILEN`:

- `LEADZ`— number of leading zeros
- `POPCNT`— number of `1` bits
- `POPPAR`— parity of integer

All functions are elemental but must take `INTEGER` arguments.

Array Combining Scatter Functions

Fortran 90 allows indirect addressing, however if,

```
INTEGER, DIMENSION(4) :: A = 1, B = 2
INTEGER, DIMENSION(3) :: W = (/1,2,2/)
```

then writing

```
A(W) = A(W) + 2*B(1:3)
```

is incorrect due to the multiple assignments to `A(2)`. We are able to use combining scatter functions:

```
A = SUM_SCATTER(2*B(1:3),A,W)
```

now `A` equals `(/5,9,1,1/)`. This performs

```
A(1) = A(1) + 2*B(1)
A(2) = A(2) + 2*B(2) + 2*B(3)
```

Array Combining Scatter Functions II

These functions allow combined assignments to vector subscripted arrays with repeated values. Consider,

```
A = PRODUCT_SCATTER(2*B(1:3),A,W)
```

`A` is now equal to `(/4,16,1,1/)`.

The following prefixes are allowed for scatter functions: `ALL`, `ANY`, `COUNT`, `IALL`, `IANY`, `IPARITY`, `MAXVAL`, `MINVAL`, `PARITY`, `PRODUCT` and `SUM`. Consider,

```
MINVAL_SCATTER((/10,-2,4,2/),(/1,1,1/),(/2,2,1,1/))
```

this gives the result `(/1,-2,1/)`.

Prefix and Suffix Functions

Both classes of functions are related

- prefix functions scan along an array. Each element of the result depends upon the preceding elements (in array element order). For example,

```
PRODUCT_PREFIX((/1,2,3,4/)) = (/1,2,6,24/)
PRODUCT_PREFIX((/1,4,7/), = ((/1, 24,  5040/),
               (/2,5,8/), =  (/2,120, 40320/),
               (/3,6,9/)) =  (/6,720,362880/))
```

- suffix functions do the same but scan *backwards*.

```
PRODUCT_SUFFIX((/1,2,3,4/)) = (/24,24,12,4/)
PRODUCT_SUFFIX((/1,4,7/), = ((/362880,60480,504/),
               (/2,5,8/), =  (/362880,15120, 72/),
               (/3,6,9/)) =  (/181440, 3024,  9/))
```

Prefix and Suffix Functions II

There are a number of different combiners which make up the prefix and suffix functions. These include the Fortran 90 reductions:

- `SUM` and `PRODUCT`: for example,

  `SUM_PREFIX`, `PRODUCT_SUFFIX`,

- `MAXVAL` and `MINVAL`: for example,

  `MAXVAL_SUFFIX`, `MINVAL_PREFIX`,

- `ALL`, `ANY` and `COUNT`: for example,

  `ALL_SUFFIX`, `ANY_PREFIX`,

plus HPF defined intrinsics: `IALL`, `IANY`, `IPARITY` and `PARITY`.

Prefix and Suffix Functions III

The functions all take (more or less) the same arguments, for example:

```
MINVAL_PREFIX(ARRAY[,DIM][,MASK][,SEGMENT][,EXCLUSIVE])
```

- a `MASK` argument works as in Fortran 90,
- `COPY_...` doesn't have `MASK` and `EXCLUSIVE`,
- `ALL_...`, `ANY_...`, `COUNT_...` and `PARITY_...` do not have `MASK` as `ARRAY` is `LOGICAL`.

`MASK` and `SEGMENT` are `LOGICAL`, `MASK` conforms to `ARRAY`, `SEGMENT` has same shape as `ARRAY`.

Example of the `MASK` argument,

```
PRODUCT_PREFIX((/1,2,3,4/), MASK=(/T,F,T,F/)) = (/1,1,3,3/)
```

SEGMENT and EXCLUSIVE

SEGMENT: apply the function to sections, for example,

```
S       =  (/T,T,T, F,F, T,T, F, T,T/)
!            -----  ---  ---  -  ---
SUM_PREFIX((/1,2,3, 4,5, 6,1, 2, 3,4/),SEGMENT=S) =
           (/1,3,6, 4,9, 6,7, 2, 3,7/)
```

EXCLUSIVE: scalar LOGICAL. If .FALSE. (default) then each element takes part in operation for its position, otherwise it does not and the first scanned element has identity value.

```
PRODUCT_PREFIX((/1,2,3,4/), EXCLUSIVE=.TRUE.) =
               (/1,1,2,6/)
SUM_PREFIX((/1,2,3,4/), EXCLUSIVE=.TRUE.) =
           (/0,1,3,6/)
```

Array Sorting Functions

There are two functions in this class:

- `GRADE_UP` — smallest first,
- `GRADE_DOWN` — largest first,

These can be used to sort multi-dimensional arrays as a whole (in array element order) or along (an optionally) specified dimension.

Each function returns a permutation of the array indices. Duplicate values remain in the original (array element) order.

Example of Array Sorting Functions

Given,

```
A = (/2,3,7,4,9,1,5,5,0,5,5/)
```

then `GRADE_DOWN(A)` is the 2D (1×11) array:

```
(/5,3,7,8,10,11,4,2,1,6,9/)
```

and

`GRADE_UP(A,DIM=1)` is the 1D array:

```
(/9,6,1,2,4,7,8,10,11,3,5/)
```

Note how the multiple values of `5` are sorted.

The result when *not* using `DIM=` has shape

```
(/ SIZE(SHAPE(A)),PRODUCT(SHAPE(A))/)
```

Otherwise the shape is the same as `A`.

Further Example of Array Sorting Functions

If `A` is the 2D array:

```
1 9 2
4 5 2
1 2 4
```

Then `GRADE_DOWN(A)` is (the coordinates)

```
1 2 2 3 3 1 2 1 3
2 2 1 3 2 3 3 1 1
```

and `GRADE_DOWN(A,DIM=1)` is

```
2 1 3
1 2 1
3 3 2
```

Storage and Sequence Association

The distribution of sequence and storage associated entities is very complex and best avoided. It is not part of HPF 2.0 (the new standard),

- sequence association does not work on distributed memory systems,
- storage association and retyping of memory does not mix well with distributed objects.

Things to avoid,

- distributing arrays in `COMMON`,
- distributing `EQUIVALENCE`d arrays,
- assumed-size arrays.

Dual Fortran 90 and HPF Codes

The are differences between Fortran 90 and HPF. To maintain dual codes:

- don't pass array sections,
- don't use pointers,
- get `HPF_LIBRARY` module,
- don't use storage and sequence association,
- use `cpp` to mask out `FORALL`, `EXTRINSIC`, etc.

As soon as Fortran 95 compilers arrive many problems will disappear.

Full HPF

Full HPF supports,

- all of Fortran 90,
- dynamic mappings, `REALIGN`, `REDISTRIBUTE`,
- inherited distributions,
- distributed derived types (not well defined),

Much care must be taken with pointers!

Performance of HPF Systems

HPF codes with 'regular computations', eg EP, fare well; those with 'irregular' array accesses, eg FFT1, do not.

EP (NAS benchmark) on SPARCCenter 1000 (2 $\times$ sun4d processors):

Source	Compiler	Exec Time (s)
f90	EPC	85.0
	Sun (‖) (1 proc)	111.0
	Sun (‖) (2 proc)	111.0
hpf	PGI (‖) (1 proc)	127.5
	PGI (‖) (2 proc)	67.6

Fortran 90 on SPARCCenter 2000 takes 115s; HPF on 8 processors takes 12s.

FFT1 (ParkBench) on 1 IPX with `epcf90` takes 0.3s; on 8 IPXs with `pghpf` takes 33s!

HPF 2

Actual language additions:

- change to Fortran 95,
- `REDUCTION` clause,
- `SORT_UP` and `SORT_DOWN` library routines.

Approved extensions,

- `GEN_BLOCK` and `INDIRECT` distributions, `SHADOW` regions, distributions `RANGE` directive.
- mapped `POINTER` objects and derived types
- processor subsets, `ON` directive, `TASK_REGIONS`,
- asynchronous I/O (`WAIT`),
- more extrinsics (C, Fortran 77, `HPF_CRAFT` and Fortran 77 local library.

HPF Kernel

'Guaranteed to execute fast and be portable'.

- no `INHERIT`.
- no pointers,
- no dynamic mapping,
- only certain types of alignment, not strided.