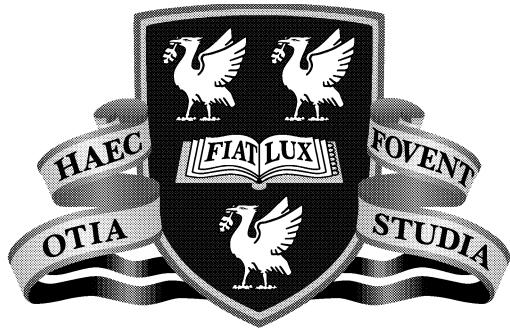


THE UNIVERSITY *of* LIVERPOOL

HPF Programming 5 Day Course Slides

Dr. A C Marshall (funded by JISC/NTI)
with acknowledgements to Steve Morgan, Dave Watson and Mike
Delves.

©University of Liverpool



THE UNIVERSITY *of* LIVERPOOL

HPF Programming (5 Day Course) Lecturers Guide

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

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

Lecture 1:
Overview, Objects and Expressions

2: 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;

Guide for slide: 2

The course:

- assumes familiarity with language such as high level languages such as Basic, ADA, C, Pascal, Algol, FORTRAN 77, APL.
- highlights new Fortran 90 features which make language more robust; for example, of those mentioned on the slide, many are object-based facilities which make it more difficult for the user to make mistakes or do daft things. Fortran 90 has traces of ADA in the sense that there are many many restrictions placed in the standard document meaning that mistakes, which in other languages (e.g., C and FORTRAN 77) would be syntactically correct but cause an odd action to be taken, are ruled out at compile time. As an example, if explicit interfaces are used then its is not possible to associate dummy and actual arguments of different types — this was possible in FORTRAN 77 and was sometimes used in anger but generally indicated an error.
- see “Programming in Fortran 90” by Morgan and Schonfelder (Alfred Waller Ltd, 1993, ISBN 1-872474-06-3) too. (Recommended text.)

3: 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.

Guide for slide: 3

- Fortran stood for *IBM Mathematical FORmula TRANslatiOn System* but this has been abbreviated to *FORmula TRANslatiOn*;
- Fortran was and has always been intended for scientific applications;
- Fortran is the oldest established high level language;
- the first compiler became available in 1957 and was written by IBM. More companies wrote compilers in the 60's but this lead to a number of dialects (by 1963 there were 40 different compilers), so there was a need to standardise the language;
- FORTRAN 77 finally standardised in 1972 (!) (—it is quite common for the Fortran version number to be out of step with the standardisation year).
- owing to standardisation, programs could be moved from platform to platform with little or sometimes no rewriting. This was one of the main reasons why the use of Fortran grew and grew.
- as compiler technology and user demands grew a new standard was required — this was completed in the late '70's and was known as FORTRAN 77 (ANSI X3.9-1978);
- FORTRAN 77 adopted as international standard by ISO in 1980;
- FORTRAN 77 is currently the most widely used “version” of Fortran. Compilers, which usually support a small number of extensions have, over the years, become *very* efficient. The technology which has been developed whilst implementing these compilers is not to be wasted and can still be applied to *all* Fortran 90 programs;
- almost as soon as it became standardised, FORTRAN 77 was outmoded compared to C, Algol, Pascal, APL, ADA etc. Many desirable features were not available, for example, in FORTRAN 77 it is very difficult to represent data structures succinctly and the lack of any dynamic storage meant that all arrays had to have a fixed size which could not be exceeded. It was clear from a very early stage that a new more modern language needed to be developed. Work began in early 80's on a language known as 'Fortran 8x'. ('x' was expected to be 8 in keeping with the previous names for Fortran.) The work took 12 years partly because of the desire to keep FORTRAN 77 a strict subset and also to ensure that efficiency (one of the bonus's of a simple language) was not compromised. Languages such as Pascal, ADA and Algol are a treat to use but cannot match FORTRAN 77 (or Fortran 90) for efficiency.
- as Fortran 90 standard is very large and complex there are a small number of ambiguities / conflicts / grey areas. These anomalies generally only come to light when compilers are developed. Since standardisation many compilers have been under development and a number of conflicts / grey areas / missing / desirable features have been identified. A new draft of Fortran, Fortran 95, will rectify the faults and extend the language in the appropriate areas;
- in Fortran 90 some features of FORTRAN 77 have been replaced by better, safer and more efficient features, some of these are to be removed from Fortran 95 although it is expected that many compilers will still support them as ‘extensions’. Tools exist to effect automatic removal of such obsolescent features, examples of these are (Pacific Sierra Research's) VAST90 and (NA Software's) LOFT90. These tools also perform differing amounts of vectorisation (transliteration of serial structures to equivalent parallel structures.)

- Fortran 90 was developed jointly between the ANSI X3 technical subcommittee X3J3 and the ISO/IEC JTC1/SC22/WG5 committee.
- in the last few years the Fortran 90 based de-facto language standard, known as High Performance Fortran (HPF), has been developed. This language contains the whole of Fortran 90 and also includes other desirable extensions. Fortran 95 will include all of the new syntactic features from HPF.
- HPF is intended for programming distributed memory machines and introduced “directives” (Fortran 90 structured comments) to give hints on how to distribute data (arrays) amongst grids of (non-homogeneous) processors. The idea is to relieve the programmer of the burden of writing explicit message-passing code; the compilation system does this instead. HPF also introduced a small number of executable statements (parallel assignments, side effect free procedures (PURE procedures)) which have been adopted by Fortran 95.
- HPF (apart from some features, for example, storage and sequence association) will be a superset of Fortran 95.

4: 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.

Guide for slide: 4

FORTRAN 77 is outmoded — many other languages have been developed which allow greater expressiveness and ease of programming.

1. FORTRAN 77 awkward ‘punched card’ source format: each line corresponds to 1 punched card with 72 columns thus each line can only be 72 characters long, this causes problems because text in cols 73 onwards is simply ignored (line numbers used to be placed in these columns). In this day and age this source format is totally unnecessary.

Other restrictions of FORTRAN 77:

- the first 5 columns are reserved for line numbers;
 - the 6th column can only be used to indicate a continuation line;
 - comments can only be initiated by marking the first column with a special character;
 - only upper case letters are allowed anywhere in the program;
 - variable names can only be 6 characters long meaning that mnemonic and cryptic names must be used all the time — maintenance unfriendly!
 - no in-line comments are allowed, comments must be on a line of their own;
2. Fortran is supposed to be a performance language — today High Performance Computing is implemented on Parallel machines — FORTRAN 77 has no in-built way of expressing parallelism. In the past, calls to specially written vector subroutines have been used or reliance has been made on the compiler to vectorise (parallelise) the sequential (serial) code. It is much more efficient to give the user control of parallelism and this has been done, to a certain extent, by the introduction of parallel ‘array syntax’.
 3. FORTRAN 77 only allows static storage for example, this means temporary short-lived arrays cannot be created on-the-fly nor can pointers be used (Pointers are useful for implementing intuitive data structures.) All FORTRAN 77 programs must declare arrays ‘big enough’ for any future problem size which is an awkward and very unattractive restriction absent from virtually all of the current popular high-level languages.
 4. problems arise with precision when porting FORTRAN 77 code from one machine to another. Many FORTRAN 77 systems implement their own extensions to give greater precision, this means that the code becomes non-portable. Fortran 90 has taken ideas for the various FORTRAN 77 extensions and improved them so that the new language is much more portable than before.
 5. in FORTRAN 77 intuitive (user-defined) data types are not available as they are in ADA, Algol, C, Pascal etc.. Their presence would make programming more robust and simpler. In FORTRAN 77 there is no way of defining compound objects.
 6. FORTRAN 77 lacked recursion which is a very useful and succinct mathematical technique. In the past this had to be simulated using a user defined stack and access routines to manipulate stack entries. Recursion is a fairly simple and a code efficient concept and was, to all intents and purposes, unavailable.
 7. in FORTRAN 77 global data is only accessible via the notoriously open-to-abuse COMMON block. The rules which applied to COMMON blocks are very lax and the user could inadvertently do quite horrendous things! Fortran 90 presents a new method of obtaining global data.
In FORTRAN 77 a user could alias arrays using the EQUIVALENCE statement which is equally as open to abuse as COMMON blocks. A great deal of errors stem from mistakes in these two areas which were generally regarded as unsafe but there was no real alternative in FORTRAN 77.

5: 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 (KINDs);
5. derived data types and operators;
6. recursion;
7. MODULEs
 - procedure interfaces;
 - enhanced control structures;
 - user defined generic procedures;
 - enhanced I/O.

Guide for slide: 5

This slide ‘answers’ the previous one,

1. Fortran 90 defined new code format — much more like every other language;
 - upto 132 columns per line;
 - more than one statement per line;
 - in-line comments allowed — makes it easier to annotate code;
 - upper and lower case letters allowed — makes code more readable;
 - it is virtually impossible to misplace a character now;
 - longer and more descriptive object names (upto 31 chars);
 - names can be punctuated by underscores making them more readable.
2. parallelism can be expressed using whole array operations including extensive slicing and sectioning facilities, and arithmetic on arrays and array sections; Operations in an array assignment statement are conceptually performed in parallel. This is adapted from APL.
The masked (parallel) assignment (**WHERE**) statement and construct has also been introduced.
3. many parallel intrinsic functions have been introduced including reduction operations such as **SUM** (add all elements in an array and return one value — the sum) and **MAXVAL** (scan all elements in an array and return one value — the biggest). This concept comes again from APL..
The provision of an enhanced set of intrinsics has increased efficiency of the language. Common operations have been identified and new intrinsics such as bit manipulation functions, precision and numeric representation inquiry functions, reshaping and retying functions and array construction functions, have been specified.
Many of the FORTRAN 77 intrinsics have been extended to accept whole array arguments.
4. the introduction of dynamic (heap) storage means that pointers can be implemented. Pointers can be used for aliasing arrays and creating dynamic data structures such as linked lists and trees. Temporary arrays can also be created on-the-fly.
5. the **KIND** facility has been introduced. This allows user to specify desired precision in a portable way. The precision of a particular type in the program can be changed by altering the value of one constant. Appropriate intrinsic inquiry functions are defined to query the precision of given objects.
6. user-defined types which are constructed from existing types can now be defined, for example, a type can be defined to represent a 3D coordinate which has three components (x, y, z) or a different type can be used to represent personal details: name, age, sex address, phone number etc. Defining objects in this way is more intuitive and makes programming easier and less error prone.

It is possible to define operators for derived and intrinsic data types which mean that so-called ‘semantic extension’ can be performed, For example, an arbitrary length integer can be implemented using a linked list structure (one digit per cell) and then all the intrinsic operators defined (overloaded) so that this type can be treated in the same fashion as all other types. The language thus appears to have been extended. All intrinsic operators +, -, *, / and ** and assignment, =, can be overloaded (defined for new types).

An ancillary standard, ‘The varying strings module’, has been defined and is implemented in a module. When using the module, the user can simply declare an object of the appropriate

type (`VARYING_STRING`) which can be manipulated in exactly the same way as the intrinsic character type. All intrinsic operations and procedures are included in the module and are therefore available for all objects of this type. All the implementation details are hidden from the user.

7. explicit recursion is now available. For reasons of efficiency the user must declare a procedure to be recursive but it can then be used to call itself.
8. facility packaging has been introduced — `MODULEs` replace many features of FORTRAN 77. They can be used for global definitions (of types, objects, operators and procedures), and can be used to provide functionality whose internal details are hidden from the user (data hiding).

`MODULEs` also provide object oriented facilities for Fortran 90, for example, it is possible to use a module to implement the abovementioned long integer data type — all required facilities (for example, definitions, objects, operators, overloaded intrinsics and manipulation procedures) may be packaged together in a `MODULE`, the user simply has to `USE` this module to have all the features instantly available. The module can now be used like a library unit.

Useful libraries can be written and placed in a module.

`BLOCK DATA` subprograms are now redundant since a `MODULE` can be used for the same purpose.

- procedure interfaces can be declared. The interface specifies to the calling program unit, all pertinent details about the procedure and its arguments, this means:
 - ◊ programs can be separately compiled — more efficient development;
 - ◊ array bounds do not have to be passed as arguments;
 - ◊ array shape can be inherited from the actual argument;
 - ◊ better type checking across procedure boundaries;
 - ◊ efficient code can be generated.
- new control constructs have been introduced, for example,
 - ◊ `DO ... ENDDO` (not part of FORTRAN 77) this will reduce use of numeric labels in the program;
 - ◊ `DO ... WHILE` loop;
 - ◊ the `EXIT` command for gracefully exiting a loop;
 - ◊ the `CYCLE` command for abandoning the current iteration of a loop and commencing the next one;
 - ◊ named control constructs — useful for code readability;
 - ◊ `SELECT CASE` control block. Present in many other languages and is more succinct, elegant and efficient than an `IF ELSEIF ... ELSEIF ENDIF` block.
- non-advancing I/O has also been introduced.

6: 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;

Guide for slide: 6

- All of FORTRAN 77 is included in Fortran 90 but as the language evolves certain features become redundant.
- FORTRAN 77 provided better ways to program certain actions so these actions are to be removed from Fortran.
- They were kept in FORTRAN 77 owing to the existence of dusty deck codes
- It is as well to let programmers know what not to use so they don't!!
- automatic translation of these features is possible — use VAST90 or loft90.

7: 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;
- ASSIGNED GOTO statements;
- ASSIGNED 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;

Guide for slide: 7

As these features will be removed from Fortran they should not be used in programs:

- the arithmetic IF statement — use IF statement instead;

It is a three way branch statement of the form,

IF(< expression >) < label1 >, < label2 >, < label3 >

here *< expression >* is any expression producing a result of type integer, real or double precision, and the three labels are statement labels of executable statements. If the value of the expression is negative, execution transfers to the statement labelled *< label1 >*; if the value is zero, transfer is to the statement labelled *< label2 >*, and if it is positive execution transfers to *< label3 >*. The same label can be repeated.

This relic of the original Fortran has been redundant since the early 1960s when the logical IF and computed GOTO were introduced.

- **ASSIGN** statement;

used to assign a statement label to an INTEGER variable (the label cannot be interpreted as an integer). Used in a GOTO or FORMAT statement.

ASSIGN < label > TO < integer-variable >

- **ASSIGNED GOTO** statements;

Historically this was used to simulate a procedure call before Fortran had such procedures.

- **ASSIGNED FORMAT** statements;

The ASSIGN statement can be used to assign a label to an integer which is subsequently referred to in an input/output statement

The same functionality can be obtained through use of character variables to hold FORMAT specifications.

- Hollerith format strings were used to represent strings in a format statement like this

```
      WRITE(*,100)
100   FORMAT(16HTITLE OF PROGRAM)
```

use CHARACTER strings with single or double quotes now,

```
      WRITE(*,100)
100   FORMAT('TITLE OF PROGRAM')
```

- the PAUSE statement - use a PRINT statement and a READ statement which waits for input; PAUSE was used to suspend execution until a key was pressed on the keyboard:

PAUSE < stop code >

The *< stop code >* is written to the standard output at the PAUSE statement.

- REAL and DOUBLE PRECISION DO-loop control expressions and index variables - use integral variables and construct real variable within the loop;

A loop with real valued DO-loop control expressions could easily loop a different number of times on different machines - a loop with control expression 1.0, 2.0, 1.0 may loop 1 or 2 times - 1.0 + 1.0 could equal, say, 1.99, or 2.01. The first evaluation would execute 2 times whereas the second would only give 1 execution.

- shared DO-loop termination - use separate END DO statements;

A number of DO loops can currently be terminated on the same (possibly executable) statement - this causes all sorts of confusion, when programs are changed so that the loops do not logically end on a single statement any more.

```
DO 100 K=1,N
DO 100 J=1,N
DO 100 I=1,N
...
100 A(I,J,K)=A(I,J,K)/2.0
```

Use END DO instead.

- alternate RETURN - use a return code and a GOTO statement or some equivalent control structure.

This allows a calling program unit to specify labels as arguments to a called procedure as shown. The called procedure can then return control to different points in the calling unit by specifying an integer parameter to the RETURN statement which corresponds to a set of labels specified in the argument list.

```
...
CALL SUB1(x,y,*98,*99)
...
98 CONTINUE
...
99 CONTINUE
...

SUBROUTINE SUB1(X,Y,*,*)
...
RETURN 1
...
RETURN 2
END
```

- branching to an END IF from outside its block — branch to the next statement instead or else add a CONTINUE statement.

8: 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;

Guide for slide: 8

- fixed source form layout - use free form;
mentioned before - free form is less error prone and more intuitive
- implicit declaration of variables - use **IMPLICIT NONE**;
mentioned before - saves undeclared variables slipping through
- **COMMON** blocks - use **MODULE**;
use the more safer **MODULE** program unit - see later.
- assumed size arrays - use assumed shape;
assumed shape more reliable - see later. Use **TRANSFER** function for changing the type of an object, use **RESHAPE** for changing the shape of an array.
- **EQUIVALENCE** statements;
this was used to rename and retype an area of storage — use **POINTER** variables for aliasing, use **TRANSFER** function for changing the type of an object
- **ENTRY** statements;
this was used to ‘jump in’ to a specified procedure at a specified point in the executable code. Can use internal procedures for this now.
- the computed **GOTO** statement — use **IF** statement instead;

9: 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;

Guide for slide: 9

FORTRAN 77 had virtually no Object Oriented features, Fortran 90 adds much, but by no means all, the required functionality. As usual there is a trade off with efficiency, one of the ultimate goals of Fortran 90 is that the code *must* be efficient.

The object oriented (or object based) facilities available are:

data abstraction

- ◊ user derived types provide a certain degree of abstraction,
- ◊ no enumerated types,
- ◊ a lack of parameterised derived types. The idea would be to define a skeleton type which could be supplied with KIND values in such a way that the individual components are declared with these KIND. As currently defined, derived types can have kind values for the components but they are fixed, for example the following two types are totally separate:

```
TYPE T1
    INTEGER(KIND=1) :: sun_zoom
    REAL(KIND=1)    :: spock
END TYPE T1
TYPE T2
    INTEGER(KIND=2) :: sun_zoom
    REAL(KIND=2)    :: spock
END TYPE T2
```

If the kind selection could be deferred until object declaration then they could be considered to be parameterised.

- ◊ Subtypes provide data abstraction. Subtypes are, as their name suggests, a subclass of a parent type. A common example may be a positive integer type which has exactly the same properties as an intrinsic INTEGER type but with a restricted range. Subtypes are expensive to implement but provide range checking security.

data hiding

- ◊ advanced data hiding, can tag any entity in a module with a visibility attribute,
- ◊ PRIVATE and PUBLIC statements are available but are not necessarily sufficient.

encapsulation

- ◊ very closely aligned with data hiding
- ◊ available through MODULEs / MODULE PROCEDUREs and USE statements and backed up by the renames and ONLY facilities,

inheritance and extensibility

- ◊ Fortran 90 supports supertypes — user-defined types can include other defined types,
- ◊ new MODULEs may inherit functionality from previously written MODULEs by using them, however, entities that are PRIVATE in the use-associated module cannot be seen in the new module. There really needs to be a third type of accessibility which allows PRIVATE objects to be accessible outside the module under certain circumstances this would allow for much greater extensibility.

- ◊ Fortran 90 does not have subtypes so functional and object inheritance cannot be transmitted this way, this is the main drawback towards OOness, (mind you ADA has subtypes but is still not OO language),
- ◊ being able to define subtypes which inherit functions and the like which are valid for the parent type is a very important aspect of OOness - Fortran 90 does not have this.

□ *polymorphism*

- ◊ the generic capabilities are very advanced, however, polymorphism must be programmed by the user, i.e., specific procedures must be user-written to support the generic interface,
- ◊ Fortran 90 does not support dynamic binding, i.e., the resolution of generic calls at run-time,
- ◊ cannot pass generic procedures as actual procedure arguments — a separate procedure must be exist for each different dummy procedure, (different result type, number of arguments etc).
- ◊ the lack of subtypes rules out many opportunities for polymorphism

□ *reusability*

- ◊ MODULEs can be used as libraries of types, object declarations and functionality and can be used in other modules.

There are, however, many shortfalls in its capabilities such as the lack of parameterised derived types

10: 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
```

Guide for slide: 10

The program highlights the following:

- free format source code — executable statements do not have to start in or after column 7 as they do in FORTRAN 77.
- **MODULE Triangle_Operations** — a program unit used to house procedures.
- **IMPLICIT NONE**
IMPLICIT NONE — makes declaration of variables compulsory throughout the module. It applies globally.
- **CONTAINS** — specifies that the rest of the MODULE contains procedure definitions.
- **FUNCTION Area(x,y,z)**
this declares a named function and the number and name of its arguments.
- **REAL :: Area ! function type**
REAL Area — **FUNCTION** procedures return a result in a ‘variable’ which has the same name as the function. The type of the function result must be declared in either the header or in the declarations.
The ! initiates a comment, everything after this character on the same line is ignored by the compiler.
- **REAL, INTENT(IN) :: x, y, z**
the type of the dummy arguments must always be declared, they should be the same type as the actual arguments.
INTENT — this says how the arguments are to be used.
 - ◊ **IN** means the arguments are used but not (re)defined;
 - ◊ **OUT** says they are defined before being used;
 - ◊ **INOUT** says that they are used and then redefined.
- Specifying the **INTENT** is not compulsory but is good practise as it can highlight errors.
- **REAL :: theta, height**
final **REAL** statement declares local variables for use in the **FUNCTION** — they cannot be accessed in the calling program,
- **theta = ACOS((x**2+y**2-z**2)/(2.0*x*y))**
assignment statement — assigns a value to **theta** and uses some mathematical operators:
 - ◊ * - multiplication,
 - ◊ ** - exponentiation;
 - ◊ / - division
 - ◊ + - addition,
 - ◊ - - subtraction

The brackets (parenthesis) are used to group calculations (as on a calculator) and also to obtain the argument to the intrinsic function reference ACOS.

Intrinsic functions are part of the Fortran 90 language and cover many areas, the simplest and most common are mathematical functions such as SIN and COS or MIN and MAX. Many are designed to act elementally on an array argument, in other words they will perform the same function to every element of an array at the same time.

- `height = x*SIN(theta); Area = 0.5*y*height`

Two statements on one line. The ; indicates that a new statement follows on the same line. Normally there is only one per line.

The function variable `Area` must be assigned to or else the function will not return a result.

- the MODULE program unit terminates with a END MODULE statement.

- PROGRAM Triangle

PROGRAM statement — don't strictly need one but it is good practice. One per program.

- USE Triangle_Operations — attaches the MODULE to the program and allows the FUNCTION to be used by the program.

- IMPLICIT NONE

IMPLICIT NONE — makes declaration of variables compulsory;

- REAL :: a, b, c, Area

REAL — declaration of real valued objects. `a`, `b` and `c` are variables, `Area` is a function procedure. This function must be given a type because its name contains a value.

- PRINT *, 'Welcome, please enter the & ...'

PRINT statement outputs the string in quotes to standard output channel (screen). The & at the end of the line tells compiler that the line continues and the & at the start of the text tells the compiler to continue the previous line at the character following the &. If this symbol was at the start of the line, or if there were no & on the second line, the string would have a large gap in it because the indentation would be considered as part of the string.

- READ *, a, b, c

READ — waits for three things to be input from the standard input (keyboard). The entities should be separated by a space. Digits will be accepted and interpreted as real numbers; things other than valid numbers will cause the program to crash.

- PRINT *, 'Triangle''s area: ', Area(a,b,c)

the PRINT statement contains a function reference, `Area`, and the output of a string.

The '' are transformed, on output, into a single ', a single quote in their place would not be output because the compiler would think it was the end of the string and would flag the character s as an error. The string could be enclosed by "s which would allow a single ' to be used within the string. The same delimiter must be used at each end of the string.

The function call to `Area` invokes the FUNCTION with the values of `a,b,c` being substituted for `x,y,z`.

- ◊ `a,b,c` are known as actual-arguments,
- ◊ `x,y,z` (in `Area`) are dummy-arguments.
- ◊ `a,b,c` and `x,y,z` are said to be argument associated
- ◊ cannot refer to `a,b,c` in the function they are not in scope.

- END PROGRAM Triangle

The END PROGRAM statement terminates a program.

11: 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
```

Guide for slide: 11

Can be mentioned with respect to the Triangle Example slide.

- the character set now includes amongst others: &, \$ and ;.
- Fortran 90 has two source forms, an old form compatible with that used in FORTRAN 77, **fixed format**, and a new form more suited to the modern computing environment, **free format**. The old form used a very strictly defined layout of program statements on lines. This was well suited to the expression of programs when the main method of entering programs into a computer was by the use of stacks of cards with holes punched in them to represent the characters of the program statements. Such punched card systems also worked with a very restricted set of characters; only upper-case letters, for example, were allowed. The new form is designed to be easier to prepare and to read using the sort of keyboard / display that is now ubiquitous. This new form uses a much wider character set and it allows much greater freedom in laying out statements on a line. It is this new form that we are going to describe here since this is the form we would expect any new programmer to employ. The old form will not be described in any great detail.
- if a ‘!’ appears on a line (but not within a string) then any following text on the same line is treated as a comment.
- the old “character in column 6” method of line continuation is replaced by an & at the end of a line.
 & is ineffective within a comment and cannot be on a line on its own.
- the ‘;’ allows two statements (or more) to occupy the same line.
- in the new free-format blanks are significant. They cannot be embedded into names or keywords but there can be a blank between an END statement and the construct name. In general using common sense will suffice.

12: 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		

Guide for slide: 12

- the _ (the underscore) is useful for punctuating object names.
- although both upper and lower case letters can be used there is no difference and the language is not case sensitive. `integer` has exactly the same meaning as `INTEGER`.

It is good practice to use upper case for keywords and lower case for user defined names. Some may advise that matrix names be in upper case!

- In general the alphanumeric characters can be used in object names and the other symbols are used as delimiters or in mathematical expressions.
- \$ and ? have no real meaning in the language.
- all characters can be embedded within a string.

13: 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).

Guide for slide: 13

- Blanks were not significant to FORTRAN 77 fixed format - they are to Fortran 90 free format.
- Consecutive spaces/blanks have the same meaning as one blank.
- cannot place them
 - ◊ within names, e.g., INT EGER
 - ◊ within keywords, e.g., running total
- must place them
 - ◊ between 2 keywords so the compiler can tell where one finishes and the other starts
 - ◊ between keywords and names for the same reason
- Blanks are optional between certain pairs of keywords such as: END <*construct*>, WHERE <*construct*> is DO, FUNCTION, PROGRAM, MODULE, SUBROUTINE and so on. Blanks are mandatory in other situations. Adding a blank between two valid English words is generally a good idea and will be valid.

14: 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

Guide for slide: 14

- Fortran 90 defines a number of names or keywords. Unlike most languages these keywords may also be used as variable or procedure names although this practise is clearly a terrible idea!
- names must start with **a-z** or **A-Z**. Remember, case is NOT significant. Some compilers allow a \$ or an _ to be used but this is **not** standard conforming.
- as bullet says some compilers allow a \$ or an _ to be used but this is **not** standard conforming.
- This kind of punctuation (_) should be encouraged for readability
- user defined names can be up to 31 chars.
- With the new long names facility in Fortran 90 — symbolic names should be significant and descriptive. Its worth spending a minute or so choosing a good name which other programmers will be able to understand.

15: 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!!!"
```

Guide for slide: 15

- In free format the ! comment initiator is the only way of introducing a comment the C in column 1 is meaningless because column numbers are not counted.
- comments, however brief, should be heartily encouraged!
- everything on a line after the ! is ignored.
- comments cannot be continued onto the next line - must re-initiate them.
- the ! in a character context is treated like any other character and has no special meaning.
- as example indicates, some compilers interpret and act upon structured comments - this is clever as other compilers will ignore them so they don't change the meaning of the program. The compiler manual will describe this form of commentary which is not officially part of the language.

16: Statement Ordering

The following table details the prescribed ordering:

PROGRAM, FUNCTION, SUBROUTINE, MODULE or BLOCK DATA statement		
USE statement		
FORMAT and ENTRY statements	IMPLICIT NONE	
	PARAMETER statement	IMPLICIT statements
	PARAMETER and DATA statements	Derived-Type Definition, Interface blocks function statements and specification statements
	DATA statements	Executable constructs
CONTAINS statement		
Internal or module procedures		
END statement		

Guide for slide: 16

The table (or one very like it) can be found in the Fortran 90 standard.

Fortran has some quite strict rules about the order of statements. Basically in any program or procedure the following rules must be used:

1. The program header statement must come first, (PROGRAM, FUNCTION or SUBROUTINE). PROGRAM statement is optional but its use is recommended.
2. All the specification statements must precede the first executable statement. (Even though DATA statements may be placed with executable text it is far clearer if they lie in the declaration area. It is also a good idea to group FORMAT statements together for clarity.)
3. The executable statements must follow in the order required by the logic of the program.
4. The program or procedure must end with an END statement.

Execution of a program begins at the first executable statement of the **MAIN PROGRAM**, when a procedure is called execution begins with the first executable statement after the invoked entry point. The non-executable statements are conceptually ‘executed’ (!) simultaneously on program initiation, in other words they are referenced once and once only when execution of the main program begins.

PROGRAM, FUNCTION, SUBROUTINE, MODULE or BLOCK DATA statement:

These describe a *program unit*.

- ◊ There can be only 1 main PROGRAM,
- ◊ there may be many uniquely named FUNCTIONS and SUBROUTINES program units (procedures).
- ◊ there may be any number of uniquely named MODULES which are associated with a program through a USE statement. Modules are very flexible program units and are used to package a number of facilities (for example, procedures, type definitions, object declarations, or semantic extensions). Their use is very much encouraged and they replace a number of unsafe features of FORTRAN 77.
- ◊ There can be only one BLOCK DATA subprogram — these will not be described as part of this course — their purpose is to define global constants or global initialisation and this is best done by a MODULE and USE statement.

USE statement:

This attaches a module whose entities become *use-associated* with the program unit. When a module is used its contents are accessible as if they had been declared explicitly in the program unit. Modules may be pre-compiled (like a library) or may be written by the programmer.

Any global entities should be placed in a module and then used whenever access is required.

IMPLICIT NONE - should be after USE statement - its use is implored.

FORMAT and ENTRY - format statements should be grouped together somewhere in the code. ENTRY statements provide a mechanism to ‘drop’ into a procedure halfway through the executable code. Their use is outmoded and strongly discouraged owing to its dangerous nature.

□ **PARAMETER** statement and **IMPLICIT** statement:

IMPLICIT typing should not be used - **IMPLICIT NONE** should be the only form of implicit typing considered. (**IMPLICIT** statements allow the user to redefine the implication of an object's first letter in determining what its implicit type will be if it is not declared. Dangerous feature. Cannot have an **IMPLICIT** statement if there is an **IMPLICIT NONE** line.)

PARAMETER statements - it is suggested that the attributed form of **PARAMETER** declaration be used - i.e., where the **PARAMETER** descriptor is placed in the object definition statement.

□ **DATA** statements should (but don't have to) be placed in the declaration - common practice puts them after the declarations and before the executables.

□ Declarations - obviously any object that is to be used in a declaration must itself have already been defined.

Interface blocks are generally placed at the head of the declarations and are grouped together. Statement functions are a form of in-line statement definition - internal procedures should be used instead

□ executable statements are things like **DO** statements, **IF** constructs and assignment statements.

□ the **CONTAINS** separates the “main” program unit from any locally visible procedures

□ the internal procedures follow the same layout as a (normal) procedure except that they cannot contain a second level of internal procedures.

□ the **END** statement is essential to delimit the current program unit.

17: 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
```

Guide for slide: 17

Each type has

- a name
- a set of valid values
- a means to denote values
- a set of operators

Note

- Most programming languages have the same broad classes of objects
- The three classes cannot be intermixed without explicit coercion being performed.
- NOTE that REAL and DOUBLE PRECISION objects are approximate. DOUBLE PRECISION should not really be used — in FORTRAN 77 an object of this type had greater precision than REAL — in Fortran 90 the precision of a REAL object may be specified making the DOUBLE PRECISION data type redundant.
- all numeric types have finite range.
- the above are default types, non-default types are types of different kinds.

18: 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;
- REALs contain a decimal point, INTEGERS do not,
- REALs 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.

Guide for slide: 18

- there are only two logical values;
- integers are represented by a sequence of digits with a + or - sign, + signs are optional;
- REAL constants contain a decimal point or an exponentiation symbol. INTEGER constants do not.
- character literals are delimited by the double or single quote symbols, " and ';
- two occurrences of the delimiter inside a string produce one occurrence on output for example 'Mau''dib' but not "Mau'"dib" because of the differing delimiters;
- there is only a finite range of values that numeric literals can take — it will be different for each compiler.
- constants may also include a kind specifier but this is described later.

19: 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 REALs.

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 D030I and sets it to 1.1000 instead of performing a loop 1000 times!

Guide for slide: 19

- if the first letter of an undeclared variable is I, J, K, L, M or N the variable is taken to be an INTEGER other wise it is REAL.
- IMPLICIT NONE should be used as a matter of course. Not using it is asking for trouble as undeclared variables slip through.
- the type mapping of the alphabetic letters can be changed by using the IMPLICIT statement.
- the example sighted is reputed to have caused the crash of the American Space Shuttle. An expensive missing comma!

20: 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).

Guide for slide: 20

- in Fortran 90 objects can be attributed at the same time as being declared.
- <*attribute-list*> includes PARAMETER, SAVE, INTENT, POINTER, TARGET, DIMENSION, (for arrays) or visibility attributes — all described later.
- the :: is actually optional, however, it does no harm to use it!
- If <*attribute-list*> is present then so must be ::.
- an object may be given more than one attribute per declaration but some cannot be mixed (such as PARAMETER and POINTER).
- variables may be initialised.

21: 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
```

Guide for slide: 21

- Default CHARACTER s have 1 character.
- CHARACTER(LEN=) is the same as CHARACTER* format
- CHARACTER declarations may be attributed, for example, the POINTER and DIMENSION attributes.
- there is a notable difference between a character variable of 10 characters (CHARACTER(LEN=10) or CHARACTER*10) and a character array of 10 elements. The first is scalar, the second is non-scalar.

22: 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.

Guide for slide: 22

- it is recommended that the attributed form is used
- CHARACTER parameters can assume their length. Can retrieve this length using LEN intrinsic function.

Parameters should be used:

- if it is known that a variable will only take one value;
forces it to be unchanged throughout the program. If any statement tries to change the value an error will result.
- for legibility where a ‘magic value’ occurs in a program such as π ;
symbolic names are easier to understand than a meaningless number.
- for maintainability when a ‘constant’ value could feasibly be changed in the future
Allows it to be adjusted throughout the program by changing 1 line.

23: 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.

Guide for slide: 23

- some compilers zero all objects when the program is loaded; most just allow the variables to contain jibberish, it is good practise to initialise variables.
- variables can be initialised in a number of ways
 - ◊ PARAMETER statements
 - ◊ DATA statements
 - ◊ type declaration statements (with an =*<expression>*) clause.
- can be initialised with constants (literals or PARAMETERS)
- CHARACTER s – if the object and initialisation expression are of different lengths then either
 - ◊ the object will be padded with blanks on the RHS, or
 - ◊ the initialisation expression will be truncated on the right so it will fit.
- can do many initialisations per line.
- constants on LHS of assignments must be able to be represented by the data type on the RHS
- if PARAMETER attribute occurs then objects must be initialised
- dummy args cannot be initialised.
- :: must appear if = does.
- many attributes cannot appear with =, e.g., ALLOCATABLE.
- all objects that are initialised have a static storage class, it is as if they had the SAVE attribute,

Initialisation expressions,

- all information used in an initialisation expression must be available - if you can't see it then neither can the compiler.
- arrays may be initialised by specifying (conformable) scalar value or by using an array constructor.

24: 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.

Guide for slide: 24

- Expressions are made from one operator (e.g., +, -, *, /, // and **) and at least one operand.
- Operands are expressions.
- expressions have types - one of the intrinsic types or a user defined types.
- operators can be monadic and dyadic (take 1 or 2 operands)
- intrinsic operators can be overloaded to apply in more situations.

Intrinsic operators are defined for intrinsic types,

- numeric intrinsic operators are
 - ◊ + addition
 - ◊ - subtraction
 - ◊ * multiplication
 - ◊ / division
 - ◊ ** exponentiation
- CHARACTER has only // concatenation operator
- LOGICAL has all expected boolean operators

User defined operators can be defined to do virtually anything (with 1 or 2 operands) - can be applied to any type or combination of types.

They can be very powerful when used in conjunction with derived types and modules as a package of objects and operators.

25: 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.

Guide for slide: 25

Assignment:

- = is a dyadic operator
- is defined between all numeric types. The two operands of = do not have to be the same intrinsic type.
- is defined between two objects of the same type and for all other types (including user types).
- assignment can be overloaded by writing a procedure with two arguments of the desired type which will define what assignment means between two object of these types. Assignment overloading is discussed later.
- can use with overloaded operators in a module to provide semantic extension, e.g., varying strings module.

NOTE: SIN works in radians,

26: Intrinsic Numeric Operations

The following operators are valid for numeric expressions,

- ** exponentiation, dyadic operator, for example, $10\text{**}2$, (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
```

Guide for slide: 26

Numeric:

- numeric operators behave exactly like you would expect - must take into account the type of the object where the result will be stored (See later.)
- ****** is only operator that is evaluated from right to left, i.e., the exponent is worked out first, all others are evaluated from left to right.
- cannot divide by zero (as usual).
- + and - take 1 or 2 operands
- LHS of ****** can be a scalar, whole array or a subsection of an array - see later.
- operators have a predefined precedence, i.e., what order they are evaluated in, see later.

The example demonstrates,

- **a = b-c** the - operator applied to two objects (arrays or scalar),
- **f = -3*6/5** the use of a monadic - operator. verb f is either array or scalar (can't tell).

27: 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
```

Guide for slide: 27

Substrings are dealt with in more detail later, the example selects

- the whole string
- the first character
- the 2nd, 3rd and 4th characters
- the substring must be specified *from* a position *to* a position, a single subscript is no good.

There is only one intrinsic CHARACTER operation // which concatenates two strings.

- // cannot be mixed with other types
- the object **name** *must* be a PARAMETER in order to appear in an initialisation expression.
- both operands must be of the same kind (see later).

28: 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.
```

Guide for slide: 28

Relational operators

- compare the values of two operands
- deliver a logical result
- can be applied to numeric operands (restrictions on **COMPLEX** — can only use **.EQ.** and **.NE.**).
- can be applied to default **CHARACTER** objects - both objects are made to be the same length by padding the shorter with blanks. These operators refer to ASCII order.
- cannot be applied to **LOGICAL** objects,

The example demonstrates,

- simple logical assignment using a relational operator,
- an IF statement using a relational operator,

29: 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..

Guide for slide: 29

- logical operators can only be used between LOGICAL expressions.
- have a precedence (see later),
- can be used in any LOGICAL context for example, in an IF statement.

30: 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 **.

Guide for slide: 30

- Intrinsically defined order cannot be changed.
- User defined operators have a fixed precedence - cannot change this.
- the order is fairly intuitive,
- we say that the operator with the highest precedence has the *tightest binding*.
- evaluation order can be forced by using parenthesis - expressions in the brackets are evaluated first,
- two operators cannot be adjacent. one times minus one is written $1*(-1)$ and not $1*-1$
- in contexts of equal precedence left to right evaluation is performed except for $**$ where its is right to left. This is important when teetering around the limits of the machines representation. (See later.) Consider $A-B+C$ and $A+C-B$, if A were the largest representable number and C is positive and smaller than B the first expression is OK but the second will crash the program with an overflow error.
- have to be careful with the division operator - it is good practice to put numerator and denominator in parentheses.

$(A+B)/C$

is not the same as

$A+B/C$

but

$(A*B)/C$

is equivalent to

$A*B/C$

31: 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.

Guide for slide: 31

The precedence is worked out as follows

1. in an expression find the operator(s) with the tightest binding
2. if there is more than one occurrence of this operator then the separate instances are evaluated left to right.
3. place the first executed subexpression in brackets to signify this,
4. continue with the second and subsequent subexpressions,
5. move to next most tightest binding operator and follow the same procedure.

First example,

- tightest binding operator is `**` — `c**d` is first executed subexpression - surround with brackets,
- `/` and `*` are the second: `b/5.0` is evaluated next followed by `1*e` - put these in brackets
- `+` and `-` are next in line - they are executed from left to right.
- at last the assignment is done.

Lecture 2:
Control Constructs

33: 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);

All structured programming languages need constructs which provide a facility for conditional execution. The simplest method of achieving this functionality is by using a combination of IF and GOTO, which is exactly what FORTRAN 66 supported. Fortran has progressed since then and now includes a comprehensive basket of useful control constructs.

- conditional execution constructs, (IF ... THEN ... ELSEIF... ELSE ... END IF);

These are the basic conditional execution units. They are useful if a section of code needs to be executed depending on a series of logical conditions being satisfied. If the first condition in the IF statement is evaluated to be true then the code between the IF and the next ELSE, ELSEIF or END IF is executed. If the predicate is false then the second branch of the construct is entered. This branch could be either null, an ELSE or an ELSEIF corresponding to no action, the default action or another evaluation of a (possibly) different predicate with execution being dependent upon the result of the current logical expression. Each IF statement has at least one branch and at most one ELSE branch and may contain any number of ELSEIF branches. Very complex control structures can be built up using multiply nested IF constructs.

END IF terminates an IF block.

Before using an IF statement, a programmer should be convinced that a SELECT CASE block or a WHERE assignment block would not be more appropriate.

- loops, (DO ... END DO);

Basic form of iteration mechanism. This structure allows the body of the loop to be repeatedly executed a number of times. The number of iterations can be a constant, variable (an expression) or can be dependent on a particular condition being satisfied. DO loops can be nested.

- multi-way choice construct, (SELECT CASE);

Due to the nature of this construct it very often works out (computationally) cheaper than an IF block with equivalent functionality. This is because in a SELECT CASE block a single control expression is evaluated once and then its (single) result is compared with each branch. With an IF .. ELSEIF block a different control expression must be evaluated at each branch. Even if all control expressions in an IF construct were the same and were simply compared with different values, the general case would dictate that the SELECT CASE block were more efficient.

There are also two more minor cases:

- unconditional jump statements, (GOTO);

Direct jump to a labelled line - very powerful, very useful and very open to abuse. Unstructured jumps can make a program virtually impossible to follow - the GOTO must be used with care. It is particularly useful for handling exceptions - that is to say - when emergency action is needed to be taken owing to the discovery of an unexpected error.

- I/O exception branching, (ERR=, END=, EOR=);

This is a slightly oddball feature of Fortran in the sense that there is currently no other form of exception handling in the language. (The feature originated from FORTRAN 77.) It is possible to add qualifiers to I/O statements to specify a jump in control should there be

an unexpected I/O data error, should the end of a file be encountered or should there be an unexpected end of record.

It is always good practice to use at least the `ERR=` qualifier in I/O statements and does not harm things to comprehensively guard against any other such errors. See later for more about I/O.

34: 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
```

Guide for slide: 34

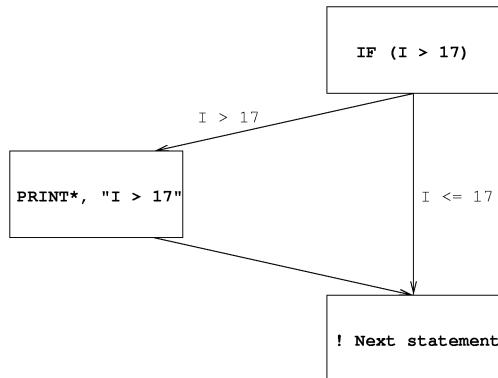
- The basic scenario is if the *<logical-expression>* evaluates to .TRUE. then the *<exec-stmt>* is executed, otherwise it isn't!
- The *<exec-stmt>* cannot be an IF statement or a statement that ends a program unit.
- This is the most basic form of conditional execution in that there is only an option to execute one statement or not — the general IF construct allows a block of code to be executed.
- This type of IF statement still has its use as it is a lot less verbose than a block-IF with a single executable statement.
- The *<logical-expression>* is any expression that evaluates to either .TRUE. or .FALSE.. This has been described earlier. Relational operators are very often used here.
- As REAL numbers are represented approximately, it makes little sense to use the .EQ. and .NE. relational operators between real-valued expressions. For example, there is no guarantee that 3.0 and 1.5 * 2.0 are equal. If two REAL numbers / expressions are to be tested for equality then one should look at the size of the difference between the numbers and see if this is less than some sort of tolerance (using the EPSILON function — see later).

35: Visualisation of the IF Statement

Consider the IF statement

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

this maps onto the following control flow structure,



Guide for slide: 35

36: 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.

Guide for slide: 36

The first example is a 2-way choice. If `x` is zero then the first `PRINT` is executed otherwise the second is executed instead.

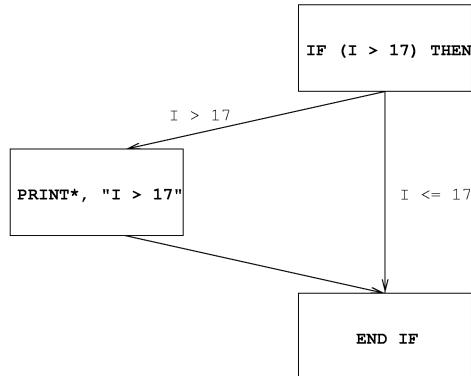
The second example demonstrates `ELSEIF` clauses. If `i .EQ. 0` is not true we must look at whether `i .GT. 0` is. If this is not true we jump to the `ELSE` clause.

37: 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,



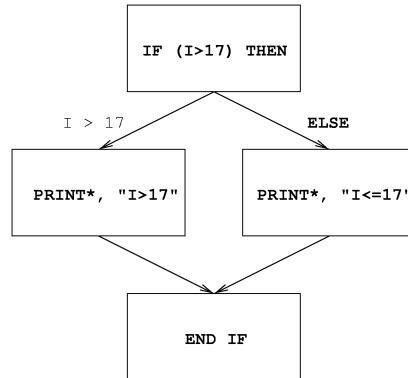
Guide for slide: 37

38: 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,



Guide for slide: 38

39: 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.

Guide for slide: 39

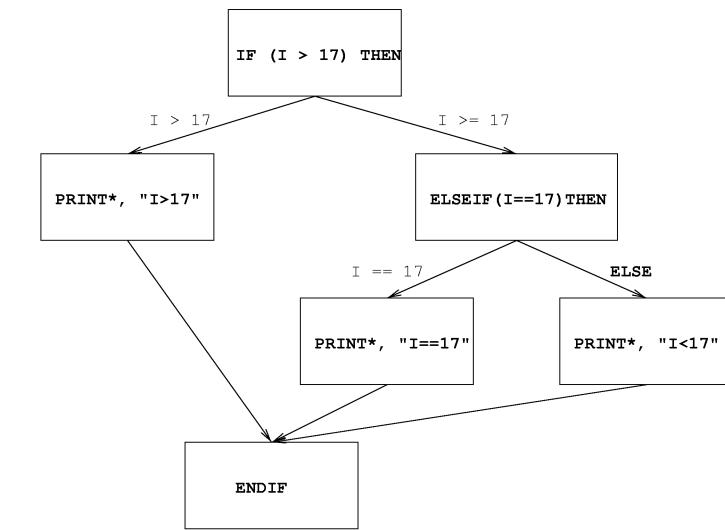
- The block-IF is more flexible than the single statement IF as there can be a number of alternative mutually exclusive branches guarded by (different) predicates. The control flow is a lot more structured than if a single statement IF plus GOTO statements were used. The scenario is that the predicates in the IF or ELSEIF lines are tested in turn - the first one which evaluates as true transfers control to the appropriate inner block of code; when this has been completed control passes to the ENDIF statement and thence out of the block. If none of the predicates are true then the <*else-block*> (if present) is executed.
- Statements in either the <*then-block*>, the <*else-block*> or the <*elseif-block*> may be labelled but jumps to such labelled statements are permitted only from within the block containing them. Entry into a block- IF construct is allowed only via the initial IF statement. Transfer out of either the <*then-block*>, the <*else-block*> or the <*elseif-block*> is permitted but only to a statement entirely outside of the whole block defined by the IF...END IF statements. A transfer within the same block-IF between any of the blocks is not permitted.
- A further IF construct may appear in the <*then-block*>, the <*else-block*> or the <*elseif-block*>. This is now a nested IF structure.
- Please note that both the <*else-block*> and the <*elseif-block*> are optional.
- certain types of statement, e.g., END SUBROUTINE, END FUNCTION or END PROGRAM, statement are not allowed in the <*then-block*>, the <*else-block*> or the <*elseif-block*>.

40: 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,



Guide for slide: 40

41: 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.

Guide for slide: 41

- The example has two nested and one named IF block. Nesting can be to any depth (unless the compiler prohibits this — even if it does the limit will almost certainly be configurable).
- The names are only intended to make the code clearer and serve no real purpose (*cf* DO-loop names which do).
- The name is declared implicitly by naming a construct, names cannot hold any values, be assigned to or be used in any other context.
- If a name is given to the IF statement then it *must* be present on the ENDIF statement but not necessarily on the ELSE or ELSEIF statement. If a name is present on the ELSE or ELSEIF then it must be present on the IF statement.
- cannot have a statement that ends a program unit within an IF block.

42: 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.

Guide for slide: 42

- A loop comprises a block of statements that are executed cyclically. When the end of the loop is reached, the block is repeated from the start of the loop. Loops are differentiated by the way they are terminated. Obviously it would not be reasonable to continue cycling a loop forever, there must be some mechanism for a program to exit from a loop and carry on with the instructions following the End-of-loop.
- The block of statements making up the loop is delimited by DO and END DO statements. This block is executed as many times as is required. Each time through the loop, the condition is evaluated and the first time it is true the EXIT is performed and processing continues from the statement following the next END DO.
- the EXIT statement tells control to jump out of the **current DO** loop, i.e., to the statement after the next END DO.
- The statements between the DO and its corresponding END DO must constitute a proper block. The statements may be labelled but no transfer of control to such a statement is permitted from outside the loop-block. The loop-block may contain other block constructs, for example, DO, IF or CASE, but they must be contained completely; that is they must be properly nested.
- This type of conditional-exit loop is useful for dealing with situations when we want input data to control the number of times the loop is executed.

43: 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.

Guide for slide: 43

- Situations often arise in practice when, for some exceptional reason, it is desirable to terminate a particular pass through a loop and continue immediately with the next repetition or cycle.
- This can be achieved in Fortran 90 by arranging that a CYCLE statement is executed at an appropriate point in the loop.
- If executed, the statement:

```
IF (i >= 50 .AND. I <= 59) CYCLE
```

transfers control to the innermost DO statement which begins a new iteration.

- a CYCLE or EXIT statement which is not within a loop is an error.

44: 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.

Guide for slide: 44

- It is possible to jump out of more than the innermost loop. If the loops are named then it can be specified which loop it is wished to exit.
- In the example the loops are named in the same way as IF statements. If the DO statement has a name then so must the END DO.
- Unlike IF construct names the DO-loop names do serve a syntactic purpose, they can be used to modify control flow.
- IF (a.GT.b) EXIT outa causes control to immediately jump to the last statement shown, the END DO.
- IF (c.GT.d) EXIT inna jumps to the statement after END DO inna.
- if the name is missing then the directive applied, as usual, to the next outermost loop.

45: 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 $<\text{expr3}>$ is absent it is assumed to be equal to 1.

Guide for slide: 45

- An indexed loop could be achieved using an induction variable and EXIT statement, however, the indexed DO loop is better suited as there is less scope for error.
- The loop can be named and the $\langle exec-stmts \rangle$ could contain EXIT or CYCLE statements.
- the loop runs from $\langle expr1 \rangle$ to $\langle expr2 \rangle$ in steps of $\langle expr3 \rangle$.
- an indexed loop **cannot** have a WHILE clause, however, this can be simulated with an EXIT statement if desired.
- Note: Morgan and Schonfelder book is wrong with iteration count.
- if the $\langle expr3 \rangle$ (stride) is missing it is assumed to be 1 this is the only index specifier that is optional.
- it seem to be a common misconception that Fortran loops always have to be executed once; this came from FORTRAN 66 and is now incorrect. Zero executed loops are useful for programming degenerate loops.
- the iteration count is worked out as follows (adapted from the standard, [?]):
 1. $\langle expr1 \rangle$ is calculated,
 2. $\langle expr2 \rangle$ is calculated,
 3. $\langle expr3 \rangle$, if present, is calculated,
 4. the DO variable is assigned the value of $\langle expr1 \rangle$,
 5. the iteration count is established (using the formula on the slide)
- then the execution cycle is performed as follows (adapted from the standard):
 1. the iteration count is tested and if it is zero then the loop terminates,
 2. if it is non zero the loop is executed,
 3. (conceptually) at the END DO the iteration count is decreased by one and *the DO variable is incremented by $\langle expr3 \rangle$*
 4. control passes to the top of the loop again and the cycle begins again
- note how the DO variable can be greater than $\langle expr2 \rangle$.
- the DO variable cannot be assigned to within the loop.

46: 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
```

Guide for slide: 46

1. upper bound not exact,

According to the rules (given with the previous slide) the fact that the upper bound is not exact is not relevant, the iteration count is $\text{INT}(29/2) = 14$, so *i* will take the values 1,3,..,27,29 and finally 31 although the loop is not executed when *i* holds this value - this is its final value

2. negative stride,

similar to above except the loop runs the other way. *j* will begin with the value 30 and will end up being equal to 0.

3. a zero-trip loop,

This is a false example in the sense that the loop bounds are literals and (it is hoped) there would be no point in coding a loop of this fashion as it would never ever be executed! However, if symbolic bounds were to evaluate to these values, then, the loop would not be executed. Firstly, *k* is set to 30 and then the iteration count would be evaluated and set to 0. This would mean that the loop is skipped - the only consequence of its existence being that *k* holds the value 30.

4. missing stride,

As the stride is missing it must take its default value which is 1. This loop runs from 1 to 30 so the implied stride means that the loop is executed 30 times.

47: 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.

Guide for slide: 47

Fortran 90 is not block structured, all DO variables are visible after the loop and have a specific value. The example here demonstrates this well, the loop is executed three times and I is assigned to 4 times.

I can be freely used elsewhere in the program and used in other loops if desired.

48: 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*, "Meeeeoow"
END SELECT
! ENDIF
```

Guide for slide: 48

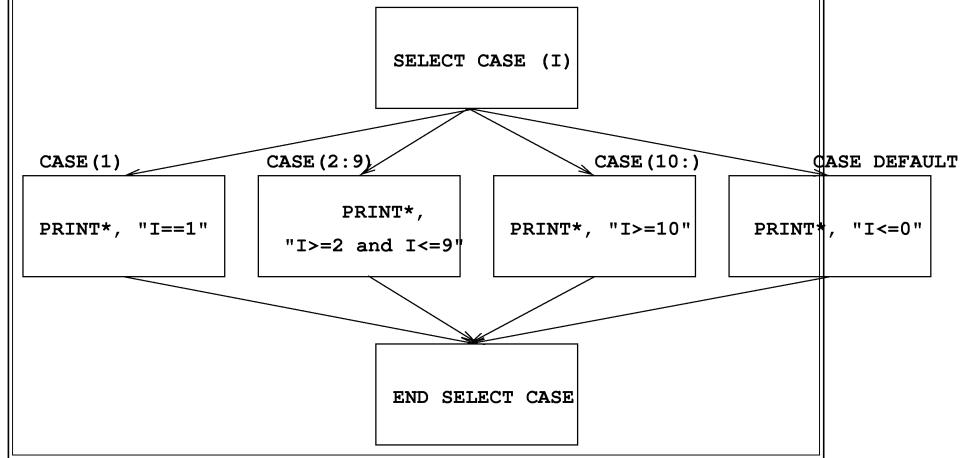
- the $<case-expr>$ in this case is `num`.
- the first $<case-selector>$, `(6,9,99,66)` means “if `num` is equal to either 6, 9, 66 or 99 then”,
- the second $<case-selector>$, `(10:65,67:98)` means “if `num` is between 10 and 65 (inclusive) or 67 and 98 (inclusive) then”,
- if a case branch has been executed then when the next $<case-selector>$ is encountered control jumps to the `END SELECT` statement.
- if the above case expression is not satisfied then the next one is tested. `(100:)` specifies the range of greater than or equal to one hundred. If `num` satisfies this then the computer barks(!)
- a case structure can also operate on a scalar `CHARACTER <case-expr>` which could be matched to a list of possible character values or, indeed, a list of possible ranges.

49: 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,



Guide for slide: 49

50: 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.

Guide for slide: 50

- **SELECT CASE** is more efficient than **ELSEIF** because there is only one expression that controls the branching. The expression needs to be evaluated once and then control transferred to whichever branch corresponds to the expressions value. An **IF .. ELSEIF ...** has the potential to have a different expression to evaluate at each branch point making it less efficient.
- **CASE** constructs may be named - if the header is named then so must be the **END SELECT** statement. If any of the **CASE** branches are named then so must be the **SELECT** statement and the **END** statement.
- the **<case-expr>** is evaluated and compared with the **<case-selector>**s in turn to see which branch to take. The **<case-expr>** must be either scalar **INTEGER**, **CHARACTER** or **LOGICAL** valued.
- the **<case-selector>**s can specify a single value, a list or a bounded or unbounded range or a list of ranges. The first one which is encountered which contains the (scalar) **<case-expr>** is the branch which is chosen.
- there can be any number of **<case-selector>**s.
- the **<case-selector>** is a single value or range specifier surrounded by parenthesis.
- a range specifier is a lower and upper limit separated by a single colon. One or other of the bounds is optional giving an open-ended range specifier.
- if no branches are chosen then the **CASE DEFAULT** is executed (if present).
- there can only be one **CASE DEFAULT** block.
- when the **<exec-stmts>** in the selected branch have been executed control jumps out of the **CASE** construct (via the **END SELECT** statement).
- as with other similar structures it is not possible to jump into a **CASE** construct.
- any number of **CASE** constructs can be nested.

51: 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
```

Guide for slide: 51

- The simplest form output is the PRINT statement.
- The output goes to the standard output channel.
- the * takes the place of a format statement number - this point should be glossed over at the present time.
- the PRINT statement takes a comma separated list of things to print!
- each PRINT statement begins a new line.
- the ‘arguments’ to the print statements can be (virtually) anything.
 - ◊ literal (numeric, character, boolean).
 - ◊ variables or expressions (numeric, character, boolean).
 - ◊ scalar or array valued.

Cannot be a derived type with a POINTER or private component (see later).

- PRINT*, performs no (controllable) formatting - Fortran 90 supports a great wealth of output (and input) formatting which is not described here!
- PRINT*, long_name simply prints out variable contents!

```
PRINT*, "Spock says ""illogical&
&Captain"" "
```

- if a CHARACTER string crosses a line indentation can still be used if an & is appended to the end of the first line and the position from where the string is wanted to begin on the second - see the Spock line in the example. The &s act like a single space.
- note the double " in the string the first one *escapes* the second and allows it to be printed.
- notice how the output has many more spaces than the PRINT statement indicates. This is because output is unformatted. The default formatting is likely to be different between compilers.
- LOGICAL values are always represented a T or F

52: 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;

Guide for slide: 52

- non-advancing I/O is available - have to specify it (see later).
- the PRINT statement can transfer any object of intrinsic type to the standard output - can also do the non-default kinds (which of course are intrinsic types).
- a string cannot contain a single unescaped delimiter of the same type that is used to delimit it. In order to output a " in the middle of a string which is delimited by " characters a double " must be used. Like wise for the ' delimiter. If the alternative delimiter is used then a single instance is OK. For example

```
"Man from Delmonte, he say 'Yes'"  
'Man from Delmonte, he say "Yes"'
```

53: 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;

Guide for slide: 53

- the * format specifier the READ statement is comparable to the functionality of PRINT i.e., unformatted data can be read.
- actually this is not strictly true, formatted data can be read but the format cannot be specified.
- can have more than one object per READ.
- each separate input entity must be blank separated.

t two are simply ignored.

- any intrinsic type can be read including non-default kinds. (No default kind values will look very much like normal values except they may not fit into the representation of default kinds.

54: 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.

Guide for slide: 54

- Must take care with mixed mode arithmetic, (expressions containing objects of different types).
- Numeric and non-numeric types cannot be mixed using intrinsic operators.
- Cannot mix LOGICAL and CHARACTER.
- In general, for a numeric subexpression (of one operator and two operands) if the types are different then the lower type is promoted or coerced to a higher type. The types have the following ordering:
 1. INTEGER - lowest
 2. REAL
 3. DOUBLE PRECISION
 4. COMPLEX - highest

Thus for a subexpression mixing an INTEGER and a REAL the INTEGER is first converted to a REAL and then the expression is evaluated.

55: 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.

Guide for slide: 55

When the whole RHS expression of a mixed type assignment statement has been evaluated it has a type, this type must then be converted to fit in with the LHS. This conversion could be either a promotion or a relegation. For example, RHS = INTEGER, LHS = REAL the LHS needs promoting to be a REAL value; RHS = REAL and LHS = INTEGER, the REAL needs relegating to be INTEGER.

- LHS INTEGER; RHS REAL— the right hand side is evaluated and then the value is truncated (all the decimal places lopped off) then assigned to the LHS.
- LHS REAL; RHS INTEGER— the right hand side expression is simply stored (approximately) in the LHS. The INTEGER is promoted as if it were an actual-argument of the REAL intrinsic function.

Note: as real values are stored approximately

```
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
```

- j would truncated to 0 because numbers are aways truncated *towards* zero.
- the result of a - b would be close to 1.0 (it could be 1.0000001 or it could be 0.999999999) this means that after truncation k could contain either 0 or 1,

56: 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.

Guide for slide: 56

- If one integer divides another in a subexpression then the type of that subexpression is **INTEGER** even if the context is a simple assignment statement with only one operator and two operands (like in the slide).
- in other words the following happens:
 1. the RHS is evaluated
 2. the integral result of the RHS is converted to a **REAL** value before assignment to the LHS.
- The integer expression $1999/1000$ is evaluated and then truncated towards zero to produce an integral value. It says in the Fortran 90 standard, P84 section 7.2.1.1, “The result of such an operation [integer division] is the integer closest to the mathematical quotient and between zero and the mathematical quotient inclusively.”
- likewise $-1999/1000$.
- c is 2.0 because $2000/1000$ is calculated.
- d and e are both 1.999 because both RHS’s are evaluated to be real numbers, in $1999.0/1000$ $1999/1000.0$ the integers are promoted to real numbers before the division.

57: 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 SUBROUTINEs), for example, SYSTEM_CLOCK and DATE_AND_TIME.

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

These procedures vary in what arguments are permitted. Some procedures can be applied to scalars and arrays, some to only scalars and some to only arrays.

- elemental - this means they apply to scalar objects as well as arrays — when an array argument is supplied the same function is applied to each element of the array at (conceptually) the same time.
 - ◊ numeric, for example, `SUM` or `SIN`;
 - ◊ character, for example, `INDEX` and `TRIM`;
 - For example, string manipulation, relational comparisons.
 - ◊ bit, for example, `IAND` and `IOR`;
 - A collection of intrinsics for manipulating and setting up bit variables,
- inquiry, for example, `ALLOCATED` and `SIZE`;
 - Can find out about:
 - ◊ the status of dynamic objects.
 - ◊ array bounds, shape and size.
 - ◊ kind parameters of an object and available kind representations, (useful for portable code).
 - ◊ the numerical model; used to represent types and kinds.
 - ◊ argument presence (for use with `OPTIONAL` dummy arguments).
- transformational, for example, `RESHAPE` and `TRANSPOSE`. (Mainly arrays)
 - ◊ `REPEAT` (for characters - repeats strings)
 - ◊ mathematical reduction procedures, i.e., given an array return an object of less rank;
 - ◊ array manipulation - shift operations, `RESHAPE`, `PACK`,
 - ◊ type changing, `TRANSFER` copies bit-for-bit to an object of a different type. (Stops people doing dirty tricks like changing the type of an object across a procedure boundary.)
 - ◊ `PRODUCT` and `DOT_PRODUCT` (arrays),
- miscellaneous (non-elemental) including timing routines, for example, `SYSTEM_CLOCK`, `DATE_AND_TIME`, random number seeding

58: 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
```

Guide for slide: 58

- argument to REAL can be INTEGER, DOUBLE PRECISION or COMPLEX.
- argument to INT can be REAL, DOUBLE PRECISION or COMPLEX.
- argument to DBLE can be INTEGER, REAL or COMPLEX.
- there are also other functions to get integers from non-integer values:
 - ◊ CEILING — smallest integer greater or equal to the argument,
 - ◊ FLOOR — largest integer less or equal to the argument,
 - ◊ NINT — nearest integer
- CMPLX converts to a complex value.
- argument IACHAR must be a single CHARACTER.
- argument ACHAR must be a single INTEGER.

59: 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

Guide for slide: 59

The slide details the function names and, when necessary, details about the required arguments.

As all are elemental they can accept array arguments, the result is the same shape as the argument(s) and is the same as if the intrinsic had been called separately for each array element of the argument.

- ASIN, ACOS — arcsin and arccos.

The argument to each must be real and $\leq |1|$, for example, ASIN(0.84147) has value 1.0 (radians).

- ATAN — arctan.

The argument must be real valued, for example, ATAN(1.0) is $\frac{\pi}{4}$, ATAN(1.5574) has value 1.0.

- ATAN2 — arctan; the principle value of the nonzero complex number (X,Y), for example, ATAN2(1.5574,1.0) has value 1.0.

The two arguments (Y, X) (note order) must be real valued, if Y is zero then X cannot be. These numbers represent the complex value (X,Y).

- TAN, COS, SIN — tangent, cosine and sine.

Their arguments must be real or complex and are **measured in radians**, for example, COS(1.0) is 0.5403.

- TANH, COSH, SINH — hyperbolic trig functions.

The actual arguments must be REAL valued, for example, COSH(1.0) is 1.5431.

- EXP, LOG, LOG10, SQRT — e^x , natural logarithm, logarithm base 10 and square root.

The arguments must be real or complex (with certain constraints). Note that the argument to SQRT cannot be INTEGER, for example, EXP(1.0) is 2.7182.

Note all angles are expressed in radians.

60: Numeric Intrinsic Functions

Summary,

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

Guide for slide: 60

The slide gives details regarding the names and arguments of these intrinsics.

As all are elemental they can accept array arguments, the result is the same shape as the argument(s) and is the same as if the intrinsic had been called separately for each array element of the argument.

ABS — absolute value.

The actual argument can be **INTEGER**, **REAL** or **COMPLEX**, the result is the same type as the argument except for complex where the result is real valued, for example, **ABS(-1)** is 1, **ABS(-.2)** is 0.2 and **ABS(CMPLX(-3.0,4.0))** is 5.0.

AINT — truncates to a whole number.

The argument and result are real valued, for example, **AINT(1.7)** is 1.0 and **AINT(-1.7)** is -1.0.

ANINT — nearest whole number.

The argument and result are real valued, for example, **AINT(1.7)** is 2.0 and **AINT(-1.7)** is -2.0.

CEILING, **FLOOR** — smallest **INTEGER** greater than (or equal to), or biggest **INTEGER** less than (or equal to) the argument.

Argument is **REAL**, for example, **CEILING(1.7)** is 2 **CEILING(-1.7)** is 1.

CMPLX — converts to complex value.

The argument must be two real numbers, for example, **CMPLX(3.6,4.5)** is a complex number.

DBLE — coerce to **DOUBLE PRECISION** data type.

Arguments must be **REAL**, **INTEGER** or **COMPLEX**. The result is the actual argument converted to a **DOUBLE PRECISION** number.

DIM — positive difference.

Arguments must be **REAL** or **INTEGER**. If X bigger than Y then **DIM(X,Y)** = **X-Y**, if **Y > X** and result of **X-Y** is negative then **DIM(X,Y)** is zero, for example, **DIM(2,7)** is 0 and **DIM(7,2)** is 5.

INT truncates to an **INTEGER** (as in integer division)

Actual argument must be numeric, for example **INT(8.6)** is 8 and **INTCMPLX(2.6,4.0)** is 2.

MAX and **MIN** — maximum and minimum functions.

These must have at least two arguments which must be **INTEGER** or **REAL**. **MAX(1.0,2.0)** is 2.0,

MOD — remainder function.

Arguments must be **REAL** or **INTEGER**. The result is the remainder when evaluating **a/p**, for example, **MOD(9,5)** is 4, **MOD(-9.0,5.0)** is -4.0.

MODULO — modulo function.

Arguments must be **REAL** or **INTEGER**. The result is **amodb**, for example, **MODULO(9,5)** is 4, **MODULO(-9.0,5.0)** is 1.0.

- **REAL** — converts to REAL value.

For example, **REAL(5)** is 5.0

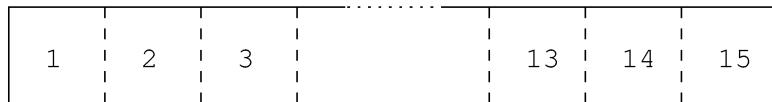
- **SIGN** — transfers the sign of the second argument to the first.

The arguments are real or integer and the result is of the same type and is equal to $\text{ABS}(a) * (b/\text{ABS}(b))$, for example, **SIGN(6,-7)** is -6, **SIGN(-6,7)** is 6.

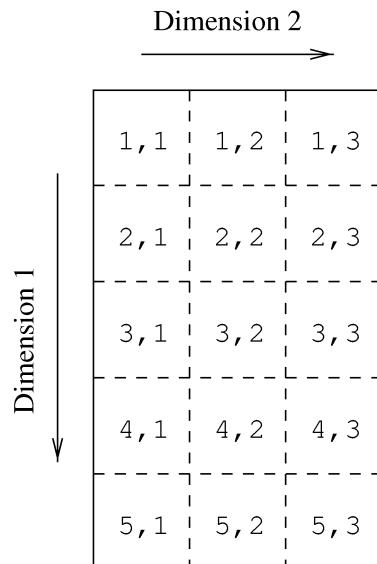
Lecture 3:
Arrays

62: 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 5×3 array as:



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

Guide for slide: 62

63: 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.

Guide for slide: 63

If the lower bound is not explicitly stated it is taken to be 1.

- *explicit-shape arrays* can have symbolic bounds so long as they are initialisation expressions — evaluable at compile time.
- *rank* — up to and including 7 dimensions.
- *extent* — can be zero
- *size* — can specify size for a particular dimension.
- *shape* — shape is expressed as an array of extents
- *conformable* — for operations between two arrays the shapes (of the sections) must (generally) conform (just like in mathematics).
- there is no *storage association* for Fortran 90 arrays.

64: 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`),

Guide for slide: 64

Declared using the **DIMENSION** attribute

- **REAL, DIMENSION(-10:-1) :: X** demonstrates that bounds can begin below zero,
- if the lower bound is not specified then the default lower bound is 1,
- initialisation expressions are allowed in bound specifications expressions,
- zero sized arrays are useful because no extra code has to be added to test for zero extents in any of the dimensions - statements including references to zero sized arrays are generally ignored. If **lda** were set to be zero,

```
INTEGER, PARAMETER :: lda = 0
```

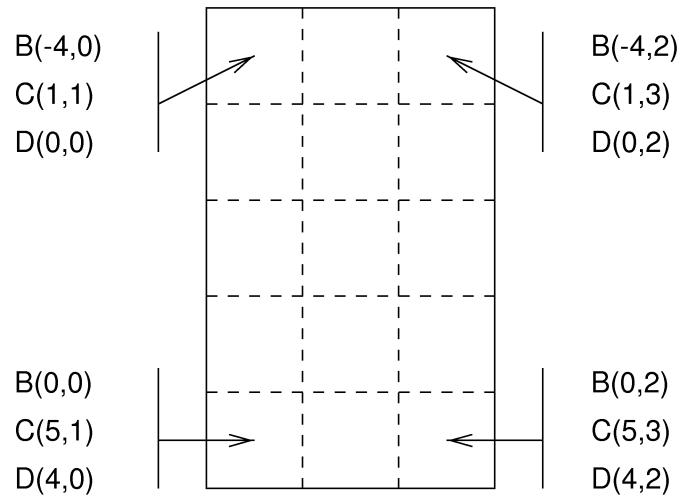
then the array Y would be zero sized.

- there are different types of arrays - these will be covered later.

65: 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) 7th element of A, or C(3,2), 3 elements down, 2 across.



Guide for slide: 65

This slide allows visualisation of the arrays which were declared on the previous slide.

- The top matrix (**A**) is a 1D 15 element vector. The default lower bound is 1.
- The lower matrix is a 5×3 2D array, it is the same shape as **B**, **C** and **D** and the corresponding elements of each matrix are shown.

66: Array Conformance

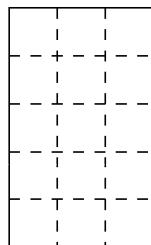
Arrays or sub-arrays must conform with all other objects in an expression:

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

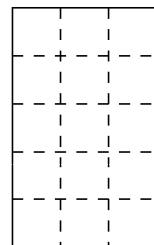
`C = 1.0 ! is valid`

- two array references must conform in their shape.

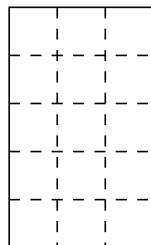
Using the declarations from before:



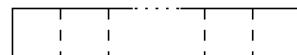
`C = D`



Valid



`B = A`



Invalid

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

Guide for slide: 66

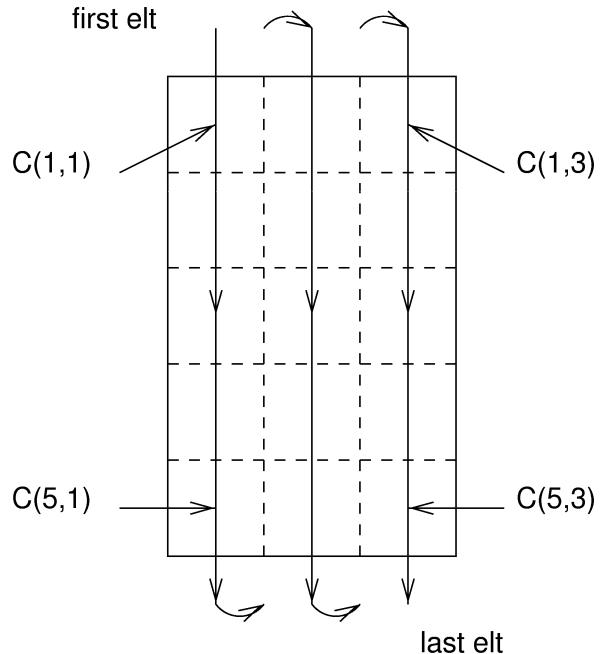
- $C = 1.0$: the value 1.0 is assigned to all elements of C.
- C and D are the same shape so conform.
- A = B: even though A and B have the same number of elements (15) the two arrays do not conform - conformable means 'same shape'.
- two arrays of different types can be used in the same expression and will have the relevant type coercion performed just like scalars.
- there is a difference between a scalar and an array with one element.
- see later for intrinsics that are able to change the shape of arrays.

67: 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:



$C(1,1), C(2,1), \dots, C(5,1), C(1,2), C(2,2), \dots, C(5,3)$

Guide for slide: 67

- the lack of implicit storage association makes it easier to write portable programs and allows compiler writers more freedom to implement local optimisations. For example, on distributed memory computers an array may be stored over 100 processors with each processor owning only a small section of the whole array.
- FORTRAN 77 did have storage association. It specified that array elements were stored in column major form, (i.e. with the indices of the lowest dimension varying most rapidly).
- This ordering is used in array constructors, I/O statements, certain intrinsics (`TRANSFER`, `RESHAPE`, `PACK`, `UNPACK` and `MERGE`) and any other contexts where an ordering is needed.

68: 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.

Guide for slide: 68

- A particular element of an array is accessed by subscripting the array name with an integer which is within the bounds of the declared extent. Subscripting directly with a **REAL**, **COMPLEX**, **CHARACTER**, **DOUBLE PRECISION** or **LOGICAL** is an error.
- A whole array can be referenced by using only its name with no subscripts nor parentheses.
- A whole array can also be referenced by subscripting it with a single colon this is shorthand for `lower_bound:upper_bound`.
- B and C must be same shape.
- `A(1)` is a Scalar variable.
- Scalars (literals and variables) conform to arrays.
- Parts of arrays can be referenced, these are specified using the colon range notation first encountered in the **SELECT CASE** construct. In addition, the sequence of elements can also have a stride (like **D0** loops) thus the section specification is denoted by a *subscript-triplet* which is a linear function.
- In summary both scalars and arrays can be thought of as objects - (more or less) the same operations can be performed on each with the array operations being performed in parallel.
- All the elemental procedures mentioned before (**SIN**, **LOG**, **IBTEST**, etc etc) work with array arguments.

69: 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.

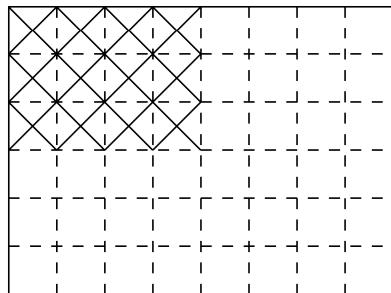
Guide for slide: 69

- A whole (or section of an) array can be treated like a single variable in that all intrinsic operators which apply to intrinsic types have their meaning extended to apply to conformable arrays.
- The assignment $B = C * D - B^{**2}$ is an elemental assignment and is performed between each corresponding element of each array.
- Clearly $C*D$ is not ‘traditional’ matrix multiplication. For this effect `MATMUL(C,D)` should be used instead.
- Recall that the RHS operand of the `**` operator must be scalar.
- Operators have the same precedence as for scalars and behave in exactly the same way with all their operations being performed in parallel.
- Many of the intrinsics are elemental including all numeric, mathematical, bit, character and logical intrinsics.

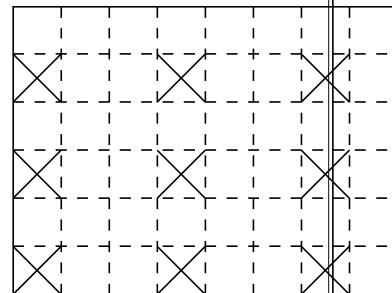
70: Array Sections — Visualisation

Given,

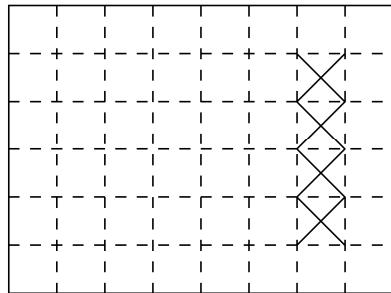
```
REAL, DIMENSION(1:6,1:8) :: P
```



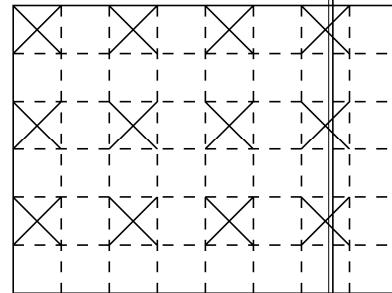
$P(1:3,1:4)$



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



$P(2:5,7)$



$P(2:5,7:7)$

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

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.

Guide for slide: 70

The sections:

- $P(1:3,1:4)$ is a 3×4 section; the missing stride implies a value of 1,
- $P(2:6:2,1:7:3)$ is a 3×3 section.
- $P(2:5,7)$ is a 1D array with 4 elements.
- $P(2:5,7:7)$ is a 4×1 section.
- $P(1:6:2,1:8:2)$ is a 3×4 section.

Conformance:

- $P(1:3,1:4) = P(1:6:2,1:8:2)$ is valid — both LHS and RHS are 3×4 sections.
- $P(1:3,1:4) = 1.0$ is valid as a scalar on the RHS conforms to any array on the LHS of an assignment.
- $P(2:6:2,1:7:3) = P(1:3,1:4)$ is not — trying to equate a 3×3 section with a 3×4 section, the arrays do not conform.
- $P(2:6:2,1:7:3) = P(2:5,7)$ is not trying to equate a 3×3 section with a 4 element 1D array.

Also note

- $P(2:5,7)$ is a 1D section — the scalar in the second dimension ‘collapses’ the dimension.
- $P(2:5,7:7)$ is a 2D section — the second dimension is specified with a section (a range) not a scalar so the resultant sub-object is still two dimensional.

71: 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.

Guide for slide: 71

- the stride specification follows the same principle as for a DO loop — if the upper bound is not a combination of the lower bound plus multiples of the stride then the actual upper bound is different from that stated.
- negative strides are permitted.
- zero sections are permitted
- so long as the section specifiers are scalar integers then any expression (even involving arrays and reduction functions) can be used.
- if $<bound1>$ or $<bound2>$ are absent then the lower or upper bound of the dimension is implied. Thus $A(:)$ refers to the whole array.
- if the $<stride>$ is absent then it is taken to be 1.
- $A(m:n:k)$ - from m to n in strides of k .
- $A(m:n:k)$ - from m to n in strides of k .
- $A(8:3:-1)$ - from 8 to 3 in steps of -1
- $A(8:3)$ - from 8 to 3 in steps of 1, i.e., this is a zero sized section
- $A(m:)$ - from m to the upper bound in steps of 1
- $A(:n)$ - from the default LWB to n in steps of 1
- $A(:,:,2)$ from the lower bound to the upper bound in strides of 2.
- $A(m:m)$ one element array. Different from $A(m)$.

72: 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.

Guide for slide: 72

This is the same order as FORTRAN 77 defined for arrays.

- An array of more than one dimension is not formatted neatly, if it is desired that the array be printed out row-by-row (or indeed column by column) then this must be programmed explicitly.
- when reading in an array any number of carriage returns may be inserted amongst the data — they are simply ignored.

73: Array I/O Example

Consider the matrix A:

1	4	7
2	5	8
3	6	9

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
```

Guide for slide: 73

- Arrays are simply output in "array element order"
- Note that the default format is used and it may not look like this!
- Any functions are evaluated before output.

74: 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;

Guide for slide: 74

These intrinsics allow the user to quiz arrays about their attributes and status and are most often used on Dummy arguments in procedures.

- **LBOUND, UBOUND**, give the bounds of the array a viewed in the current scoping unit. The result is a one dimensional array containing the bounds of an array or, if a dimension is specified, a scalar containing the bound in that dimension.

When an array is an assumed shape dummy argument, its bounds always begin from 1 - this is not necessarily the case for the actual argument. The result is returned

- if the **DIM=** argument is present then the **LBOUND, UBOUND**, result is always scalar,
- for **SHAPE**, the result is always array valued unless the argument is scalar,
- for the **SIZE** intrinsic, **SOURCE** must always be an array and the result is *always* scalar,
- **ALLOCATED** returns **.TRUE.** or **.FALSE.** depending on array status (allocated or not).
- all these intrinsics are useful in procedures - see later!

75: 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.

Guide for slide: 75

- the (/ and /) delimit the values.
- the values are placed into the array in array element order.
- anything within the delimiters must be a scalar expression.
- `heights(1:4)` is filled in order with the values 5.10, 5.6, 4.0, 3.6.
- for CHARACTER constructors the string within the constructor must be the correct length for the variable, this is why 'RED' is padded with blanks.
- `ints` will contain the values (/ 100, 1, 2, 3, 4, 5, 6, 7, 8, 100 /).
- an implied DO may be used in the constructor to specify a sequence of constructor values. There may be any number of separate implied DOs which may be mixed with other specification methods.
- type conversions are performed as in regular assignment.
- only 1D constructors are permitted - see later for RESHAPE.
- the constructor must be of the correct length for the array section - they must conform.

76: The RESHAPE Intrinsic Function

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

RESHAPE(SOURCE, SHAPE)

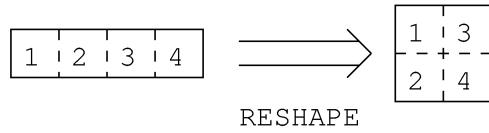
For example,

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

A is filled in array element order and looks like:

1 3
2 4

Visualisation,



Guide for slide: 76

- **RESHAPE** changes the shape of **SOURCE** to the specified **SHAPE**.
- **SOURCE** must be intrinsic typed array, it cannot be an array of user-defined types,
- **SHAPE** is a one dimensional array specifying the target shape. It is convenient to use an explicit array constructor for this field in order to make things clearer,
- in the example a 2×2 array is being constructed,
- clearly the result of **RESHAPE** must conform to the array object on the LHS of the =,

77: 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((/1,0,0,0,1,0,0,0,0,1/),(/3,3/))
```

Guide for slide: 77

- PARAMETERs declared as expected.
- RESHAPE can be used in initialisation statements as long as all components of it can be evaluated at compile time - (just like in initialisation expressions).
- note how in a PARAMETER initialisation, the length of the string `lights` can be assumed from the length of the constructor values. The strings in the constructor *must* all be the same length.
- because `unit_matrix` is a PARAMETER it may also be used in initialisation expressions.

78: 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.

Guide for slide: 78

- Fortran 90 has dynamic storage this means memory can be grabbed (from a part of memory called the *heap store*), used and then put back at any point in the program.
- This allows the creation of "temporary" arrays which can be created used and discarded
- there is a certain overhead in managing dynamic or ALLOCATABLE arrays - explicit-shape are cheaper and should be used if the size is known and the arrays are persistent (i.e., are used for most of the life of the program).
- the dynamic or heap storage is also used with pointers and obviously only has a finite size - there may be a time when this storage runs out. There will almost certainly be an option of the compiler to specify / increase the size of heap storage.
- *deferred-shape* arrays have deferred bound specification - hence their name.
- *deferred-shape* arrays *must* be given the ALLOCATABLE attribute.
- the ALLOCATE statement grabs the heap storage and sets up an array. with specified bounds.
- they follow the same general rules as "normal" arrays, i.e., bounds can begin anywhere, zero sized arrays are allowed, etc.
- cannot pass *unallocated* allocatables as actual arguments.
- the STAT=reports on success / failure of storage request. If it is supplied then the keyword must be used to distinguish it from an array that needs allocating.
- ierr is INTEGER and should always be used.
- a list of objects can be allocated together but ideally only one array per ALLOCATE statement should be given. If there is more than one and the allocation fails it is not possible to easily tell which allocation was responsible.

79: 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 DEALLOCATED then this storage becomes inaccessible.

Guide for slide: 79

- as a matter of course the intrinsic inquiry function, **ALLOCATED**, should be used to check on the status of the array before attempting to **DEALLOCATE**,
- the **STAT=** field is optional but its use is recommended, it reports on the success / failure of the **DEALLOCATE** request in an analogous way to the **ALLOCATE** statement.
- see later for procedures and the **SAVE** attribute - always deallocate arrays as soon as they are finished with.

80: 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

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

and,

$$I = \begin{pmatrix} [2] & 0 \\ 0 & [2] \end{pmatrix}$$

then

$$A = \begin{pmatrix} [0.5] & \cdot \\ \cdot & [2.0] \end{pmatrix}$$

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

Guide for slide: 80

- the WHERE statement is used when an array assignment is to be performed on a subset of the LHS array elements.
- cannot have a scalar on the LHS in a WHERE statement / construct.
- in the first example, if I is a 2D array, for all values of j and k where the mask, ($I(j,k) \neq 0$), is .TRUE. the assignment $A(j,k) = B(j,k)/I(j,k)$ is performed for the cases where the mask is .FALSE. no action is taken.
- conformable array sections may be used in place of the whole arrays.
- every WHERE statement must contain an array assignment statement.

81: 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.

Guide for slide: 81

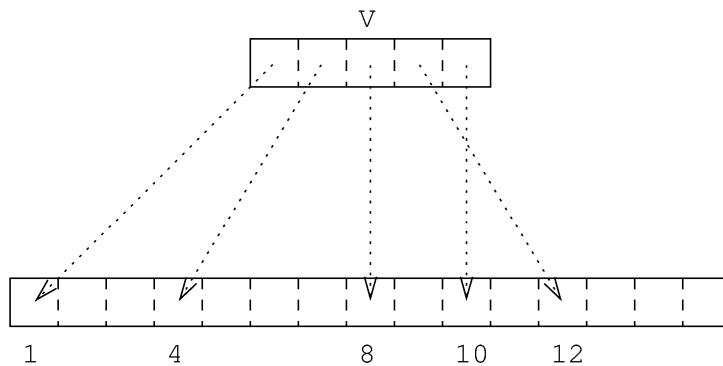
- all statements in a WHERE construct must be array assignments.
- wherever $(A(j,k) > 0.0)$ is .TRUE., the assignments $B(j,k) = \text{LOG}(A(j,k))$ and $C(j,k) = \text{SQRT}(A(j,k))$ are made, otherwise the assignments in the ELSEWHERE block are made instead.
- note that the two branches are executed sequentially.
- the separate assignment statements are executed sequentially but the individual elemental assignments within each statement are (conceptually) executed in parallel.
- conformable array sections may be used in place of the whole arrays.
- all statements within a WHERE must be array assignments.

82: 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))
```

Guide for slide: 82

- Indexing indirection can be introduced by using vector-valued subscripts, this is where a 1D vector is substituted for the subscript-triplet specifier.
- This means that an array section can be specified where the order of the elements do not follow a linear pattern.
- This form of subscripting is **very inefficient** and should not be used unless absolutely necessary.
- Operations conceptually done in parallel.
- Vector-valued Subscripts can be used on either side of the assignment operator.
- Can refer to the same element twice in the same statement on the RHS.
- In order to preserve the integrity of parallel array operations we cannot assign to the same element twice in the same statement, i.e., subscripts on the LHS of the = must be unique. Fortran 90 conceptually assigns to all array elements referenced in a single statement at the same time, clearly this is not possible when the same elements are assigned to twice.
- It is only possible to use one dimensional arrays as vector subscripts, if a 2D section is to be defined then two 1D vectors must be used,

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

is valid

83: 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))
```

Guide for slide: 83

- useful when developing and testing code for setting up random valued objects.
- **RANDOM_SEED** can be used to modify the behaviour of **RANDOM_NUMBER**. By supplying different parameters to this intrinsic subroutine the random number generated may be supplied with a specific array valued seed, report on which seed is in use and report on how big an array is needed to hold a seed. Not necessary to know any more at this stage.
- Using the same seed on separate executions will generate the same sequence of random numbers.

84: 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 (n) then the result has shape (m) ;

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

Guide for slide: 84

DOT_PRODUCT:

- is equivalent to

$$\text{DP} = \text{SUM}(\text{A} * \text{B})$$

or for COMPLEX arguments

$$\text{DP} = \text{SUM}(\text{CONJG}(\text{A}) * \text{B})$$

- is the mathematical dot product of two vectors,
- VEC1, VEC2 must conform in size (and must be 1D),
- don't get this intrinsic confused with DPRD the DOUBLE PRECISION product function or PRODUCT the intra-matrix product (see later).
- For LOGICAL arrays, the operation is a logical .AND.,

$$\text{DP} = \text{LA}(1) \text{.AND.} \text{LB}(1) \text{.OR.} \text{LA}(2) \text{.AND.} \text{LB}(2) \text{.OR.} \dots$$

and is equivalent to

$$\text{DP} = \text{ANY}(\text{LA} \text{.AND.} \text{LB})$$

MATMUL:

- is the mathematical matrix multiply intrinsic, it is not MAT1*MAT2, it is equivalent to setting element (i,j) of the result to:

$$\text{SUM}(\text{MAT1}(i,:) * \text{MAT2}(:,j))$$

- arrays must match in the specified dimensions (as you would expect); they do not have to conform,
- arrays can only be of rank 1 or rank 2. There must be one of each. i.e., if MAT1 is 1D then MAT2 must be 2D, and vice-versa.
- 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 operation is a logical .AND.: and is equivalent to (i,j) of the result to:

$$\text{ANY}(\text{MAT1}(i,:) \text{.AND.} \text{MAT2}(:,j))$$

85: 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 (X)

7	9	-2	4	8	10	2	7	10	2	1
---	---	----	---	---	----	---	---	----	---	---



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

Guide for slide: 85

- the result of these intrinsics is either the maximum (or minimum) value from the argument list or a conformable array filled with the maximum (or minimum) values from the correspond locations of the source arrays, there can be any number of source objects (arrays or scalars).

86: 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,

MAXLOC(X) = (/6/)

7	9	-2	4	8	10	2	7	10	2	1
---	---	----	---	---	----	---	---	----	---	---



A 2D example. If

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

then

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

Guide for slide: 86

- the result of these intrinsics is a 1D array of the coordinates of the maximum or minimum value,
- if the MASK is present the only elements considered are those corresponding to where the mask is .TRUE.,
- the result is **always** array valued even if it is only one number,
- for the first example, MAXLOC(ARRAY,ARRAY.LE.7), only the following elements are considered

```
ARRAY =  0  -1   1   6   -4  
         1  -2   5   4   -3  
         3    .   3  -7   0
```

The largest is element A(1,4)

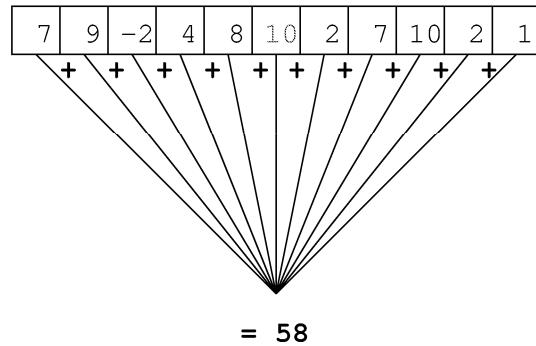
- second example -7 is smallest value which is element A(3,4),
- the third example is finding the location which holds the largest element in the array (/1,4/).

87: 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:

$$\text{SUM}(\mathbf{W}) = 58$$



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/)

Guide for slide: 87

88: Array Reduction Intrinsics (Cont'd)

These functions operate on arrays and produce a result with less dimensions than 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 $n - 1$.

Reduction functions are aptly named because an array is operated upon and a result obtained which has a smaller rank than the original array. If DIM is absent or the source array is of rank 1 then the result is scalar, otherwise the result is of rank $n - 1$.

ALL(MASK[,DIM])

- ◊ .TRUE. if *all* values of the mask are .TRUE. along dimension DIM, the result is an array of rank $n - 1$. For example, consider a 2D array, if DIM=2 then the function returns a 1D vector with the result being as if the ALL function has been applied to each column in turn. If DIM=1 the result is as if the ALL function had been applied to each row in turn.
- ◊ .TRUE. if array is zero sized,
- ◊ if DIM is absent then the whole array is considered.

ANY(MASK[,DIM])

- ◊ .TRUE. if *any* values are .TRUE. along dimension DIM, the result is an array of rank $n - 1$
- ◊ .FALSE. if array is zero sized,
- ◊ if DIM is absent then the whole array is considered.

COUNT(MASK[,DIM])

- ◊ number of .TRUE. elements in an array along dimension DIM, the result is an array of rank $n - 1$,
- ◊ zero if array is zero sized,
- ◊ if DIM is absent then the whole array is considered.

MAXVAL(SOURCE[,DIM][,MASK])

- ◊ maximum Value in an array
- ◊ if DIM is specified the result is an array of rank $n - 1$ of maximum values in other dimensions
- ◊ if MASK is present then the survey is only performed on elements of SOURCE which correspond to .TRUE. elements of MASK
- ◊ if DIM is absent the whole array is considered and the result is a single scalar.
- ◊ the largest negative number of the appropriate kind is returned if the array is zero sized.

MINVAL(SOURCE[,DIM][,MASK])

- ◊ minimum value in an array (in an optionally specified dimension under an optional mask);
- ◊ if DIM is specified the result is an array of rank $n - 1$ of minimum values in other dimensions
- ◊ if MASK is present then the survey is only performed on elements of SOURCE which correspond to .TRUE. elements of MASK
- ◊ if DIM is absent the whole array is considered and the result is a single scalar.

- ◊ the smallest positive number of the appropriate kind is returned if the array is zero sized.
- **PRODUCT(SOURCE[,DIM][,MASK])**
 - ◊ product of array elements
 - ◊ if DIM is specified the result is an array of rank $n - 1$ of the product
 - ◊ if MASK is present then the product only involves elements of SOURCE which correspond to .TRUE. elements of MASK
 - ◊ if DIM is absent the whole array is considered and the result is a single scalar.
 - ◊ if the array is zero sized then the product is 1
- **SUM(SOURCE[,DIM][,MASK])**
 - ◊ sum of array elements
 - ◊ if DIM is specified the result is an array of rank $n - 1$ of the sum
 - ◊ if MASK is present then the sum only involves elements of SOURCE which correspond to .TRUE. elements of MASK
 - ◊ if DIM is absent the whole array is considered and the result is a single scalar.
 - ◊ if the array is zero sized then the sum is 0

Lecture 4:
Program Units and Interfaces

90: 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).

Guide for slide: 90

- execution always begins in the main PROGRAM.
- SUBROUTINE and FUNCTION calls may be made in the main program. A call transfers control temporarily to a parameterised named sequence of code which performs a specific task — a procedure. When this procedure has finished control is passed back to the line following the invocation.
- MODULE program units are new to Fortran 90 and provide the functionality to replace the need for things like COMMON, INCLUDE, BLOCK DATA and add a (limited) ‘object oriented’ aspect to the language.
- Modules are probably the most important new feature of Fortran and it will be shown why their use is so important.
- SUBROUTINE and FUNCTION are also known as procedures.
- procedures should be as flexible as possible to allow for software reuse.

91: 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> ] ]
```

Guide for slide: 91

- This is the only compulsory program unit.
- the PROGRAM statement is optional however it is good policy to always use one. It can be any valid Fortran 90 name.
- the <*main program name*> is optional however it is good policy to always use one,
- the main program contains declarations and executable statements and may also contain internal procedures. The procedures are separated from the rest of the main program by a CONTAINS statement. These procedures may only be called from within the surrounding program unit — they automatically have access to all the host program unit's declarations but may also override them. These internal procedures may not contain further internal procedures, in other words the nesting level is a maximum of 1. The diagram shows two internal procedures, Sub1 and Funkyn indicating that there may be any number of internal procedures (subroutines or functions). They are wholly contained within the main program.
- the main program may also contain calls to EXTERNAL procedures. The diagram represents two external procedures, ExtSub1 and ExtFunn indicating that there may be any number of external procedures (subroutines or functions). External procedures are totally outside of the main program. This will be discussed later.
- the main program must contain, as its last statement, an END. For neatness sake this should really be suffixed by PROGRAM (ie so it reads END PROGRAM) and should also have the name of the program unit attached too. Using as descriptive as possible END statements helps to reduce confusion.

92: 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
```

Guide for slide: 92

- The example demonstrates a main program which calls an intrinsic function, (FLOOR), and an internal procedure, (Negative)
- Although not totally necessary, the intrinsic procedure is declared in an INTRINSIC statement (the type is not needed — the compiler knows the types of all intrinsic functions).
- The internal procedure is ‘contained within’ the main program so does not require declaring in the main program.

93: 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!

Guide for slide: 93

- a procedure is a block of code that performs a particular task. Procedures should generally be used if a task has to be performed two or more times, this will cut down on code duplication.
- it is generally accepted that procedures should be no more than 50 lines long in order to keep the control structure simple and to keep the number of program paths to a minimum,
- procedures should be as flexible as possible to allow for software reuse. Try to pass as many of the variable entities referenced in a procedure as actual arguments, do not rely too heavily on global data or host-association (see later).
- try to give procedures useful names and initial descriptive comments,
- there is absolutely NO POINT in reinventing the wheel - if a procedure or collection of procedures already exist then it is invariably a good idea to use them. (There are over 100 intrinsic procedures and a very very large amount of existing libraries,)
 - ◊ the NAg library deals with solving numerical problems and is ideal for engineers and scientists. The NAg Fortran 90 Mk I library f190 has just been released as a successor to the well respected and popular FORTRAN 77 library.
 - ◊ GKS is an international graphics standard - an implementation exists in FORTRAN 77 which should generally work with Fortran 90 programs (the depends on the compiler / linker).
 - ◊ other libraries include:
 - BLAS, Basic Linear Algebra Subroutines, for doing vector, matrix-vector and matrix-matrix calculations. Should *always* use these.
 - IMSL, akin to NAg Library
 - LaPACK, linear algebra package
 - Uniras, graphics routines, very comprehensive
 - ◊ there is an auxiliary Fortran 90 standard known as the “Varying String” module. This is to be added to the Fortran 90 standard and will allow users to define and use objects of type VARYING_STRING where CHARACTER objects would normally be used. The Standard has already been realised in a module (by Lawrie Schonfelder at Liverpool University). All the intrinsic operations and functions for character variables have be overloaded (see later for operator overloading) so that VARYING_STRING objects can be used in more or less the same way as other intrinsic types.
 - ◊ It is envisaged that many modules will be implemented which will encompass a large functionality. Due to the portable nature of Fortran 90 these modules and their associated procedures will be widely available and will result in grater software reuse. (One of the goals of Object Oriented programming.)

94: 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.

Guide for slide: 94

The subroutine here simply prints out its argument. Internal procedures can ‘see’ all variables declared in the main program. If an internal procedure declares a variable which has the same name as a variable from the main program then this supersedes the variable from the outer scope for the length of the procedure.

Using a procedure here allows the output format to be changed easily. To alter the format of all outputs, it is only necessary to change on line within the procedure.

95: 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.

Guide for slide: 95

Functions operate on the same principle to SUBROUTINEs, the only difference being that a function returns a value. In the example, the line

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

will substitute the value returned by the function for $F(a,b)$, in other words, the value of $\sqrt{a^2 + b^2}$.

96: 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.

Guide for slide: 96

- The parenthesised names following a function or subroutine name are call arguments. Procedures may have any number of such arguments.
- arguments at a call site are actual arguments (they use the actual name) and arguments in the procedure declaration are dummy arguments as their name is a substitute for the real name. A reference to a dummy argument is really a reference to its corresponding actual argument, for example, changing the value of a dummy argument actually changes the value of the actual argument.
- arguments must correspond in type, kind and rank. [FORTRAN 77 programs which flouted this requirement were not standard conforming but there was no way for the compiler to check.]

97: 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 known 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 program's memory between calls.

x will probably have a different size and shape on each call.
The space usually comes from the program's stack.

Guide for slide: 97

When a procedure is called, any local objects are brought into existence for the duration of the call. Thus if an object is assigned to on one call, the next time the program unit is invoked a totally different instance of that object is created with no knowledge of what happened during the last procedure call.

98: 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.

Guide for slide: 98

99: 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>] ]
```

Guide for slide: 99

- the structure is similar to that of the main PROGRAM unit except a SUBROUTINE can be parameterised (with arguments) and these must be declared in the declarations section.
- a SUBROUTINE may include calls to other program units either internal, external or visible by USE association (i.e., in a module — see later).
- in order to promote optimisation a recursive procedure must be specified as such, i.e., it must have the RECURSIVE keyword at the beginning of the subroutine declaration.
- if a SUBROUTINE has no arguments (parameters) then the parenthesis in the SUBROUTINE statement are optional (unlike a FUNCTION).
- as with the main PROGRAM the SUBROUTINE must terminate with an END statement and it is jolly good show to append SUBROUTINE and the name of the routine to this line as well.
- SUBROUTINES may contain internal procedures but only if they themselves are *not* already internal.

100: 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
```

Guide for slide: 100

The principle is the same as for calling an internal procedure except for two main things

- whereas an internal procedure has access to the host's declarations (and so inherits, amongst other things, the **IMPLICIT NONE**) external procedures do not. An **IMPLICIT NONE** is needed in every external procedure.
- the external procedure should be declared in an **EXTERNAL** statement. (This is optional but is good practise.)

101: 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.

Guide for slide: 101

- A **FUNCTION** is very similar to a **SUBROUTINE** except that it has a type, <*prefix*>. The type can either be declared in the declarations area of the code or by preceding the function name (with no arguments or parentheses) by a type specifier. It is a matter of personal taste which method is adopted, the two syntax specifications detail both cases.
- **RECURSIVE** functions must be explicitly defined.
- owing to the possibility of confusion between a variable name and a function reference the parentheses are **not** optional.
- the function name <*procname*> is the result variable.
- the function must contain a statement which assigns a value to the function name — a routine without one is an error.
- use the full form of the **END** statement.
- functions can return results of any type (including user defined types and pointers), kind and rank.

102: 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
```

Guide for slide: 102

- the dummy and actual arguments must correspond in number, type and kind (as with SUBROUTINES).
- can have functions which have arguments that are procedures.
- can have function that have no arguments. An empty set of brackets must be supplied.
- if the same function is called twice in the same statement problems may occur — a function call should not alter its arguments or any global variables.

103: 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).

Guide for slide: 103

New to Fortran 90,

- INTENT(IN)- value on entry equals value on exit - variable is not assigned to, (cf. constant).
- INTENT(OUT)- value on entry is undefined (or not used until assigned to). Must be assigned to.
- INTENT(INOUT)- object can be both used and assigned to.

The example,

- `arg1` has INTENT(IN) so it is simply used and not assigned to within the procedure.
- `arg2` has INTENT(OUT) so it is assigned to at least once inside the procedure.
- `arg3` has INTENT(INOUT) so it is first used and then reassigned to inside the procedure.
- INTENT not essential but it aids optimisation - can pass INTENT(IN) objects by value.
- also aids safety - if an INTENT(IN) object is assigned to an error will be generated. If an INTENT(OUT) object is not assigned to an error will also be generated.

104: 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;

Guide for slide: 104

- an example of a block structured language is Pascal. Here the scope of a variable can be limited to a block construct (such as a loop) by making the declaration inside the block.
- a program unit extends for its inception to termination by a corresponding END statement;
- internal procedures can access all the objects of their containing outer program units. Internal procedures may have parameters and any actual parameters must not be assigned to. The internal procedures may declare new (local) objects with the same names as objects from the calling program unit - in this case the outer objects will become inaccessible from within the internal procedure.
- USE association attaches a module to the program unit and allows all objects with public visibility to be accessed.
- global data should be provided by a module. See later.

105: 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*.

Guide for slide: 105

Internal procedures are new to Fortran 90:

- PrintPay** and **NewTax** have access to the declaration of **NumberCalcsDone** by *host association*. Variables passed to procedures as arguments are available by argument association.
- NewTax** cannot access any of the declarations of **PrintPay** and *vice-versa*.
- PrintPay** and **NewTax** could, if desired, invoke other internal procedures which are contained by the same program.
- Upon return from either procedure the value of **NumberCalcsDone** will have increased by one owing to the last line of the procedure.

106: 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.

Guide for slide: 106

- if an internal procedure declares an object with the same name as one in the outer (containing) procedure then this new variable supersedes the one from the outer scope for the duration of the procedure, for example, C in Sub.
- Sub accesses A (known locally as D) by argument association.
- Sub accesses B (and A) from Proggie. C is entirely local and supersedes C from Proggie for the duration of the call. When Sub is exited and control is returned to Proggie, the value that C had before the call to the subroutine is restored.
- Sub is forbidden to change the value or availability of A because it A associated with a dummy argument (P180 of Fortran 90 standard).

107: 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.

Guide for slide: 107

- The initialisation `NumInvocations = 0` is performed **once** conceptually at program start-up.
- the value of `NumInvocations` is retained between invocations of the procedure. The variable is said to be *static*.
- objects initialised in a procedure implicitly have the `SAVE` attribute, in the first example, the following is exactly equivalent:

```
INTEGER :: NumInvocations = 0
```

- giving the whole procedure the `SAVE` attribute means every local object (not dummy arguments) is placed in the static storage class.
- clearly, the `SAVE` attribute has no meaning in the main program

108: 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.

Guide for slide: 108

INTERFACES are new to Fortran 90,

- it is possible (often mandatory) to declare the explicit interface of an external procedure that is referenced in a particular program unit.
- it is generally a good idea to make all interfaces explicit.
- an interface declaration is initiated with the INTERFACE statement and terminated by END INTERFACE.
- the interface gives the characteristics of the dummy arguments, (for example, the name, type, kind, ranks and whether it is OPTIONAL; the INTENT should also be given but as this has not been covered it is omitted), and the characteristics of the procedure (i.e. name, class and type for functions). An interface cannot be used for a procedure specified in an EXTERNAL statement (and vice-versa).
- the USE statement is required in order that the kind of the REAL valued dummy arguments is known.
- an interface declaration is included once in the declarations part of the calling program unit.
- interfaces must match the procedure that they refer too.
- interfaces are only relevant for external procedures as internal procedures have an implicitly visible interface. Interfaces to intrinsic procedures are also implicit.

109: 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.

Guide for slide: 109

110: 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.

Guide for slide: 110

- using INTERFACE provides for better optimisation and type checking.
- INTERFACEs allow compilers to perform efficient separate compilation.

111: 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).

Guide for slide: 111

Explicit interfaces are mandatory if the procedure:

- has dummy arguments that are assumed-shape arrays, pointers or targets.

This is so the compiler can figure out what information needs to be passed to the procedure, for example, the rank, type and bounds of an array whose corresponding dummy argument is an assumed-shape array (see later), or the types and attributes of pointers or targets.

- has OPTIONAL arguments.

So the compiler knows the names of the arguments and can figure out the correct association when any of the optional arguments are missing.

- has an array valued or pointer result (functions).

Needs to know to pass back the function result in a different form from usual.

- has a result that has an inherited LEN=* length specifier (character functions).

The compiler needs to know to pass string length information to and from the procedure.

Explicit interfaces are mandatory if 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

113: 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.

Guide for slide: 113

- *explicit-shape* — all bounds specified. The actual argument that becomes associated with an explicit-shape dummy must conform in size and shape. No explicit INTERFACE is required. This approach is very inflexible very inflexible as only arrays of one fixed size can be passed. This form of declaration is best suited to non-dummy arguments.
- *assumed-shape* — no bounds specified, all inherited from the actual argument. An explicit interface must be provided. This approach should be used in preference to *assumed-size*. Arrays must match in type, kind and rank.
- an actual argument can be an ALLOCATABLE but a dummy argument cannot be - this means effectively that an ALLOCATABLE must be allocated before being used as an argument.

114: 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.

Guide for slide: 114

New to Fortran 90 — this is the recommended way to pass arrays as arguments to procedures:

- only the rank and type (and kind) need to be specified in the procedure, the exact shape is assumed.
- sections can be passed so long as they are regular, i.e., not defined by vector subscripts. This is to do with efficiency, as a vector subscripted section will be non-trivial to find in the memory.
- if an actual argument were an assumed-size array then the bound / extent information of the last dimension would not be known so this information could not be passed to a further procedure.
- the second section in the third call `Y(2:4,4)` has one dimension, the dummy only has 2.
- if the procedure is external then an `INTERFACE` block is needed to transfer type, kind, bound / extent information into the procedure.

115: 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.

Guide for slide: 115

- automatic arrays have local scope and a limited lifespan. They cannot have the **SAVE** attribute as they are created and destroyed with each invocation of the procedure. They are traditionally used for workspace.
- they take their size from dummy arguments:
 - ◊ either from scalar dummy arguments which are passed as arguments, A1,
 - ◊ from characteristics of other dummy arguments, A2,
 - ◊ or a combination of the above.

116: 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.

Guide for slide: 116

- **SAVE** gives **X** the static storage class which means that it persists between procedure calls.
- objects in **COMMON** cannot be given the **SAVE** attribute they already are static objects.

117: 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.

Guide for slide: 117

- the size of the result of the function is determined in a fashion which is similar to the way that automatic arrays are declared,
- the bounds of `funnie` are inherited from the actual argument and then used to determine the size of the result arrays.
- a fixed sized result could be returned by declaring the result to be an explicit-shape array,
- by default, functions are assumed to return a scalar result - an explicit interface is needed if they return anything else.
- the result could be determined from any of the other dummy arguments (just like automatic arrays).
- if the function is external then an interface is mandatory.

118: 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;

Guide for slide: 118

Modules are new to Fortran 90. They have a very wide range of applications and their use should be encouraged. They allow the user to write object based code which is generally accepted to be more reliable reusable and easier to write than regular code.

A **MODULE** is a program unit whose functionality can be exploited by any other program unit which attaches it (via the **USE** statement). A module can contain the following:

- global object declaration;

modules should be used in place of FORTRAN 77 **COMMON** and **INCLUDE** statements. If global data is required, for example, to cut down on argument passing, then objects declared in a module can be made visible wherever desired by simply using the module. The objects can be given a static storage class so will retain their values between uses. (Derived types should be declared in a module — see later.)

Sets of variable declarations which are not globally visible can also be made in a module and used at various places in the code which achieves the functionality of the **INCLUDE** statement.

- interface declaration;

It is sometimes advantageous to package **INTERFACE** definitions into a module and then use the module whenever an explicit interface is needed. This should be done in conjunction with the **ONLY** clause. Only really useful when module procedures cannot be used.

- procedure declaration;

Can put any procedures in a module which will be visible to any program unit which uses the module. This has the advantage that all the interfaces are already explicit within the module, so there is no need for any **INTERFACE** blocks to be written.

We only discuss the above here, the following topics will be discussed later:

- semantic extension,
- controlled object accessibility,
- packaging derived types,
- defining and overloading operators and procedures,
- defining generic interfaces to provide object oriented capabilities.

119: 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> ] ]

```

Guide for slide: 119

- <*module name*> is the name that appears in the USE statement, it is not (but can be) the filename.
- partial-inheritance is possible. Basically this means that one module can inherit the environment of another module either fully or partially by using a USE or USE .. ONLY clause.
- declarations and specifications include:
 - ◊ user-defined type definitions,
 - ◊ USE statements,
 - ◊ object definitions, (anything that could appear in a main program can be found here),
 - ◊ object initialisations
 - ◊ accessibility statements (see later),
 - ◊ interface declarations (of external procedures);
- the CONTAINS block houses module procedures:
 - ◊ these procedures are accessible when the module is used,
 - ◊ each module procedure may also contain internal procedures,
 - ◊ the module procedure are written in exactly the same way as regular (external) procedures.
- Entities of other modules can be accessed by a USE statement, *use association*, as the first statement in the specification part.
- Non-circular dependent definition chains are allowed (one module USEs another which USEs another and so on) providing a partial-inheritance mechanism.

120: 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
```

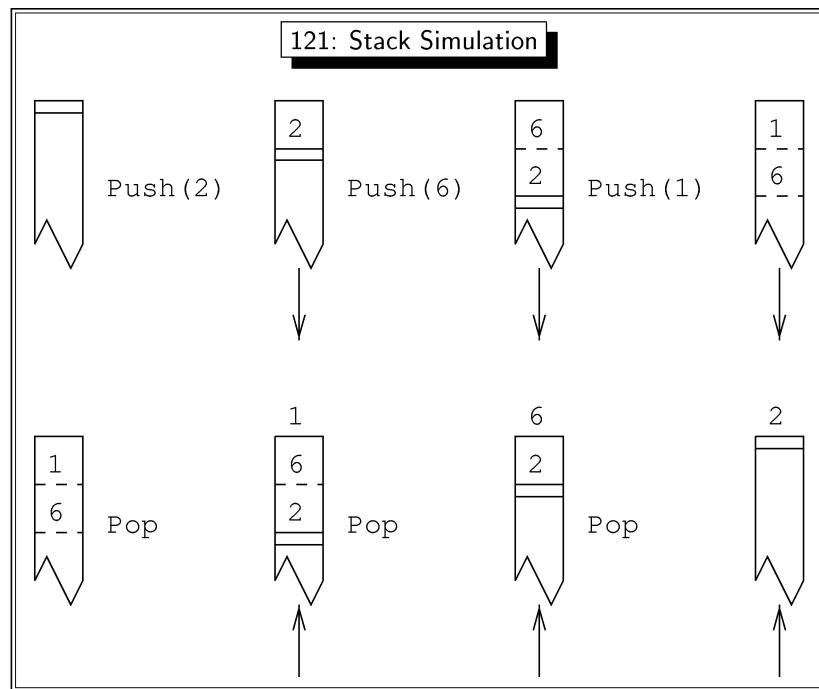
MODULEs should be placed *before* the program.

Guide for slide: 120

- in FORTRAN 77 **COMMON** was the only way to achieve global storage.
- **COMMON** is generally regarded as an unsafe feature because:
 - ◊ a **COMMON** block names a specific area of memory and splits it up into areas that can be referred to as scalar or array objects.
 - ◊ for a single **COMMON** block it is possible to specify a different partition of data objects each time the block is accessed.
 - ◊ each partition can be referred to by any type, in other words the storage can be automatically retyped and repartitioned with gay abandon - this leads to all sorts of insecurities.

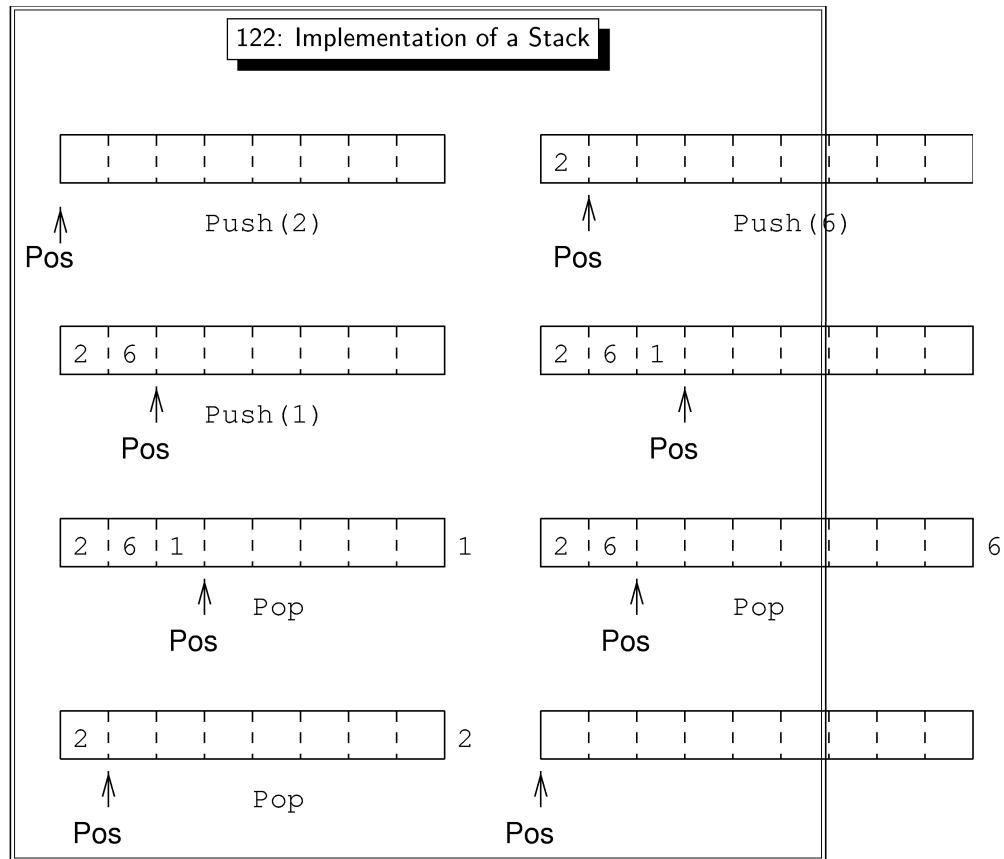
If the use of a **COMMON** block is absolutely essential (for example, when dealing with FORTRAN 77 programs) then putting a single instance of the block in a **MODULE** will decrease the possibility of the above type of misuse. This will mean the same **COMMON** block layout is used every time.

- modules provide a neater alternative method of implementing global data.
- declaring static objects, (objects with the **SAVE** attribute,) in a module can replace the need for **COMMON**.
- wherever the **MODULE** is used, the global objects are visible.
- name clashes between user and module entities can be avoided by using the object renaming facility.
- if the objects are not static then it must be ensured that at least one use of the module is always in operation - this usually means using it in the main program. If the objects are not static and the module is only used in two separate subroutines that are called directly by the main program then the data will disappear after the first use goes out of scope and be redeclared when the second is entered. (This can be useful for declaring an identical set of local objects in a number of independent procedures.)
- in the example the module **Pye** contains global data (the constant **Pi**). Its value can be accessed by **USEing** the module.



Guide for slide: 121

- A stack can be thought of as a pile of things that can only be accessed from the top. You can put something on the top of the pile: a Push action, and you can grab things off the top of the pile: a Pop action.
- The diagrams show the conventional visualisation of a stack as stuff is piled on the top the previous top elements get pushed down, as things are popped off the top the lower stack entries move towards the top.



Guide for slide: 122

- The diagrams show how we will simulate the stack using an array with a pointer to indicate where the top of the stack is. As elements are pushed onto the stack they are stored in the array and the ‘top’ position moves right; as things are popped off, the top of the stack moves left. This we need three things to simulate a stack
 - ◊ a current position marker,
 - ◊ the maximum size of the stack (array)
 - ◊ the array to hold the data
- This data is needed by both Push and Pop, it should be global.

123: 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
```

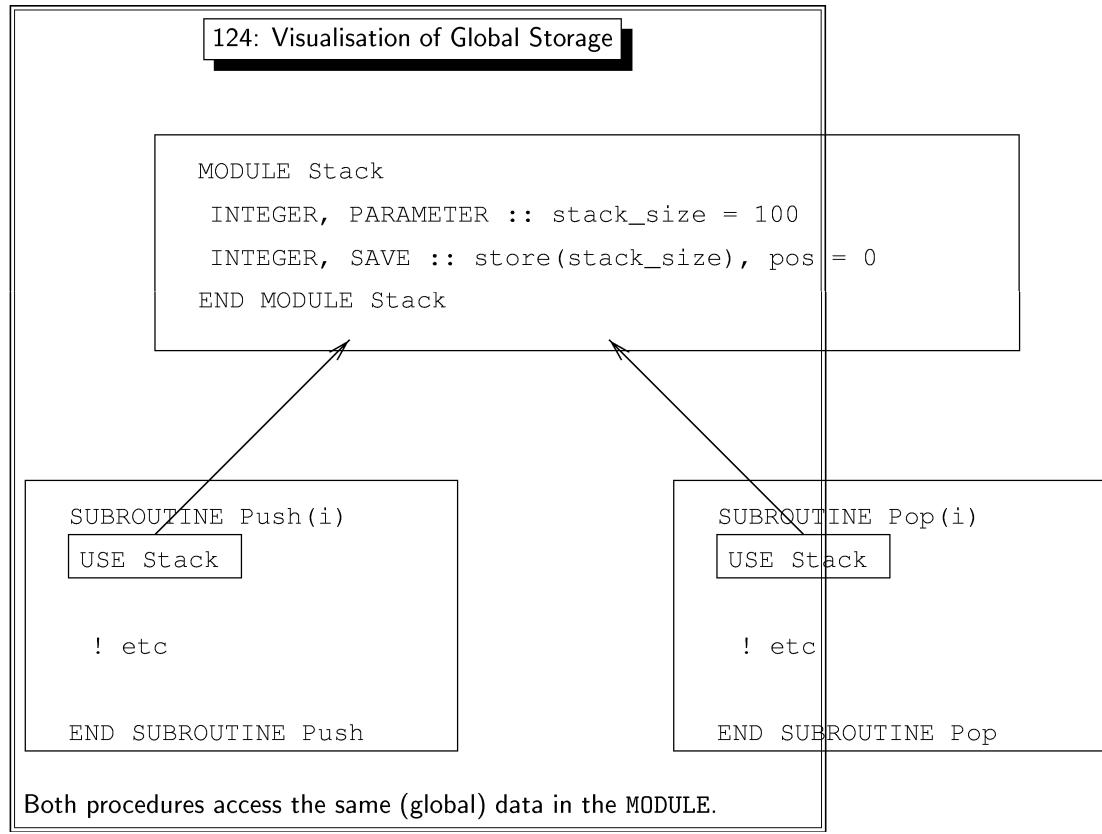
and two access functions,

```
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.

Guide for slide: 123

- if `stack_size` and `pos` did not have the `SAVE` attribute and the module was not constantly in use in at least one other program unit then these objects would not be persistent. It is necessary to either `USE` the module in the main program or else give global data the `SAVE` attribute. The latter is the safest!!
- the `PARAMETER` allows the stack size to be easily modified and does not need the `SAVE` attribute (because its a constant),
- the `push` and `pop` procedures can be called with one argument — this argument will either be the element to put on or take off the stack.
- the current state of the stack, i.e., the number of elements on the stack and the values of each location, are stored as global data,
- any program that `USES stack` has access to the global stack with whatever data is currently stored there.
- if the stack limit is exceeded the program will terminate with an error. You would probably not do this in practise!
- this is **not** the best way of simulating a stack but simply demonstrates global data. (See later for an improvement.)



Guide for slide: 124

The diagram indicates how the module sits in the memory separated from the subroutines. Both subroutines USE the module, so both have access to it which means that both can read and write to it. This removes the need for communicating the current state of stack the via argument lists.

125: 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.

Guide for slide: 125

- when the module is used the interfaces are available to the compiler which can only aid optimisation,
- this method means that it is only necessary to type the interface once (in the module) thus almost eliminating the scope for typos and mismatches.
- can collect all interfaces together which is a handy reference during coding,

126: 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.

Guide for slide: 126

- the module contains all the interfaces.
- the module merely needs to be used to make interfaces explicit, however, this is **not** the best way to use modules, we should use *encapsulation* (see later).

127: 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.

Guide for slide: 127

- placing interface definitions in a module is OK if all procedures are independent (don't call each other) but this is not usually the case,
- in the previous example, if `sub1` wanted to call `sub2` it would need to have an explicit interface. This could be achieved by inserting the statement, `USE my_interfaces`, into `sub1`, however, this causes a problem as `sub1` will have its own interface contained within its body which is not legal (duplicate declarations). (A `USE ONLY` clause would have to be used to skirt around this problem but this solution is too messy.)
- if we encapsulate the procedures into the module then the need for INTERFACE blocks is removed. The fact that the procedures are defined *in* the module means that their interfaces can be seen by any program unit that uses the module. These procedures are now called *module procedures* and can call each other with out any extra declarations.
- encapsulation removes the problems of duplicity.
- this method of encapsulation should be employed whenever feasible.

128: 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.

Guide for slide: 128

- sub1** and **sub2** are now called module procedures,
- module procedures can call other module procedures at the same level **sub1** can call **sub2** or vice versa.
- module procedures can contain one level of internal procedures.
- name clashes can be removed by using the renaming facility
- the module could be compiled separately.
- the module can still contain global data (as indicated)

129: 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.

Guide for slide: 129

- everything to do with the stack is now defined in one place — USE `stack` provides all declarations, access functions and storage to implement a simple integer stack.

130: 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.

Guide for slide: 130

- this renaming facility is essential otherwise user programs would often require rewriting owing to name clashes, in this way the renaming can be done by the compiler.
- renaming should not be used unless absolutely necessary as it can add a certain amount of confusion to the program,
- as many renames as desired can be made in one USE statement, they are just supplied as a comma separated list.
- can only rename the same object once per program unit.

131: 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.

Guide for slide: 131

- can mix renaming and use-only in one USE statement.
- it is possible to restrict the availability of objects declared in a module made visible use association by using the ONLY clause of the USE statement,
- this could be used as an alternative to renaming module entities, a situation may arise where a user wishes to use 1 or 2 objects from a module but discovers that if the module is accessed by use association there are a couple of hundred name clashes. The simplest solution to this is to only allow the 1 or 2 objects to be seen by his / her program. These 1 or 2 objects can be renamed if desired.
- using the ONLY clause gives the compiler the option of just including the object code associated with the entities specified in the ONLY clause instead of the code for the whole module. This has the potential to make the executable code smaller and faster.
- the USE statement with its renaming and restrictive access facilities gives great control enabling two modules with similar names to be accessed by the same program with only minor inconvenience.

132: 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,

<i>s-1</i>	3 2 1 0	
	0 .. 0 0 0 0	value = 0

<i>s-1</i>	3 2 1 0	
	0 .. 0 1 0 1	value = 5

<i>s-1</i>	3 2 1 0	
	0 0 1 1	value = 3

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

Guide for slide: 132

All variables used as bit arguments must be INTEGER valued. The model for bit representation is that of an unsigned integer so 0 would have a bit representation of all 0's, 1 would have all zeros except for a 1 in the last position ((position zero) 00...0001).

The number of bits in a single variable depends on the compiler - parameterised integers should be used to fix the number of bytes. BIT_SIZE gives the number of bits in a variable.

Assume that A has the value 5 (00...000101) and B the value 3 (00...000011) in the following:

- BTEST — bit value.

.TRUE. if the bit in position pos of INTEGER i is 1, .FALSE. otherwise, for example, BTEST(A,0) has value .TRUE. and BTEST(A,1) is .FALSE..

- IAND — bitwise .AND..

The two arguments are INTEGER the result is an INTEGER obtained by doing a logical .AND. on each bit of arguments, for example, IAND(A,B) is 1 (00...000001).

- IBCLR — bit clear.

Set bit in position pos to 0, for example, IBCLR(A,0) is 4 (00...000100).

- IBITS — extract sequence of bits.

IBITS(A,1,2) is 2 (10 in binary) or effectively 00...00010.

- IBSET — set bit

Set bit in position pos to 1, for example, IBSET(A,1) is 7 (00...000111).

- IEOR, IOR — bitwise .OR. (exclusive or inclusive).

For example, IEOR(A,B) is 6 (00...000110) and IOR(A,B) is 7 (00...000111).

- ISHFT, ISHFTC — logical and circular shift.

Shift the bits shft positions left (if shft is negative bit move right). ISHFT fills vacated positions by zeros ISHFTC wraps around. for example, ISHFT(A,1) is 00...001010 (10), ISHFT(A,-1) is 00...000010 (2). ISHFTC(A,1) is 00...001010 (10), ISHFTC(A,-1) is 10...000010 (-2147483646).

- NOT — compliment of whole word.

NOT(A) is 11...111010 (-6).

- MVBITS (SUBROUTINE) — copy a sequence of bits between objects.

For example MVBITS(A,1,2,B,0) says “move 2 bits beginning at position 1 in A to B position 0, this gives B the value 00...000010 (2).

133: 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.

Guide for slide: 133

- **MERGE(TSOURCE,FSOURCE,MASK)**— merge two arrays under a mask, whether to include TSOURCE or FSOURCE in the result depends on LOGICAL array MASK; where it is .TRUE. the element from TSOURCE is included otherwise the element from FSOURCE is used instead.
- **SPREAD(SOURCE,DIM,NCOPIES)**— replicates an array by adding NCOPIES along a dimension. The effect is analogous to taking a single page and replicating it to form a book with multiple copies of the same page. The result has one more dimension than the source array.
- **PACK(SOURCE,MASK[,VECTOR])**— pack array into a one-dimensional array under a mask, useful for compressing data.
- **UNPACK(VECTOR,MASK,FIELD)**— unpack a vector into an array under a mask, useful for uncompressing data.

134: 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,

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

and

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

and

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

we find

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

Guide for slide: 134

- RESHAPE puts the elements into the LHS arrays in array element order, so

```
TSOURCE = 1 5 9  
        3 7 11
```

and

```
FSOURCE = 0 4 8  
        2 6 10
```

- T and F defined as:

```
LOGICAL, PARAMETER :: T = .TRUE.  
LOGICAL, PARAMETER :: F = .FALSE.
```

so

```
MASK = T T F  
      F F T
```

thus we get from TSOURCE

```
1 5 .  
. . 11
```

and from FSOURCE

```
. . 8  
2 6 .
```

135: 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

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

and

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

Guide for slide: 135

- DIM= 2 (the first example) — this means we spread (/ 5, 7/) along dimension 2 (the direction of a row).
- DIM= 1 (the second example) — this means we spread (/ 5, 7/) along dimension 1 (the direction of a column).
- NCOPIES= 4 — which means we want 4 copies of the specified section,

136: 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

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

and

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

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/).

Guide for slide: 136

- If **VECTOR** is present, the result size is that of **VECTOR**; otherwise, the result size is the number t of **.TRUE.** elements in **MASK** unless **MASK** is scalar with the value **.TRUE.** in which case the result size is the size of **SOURCE**.
- Element i of the result is the element of **SOURCE** that corresponds to the i th **.TRUE.** element of **MASK**, (in array element order,) for $i = 1, \dots, t$.
- If **VECTOR** is present and has size $n > t$, element i of the result has the value **VECTOR(i)**, for $i = t + 1, \dots, n$.
- so for the first example, **VECTOR** is absent meaning that size of the result is the number of **.TRUE.** elements in **MASK**, i.e., 3. (This is with the previous **MASK**.)

The elements of **MASK** that are true are (1,1), (1,2), (2,3)

```
1   5   .
     .  11
```

giving the result (/1, 5, 11/).

- the second example, **VECTOR** is present so the result is of size 4. The first t elements of the results are as before corresponding to the **.TRUE.** elements of the mask. The remaining values are taken from **VECTOR** the 4th value of the result is the $t + 1$ th element of **VECTOR**, 6. Hence the result (/1, 5, 11, 6/).
- the third example, the mask is scalar so the first 6 elements are **A** and the rest from the end of **VECTOR**.

137: UNPACK Intrinsic

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

If

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

and

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

then

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

and

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

Guide for slide: 137

- The element of the result that correspond to the i th .TRUE. element of the mask (in array element order) has the value VECTOR for $i = 1, \dots, t$ where t is the number of .TRUE. elements of the mask, each other element has a value equal to the corresponding element of FIELD
- for the first example, the portions of the result originating from VECTOR and corresponding to the .TRUE. values of MASK are

$$\begin{matrix} 6 & 5 & . \\ . & . & 4 \end{matrix}$$

and the rest of this is filled in by elements of FIELD

$$\begin{matrix} . & . & 1 \\ 7 & 7 & . \end{matrix}$$

- for the second example, the portions of the result corresponding to the .TRUE. values of MASK are

$$\begin{matrix} . & . & 1 \\ 3 & 2 & . \end{matrix}$$

and the rest of this is filled in by elements of field

$$\begin{matrix} 9 & 5 & . \\ . & . & 3 \end{matrix}$$

138: 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) /))
...
```

INTEGER	0 .. 0 1 0 1	B
REAL	0 .. 0 1 0 1	A
REAL 0 1 0 1	AA
COMPLEX	0 .. 0 1 0 1	C

Guide for slide: 138

Most languages have a facility to change the type of an area of storage, in FORTRAN 77 people used EQUIVALENCE. Fortran 90 adds different facility.

TRANSFER takes the bit pattern of the underlying representation and interprets it as a different type.

- SOURCE can be array or scalar valued of any type
- MOLD can be array or scalar valued of any type and specifies the result type (which is the same shape as SOURCE),
- in the above example there is a big difference between MOLD being array valued (/ 0.0 /) and scalar 0.0
 - ◊ the result of TRANSFER(B, (/ 0.0 /)) is an array, this is assigned element-by-element to A
 - ◊ the result of TRANSFER(B, 0.0) is a scalar, this scalar is assigned to every element of AA. (The first half of B(1) (4 bytes) is interpreted as a REAL number.)
- (/ (0.0,0.0) /) represents an array valued COMPLEX variable.

Lecture 6:
Data Parallelism

140: 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 $(\text{communication time}) / (\text{computation time})$,
- balance load over processors,

Guide for slide: 140

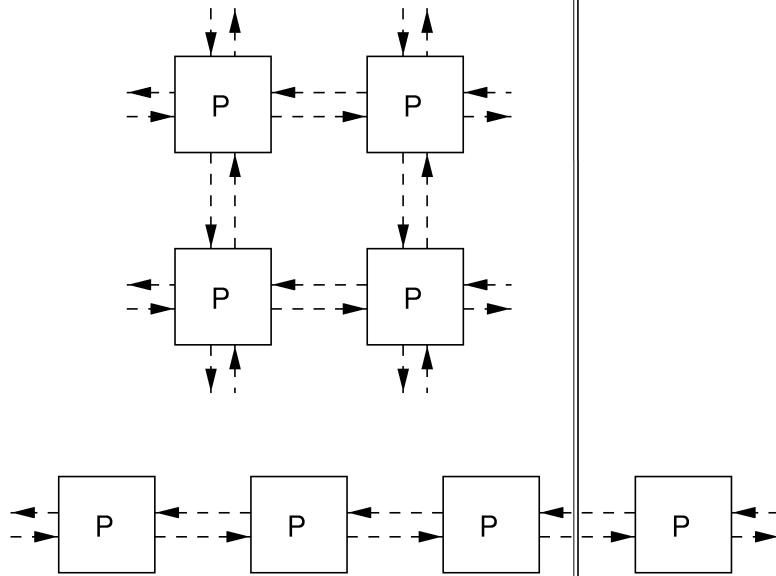
The slide gives the theory, however, parallelism is not guaranteed to work. Some algorithms do not parallelise well, others need to be rewritten in order to be suitable for parallel machines.

Relatively speaking, communication time is much more expensive than computation time. When a message is sent there is always a ‘start-up’ time (or delay) plus the time taken for the message to be delivered. Sometimes it may be advantageous to perform some extra computations in order to cut down the amount of communications.

Balancing the load ensures that there is little waste of resources due to processors sitting idle.

141: Processor Configurations

Processors can be interconnected in a number of ways:



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

Guide for slide: 141

A blank slide could be placed over this one and arrays drawn to overlay the processor grids. It can then be pointed out that in the 2D case P1 gets the top left corner of the matrix, P2 the top right and so on.

A more complex configuration is a hypercube. This is similar to a $2 \times 2 \times 2$ cube except it is constructed in much more than 3 dimensions.

142: Parallel Programs

Three main classes,

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

High Performance Fortran (HPF) concentrates on Data Parallelism.

Guide for slide: 142

- **Data Parallelism:** Such as HPF. Spread the data over a set of processors and execute same the program on each processor, this means that each processor operates on its own subset of the overall data.
- **Task Parallelism:** Two or more cooperating processes. Different tasks will be written to solve different parts of the problem. These tasks will pass data between themselves in pursuit of the solution.
- **Master-Slave:** One master process controls one or more slaves, it is responsible for the distribution of the data, general coordination of the solution methodology and collecting the results at the end.

143: 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.

Guide for slide: 143

The High Performance Fortran Forum (HPFF) is an informal group formed by interested parties from both hardware and software vendors and academic institutions from all over the world. Most of the ‘major players’ are involved, there are delegates from:

- Convex
- Cray Research
- DEC
- Fujitsu
- Hewlett Packard
- IBM
- Intel
- Meiko
- Sun MicroSystems
- Thinking Machines

The group first met at Supercomputing ’91, and had their first ‘solo’ meeting in Houston, Tx in Jan ’92 with 130 attendees. Their intention was to make swift progress (unlike x3j3/WG5, the Fortran Working Group!) and a draft standard was circulated a year after the groups inception at Supercomputing ’92. After a period of public comment the *de facto* standard (HPF v1.0) was officially distributed in May 1993. The whole standardisation process was unusual because most of the discussions were by Email.

MIMD stands for Multiple Instruction Multiple Data, for example, the Cray T3D.

SIMD stands for Single Instruction Multiple Data, for example, ICL DAP.

144: 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!

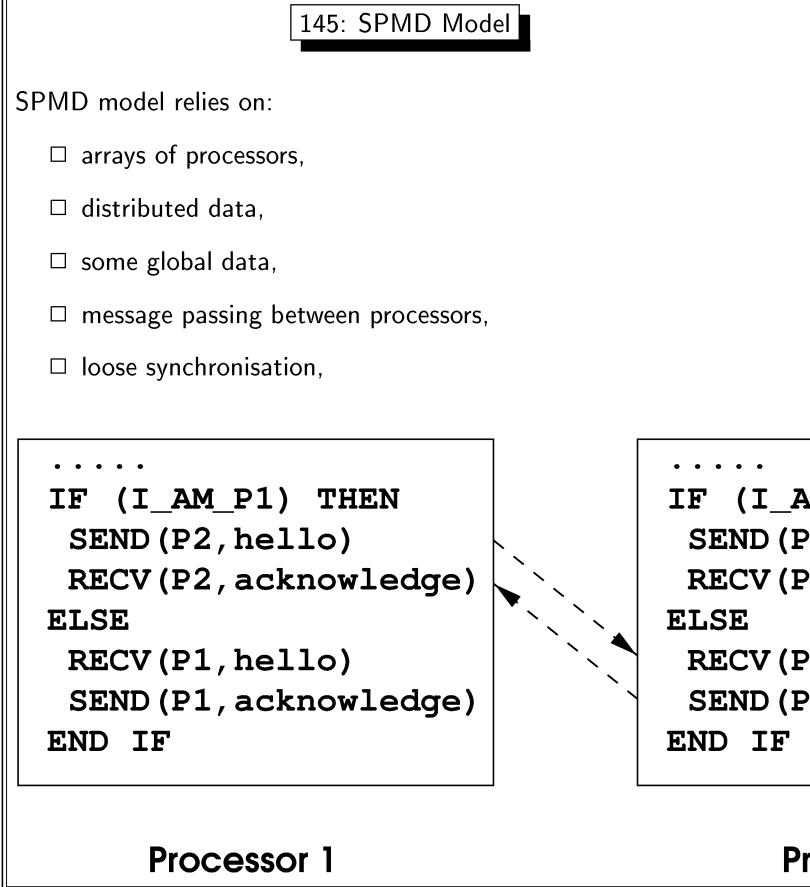
Guide for slide: 144

It has been said that explicit message-passing programs are to HPF as assembler programs are to high level languages. The analogy runs quite well as HPF programs are simple to write but will never quite achieve the efficiency of an explicit message-passing equivalent.

SPMD stands for Single Program Multiple Data.

The HPF directives specify the location (or layout) of the data and the parallel array syntax, intrinsic functions, HPF library calls and HPF FORALL or INDEPENDENT DO loops specify the parallelism.

A Fortran 90 program can be passed unaltered through an HPF compiler and will be parallelised using the system defaults. This will not be the most effective way to parallelise code but it in many cases will give a significant increase in speed.



Guide for slide: 145

Loose synchronisation means that it is occasionally necessary for processors to synchronise. The processors do not have to operate in lockstep!

146: 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
```

Guide for slide: 146

The examples of the message passing calls are pseudo code, most calls look a bit like this.

MPI is the new international standard and is expected to become much more prominent in the coming years. It is *very* similar to PVM but has a little extra functionality.

Parmacs was Europe's attempt at a message passing system. It was very successful and provided a lot of input to the MPI project.

PVM is currently (1996 Jun) the most popular.

147: 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),

Guide for slide: 147

The conceptual processor grids do not have to bear any relation to the underlying physical processor configurations — these are merely the way in which the HPF program views the system.

Each of the executable mechanisms in the list specify some form of parallelism. The compiler will spot when constructs such as these are being used and can then parallelise the code at this point.

- Fortran 90 contains array syntax and intrinsics to express parallelism
- **FORALL** is a generalised array assignment statement (and is in Fortran 95)
- **INDEPENDENT** loops are **D0** loops which have the property that the order in which the iterations are executed is unimportant. This means that they can all be executed at the same time.
- **PURE** procedures are ‘side-effect free’ procedures. They can be called in the body of **FORALL** and it is clear that one instance of a **PURE** procedure will not interfere with another.
- extrinsics are procedures written in a different language (ie non-HPF)
- The HPF library module and the new Intrinsics are comparable to the Fortran 90 intrinsics.

148: 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!

Guide for slide: 148

A Subset language was defined to provide a ‘fast-track’ for useful, yet basic, implementations; the full HPF language was recognised as being tricky to implement. Some of the Full HPF features have yet to be implemented by *any* of the vendors. Here we look at *most of* Full HPF!

The subset was originally designed to cover most of the common and useful Fortran 90 features. MODULEs were not deemed that important by HPFF but most vendors implemented them very early in the development cycles as users find them very powerful. Examples of Fortran 90 features absent from subset HPF are derived types, parameterised types and pointers. All but the latter are implemented by most compilers now.

149: 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.

Guide for slide: 149

The Full HPF (v1) specification includes some features that are much harder to implement and less useful than other simpler features.

We study the most useful HPF features and omits some of the more complex. Full HPF allows dynamic objects — ie, objects that can be remapped part-way thorough execution of a program unit — and inherited mappings (in a procedure). Both these features are *very* difficult to implement efficiently!

150: HPF Directives

Can give 'hints' to the compiler:

`!HPF$ <hpfdirective>`

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.

Guide for slide: 150

These directives are supplied to give hints to the compiler — the compiler is at liberty to ignore or modify any directives. These directives are Fortran 90 comments which means that they are ignored by Fortran 90 compilers — they do not change the semantics of the code at all.

There are two different declaration styles: the FORTRAN 77 style and the Fortran 90 style. It is recommended that the Fortran 90 style be used. There are one or two instances where the FORTRAN 77 style of declaration leads to ambiguous code; using the Fortran 90 declaration will alleviate the need to remember the specific cases that cannot be expressed in FORTRAN 77 syntax.

Most directives are concerned with data distribution - the **INDEPENDENT** directive (and its **NEW** clause) are the only executable directives. Most implementations currently (Jun '96) parse, but do not act upon, the **INDEPENDENT** directive.

A directive is continued in the same way as a Fortran 90 statement but the continued line must also begin with the directive origin (**!HPF\$**).

151: 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.

Guide for slide: 151

These declarations are in the Fortran 90 style. The HPF specification states that all processor grids in one program must contain the same number of processors. This is to give a handle on the available resources. Also, if two processors arrangements are the same shape (conformable) then the corresponding elements of each arrangement refer to the same physical processor.

Can also declare a scalar processor.

152: 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).

Guide for slide: 152

DISTRIBUTE tells the compiler how to divide the data between the processors. The idea is that the particular distribution methods are applied to the object to be distributed in each dimension.

There must be as many non-* distribution methods as there are dimensions of the processor grid.

The first distribute statement:

```
!HPF$ DISTRIBUTE (BLOCK) ONTO P1      :: A    !1-D
```

takes the matrix A divides it into equal sized contiguous blocks (except the last block which may be smaller than all the rest) and gives one block to each processor. A has 50 elements and P1 is of size 4, therefore each processor gets $\lceil \frac{50}{4} \rceil = 13$ elements, except the last, P1(4), which gets the remainder (11).

Cyclic distribution dishes the array elements out in much the same way as one would deal a pack of cards. Each processor gets one element in turn and when all processors have received one element, the first processor receives a second (and so on).

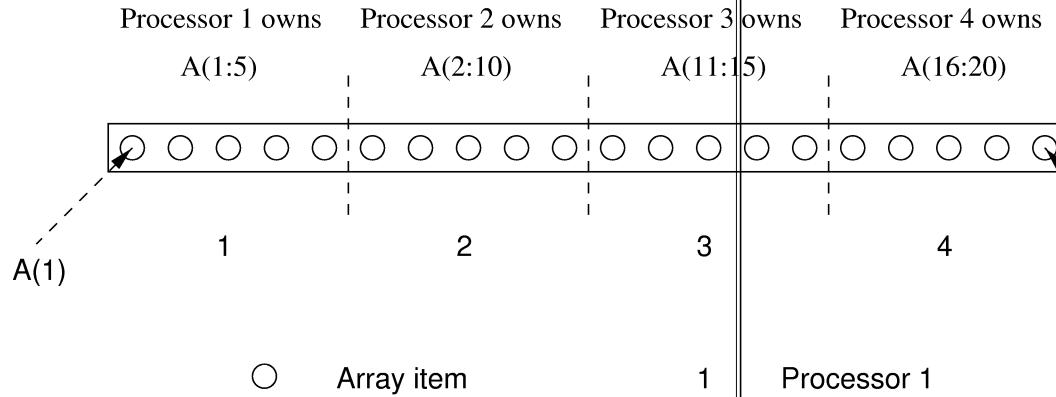
* distribution means that the dimension is ignored; this generally means that each processor gets the whole row (as in this case) or column.

The last directive in the block demonstrates the FORTRAN 77 style of declaration.

153: 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.

Guide for slide: 153

Here, the array has 20 elements and this is distributed blockwise onto a four element processor chain. This means that by using the formula: $\lceil \#A/\#P \rceil$, each processor gets a block of 5 elements. The first processor gets the first block, $a(1:5)$, the second gets elements $A(6:10)$ and so on.

154: Cyclic Distribution

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

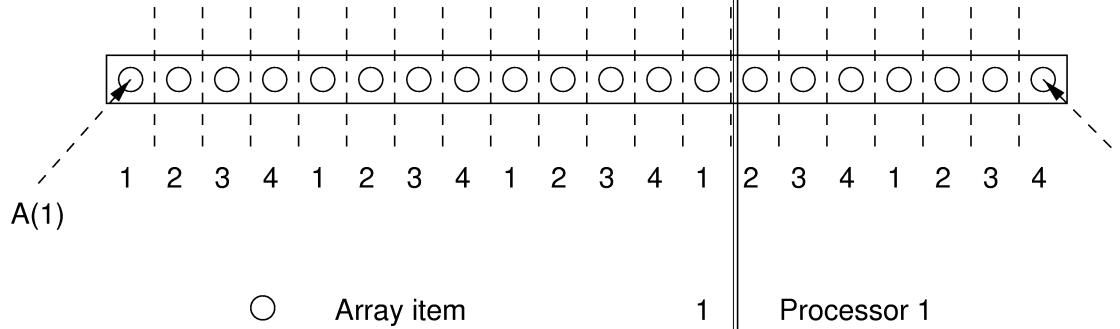
```
PROGRAM Round_Robin
    REAL, DIMENSION(20)      :: A
!HPF$ PROCESSORS, DIMENSION(4)  :: P
!HPF$ DISTRIBUTE (CYCLIC) ONTO P :: A
    ....
```

Processor 1 owns $A(1::4)$

Processor 2 owns $A(2::4)$

Processor 3 owns $A(3::4)$

Processor 4 owns $A(4::4)$



○ Array item

1

Processor 1

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.

Guide for slide: 154

As with the previous example, each processor gets five elements, but this time a different set of elements are received. The elements are dished out in a round-robin fashion starting from processor 1. The diagram indicates the elements that each processor is given.

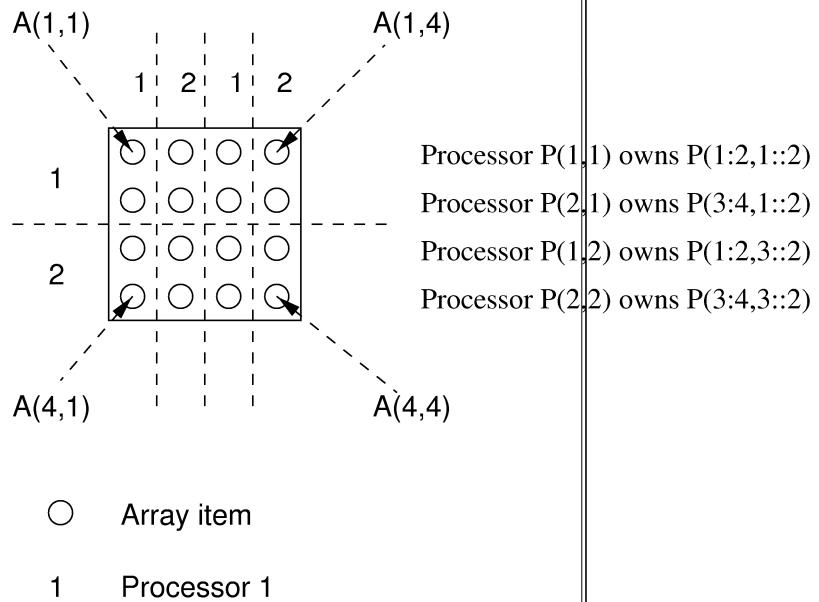
The formula for the maximum number of elements that a processor can receive is the same as for BLOCK distribution.

155: 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

```



Guide for slide: 155

2D distribution is simple. The example has **BLOCK** distribution in the first dimension (traversing the rows) and **CYCLIC** in the second dimension (traversing the columns).

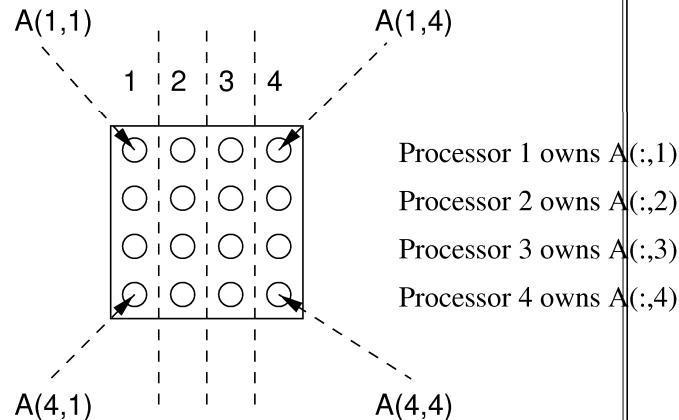
Thusly, the first dimension is partitioned into equal sized blocks and the second dimension is divided up in a round robin fashion. It transpires that:

- $P(1,1)$ owns $A(1:2,1:3:2)$,
- $P(2,1)$ owns $A(3:4,1:3:2)$,
- $P(1,2)$ owns $A(1:2,2:4:2)$,
- $P(2,2)$ owns $A(3:4,2:4:2)$.

156: 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.

Guide for slide: 156

Here, the first dimension is not distributed, the second dimension is distributed blockwise. This means that the second dimension is divided into equal sized blocks (the first dimension is not partitioned at all) and the sections are then distributed *onto a 1D processor grid*.

- $\mathbf{Q}(1)$ owns $\mathbf{A}(:,1)$,
- $\mathbf{Q}(2)$ owns $\mathbf{A}(:,2)$,
- $\mathbf{Q}(3)$ owns $\mathbf{A}(:,3)$,
- $\mathbf{Q}(4)$ owns $\mathbf{A}(:,4)$.

There must be the same number of non-* distributed dimensions as there are dimensions in the processor grid.

157: 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),

Guide for slide: 157

- if the `ONTO` clause is absent then objects are distributed onto the default grid (which is often supplied at run-time as a command line argument).
- `BLOCK` distribution is useful for neighbourhood calculations. Many algorithms are of this form.
- `CYCLIC` gives a good load balance. Each processor will have the same number of elements (give or take 1 element) so will perform the same number of calculations. As each processor receives a ‘random scattering’ of elements then the intensity of calculation should also be random meaning that every processor should have to do more or less the same amount of work.

Load balancing is important because at many points throughout the program there will be synchronisation points. Processors must wait until the last processor has finished its calculation before the next calculation can be started.

- It can generally be assumed that all scalar variables will be replicated on every processor. This means that the value of a particular scalar will be the same on every processor. The compiler will be responsible for keeping these values coherent except when control returns from an extrinsic. It is up to the user to ensure coherency in this case.

158: 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.

Guide for slide: 158

Distribution of allocatable arrays is based on exactly the same concepts as for distributing regular arrays except that the distribution is performed immediately after the allocation has been performed.

The array is effectively undistributed at deallocation.

159: 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.

Guide for slide: 159

This is a rule of thumb and is not always followed, for example, if all the RHS objects are co-distributed then, instead of all the RHS elements being sent to the owner of the LHS for computation, the computation of the result may take place on the home processor of the RHS elements and then be sent to the owner of the LHS for assignment. This would reduce the number of communications required.

It is up to the compiler to decide when to follow the owner-computes rule and when not to.

160: 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.

Guide for slide: 160

As mentioned earlier, the compiler will uphold the coherency of replicated scalars.

To achieve the effect of having a different scalar value on each processor, one must declare an array with as many elements as there are processors. This array can be distributed so that each processor receives just one element - this element can now be treated like a local scalar.

161: 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)
```

Guide for slide: 161

- A is involved in neighbourhood calculations. A(3) is combined with A(1) and assigned to A(2)
- B an 'odd shaped' inner section of B is used - in order to balance the load a CYCLIC distribution should be used.
- the calculation of an element of C involves a whole column of D. This means that every element of C should be aligned with the corresponding whole column of D.

162: Successive Over Relaxation Example

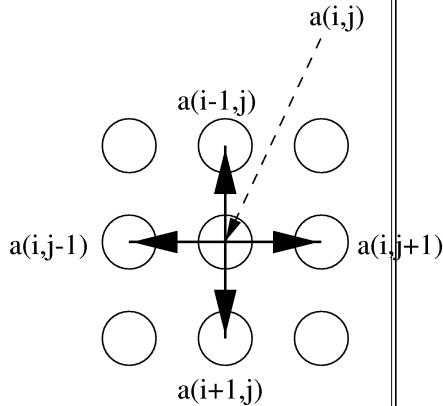
Successive over relaxation is used in many HPF kernels:

```

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.

Guide for slide: 162

This is a common technique in HPF programs. BLOCK distribution is perfect as most calculations will involve all local calculations. It is only the calculation of the border elements that require elements from other processors.

The diagram shows how one element accesses its four closest neighbours when being updated.

163: 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.

Guide for slide: 163

Care must be taken with replication. Many compilers assume that if an array is not mentioned in any distribution directive then it should be replicated, a copy of the *whole* array is given to each processor. If this is not what is required then performance will suffer greatly.

The second style of distribution (with the processor array missing) is useful during development work when the number of processors may differ between runs.

164: 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.

Guide for slide: 164

It is fairly easy to write a correct HPF program but it is an awful lot harder to write an efficient one. The technique is more or less

1. write a correct serial version of the intended HPF program, test and debug,
2. add distribution directives so as to generate the minimum number of communications in pursuit of the solution.

Data locality is the key - well chosen alignment and distribution strategies will enhance performance significantly.

Programmers should always add as many directives as possible: always add **INDEPENDENT** directives where correct, always align things explicitly and so on.

The most important factor is to obtain a good parallel algorithm, many serial algorithms are wholly inappropriate for HPF programs.

165: 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.

Guide for slide: 165

All the points on the slide will result in poor execution speeds.

- if there are complicated subscript expressions then the compiler may not be able to efficiently work out which elements are needed from other processors,
- vector subscripting is a very powerful technique which is not handled efficiently by the current compilers. Use with caution.
- sequential do loops may well be executed sequentially, only a very sophisticated compiler will be able to detect parallelism in such structures. This may mean that processor 1 performs its iterations of a loop, when it has finished it informs processor 2 of this fact which means that processor 2 can perform its iterations, when finished it will inform processor 3 and so on. If there are hundreds of processors then this kind of loop will absolutely *cripple* performance.
- remapping object can be performed across procedure boundaries, unnecessary remappings are *very* time consuming. Again, be careful.
- if the system is unevenly balanced then some processors will be constantly waiting for other processors to finish. This will also ruin performance.

166: 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](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](http://www.lpac.ac.uk/SEL-HPC/Materials/HPFlibrary/HPFlibrary)

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

Guide for slide: 166

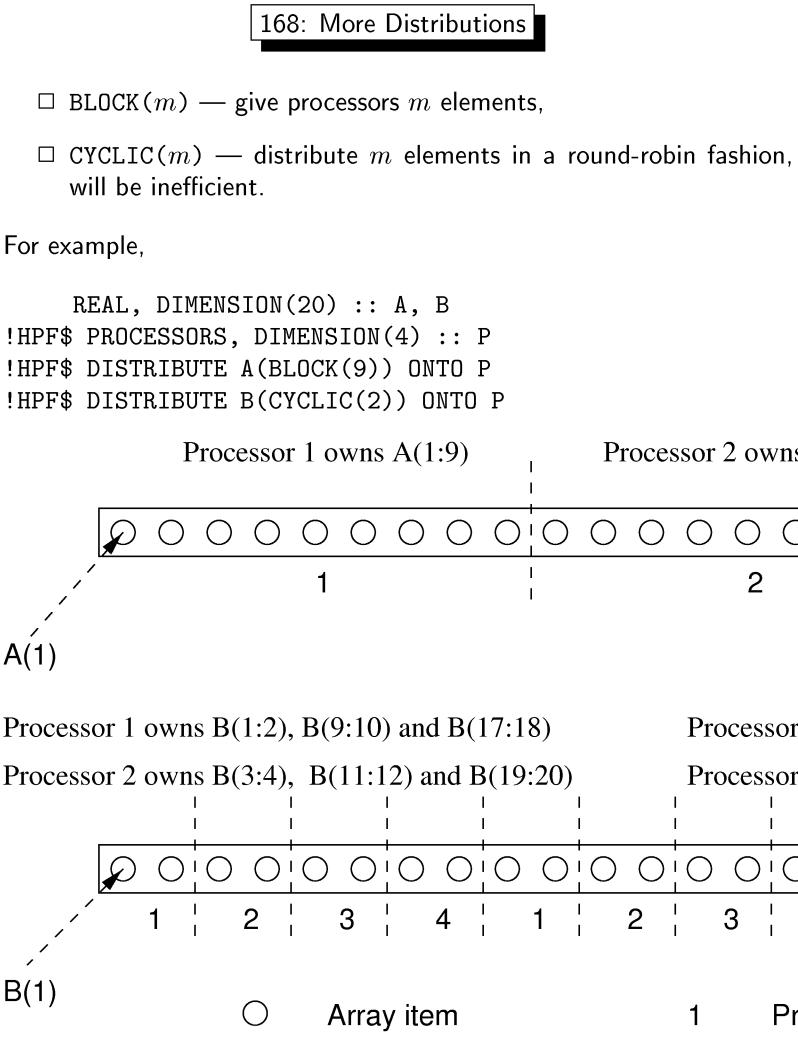
The Liverpool HPF page contains links to many useful places including the HPFF Home Page and the Edinburgh Parallel Computing Center's HPF Course notes which make a useful accompaniment to this course.

The HPFF Home page contains the HPF Specification.

The HPF Handbook is very informative but does contain material from the language Specification document.

A Fortran 90 version of the HPF Library module is available to allow users to develop HPF codes by first creating a working Fortran 90 Fortran 95 program.

Lecture 7:
Alignment and Distribution



Guide for slide: 168

These distributions are similar to each other. In both cases a block of elements of a specific size is given to each processor in turn. The size of block, m , is specified in parentheses after the distribution method. The only real difference between the two methods is that **BLOCK**(m) implies that each processor *must* get *at most one* block of elements whereas **CYCLIC**(m) implies that each processor *may* get *more than one* block of elements.

BLOCK(m) implies that after giving each processor m elements, all the array should have been distributed; if there are any elements left over an error will have occurred. **CYCLIC**(m) distribution assigns a block of m elements to each processor and then returns to the first and gives a second block to each processor and so on, until all the array elements have been used up.

CYCLIC(m) retains characteristics of both **BLOCK** and **CYCLIC** distributions: in theory, blocks of m elements are grouped together which is useful for neighbourhood calculations and the cyclic distribution policy should promote a reasonable degree of load balancing. Unfortunately, possible benefits are often outweighed by the extra work involved in keeping track of where each array element is resident. The elements owned by a specific processor when using regular **BLOCK**, **CYCLIC** or **BLOCK**(m) distributions can be represented internally by the compiler as a single subscript-triplet, however, **CYCLIC**(m) requires that a *union* of subscript-triplets be used. This factor will add extra complexity to the compiled program as the executable code will contain an extra level of loop nesting compared to other distributions.

BLOCK(m) is useful if **BLOCK** distribution is desired but only over a subset of processors.

The examples given on the slides demonstrate how **BLOCK**(m) has ensured that the first two processors get the majority of the array A, and how **CYCLIC**(m) distribution has assigned a number of discrete blocks of elements of B to each processor.

169: 2D Example

Consider the following 2D array A,

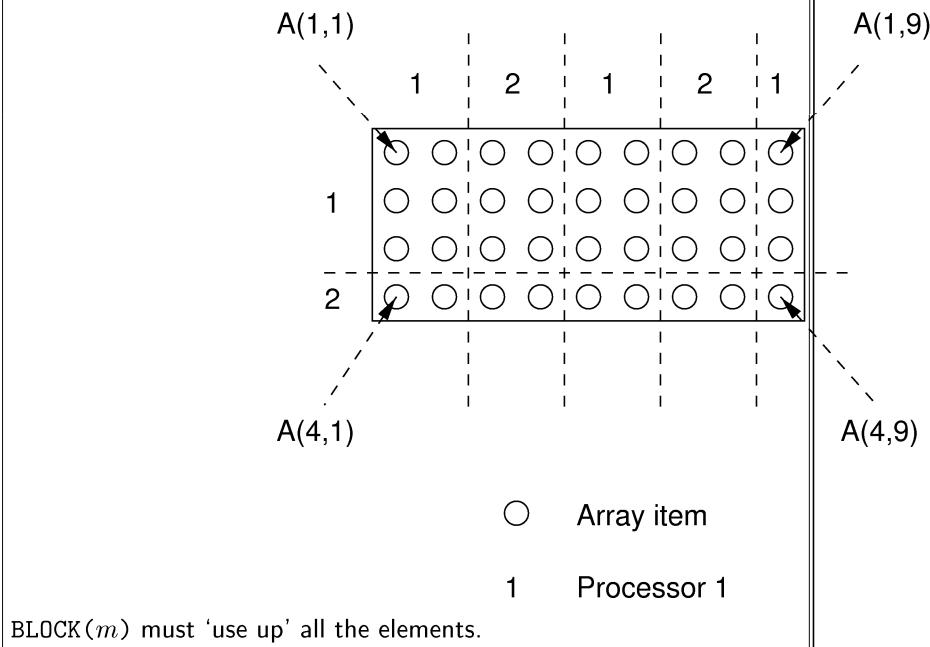
```
REAL, DIMENSION(4,9)      :: A
!HPF$ PROCESSORS, DIMENSION(2)      :: P
!HPF$ DISTRIBUTE (BLOCK(3),CYCLIC(2)) ONTO P :: A
```

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)



Guide for slide: 169

This slide demonstrates 2D distribution. The principal is exactly the same as before. Dimension one (going down the columns) is distributed **BLOCK(3)** and the second dimension is **CYCLIC(2)**.

Ownership,

- $P(1,1): A(1:3,1:2), A(1:3,5:6), A(1:3,9:9).$
- $P(2,1): A(4:4,1:2), A(4:4,5:6), A(4:4,9:9).$
- $P(1,2): A(1:3,3:4), A(1:3,7:8).$
- $P(2,2): A(4:4,3:4), A(4:4,7:8).$

It can be seen that the **CYCLIC(2)** distribution in the second dimension means that the “local set” (the set of elements local to a specific processor) has to be stored as a union of subscript-triplets.

170: 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.

Guide for slide: 170

Alignment of two or more arrays ensures that corresponding elements reside on the same processor. The idea is to view the usage patterns of the arrays and ‘calculate’ how they should be aligned with one another. The simplest case would be the array assignment:

$$A = B + C$$

Here, $A(i)$ is used in the same calculation as $B(i)$ and $C(i)$. It would therefore be sensible to ensure that $A(i)$, $B(i)$ and $C(i)$ reside on the same processor for all i ; this would mean that each component of the array assignment can be performed without any communications.

Alignment can also be used for other things too. It is possible to manually offset arrays relative to each other and it is also possible to specify explicit replication and collapsing of dimensions or of whole objects.

171: 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.

Guide for slide: 171

There is a wealth of alignment syntax. There are a number of different ways of expressing the same thing. It *can* be confusing.

The attributed (Fortran 90) style of declaration is recommended as it allows concise yet clear alignment. The first example

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

aligns A, B and C with T. T is known as the *align-target* and is now the only object that can be distributed. A, B and C can be thought of as “hanging off” T; if T is distributed then so are A, B and C. (Note in this case T can be an array or a TEMPLATE [see later if covered in this course]).

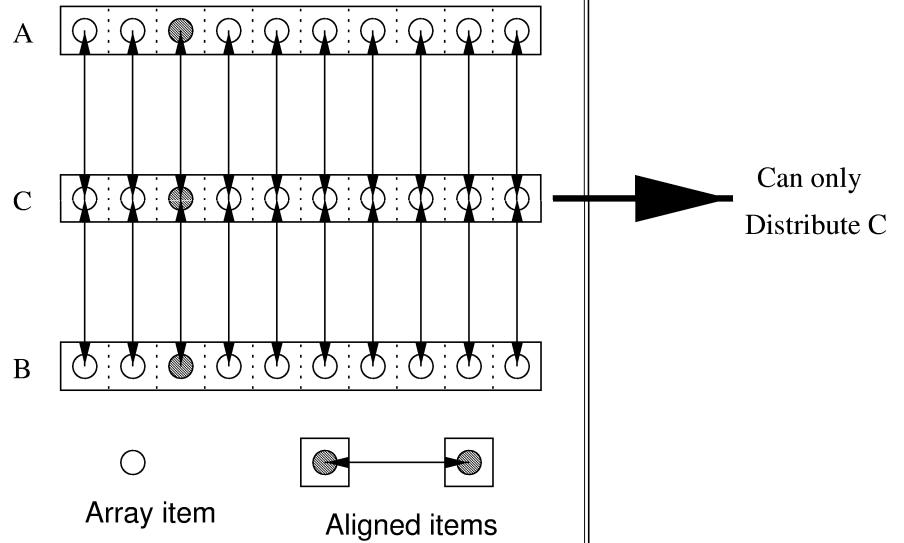
The colons : in the align statement imply shape conformance (*cf* Fortran 90). If A, B, C and T are not the same size and shape then the program is erroneous.

Other alignment syntax will be presented later.

172: 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.

Guide for slide: 172

This is a similar example to the previous slide. A, B and C are aligned and must conform. A(j) and B(j) will reside on same processor as C(j).

The diagram is intended to indicate what is said in the above paragraph! The arrowed lines show alignment and the shaded blobs reiterated this. The processor that receives one of the shaded blobs will also receive the others. Since C is the *align-target* it is the only object that may be distributed.

The alternative alignment syntax

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

implies the same alignment, (ie, that A(j) and B(j) will reside on same processor as C(j)), but does not imply that all arrays are the same size and shape. It does, however, imply that C is the same size or larger than A and B in each dimension. (If this was the case then A(1) and B(1) are aligned with C(1) and so on until the extents of A and B are reached. The j is best thought of as a ‘symbol’ and not as a variable. j does not need a value at this point in the program. It is like saying

$$\forall j \text{align } A(j) \text{ and } B(j) \text{ with } C(j)$$

Colon notation makes a slightly stronger statement than symbol notation.

173: Simple 2D Alignment Example

Given,

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

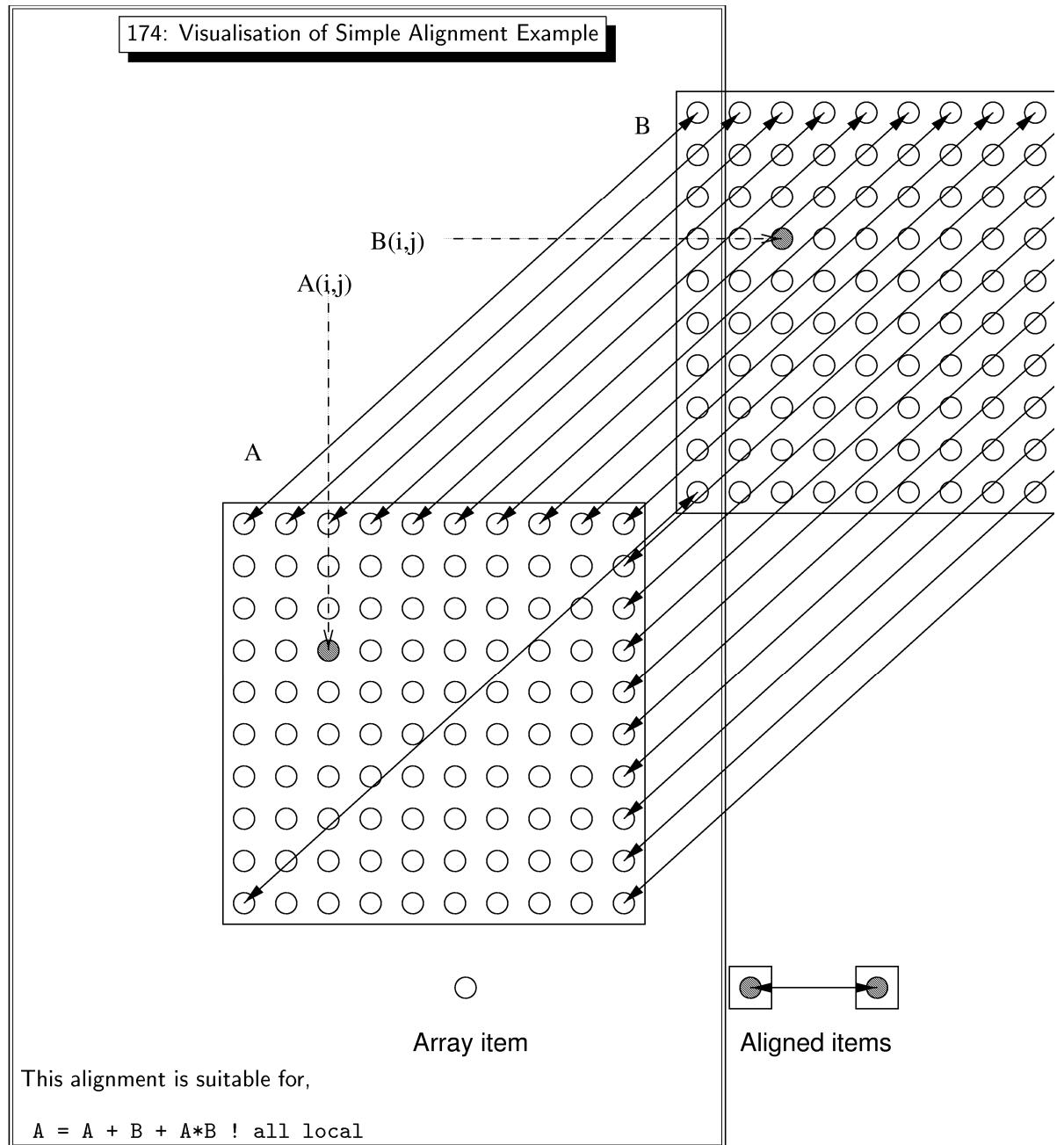
This says: $\forall i, j$, elements $A(i,j)$ and $B(i,j)$ are local.
The following align statement is equivalent but does not imply shape conformance:

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

Guide for slide: 173

This is much the same as the previous example except that we are aligning 2D arrays. Each array is aligned dimension by dimension. The net effect of this alignment is that $A(i,j)$ and $B(i,j)$ reside on the same processor.

The diagram on the next slide should highlight this alignment.



Guide for slide: 174

This diagram is supposed to show how elements $A(1,1)$ and $B(1,1)$ are aligned with each other. Alignment between every element of the first row of A and the first row of B is shown.

The two shaded blobs are also aligned and are therefore resident on the same processor.

175: 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(:, :) 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.

Guide for slide: 175

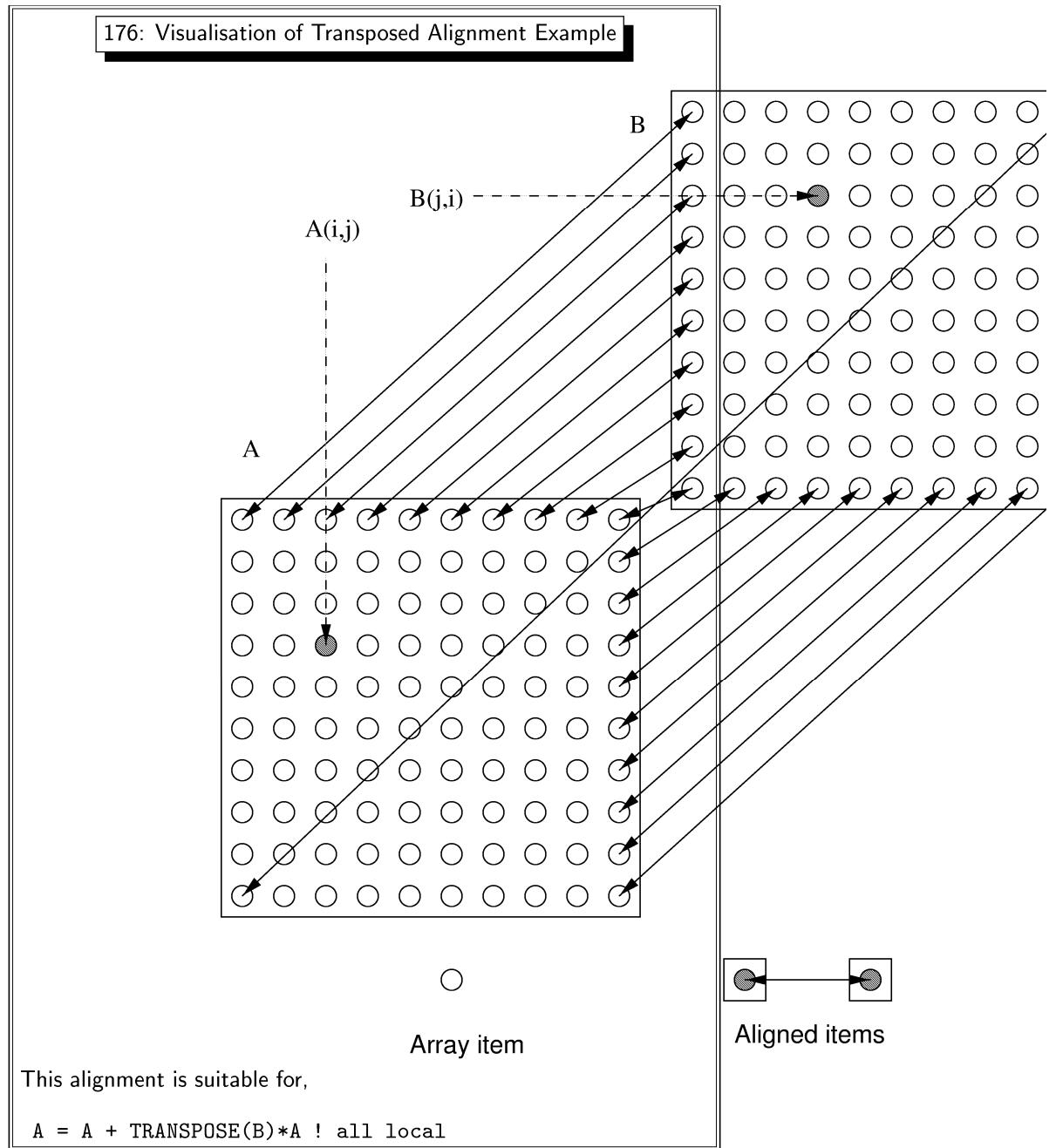
Here we are aligning a row of A with a column of B and vice-versa. The colon syntax cannot be used to specify this, we must also use the symbolic syntax. The statement:

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

uses two symbols (a colon and i) to specify the first dimension of A is aligned with the second dimension of B (the symbol i) and the second dimension of A is aligned with the first dimension of B (the colon symbol). Again the colon implies conformance, in other words, the first dimension of B has the same extent as the second dimension of A.

The second example is similar to the first except that the symbol j is used and the conformance between dimensions is for the first dimension of A and the second dimension of B.

The last example uses i and j as the symbols to achieve the same effect as the first. The only difference is that there is nothing implied about the extents of the dimensions.



Guide for slide: 176

This diagram is supposed to show how elements $A(1,:)$ (the first row of A) and $B(:,1)$ (the first column of B) are aligned with each other and so on. A is aligned with the transpose of B.

The two shaded blobs are also aligned and are therefore resident on the same processor.

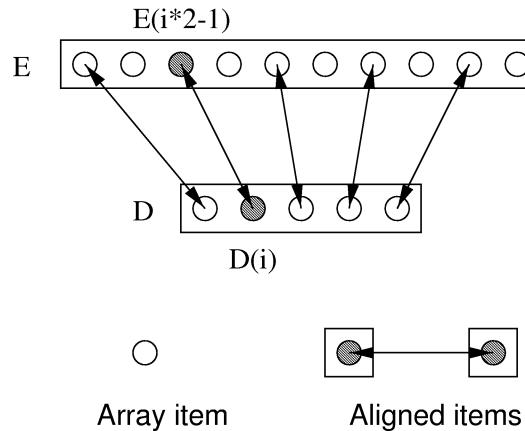
177: Strided Alignment Example

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
```

Guide for slide: 177

A more complex example. Here, D(1) is aligned with E(1), D(2) is aligned with E(3), D(3) with E(5) and so on. This form of strided alignment is specified using the subscript-triplet (or colon) notation.

It can be seen that every second element of E has an element of D aligned to it.

The second example in the middle of the slide demonstrates how the symbolic notation may also be used.

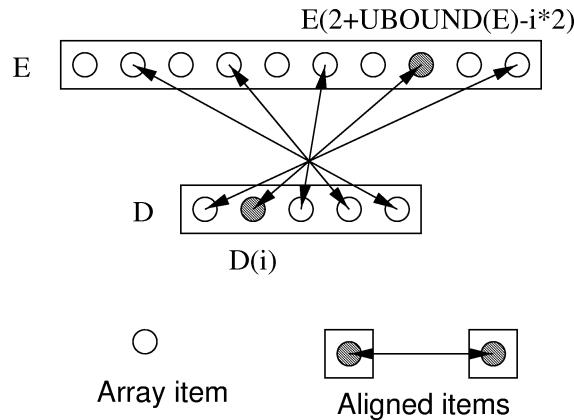
178: Reverse Strided Alignment Example

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
```

Guide for slide: 178

This is much the same as the previous example except that D is aligned in the reverse order. D(5) is aligned with E(10), D(4) is aligned with E(8), and so on.

The first example shows how to achieve this using subscript-triplet notation and the second (in the middle of the page) uses the symbolic notation.

179: 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.

Guide for slide: 179

Given the patterns of array references in the program **Warty**, we would make the decision to align the arrays as stated on the slide. Our main concern is to minimise the communications in a program.

The assignment

$$E = D(:,4) + C(:,2)$$

Combines E(1) with D(1) and C(1); E(2) with D(5) and C(3); E(3) with D(9) and C(5) and so on. We can use the subscript-triplet expressions in the assignment as the basis for our alignment specification.

The alignment that is given will result in zero communications.

(Clearly this is an artificial example. In the real world, there will be many different assignments and it will be much tougher to choose a good alignment. This is one of the most difficult area of HPF programming.)

Notice how it is only the align-target D that is distributed.

180: 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.

Guide for slide: 180

Again, allocatable arrays behave in a very similar way to regular arrays. The main thing to remember is that the align-target must already be in existence if something is to be aligned to it. The alignment and / or distribution takes place immediately after allocation.

181: 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.

Guide for slide: 181

Align (and distribution) statements containing references to allocatable arrays appear in the declarations area of the code as normal but the actual alignment (or distribution) only takes place after an object has been brought into existence. The first brace of ALLOCATE statements are OK as the align-target B is allocated before the *alignee* A. The second ALLOCATE is also OK as the statement is executed from left to right; this means that B is again allocated before A.

If one was to write

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

then the alignee, A, would be allocated and an attempt would be made to align it with the align-target B. Since B does not yet exist, an error would ensue. The same would happen if both allocations appear in the same statement but in the wrong order, viz:

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

B should have been allocated first.

182: 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.

Guide for slide: 182

There are quite reasonable guidelines on what can and cannot be done with allocatables arrays.

Aligning **X** with **A** (the first example) is not permitted because the colon notation implies that **A** must be the same size and shape as **X**. Since **A** is allocatable this may not be the case. It could also be that **A** is smaller than **X** which again would not make sense. [Note: there is a problem in aligning allocatables to **TEMPLATES** (see later for templates). Templates have a fixed size, so how does one know whether the allocatable will be larger than the template in one or more dimensions?]

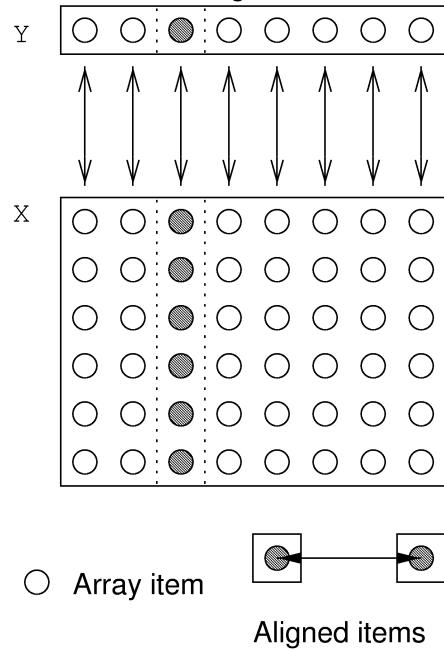
The second “pitfall” is due to the colon notation suggesting that **A** and **B** are conformable but the **ALLOCATE** statements clearly stating that they are not. The third example, which correctly states what was intended by the second example, does not use the colon notation and therefore does not imply conformance.

183: 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 $Y(i)$.

Guide for slide: 183

Collapsing dimensions is useful for certain reference patterns, for example if a whole row or column is used in the calculation of a single element. The * distribution method allows for dimensions to be collapsed, but this can also be achieved during alignment.

The general rule is: *if a * appears in the align-source-list (the parenthesised list on the LHS of the WITH clause associated with the alignee) then the corresponding dimension is collapsed.* (I like to think of it as aligning all dimensions that contain a colon; the * dimensions are left over so are collapsed.)

In the example, the * is in the first dimension meaning that the first dimension of X is not distributed. The whole of the first *column* of X is aligned with the first *element* of Y, the whole of the second column of X is aligned with the second column of Y, and so on. The processor which received Y(i) will also receive X(:,i).

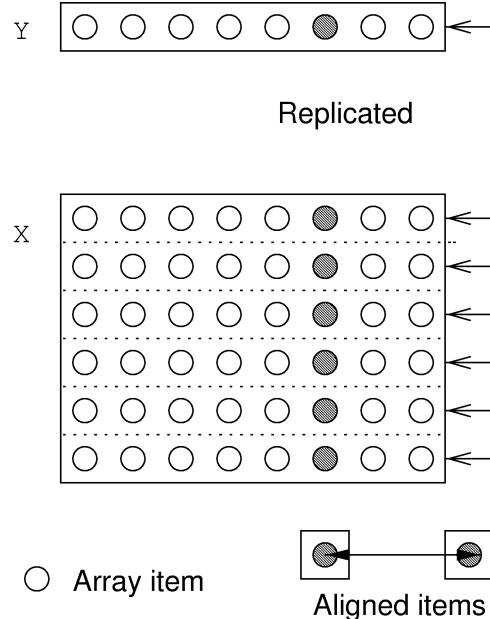
There must be the same number of non-* dimensions on either side of the WITH clause.

184: 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 $X(:,i)$.

Guide for slide: 184

Replication is also useful for specific reference patterns. A good example would be if the same section of an array is used in a calculation on every processor.

The general rule is: *if a * appears in the align-subscript-list (the parenthesised list on the RHS of the WITH clause associated with the align-target) then the corresponding dimension is collapsed.* In the quoted example, the * is in the first dimension meaning that Y is replicated along every row of X. (I like to think of it as aligning all dimensions that contain a colon; the * dimensions are left over so it is in this dimension that the alignee is replicated.)

A copy of the first *element* of Y is aligned with the first *element* in every row of X, a copy of the second element of Y is aligned with the second element of every row of X, and so on. The compiler make sure that all the copies have the same value.

Every processor which receives any element of $X(:, i)$ will also receive a copy of $Y(i)$.

There must be the same number of non-* dimensions on either side of the WITH clause.

185: 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.

Guide for slide: 185

```
!HPF$ ALIGN Y(:) WITH A(:,*)  
! Y aligned with each col of A
```

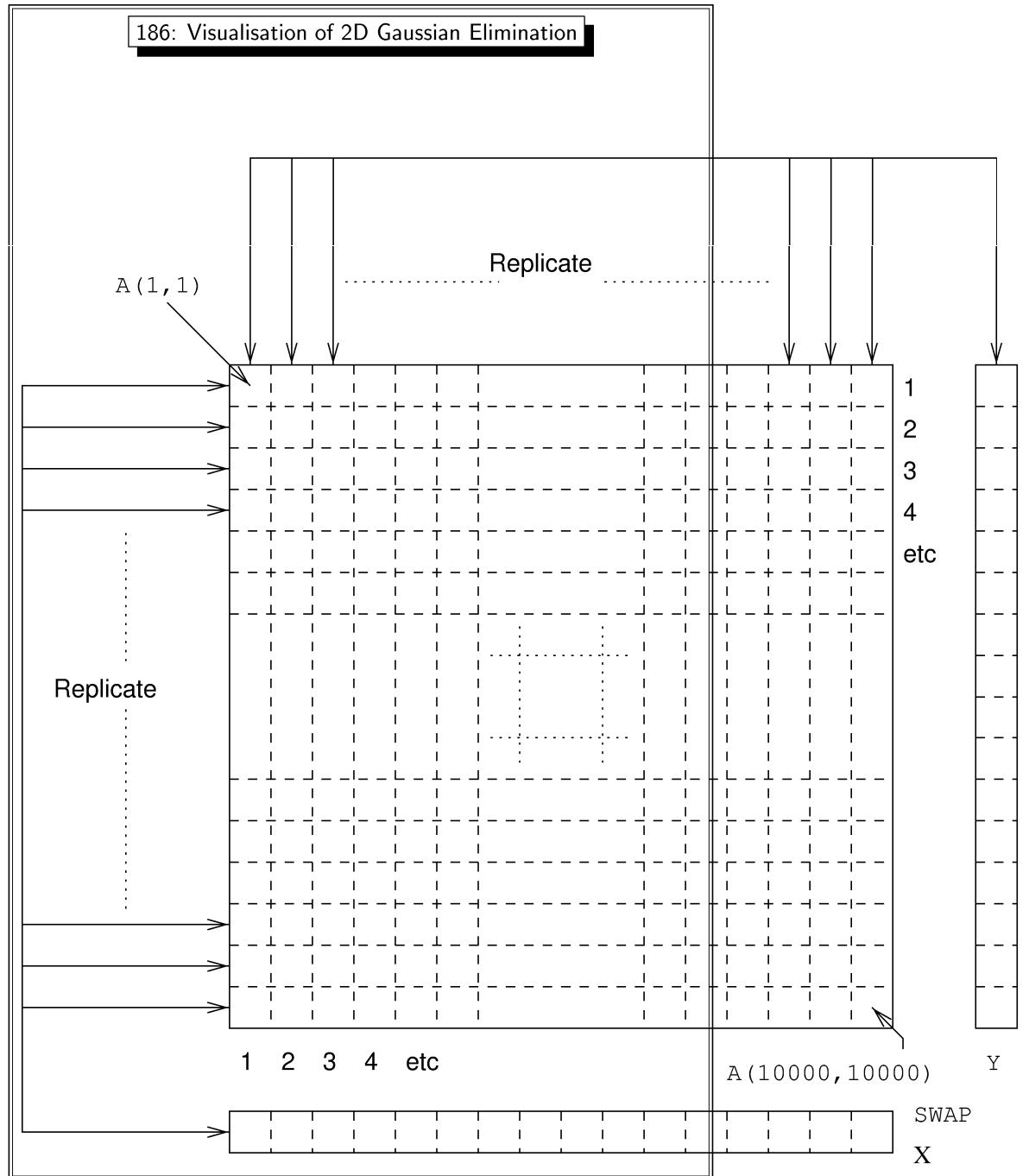
$Y(k)$ is always used in the same statement as $A(k,i)$. This implies that Y should be aligned with the first dimension of A . In order that the value of i is unimportant $Y(k)$ should be local to $A(k,:)$ in other words a copy of Y should be aligned to every column of A .

```
!HPF$ ALIGN Swap(:) WITH A(*,:)  
! Swap aligned with each row of A
```

A similar argument to above except that $SWAP(k)$ should be local to $A(i,k)$, ie $SWAP$ should be aligned with the second dimension of A . In order that the value of i is unimportant $SWAP(k)$ should be local to $A(:,k)$, in other words a copy of Y should be aligned to every row of A .

```
!HPF$ DISTRIBUTE A(CYCLIC,CYCLIC) ! onto default grid
```

There is no advantage in using **BLOCK** distribution, calculation of a particular element does not use that elements neighbours in the calculation, it is therefore safe to balance the load by using **CYCLIC** distribution.



Guide for slide: 186

This diagram is supposed to represent that Y is replicated along each column of A and SWAP (and X) with each row of A.

187: 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
```

Guide for slide: 187

These new intrinsics provide information about the *physical processors*, ie, the hardware inside the box and *not* the configuration as specified in a PROCESSORS directive.

These two intrinsics can be used in *specification-expressions*, they can be thought of as constants *for one run of the program only*. Their value cannot be assigned to a PARAMETER or even to a variable. The following will be flagged as an error:

```
INTEGER, PARAMETER :: NoOfPs = NUMBER_OF_PROCESSORS()  
INTEGER :: NumProcs = NUMBER_OF_PROCESSORS()
```

both functions may only be used in the context shown on the slide.

The NUMBER_OF_PROCESSORS intrinsic has one optional argument, DIM. If this is specified then the number of processors in that particular dimension is returned. For example, on a 2048 processor hypercube machine

```
PRINT*, NUMBER_OF_PROCESSORS(2)
```

will display 2 whereas

```
PRINT*, NUMBER_OF_PROCESSORS()
```

would give 2048

For a single processor workstation PROCESSORS_SHAPE returns zero-sized array of rank one.

188: Template Syntax

Templates are intended to make alignments easier and clearer.

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

Guide for slide: 188

Templates are conceptual objects. They do not take up any storage space and must be statically defined. (This means that they do not interact well with ALLOCATABLE arrays.) One way to think of a template is as an array with zero-sized elements (which cannot be assigned to).

The ordering implied on the slide is not strict. It is quite legal to align arrays to a template *before* specifying its distribution, however, the first stage must always be to declare the template.

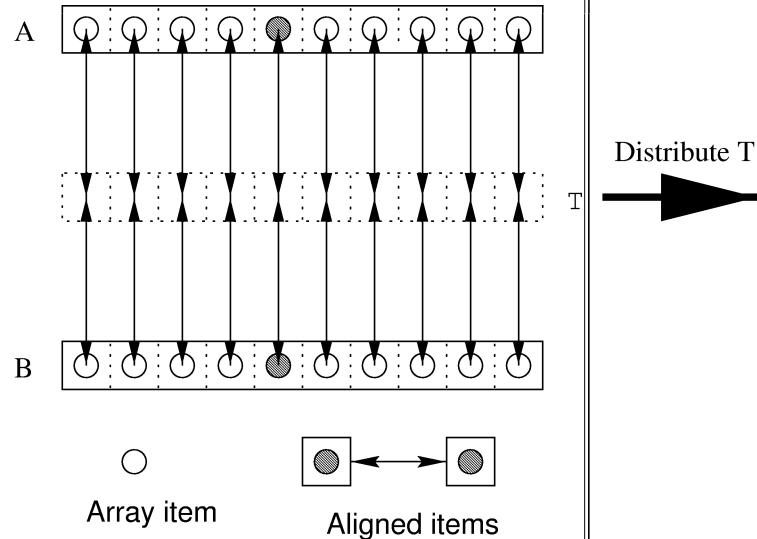
Distribution of a template follows exactly the same principles as distributing an array, with respect to distribution methods, blocksizes and so forth.

There is a school of thought that says that TEMPLATES are an unnecessary addition to the functionality of the language.

189: 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
```



Guide for slide: 189

The alignment between an array and a template works in exactly the same way as between two arrays.

In this example, a 10 element TEMPLATE, T is declared. Its distribution method is BLOCK and there are two arrays aligned to it, A and B. Since T is used as the *align-target* it is the only entity that may be distributed. (A and B are alignees so may not be.)

190: 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.

Guide for slide: 190

The combined declaration format is more concise and intuitive. The distribution method is an attribute of the TEMPLATE and not of the arrays that are aligned to the template.

191: Another Template Example

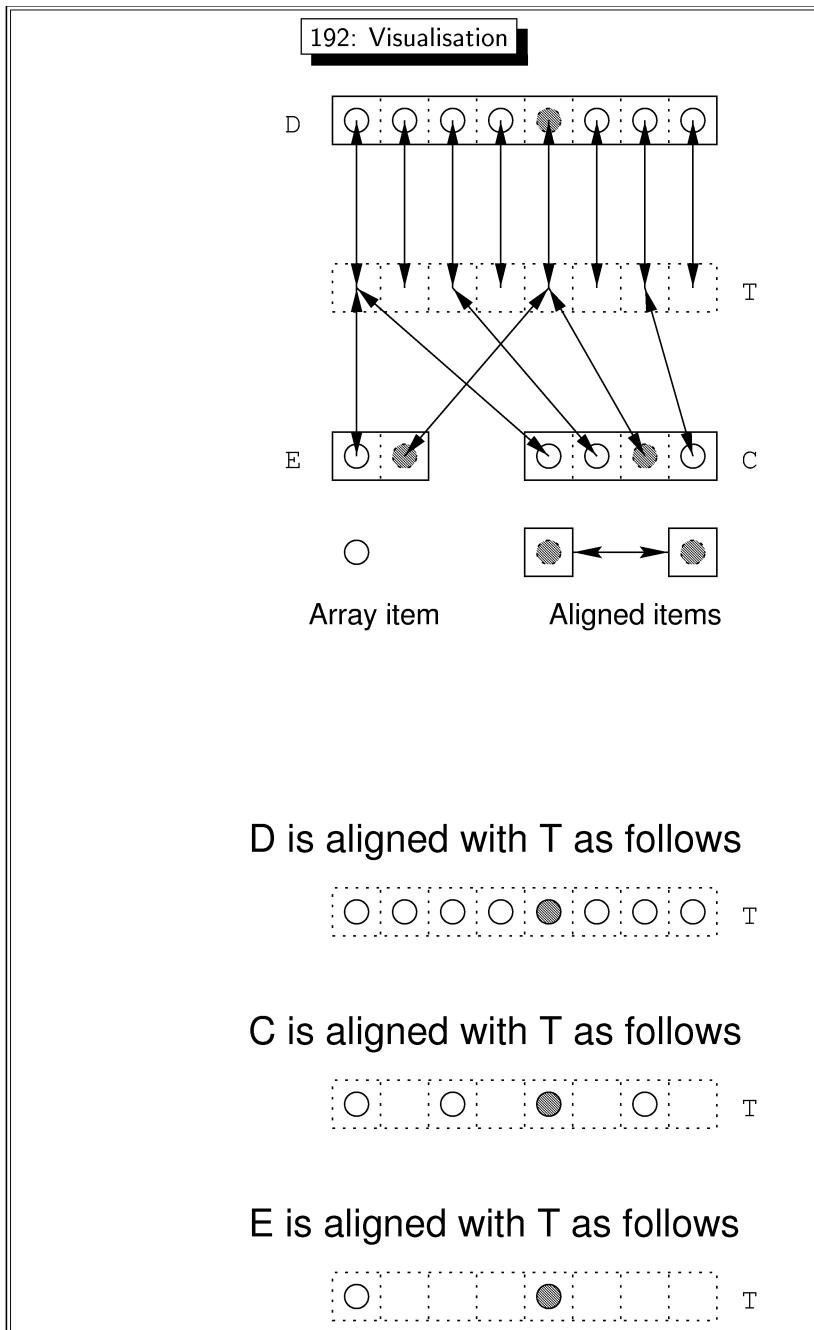
Recall an earlier example,

```
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.

Guide for slide: 191

This is very similar to an earlier example which demonstrated stride array alignment. This time, however, all the arrays are aligned to a template which is then distributed. As expected, the template must be the same size as or larger in each dimension than anything that is aligned to it.



Guide for slide: 192

The diagrams show how each array is aligned with the template. At the bottom there is an exploded view of how each individual array relates to the template. The element from each array denoted by the shaded blob can all be seen to be aligned with T(5).

193: Alignment and Distribution of Templates

- ALIGN A(:) WITH T1(:,*) — $\forall i$, 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(:) — $\forall i$, 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.

Guide for slide: 193

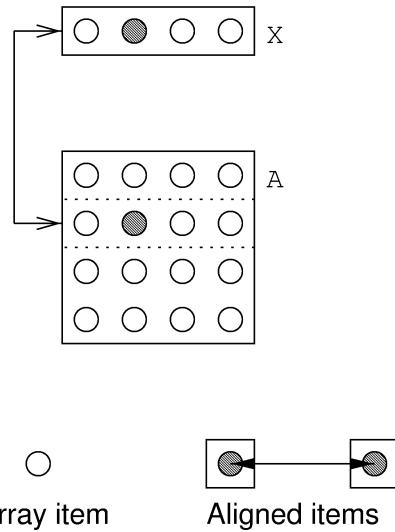
The five examples have already been met when dealing with the distribution of arrays. The colon notation implies conformance between the two objects, the symbolic notation is used when there is no shape conformance. This is just the same as with arrays.

194: 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
```

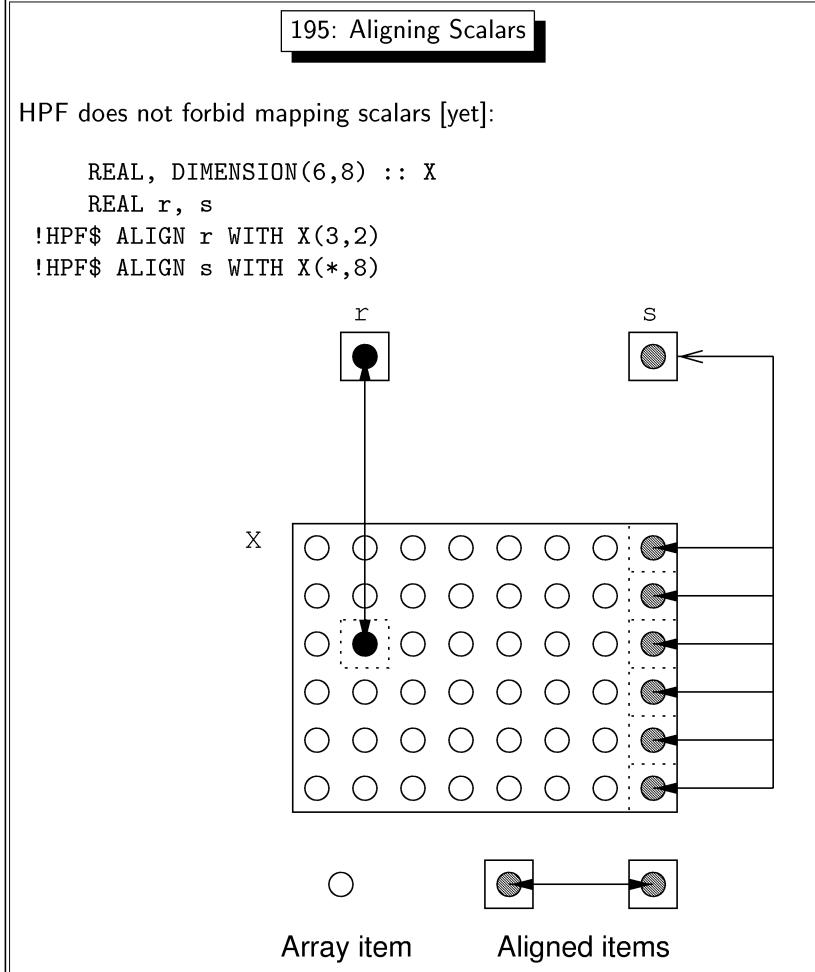
In this case *i* is *not* used as a symbol it needs a value. Since *i* = 2 the section is aligned with row 2.



Guide for slide: 194

It is possible to embed an array within a template; the given example aligns a 1D vector, X , with a specific row of a template. In this case the i is *not* used as a symbol but is used as a variable (with a value). It is easy to tell that this is what is happening because there is not corresponding i on the LHS of the WITH clause. If i did not possess a value at the point at which it was used then the program would be erroneous.

Remember if a variable name is used simply as a position marker then there will be an occurrence at *both* sides of the WITH clause; if its value is to be used it will only appear on one side.



Guide for slide: 195

It is possible to align a scalar with a specific template location or series of template locations.
HPFF are thinking of withdrawing this feature.

196: 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
```

Guide for slide: 196

As all HPF compilers are able to supply their own default mappings which may but is not bound to be replication. It is often prudent to explicitly replicate objects if this is what is desired. The slide shows the procedure which can be followed in order to achieve this.

Lecture 8:
Forall and Independent Loops

198: 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.

Guide for slide: 198

The HPF data distribution directives merely tell the compiler how to distribute arrays amongst the available processors, they do not express parallelism.

Parallelism is expressed through the use of special (often Fortran 90) statements such as array assignments, intrinsic calls with array arguments or any of the new HPF features:

- FORALL** statement,
- PURE** procedures.
- INDEPENDENT** directive,
- NEW** variables,

FORALL and **INDEPENDENT / INDEPENDENT NEW DO** loops allow parallel assignment to arrays without actually using an array assignment statement. They are very expressive language features and often replace the need for contorted and multiply nested calls to array valued Fortran 90 intrinsics. **PURE** procedures are simply procedures that can be invoked from within a **FORALL** statement or construct. They are side-effect free. All Fortran 90 intrinsic procedures are **PURE**.

Both **FORALL** and **PURE** are included in the new Fortran standard, informally known as Fortran 95.

199: 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..

Guide for slide: 199

When used without the mask, the **FORALL** statement simply performs conceptually parallel assignment — any assignment that can be expressed in array syntax can also be expressed by a **FORALL** statement (or construct). The **FORALL** statement can also be used to express parallel assignments which it is impossible or very tricky to code using Fortran 90 features. Note that although the index variables look like the **DO** loop equivalent — the colons imply that this is array syntax.

If a mask is added, such as in the given example, **FORALL** can be made to resemble a **WHERE** statement. Indeed, the example could actually be expressed using a **WHERE** statement:

```
WHERE ( A.NE.0 ) A = 1/A
```

Again, very complex masked assignments can be made syntactically simpler by the use of a masked **FORALL** statement.

200: 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.

Guide for slide: 200

It would be very awkward to express the assignments given in the first two bullet points using only array syntax and intrinsic procedure calls. The first performs parallel assignment of a vector to the diagonal of a matrix and the second gives a loop which contains parallel assignment to a section of an array. Each time around the loop the section being assigned to grows in size. (Note that the DO loop here is still executed sequentially, it could be parallelised by prefixing it with a

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

directive, [see later for a discussion of the INDEPENDENT directive,] or by using a nested FORALL construct:

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

[See later for a discussion of the FORALL construct].)

The second bullet-point example shows how the FORALL indices can be used in the RHS expression. This is easy to perform in a sequential DO loop but is not easy to express in array syntax. This FORALL is also masked so that the diagonal elements remain unchanged. REAL is an intrinsic procedure (function) so by definition is PURE.

The third example again shows how PURE procedures can be used in a FORALL body. Here, SIN could be the Fortran 90 intrinsic function or it could be a user-defined PURE function.

The final bullet point presents an example of parallel array assignment using indirect addressing (vector subscripting). The use of the FORALL index variable in the RHS would make this assignment difficult to express using regular Fortran 90 syntax.

201: 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.

Guide for slide: 201

It is important to be aware of the order of execution of a FORALL statement. Since FORALL is executed in parallel over a number of processors and since there is a distinct and rigid ordering implied in its execution, each processor must be sure that all other processors have completed the current step before any processor is allowed to continue. This means that there are synchronisations between every step in the execution process. In other words, *every* processor must have finished evaluating the subscript expressions before *any* processor begins to work out the mask. Likewise the mask calculation must be completed by every processor before calculation of the RHS may begin, and so on.

As always in Fortran, it is not permissible to perform assignment to the same element more than once in the body of a parallel assignment statement (such as FORALL). Compare this with parallel assignments to vector subscripted LHS's.

202: 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)
```

Guide for slide: 202

Take care - although a **FORALL** may look a bit like a **DO** loop it is more akin to an array assignment statement. A **DO** loop has an implied sequential ordering, a **FORALL** statement is completed in one go!

203: 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 FORALLs,

```
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
```

Guide for slide: 203

The **FORALL** construct is very similar to the **FORALL** statement except that the indices and mask can be used to control more than one parallel assignment (a bit like the **WHERE** construct). As well as containing regular assignment statements, the body of a **FORALL** may also contain a further **FORALL** block or a **WHERE** block. This can lead to mind-bendingly complex assignments being expressed very concisely!

The two examples given on the slide demonstrate a **FORALL** containing two assignments and a **FORALL** containing nested **WHERE** and **FORALL** statements.

204: 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.

Guide for slide: 204

Pure procedures are prefixed by the **PURE** keyword in the same way as recursive procedures are prefixed by the **RECURSIVE** keyword. They must be side-effect free in the sense that they cannot change the global state of the program. (If they could then the order of invocation of the individual procedures would be significant.) Clearly, any other procedures that are invoked by a pure procedure must themselves be pure. (All Fortran 90 intrinsics and Fortran 95 **ELEMENTAL** functions are pure.)

The **ALLOCATE** statement is not allowed in case one of the processors that is running a copy of a procedure is unable to fulfill the allocate request. External I/O is disallowed in case any of the **READ / WRITE** statements lead to an error, such as writing to a full disk.

Note that a pure procedure may still involve communications. If it does then there will be synchronisation points within the procedure. Describing a procedure as **PURE** does not mean that it does not involve communications.

205: Pure Procedures

Must follow certain rules:

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

Guide for slide: 205

There are a number of rules which are applied to determine whether a procedure is pure or not. They are all concerned with making sure that each copy of the procedure does not interact with any other copy of the same procedure or change the global state of the program.

To make things simple, **PURE** functions must behave like mathematical functions in the sense that their arguments cannot be changes (must have **INTENT(IN)**).

206: 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)
```

Guide for slide: 206

Note the PURE prefix. Should give an explicit interface so that the compiler can witness that the procedure is pure.

207: 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)
```

Guide for slide: 207

Here, the subroutine invokes a PURE function F. The argument intents are given in an interface for F. Each processor can happily execute its copy of G in parallel with each other processor.

It is *very* good form to give an interface.

208: 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.

Guide for slide: 208

This example merely illustrates that if a PURE procedure contains an IF block then each processor could be performing a different task at the same time.

209: 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
```

Guide for slide: 209

INDEPENDENT appears in the executable area of a program unit — it must prefix a **DO** loop or **FORALL** statement or construct. In both case the directive asserts that no ‘iteration’ affects any other iteration either directly or indirectly.

Applying an **INDEPENDENT** directive to a **DO** loop will allow the iterations to be performed in parallel.

Applying an **INDEPENDENT** directive to **FORALL** means that there does not have to be a synchronisation point between calculating the RHS expression and beginning to assign to the LHS. This should speed up execution.

210: 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.

Guide for slide: 210

All the conditions imposed on INDEPENDENT ensure that it really does not matter which order the iterations are performed in.

If an INDEPENDENT loop

- assigns to same element twice parallel integrity would be lost,
- contains EXIT, STOP or PAUSE then the iterations must progress sequentially so as to ensure that the loop is terminated on the correct iteration,
- contains jumps out of loop then, again, the iterations must progress sequentially so as to ensure that the loop is terminated on the correct iteration,
- performs external I/O then the iterations must progress sequentially so as to ensure that the file is read from or written to in the correct order,
- prefixes statements other than DO or FORALL then it has no meaning and will cause an error.

(Hint: To decide whether a Fortran 90 or FORTRAN 77 loop is INDEPENDENT reverse the upper and lower bounds and negate the stride. If the loop still produces *exactly* the same effect then it must be INDEPENDENT.)

211: 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
```

Guide for slide: 211

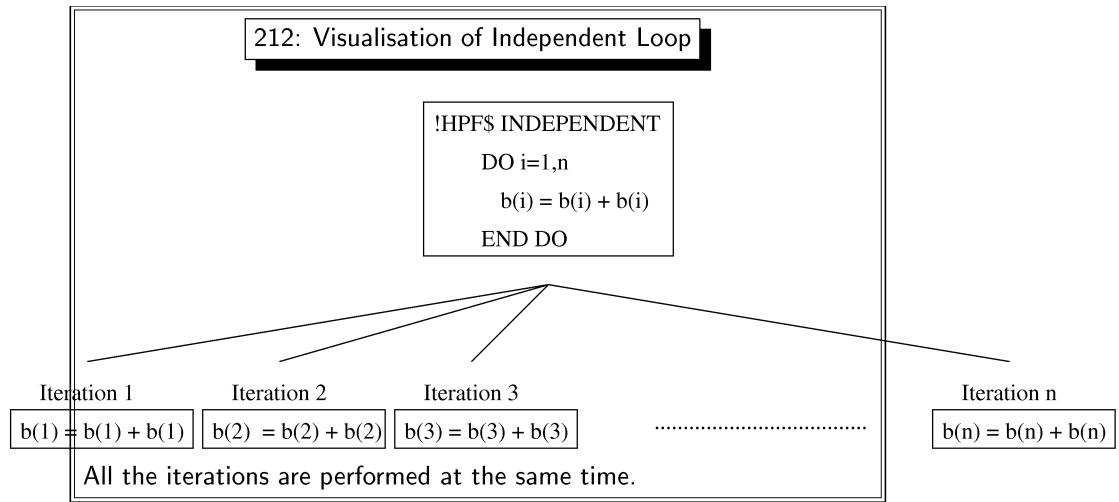
The concept of data dependency is relevant here. We must ask the question: *does a later iteration depend on the result of an earlier one?* (This is very similar to asking the question: *can we perform the iterations in the reverse order?*)

The first example clearly has no dependency, the array item $b(i)$ read and written on the same iteration and is not accessed again. This loop is ripe for being denoted as INDEPENDENT.

The second and third loops contain a forward and backward dependency respectively. For the first of these, if the iteration $i+1$ were performed before iteration i then the value of $b(i+1)$ used to calculate $b(i)$ would be incorrect. The third loop contains a backward reference. If iteration i were performed before iteration $i-1$ then the value of $b(i-1)$ used to calculate $b(i)$ would be incorrect.

The forth and final loop is INDEPENDENT. None of the values of the array b are updated, so, assuming that b is not storage associated with a or that b and a are not pointers to the same area of memory, the order that the iterations performed in are unimportant.

The value of i will be set at the end of the loop so that it has the same value as it would in an ordinary Fortran 90 program.



Guide for slide: 212

The diagram illustrates how the compiler views an INDEPENDENT DO loop. The compiler is able to arrange that *every* iteration of the loop is (conceptually) executed at the same time. The diagram illustrates this by enumerating each iteration and then placing them side by side to denote potential parallelism.

213: 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.

Guide for slide: 213

The first independent DO loop is similar to the last loop of the previous examples slide. The object **b** is not assigned to within the loop body which means that assuming there is no ‘hidden’ connection between **a** and **b** such as storage association, then no dependence can arise. The FORALL statement is independent for the very same reason, here, each processor can perform its local calculations without having to check on the progress of other processors; it is able to perform all its assignments to **a** before having to synchronise.

The final example on this slide illustrates how a DO loop which involves assignment to a vector subscripted (indirectly addressed) object can be parallelised. The INDEPENDENT directive asserts that all values of **index(1:n)** are unique. This DO loop could be rewritten as an array assignment:

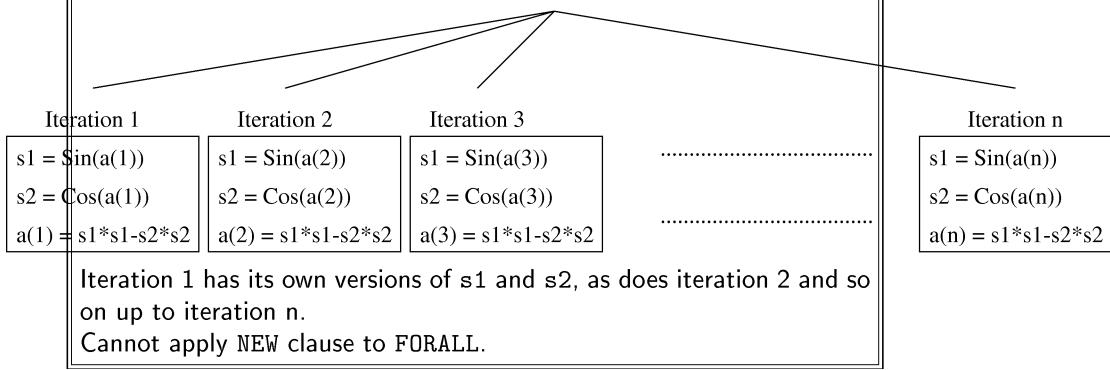
```
A(INDEX) = B(0:n-1) + B(1:n) + B(2:n+1)
```

to achieve the same effect.

214: INDEPENDENT NEW Loops

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

```
!HPF$ INDEPENDENT, NEW(s1,s2)
DO i=1,n
  s1 = Sin(a(i))
  s2 = Cos(a(i))
  a(i) = s1*s1-s2*s2
END DO
```



Guide for slide: 214

The **NEW** clause tells the compiler that in order to parallelise the following loop, each iteration (in practise, each processor,) should make its own (local) version of the listed scalar variables. In the cited example, the iterations may be performed in parallel if and only if each iteration uses its own version of **s1** and **s2**.

The diagram illustrates that the iterations can be performed in parallel — each individual iteration includes references to its own **s1** and **s2**.

215: 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.

Guide for slide: 215

Note that there are a few rules associated with NEW variables. Referring back to the previous slide, at the end of the INDEPENDENT loop **s1** and **s2** do not have well defined values. (There have been **n** copies of these variables — which copies hold their true value?) It is therefore forbidden to attempt to use a NEW variable after its appearance in a NEW clause without first assigning a fresh value to it. This means:

```
!HPF$ INDEPENDENT, NEW(s1,s2)
DO i = 1,n
  s1 = SIN(a(i))
  s2 = COS(a(i))
  a(i) = s1*s1-s2*s2
END DO
k = s1+s2
```

is forbidden due to the last-line referencing of **s1** and **s2**.

NEW is not valid with FORALLs because it is not needed. The only entities that NEW would make any sense being applied to are index variables of nested FORALLs, however, due to the fact that it is not valid to refer to these index variables outside of the FORALL, the compiler is able to create and destroy temporary versions of these indices at will with no modification of the semantics of the program.

Henry Zongaro postulated that “a variable which appears in NEW must not be specified in an EQUIVALENCE statement”. This seems very reasonable and, even though this is not laid down in the standard, the rule should be obeyed.

New variables cannot be SAVED or be dummy arguments or pointers because this would compromise the compilers ability to create instances of the NEW variables on each processor in the grid.

216: 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.)

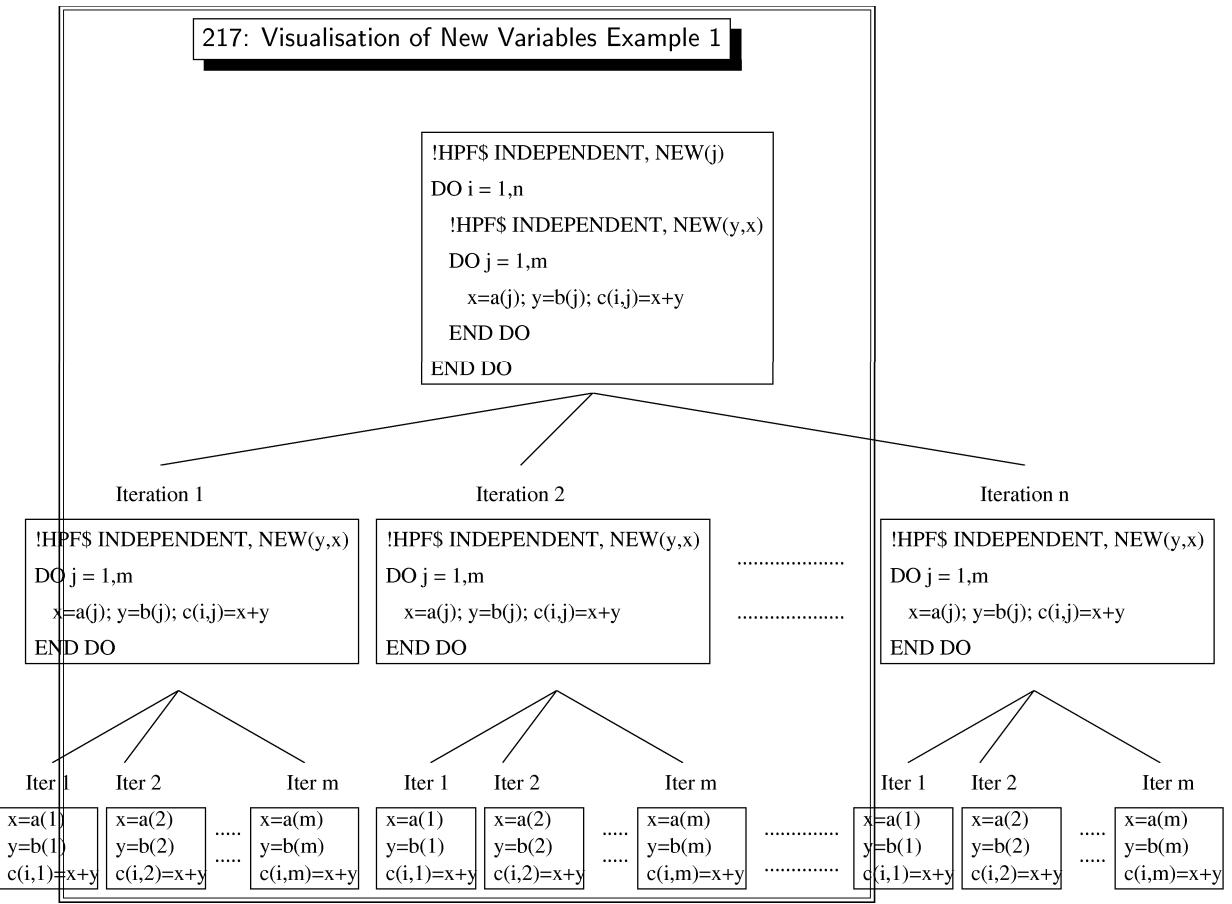
Guide for slide: 216

With a nested loop such as this, it is the inner loops that must be examined first. Inspection of the *j* loop reveals that if copies were made of *x* and *y* for each iteration the loop would be INDEPENDENT. In order to parallelise, the directive:

```
!HPF$ INDEPENDENT, NEW (x,y)
```

should prefix the inner loop.

Attention can now turn to the outermost loop. We could perform all the *i* iterations at the same time if there was a separate DO *j* loop for each iteration. Specifying *j* to be a NEW variable would do exactly this.



Guide for slide: 217

Looking at the outermost leaves of the diagram will illustrate what can be performed in parallel. It can be seen that all $n \times m$ iterations can be performed independently. There is no interaction between any iterations.

218: 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.

Guide for slide: 218

The final NEW variables example illustrates a situation when only the outermost loops can be executed independently.

Beginning at the innermost loop it is clear that $a(i_1, i_2, i_3)$ is assigned to n_4 times during execution of the i_4 loop. The value set on one iteration is used in the next. Clearly, due to the multiple assignments to $a(i_1, i_2, i_3)$, all iterations of the i_4 loop *cannot* be performed at the same time! This innermost loop is therefore not independent. The next loop, however, is INDEPENDENT. There is no problem with $a(i_1, i_2, i_3)$ being assigned to on more than one iteration since now i_3 is the index variable. The i_3 loop is independent because we can perform all n_3 instances of the i_4 loop at the same time.

The i_2 and i_1 loops are clearly independent because again $a(i_1, i_2, i_3)$ is not assigned to on more than one iteration.

219: 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!

Guide for slide: 219

Parallel input and output is a difficult area. The HPFF addressed this area but in the end decided not to put anything into the specification.

In theory, one disk could be attached to each processor in the grid and the local sections of the arrays could be stored on the local disks. In practise there are a whole host of problems associated with this — what if the program is recompiled with the distributions of the object changed. Each processor will no longer own the same parts of the arrays as it used to but so the data stored on the local disks will not have changed!

Most HPF compilers only allow I/O via one of the processing nodes. All input is read by this node and the data communicated to the processor that actually owns it. Likewise any output must be sent to this same node where it is then written to disk. The amount of work needed to perform such a task is quite large, first of all, each processor must send its local set to the designated processor. This processor must then reconstruct the whole array in its local memory and finally output to disk. Whilst this is happening, the rest of the processors must generally wait for confirmation that everything has gone smoothly. A badly placed output statement in the middle of a nested loop will totally destroy performance.

Lecture 9:
Procedures

221: 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.

Guide for slide: 221

In Fortran 90, information is communicated into a procedure either via the argument list or else via global data stored in a **MODULE** (or a **COMMON** block).

Due to the nature of HPF, it is not permitted to supply HPF objects, for example, **TEMPLATE** or **PROCESSORS** grids, as actual arguments — a different method must be employed. It is possible (and, indeed, desirable) to use **MODULEs** to allow global mapping objects to be visible to all subprograms. This will be discussed later.

HPF supports a number of mechanisms for informing a procedure of the mapping of its dummy arguments. Every object that appears as a dummy argument must have some sort of mapping. HPF allows the programmer to say one of the following things about each argument:

- “align the data as follows”. This is called **prescriptive** distribution,
- “the data is aligned as follows”. This is called **descriptive** distribution,
- “inherit the distribution from the dummy arguments”. This is called **transcriptive** distribution.

During the development of HPF compilers it has become clear that it will *never* be easy to implement transcriptive (or **INHERITED**) distribution efficiently. The problems arise with array sections being used as actual arguments; in Fortran 90 (and of course HPF) it is possible to collapse one or more dimensions of an actual argument by using a scalar index, for example,

```
CALL Subby(A(i,:,:,:))
CALL Subby(A(:,j,:,:))
CALL Subby(A(:,:,k,:))
CALL Subby(A(:,:,:,:1))
```

In each of the invocations above, the dummy argument will be mapped to a different subset of processors so, upon compilation, the compiler will have to generate four different strains of **Subby** (one for each different dimension that is collapsed). For each of these strains, there will have to be one version of the procedure for each of the different distribution methods present in the program (could be five). This could lead to 20 different versions being needed for one single argument.

If our subroutine had more than one argument then a version of **Subby** would have to be generated for each possible combination of arguments, with two 4D array arguments this would mean 20×20 or 400 instances of **Subby** must be generated. It can be seen that things can very soon get out of hand!

The only way to get around this problem is to generate a general version of the **Subby** subroutine to cater for all possible mappings and combination of mappings. This routine would effectively have to resolve mappings at run-time which would slow down execution to an unacceptable level (**LPF** - Low Performance Fortran!)

In this course we only discuss prescriptive and descriptive distributions.

222: 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.

Guide for slide: 222

Procedures can contain declarations of the same classes of HPF objects that occur in a main program. A procedure can declare a processor grid or a template, can align arrays relative to each other or to a template and can specify distribution methods. Here, we are only concerned with two different styles of declarations in procedures: prescriptive and descriptive. Prescriptive directives look exactly the same as main program directives; descriptive directives are actually assertions about the global state of the program and look slightly different to the directives that have already been met.

Although not compulsory, explicit interfaces should be given for *all* external procedures. In any case, HPF programs should always use assumed-shape arrays which means that the supplication of interfaces will be mandatory. Explicit-shape arrays could also be used but these are not as flexible and are therefore not recommended. Assumed-size arrays (FORTRAN 77 style dummy array arguments that rely on sequence association should also be avoided.

(This means that for an assumed-size array element $A(i+1)$ occupies the previous memory location to $A(i)$, this is clearly not true for an HPF program; $A(i)$ and $A(i+1)$ may well be on separate processors - they could feasible even be on opposite sides of the world! HPF programs should not use any sequence associated objects.)

In general, it is not a good idea to remap data on entry to (or on exit from) a procedure. Data remapping is *extremely* time-consuming and will become a real bottleneck if it allowed. Data will be remapped if, for example, an actual argument is mapped with BLOCK distribution but the corresponding dummy is given CYCLIC distribution. Every processor will have to scatter all its elements to other processors and receive all its new elements from the processors that previously owned them.

When a procedure is exited, any dummy arguments which were remapped on entry must be remapped back again to their original mapping, in other words, a procedure call involving remapping does not permanently alter the mapping of an object, it just modifies it for the duration of the procedure.

223: 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(:,:)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P :: T
  ...
END SUBROUTINE Subby
```

Inside Subby the data will be mapped as specified.

Guide for slide: 223

In the subroutine **Subby**, the three dummy arguments **A**, **B** and **RES** will be mapped as specified in the HPF directives. If the actual arguments are not mapped like this at the call site then remapping will occur.

If an interface is to be given then the mapping directives which apply to the dummy arguments should be included.

224: 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.

Guide for slide: 224

Here, the directives are asserting that the distribution already is as specified in the directives. The *, when applied to an object or distribution method, says that this is how things already are. So

```
!HPF$ DISTRIBUTE *(BLOCK,BLOCK) ONTO *P :: T
```

says “the TEMPLATE T is already distributed (BLOCK,BLOCK) on to the processor grid P”. The compiler is bound to check that this is indeed the case and, if it is not, it must perform the relevant remapping to make it so. The HPFF have modified their language specification to say that if the mapping is not as asserted in the procedure then an explicit interface must be provided to allow remapping to take place.

Because these descriptive mapping assertions give the compiler knowledge about where data is located, it is recommended that this style of directives are used.

If an interface is to be given then, again, the mapping directives which apply to the dummy arguments should be included.

225: 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.

Guide for slide: 225

The last directive presents the strongest and most useful statement to the compiler.

226: 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.

Guide for slide: 226

Describing the mapping of dummy arguments is a complex area with HPF. Descriptive distributions should be used and interface should always be given.

It is unclear whether inherited mappings will ever become efficient enough to actually use — it currently seem a good idea to avoid them!

227: 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.

Guide for slide: 227

In order that the compiler can relate the mapping of one object to the mapping of another, it is good practise to supply a module containing global definitions of the PROCESSORS, TEMPLATES and their distribution methods. This module should be USED in every procedure and all dummy arguments should be aligned to the global template.

The cited example shows how the descriptive distribution directives in the procedure can now be simplified.

228: 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.

Guide for slide: 228

There is a potential naming problem here. Templates and processors are not Fortran 90 objects and so cannot appear in USE statements making them impossible to rename or USE ONLY.

229: 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.

Guide for slide: 229

It is a *very* good idea to include interfaces for all external procedures. These interfaces should obviously contain all the usual Fortran 90 information (especially the `INTENT`) but, in addition, should contain all relevant mapping information. Specifically they should contain:

- any modules containing global mapping information
- the mapping of any dummy arguments
- the mapping of any result variables (for `FUNCTIONs`)

230: 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 (:,:)
!HPF$ ALIGN (:,J) WITH A(J*2-1,:2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P :: A, B, Res
...
END SUBROUTINE Soobie
```

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

Guide for slide: 230

The example demonstrates how local objects may be aligned to dummy arguments.

231: 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.

Guide for slide: 231

Here the result of the function is aligned to one of the dummy arguments.

232: 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,

Guide for slide: 232

Clearly, the overhead of remapping one or more objects varies from system to system. Remapping across a procedure boundary on a shared memory multiprocessing system would take less time than remapping over a wide-area-network of distributed workstations, however, a rule of thumb should be that remapping should not take place unless *absolutely essential*.

An experiment was made with one of the commercial tools, the NA Software HPF Compiler v1.0. Two versions of the code given on the slide were compiled and executed on a Sun Multiprocessing system (using Unix sockets for communication). The version which involved remapping (from (BLOCK,BLOCK) to (CYCLIC,CYCLIC) and then back again) took 100 times longer to execute than the version which involved no remapping. This is a very significant amount.

Note how the procedure does not specify the INTENT of the dummy arguments. This will be seen to also have a bearing on the overhead of remapping.

233: 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**.

Guide for slide: 233

The example on the previous slide was modified slightly to include the argument intents. The version which involves remapping is now able to execute in just over half of the time, the reason for this is that only one remapping per argument is required instead of two.

`arg1`, which has `INTENT(IN)` is only remapped on procedure entry. The compiler is able to make this optimisation by creating a *copy* of the actual argument, remapping this copy (leaving the original in-place) and then simply discarding this copy (which has not been modified) at the end of the procedure. If the intent is not specified then the compiler has to ‘play-it-safe’ and cannot make this optimisation.

The purpose of this slide is to highlight just how important it is to specify the argument intent.

234: Passing Array Sections

Without using INHERIT, this is very complex. Consider

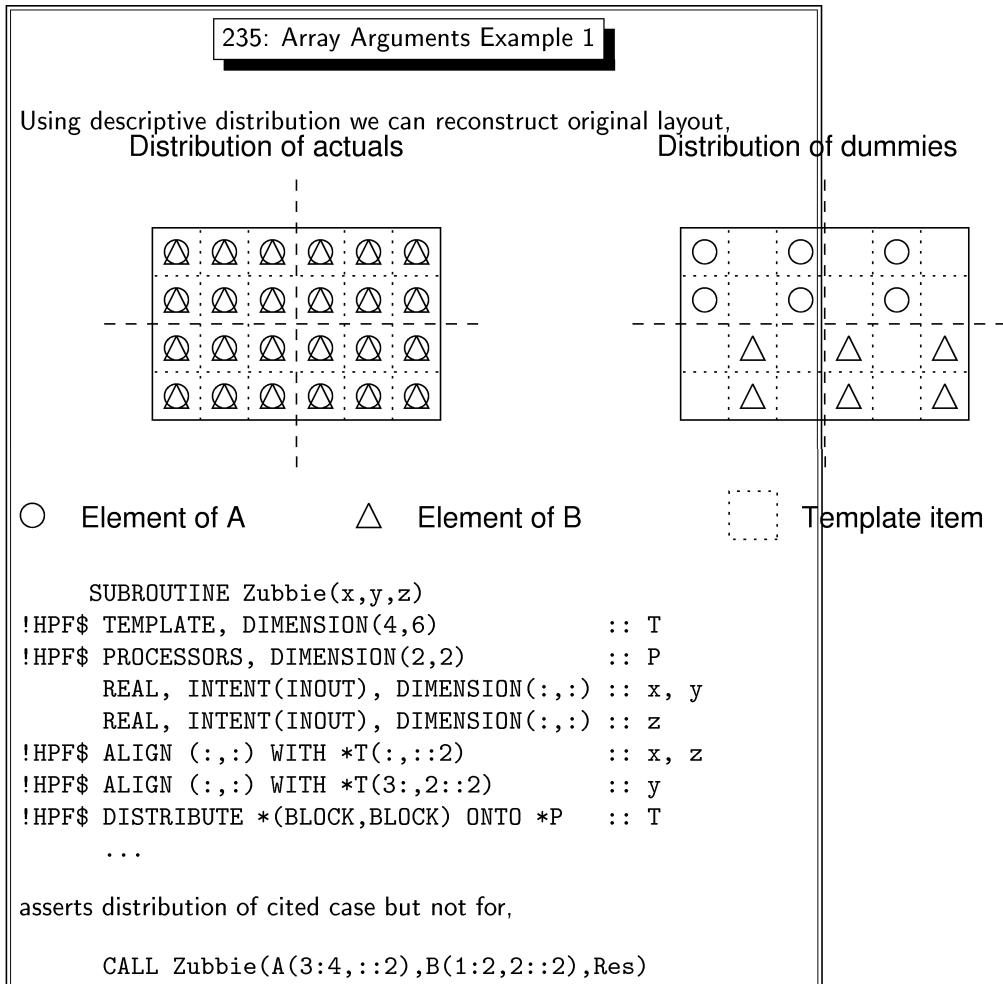
```
REAL, DIMENSION(4,6) :: A, B
REAL, DIMENSION(2,3) :: Res
!HPF$ PROCESSORS, DIMENSION(2,2) :: P
!HPF$ ALIGN B(:, :) WITH A(:, :)
!HPF$ ALIGN Res(:, :) WITH A(:, :, 2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P :: A, B, Res
...
CALL Zubbie(A(1:2,1::2),B(3:4,2::2),Res)
```

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.

Guide for slide: 234

The problem with passing array sections is how to describe the current mapping of the dummy arguments to the procedure. The cited example specifies an alignment between the actual arguments: `B` and `Res` and the align-target, `A`. In order to avoid any remapping when the subroutine `Zubbie` is invoked, the relationship (alignment) between its dummy arguments should be fully described. Specifying this alignment is going to be non-trivial due to the way in which the actual arguments are referenced (sectioned).



Guide for slide: 235

This assumes that **Zubbie** is invoked by the **CALL** statement given on the previous slide. The two diagrams show how the actual arguments, **A** and **B**, are aligned at the call site and how the two corresponding dummy arguments **X** and **Y** are aligned in the procedure. It can be seen that **A** and **B** are such that element **A(i,j)** is aligned with **B(i,j)**. It is perhaps best to view both **A** and **B** as being aligned to a conformable template.

The only way that the alignment of **x** and **y** can be described is by reconstructing this template and then explicitly aligning **X** and **Y** in exactly the same way as the two actual arguments were. In other words, we must calculate the alignment, with respect to the template, of the subsections of **A** and **B** that were used as arguments to **Zubbie**.

It can be seen that the alignment given in the procedure corresponds to the layout in the second diagram. By using the original template, we have been able to reconstruct the relative alignment of the dummies.

There is one large problem with this approach; what happens if **Zubbie** is invoked with different sections of **A** and **B** (such as that given at the bottom of the page)? Clearly, our carefully constructed alignment is no longer valid and, if it were used, would result in data remapping. If this approach is adopted then a separate subroutine for each invocation must be written. We get into even deeper water if the sectioning of **A** or **B** is performed by variables instead of literals.

It may be a good idea to only pass whole objects to procedures!

Note that the use of a module to hold the **PROCESSORS** and **TEMPLATE** declarations and the distribution of the template would simplify matters here.

236: 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.

Guide for slide: 236

Unless the HPF compiler is really smart¹, the above rewrite of **Zubbie** will generate two remappings. What will probably happen is that the compiler will remap the dummy arguments so that they are distributed (BLOCK,BLOCK). This will give the following distribution:

- P(1,1) owns X(1:2,1:2) and so on,
- P(2,1) owns X(3,1:2) and so on,
- P(1,2) owns X(1:2,3) and so on,
- P(2,2) owns X(3,3) and so on,

This means that all three arguments are remapped on entry and on exit which will be very inefficient.

¹which, currently, most are not!

237: 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,

Guide for slide: 237

This slide demonstrates a further example of ‘accidental’ remapping. Here the actual arguments are 1D sections of 2D objects. As the original objects are distributed over a 2D grid, the 1D dummy arguments will only be mapped over a 1D slice of the processor array. The problem arises because the procedure specifies that the dummy arguments are distributed over the default processor grid (which may well contain *all* the processors).

For example, if the system is executing on a 16-processor grid, which is either viewed as a linear chain of 16 processors or as a 4×4 square grid, (recall that all processor arrangements in a single HPF program must contain the same number of processors,) then $A(i,:)$ and $b(i,:)$ will be distributed onto a 4 processor slice (of the 4×4 grid,) but the dummy arguments x and y will be distributed over the whole 16-processor chain. Due to the lack of intent specification, two remappings will take place, one on entry and one on exit.

238: 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.

Guide for slide: 238

This slide demonstrates what happens if a single array item is used as an actual argument. The subroutine **Schmubbie** is expecting three dummy arguments, two scalars followed by an array. Taking the actual arguments in order, $A(1,1)$ is one element of a distributed object, in this case it will be mapped onto processor $P(1,1)$, the procedure specifies no mapping for the corresponding scalar dummy argument, this generally means that the argument should be replicated. This means that the value of $A(1,1)$ must be broadcast to *every* processor.

The second actual argument is a scalar so will already be replicated, the corresponding dummy is also scalar so will also be replicated. This will clearly involve no remapping. The third argument is a distributed array which will remain distributed and again will cause no remapping.

239: 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.

Guide for slide: 239

Each compiler may offer a different solution to the problem of passing array sections.

240: 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!

Guide for slide: 240

This approach could be one solution. Here, the HPF compiler has an extension which allows processor *subsets* to be used in a program. (Processor subsets are included in HPF2.) The problem is that the subsections $A(i,:)$ and $b(i,:)$ are only distributed over a (10 element) slice of the processor array but HPF will not allow this to be specified. We could reconstruct the original template and so on but this is too complex. A simpler approach may be to allow processor subsets, however, as we will see on the next slide, this can lead to more problems.

241: 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

Guide for slide: 241

HPF also specifies “*If two processor arrangements have the same shape, then corresponding elements of the two arrangements are understood to refer to the same abstract processor.*” (HPF Spec v1.1, section 3.7, P40.)

This means that the two processor arrays P_1 and P_2 refer to the same set of four processors. If we allow the extension of the previous slide, then the HPF semantics says that the dummy arguments x and y are mapped to the *same* 4-element processor array but in reality P_1 and P_2 are quite clearly *not* the same array.

It is much simpler to pass whole arrays!

Lecture 10:
Extrinsics, HPF Library and HPF in the Future

243: 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.

Guide for slide: 243

The EXTRINSIC keyword may be applied to a procedure name in an interface to indicate that it is non-HPF. It is up to a given compiler to support extrinsic procedures in other languages; the HPF specification does not require a compiler to support *any* extrinsics at all.

The example on the slide gives an example of how a C extrinsic may be declared. The parenthesised symbol which follows the EXTRINSIC keyword will be defined in the compiler manual and each compiler is likely to have a different notation for this keyword. Other supported keywords may be F77, FORTRAN, F77_LOCAL, PASCAL or ADA. If an extrinsic function is called, then its interface *must* be given at the call site. It is *very* important that this interface contains all mapping directives.

244: 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.

Guide for slide: 244

This is the INTERFACE for the example on the next slide. The reason for all the arguments is that when control is inside this procedure, the processor has no idea that it is inside an extrinsic procedure that was called from an HPF program. The processor must therefore be given all the information it needs in order to calculate how big the dummy array arguments are. (FORTRAN 77 is different from Fortran 90. The mechanism for passing arrays into procedures is much more primitive, the sizes of each dimension except the last must be supplied in the declaration of the dummy argument. The last dimension may be left as a *; if the actual extent of this dimension is required then it must be passed as an argument.)

It can be seen that, even though the extrinsic is written in FORTRAN 77, (the precursor to Fortran 90,) the interface is expressed in terms of HPF. As usual, the INTENT should be specified to aid the compiler in optimisation.

The mapping information is *very* very important as when control passes to the extrinsic, the compiler must make sure that the mapping specified in the interface is really the way that things are. If there were no mapping directives specified in the interface, the compiler will have to ensure that all the dummy arguments possess the default mapping which is usually replication. If this was the case, every time the extrinsic subroutine `Calc_u_like` is called, the array actual arguments would have to be remapped so that they were replicated on every processor in the grid.

The keyword `F77_LOCAL` is supported by the Portland Compiler, it allows extrinsics to be written in the FORTRAN 77 language. (The Portland Compiler also defines a set of message passing routines which allow different instances of the `F77_LOCAL` extrinsic to communicate with each other. Explanation of this style of programming is outside the scope of this course.)

It should be pointed out that most compilers will enforce a barrier synchronisation upon entry to and exit from an EXTRINSIC, this will cause an overhead.

245: 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.

Guide for slide: 245

These sorts of extrinsic functions are often called *local routines* because they only have a local picture of the machine that they are executing on. Such routines only operate on local data and may have no concept that they are part of a network of processors.

246: 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
```

Guide for slide: 246

Once inside the extrinsic, the first task is to calculate the portions of A, B and C that are local. This can be worked out from the original size of the matrices, the number of processors in each dimension of the HPF processor arrangement and the distribution method.

In the calling program, P_Nos has same number of elements as there are processors and is distributed blockwise so that each processor gets a single element of this array. It has been arranged that this element contains the number of the processor; the first processor in the chain is number 1, the second is number 2 and so on. Since the size of A and the total number of processors are given, each extrinsic can work out the number of array elements that it owns. The calculation of Blk_Siz works out the maximum number of elements that any processor may have; the calculation of My_Blk_Siz works out the size of the local set. (Recall that in some situations, the last processors in a chain may have less elements than the first processors on a chain.) Once the local set has been established, calculations may proceed as normal (using FORTRAN 77 syntax).

C routines will have to follow a similar method in order to establish their local sets. It should be mentioned that C arrays are stored row-wise so any arrays passed as dummy arguments will need to be transposed — this will generate a large overhead.

247: 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.

Guide for slide: 247

Some compilers, (but not Portland,) may support F90_LOCAL routines. If this is the case, then matters can be much simplified by using assumed-shape arrays. In this example, there is no need to calculate the local set; the Fortran 90 SIZE intrinsic could be used if this needs to be established.

Note how the extrinsic is relying on the sensible way in which Fortran 90 treats zero-sized arrays to remove the need for block-size calculations.

248: 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.

Guide for slide: 248

All these rules aim to make sure that the invocation of an EXTRINSIC is seamless. Guards are set against the extrinsic spawning processes which remain active after control has returned to HPF; if this was not enforced these processes could interact with execution in a destructive way.

The extrinsic should also leave the global state of the program in the same way as it found it. In other words, if any remapping takes place within the extrinsic, then it must be undone before control returns to HPF. The extrinsic must also ensure that scalar variables, which HPF goes to great lengths to ensure are equal across every processor, do not become corrupted.

249: Uses of Extrinsics

EXTRINSICs 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.

Guide for slide: 249

An extrinsic may be used to call any piece of code written in any language which performs any task as long as the compiler supports that interface.

Calls could be made to specialist sequential libraries such as the NAG FORTRAN 77 or Fortran 90 libraries.

250: 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.

Guide for slide: 250

One of the problems with current HPF compilers is the lack of support for INDEPENDENT DO loops. In general, the INDEPENDENT prefix is parsed and then ignored which means that INDEPENDENT loops are executed sequentially. It must be stressed that this situation is only temporary and will be resolved in the near future as tools mature.

As long as the INDEPENDENT DO loop does not involve any communication it is possible to simulate independent execution via the use of a cunningly written extrinsic procedure. The basic idea is to replace an INDEPENDENT DO loop with a call to a local extrinsic into which a modified version of the loop is placed. There is a small overhead as the extrinsic function will have to work out what the loop bounds should be but the net effect is that every extrinsic is able to execute its DO loop at the same time, that is, in parallel. One should bear in mind that a compiler will generate a barrier synchronisation at entry to and exit from an EXTRINSIC.

Clearly, the extrinsic must be supplied with enough information to work out which array elements it owns. This is trivial for a F90_LOCAL but a bit more tricky for a F77_LOCAL extrinsic.

251: 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.

Guide for slide: 251

This slide demonstrates the use of the NAg FORTRAN 77 library from within an HPF program. The cited example calls one of the simplest NAg library routines, X01AAF which returns the value of π . Clearly, a far more substantial routine could have been called from within the extrinsic instead.

The extrinsic does not need any mapping information because it does not reference any mapped objects.

All scalars in a program *must* have the same value. As Pi is a scalar routine each version of it must return the same value (which they do).

It must be ensured that the correct libraries are linked at compile time. Use the link editors' -l switch:

```
$ pghpf -Mnohpfc NagLib_Calc_u_like.f  
$ pghpf Extrinsic.hpf NagLib_Calc_u_like.o -lnag
```

252: New Intrinsic 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.

Guide for slide: 252

The first category of intrinsics adds two system inquiry procedures, `NUMBER_OF_PROCESSORS` and `PROCESSORS_SHAPE`, which have been met before. They can be used in declarations but **cannot** be used to initialise variables, for example,

```
INTEGER, DIMENSION(NUMBER_OF_PROCESSORS()) :: A
```

is OK but

```
INTEGER, PARAMETER :: NOP = NUMBER_OF_PROCESSORS()  
INTEGER, DIMENSION(NOP) :: A
```

is not (either with or without the `PARAMETER` attribute). This is because, even though the two intrinsics return a value that is constant for one run of a program, the value may differ during other runs.

The second category adds one new computational intrinsic and extends two others. The first procedures is `ILEN` which is an elemental function which returns the number of bits needed to store its `INTEGER` valued argument.

The `DIM=` specifier in the location intrinsics behaves exactly as it does with other intrinsics.

253: 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.

Guide for slide: 253

- Mapping inquiry functions

These routines are useful for HPF_LOCAL Exinsics. They allow a naive local run-time to discover how its arguments are mapped and what objects they are aligned with and distributed onto.

- Array reduction functions

These routines supplement the Fortran 90 reduction functions. Reduction functions work well in a parallel environment.

- Bit manipulation functions

These routines supplement the Fortran 90 bit functions. Three new functions are defined and can be used in conjunction with the existing procedures.

- Array combining scatter functions

These functions allow parallel execution when the LHS object in an array assignment statement is indexed by a vector subscript that has repeated values. Using the combining scatter functions allows ‘something sensible’ to be done when this situation is encountered.

- Prefix and suffix functions

These procedures provide an alternative to the Fortran 90 reduction intrinsics. These new functions can be used in the same situations as before but store the result in a different way!

- Array sorting functions

These allow multidimensional arrays to be graded (sorted) on specific keys.

254: 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.

Guide for slide: 254 **HPF_ALIGNMENT,**

Returns information about the alignment of an argument. It is possible to find out: the upper and lower bounds and the stride that were used in the alignment; if and how the axes of the alignee and the align-target are permuted; whether any axes are reversed and the effect of any replication that may have been specified.

 HPF_DISTRIBUTION,

Returns information about the distribution of an array or template argument. It is possible to find out: the distribution method of each axis; the block-size in each axis and the rank and shape of the grid that an array or template is distributed onto.

 HPF_TEMPLATE,

Returns information about the array or template to which an object is aligned. It is possible to find out: the rank of the align-target to which the argument is aligned; the upper and lower bounds of the align-target to which the argument is aligned and the total number of objects aligned with the align-target.

255: 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.

Guide for slide: 255

The array DISTS must be declared in such a way that its result can be understood. There are 3 possible values that the function can return in its AXIS_TYPE argument: BLOCK, CYCLIC and COLLAPSED; this means that in order to return a unique result DISTS must be at least 2 characters long.

BLK_SIZE(2) is not stated because the second dimension of A has been collapsed — its value is compiler dependent.

256: New HPF Reduction Functions

There are four FUNCTIONS 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.

Guide for slide: 256

Like all reductions, each function has optional **DIM** and **MASK** arguments. These work in exactly the same way as for the **SUM** and **PRODUCT** reductions.

IALL, **IANY** and **IPARITY** are all bit manipulation functions. **PARITY** operates on logical values. All are elemental.

257: 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..

Guide for slide: 257

IANY(A) is the same as

... (IOR(IOR(IOR(A(1),A(2)),A(3)),A(4)),...)

IPARITY(A) is the same as

... (IEOR(IEOR(IEOR(A(1),A(2)),A(3)),A(4)),...)

258: 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.

Guide for slide: 258

These functions report on the bit patterns of INTEGER variables. The Fortran 90 Standard (Section 13.5.7) explains how integers are interpreted. Consider the artificial binary number $I = 00001011$,

- `LEADZ(I)` is 4
- `POPCNT(I)` is 3
- `POPPAR(I)` is 1 (the alternative is 0)

Again, all of these functions are elemental. The result is always of the same kind as the argument.

259: 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)
```

Guide for slide: 259

The assignment:

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

is not legal Fortran 90 due to the non-uniqueness of the elements of W . The statement is asking for the two assignments:

$$\begin{aligned} A(2) &= A(2) + 2*B(2) \\ A(2) &= A(2) + 2*B(3) \end{aligned}$$

to be made in parallel which is clearly impossible. The combining scatter functions allow parallel assignments to the same array element to be performed in a structured (defined) way; these functions state how two or more expressions that are supposed to be assigned to the same result element should be combined. The example given here adds the two expressions together *before* making the assignment. There are other options which are outlined on the next slide.

260: 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/).

Guide for slide: 260

The PRODUCT_SCATTER example performs:

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

in parallel.

In the MINVAL_SCATTER example, the index vector is now (/2,2,1,1/), the vector being assigned to is (/1,1,1/) and the vector that is being indexed is (/10,-2,4,2/).

Working along the index vector, the result of MINVAL((/10,-2,1/)) is assigned to position 2 (10 and -2 are the first two elements of (/10,-2,4,2/) and the 1 is from the original array being assigned to).

Likewise, MINVAL((/4,2,1/)) is placed in position 1; position 3 is unaltered as it is not indexed.

values to be assigned	10	-2	4	2
indices to be assigned to	2	2	1	1

261: 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/))
```

Guide for slide: 261

Prefix functions go from left to right along an array. The difference between PRODUCT_PREFIX and PRODUCT is that the latter returns a single value whereas the former returns an array result. This array contains the value obtained by multiplying the current element by all the elements to the left of it. In the case of a 2D (or more) example, the array is traversed in array element order.

Suffix functions are exactly the same except that the array is traversed from right to left (or for multi-dimensional arrays in reverse array element order).

262: 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.

Guide for slide: 262

There are prefix and suffix functions corresponding to *all* the Fortran 90 and new HPF reduction functions. They all work in the same way and have more or less the same set of arguments.

263: 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/)`

Guide for slide: 263

Only the elements corresponding to .TRUE. positions of the mask contribute to the results, the positions of the source array which correspond to the .FALSE. mask elements are ignored.

264: 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/)
```

Guide for slide: 264

Optional Slide

When both a MASK and a SEGMENT argument are supplied the result is partitioned and the function only applied to the .TRUE. elements of each partition.

265: 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.

Guide for slide: 265

These functions do not perform ‘in-place’ sorts but instead return a set of indices (a permutation of the original) which define the sorted array.

The result is always a 2D array with the first row being all the x-coordinates, the second row being all the y-coordinates, the third row being all the z-coordinates and so on.

266: Example of Array Sorting Functions

Given,

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

then $\text{GRADE_DOWN}(A)$ is the 2D (1×11) array:

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

and

$\text{GRADE_UP}(A, \text{DIM}=1)$ 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 $\text{DIM}=$ has shape

$(/ \text{SIZE}(\text{SHAPE}(A)), \text{PRODUCT}(\text{SHAPE}(A))/)$

Otherwise the shape is the same as A.

Guide for slide: 266

The result of these functions is always a 2D array (of coordinates) unless the `DIM=` specifier is used in which case the result is the same shape as array.

Elements of the same value are ordered by their position in array element order. For the repeated value of 5 (which occurs four times in the source array), the result contains element 7 before element 8 which comes before element 10, and so on.

267: 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
```

Guide for slide: 267

The result of `GRADE_DOWN(A)` is an array containing the coordinates (1,2), (2,2), (2,1) etc. Since A is a 2D array of 9 elements the result has 9 coordinates, ie 18 elements for that column.

The result `GRADE_DOWN(A,DIM=1)` is three columns, each containing the indices of the sorted elements.

268: 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 retying of memory does not mix well with distributed objects.

Things to avoid,

- distributing arrays in COMMON,
- distributing EQUIVALENCEd arrays,
- assumed-size arrays.

Guide for slide: 268

Note: the distribution of sequence and storage associated objects is now not part of the new HPF standard, HPF 2.0.

One of the hardest tasks of the HPFF was to try to marry FORTRAN 77 style sequence and storage association with the HPF concept of distributed objects. Many of the largest and most computationally intensive research programs that would benefit the most from conversion to HPF will contain at least some sequence or storage association. In HPF 1 High Performance Fortran compilers were bound to implement Chapter 7 of the HPF Specification; now they are not!

Due to the separate compilation nature of FORTRAN 77, it was (and indeed still is) possible to perform many dirty tricks with arrays when used as procedure arguments. For example, in FORTRAN 77 programs it was possible, and common, to use assumed-size arrays to change the shape of an array across a procedure boundary:

```
PROGRAM MAIN
REAL A(10,10)
...
CALL SUBBO(A)

SUBROUTINE SUBBO(X)
REAL X(*)
...
```

As a FORTRAN 77 compiler cannot necessarily see `SUBBO` when compiling `MAIN`, it had to rely on the fact that the whole of the actual argument `A` is stored as a contiguous block in memory. The assumption that memory is linear is of course not the case for a distributed object on a distributed memory machine. Given that Fortran program units can be separately compiled, it can be seen that supporting legacy FORTRAN 77 codes could prove to be a taxing problem.

There are other problems too, especially in the area of storage association, `EQUIVALENCE` and `COMMON` blocks. `EQUIVALENCE` can be used to change the type and variable name that a particular area of memory is referred through; `COMMON` blocks provide a mechanism to access global data. A `COMMON` block reserves a contiguous area of memory which is accessed through the specified names. Each instance of a common block can be different which means that the partitioning of the memory specified by the block can vary from routine to routine. For example, the following `COMMON` block could appear in different procedures in a *separately compiled* Fortran program:

```
COMMON /BLEURGH/ X(10,10),Y(2,22),I(100)
...
COMMON /BLEURGH/ Z(12,12),J(10,10)
...
COMMON /BLEURGH/ I(44),J(100),X(2,50)
```

If any of the arrays in the `COMMON` blocks are to be distributed then an HPF compiler will have a problem! It is highly likely that, without prior warning, the arrays from these `COMMON` blocks would be distributed using the default mapping as a compiler is at liberty to supply a default (or implicit) mapping for arrays without explicit mapping directives. This mapping is typically `*` (replication).

The HPFF have defined some additions to Fortran 90 to try to lever sequence and storage association back into the arena. It is possible to categorise two distinct situations, one where objects

used in sequence and storage associations can be used as distributed objects and the second where they cannot. In general, if a **COMMON** block is used in the same way on every occurrence then it is possible to distribute its elements. (After the abovementioned **COMMON** block problem was recognised it was considered good programming style to place each **COMMON** block in an **INCLUDE** file and to attach this file whenever access to the **COMMON** block was desired. This would ensure that every use was forced to be the same.)

There is a **SEQUENCE** directive which can be applied either in general or to a named entity (variable name or **COMMON** block). This will mean that the named entities are stored sequentially and it will also allow **COMMON** blocks such as **BLEURGH** to be used safely in HPF programs. The **SEQUENCE** directive must be applied to every instance of **BLEURGH**.

```
!HPF$ SEQUENCE :: /BLEURGH/
```

The HPFF have introduced the concept of covers and aggregates. In Fortran it is possible to use a **COMMON** or **EQUIVALENCE** statement to string together groups of variables by overlapping storage, a very simple example would be:

```
REAL A(2), B(2), C(2)
EQUIVALENCE (A(2),B(1))
EQUIVALENCE (B(2),C(1))
```

here **A** and **C** are linked by **B** so must follow each other in memory. This group of overlapping variables is called an *aggregate variable group*. We can define a *cover* for this group in a further **EQUIVALENCE** statement:

```
REAL COVER(4)
EQUIVALENCE(COVER(1),A(1))
```

The aggregate variable group are part of a single storage sequence (aliased as **COVER**) of size 4. Neither **A**, **B** or **C** can be mapped but **COVER**, as long as it is one dimensional, can be.

Some variables are automatically sequential (ie they implicitly have the **SEQUENCE** attribute). A variable is sequential if it is an assumed size array, if it appears in a sequential common block, is a member of an aggregate variable group, if it is a derived type with the **SEQUENCE** attribute or is given the HPF **SEQUENCE** attribute. A sequential variable can be storage or sequence associated ; nonsequential variables cannot.

A sequential variable cannot be mapped unless it is a scalar or a rank one array that is an aggregate cover. This means that an assumed sized array cannot be distributed as it can never be an aggregate cover.

Recent developments at the HPFF have indicated that due to the complexity and non-usefulness of the mapping of sequential variables, it looks likely that the next version of HPF, HPF2, will not allow such mappings. It will be defined as a language extension!

269: Dual Fortran 90 and HPF Codes

There 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.

Guide for slide: 269

The HPF directives are comments so are ignored by Fortran 90 compilers. The other main additions to Fortran 90 syntax are **PURE**, **FORALL**, **EXTRINSIC** and the functions from the HPF Library module. Now both **PURE** and **FORALL** are included in Fortran 95 and are starting to appear in new Fortran compilers already, the HPF Library module is available as a Fortran 90 **MODULE** accessible via the HPF Web Site given earlier so the only major feature that is absent from Fortran are **EXTRINSIC** procedures (x3j3/WG5 seem to be going in another direction). It should be possible to mask out **EXTRINSIC** references by using a preprocessor such as **cpp** or **fpp**.

Pointers and passing array sections should also be avoided as neither feature is HPF-friendly! The previous slide has highlighted the problems of sequence and storage association which should also be avoided.

270: 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!

Guide for slide: 270

This course has mainly covered the original Subset HPF although some features of the full language, such as **FORALL** constructs and **PURE** procedures, have been explained. The main areas of Full HPF that have not been mentioned are:

- dynamic mappings, **REALIGN**, **REDISTRIBUTE**,
- inherited distributions,
- distributed derived types (not well defined),
- pointers

In the main, none of these features are supported by HPF compilers and in any case, the first two bullet points are thought to reduce the efficiency of HPF code. HPFF have realised that details about the mapping of derived types are a bit thin on the ground and have drafted out a better explanation. Pointers have also been looked at and will be discussed at a much greater length in the next HPF standard document.

271: 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 pghpff takes 33s!

Guide for slide: 271

272: 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 intrinsics (C, FORTRAN 77, HPF_CRAFT and FORTRAN 77 local library).

Guide for slide: 272

The first draft of a new HPF standard, known as HPF2.0

<ftp://titan.cs.rice.edu/public/HPFF/hpf2/hpf-report.ps>

was issued in October 1996. It has been drafted in the light of both user and vendor experience with the original HPF specification.

A number of problems were identified with the first specification, these fell in two areas,

1. missing functionality,
2. inefficient language features.

The new specification has addressed both these areas. Some features have been moved out of the language but have been kept as “approved extensions”; a large number of new features have been introduced, most of these are designated as approved extensions, but a small number have actually been added to the language itself.

In addition to this, HPF2.0 is not partitioned in Full and Subset features and is now bound to Fortran 95 (as opposed to Fortran 90) so features such as FORALL statements and constructs, PURE functions and the extensions to MINLOC and MAXLOC are no longer detailed in the HPF specification. In addition, Fortran 95 contains the nested WHERE construct which is guaranteed to tie the brain in knots!

There follows a brief description of the changes.

Inefficient features are have been removed from the language and placed in the list of approved extensions and a few new features have been added.

There are no new Data Mapping features.

- REDISTRIBUTE and REALIGN are deemed too inefficient to be in the main language,
- the DYNAMIC attribute which is related to the above,

In addition to these the rules have been changed regarding the INHERIT directive and explicit interfaces and now required if remapping occurs across a procedure boundary. It is also not permitted to map sequential arrays (eg, assumed-size arrays).

- general block GEN_BLOCK distribution,
Allows explicit specification of blocksizes. The idea is that an integer array is able to hold the size of block that each processor receives. This feature gives much more control over load balancing.
- INDIRECT mappings
Allows an INTEGER vector to hold the number of the processor that a particular array element is mapped to. This sort of feature is useful in finite element codes, it allows non-adjacent array elements to be mapped to the same processor without having to re-order the array.

RANGE directive:

This directive can appear in procedures and specifies the range of possible mappings that a dummy argument may hold. It is used in conjunction with transcriptive mapping specifications (**INHERITED** mappings) of **DYNAMIC** objects.

PROCESSORS subsets:

Can now map object to a subset of the total number of processors. This may aid efficiency and is also used with the new tasking features of the language.

SHADOW directive:

Allows the specification of a **SHADOW** or halo region to be applied to a mapped object. This is particularly useful for **BLOCK** distributions which access neighbouring cells. Elements that lie on the border between two processors will always generate communications when they are being updated. If there is a shadow region, an off-block element may be duplicated and a local copy stored in order to promote efficiency. The size of the shadow region specifies how many extra “non-local” elements are stored locally. The shadow regions are updated at a later date by the compiler.

Many compilers already implement this feature by default but sometimes they are not clever enough to decide what the size of the region should be.

objects with the **POINTER** attribute can be mapped.

derived type components can now be mapped.

The change to Fortran 95 has meant that **FORALL** is now not part of the HPF specification.

REDUCTION directive:

An extension to the **INDEPENDENT** directive which states that the following **DO** loop contains a reduction operation. This will allow efficient generation of communications patterns as it maps well to the reduction routines that are found in all message passing subsystems.

extensions to **EXTRINSIC** syntax:

The method of declaring an extrinsic has been extended and extrinsic **PROGRAMs**, **MODULEs** and **BLOCK DATA** subprograms can now be specified. **HPF**, **C** and **FORTRAN** are defined as extrinsic keywords.

There are no features in this category.

subsets of processors and **ON** statement:

It will be possible to distribute objects onto subsets of processors and to specify that a particular assignment is executed on a particular processor.

ON blocks:

Allows the compiler to be informed of the best location to perform a computation, in effect, the **ON** directive allows the “owner-computes rule” to be overridden. The **ON** directive may specify a single processor or a subset of processors (which introduces the concept of “active processor set”) and may also include a **HOME** expression to denote that the processor(s) which owns a particular element (or set of elements) should perform the calculation.

□ **RESIDENT** directive/clause:

The **RESIDENT** clause can be used in conjunction with the **ON** directive and identifies variables that are resident on the current active processor set. In other words, all the objects listed in the **RESIDENT** clause are local to the processor subset specified by the surrounding **ON** directive; processors absent from this subset will not have to partake in any communication required by the calculations in the **ON** block — they can perform an entirely separate task.

□ **TASK_REGION** blocks:

The provision of **TASK_REGION** blocks allows parallel sections (two disjoint subsets of processors performing different tasks at the same time), nested parallelism (nested **TASK_REGIONS** can further subdivide an active processor set to form a tree of independent tasks) and data parallel pipelines (the results from one processor subset can be piped into a different disjoint set thus forming a pipeline).

□ new library procedures:

SORT_UP and **SORT_DOWN** procedures are added to the library. Some procedures have been slightly modified.

□ new intrinsic procedures

A generalised (ie, multi-dimensional) **TRANSPOSE** has been introduced which accepts an input array and a 1-D vector that specifies the axis permutation.

In order that the active processor set can be determined during execution, two new intrinsics, **ACTIVE_NUM_PROCS** and **ACTIVE_PROCS_SHAPE** have been added. Additions to the existing mapping inquiry procedures have also been made to take account of the new distribution methods.

□ asynchronous I/O

Allow I/O to be performed whilst the program is executing. An extension to the **OPEN** statement configures a particular unit for asynchronous transfer and a new command, **WAIT**, forces execution to suspend until the previously invoked I/O has terminated.

□ recognised externally supported HPF Extrinsics: The HPFF have recognised two interfaces to extrinsic mechanisms; these are **HPF_CRAFT** and FORTRAN 77 local library. The recognition of these interfaces also highlights the mechanism whereby an extrinsic interface may be submitted to the HPFF for official recognition. There now exists a standardised method of calling **HPF_CRAFT** or FORTRAN 77 local library routines from within HPF.

273: HPF Kernel

'Guaranteed to execute fast and be portable'.

- no INHERIT.
- no pointers,
- no dynamic mapping,
- only certain types of alignment, not strided.

Guide for slide: 273

The original intention was to define a kernel in second HPF specification; this plan was abandoned.