Technical University of Denmark

DANISH CENTER FOR APPLIED MATHEMATICS AND MECHANICS

High Performance Computing

FORTRAN, OpenMP and MPI

41391

Content

- Day 3:
  - Array features
    - Zero size array.
    - Assumed size /Assumed shape arrays.
    - Automatic/ALLOCATABLE/POINTER arrays.
    - ALLOCATE/DEALLOCATE/NULLIFY.
    - FORALL, WHERE.
    - PURE/ELEMENTAL procedures.

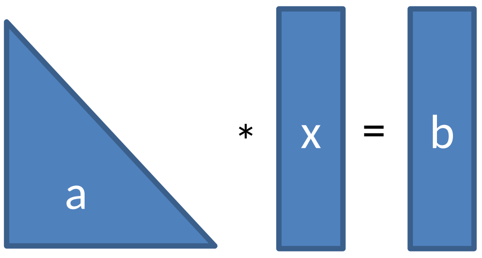
Zero size arrays

- FORTRAN allows zero size arrays:

  Example: solve lower-triangular set of linear equations: a*x = b:

```
DO i=1,n
    x(i) = b(i)/a(i,i)
    b(i+1:n) = b(i+1:n) – a(i+1:n,i)*x(i)
ENDDO
```

- The subsection of b has zero length when i = n



Zero size arrays

- Double DO version (preferred solution):

```
DO i=1,n
    x(i) = b(i)/a(i,i)
    DO j=i+1,n
        b(j) = b(j) – a(j,i)*x(i)
    ENDDO
ENDDO
```

Notice: for i=n we have:
DO j=n+1,n which will not enter the loop

Assumed-size arrays

- The size of arrays passed to subprograms can be passed along with the array (FORTRAN 77 style):

  Example:

  ```
  SUBROUTINE SUB(A,lda,ni,nj)
  INTEGER, INTENT(IN) :: lda ! Leading dimension(s)
  INTEGER, INTENT(IN) :: ni,nj
  REAL, DIMENSION(lda,*) :: A ! Last dimension can be left as: *
  A = 0 ! Is illegal – since A is not assumed-shape (:,:) style
  DO j=1,nj
      DO i=1,ni
          A(i,j) = … ! This is legal
  ```

Assumed-size arrays

```
PROGRAM main
REAL, DIMENSION(7,5) :: field
CALL sub(field(3,4),7,3,2)
… and in the subroutine:
SUBROUTINE sub(A,lda,ni,nj)
INTEGER :: lda,ni,nj
REAL, DIMENSION(lda,*) :: A
DO j=1,nj
   DO i=1,ni
      A(i,j) = …
! will access the elements: field(3:5,4:5)
```

Assumed-shape arrays

- For assumed-shape arrays (:-notation) the size of the arrays passed to subprogram can be inquired by the subprogram (but requires an INTERFACE).
- Example:

```
program main
real, dimension(300,300) :: matrix
interface
    subroutine sub(array)
    real, dimension(:,:) :: array
    end subroutine sub
end interface
call sub(matrix)
…
```

- The interface only passes the SHAPE – not the lower and upper bounds. Thus if the *actual* argument has the DIMENSION(-1,10) the *dummy* argument will be DIMENSION(1:12).

Assumed-shape arrays

Example:

SUBROUTINE SUB(field)

REAL, DIMENSION(:,:) [,POINTER] :: field

PRINT*,'LBOUND(field) = ',LBOUND(field)

PRINT*,'UBOUND(field) = ',UBOUND(field)

- A POINTER array is NOT considered assumed-shape, and if the actual and dummy arguments are given the POINTER or ALLOCATABLE attribute the print will return the bounds of the actual argument.
- If the dummy argument is an assumed-shape (non-POINTER/ALLOCATABLE) the LBOUND will return 1,1 and the UBOUND the total extend of the array.

Assumed-shape arrays

```
program main
implicit none
real, dimension(-1:10) :: data
integer :: i
interface
   subroutine sub(vector)
   real, dimension(:) :: vector
   end subroutine sub
end interface

call sub(data)
do i=-1,4
   print*,'data(i) = ',i,data(i)
enddo
end program main

subroutine sub(vector)
implicit none
real, dimension(:) :: vector
integer :: i

do i=lbound(vector,1),ubound(vector,1)
   vector(i) = i
enddo
return
end subroutine sub
```

Will print:

```
-1 1.00000
 0 2.00000
 1 3.00000
 2 4.00000
 3 5.00000
 4 6.00000
```

Automatic arrays

- Arrays can be declared in subprograms based on dummy arguments (so-called *automatic* arrays):

  Example:

  ```
  SUBROUTINE swap(a,b)
  REAL, DIMENSION(:), INTENT(INOUT) :: a,b
  REAL, DIMENSION(SIZE(a)) :: work ! Will be allocated on the stack !
  REAL, DIMENSION(100000) :: aux ! Will be allocated on the stack !
  work = a; a = b; b = work
  END SUBROUTINE swap
  ```

DO NOT USE AUTOMATIC ARRAYS!
It may/will crash during:
1) **The declaration of the work array**
2) **or/and during the assignment**

Stack

- Automatic arrays are allocated on the *stack* – **without** a return / error status.
- Local arrays are (normally) allocated on the stack:
  - Local static arrays has no return status.
  - ALLOCATE of local POINTERs/ALLOCABLEs returns a status.
- The stack is usually of limited size (try the UNIX command: `ulimit -a;` gbar: ca. 10MB)
- Advice: place the subprogram in a module and place the local arrays in the module holding the subprogram – then stored on the heap.

Local arrays

Example: local arrays allocated on the stack:

```
SUBROUTINE swap(a,b)
REAL, DIMENSION(:), POINTER :: work ! Local array
REAL, DIMENSION(:), INTENT(INOUT) :: a,b
! PRO: ALLOCATE will at give return status;
! CON: the work array will/may be allocated on the (small)
! stack and easily fail !
ALLOCATE(work(SIZE(a)),STAT=info)
work = a; a = b; b = work
DEALLOCATE(work,STAT=info)
END SUBROUTINE swap
```

Local arrays

Example: always place local arrays into the module!

```
MODULE m_swap
REAL, DIMENSION(:), POINTER :: work
CONTAINS
  SUBROUTINE swap(a,b)
  REAL, DIMENSION(:), INTENT(INOUT) :: a,b
  ALLOCATE(work(SIZE(a)),STAT=info)
  work = a; a = b; b = work
  DEALLOCATE(work,STAT=info)
  END SUBROUTINE swap
END MODULE m_swap
```

Will crash if size(a).NE.size(b)
So: check the size of a and b
and decide what to do !
then use DO loops for the copy !
You can also keep the work array
allocated to save time if you call
the swap routine repeatedly.

Allocatable arrays

- The *rank* of an allocatable array is specified when it is defined.
- The *bounds* of the array are undefined until the ALLOCATE statement:

  Example:

```
MODULE m_swap
    REAL, DIMENSION(:), ALLOCATABLE :: work
    CONTAINS
    SUBROUTINE swap(a,b)
    REAL, DIMENSION(:), INTENT(INOUT) :: a,b
    ALLOCATE(work(SIZE(a)),STAT=info)
```

Re-allocatable arrays

- ALLOCATABLE (or POINTER) arrays cannot be *reallocated* (grow/shrink while preserving the content of the array) using a single call.
- To reallocate an array (A) do:
  - create a work array (W) of the size of A.
  - copy the data from A to W.
  - deallocate A.
  - allocate A with the new required size.
  - copy the data from the W to A.
  - deallocate W.

Re-allocatable arrays

- OR: to reallocate an (POINTER) array (A) do:
  - create work array (W) of the new desired size of A.
  - copy the data from A to W.
  - deallocate A.
  - let A point to W (saving the last copy back !)
- OR: use MOVE_ALLOC() for allocatable arrays:
  - create work array (W) of new desired size of A.
  - copy the data from A to W.
  - call move_alloc(W,A)

The ALLOCATE statement

- ALLOCATE(allocate-list,[STAT=stat])
  - allocate-list: allocate-object[(array-bounds-list)]
  - array-bounds-list: [lower-bounds:]upper-bounds

  Example:

  REAL, DIMENSION(:,:), ALLOCATABLE :: array1

  REAL, DIMENSION(:,:), POINTER :: array2

  INTEGER :: istat

  ALLOCATE(array1(-10:5,5),STAT=istat)

  ALLOCATE(array2(100,100),STAT=istat)

The ALLOCATE statement

- The allocate-object can be an ALLOCATABLE or POINTER array.
- Each lower/upper-bound is a scalar integer expression.
- The default lower-bound is 1.
- The number of array bounds in a list must match the rank of allocate-object.
- The STAT returns zero on success.
- If the STAT is absent and the allocation is unsuccessful, the program execution will stop.
- **ADVICE: ALWAYS use STAT and ALWAYS check the value of STAT !**

The ALLOCATE statement

- During the call to ALLOCATE the array boundary is undefined, thus:
  - ALLOCATE(a(SIZE(b)),b(size(a))) ! Is illegal
  - ALLOCATE(a(n),b(SIZE(a))) ! Is illegal
  - ALLOCATE(a(n)); ALLOCATE(b(SIZE(a))) ! Is legal
- To allocate an already allocated
  - ALLOCATABLE array is illegal.
  - POINTER array is legal (but leave the previous target inaccessible!).

The DEALLOCATE statement

- DEALLOCATE(allocate-object-list[,STAT=stat])

  Example:

  REAL, DIMENSION(:,:), ALLOCATABLE :: array1

  REAL, DIMENSION(:,:), POINTER :: array2

  INTEGER :: istat

  DEALLOCATE(array1,STAT=istat)

  DEALLOCATE(array2,STAT=istat)

  ! Or DEALLOCATE(array1,array2,STAT=istat)

The DEALLOCATE statement

- The STAT returns zero on success.
- If the STAT is absent and the deallocation is unsuccessful, the program execution will stop.
- **ADVICE: ALWAYS use STAT and ALWAYS check the value of STAT !**
- There must be no dependencies in the list of objects to allow a one-by-one deallocation.
- If an array is deallocated **other** pointers to the array are left *dangling* !

The NULLIFY statement

- NULLIFY(pointer-object-list)
  - Disassociates a pointer from its target.
  - Null pointer is different than undefined (dangling) and nullified pointers can be tested by the intrinsic routine: ASSOCIATED().
  - Nullify does not deallocate the target (if deallocation is needed DEALLOCATE should be used).

The WHERE statement

- The WHERE statement performs array operations for certain elements only:

  [name:] WHERE (*logical-array-expr*)

  array-variable=expr

  [ELSE WHERE]

  array-variable=expr

  END WHERE [name]

  Example:

  WHERE (a > 0.0) a = 1.0/a ! a is a real matrix

DO NOT USE IT!

1) **It is NOT more efficient that the DO.**
2) **It required the SHAPE to be identical for all arrays.**
3) **It may allocate (hidden) extra memory! (and crash in the process).**

The FORALL statement

- FORALL is like DO but does not require order:

  [*name*:] FORALL *(index=lower:upper[:stride]* &

  [,*index=lower:upper[:stride]* …

  [,*scalar-local-expr*])

  [*body*]

  END FORALL [*name*]

  Example:

```
FORALL (i=1:N)
    A(i) = 1.0
END FORALL
```

Example:

```
FORALL (i=1:n)
    WHERE (a(i,:) == 0) a(i,:)=i
    b(i,:)=i/a(i,:)
END FORALL
```

PURE procedures

- A procedure (subroutine/function) is PURE if:
  - A *function* does not alter any dummy arguments.
  - It does not alter any part of a variable accessed by host or use association (through module data).
  - It contains no local variables with the SAVE attribute.
  - Performs no operation on an external file.
  - Contains no stop statement.
- The compiler will produce an error if these rules are violated.

PURE procedures

- To assert that a procedure has no side-effects add the PURE keyword to the subroutine or function statement.

  Example:

  PURE FUNCTION distance(p,q)
- An interface block is required if a pure procedure is to be used as pure (i.e. pure procedures can also be used without an interface).

ELEMENTAL procedures

- Intrinsic function are *elemental* – i.e. the result is given the shape of the input.

  Example:

  REAL :: scalar

  REAL, DIMENSION(100) :: vector

  scalar = 1.0; scalar = SQRT(scalar)

  vector(1:100) = 3.0; vector = SQRT(vector)

ELEMENTAL procedures

- Non-intrinsic procedures can be *elemental* if:
  1. The declaration contains the prefix ELEMENTAL.
  2. All the arguments are conformable (same shape).
  3. The procedure is PURE.
  4. All dummy arguments and function results must be **scalar** variables without the pointer attribute.

Example:

```
ELEMENTAL SUBROUTINE swap(a,b)
REAL, INTENT(INOUT) :: a,b
REAL :: work
work = a; a = b; b = work
END SUBROUTINE swap
```

This swap routine can now be called with any REAL scalar or array data !

ELEMENTAL procedures

```
PROGRAM main
INTERFACE swap
   ELEMENTAL SUBROUTINE swap(a,b)
   REAL, INTENT(INOUT) :: a,b
   END SUBROUTINE swap
END INTERFACE swap
REAL                  :: sx,sy
REAL, DIMENSION(2) :: vx,vy
sx = 1.0;  sy = 3.0
vx(1) = 1.0; vx(2) = 2.0
vy(1) = -23.; vy(2) = -345
CALL swap(sx,sy) ! We can call swap() with any rank !
CALL swap(vx,vy) ! But we need overloading for different KIND !
END PROGRAM main
```

ELEMENTAL procedures are EXTREMELY powerful !!
AGAIN: be careful: lots of things are going on BEHIND the SCENE and ELEMENTAL functions will be SLOWER than ordinary overloaded functions

Explicit overloading

```
MODULE oswap
    REAL :: swork ! Work array for scalar swap
    REAL, DIMENSION(:), ALLOCATABLE :: vwork ! Work array for vector swap
    INTERFACE swap
      MODULE PROCEDURE sswap, vswap
    END INTERFACE  swap
    CONTAINS
      SUBROUTINE sswap(a,b) ! Scalar version
      REAL :: a,b
      swork  = a; a = b; b = swork
      END SUBROUTINE sswap
      SUBROUTINE vswap(a,b) ! Vector version
      REAL, DIMENSION(:) :: a,b
      ALLOCATE(vwork(SIZE(a)))
      vwork = a; a = b; b = vvwork
      DEALLOCATE(vwork)
      END SUBROUTINE vswap
END MODULE oswap
```

A safe alternative to ELEMENTAL procedures:
use explicit overloading !

ELEMENTAL procedures

- If a generic procedure reference is consistent with BOTH an elemental and a non-elemental procedure, the non-elemental procedure is invoked.
- Thus we can write efficient specific versions and leave the ELEMENTAL to the general case!

Array sub-objects

- Subsections of an array may be extracted by:

  a(i,1:n) ! Elements 1 to n of row i

  a(1:m,j) ! Elements 1 to m of column j

  a(i,:) ! The whole row i

  a(i,1:n:3)! Elements 1,4,… of row i

  v((/1,7,3,2/)) ! Is a vector of length 4 with the elements: v(1), v(7), v(3), v(2)

Array sub-objects

equivalently:

v(list), where

list(1) = 1; list(2) = 7

list(3) = 3; list(4) = 2

- Vector subscript must be unique when appearing on the LHS:

v(/1,7,3,7/) = (/1,2,3,4/) ! Is illegal

since 2 and 4 cannot both be stored in 7

Array sub-objects

- Subscript: lower:upper:stride
  - Lower and upper may exceed the range of the array
  - Stride has to be non-zero.

  Example:

  A(i,1:11:3) ! Address 1,4,7,10 of column i

  A(i,10:1:-3) ! Address 10,7,4,1 of column i

Arrays of pointers

- Array of pointers does not exist in FORTRAN.
- The same effect can be achieved by an array of user defined type:

  Example: lower-triangular matrix

```
TYPE row
    REAL, DIMENSION(:), POINTER :: r
END TYPE row
TYPE (row), DIMENSION(n):: s,t
DO i=1,n
    ALLOCATE(t(i)%r(i),STAT=istat)
ENDDO
```

Pointers as aliases

- Example:

```
REAL, DIMENSION(:,:), POINTER :: window
window => table(m:n,p:q)
DO j=1,q-p+1
    DO i=1,n-m+1
        window(i,j) = …
    ENDDO
ENDDO
```