Introduction to High Performance Computing

Autumn, 2018

Lecture 8

Imperial College
London

Prasun Ray
30 October 2018

Fortran so far

- **Basic structure: must have main** Program**, may have any number of subroutines. All variables used in main program or subroutine must de declared after header**

- **Variable types:**

```
integer :: i1,j1,N
integer, parameter :: c = 2 !variables declared as parameters
cannot be changed within program
real(kind=8)
real(kind=8), dimension(10)
```

Fortran so far

- **Basic structure: must have main** Program**, may have any number of subroutines. All variables used in main program or subroutine must de declared after header**

- **Variable types:**

```
integer :: i1,j1,N
integer, parameter :: c = 2 !variables declared as parameters
cannot be changed within program
real(kind=8)
real(kind=8), dimension(10)
```

- **Loops and if statements:**

```
do i1 = 1,N,2 !loop from 1 to N with steps of 2

if (N <= size(array1)) then
```

See end of lecture 7 slides

Subroutines

From midpoint.f90:

```
!----------------------------------
!subroutine integrand
!   compute integrand, 4.0/(1+a^2)
!----------------------------------

subroutine integrand(a,f)
    implicit none
    real(kind=8), intent(in) :: a
    real(kind=8), intent(out) :: f
    f = 4.d0/(1.d0 + a*a)
end subroutine integrand
```

Subroutines

From midpoint.f90:

```
!-----------------------------------
!subroutine integrand
!   compute integrand, 4.0/(1+a^2)
!-----------------------------------

subroutine integrand(a,f)
    implicit none
    real(kind=8), intent(in) :: a
    real(kind=8), intent(out) :: f
    f = 4.d0/(1.d0 + a*a)
end subroutine integrand
```

- **Called from main program (or another subroutine):**

```
call integrand(xm,f)
```

- **Main program sends** xm **and** f **into** integrand**.**
- **From** intent **labels,** xm **is input into** integrand, f**, is output back to calling program**

More on Fortran

- Arrays
- Functions
- Modules

Arithmetic with arrays

- **Fortran 90 supports Matlab-style arithmetic with arrays**

```
program array1

    !Variable declarations:
    implicit none
    integer :: i1,j1,N
    integer, parameter :: c = 2 !variables declared as parameters
cannot be changed within program
    integer, dimension(4) :: x,y,z
    integer, dimension(4,4) :: A

    x= (/1,2,3,4/) !initialize x

    y = c*(x*x) !c*((x(1)*x(1),x(2)*x(2),...,x(4)*x(4))

    print *, 'x=',x
    print *, 'y=',y
```

Arithmetic with arrays

- **Fortran 90 supports element-by-element arithmetic with arrays**

```
program array1

    !Variable declarations:
    implicit none
    integer :: i1,j1,N
    integer, parameter :: c = 2
    integer, dimension(4) :: x,y,z
    integer, dimension(4,4) :: A

    x= (/1,2,3,4/) !initialize x

    y = c*(x*x) !c*((x(1)*x(1),x(2)*x(2),...,x(5)*x(5))

    print *, 'x=',x
    print *, 'y=',y
```

```
$  gfortran -o array1.exe array1.f90
$ ./array1.exe
 x=           1           2           3           4
 y=           2           8          18          32
```

Arithmetic with arrays

```
!    construct matrix A = [x' 2*x' 3*x' 4*x']
    do i1=1,4
        A(:,i1) = i1*x
        print *, 'i1th column of A=',A(:,i1)
    end do
```

```
 column               1 of A=            1            2            3            4
 column               2 of A=            2            4            6            8
 column               3 of A=            3            6            9           12
 column               4 of A=            4            8           12           16
```

- **Array slicing:** x(:) **all elements in x**
  x(start:stop:step)

Arithmetic with arrays

```
!    construct matrix A = [x' 2*x' 3*x' 4*x']
    do i1=1,4
        A(:,i1) = i1*x
        print *, 'i1th column of A=',A(:,i1)
    end do
```

```
 column           1 of A=          1          2          3          4
 column           2 of A=          2          4          6          8
 column           3 of A=          3          6          9         12
 column           4 of A=          4          8         12         16
```

- **Built-in functions for transpose, dot product, matrix multiplication:** *transpose, dot_product, matmul*
- **Most other linear algebra requires libraries such as *lapack***

Fortran arrays

- Often don’t know size of arrays in advance of program execution
- Fortran 77 approach: define arrays large enough for any (reasonable) problem size
- Fortran 90 allows *dynamic array allocation:*

Fortran arrays

- Often don't know size of arrays in advance of program execution
- Fortran 77 approach: define arrays large enough for any (reasonable) problem size
- Fortran 90 allows *dynamic array allocation:*

1. Declare arrays as allocatable:

```
integer, allocatable, dimension(:) :: x,y,z
integer, allocatable, dimension(:,:) :: A
```

2. Set size when needed:

```
allocate(x(4),y(4))
allocate(A(4,4))
```

Fortran arrays

- Often don’t know size of arrays in advance of program execution
- Fortran 77 approach: define arrays large enough for any (reasonable) problem size
- Fortran 90 allows *dynamic array allocation:*

1. Declare arrays as allocatable:

```
integer, allocatable, dimension(:) :: x,y,z
integer, allocatable, dimension(:,:) :: A
```

2. Set size when needed:

```
allocate(x(4),y(4))
allocate(A(4,4))
```

3. Deallocate when done (frees up memory and data is “erased”)

```
deallocate(A,x,y)
```

Fortran arrays: example

- **Previous example: computing** *sin(i), i=1,2,..,N*
- **We had:**

```
real(kind=8), dimension(10) :: array1
```

and:

```
!check that N is smaller than size of array1:
   if (N <= size(array1)) then
       call calculations(N,array1)
   else
      print *, 'N must be smaller than', size(array1)
      STOP
   end if
```

Very awkward! Use allocatable arrays instead.

Fortran arrays: example

- **Declare allocatable array (see *example2b.f90):***

```
real(kind=8), allocatable, dimension(:) :: array1
```

allocate:

```
!read data from data.in
    open(unit=10, file='data.in')
         read(10,*) N
    close(10)

    allocate(array1(N))
```

No need to check if size of array is large enough!

Fortran arrays: example

- **Declare allocatable array (see *example2b.f90):***

```
real(kind=8), allocatable, dimension(:) :: array1
```

allocate:

```
!read data from data.in
    open(unit=10, file='data.in')
         read(10,*) N
    close(10)

    allocate(array1(N))
```

No need to check if size of array is large enough!
Finally, deallocate:

```
!print 1st N elements of array
       print *, 'array1=',array1(1:N)

       deallocate(array1)
```

Fortran functions

- **Two kinds of fortran sub-programs: *subroutines* and *functions***
- **Basic idea: input → function(input) → output**
- **Fortran syntax:**
  1. Function must be declared in calling program as external, *e.g.*
  ```
  real(kind=8), external :: function_name
  ```
  2. Function call is intuitive:
  ```
  out = function_name(in1,in2,in3)
  ```

Fortran functions

- **Two kinds of fortran sub-programs: *subroutines* and *functions***
- **Basic idea: input → function(input) → output**
- **Fortran syntax:**

  1. Function must be declared in calling program as external, *e.g.*

  ```
  real(kind=8), external :: function_name
  ```

  2. Function call is intuitive:

  ```
  out = function_name(in1,in2,in3)
  ```

  3. The function header looks similar to a subroutine:

  ```
  function function_name(var1,var2,var3)
  ```

  4. But the function name is also a variable that must be declared, and function_name **will be returned to the calling program**

Fortran functions

```fortran
!Simple example of Fortran function
!Two numbers are added together

program function_example
    implicit none
    real(kind=8) :: x,y,z
    real(kind=8), external :: sumxy !function called in main program
    x = 2.d0
    y = 3.d0
    z = sumxy(x,y)
end program function_example
```

Fortran functions

```
!Simple example of Fortran function
!Two numbers are added together

program function_example
    implicit none
    real(kind=8) :: x,y,z
    real(kind=8), external :: sumxy !function called in main program
    x = 2.d0
    y = 3.d0
    z = sumxy(x,y)
end program function_example
!------------------------------------------------------------------
function sumxy(x,y)
    implicit none
    real(kind=8), intent(in) :: x,y
    real(kind=8) :: sumxy  !function name is a variable
    sumxy = x + y
end function sumxy
```

See *function_example.f90*

Functions

- Use functions when you want to return one variable

- Otherwise, use subroutines

Modules

How complicated is your program?

- **If it contains a few tasks: break problem into subroutines and functions**
- **But what if you have 15 sub-programs? What if you have 50 (not unusual)?**
- **Should package subprograms and required variables into *modules***
- **As code becomes “big”, planning becomes essential!**

Basic module structure

```
module module_name

    !1. variable declarations

contains

    !2. subroutines and functions

end module module_name
```

Module variables are “available” in all module subroutines and functions (do not need to be re-declared)

Basic module structure

```
module module_name

    !1. variable declarations

contains

    !2. subroutines and functions

end module module_name

program module_example
    use module_name
    implicit none
    !variable declarations

    !code

end program module_example
```

Variables and sub-programs in *module_name* are available in main program which *uses* module_name

Module example: *module_circle.f90*

- **Module which contains functions for computing circumference and area of circle**

- **Module variable: *π***

- ***radius* is set in calling program which *uses* the module**

Module example: *module_circle.f90*

```
module circle
!1. variable declarations
    implicit none
    real(kind=8) :: pi
    save
contains
!2. subroutines and functions
subroutine initialize_pi()
    implicit none

    pi = acos(-1.d0)

end subroutine initialize_pi
```

Module example: *module_circle.f90*

```
module circle
    implicit none
    real(kind=8) :: circle_pi
    save
contains
!2. subroutines and functions
subroutine initialize_pi()
    implicit none

    circle_pi = acos(-1.d0)

end subroutine initialize_pi

function circumference(radius)
    !compute circumference of circle given the radius
    implicit none
    real(kind=8), intent(in) :: radius
    real(kind=8) :: circumference

circumference = 2.d0*circle_pi*radius

end function circumference

!***Similar function for computing area here***
end module circle
```

Module example: *module_circle.f90*

Main program *uses* circle (main_circle.f90):

```
program main
    use circle
    implicit none
    real(kind=8) :: radius,C,A

    call initialize_pi()

    radius = 2.d0

    C = circumference(radius)

    A = area(radius)

!code continues...
```

Module example: *module_circle.f90*

Compiling and running:

```
$ gfortran -o circle.exe module_circle.f90 main_circle.f90

$ ./circle.exe
 radius=   2.0000000000000000
 circumference=   12.566370614359172
 area =    12.566370614359172
```

Notes:

- **Modules must be compiled before files which use them**
- **Compilation produces *circle.mod* which is needed when compiling main_circle.f90**