Introduction to High Performance Scientific Computing

Autumn, 2016

Lecture 10

Imperial College London

Prasun Ray

10 November 2016

Today

- A few comments on Fortran
- F2Py with fortran modules
- Timing Fortran code
- Networks

Imperial College London

Notes on Fortran

- Do not attempt to guess Fortran syntax! – look it up instead
- Do not develop Fortran code in Python using f2py – get a Fortran-only code working first
- Test code (compile and run) after adding a few (4-5) lines of code. Don't write 100 lines and then test!
- Lecture 7: code structure, variable types, loops, if-then, subroutines
- Lecture 8: allocatable arrays, functions modules
- Lecture 9: Lapack, f2py

Imperial College London

Notes on Fortran

Debugging code

- **If the compiler is giving you a series of errors, look at the topmost error message. It will include a line number**
- **If the code runs but crashes with a segmentation fault: probably a problem with array indices (e.g. trying to access the 12th element of a** dimension(11) **array**
- **If you can't tell where the code is crashing, add print statements, e.g.** print *, 1 **--some code–** print *, 2
  When the code is run, if the 1 prints to screen but not the 2, you know where the problem is.
- **If the code runs, but gives the wrong answer, add print statements outputting values of variables, try a small problem size where you know what values the variables can take**

Imperial College London

Notes on Fortran

- **Modules consist of:**
  1. **Module variables**
  2. **Module sub-programs**
- **Module variables are "available" throughout the module**
  - **They *do not* need to be declared**
  - **They *do not* need to be provided as input/output in the sub-program header**
- **Module variables and module sub-programs are also "available" in any program or sub-program that *uses* the module**
- **A module by itself doesn't do anything**
  - **There should be a "main" program which uses it**
  - **You can compile a module by itself: gfortran -c module.f90**
  - **But to generate an executable, you need a *program*:**
    gfortran -o program.exe module.f90 program.f90
  When re-compiling, first remove the previous .mod file

Imperial College London

Notes on Fortran

- **Compiling code that uses lapack routines:**

```
$ gfortran –c program.f90
$ gfortran –o program.exe program.o –llapack
```

or

```
$ gfortran –o program.exe program.f90 –llapack
```

Similarly, with f2py:

```
$ f2py –llapack –c program.f90 –m module_name
```

Imperial College London

F2Py and fortran modules

- F2Py will recognize subroutines and functions in modules
- What about variables?
  - Try f2py with circle module from last week (*f2pymodule_circle.f90)*

```
$ f2py –c f2pymodule_circle.f90 –m cmod
```

```
In [10]: import cmod

In [11]: cmod.<tab>
cmod.circle  cmod.so
```

Need to look at *cmod.circle*

Imperial College London

F2Py and fortran modules

Need to look at *cmod.circle*:

```
In [12]: cmod.circle?
Docstring:
'd'-scalar
initialize_pi()

Wrapper for ``initialize_pi``.

circumference = circumference(radius)

Wrapper for ``circumference``.

Parameters
----------
radius : input float

Returns
-------
circumference : float
area = area(radius)
```

And similar info for "area"

Imperial College London

F2Py and fortran modules

How do we access variables and methods in *cmod.circle*?

```
In [23]: cmod.circle.<tab>
cmod.circle.area           cmod.circle.circumference
cmod.circle.initialize_pi  cmod.circle.pi
```

Can initialize *pi* in python:

```
In [9]: cmod.circle.pi
Out[9]: array(0.0)

In [10]: cmod.circle.pi = pi

In [11]: cmod.circle.pi
Out[11]: array(3.141592653589793)
```

Imperial College London

F2Py and fortran modules

How do we access variables and methods in *cmod.circle*?

```
In [23]: cmod.circle.<tab>
cmod.circle.area            cmod.circle.circumference
cmod.circle.initialize_pi   cmod.circle.pi
```

Can initialize ***pi*** **in python:**

```
In [9]: cmod.circle.pi
Out[9]: array(0.0)

In [10]: cmod.circle.pi = pi

In [11]: cmod.circle.pi
Out[11]: array(3.141592653589793)
```

Can also initialize allocatable arrays, see ***f2pymodule_circle.f90…***

Imperial College London

F2Py and fortran modules

Can also initialize allocatable arrays, see ***f2pymodule_circle_array.f90*****:**

```
!module for computing circumference, area, and "mass" of circle
module circle
    implicit none
    real(kind=8) :: pi
    real(kind=8), allocatable, dimension(:) :: weights,mass
    save

        …
        …

subroutine compute_mass(radius,mass)
    !compute mass = weights*area
    implicit none
    real(kind=8), intent(in) :: radius
    real(kind=8), intent(out) :: mass(:)

    mass = weights*area(radius)
end subroutine compute_mass
```

Imperial College London

F2Py and fortran modules

Basic steps:

1. **Compile with f2py:** `$ f2py -c f2pymodule_circle_array.f90 -m cmoda`

Imperial College London

F2Py and fortran modules

Basic steps:

1. Compile with f2py: `$ f2py –c f2pymodule_circle_array.f90 –m cmoda`
2. Import module in python: `In [4]: import cmoda`

Imperial College London

F2Py and fortran modules

Basic steps:

1. Compile with f2py: `$ f2py –c f2pymodule_circle_array.f90 –m cmoda`
2. Import module in python: `In [4]: import cmoda`
3. Initialize variables:

```
In [15]: cmoda.circle.pi=pi

In [16]: cmoda.circle.weights=arange(5)

In [17]: cmoda.circle.pi,cmoda.circle.weights
Out[17]: (array(3.141592653589793), array([ 0., 1., 2., 3., 4.]))
```

Imperial College London

F2Py and fortran modules

Basic steps:

1. Compile with f2py: `$ f2py –c f2pymodule_circle_array.f90 –m cmoda`
2. Import module in python: `In [4]: import cmoda`
3. Initialize variables:

```
In [15]: cmoda.circle.pi=pi

In [16]: cmoda.circle.weights=arange(5)

In [17]: cmoda.circle.pi,cmoda.circle.weights
Out[17]: (array(3.141592653589793), array([ 0., 1., 2., 3., 4.]))
```

4. Compute mass:

```
In [18]: cmoda.circle.compute_mass(2.0)

In [19]: cmoda.circle.mass
Out[19]: array([  0. , 12.56637061, 25.13274123, 37.69911184,
50.26548246])
```

Imperial College London

Finite difference methods

- Finite difference methods are a standard approach for numerical differentiation.
- They form the basis for a wide variety of methods used to solve partial differential equations
- See online supplementary class notes: *Notes on numerical differentiation with finite difference methods*

Imperial College London

Unix *time* command

Can use *time* to obtain timing info for any unix command:

```
$ time ./midpoint.exe
 N=      512000
 sum=   3.1415926535901515
 error=   3.5837999234900053E-013

real   0m0.015s
user   0m0.010s
sys 0m0.003s
```

- real is approximately the wall-clock time
- user is time spent executing the program
- sys is time spent on system tasks required by program

Imperial College London

Fortran timing functions

- Unix *time* doesn't tell you how much time different parts of program take
- system_clock and cpu_time gives wall time and cpu time between two points in code
- See midpoint_time.f90:

```
!timing variables
    real(kind=8) :: cpu_t1,cpu_t2,clock_time
    integer(kind=8) :: clock_t1,clock_t2,clock_rate
```

Imperial College London

Fortran timing functions

- Unix *time* doesn't tell you how much time different parts of program take
- system_clock and cpu_time gives wall time and cpu time between two points in code
- See midpoint_time.f90:

```
!timing variables
    real(kind=8) :: cpu_t1,cpu_t2,clock_time
    integer(kind=8) :: clock_t1,clock_t2,clock_rate

 call system_clock(clock_t1)
 call cpu_time(cpu_t1)
    !loop over intervals computing each interval's contribution to
integral
... Midpoint quadrature ...
 call cpu_time(cpu_t2)
 print *, 'elapsed cpu time (seconds) =',cpu_t2-cpu_t1

 call system_clock(clock_t2,clock_rate)
 print *, 'elapsed wall time (seconds)= ',
                                dble(clock_t2-clock_t1)/dble(clock_rate)
```

Imperial College London

Fortran timing functions

- Unix *time* doesn't tell you how much time different parts of program take
- system_clock and cpu_time gives wall time and cpu time between two points in code
- See midpoint_time.f90:

```
$ ./midpoint_t.exe
 elapsed cpu time (seconds) =   8.6239999999999997E-003
 elapsed wall time (seconds)=    9.12799966E-03
 N=      512000
 sum=   3.1415926535901515
 error=   3.5837999234900053E-013
```

- Can place timing commands throughout code to find bottlenecks

Imperial College London

Fortran timing functions

- Also, often have a *theoretical estimate* of how cost scales with problem size
- A method may require O(N) (or $O(N\ln_2 N)$) or $O(N^2)$) operations
- But does your implementation of the algorithm match theory?
- How do compiler optimizations affect performance?
- Carefully timing code while varying the problem size can help answer these questions

Imperial College London

Networks

Examples of significant networks include:

- Social networks
- World-wide web
- Internet
- Air transportation network
- Cellular network

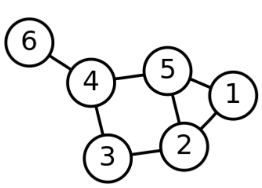
The science of networks is an important, rapidly growing field

Networks: basics

- A network has N *nodes* and L links between nodes
- Each node has a label, e.g. 1, 2, …, N
- Then a link between node i and j can be represented simply as (i, j)

Networks: basics

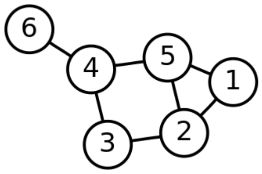
- A network has N *nodes* and L links between nodes
- Each node has a label, e.g. 1, 2, …, N
- Then a link between node i and j can be represented simply as (i, j)



Example: 6 nodes, 7 links
Node one has two edges: (1,2) and (1,5)

The graph can be represented by the *adjacency matrix*, A
$A_{ij}=1$ if there is link between nodes i and j

Networks: basics



Example: 6 nodes, 7 links
Node one has two links: (1,2) and (1,5)

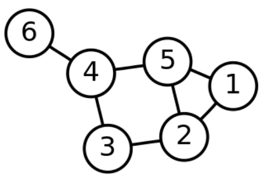
The graph can be represented by the *adjacency matrix*, A
A_{ij}=1 if there is link between nodes i and j

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

A is symmetric.

Imperial College
London

Networks: basics



Example: 6 nodes, 7 links
Node one has two edges: (1,2) and (1,5)

The graph can be represented by the *adjacency matrix*, A
A_{ij}=1 if there is link between nodes i and j

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

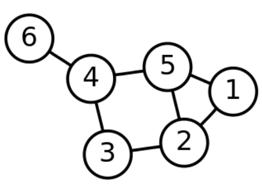
Can also represent connected portions of graph with edge list:

$$\begin{bmatrix} 1 & 1 & 2 & 2 & 3 & 4 & 4 \\ 2 & 5 & 3 & 5 & 4 & 5 & 6 \end{bmatrix}$$

A is symmetric.

Imperial College
London

Networks: basics



The *degree* of a node is the number of is the total number of links connected to it:
$q_1 = 2$, $q_5 = 3$, …

The *degree distribution*, P(q) is particularly important. P(q) is the fraction of nodes in the graph with degree = q

P(1) = 1/6, P(2) = 2/6, P(3) = 3/6

Homework 3: You will work with a simple model for growing networks.

Imperial College
London