Numerical Modelling in **FORTRAN**
day 7

Paul Tackley, 2019
Today’s Goals

1. Makefiles
2. Intrinsic functions
3. Pointers
4. Optimisation: Making your code run as fast as possible
Projects: start thinking about

Agree topic with me before final lecture
Makefiles

• If your programs are split over several files, it may be easiest to write instructions for compiling them in a **makefile** (this applies mainly to unix-like systems)

• This also makes sure that the various options you use remain the same

• The makefile defines dependencies, so it only recompiles source files that have changed since the last compilation

• See manual pages for full information
Example makefile

% example makefile

FFLAGS = -O2
OBJECTS = main.o modules.o

myproject: $(OBJECTS)
gfortran -o myproject $(OBJECTS)

main.o : main.f90 stuff.mod
gfortran -c main.f90

stuff.mod : module.o

modules.o : modules.f90
gfortran -c modules.f90

USAGE: Just type in “make” or “make myproject”
Intrinsic functions

• There are many more intrinsic functions than we have used!
• It is good to browse a list so that you know what is available, for example at
  • http://www.nsc.liu.se/~boein/f77to90/a5.html
• This doesn’t list cpu_time(), which is useful for checking how long parts of the code take
Pointers

• Point to either
  – (i) data stored by a variable declared as a target, or
  – (ii) another pointer, or
  – (iii) an area of allocated memory

• (i) and (ii) are mainly useful for creating interesting data structures (like linked lists); for us (iii) will be the most useful.

• “=>” is used to point a pointer at something

• See examples next slide

• Functions can also return pointers (“pointer functions”)
```fortran
program pointless
  implicit none
  real,pointer:: p
  real,target:: a=3.1,b=4.7
  p=>a               ! point p to a
  print*,p           ! prints 3.1 (value of a)
  p=4.0             ! changes a to 4.0
  print*,a           ! prints 4.0
  p=>b               ! now points to b
  print*,b           ! prints 4.7
  print*,associated (p) ! test status: prints true
  nullify(p)         ! points p at nothing
  print*,associated (p) ! prints false
end program pointless
```
program pointarray
  implicit none
  integer,parameter:: n=7
  real,target:: T(n,n)
  real,pointer:: bot_boundary(:), &
       top_boundary(:), center4(:,:)

  top_boundary  => T(:,1)
  bot_boundary  => T(:,n)
  center4      => T(n/2:n/2+1,n/2:n/2+1)

  call random_number(T)
  top_boundary  = 0.0   ! easy way to do
  bot_boundary  = 1.0   ! boundary conditions
  center4      = 1.0

  print*,T
end program pointarray
Pointers in defined types
similar to allocatable array (i.e., not pointing to a variable)

program pointertype
implicit none

type array2D
  real, pointer:: a(:,:, :) ! 2D array
end type array2D

type(array2D), allocatable:: T(:) ! 1D array of 2D arrays
integer i, n, ng

print '(a,$)', "Number of multigrid levels:"
read *, ng
allocate (T(ng)) ! allocate number of grids
do i = 1, ng
  n = 2**(i+1) ! #grid points in this grid
  print *, 'Allocating grid', n, n
  allocate (T(i)%a(n, n)) ! each grid
end do

!.... rest of program

end program pointertype
Speed and optimization

• Running large simulations can take a long time => speed is important. **Optimization** = making it run as fast as possible

• First consideration: use the most efficient algorithm, e.g., multigrid

• Then: get code working using code that is easy-to-read and debug

• Finally: Find out which part(s) of the code are taking the most time, and rewrite those to optimize speed

• Code written for maximum speed may not be the most legible or compact!
Manual versus automatic optimisation

• Many steps can be done automatically by the compiler. Use appropriate compiler options (see documentation), e.g.,
  – -O2, -O3: selects a bunch of optimisations
  – -unroll unroll loops
  – etc. (see compiler documentation)

• Some need to be done manually. In general, try to write code in such a way that the compiler can optimise it!
Example of compiler optimisation

- Solution AdvDif code, test case, 32-bit, gfortran, iMac. (use “time a.out”)  

<table>
<thead>
<tr>
<th>Options</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recommended debugging</td>
<td>27.05</td>
</tr>
<tr>
<td>none</td>
<td>15.27</td>
</tr>
<tr>
<td>-O1</td>
<td>3.371</td>
</tr>
<tr>
<td>-O2</td>
<td>3.368</td>
</tr>
<tr>
<td>-O3</td>
<td>2.754</td>
</tr>
<tr>
<td>-O3 –ffast-math –ftree-vectorize</td>
<td>2.688</td>
</tr>
<tr>
<td>-Ofast</td>
<td>2.652</td>
</tr>
</tbody>
</table>
Example of compiler optimisation

- Solution convection code (this week), test case, 32-bit, gfortran, macbook pro

<table>
<thead>
<tr>
<th>Options</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>31.57</td>
</tr>
<tr>
<td>-O1</td>
<td>24.741</td>
</tr>
<tr>
<td>-O2</td>
<td>23.578</td>
</tr>
<tr>
<td>-O3</td>
<td>23.625</td>
</tr>
<tr>
<td>-O2 –ffast-math</td>
<td>20.958</td>
</tr>
<tr>
<td>-O2 –ffast-math –ftree-vectorize</td>
<td>20.552</td>
</tr>
</tbody>
</table>
Manual optimization step 1: Identify bottlenecks

- The 90/10 law: 90% of the time is spent in 10% of the code. Find this 10% and work on that!
- e.g., Use a profiler. Or put cpu_time() statements in to time different subroutines or loops
- Usually, most of the time is spent in loops. In our multigrid code it is probably the loops that update the field and calculate the residue. **Optimization of loops is the most important consideration.**
To understand optimization it is important to understand how the CPU works

• Two aspects are particularly important:
  – Cache
  – Pipelining
Cache memory

- Fast, small memory close to a CPU that stores copies of data in the main memory

Substantially reduces latency of memory accesses. Designing code such that data fits in cache can greatly improve speed. Good design includes memory locality, and not-too-large size of arrays.
Athlon64 multiple caches

Main Memory
\(\leq 8\) GB

Other CPUs

L2 Unified
1 MB 16-way

L2 ITLB
512 entries
4-way

L1 Instruction Cache
64KB 2-way

L2 DTLB
512 entries
4-way

L1 ITLB
4 KB
32 entries
full assoc

L1 DTLB
4 KB
32 entries
full assoc

L1 Data Cache
64KB 2-way 2 ports
Tips to improve cache usage

• Memory locality: data within each block should be close together (small stride). Appropriate data structures and ordering of nested loops. (see later)

• Arrays shouldn’t be too large. e.g., for a matrix*matrix multiply, split each matrix into blocks and treat blocks separately
CPU architecture and pipelining
(images from http://en.wikipedia.org/wiki/Central_processing_unit)

Executing an instruction takes several steps. In the simplest case these are done sequentially, e.g.,

15 cycles to perform 3 instructions!
Pipelining

If each step can be done independently, then up to 1 instruction/cycle can be sustained => 5* faster

Basic 5-stage pipeline. Like an assembly line. Several cycles are needed to start and end the pipeline.
Like a car assembly line

1913: sped up assembly of Model T Ford from 12 hours to <3 hours!
Superscalar pipeline

More than 1 instruction per cycle (2 in the example below)
Pipelining in practice

• Done by the compiler, but must write code to maximize success
• Branches (e.g., “if”) cause the pipeline to flush and have to restart
• Avoid branching inside loops!
• Helps if data is in cache
Summary

• Goal is to maximize use of cache and pipelining.
• Design code to reuse data in cache as much as possible, and to stream data efficiently through the CPU (pipeline / vectorisation)
More information

• Wikipedia pages
  – http://en.wikipedia.org/wiki/Loop_optimization
  – http://en.wikipedia.org/wiki/Memory_locality

• http://www.ibiblio.org/pub/languages/fortran/ch1-9.html

• http://www.azillionmonkeys.com/qed/optimize.html

• ETH course “How to write fast numerical code” (Frühjahrsemester)
Time taken for various operations

- Slowest: sin, cos, **, etc.
- sqrt
- /
- *
- fastest: + -
- Simplify equations in loops to minimize number of operators, particularly slow ones!
Loop optimization (1)

• Remove conditional statements from loops!

SLOW

```plaintext
do i=1,n
  if (condition) then
    a(i) = b(i)+c(i)
  else
    a(i) = b(i)-c(i)
  end if
end do
```

FAST

```plaintext
if (condition) then
  do i=1,n
    a(i) = b(i)+c(i)
  end do
else
  do i=1,n
    a(i) = b(i)-c(i)
  end do
end if
```
Loop optimization (2)

- Data locality: fastest if processing nearby (e.g., consecutive) locations in memory
- Fortran arrays: first index accesses consecutive locations (opposite in C)
- Order loops such that first index loop is innermost, 2nd index loop is next, etc.

SLOW

```plaintext
do i=1,n
   do j=1,m
      a(i,j) = b(i,j)+c(i,j)
   end do
end do
```

FAST

```plaintext
do j=1,m
   do i=1,n
      a(i,j) = b(i,j)+c(i,j)
   end do
end do
```
Loop optimization (3)

- **Unrolling**: eliminate loop overhead by writing loops as lots of separate operations
- **Partial unrolling**: reduces number of cycles, reducing loop overhead

This can be done automatically by the compiler

Original

```
do i=1,n
  a(i)=b(i)*c(i)
end do
```

Unrolled by factor 4

```
do i=1,n,4
  a(i)  =b(i)  *c(i)
  a(i+1)=b(i+1)*c(i+1)
  a(i+2)=b(i+2)*c(i+2)
  a(i+3)=b(i+3)*c(i+3)
end do
```
Loop optimization (4)

• fusing + unrolling: see below

Original loop

```
do j=1,2*n
  do i=1,m
    a(i)=a(i)+1./real(i+j)
  end do
end do
```

Partial unrolling

```
do j=1,2*n,2
  do i=1,m
    a(i)=a(i)+1./real(i+j)
    a(i)=a(i)+1./real(i+j+1)
  end do
end do
```

+fusion (reduces number of writes by factor 2)

```
do j=1,2*n,2
  do i=1,m
    a(i)=a(i)+1./real(i+j)+1./real(i+j+1)
  end do
end do
```
Loop optimization (5): other things

• simplify calculated indices
• use registers for temporary results
• Put invariant expressions (things that don’t change each iteration) outside the loop
• Loop blocking/tiling: splitting a big loop or nested loops into smaller ones in order to fit into cache.
Other Optimizations

• Use binary I/O not ascii

• Avoid splitting code into excessive procedures.
  – Overhead associated with calling functions/subroutines
  – Reduces ability of compiler to do global optimizations

• Use procedure inlining (done by compiler): compiler inserts a copy of the function/subroutine each time it is called

• Use simple data structures in major loops to aid compiler optimizations (defined types may slow things down)