Numerical Modelling in FORTRAN day 7

Paul Tackley, 2018
Today’s Goals

1. **Optimisation**: Making your code run as fast as possible

2. Combine advection-diffusion and Poisson solvers to make a **convection simulation program**
Projects: start thinking about

Agree topic with me before final lecture
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 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
  <= 8 GB

L2 Unified
  1 MB 16-way

L2 ITLB
  512 entries
  4-way

L2 DTLB
  512 entries
  4-way

L1 Instruction Cache
  64KB 2-way

L1 ITLB
  4 KB
  32 entries
  full assoc

L1 DTLB
  4 KB
  8 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**

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

**FAST**

```fortran
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

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
```

This can be done automatically by the compiler
Loop optimization (4)

- fusing + unrolling: see below

Original loop

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

Partial unrolling

```plaintext
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)

```plaintext
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)
Now back to iterative solvers...
Goal for today’s exercise: Calculate velocity field from temperature field

=> convection

e.g., for highly viscous flow (e.g., Earth’s mantle) with constant viscosity (P=pressure, Ra=Rayleigh number):

\[-\nabla P + \nabla^2 \vec{v} = -Ra \frac{\partial T}{\partial x} \]

(y points up)

Substituting the streamfunction for velocity, we get:

\[
(v_x, v_y) = \left( \frac{\partial \psi}{\partial y}, -\frac{\partial \psi}{\partial x} \right)
\]

\[\nabla^4 \psi = Ra \frac{\partial T}{\partial x}\]

writing as 2 Poisson equations:

\[\nabla^2 \psi = \omega\]

\[\nabla^2 \omega = Ra \frac{\partial T}{\partial x}\]

the streamfunction-vorticity formulation
Example of convection (in 3-D)

$Ra = 10^7$
Mathematical aside

\[-\nabla P + \nabla^2 \vec{v} = -Ra.T\hat{y}\]

Is a vector equation with x and y parts:

\[-\frac{\partial P}{\partial x} + \nabla^2 v_x = 0 \quad (1) \]
\[-\frac{\partial P}{\partial y} + \nabla^2 v_y = -Ra.T \quad (2)\]

Take \( \frac{\partial (1)}{\partial y} - \frac{\partial (2)}{\partial x} \) \Rightarrow

\[\nabla^2 \left( \frac{\partial v_x}{\partial y} - \frac{\partial v_y}{\partial x} \right) = Ra \frac{\partial T}{\partial x}\]

Use streamfunction:

\[\nabla^2 \left( \frac{\partial^2 \psi}{\partial y^2} + \frac{\partial^2 \psi}{\partial x^2} \right) = \nabla^2 \nabla^2 \psi = Ra \frac{\partial T}{\partial x}\]

This is the z-component of the curl of the momentum equation:

\[\nabla \times \mathbf{F} = \left( \frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \right) \hat{x} + \left( \frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \right) \hat{y} + \left( \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right) \hat{z}\]
Combine advection-diffusion and Poisson solver routines

• Initialise
  – T=random or spike or ...
  – S, W=0
  – grid spacing $h=1/(ny-1)$ and diffusive timestep

• Timestep until desired time is reached
  – calculate right-hand side=$Ra \cdot dT/dx$
  – Poisson solve to get $W$ from rhs
  – Poisson solve to get $S$ from $W$
  – calculate $V$ from $S$
  – calculate advective timestep and min(adv,diff)
  – take advection and diffusion time step
  – $t=t+dt$: have we finished? If not, loop again.

Use multigrid solver

Use existing adv-diff routines
START

INPUT
number grid points: nx,ny
Rayleigh number: Ra
time-step as: a_dif, a_adv
integration time: total_time
Initial condition choice

TIME STEP
Calculate Ra*dT/dx->W->S->v
Calculate dt_adv -> dt
Calculate del2(T)
Calculate v.grad(T)
T = T + dt*(d2T-vgradT)
time = time + dt

INITIALISE
h=1/(ny-1); dt_dif=a*h**2/k
Allocate T, S, W, vx, vy
Initialise T field
S=W=v=0
time=0

time>total_time?

Yes

END

No
Program definition

• Input parameters:
  – \( nx, ny \): number of grid points \( y=\text{vertical}, \ \text{grid spacing is the same, i.e., } dx=dy=h \)
  – \( Ra \): Rayleigh number (typical range \( 1e3-1e7 \))
  – \( \text{total\_time} \): large enough to reach stable state

• Boundary conditions
  – \( T \): as before, \( T=1 \) at bottom, \( 0 \) at top, \( dT/dx=0 \) at sides
  – \( S \) and \( W \): \( 0 \) at all boundaries

• Example structure
  – Module with Poisson solver routines
  – Module with 2\_deriv, v\_gradT, S\_to\_V routines
  – main program

• Try to write frames at different times and make an animation!
Exercise

• Get everything together and run a test case with
  – nx=257, ny=65 (i.e., aspect ratio 4)
  – Ra=1e5
  – total_time=0.1
  – err=1.e-3
  – Random initial T field

• Email code, plots and parameter file for this case.

• Due in 2 weeks: 03.12.2018
Should look something like this

Note: It won’t look exactly the same, due to the random initial T field. You might get 2 or 4 cells instead of 3.