

Numerical Modelling in **FORTRAN** day 8

Paul Tackley, 2014

Today' s Goals

1. **Review** operators from last week
2. **Makefiles**
3. **Optimisation:** Making your code run as fast as possible
4. Continue **writing convection programs** (due next week)
5. Start thinking about projects (see day 1 slides)

Review operators from last week

- Generic procedures
- New operators
- Overloaded operators (e.g., $+$, $-$, $=$)

the interfaces

```

module coords
  implicit none

  type point
    real:: x,y,z
  end type point

  interface apbxc ! define generic interface
    module procedure rapbxc,iapbxc
  end interface

  interface operator (.distance.) ! new operator
    module procedure pointseparation
  end interface

  interface operator (+) ! new version of +
    module procedure pointplus
  end interface

  interface operator (-) ! new version of -
    module procedure pointminus
  end interface

  interface assignment (=) ! new "="
    module procedure absvec ! converts
  end interface ! point to real

contains

```

the procedures

contains

```
real function rapbxc(a,b,c) ! real version of fn
  real,intent(in):: a,b,c
  rapbxc = a+b*c
end function rapbxc

integer function iapbxc(a,b,c) ! int version
  integer,intent(in):: a,b,c
  iapbxc = a+b*c
end function iapbxc

real function pointseparation(a,b) ! for .distance.
  type(point),intent(in):: a,b
  pointseparation = sqrt( &
    (a%x-b%x)**2+(a%y-b%y)**2+(a%z-b%z)**2)
end function pointseparation

function pointplus(a,b) ! for +
  type(point):: pointplus
  type(point),intent(in):: a,b
  pointplus%x = a%x + b%x
  pointplus%y = a%y + b%y
  pointplus%z = a%z + b%z
end function pointplus

function pointminus(a,b) ! for -
  type(point):: pointminus
  type(point),intent(in):: a,b
  pointminus%x = a%x - b%x
  pointminus%y = a%y - b%y
  pointminus%z = a%z - b%z
end function pointminus

subroutine absvec(a,b) ! for = (distance
  real,intent(out):: a ! from origin)
  type(point),intent(in):: b
  a = sqrt(b%x**2+b%y**2+b%z**2)
end subroutine absvec

end module coords
```

test
program

```
program test
  use coords
  real:: a=1.2,b=3.4,c=5.6,d
  integer:: i=2,j=3,k=4
  type(point):: p1,p2,p3
  p1%x=1.2; p1%y=0. ; p1%z=3.1
  p2%x=0. ; p2%y=1.2; p2%z=1.7

  print*,apbxc(a,b,c)    ! real arguments
  print*,apbxc(i,j,k)    ! integer argumnts

  p3 = p1 - p2           ! using overloaded -
  p3 = p1 + p2           ! using overloaded +

  d = p1                  ! using overloaded =,

  d = p1.distance.p2 ! using new operator
end program test
```

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

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

Dependent on

`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`

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

: gives dependencies

FFLAGS lists flags to be used when compiling fortran

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

Rename is file as 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

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.,
 - -O means optimisation — **-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, test case, 32-bit, gfortran, macbook pro

Options	Time (s)
None	31.57
-O1	24.741
-O2	23.578
-O3	23.625
-O2 -ffast-math	20.958
-O2 -ffast-math -ftree-vectorize	20.552

Manual optimization step 1: Identify bottlenecks

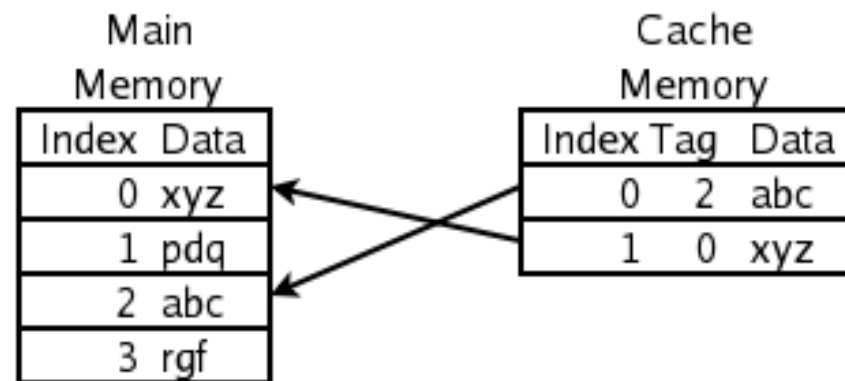
- The 90/10 law: 90% of the time is spent in 10% of the code. Find this 10% and work on that!
- e.g., Use a profiler. Or put `cpu_time()` statements in to time different subroutines or loops
- Usually, most of the time is spent in loops. In our multigrid code it is probably the loops that update the 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

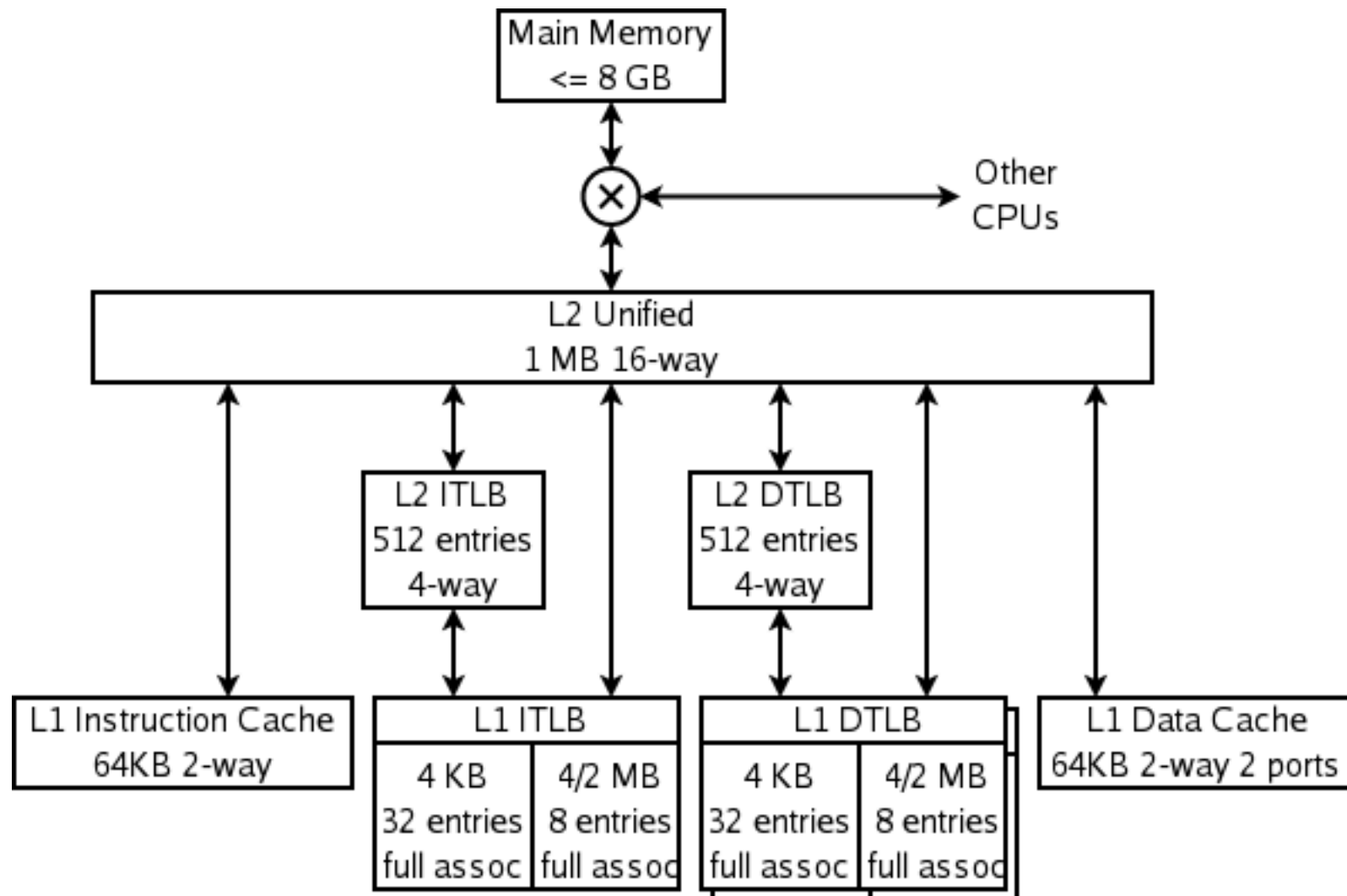
- Fast, small memory close to a CPU that stores copies of data in the main memory



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

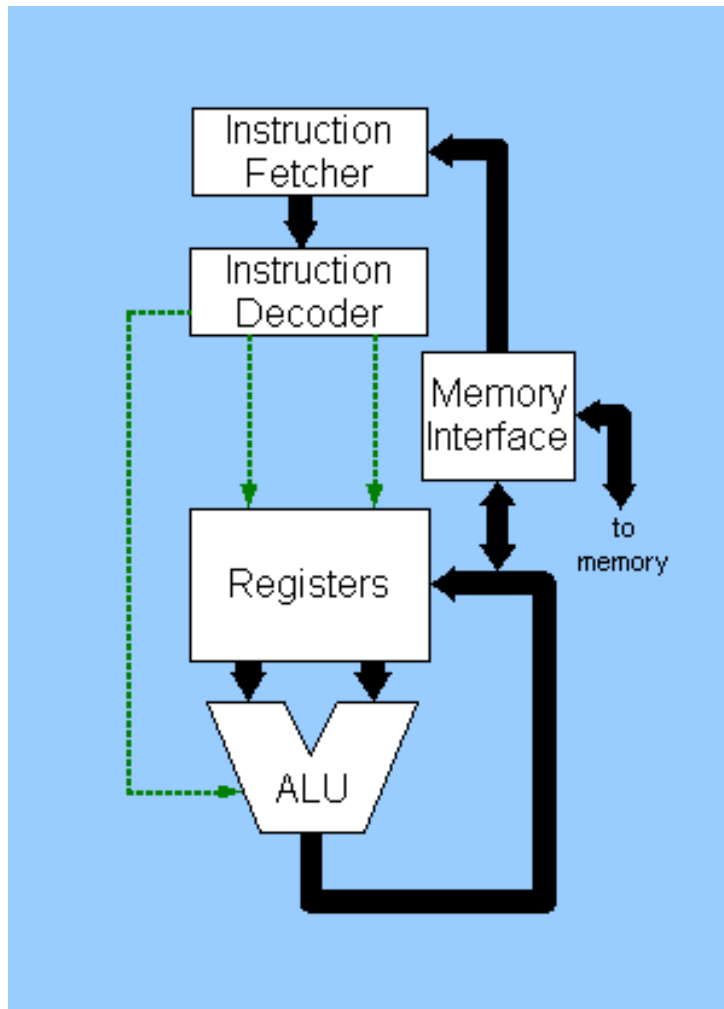


Tips to improve cache usage

- **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

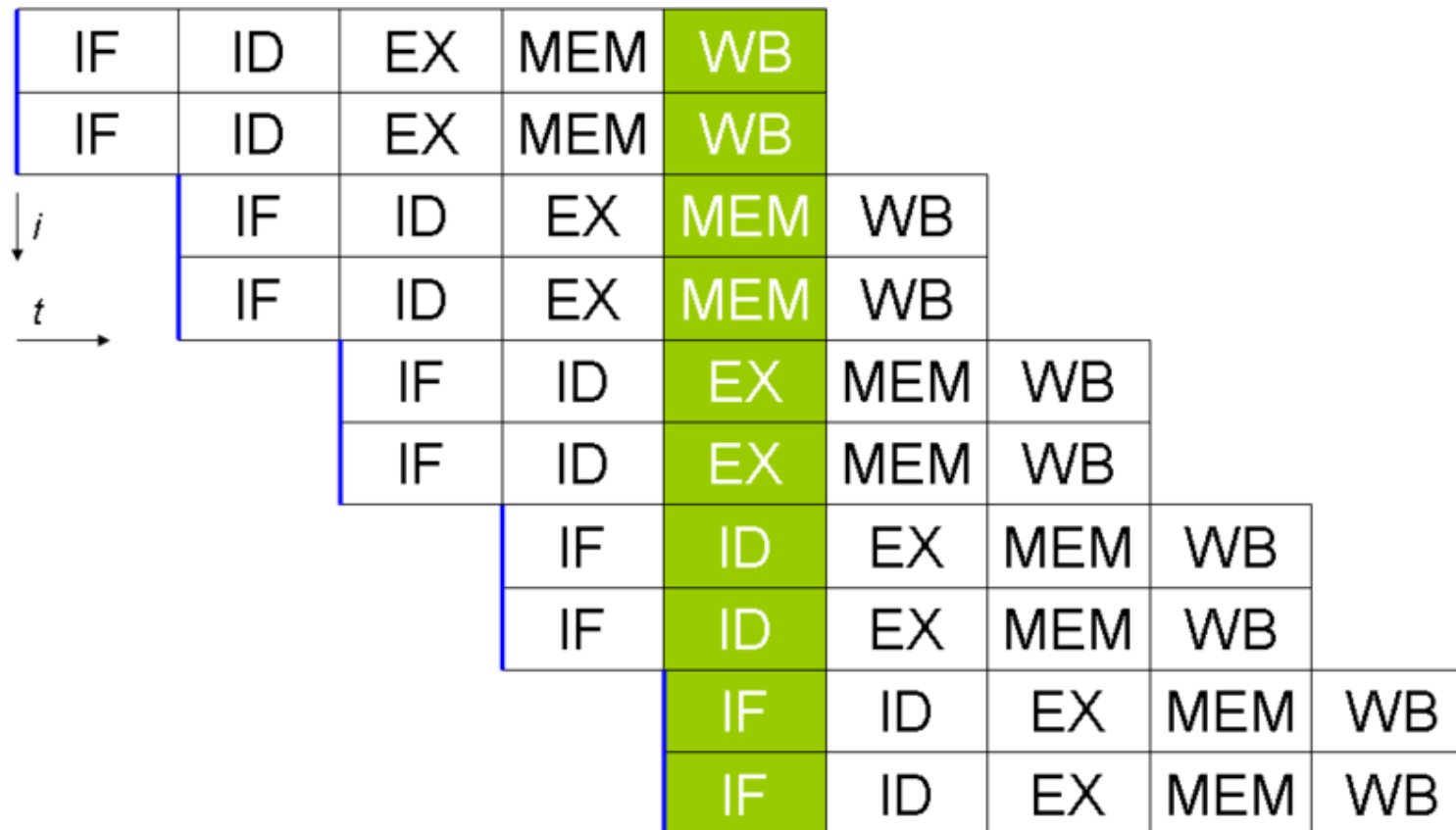
If each step can be done independently, then up to 1 instruction/cycle can be sustained => 5* faster



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

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/Optimization_\(computer_science\)](http://en.wikipedia.org/wiki/Optimization_(computer_science))
 - http://en.wikipedia.org/wiki/Memory_locality
 - http://en.wikipedia.org/wiki/Software_pipelining
- <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” (Frühjahrssemester)

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

```
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

```
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

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

FAST

```
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

```
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)