Simulation course FYSM350

Fortran : Vesa Apaja email: vesa.apaja@gmail.com

MD : Hannu Häkkinen

MC : Juha Merikoski

www info:

Course homepage:

http://www.phys.jyu.fi/homepages/merikosk/Simu2006.html

(a link to this page is in korppi)

Useful material:

http://www.csc.fi/oppaat/f95/

Introduction

- History

  “Dad's FORTRAN”

  First version born late 50's at IBM

  First standard 1966 : FORTRAN 66

  Second standard 1977: FORTRAN 77

  --------------- prehistory --------------------------------

  Third standard 1990 : Fortran 90

  Fourth standard 1995 : Fortran 95

  Fifth standard 2003 : Fortran 2003

Describing his early work on FORTRAN, John Backus said:-
We did not know what we wanted and how to do it. It just sort of grew. The first struggle was over what the language would look like. Then how to parse expressions - it was a big problem and what we did looks astonishingly clumsy now....

Free compilers

Intel Fortran (ifort or ifc)

http://www.intel.com/software/products/compilers/flin/noncom.htm

- Windows or linux, any Intel or AMD processor
- Optimising f95 compiler with 2003 “readiness”
- Free for non--commercial use; Windows version is only evaluation copy
- After registering you get a license number
- Easy to install (linux: less than 5 mins)

Gnu g95

http://g95.sourceforge.net/

Source code available - written in C   :^)

precompiled binaries for

Linux x86, Cygwin x86, Windows x86, Powerpc, FreeBSD x86,
Sparc Solaris, Linux IA64, Linux x86_64/EMT64, Linux Alpha,
Irix Mips   *etc.*

Commercial compilers

Often linked to commands **f90** or **f95**

- Pathscale **pathf90**
- PGI (Portland group) **pgf90**
- MIPSPRO (SGI Irix machines)
- Lahey (Fujitsu) **lf95**
- Absoft
- Compaq Fortran (Digital Fortran)

Intel: Compiler options

Try **ifort -help** or **man ifort**

Development phase:

ifort -check all program.f90

Production phase:

Always try : **-fast**

ifort -O3 program.f90 Use always at least this

ifort -tpp7 -xP pentium 4 processor

Expect output lines "*remark: LOOP WAS VECTORIZED*"

g95: Compiler options

Development phase:

g95 -Wall -std=f95 program.f90

Production phase:

g95 -O3 -std=f95 program.f90

Try also : **-O2 , -funroll-all-loops -mtune=pentium4**
(practically all options of the gcc compiler apply)

First program

```
program test
  implicit none
  integer:: i
  real (kind(1.d0)) :: x,y
  complex (kind(1.d0)) :: c
  x = 5.d0
  y = 20.d0
  c = cmplx(x,y,kind(1.d0))
  print*,x,y,c
end program test
```

Good habit: use this always if your program is meant for serious use

Unused variable: waste of space

Double precision:
~ 16 decimal accuracy

Internal function “**cmplx**”

Convert two double precision numbers to a complex number

output:

```
    5.00000000000000       20.0000000000000
  (5.00000000000000,20.0000000000000)
```

Formatted output

Most important format codes: examples

2f15.5	two floating point numbers, use 15 characters and give 5 decimals
f0.10	one floating point numbers, 10 decimals, as much field as it takes
a20	one 20 characters long field
d15.10	one number of the exponent type **10.d12**
5x	5 spaces

Bad format codes may

- cause the program to terminate
- clobber the output to a useless form:

```
write(*,'(3f0.3)') 0.1234d0,0.5678d0,0.7890d0          0.1230.5680.789
```

Examples:

```
write(*,'(d15.5)') 1.123456789d12          0.11235D+13
write(*,'(g15.5)') 1.123456789d12          0.11235E+13
write(*,'(f15.5)') 1.123456789d12      ***************  ←  Field is too short
write(*,'(e15.5)') 1.123456789d12          0.11235E+13
```

Simple formatted output to a file

```
program test
   implicit none
   integer, parameter:: dbl=kind(1.d0)
   integer:: i
   real (dbl), parameter:: dx=0.1d0
   real (dbl) :: x
   write(12,'(2a15)') 'x','sin(x)'
   do i = 1, 10
      x = (i-1)*dx
      write(12,'(2f15.10)') x,sin(x)
   end do
end program test
```

"Automatic" file I/O:
Output to file fort.12

Unit number

Unit * : default
Unit 5 : keyboard
Unit 6 : screen

Formatted output cont'd

- Format strings in write statements:

```
write(55,”result = ”,f15.10)')   res
```

- For frequently occurring formats use a separate **format** statement:

```
...
kin_E = 12.66666666666d0
pot_E = -1.44444444444d0
write(*,900) '   kinetic energy',kin_E          kinetic energy =   12.6666666667
write(*,900) 'potential energy',pot_E       potential energy =    -1.4444444444
...
900 format(1x,a20," = ",f15.10)
...
```

Basic loop structures

```
j = 6
do i = 1, 10
   print*,i
   if(i==j) exit
end do
```

```
do i = 1, 10
   if(i<5) cycle
   print*,i
end do
```

```
i = 0
do while (i<3)
   i = i +1
end do
```

very useful !

e.g. **In force calculations particles don't exert force on themself, so for i:th particle you need to skip particle j if j=i**

If something goes wrong use **stop**:

```
subroutine mysub(x)
...
if(x<0.d0) stop 'mysub: negative x'
...
```

In bigger programs it's a good idea to let statements tell where they are ...

... and what exactly went wrong

Formatted output on screen

```
program test
  implicit none
  integer, parameter:: dbl=kind(1.d0)
  integer:: i
  real (dbl), parameter:: dx=0.1d0
  real (dbl) :: x
  write(*,'(2a15)') 'x','sin(x)'
  do i = 1, 10
    x = (i-1)*dx
    write(*,'(2f15.10)') x,sin(x)
  end do
end program test
```

Output format

```
              x         sin(x)
   0.0000000000   0.0000000000
   0.1000000000   0.0998334166
   0.2000000000   0.1986693308
   0.3000000000   0.2955202067
   0.4000000000   0.3894183423
   0.5000000000   0.4794255386
   0.6000000000   0.5646424734
   0.7000000000   0.6442176872
   0.8000000000   0.7173560909
   0.9000000000   0.7833269096
```

Subroutines and functions

```
subroutine swap(a,b)
   implicit none
   real(kind(1.d0)):: a,b,tmp
   tmp = a
        a = b
        b = tmp
end subroutine swap
```

in calling program:

```
  ...
  call swap(c,d)
  ...
```

```
real(kind(1.d0)) function poly(x)
   implicit none
   real(kind(1.d0)) :: x
   poly = x**6+x**3-x**2+x+5.d0
 end function poly
```

in calling program:

```
...
   integer, parameter:: dbl=kind(1.d0)
   real(dbl),external:: poly
   print*,poly(2)
...
```

!

- Arguments: “Pass by reference”, the content of a variable can *really* be changed
- Since Fortran 90: functions can return array valued results

  example: internal function for matrix multiplication

How to make a function that returns an array

Advanced feature

```
program test
  implicit none
  interface
    function sqroot(x) result(y)
      real(kind(1.d0)):: x(:)
      real(kind(1.d0)):: y(size(x))
    end function sqroot
  end interface
  integer:: i
  real(kind(1.d0)):: x(5)
  do i = 1, 5
    x(i) = 5.d0*i
  end do
  write(*,'("     x ",5f10.3)') x
  write(*,'("sqrt(x)",5f10.3)') sqroot(x)
end program test
```

```
function sqroot(x) result(y)
  implicit none
  integer:: i
  real(kind(1.d0)) :: x(:),y(size(x))
  do i = 1, size(x)
    y(i) = sqrt(x(i))
  end do
end function sqroot
```

The result is *not* called sqroot

No sqroot=...

Interface: need to tell the compiler that we want back an array y(:)

One cannot define

real(kind(1.d0)), external:: sqroot(5)

```
      x     5.000    10.000    15.000    20.000    25.000
sqrt(x)     2.236     3.162     3.873     4.472     5.000
```

Warning: don't compute sqrt like this; sqrt(x) does exactly the same !

Intrinsic functions are “elemental”

and know how to operate on many types of data

```
program test
  implicit none
  integer:: i,j
  real(kind(1.d0)):: array(4,4)
  do i = 1, 4
    do j = 1, 4
      call  random_number(array(i,j))
    end do
  end do
  write(*,'(4(4f5.2,/))') array,sin(array)
end program test
```

4x4 array

built-in random number generator

/ means newline

```
0.00 0.96 0.80 0.09
0.03 0.84 0.83 0.89
0.35 0.34 0.35 0.70
0.67 0.92 0.87 0.73
```

Original array

```
0.00 0.82 0.71 0.09
0.03 0.74 0.74 0.78
0.35 0.33 0.34 0.64
0.62 0.79 0.77 0.67
```

sin() of every element taken separately

```
print*,exp(1.0)                  2.718282
print*,exp(1.d0)                 2.71828182845905
print*,exp(cmplx(1.d0,2.d0))     (-1.131204,2.471727)
```

compilers don't like this: exp(1)

Functions are mostly generic:
Argument type determines
the output type
Exception: cmplx() !

Naming loops

```
iloop: do i = 1, 3
    jloop: do j = 1, 3
       kloop: do k = 1,3
          print*,i,j,k
          if(i==k) exit jloop
       end do kloop
    end do jloop
end do iloop
```

i	j	k
1	1	1
2	1	1
2	1	2
3	1	1
3	1	2
3	1	3

exit jloop

Program can **jump out of several nested loops**, not just the present loop (**if(i==k) exit** would do that)

Equally important : In a long program it's hard to tell what loop does a bare end do really end.

Passing functions as arguments

Often you need to tell a function or a subroutine to use a certain function, which is passed to them as an argument. A typical example is an integral.

```
! ------------------------------
! tests subroutine integral
! ------------------------------
program test
 implicit none
 real(kind(1.d0)):: pi,res,step,a,b
 intrinsic:: dsin
 pi   = 4.d0*atan(1.d0)
 step = 0.1d0
 a    = 0.d0
 b    = pi
 do
   call integral(dsin,a,b,step,res)
   write(*,'(2es25.16)') step,res
   step = 0.1d0*step
   if(step<1.d-8) exit
 end do
end program test
```

```
! --------------------------------------
! computes the definite integral
! of f(x) from a to b with step dx
! --------------------------------------
subroutine integral(f,a,b,dx,res)
  implicit none
  real(kind(1.d0)), intent(in):: a,b,dx
  real(kind(1.d0)),intent(out):: res
  real(kind(1.d0)), external :: f
  real(kind(1.d0)):: x
  res = 0.d0
  x   = a
  do
    res = res + f(x)
    x = x + dx
    if(x>b) exit
  end do
  res = dx*res
 end subroutine integral
```

intent(in) :variables we are *not allowed* to change (by mistake) inside the subroutine.

intent(out) : variables we *must* give a value

external :: f

tells the compile that **f** is not a table, but a function

```
1.0000000000000001E-01   1.9995479597125969E+00
1.0000000000000002E-02   1.9999900283082577E+00
1.0000000000000002E-03   1.9999999540411613E+00
1.0000000000000003E-04   1.9999999986726047E+00
1.0000000000000004E-05   1.9999999999843638E+00
1.0000000000000004E-06   1.9999999999895848E+00
1.0000000000000005E-07   2.0000000005709588E+00
1.0000000000000005E-08   2.0000000077288904E+00
```

As an advanced feature, the next page explains the reason for using **dsin** and not **sin**

Passing functions as arguments

Advanced feature

After just stating that functions are generic ... compilers are no infallible

```
program test
  implicit none
  real(kind(1.d0)):: x,sini
  intrinsic:: dsin  <------  !
  x = atan(1.d0)  ! this is pi/4
  print"('  argument ',f15.5)",x
  print"('    sin(x) ',f15.5)",sin(x)
  print"('own sin(x) ',f15.5)",sini(dsin,x)
end program test                              !

real(kind(1.d0)) function sini(f,x)
  implicit none
  real(kind(1.d0)),intent(in):: x
  real(kind(1.d0)),external:: f
  sini = f(x)
end function sini          !
```

```
  argument       0.78540
    sin(x)       0.70711
own sin(x)       0.70711
```

Try with **sin** instead of **dsin**:

- Intel ifort version 9.0 still compiles:

```
  argument       0.78540
    sin(x)       0.70711
own sin(x)      -0.74464 rubbish!
```

- g95 refuces to compile with **sin**:

  Internal error: g95_get_typenode(): Bad typespec

  But: **real(kind(1.d0)), intrinsic:: sin**

  and it *will* compile, with the same wrong result :(

Conclusion: Be careful to check that the program sends the correct intrinsic function!

The compiler may guess it wrong and send a single precision routine to a double precision routine

Modules

Modules replace the old **common** block structure: more versatile, easier to debug (both for you and the compiler). Typically one defines global parameters, data structures and utility functions in modules.

example of file modules.f90:

```
module kinds
 integer, parameter:: dbl=kind(1.d0)
end module kinds

module parameters
  integer, parameter:: natoms = 100   ! # of atoms
  integer, parameter:: ndim = 3        ! dimension
  integer, parameter:: ntherm=100     ! thermalization period
end module parameters

module positions
  use kinds
  use parameters, only: natoms, ndim
  real(dbl) :: r(ndim,natoms)
end module positions
```

module positions "inherits" the module kinds (so that we can use dbl)

only attribute: use only part of the module parameters

outline of file main.f90:

```
program main
  use kinds
  use parameters:: only: ntherm
  implicit none
  ...
end program main
```

Important: modules must be compiled before they can be used in a use statement

typically compile using order f90 modules.f90 main.f90

Modules may contain functions or subroutines:

keyword **contains**

For example spline subroutines and random number generators are easy to use f they are in a module

Outline of a spline module:

```
module splines
 contains
  subroutine spline(x,y,n,y2)
    ...
  end subroutine spline
  subroutine splint(x,y,y2,n,xx,fxx)
    ...
  end subroutine splint
end module splines
```

Outline of a random number module:

```
module random
 contains
  subroutine rng(x,n)
    ...
  end subroutine rng
end module random
```

First simulation – with graphics !

File gnusimu_example.f90

This simulates diffusion of non-interacting particles in 2D, starting from all particles in the same point (0,0).

Usage:

in unix shell, type *mkfifo fifo* , then *a.out &* and then *gnuplot < fifo*

Features:

- inquire statement : used to make sure the file fifo exists
- open the file 'fifo' with status='old': the file has to exist already

  status may be 'old' (must exist), 'new' (created now), 'unknown' (we don't know nor care if it is there or not)
- write gnuplot command (unset key *etc.*) to the file
- write particle positions in table r(2,n) to file one by one; actually r(2,n) is n two-dimensional vectors
- get a random shift dr(2) using the built-in random number generator random_number from range [0,1]
- shift dr by -0.5, because we want a random number [-0.5:0.5]
- move each particle amount dr : add vector dr to coordinate vectors r
- periodic boundary conditions: if a particle leaves the box, it re-enters from the other side

  realized using the where construct: where(condition) do-something

```
where(r(:,i)<-1.d0) r(:,i)  =  r(:,i)+2.d0
```

tests elements <u>separately</u>

applies operation to the element, which met the condition

Example:

```
integer:: a(2)
a(1) = 60
a(2) = 100              now a = 60, 100
where(a>70) a = a-10    now a = 60, 90
```

Limits of data types

Intrinsic functions tiny or huge tell how small or big values a data type can have.

```
program main
  implicit none
  integer :: i
  integer(8) :: j
  real:: x_single
  real(kind(1.d0)):: x
  print*,'          max i (integer) ',huge(i)
  print*,'   max j (integer kind 8) ',huge(j)
  print*,' max x (single precision) ',huge
(x_single)
  print*,' max x (double precision) ',huge(x)
  print*,' min x (single precision) ',tiny(x_single)
  print*,' min x (double precision) ', tiny(x)
end program main
```

Very rarely needed, but it's good to know a big integer is already there

```
          max i (integer)   2147483647
   max j (integer kind 8)   9223372036854775807
 max x (single precision)   3.4028235E+38
 max x (double precision)
1.797693134862316E+308
 min x (single precision)   1.1754944E-38
 min x (double precision)   2.225073858507201E-308
```

This is what happens after an integer exceeds it's upper limit:

```
program main
  implicit none
  integer :: i
  i = 1
  do
    if(i<0) exit
    i = i + 1
  end do
print*,i
end program main
```

This test should always fail, so the loop should never end!

```
-2147483648
```

positive integer turns negative !

Overflowing integers cause erratic behaviour

231 = 2147483648, so in binary the biggest *positive* i 2147483647 is 1111111111111111111111111111111 (30 1's)**

Limits of data types : cont'd

In double precision you can express numbers as small as 2.225073858507201E-308
and as big as 1.797693134862316E+308

Can there ever be a situation when these are not enough? The universe has about 10100 atoms !?**

BUT: Your variables may be able to express the single numbers,
but if you try to compute with *both* big and small numbers you get trouble

```
program main
  implicit none
  real(kind(1.d0)):: x,y
  x = 1.d23
  y = 1.d-11
  print*,x                    9.999999999999999E+022
  print*,y                    9.999999999999999E-012
  print*,(x-y)/x,y/x           1.00000000000000      9.999999999999999E-035
end program main
```

definitely NOT too much asked for double precision

not quite the result,
we lost y completely

computer couldn't store so small *deviation* from 1
in the form a*10b with 16 decimals in a**

Conclusion: Always choose units so that numerically stored numbers are about 1

Physical constants may be huge or tiny: Planck's constant, Avogadro number, Ångström *etc.*
Avoid using their tabulated SI unit values as such.

Allocating memory: automatic and manual allocation

```
subroutine swap(A,B,n,m)
  implicit none
  integer, intent(in):: n,m
  real(kind(1.d0)):: A(n,m),B(n,m)
  real(kind(1.d0)):: C(n,m)
  C = A
  A = B
  B = C
end subroutine swap
```

Matrices to be swapped

automatically allocated table:
after leaving the subroutine C
is also *deallocated* automatically

Same operation using manual allocation:

```
subroutine swap(A,B,n,m)
  implicit none
  integer, intent(in):: n,m
  real(kind(1.d0)):: A(n,m),B(n,m)
  real(kind(1.d0)),allocatable:: C(:,:)
  allocate(C(n,m))
  C = A
  A = B
  B = C
 deallocate(C)
end subroutine swap
```

Define that C will be have two indices

Allocate C

This may be left out, C will be *deallocated* anyhow
when the program leaves the subroutine

After leaving, the content of C is forgotten

The save attribute

Usually functions and subroutines automatically deallocate internal variables and their content is lost.
To have a variable that keeps it's content between subsequent call to a subroutine use the **save** attribute.
A typical example is a *counter,* that counts how many function calls have been made.

```
real(kind(1.d0)) function f(x)
  implicit none
  real(kind(1.d0)), intent(in):: x
  integer, save:: count=0
  f = sin(x)
  count = count + 1
  if(count>100) stop 'more than 100 calls'
end function f
```

Variable count will remain in memory between calls to function f
notice how count is initialized to 0

Another example is tables you want to allocate only once:

```
subroutine collect(vect,n)
  implicit none
 integer, intent(in):: n
  real(kind(1.d0)), intent(in)::vect(n)
  real(kind(1.d0)), allocatable, save, dimension(:):: vsum
  integer, save:: count=0
  if(count==0) allocate(vsum(n))
  vsum = vsum + vect
  count = count + 1
  write(*,*) count,vsum/count
end subroutine collect
```

Reading in data : read_1_example.f90

Very simple data input

```
program test
  implicit none
  real(kind(1.d0)):: x
  print*,'give x'
  read*,x
  print*,'ok, got ',x
end program test
```

$\leftarrow$ **Waits for a number,**
usually crashes on character input

Reading in data : read_2_example.f90

```
! ------------------------------
! Example program
! input from a file
! ------------------------------
program test
 implicit none
 integer:: io,i,k
 integer, parameter:: n = 10
 real(kind(1.d0)):: x(n),y(n)
 !
 ! read in x
 !
 open(12,file='input',status='old')
 do i = 1, n
   read(12,*,iostat=io) x(i),y(i)
   if(io/=0) then
     print*,'EOF while reading data'
     exit
   end if
 end do
 print*,'got ',i-1,' value pairs x,y'
 do k = 1, i-1
  write(*,'(2f7.3)') x(k),y(k)
 end do
end program test
```

If the file 'input' contains

```
1   1.23
2   2.22
3   3
4.5 4
5.2 5
6.1 6
7.2 7
8.1 8
```

the output is

```
EOF while reading data
got  8  value pairs x,y
 1.000  1.230
 2.000  2.220
 3.000  3.000
 4.500  4.000
 5.200  5.000
 6.100  6.000
 7.200  7.000
 8.100  8.000
```

Status is 'old': we don't want to create the data file with this program, just read it in.

Iostat=io puts to variable io a non-zero value if reading fails, in this case the file ends before all n data has been read in. The program prints out all values it got.

Extra functionality: rewind(12) rewinds the device 12 back to beginning, in this case the contents of the file 'input' could be read again.

Reading in parameters : namelist_example.f90

Namelist is a neat way to read in simulation parameters:

```
program test
  implicit none
  integer:: natoms=100, ndim=3
  real(kind(1.d0))::
boxx=0.d0,boxy=0.d0,boxz=0.d0
  namelist /para/ natoms, ndim, boxx,boxy,boxz

  read(*,nml=para)
  write(*,'(a21)') 'simulation parameters'
  write(*,'(21("="))')
  write(*,98) 'natoms',natoms
  write(*,98) 'ndim',ndim
  write(*,99) 'boxx',boxx
  write(*,99) 'boxy',boxy
  write(*,99) 'boxz',boxz

98 format(a10," = ",i8)
99 format(a10," = ",f8.3)
end program test
```

Default values

List of names that can appear in the namelist

Read in the namelist

Example: if the file 'data' contains

```
&para
ndim = 2,
natoms = 100,
boxx = 12.d0,
boxy = 10.d0
/
```

then the output of the command

a.out<data is

```
simulation parameters
=====================
    natoms =      100
      ndim =        2
      boxx =   12.000
      boxy =   10.000
      boxz =    0.000
```

The order of items is irrelevant, but the names have to match those given in the namelist statement

Reading and writing binary data files: binary_data_example.f90

A very effective way to store big chunks of data without losing accuracy in conversions

```
program test
 implicit none
 integer:: i,k
 integer, parameter:: n = 10
 real(kind(1.d0)):: x(n),y(n)
 !
 ! fill table x
 !
 do i = 1, n
   x(i) = dble(i)
 end do
 !
 ! write x to a binary data file
 !
 open(12,file='input.bin',form='unformatted')
 write(12) x
 rewind(12)
 read(12) y
 write(*,*) "Binary date write/read test"
 write(*,'(2a10)') 'wrote','read'
 do k = 1, n
   write(*,'(2f10.3)') x(k),y(k)
 end do
end program test
```

Creating a temporary binary file, named automatically:

```
 open(12,status='scratch', form='unformatted')
 write(12) myhugedataarray
 rewind(12)
 ... DON'T CLOSE THE SCRATCH FILE YET!
 read(12) anotherhugearray
 close(12)
```

form = 'unformatted' means binary data

form = 'formatted' means ascii data

No format for unformatted data :^)

we just dump x to disk

```
Binary date write/read test
    wrote      read
    1.000     1.000
    2.000     2.000
    3.000     3.000
    4.000     4.000
    5.000     5.000
    6.000     6.000
    7.000     7.000
    8.000     8.000
    9.000     9.000
   10.000    10.000
```

Endianness: a binary data issue

Advanced feature

This is too much for beginners, just skip it unless you already know what endianness or bit order means.

It's just a matter of concensus in whether binary data is written more significant or less significant bits first. In this respect Linux machines are usually configured to be natively *little endian*, but many other Unix machines are *big endian*. As a result, binary data written in one machine is unreadable (or reads in wrong) in the other unless you fix the problem. The best way to transfer binary data files among different machines is to use same endianness in both, defined via a specific *compiler option:*

Intel: compile programs using

```
ifort -convert big_endian program.f90
```

g95 : **g95 -fendian=big program.f90**

or

set the environment variable G95_ENDIAN

in bash shell : **export G95_ENDIAN=big**

in tcsh or csh : **setenv G95_ENDIAN big** (or: BIG)

then as usual **g95 program.f90**

Makefile

Helps compilation and maintenance of a program group spread over several files.

make looks for a file called Makefile or makefile - in which order seems to vary.

The command make takes the first target from the makefile and does what the target needs.

- **TARGET is what we want to make**
  - **A binary executable**
  - **an object file**
  - **Clean up, delete old object files, temporary data files *etc*.**
- **DEPENDENCY LINES tell what the target depends on**
  - **If you change a module, you *MUST* recompile every routine that depends on it**
- **RULES how to create a file**
  - **The binary executable depends on object files, we need to tell how object files *.o are made from the program files *.f90**

Syntax

```
TARGET: DEPENDENCIES
[press tab]  RULE
```

$\uparrow$ **!**

Example:

```
main.o : main.f90
        f90 -c main.f90
```

Why use a makefile: make compares the modification times (timestamps) of the targets and their dependencies and updates them *if necessary* by recompiling. Changes made to a routine will be made known to all routines that depend on it.

=> If you have 23 subroutines and you edit one of them you don't recompile every one of them.

Makefile cont'd

Example: **modules are in file modules.f90, main program in file main.f90, subroutines in files sub1.f90 and sub2.f90. The subroutine sub2.f90 doesn't use any modules from file modules.f90.**

```
#
# Example makefile    Comments begin with #
#
.SUFFIXES: .f90 .o     List iof suffixes to be searched for
F90 = g95     Name of the Fortran compiler
OPT = -O3     Compiler options
MOD = modules.o
OB1 = main.o sub1.o               aliases to collections of object codes
OB2 = sub2.o
PRO = $(MOD) $(OB1) $(OB2)         Call the whole set of routines PRO
BIN = simu                         Name of the binary to be created is simu
$(BIN): $(PRO)
        $(F90) $(OPT) $(PRO) -o $(BIN)     Rule how to make $(BIN), that depends on $(PRO)
$(OB1): modules.o       $(OB1) depends on modules.o : recompile $(OB1) if modules.o changes
.f90.o:
                                Implicit rule how to make .o files out of .f90 files (slightly old fashioned)
        $(F90) -c $(OPT) $*.f90
clean :                         Cleaning up: make clean will delete certain files, combination
        rm -f *.o               make clean
        rm -f *.mod             make
                                should recompile the whole program
```

Makefile cont'd

If you don't want to write the Makefile yourself, use a perl script "makemake": It will look through all programs in the directory and create a Makefile. This is what came out for the example program:

```
PROG =  simu
SRCS =  main.f90 modules.f90 sub1.f90 sub2.f90
OBJS =  main.o modules.o sub1.o sub2.o
LIBS =          If you use libraries this is where to add them
CC = cc                  Unnecessary but harmless
CFLAGS = -O
FC = f77
FFLAGS = -O
F90 = f90
F90FLAGS = -O
LDFLAGS = -s    Except this -s option theMakefile works just fine.
all: $(PROG)
$(PROG): $(OBJS)
          $(F90) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)
clean:
         rm -f $(PROG) $(OBJS) *.mod
.SUFFIXES: $(SUFFIXES) .f90
.f90.o:
         $(F90) $(F90FLAGS) -c $<
main.o: modules.o
                          Dependencies came out exactly as they should  :^)
sub1.o: modules.o
```

User-defined data types

Single numbers, characters and tables are often insufficient to express data in a natural way..
Often the program can be made more readable (=easier to debug) with better data structures.

Syntax:

```
type mytype
   variable definition (integers, character etc.)
end type mytype
```

Usage:

```
type(mytype):: one_of_mytype
```

Members are referred with the % sign,
one_of_mytype%member

Hint:
in your mind, read the % sign as if it were genetive:
Mikko%father means "Mikko's father"
atom%type means "atom's type"
atom(i)%coord(k) means "atom i's coordinate k"

```
program test
 implicit none
 type human
   integer::nchildren
   character(15) :: mother, father, child(20)
 end type human

 type(human):: Mikko
 Mikko%mother="Sarah"
 Mikko%father="Jack"
 Mikko%child(1) = "Roger"
 Mikko%child(2) = "Gillian"
 Mikko%nchildren = 2

 print*,"Mikko's father is ",Mikko%father
 print*,"Mikko's children are ",Mikko%child(1:Mikko%nchildren)
end program test
```

```
 Mikko's father is Jack
 Mikko's children are Roger          Gillian
```

BLAS Basic Linear Algebra Subroutines

Level 1 BLAS: scalar, scalar-vector or vector-vector operations
Level 2 BLAS: vector-matrix operations
Level 3 BLAS: matrix-matrix operations

Compile yourself (www.netlib.org) or use vendor-specific implementations:

AMD ACML

Apple Velocity Engine

Compaq CXML

Cray libsci

HP MLIB

IBM ESSL

Intel MKL

NEC PDLIB/SX

SGI SCSL or complib.sgimath

SUN Sun Performance Library

Pointers

Two usages of pointers:

- **Pointers can or sometimes must be used to replace ordinary allocatable variables**
  - **User-defined data structures cannot have the allocatable attribute, but must be defined using pointers**
  - **A function *cannot* return an allocatable table, but it can return**
    - **(i) an automatically allocated table**
    - **(ii) a pointer that has been dynamically allocated (example on the next page)**
  - **Pointers can be used to make linked lists**
- **Pointers as aliases to already allocated variables**
  - **as a name for a part of table**

Example:

N particles have 3 coordinates each and there are **Nconf** sets of these N particle simulations, everything stored in a big table **r_all**

```
real(kind(1.d0)), allocatable, target:: r_all(:,:,:)
real(kind(1.d0)), pointer:: r(:,:)
allocate (r_all(Nconf,3,N))
r => r_all(iconf,:,:)        !  r is now the coordinates of the set iconf
...
 all that is done here to r(k,i) affects automatically r_all(iconf,k,i)
...
```

Pointers cont'd

This example shows how to allocate memory in a function. A table cannot have the "allocatable" sttribute in a function, so one *must* use a pointer.

```
program test
  implicit none
  interface
    function func(n) result(tab)
      integer,intent(in):: n
      real(kind(1.d0)), pointer:: tab(:)
    end function func
  end interface
  integer,parameter:: dbl=kind(1.d0)
  real(dbl), pointer:: ptr(:)
  ptr => func(6)
  print*,ptr
  deallocate(ptr)
end program test

function func(n) result(tab)
  implicit none
  integer,intent(in):: n
  integer,parameter:: dbl=kind(1.d0)
  real(dbl), pointer:: tab(:)
  allocate(tab(n))
  tab = 1.d0
end function func
```

Here we create a 6 element table, allocate memory for it and set all elements to 1.d0

Output: 1. 1. 1. 1. 1. 1.

Cannot use "allocatable" here, must use "pointer"

Things that are different in this program:

- No "external" attribute to function func
- No "real(kind(1.d0))" typing of function func (not returning a single number)
- Explicit interface: func deals with a pointer and that needs to be told to the compiler
- *Any* kind of call to function func allocates memory, for example just print*,func(3) would allocate memory. Better free the memory in the end to be sure there is no memory leakage
- Give the name prt to the newly made table to be able to use it later
- Function func return a pointer, that points to an allocated memory block

Common blocks – a FORTRAN 77 construct

The use of common blocks should be avoided, and instead one should use modules.

One reason is that **common** declarations must be copied to every unit that uses them, in exactly the same order

common /params/ natoms,ndims in one unit and

common /params/ ndims,natoms in another

would cause the latter to think there are **natoms** dimensions and **ndims** atoms around, and the compiler has no idea something is wrong.

Copying the same common declarations around is usually replaced by an **include** statement.

In this case the common blocks are in a file, say common.f, and that file is always included in a program unit if necessary as shown below:

In file common.f

```
common /params/ natoms, ndims
common /lattice/ nx,ny,nz
```

In the program that uses the common variables

```
Program test
   include 'common.f'
   ...
```

This is almost ok, but the **include** statement is not part of the FORTRAN 77 standard, so the program may not be portable.

In practise howevere, this is not a serious problem, and Fortran 90 has include as standard.

All variables in a common block become accessible to all units having the same common block.

How to replace a common block with a module

FORTRAN 77 style

```
C
C   MAIN PROGRAM
C
      IMPLICIT REAL*8 (a-h,o-z)
      COMMON /data/ tab1(4),tab2(5)
      PRINT*,'This is the main routine and it just fills two tables
     $     in a subroutine and print them out'
      CALL fill
      PRINT*,'tab1'
      DO 50 i= 1, 4
 50     WRITE(*,'(5f7.3)') tab1(i)
      PRINT*,'tab2'
      DO 60 i= 1, 4
 60     WRITE(*,'(5f7.3)') tab2(i)
      END
C
C   Fills tables tab1 and tab2
C
      SUBROUTINE fill
      IMPLICIT REAL*8 (a-h,o-z)
      COMMON /data/ tab1(4),tab2(5)
      DO 5 I = 1, 5
        IF(i .LE. 4) tab1(i) = 1.d0*i**2
 5      tab2(i) = 1.d0*i
      RETURN
      END
```

FORTRAN 90/95 style

```
module data
  real(kind(1.d0)):: tab1(4),tab2(5)
end module data
!
!    MAIN PROGRAM
!
program main
  use data
  implicit none
  print*,'This is the main routine and it just fills two tables&
      & in a subroutine and print them out'
  call fill
  print*,'tab1'
  write(*,'(f7.3)') tab1
  print*,'tab2'
  write(*,'(f7.3)') tab2
end program main
!
!    Fills tables tab1 and tab2
!
subroutine fill
  use data
  implicit none
  integer:: i
  do i = 1, 5
    if(i <= 4) tab1(i) = dble(i**2)
    tab2(i) = dble(i)
  end do
end subroutine fill
```

Generic functions and subroutines: Overloading function names

Let's assume you have N types of data and you should do similar operations on any of them.
To begin with, you need N different subroutines, called oper_type1, oper_type2 ... oper_typeN.
In the program, you would have to call the correct subroutine for the type of data at hand.
This is not what you want to do. For example, if you want to compute the exponential of real numbers or complex numbers, you don't want to call the function "exp_real" and "exp_complex", just "exp". But you know already that the intrinsic exp() *does* all this, so there must be a way to define a generic subroutine oper, and use that instead of oper_type1, oper_type2 ... oper_typeN. This shows how to do it.

```
module operations
 interface oper
   module procedure oper_type1, oper_type2, oper_type3
 end interface
contains
 ! define operation for each data type
 subroutine oper_type1(a)
  declare here a to be of type 1
  ...
 end subroutine oper_type1
 subroutine oper_type2(a)
  declare here a to be of type 2
  ...
 end subroutine oper_type2
 etc.
end module operations
```

```
Program main
  use operations
  declare here x to be of some the N types
  declare here y to be of some the N types
  call oper(x)
  call oper(y)
end program main
```

You use the generic name oper for the operations, the compiler will automatically pick the correct subroutine from the list oper_type1, ... ,oper_typeN based on the type of the argument(s), here x or y.

All the dirty, uninteresting details are hidden in the module, the program itself is clean and easy to read

Many ways to do the same thing

Since Fortran 95 and Fortran 90 are downward compatible with FORTRAN 77, there are many syntactically different ways to do practically the same thing. The only difference is outlook and portability. I assume here, as always, that you are using a Fortran 95 compiler, *some forms are not FORTRAN 77.*

All these define a double precision real number x

```
double precision :: x
double precision x
real*8:: x
real*8 x
real (kind(1.d0)) :: x     recommended
real (kind(0.d0)) :: x     same thing
real(2) :: x               ok, but difficult to remember
```

All these do-loops do the same

"Line numbers"

```
  do 10 i = 1, 500
    print*,i
10 continue

  do 20 i = 1, 500
20 print*,i

do i = 1, 500              recommended
   print*,i
end do
```

All these define an integer i (of the same kind)

```
integer :: i             mostly used (see below)
integer i
integer(kind(1)) :: i
integer(4):: i
```

Old syntax don't allow variable to be initialised

This is ok:

```
integer:: i=12345
```

This is WRONG:

```
integer i=12345
```

REMARK:
Most compilers use 32 bit integers, but some new ones may use 64 bit integers by default. The latter can break old programs. If in doubt, printing **bit_size(i)** will tell you how many bits integer **i** uses. Another useful inquiry function is **sizeof(i).**

Loop over real points

The problem is that real variables should not be used as loop variables. Consider a BAD loop over real numbers x,

```
  do x=0.d0, 1.d0, 0.1d0  ! NEVER USE THIS
     ...
  end do
```

What is the last value of x? You might think it's x=1, but it's not that simple. If the addition of 0.1 to x gives a value 1.0000000000000001, to the computer this is above 1.d0 so it won't come out as x => the biggest x *in this case* is about 0.9. Due to this numerical glitch the loop may end at a different x in different machines, compilers or optimizations.

=> Unpredictable behaviour – compilers warn you

Here are a few common ways to make a loop that runs over evenly spaced real number grid points **x= a, a+dx, a+2dx , ... , b** with **n** points **x_i.** The lowest **x** is always **a**, the last **x** may be **b** or less.

a, b and dx are given, any index

```
x = a
do
   program lines that use x
   x = x + dx
   if(x>b) exit
end do
```

a, b and dx are given, uses index i

```
n = (b-a)/dx+1
do i = 1. n
   x = a+(i-1)*dx
   program lines that use x and i
end do
```

a, b and n are given

```
dx = (b-a)/(n-1)
do i = 1, n
   x = a + (i-1)*dx
   program lines that use x
end do
```

a, dx and n are given, b unknown

```
do i = 1, n
   x = a + (i-1)*dx
   program lines that use x
end do
```

Advanced feature

LAPACK95

LAPACK95 provides Fortran 95 style use of FORTRAN 77 routines in LAPACK/BLAS

Interfaces make possible to have : - Generic names to single and double precision routines
- Optional arguments

Drivers allocate work space and make calls to the F77 subroutines that do the actual computation

Why use LAPACK95?

- **Your routines don't get messy with work space allocations/deallocations, those are now done in the driver routine you never need to see**
- **You don't have to worry about using a routine of wrong precision, the compile can check for it**
- **The beauty of optional arguments: just leave out things you don't want.**
  For example, instead of calling the F77 routine with an argument that tells you *don't* want eigenvectors, you just call a generic f95 subroutine where the eigenvector argument is not present:
  Example: Diagonalise a tridiagonal matrix, diagonal elements in table d, off-diagonal elements in table e

  call la_stev(d,e) **computes only the eigenvalues, returned in table d(:)**

  call la_stev(d,e,z) **computes eigenvalues (in table d(:)) and eigenvectors (in table z(:,:)**

  call la_stev(d,e,z,info) **same a above, but with integer info returned**

  (if info is not present and the program fails, it will terminate and print an error message)
- **The LAPACK95 drivers have a better chance to be the same in the future, even if someone changes the computing routines in LAPACK**

The generic LAPACK95 routines are called la_XXXX , the "la_" is there to distinguish from LAPACK names

Advanced feature

LAPACK naming scheme

Taken from www.netlib.org

The name of each LAPACK95 routine has been made as similar as possible to its name in LAPACK. All driver and computational routines have names of the form LA_YYZZZ, where for some routines the 8 character is blank. The two letters YY indicate the type of matrix (or of the most significant matrix). Most of these two-letter codes apply to both real and complex matrices; a few apply specifically to one or the other, as indicated in Table 2.1.

Table 2.1: Matrix types in the LAPACK naming scheme

GB general band
GE general (i.e., unsymmetric, in some cases rectangular)
GG general matrices, generalized problem (i.e., a pair of general matrices)
GT general tridiagonal
HB (complex) Hermitian band
HE (complex) Hermitian
HP (complex) Hermitian, packed storage
PB symmetric or Hermitian positive definite band
PO symmetric or Hermitian positive definite
PP symmetric or Hermitian positive definite, packed storage
PT symmetric or Hermitian positive definite tridiagonal
SB (real) symmetric band
SP symmetric, packed storage
ST (real) symmetric tridiagonal
SY symmetric
When we wish to refer to a class of routines that perform the same function on different types of matrices, we replace the two letters by ``yy''. Thus LA_yySV refers to all the simple driver routines for systems of linear equations that are listed in Table 2.2. The last three letters ZZZ indicate the computation performed.

LAPACK95 cont'd : Example of an interface

Advanced feature

Generic name for two routines:
- Double precision routine
- Double Complex routine

DP stands for “Double Precision”

Optional arguments: may be present of not

```
INTERFACE LA_GEGS

  SUBROUTINE DGEGS_F95( A, B, ALPHAR, ALPHAI, BETA, VSL, VSR,      &
&                  INFO )
    USE LA_PRECISION, ONLY: WP => DP
    INTEGER, INTENT(OUT), OPTIONAL :: INFO
    REAL(WP), INTENT(INOUT) :: A(:,:), B(:,:)
    REAL(WP), INTENT(OUT), OPTIONAL :: ALPHAR(:), ALPHAI(:),      &
&                             BETA(:)
    REAL(WP), INTENT(OUT), OPTIONAL, TARGET :: VSL(:,:), VSR(:,:)
   END SUBROUTINE DGEGS_F95


  SUBROUTINE ZGEGS_F95( A, B, ALPHA, BETA, VSL, VSR, INFO )
    USE LA_PRECISION, ONLY: WP => DP
    INTEGER, INTENT(OUT), OPTIONAL :: INFO
    COMPLEX(WP), INTENT(INOUT) :: A(:,:), B(:,:)
    COMPLEX(WP), INTENT(OUT), OPTIONAL :: ALPHA(:), BETA(:)
    COMPLEX(WP), INTENT(OUT), OPTIONAL, TARGET :: VSL(:,:),      &
&                                      VSR(:,:)
   END SUBROUTINE ZGEGS_F95

 END INTERFACE
```

FORALL: a new f95 program structure

FORALL is a loop construct for doing things that can be done in parallel, meaning that
the order of doing things is irrelevant. Even in a single processor machines a good compiler
can take advantage of the many registers in the processors to do many tasks at once.
This was the idea. In reality compilers are much better at optimising DO's that FORALL's,
so you probably find out that a FORALL is either as fast or slower than DO - sigh.

```
program test
   implicit none
   integer, parameter:: n=7000
   integer:: i
   real(kind(1.d0)):: A(n)
   forall (i=1:n)
     A(i) = i**2
   end forall
end program test
```

**We can use forall, because the values of elements A(i) are independent.
For example, it doesn't matter whether we set A(1000) = 1000**2 before or after
we set A(2123) = 2123**2 .**

The forall loop above can also be written on a single line:

```
forall(i=1:n) A(i)
=i**2
```

SELECT CASE structure

A multiple-choice branching of the program flow. Often mor readable than a sequence of IF tests.

```
integer:: i
...
select case(i)
  case (:-1)
     print*,'i is negative'
  case (0)
     print*,'i is zero'
  case(1:)
     print*,'i is positive'
  case default
     print*,'i is probably NaN'
end select
...
```

case (:-1) means all numbers ..., -4,-3,-2,-1

If none of the cases before matched

Numerical Recipes *by Press, Teukolsky, Vetterling and Flannery, see* **www.nr.com**

Fortran 77, C, C++ and Fortran 90

- **A collection of subroutines for numerical tasks : eigenvalues, FFT, random numbers *etc.***
- **Very well known among the numerical community**
- **The efficiency of the programs divides opinions:**
  They get the job done, but are not quaranteed to give the best method for the specific task
- **The manual is a nice bedside reading for those interested in numerical methods: some of the NR books are also free online**
- **To obtain a copy of the subroutines buy them with the book – many machines have them installed already**

FFTW *see* **www.fftw.org**

Written in C, Fortran wrappers

- **“Fastest Fourier Transform in the West”**
- **Some vendor-tuned FFT's may be faster, but FFTW is at least almost as fast and <u>portable</u>**
- **If your program spends appreciable time in doing FFT's, use FFTW**
  If not, use any simple FFT, like the one in Numerical Recipes
- **Older version 2.x has different API than the newer 3.x**

Preprocessing:
The difference between program.f90 and program.F90

- **program.f : fixed form, things have to be in specific columns**
  a FORTRAN 77 routine
- **program.F : The compiler is instructed to do preprocessing to make a .f file out of the .F file (visible or not)**

- **program.f90 : free form (no column specific rules, but keep them readable and use intendation!)**
  a Fortran 90 routine
- **program.F90 : The compiler is instructed to do preprocessing to make a .f90 file out of the .F90 file**

Preprocessing is a way to make choices at compile time

For example, you may want to use the FFT provided by the ESSL library in IBM and the FFTW library in other machines

!

Example: file program.F90

Preprocessor directive

```
program test
#ifdef PARA
   print*,'here I will put my parallel code once I learn how ...'
#else
   print*,'this is my serial machine code'
#endif
   print*,'this part is done in both serial and parallel cases'
end program test
```

Compile the example using

a) f90 program.F90 -o goserial

and

b) f90 -DPARA program.F90 -o gopara

and you have two different binary programs, goserial and gopara

Try also cpp program.F90 and cpp -DPARA program.F90

Don't put preprocessor directives in a file with suffix .f90:

```
fortcom: Warning: badexample.f90, line 3: Bad # preprocessor line
#ifdef PARA
-^
```

Numerical constants

Be careful not to lose accuracy in numerical constants. Usually it's safe to have constant *two* represented as 2, 2.0 or 2.d0 (rarely as 2.q0), but sometimes the decimals are nonzero and the suffix (none, d0 or q0) tells how many of the decimals are actually kept.

Example

```
program test
  implicit none
  write(*,'(f50.45)') 0.11111111111111111111111111111111111111111111111111      Single precision constant
  write(*,'(f50.45)') 0.11111111111111111111111111111111111111111111111111d0    Double precision constant
  write(*,'(f50.45)') 0.11111111111111111111111111111111111111111111111111q0    Quadruple  precision constant
end program test
```

Practically random numbers

Output:

```
0.111111111938953399658203125000000000000000000
0.111111111111111104943205418749130331000000000
0.111111111111111111111111111111111105800000000
```

Numerical constants, cont'd

If you have a hard-wired constant always define it as a parameter:

- **You don't risk changing a parameter by mistake**
- **If you deside to change, say, 110 to 220 you don't want to edit all files that had that 110. There is a risk that you miss one or more occurrunces of that "110".**

Never ever do like this:

```
program test
  implicit none
  integer:: i,index(100)
  do i = 1, 100
      index(i)= i
   end do
   call rotateindex(index)
   write(12,*) index
end program test
subroutine rotateindex(index)
  implicit none
  integer:: index(100),i
  i = index(1)
  index(1:99) = index(2:100)
  index(100) = i
end subroutine rotateindex
```

If you wanted to change 100 to 101, then even in this short program you would have to edit all these 6 points to get it working again.

Remark 1: You cannot replace all "100" by "101" automatically, it's not enough – and often it's too much !

Remark 2: The subroutine works only for a 100 element input: a very dull idea.

Do it like this:

Change only this and you are done

```
program test
  implicit none
  integer, parameter:: n = 100
  integer:: i,index(n)
  do i = 1, n
      index(i)= i
   end do
   call rotateindex(index,n)
   write(12,*) index
end program test
subroutine rotateindex(index,n)
  implicit none
  integer:: n, index(n),i
  i = index(1)
  index(1:n-1) = index(2:n)
  index(n) = i
end subroutine rotateindex
```

This is just an example. If you want to rotate indices use the built-in function "shift".

The scope of variables

When is a variable visible to a function or a subroutine?

This won't work:

```
Program test
  implicit none
  integer:: i,k=100
  do i = 1, 5
     call addone
     print*,k
  end do
end program test

subroutine addone
  integer:: k
  k = k + 1
end subroutine addone
```

Variable k in the main program and the variable k in the subroutine addone are not the same. In fact, the k in the subroutine is completely arbitrary!

Output:

```
100
100
100
100
100
```

Add here “print*,k” to see what the subroutine thinks k is

There is no way this subroutine, which is outside the main program, receives any information about the variable k.

The scope of variables, cont'd

When is a variable visible to a function or a subroutine (or to the calling program unit)?

This works:

```
program test
 implicit none
 integer:: i,k=100
 do i = 1, 5
    call addone
    print*,k
 end do
 contains
   subroutine addone
     k = k + 1
   end subroutine addone
end program test
```

This k is visible to the subroutine, because ...

... the subroutine is contained in the main program and sees all its variables.

Output:

```
 101
 102
 103
 104
 105
```

And how easy it is to break it again!

```
program test
 implicit none
 integer:: i,k=100
 do i = 1, 5
    call addone
    print*,k
 end do
 contains
   subroutine addone
     integer:: k
     k = k + 1
   end subroutine addone
end program test
```

BAD! → integer:: k

Why bad? The added line integer:: k defined another variable k; this new k is an internal variable of the subroutine.

Output:

```
 100
 100
 100
 100
 100
```

The scope of variables, cont'd

Advanced feature

Private and public variables: yet another attribute a variable can have

A variable may be defined to be private or public to control it's visibility to other program units.

```
module params
!  real(kind(1.d0)),private:: salary,taxes
  real(kind(1.d0)):: salary,taxes
contains
  function taxprocent()
    salary = 13000.d0
    taxes  = 4210.d0
    taxprocent = taxes/salary*100.d0
  end function taxprocent
end module params


program test
  use params
  implicit none
  write(*,'("Tax Procent is ",f0.2)') taxprocent
()
  write(*,'("     Salary is ",f0.2)') salary
end program test
```

The commented line has an additional attribute “private”; It would state that salary and taxes are hidden from any users of the module params. In that case the main program couldn't print out the salary (compiler gives an error message)

All variables in a module are by default public (visible to anyone who uses the module)

You can change the default from public to private:

```
Module something
  private
  real(kind(1.d0)):: ...
  integer::...
end module something
```

Or you can specify the visibility of each variable:

```
Module something
  real(kind(1.d0)), public::  ...
  integer, private:: ....
end module something
```

Good programming habits

- **A good main program is a collection of subroutine calls.**
  The main program is a leader, it sees over that the principal tasks get gone in correct order.
  => Don't put any lengthy code segments to the main program.

```
program main
     use parameters
     implicit none
     integer:: i
     call initialize
     do i = 1, Niter
         call iterate
         if(i>Ntherm) call measure
     end do
     call errorestimation
     call saveresults
 end program
```

- **Always INDENT programs. Not too much, not too little. Nobody likes to read unindented program.**
- **Choose descriptive, *short* names. Name loops that end far away from start**
- **Keep formulas close to their mathematical form. That form has been around for decades and shows *what* you compute.**

NEVER WRITE

```
do i = 1, n
    y(i) = sin(i+j/20*n+4)*cos(i+j/20*n+4)*exp( i+j/20*n+4)
end do
```

BUT WRITE

```
do i = 1, n
    x = i+j/20*n+4
    y(i) = sin(x)*cos(x)*exp(x)
end do
```

- **Comment things that are not obvious.**
- **Don't comment everything. Your program is not a Fortran manual. If the reader doesn't know Fortran, it's not *your* fault.**

Using only some variables from a module

Syntax:

use module_name, only: list_of_variables

Example:

```
module params
  integer, parameter:: dp=kind(1.d0)
  real(dp),parameter:: pi=3.14159265358979323 8d0
  real(dp):: x,y,z
end module params

program test
  use params, only: x
  implicit none
  x = 10.d0
  print*,x
end program test
```

The restriction only means we take from the module params only x. If we wanted, we could define a variable y within the main program without touching the other y in the module.

The constants dp and pi and variables y and z are not made known to the main program.

Terminology

<u>**About arrays**</u>

Rank	= The number of array dimensions
	integer,allocatable:: index(:,:,:) array index has rank 3
Shape	= Tells what indices does the array cover
	integer :: index(5,6,2) array index has shape (5,6,2)
Explicit shape array	= The upper bound of each dimension is specified
	integer:: index(5,6,2) or integer:: index(k,l,m)
	Simple, but not very versatile; sometimes require a lot of checking by you
Assumed size array	= A Fortran 77 relict. The upper bound of some dimension is not specified
	integer:: index(5,6,*)
	Avoid, use assumed shape arrays instead
Assumed shape array	= The rank is specified, but not the shape
	integer:: index(:,:,:) can have any shape, such as index(5,6,2) or index(1,2,2) or ...
	Use frequently! Put these in modules or write an explicit interface.
	Remember also, that all lower bounds are 1 by default, unless told otherwise.

<u>**About functions and subroutines**</u>

Dummy argument	= Argument in the *declaration* of a function or a subroutine, a temporary name.	
	real(kind(1.d0)) function cube(x)	here x is a dummy argument
	subroutine mysub(input,output)	here input and output are dummy arguments
Actual argument	= Argument in the *call* of a function or a subroutine, the name of the thing that is actually sent to the function or the subroutine to operate on	
	x3 = cube(x)	here x is the actual argument
	call subroutine mysub(testinput, outcome)	here testinput and outcome are actual arguments

Command line arguments

Sometimes writing a namelist or a parameter file is too clumsy. For example, if you have written a program "sort" that sorts a file containing numerical data to ascending order according to some column, then a natural way to use it is from a command line

```
> sort file column
```

Below is a short example how to read in the command line – without any testing of error conditions to keep it basic.

```
Program main
  implicit none
  character(100):: buffer
  character(25):: filename
  integer:: column

  call getarg(1,buffer)
  read(buffer,*)  filename
  call getarg(2,buffer)
  read(buffer,*) column
  ...
end program main
```

A long string to hold the command line

Two command line arguments in this example

Notice how one reads from a character string "buffer" !

Internal read and write

One can read and write to and from other "devices" than the keyboard and files. One can also read from and write to internal sources. The example shows how to write to a character string.

```
program main
  implicit none
  integer:: i
  character(50):: s
  print*,'give an integer (not too long)'
  read(*,*) i
  write(s,'(i0)') i
  s = 'prefix.'//trim(s)//'.suffix'
  print*,'Making the character string "prefix.i-value.suffix"'
  print*,'The result is ',trim(s), ' as it should'
 end program main
```

!

s is 50 characters long => lots of blanks to remove

```
>a.out
 give an integer (not too long)
325636
 Making the character string "prefix.i-
value.suffix"
 The result is prefix.325636.suffix as it should.
```

Strict "i0" format together with trim(s) removed blanks that would otherwise spoil the output

string1//string2   use "//" to glue strings together

trim(string) removes trailing blanks from string

Few words about parallel programming

unpopular

- **HPF = High Performance Fortran**

  A collection of parallelisation directives on top of Fortran 90/95

  Shows in the program as lines starting

  !HPF$

Online course: http://www.liv.ac.uk/HPC/HPFpage.html

http://dacnet.rice.edu/Depts/CRPC/HPFF/index.cfm

popular complicated

- **MPI, especially Open MPI**

MPI = Message Passing Interface

www.open-mpi.org

Open MPI is a free implemenation of MPI

www.tu-darmstadt.de/hrz/hhlr/doku/sw/pe/am106mst02.html

Open MPI Represents the merger between three well-known MPI implementations:

FT-MPI from the University of Tennessee

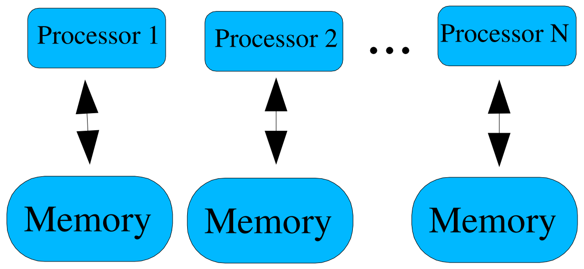
LA-MPI from Los Alamos National Laboratory

LAM/MPI from Indiana University

Can be installed even on single processor machines :^)

More about MPI on the next slide!

Distributed-memory systems



Simple

- **OpenMP**

MP = Multi Processing

www.openmp.org

A shared-memory parallel programming environment

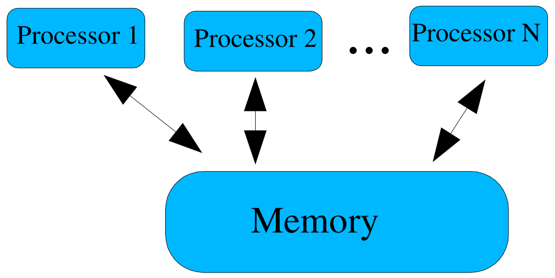
Parallelisation directives

Library routines

Shows in the program as lines starting

!$OMP **(in C they start #pragma omp)**

Shared-memory systems



MPI: *de facto* standard of parallel programming

Only choice for distributed-memory systems: All processors access their own memory; *e.g.* PC clusters

We have *nodes* processors (nodes): master is 0, slaves are *1,2,...,nodes-1*. Also called root and workers.
The parallel program calls MPI subroutines that initialise, distribute input data and collect results. The set of nodes make a group called *comm* and the node (root or a slave) where the code is running is number *rank*.

Some common MPI subroutines: (ierr=integer for an error, see *man mpi_XXXX* for more information on the other arguments)

mpi_init(ierr) Start MPI

mpi_comm_size(comm,nodes,ierr) How many nodes belong to the group comm (usually: # of processors you use)

mpi_comm_rank(comm,rank,ierr) Get the rank (0, 1, ... nodes-1) of the current processor

mpi_bcast(buffer, count, datatype, master,comm, ierr) Broadcasts a message (data) from the master to all other processes of the group

mpi_barrier(comm,ierr) All processors wait at the barrier for the rest to arrive

mpi_reduce(sendbuf,recvbuf,count,datatype,op,master,comm,ierr) Reduces values on all processes to a single value, op = operation (mpi_sum)

mpi_finalize(ierr) End parallel execution

----------------- A parallel program can be made with just the 7 subroutines given above ---

mpi_send(buf,count,datatype,dest,tag,comm,ierr) Basic send operation

mpi_recv(buf, count,datatype,source,tag,comm,status,ierr) Basic receive operation

mpi_scatter(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,master,comm,ierr) Scatter data from one task to all other tasks in a group

or mpi_scatterv(sendbuf,sendcounts(*),displs(*),sendtype,recvbuf,recvcount,recvtype,master,comm,ierr) Scatter data in parts

Just for fun

This is an excerpt from the Open MPI FAQ at www.openmpi.org, showing what problems generic names (subroutine overloading) may cause.

14. Why does compiling the Fortran 90 bindings take soooo long?

This is actually a design problem with the MPI F90 bindings themselves.
The issue is that since F90 is a strongly typed language, we have to overload each function that takes a choice buffer with a typed buffer.
For example, MPI_SEND has many different overloaded versions -- one for each type of the user buffer.
Specifically, there is an MPI_SEND that has the following types for the first argument:

logical*1, logical*2, logical*4, logical*8, logical*16 (if supported)
integer*1, integer*2, integer*4, integer*8, integer*16 (if supported)
real*4, real*8, real*16 (if supported)
complex*8, complex*16, complex*32 (if supported)
character

On the surface, this is 17 bindings for MPI_SEND. Multiply this by every MPI function that takes a choice buffer (50) and you 850 overloaded functions. However, the problem gets worse -- for each type, we also have to overload for each array dimension that needs to be supported. Fortran allows up to 7 dimensional arrays, so this becomes (17x7) = 119 versions of every MPI function that has a choice buffer argument. This makes (17x7x50) = 5,950 MPI interface functions.

To make matters even worse, consider the ~25 MPI functions that take 2 choice buffers. Functions have to be provided for all possible combinations of types. This then becomes exponential -- the total number of interface functions balloons up to 6.8M.

Additionally, F90 modules must all have their functions in a single source file. Hence, all 6.8M functions must be in one .f90 file and compiled as a single unit (currently, no F90 compiler that we are aware of can handle 6.8M interface functions in a single module).

To limit this problem, Open MPI, by default, does not generate interface functions for any of the 2-buffer MPI functions. Additionally, we limit the maximum number of supported dimensions to 4 (instead of 7). This means that we're generating (17x4*50) = 3,400 interface functions in a single F90 module. So it's far smaller than 6.8M functions, but it's still quite a lot.

This is what makes compiling the F90 module take so long.

OpenMP

Shared-memory systems: all processors access the same memory; e.g. Origin 2000 , IBM PS

- Ease of use
- Incremental parallelization
- Easy speedup on desktops (dual core machines, threads) - you can start low
- Scales well on multiprocessor machines - you can get high

Intel f90 compiler (and many other) supports OpenMP directives. Since the directives look like ! $omp xxxxxxxx
a compilation using

```
ifort program.f90        (this fails if there are calls to openMP library routines)
```

produces ordinary (serial) code, while compilation using

```
ifort -openmp program.f90
```

produces parallel code following the directives. => You have *both* a serial and a parallel version of your program.

Excerpt from the openMp sample routine ; molecular dynamics www.openmp.org/drupal/samples/md.html
By Bill Magro, Kuck and Associates, Inc. (KAI), 1998

This loop updates the positions and velocities of the particles using the velocity Verlet algorithm

```
       ...
       ! The time integration is fully parallel
!$omp  parallel do
!$omp& default(shared)
!$omp& private(i,j)
      do i = 1,np
        do j = 1,nd
          pos(j,i) = pos(j,i) + vel(j,i)*dt + 0.5*dt*dt*a(j,i)
          vel(j,i) = vel(j,i) + 0.5*dt*(f(j,i)*rmass + a(j,i))
          a(j,i) = f(j,i)*rmass
        enddo
      enddo
!$omp  end parallel do
      ...
```

OpenMP directives for parallelization of the loop

```
> setenv OMP_NUM_THREADS 8        (run with 8 threads/processors)
> ifort -openmp md.f
md.f(92) : (col. 7) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
md.f(209) : (col. 7) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
```

(the other parallelized loop was force calculation)

OpenMP timings

program md.f (openMP sample program)

lilli : SGI Altix 3000 Intel Itanium 2, 900MHz/1.5MB L3 Cache

origin: SGI Origin 3800 R12000 400MHZ

compiled: lilli ifort -fast -openmp md.f

origin f90 -Ofast=ip27 -mips4 -64 -mp md.f

(f90 = MipsPro)

Execution times in seconds

#proc	lilli	origin
1	57	72
2	35	36
4	18	19
8	10	9
16	6	5

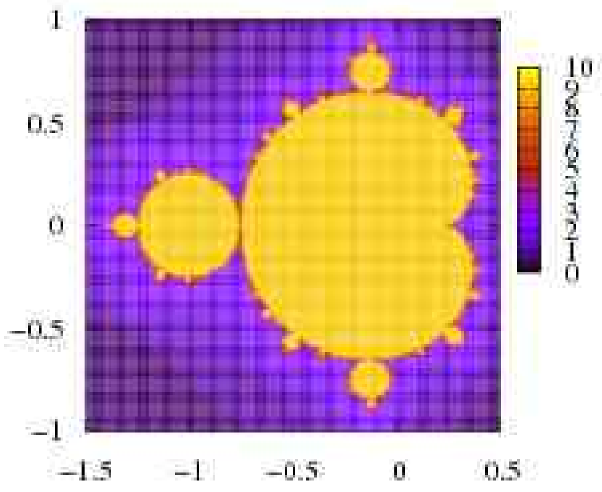
Excellent scaling !

Complex arithmetics: application to Mandelbrot set

```fortran
program mandel
 implicit none
 integer, parameter:: dp=kind(1.d0),ix=10000 ! Max # of iterations
 ! resolution d, box corners [x0,y0] and [x1,y1]
 real(dp),parameter::d=1.d-2,x0=-1.5d0,x1=0.5d0,y0=-1.d0,y1=1.d0
 integer:: i
 complex(dp):: cx,c
 real(dp):: x,y
 x = x0
 do
   y = y0
   do
     c  = cmplx(x,y,dp)
     cx = c
     do  i = 1, ix
       cx = cx**2+c
       if(abs(cx)>2) exit
     end do
     write(12,'(2f8.3,i10)') c,i
     y = y + d
     if(y>y1) exit
   end do
   write(12,*)
   x = x + d
   if(x>x1) exit
 end do
end program mandel
```

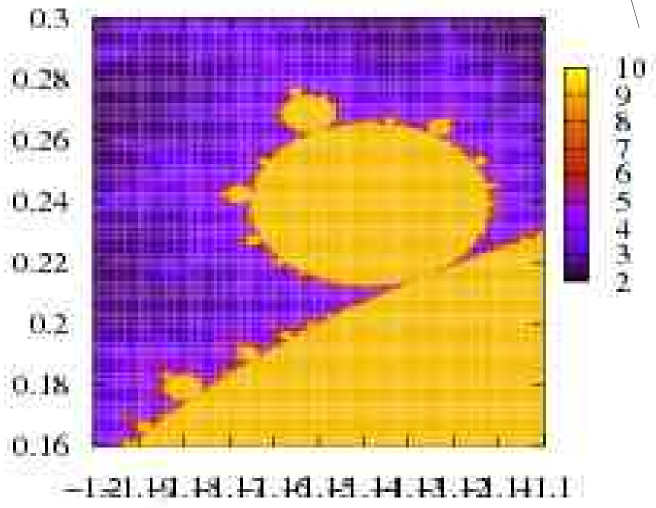
Plotting using gnuplot:

```
>gnuplot
gnuplot > set size ratio 1
gnuplot > set pm3d map
gnuplot > unset key
gnuplot > sp 'fort.12' u 1:2:(log($3))
```



d=1.d-2,x0=-1.5d0,x1=0.5d0,y0=-1.d0,y1=1.d0

d=5.d-4,x0=-1.20d0,x1=-1.1d0,y0=0.16d0,y1=0.3d0



Notice how close the program is to the mathematical algorithm!

Algotrithm

cx=c

$cx=cx^2+c$ iterate max ix iterations

if |cx|>2 the point c is definitely outside the set

store how many iteration if takes to get out, i

Mandelbrot set : A parallel version

Intel f90 compiler (version 9) can autoparallelize code via openMP. The previous Mandelbrot program won't autoparallelize (fastest way to see it is just to try), so let's tweak it a bit.

There are two problems to fix:

- IO from innermost loop : which processor/thread is supposed to do that?
- The outer do-loops had no loop index, instead there were exit tests **if(x>x1) exit**

  If the x-loop is parallelized each process must exit the loop at a different x, which is not x1.

```
program mandel
 implicit none
 integer, parameter:: dp=kind(1.d0),imax=10000
 real(dp),parameter:: d=1.d-2,x0=-1.5d0,x1=0.5d0,y0=-1.d0,y1=1.d0
 integer:: i,ix,iy,nx,ny
 complex(dp):: cx,c
 real(dp):: x,y ,t0,t1
 integer,allocatable:: set(:,:)
 nx = (x1-x0)/d + 1
 ny = (y1-y0)/d + 1
 allocate(set(nx,ny))
 do ix = 1, nx
   x = x0+(ix-1)*d
   do iy = 1, ny
     y  = y0+(iy-1)*d
     c  = cmplx(x,y,dp)
     cx = c
     do  i = 1, imax
       cx = cx**2+c
       if(abs(cx)>2) exit
     end do
     set(ix,iy)= i
   end do
 end do
 ... cut away: output result table "set"   to file...
end program mandel
```

Clearly independent loop variables ix and iy

No file output

Table set can be filled in arbitrary order

Output of intel ifort version 9.0

Optimize

Autoparallelize

Print out a report about autoparallelization

```
> ifort -O3 -parallel -par_report3 mandel.para.f90
   procedure: mandel
   serial loop: line 26: not a parallel candidate due to the loop being lexically discontinuous
   serial loop: line 22: not a parallel candidate due to insufficent work
   serial loop: line 38: not a parallel candidate due to statement at line 40
   serial loop: line 36: not a parallel candidate due to statement at line 40
mandel.para.f90(20) : (col. 3) remark: LOOP WAS AUTO-PARALLELIZED.
   parallel loop: line 20
      shared     : { "I.SI32.var$16_dv_template.dim_info.lower_bound.betype.0.0"
"I.SI32.var$16_dv_template.dim_info.spacing.betype.0.0" "P.P32.var$16_dv_template.addr_a0.betype.0.0" }
      private    : { "ix" "x" "iy" "y" "c" "cx" "i" }
      first priv.: { }
      reductions : { }
```

The i loop

The iy loop

The write statement

The ix loop

Result is a working parallel binary, just set the environment variable OMP_NUM_THREADS to the number of processors/threads and run.