From MATLAB to FORTRAN 90/95

Contrasting MATLAB and Fortran

Ge Baolai
SHARCNET
The University *of* Western Ontario

Agenda

- MATLAB vs. FORTRAN at A Glance
- FORTRAN Language Elements Highlights
- File Organization
- Compiling and Running FORTRAN Programs
- Some Performance Comparisons
- References

SHARCNET Literacy Series

Scientific and High Performance Computing

Modified for
Presentation via AccessGrid
The University *of* Western Ontario
London, Ontario
March 5, 2007

MATLAB vs. Fortran at A Glance

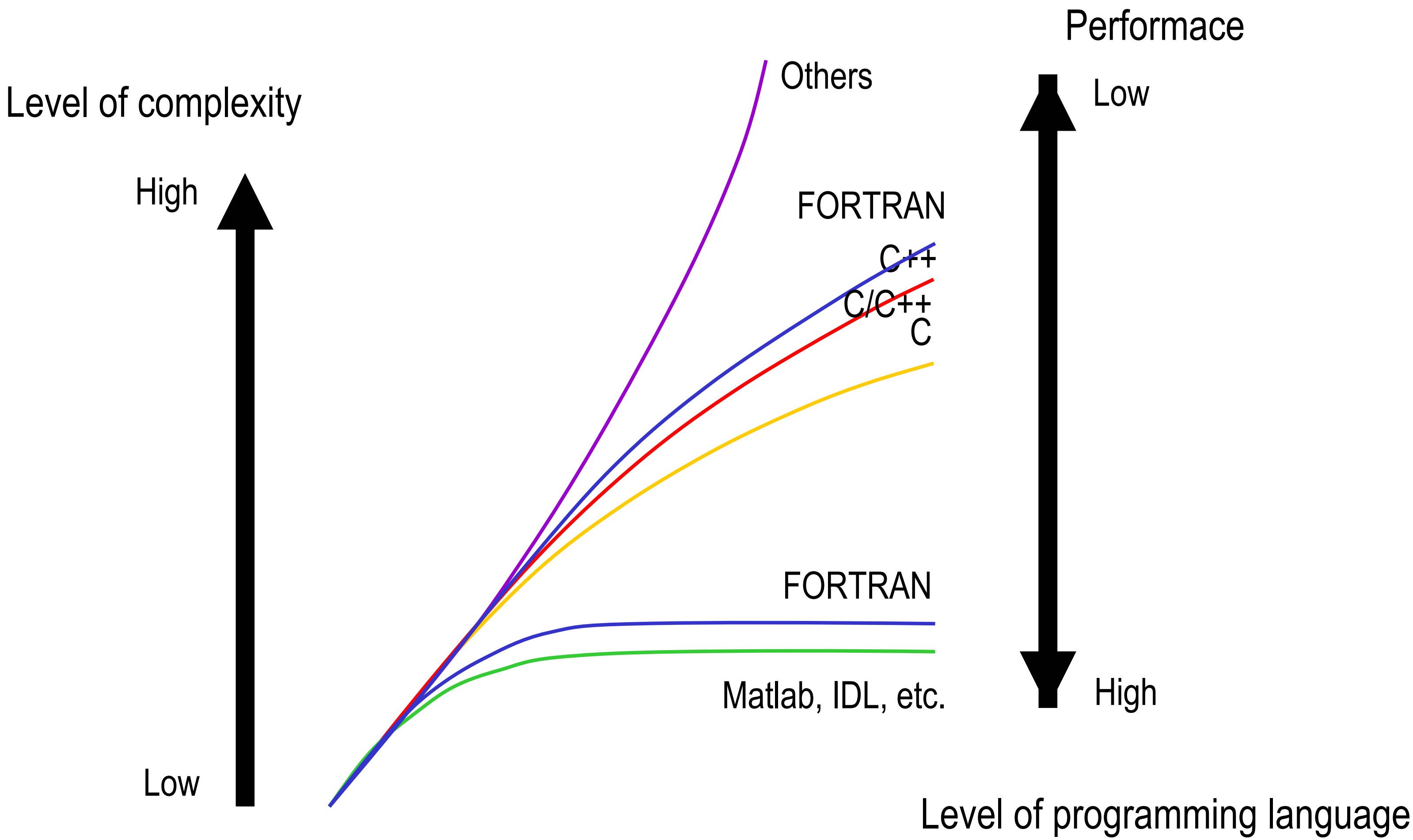
A Survey

What was the first programming language you learned at the university?

- A. C.
- B. C++.
- C. Java.
- D. Pascal.
- E. Others.

Fortran vs. MATLAB and Others

- The learning curves



Copyright © 2007 Ge Baolai

Example: *Array Construction, Overloading*

- **Matlab**

```
v = [-1, 0, 2 5, 14, 3.5, -0.02];

n = 1000;
x = 1:n;

y = cos(x)
```

- **Fortran**

```
v = (/-1, 0, 2, 5, 14, 3.5, -0.02/)

n = 1000
x = (/(i,i=1,n)/)

y = cos(x)
```

Copyright © 2007 Ge Baolai

Example: *Array Operations*

- **Matlab**

```
a = zeros(100,100);
b = zeros(100);

n = input("Enter n (n <= 100): ");


a(:,:) = 1.0;
b(:) = 2.0;
c = a(1,:) ./ b;
```

- **Fortran**

```
real, dimension(100,100) :: a
real, dimension(100) :: b, c

print *, 'Enter n (n <= 100):'
read *, n

a(:,:) = 1.0;
b(:) = 2.0;
```

Or simply

```
a = 1.0
b = 2.0
c = a(1,:) / b
```

Copyright © 2007 Ge Baolai

Loops

Integrate the initial value problem

$$y\prime = f(t, y) = \lambda y, \quad y(0) = 2$$

with step size h using

1) **Euler** method

$$\begin{aligned} y^{n+1} &= y^n + hf(t_n, y_n) \\ &= (1 + \lambda h) y^n \end{aligned}$$

2) **midpoint** scheme

$$\begin{aligned} y^{n+1} &= y^n + hf(t_{n+1/2}, \tfrac{y^n + y^{n+1}}{2}) \\ &= \frac{1 + \lambda h/2}{1 - \lambda h/2} y^n \end{aligned}$$

Note: The true solution is

$$y(t) = y(0) e^{\lambda t}$$

- **Matlab**

```
%program euler
  r = input('Enter lambda: ');
  y0 = input('Enter y(0): ');
  h = input('Enter step size: ');
  n = 1 + ceil(1/h);
  x = zeros(n,1);
  y = zeros(n,1);

  x(1) = 0.0;
  y(1) = y0;
  for i = 2:n
    x(i) = x(i-1) + h;
    y(i) = (1 + r*h) * y(i-1);
  end
  … …
%end program euler
```

Copyright © 2007 Ge Baolai

Loops (cont'd)

- **Matab**

```
%program midpoint
  r = input('Enter lambda: ');
  y0 = input('Enter y(0): ');
  h = input('Enter time step: ');
  n = ceil(1/h);
  x = zeros(n,1);
  y = zeros(n,1);
  g = (1.0 + 0.5*r*h)/(1.0 - 0.5*r*h);

  x(1) = 0.0;
  y(1) = y0;
  for i = 2:n
    x(i) = x(i-1) + h;
    y(i) = g * y(i-1);
  end
  … …
%end program midpoint
```

$$y^{n+1} = \boxed{\frac{1 + \lambda h/2}{1 - \lambda h/2}} y^n$$

- **Fortran**

```
program midpoint
  integer :: i, n
  real :: g, h, r, y0
  real, dimension(:), allocatable :: x, y
  … …
  n = ceiling(1.0/h)
  allocate(x(n),y(n))
  g = (1.0 + 0.5*r*h)/(1.0 - 0.5*r*h)

  x(1) = 0.0
  y(1) = y0
  do i = 2, n
    x(i) = x(i-1) + h
    y(i) = g * y(i-1)
  end do
  … …
end program midpoint
```

Copyright © 2007 Ge Baolai

Example: Linear Algebra Operations

- **Matlab**

Underlying operations may use **BLAS**, **LAPACK** or others

- **Fortran**

Uses highly optimized **BLAS**, **LAPACK** routines:

$$\longleftarrow \quad C = AB \quad \longrightarrow$$

Matlab: C = A * B;

Fortran: C = **MATMUL**(A, B)
or
call **GEMM**(A, B, C) ! More efficient

$$\longleftarrow \quad PA = LU \quad \longrightarrow$$

Matlab: [L, U, P] = **lu**(A)

Fortran: call **getrf**(A [,ipiv, info]) ! Done in place

Matlab: [V, E] = **eig**(A)

Fortran: Freedom for users. Need to distinguish between symmetric / Hermitian (Use **syevd/heevd**(A, W [,…])) and general cases (Check LAPACK reference).

Copyright © 2007 Ge Baolai

MATLAB and Free Fortran Compilers

- **Octave** – A MATLAB clone, free under GPL.
- **gfortran** – GNU Fortran 95 compiler.
- Intel Fortran compiler **ifort** for Linux (**“*non commercial*”** use only – really meaning for activities that one does not receive any form of compensations, such hobby).
- Sun Forte suite for SunOS.
- **FTN95** for Windows.
- Others.

References

[1] Michael Metcalf and John Reid, "***FORTRAN 90/95 Explained***", 2nd edition, Oxford University Press, New York, 2002.

[2] Sun Microsystems, Inc., "***Fortran Programming Guide***", 2005.

[3] **JTC1/SC22** – The international standardization subcommittee for programming languages (http://www.open-std.org/jtc1/sc22/).

Copyright © 2007 Ge Baolai

Language Elements of Fortran

- Format
- Data types
- Variables
- Array processing
- Control constructs
- Subroutines, functions
- I/O

Source Format

- **Fixed Format (FORTRAN 77 style)**

```
1234567 Source starts column 7
      program array
      integer i, j, m, n
      real*8 a(100,100)
c
      print *, 'Hello world!'
c
c Comments start in column 1
      do 10 j = 1, n
         do 11 i = 1, m
            a(i,j) = i * j
   11    continue
   10 continue
c
      print *, a
      stop
      end
```

- **Free Format**

```
! Comments start with an !
program array
   integer :: i, j, m, n
   real(2) :: a(100,100)

   print *, 'Hello world!'

   ! Comment can start anywhere
   do j = 1, n
      do i = 1, m
         a(i,j) = i * j
      end do
   end do

   print *, a
end program array
```

Copyright © 2007 Ge Baolai

Data Types

- **Integer** – 4 bytes.
- **Real** – single or double precision.
- **Complex** – single (8 bytes) and double precision (16 bytes).
- **Character** – character and strings.
- **Logical** – e.g. **.FALSE.** and **.TRUE.**
- **Constant** – literal constants, e.g. 123, 3.1415926…
- **Derived types** – data structures, along with numeric ***polymorphism***, lead to ***object-oriented*** programming
- **Pointer** – useful for array referencing and swapping. ***FURTHER READING!***

e.g.

```
integer(2) :: n2        ! Short
integer(4) :: n         ! Integer
real(4) :: mass         ! Single precision
real(8), dimension(10000) :: x, y
logical :: isneighour = .false.

type particle
   real m
   real x, y, z
   real u, v, w
end type particle
```


Copyright © 2007 Ge Baolai

Variables

- Case **INSENSITIVE**.
- Must start with a letter, allows underscores.
- Must declare types.
- Or declare "implicit" rule in variable names.

Example

```
integer :: n

! Single precision variables
real :: mass(10000)
real, dimension(10000) :: u_x, u_y, u_z

! Double precision
real(8), dimension(10000) :: potential

! Size determined at run time
real, dimension(:,:), allocatable :: a, b

read *, n  ! Read from standard input
allocate(a(n,n), b(n,n), x(0:n))
```

Copyright © 2007 Ge Baolai

Expressions And Assignments

- **Scalar Operations**

```
pi = 3.1415926
c = 2.0*pi*r
a = b * c + d / e
r = sqrt(x**2 + y**2)
```

```
integer :: m = 17, n = 5
real :: q, a
q = m / n
a = q * n              ! What's in a?
```

- **Character, Strings**

```
character(len=5) :: word1
character(len=5) :: word2
character(256) :: grtg_msg
```

String concatenation

```
word1 = 'Hello'
word2 = 'world'
grtg_msg = word1//','//word2//'!'
```

Trimming off trailing blanks

```
trim(grtg_msg)
```

Copyright © 2007 Ge Baolai

Expressions and Assignment (cont'd)

- **Array Operations**

```
real, dimension(1000) :: a, b

! Element wide operations
do I = 1, 1000
   a(i) = a(i) + 1.0
   b(i) = a(i)
end do

! Alternative, simple ways
a = a + 1.0
b = a

! Block assignment, evaluations
b(11:20) = a(1:10)
G(:,k) = G(:,j)/G(k,k)
H(k,:) = M(i,:)
```

Operations like

$$C \leftarrow A \oplus B$$

assumes element-by-element operations.

Data parallelism – parallel array operations is abstracted.

Copyright © 2007 Ge Baolai

Expressions and Assignment (cont'd)

- **Objects of Derived Type**

```
type particle
  real m
  real :: x, y
  real :: u, v
end type particle

type(particle) :: p, q

p = particle(0.2, -1.0, 3.5, 0.5, 2.7)
q = p

q%x = q%x – x0
q%y = q%y – y0
```

FURTHER READING: *Assignment of objects of derived type containing derived objects (**operator overloading**).*

Copyright © 2007 Ge Baolai

Control Constructs

- *DO* loops

```
do i = 1, n
  y(i) = f(x(i))
end do

do k = 1, 10000, 2
  do something
end do

do j = 1, n
  do i = 1, m
    a(i,j) = mat_setval(i,j)
  end do
end do
```

- *IF..THEN..ELSE IF..ELSE..ENDIF*

```
if (x == y) then
  do something
end if

if (x == y1) then
  do case1
else if (x == y2) then
  do case2
else if (x == y3) then
  do case3
else
  do default case
end if
```

Copyright © 2007 Ge Baolai

Control Constructs (cont'd)

- ***SELECT..CASE***

select case *expr*
case *val1*
  process case1
case *val2*
  process case2
…
case default
  process default case
end select

- **Example**

```
! Select on individual values
select case j
case 1
  call sub1
case 2
  call sub2
end select

! Select on a range
select case x
case (:-1)             ! All <= -1
  call sub1
case (1:5)             ! 1,2,3,4,5
  call sub2
…
end select
```

Copyright © 2007 Ge Baolai

Input/Ouput

- **Standard in, out**

```
print *, 'Hello, world'

print *, 'Please enter num_xpoints'
read *, num_xpoints
```

- **File I/O**

```
open(1,file='data.in',status='old')
open(2,file='data.out',status='unknown')
… …
read(1,*) a, b, c        ! Read from file data.in
… …
write(2,*) x, y, z       ! Write to file data.out
… …
close(11)
close(12)
```

FURTHER READING: Formatted I/O.

Copyright © 2007 Ge Baolai

Main And Subprograms

- **Main program**

Every program has a main program, e.g.

```
program xyz
  use mod1
  use mod2
  integer :: i, j, k, m, n
  real, dimension(:,:), allocatable :: a, b
  … …
  call input(…)
  call do_some_work(a,b,…)
  call output(…)
  … …
end program xyz
```

- **Subprograms**

May define subprograms – functions and subroutines – out side main program.

```
subroutine do_some_work(a, b, …)
  use mod2
  use mod6
  real, dimension(:,:) :: a, b
  … …
  call sub1(…)
  … …
  call sub2(…)
  … …
end subroutine do_some_work
```


Copyright © 2007 Ge Baolai

Subroutines and Functions

- **Function – returns one variable**

For example, to calculate

$$p(v) = \sqrt{\frac{2}{\pi}\frac{v^2}{a^3}}\exp(-v^2/2a^2)$$

we write a function

```
function pdf(x,a) ! Return a value
  real :: a, x, pdf

  pdf = sqrt(2.0/3.1415926)
  pdf = pdf *x*x*exp(-0.5*x*x/(a*a))/(a**3)
end function pdf
```

and use it

```
p = pdf(v,a)
```

- **Subroutine – may return more than one variable**

```
! Return two values u and v
subroutine velocity(t, x, y, u, v)
  use global_vars
  real :: t, x, y, u, v
  … …

  r = sqrt(x*x + y*y)
  u = speed * (-y / r)
  v = speed * (x / r)
end subroutine velocity
```

Can also define a function that returns an array.

Copyright © 2007 Ge Baolai

Overloading or *Numeric Polymorphism*

- **Function Overloading**

Intrinsic functions already overloaded, e.g.
one function name for all types,

```
r = sqrt(x**x + y**y)
```

instead of **SQRT** for singles and **DSQRT**
for doubles as in old days.

Copyright © 2007 Ge Baolai

Subroutines: Passing Arrays – A Famous Issue

- **A Common Mistake in Fortran**

In main

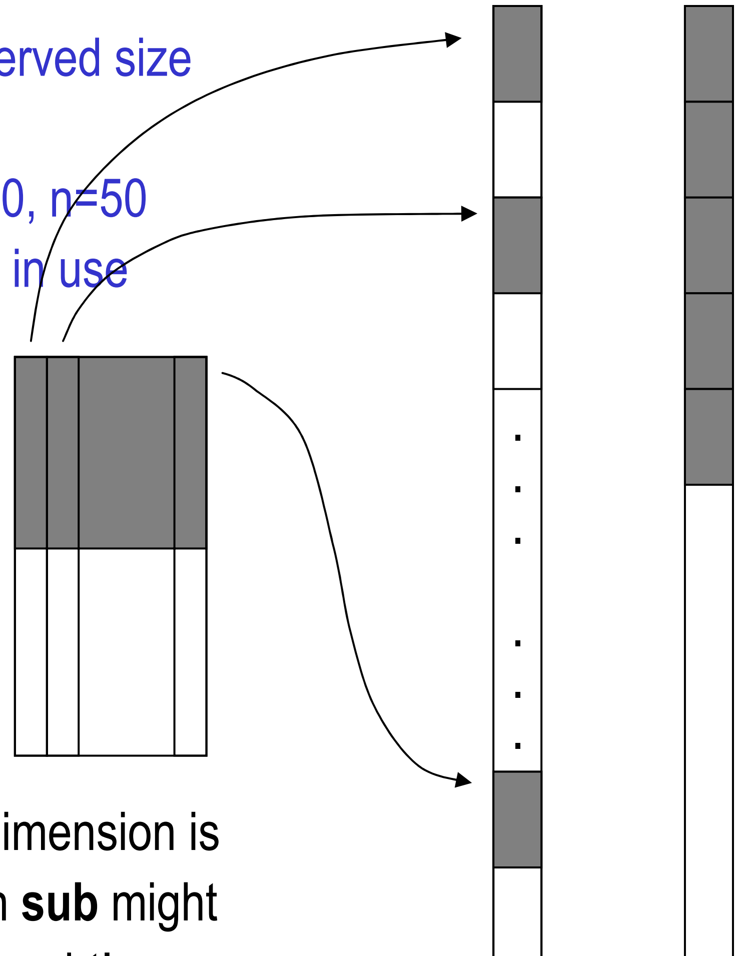
```
real a(100,50)        ! Reserved size
… …
read *, m, n          ! m=50, n=50
call sub(a, m, n)     ! Size in use
```

In subroutine sub

```
subroutine sub(a, m, n)
implicit none
integer m, n
real a(m,n)
… …
```

The “shape” – the actual first dimension is missing. Referencing to a(i,j) in **sub** might not be what you might have thought!

- **Array Elements Stored in Column**



Reference to elements in use in subroutine

In actual storage

Copyright © 2007 Ge Baolai

Subroutines: Passing Arrays – A Famous Issue

- **A Common Mistake in Fortran**

In main

```
real a(100,50)        ! Actual size
… …
read *, m, n
call sub(a, m, n)     ! Size in use
```

In subroutine sub

```
subroutine sub(a, m, n)
implicit none
integer m, n
real a(m,n)
… …
```

The "shape" – the actual first dimension is missing. The array elements in sub will be out of order!

- **Assumed-Shape Array in Fortan 90**

One way fix this is to pass the extent of one dimension – leading dimension, e.g. ld = 100

```
call sub(a, ld, m, n)
```

and define

```
subroutine sub(a, ld, m, n)
implicit none
integer m, n
real a(ld,*)
```

In Fortran 90, this is done automatically with ***assumed-shape*** in sub

```
subroutine sub(a, m, n)
real, dimension(:,:) :: a
```

Copyright © 2007 Ge Baolai

Function/Subroutine: *Variable Argument List*

- **Optional Arguments (cont'd)**

```
program optional_test
  real :: a, b, c
  interface
    subroutine sub(a, b, c)
      real :: a, b, c
      optional :: c
    end subroutine sub
  end interface

  print *, 'Enter a, b'
  read *, a, b

  call sub(a,b)
  call sub(a,b,c)
end program optional_test
```

- **Optional Arguments**

```
subroutine sub(a, b, c)
  real :: a, b, c
  optional :: c

  if (present(c)) then
    c = a + b
    print *, 'A=', a, ', B=', b, ', C=', c
  else
    print *, 'A=', a, ', B=', b
  end if
end subroutine sub
```

Copyright © 2007 Ge Baolai

Function/Subroutine: *Polymorphism*

- Define A Generic Interface

```
interface fun                ! geric name
  function fun_int(x)        ! For integer
    integer :: x
  end function fun_int
  function fun_real(x)       ! For single
    real :: x
  end function fun_real
  function fun_array(x)      ! For array
    real, dimension(:) :: x
    real :: fun_array(size(x))
  end function fun_array
end interface
```

- Different Mapping

```
integer :: j, k
real :: x, y
real, dimension(:), allocatable :: a, b
… …
k = fun(j)          ! Integer
y = fun(x)          ! Real
b = fun(a)          ! Array of reals
```

Copyright © 2007 Ge Baolai

Modules

- **Variable scope**

Variables are local to the program unit.

```
program xyz
  use mod1
  use mod2
  integer :: i, j, k, m, n
  real, dimension(:,:) :: a, b
  … …
  call input(…)
  call do_some_work(a,b,…)
  call output(…)
  … …
end program xyz
```

- **Module**

1. Commonly used to include global variables.
2. May contain function, subroutine interface (like C/C++ function prototypes).

e.g.

```
module mod1
  integer :: num_xpoints, num_ypoints
  real, dimension(:), allocateable :: x, y
  … …

end module mod1
```

Copyright © 2007 Ge Baolai

File Organization

Scope of Variables

- Variables are local to subprograms.
- To make a variable visible outside the subprogram, make it global.
- Place global variables in common blocks or modules. Include common blocks or modules in subprograms.

- **Example**

```
module globals
  real, dimension(:), allocatable :: x, y, z
end module globals

program main
  use globals
  … …
  x = x0
  call sub(a1,a2)
  … …
end program main

subroutine sub(a,b)
  use globals
  … …
  a = fun(x,y,z)
end subroutine sub
```

Copyright © 2007 Ge Baolai

Life Span of Storage

- **Local** variables are freed after calls.
- **Global**, **static** variables live until the termination of the program.
- **Dynamically allocated** variables live until deallocation calls.

- **Example: SAVE**

```
subroutine velocity(t)
  use globals
  integer, save :: count = 0
  … …
  count = count + 1
  … …
end subroutine velocity

program main
  use globals
  … …
  call velocity(t)
  … …
end program velocity
```

Copyright © 2007 Ge Baolai

File Organization

- **Single File**

One single file **prog.f90** contains

Everything. Easy to manage, e.g.

f90 prog.f90

but takes longer to (re)build.

- **Multiple files**

Source code in multiple files

main.f90

f1.f90

f2.f90

… …

f17.f90

To compile

f90 main.f90 f1.f90 f2.f90 …

Easy to make changes in one spot,

efficient to (re)build.

Copyright © 2007 Ge Baolai

Compiling and Running FORTRAN Programs

Compiling and Running Programs

- **In A Single File**

On UNIX using GNU Fortran compiler

f90 hello.f90 -o hello

This will generate an executeable a.out.
To run the program, type command

./hello

On Windows under IDE, this is usually handled by **build** and **run** options.

- **In Multiple Files**

On UNIX using GNU Fortran compiler

f90 global.f90 main.f90 sub1.f90 -o prog

This will generate an executeable a.out.
To run the program, type command

./prog

On Windows under IDE, files are organized by **projects** and compilation and execution of a program is usually handled by **build** and **run** options.

Copyright © 2007 Ge Baolai

Using Makefile

- A Simple One

```
FC = f90
SOURCE = main.f f1.f f2.f f3.f

all: program

program: $(SOURCE)
	$(FC) -o myprog $(OBJECTS) $(LDFLAGS)
```

- A Little Complex One

```
FC = f90
OBJECTS = main.o f1.o f2.o f3.o
MODULES = mod1.mod mod2.mod

all: program

%.mod : %.o
	if [ ! -f $@ ]; then rm $<; make FC=$(FC) $<; fi

%.o : %.f90
	$(FC) $(FFLAGS) -c -o $@ $<

program: $(MODULES) $(OBJECTS)
	$(FC) -o myprog $(OBJECTS) $(LDFLAGS)
```

Copyright © 2007 Ge Baolai

From MATLAB to FORTRAN 90/95

Contrasting MATLAB and Fortran

Some Performance Comparisons

- Array constructors
- Matrix multiplication
- Matrix transpose
- Loops
- LU factorization

Array Construction

- **Matlab**

```
A = zeros(n);
B = A;              % Space not created
B(1,1) = 1.0;       % Spaced created

tic
C = B – A*B;
toc
```

- **Fortran**

```
allocate(A(n),B(n))
A = 0.0
B = A               ! Data are copied to B
B(1,1) = 1.0        ! Only changes B(1,1)

call system_clock(c1,crate)
allocate(C(n,n))
call gemm(A, B, C)
C = B - C
call system_clock(c2,crate)
```

Copyright © 2007 Ge Baolai

Matrix-Matrix Multiplication

- **Matlab**

```
A = rand(m,n);
B = rand(n,s);
C = A*B;
```

Can also write as simple as

call **gemm**(A,B,C) ⟶

- **Fortran**

allocate(A(m,n),B(n,s),C(m,s))

Set values to A and B

C = **matmul**(A,B)

Alternatively, one may use BLAS Level 3 routine ***X*GEMM**, that performs

$$C = \alpha AB + \beta C$$

e.g.

call **dgemm**('N','N',m,n,s,α,A,lda,B,ldb, β,C,ldc)

Matrix Transpose

- **Matlab**

B = A'

- **Fortran**

B = **transpose**(A)

Copyright © 2007 Ge Baolai

Loops

- **Matlab**

```
for i = 1:n
  for j = 1:n
    h(i,j) = 1.0 / (i + j)
  end
end
```

A MATLAB built-in function **hilb**() does the same thing. But seems to be slower than loops?

- **Fortran**

```
do j = 1, n
  do i = 1, n
    h(i,j) = 1.0 / (i + j)
  end do
end do
```

Parallelize with threads using OpenMP

```
!omp parallel do
do j = 1, n
  !omp parallel do
  do i = 1, n
    h(i,j) = 1.0 / (i + j)
  end do
  !omp end parallel do
end do
!omp end parallel do
```

Copyright © 2007 Ge Baolai

LU Decomposition: $PA = LU$

- **Matlab**

```
A = rand(n);
[L, U, P] = lu(A);
```

- **Fortran**

```
allocate(a(n,n), ipiv(n))
do j = 1, n
  do i = 1, n
    call random_number(rv)
    a(i, j) = rv
  end do
end do
```

Call LAPACK in an old fashion:

```
call dgetrf(n, n, A, n, ipiv, info)
```

Or can be as simple as

Dimensionality is encapsulated

```
call lu(A, ipiv, info)
```

Copyright © 2007 Ge Baolai

Find How to? – Consult Math Libraries

- Linear algebra calculations such as **matrix-vector**, **matrix-matrix** operations, solving **system of linear equations**, finding **eigenvalues**, etc. can all be done by using the highly optimized subroutines provided in **BLAS** and **LAPACK**.
- Various vendor libraries that have BLAS and LAPACK and others: Intel **MKL**, Compaq **CXML**, HP **MLIB**, AMD **ACML**, VNI **IMSL**, etc.
- **FFTW** – Award winning FFT package (for UNIX).
- **PETSc** – Partial differential equation solvers.
- **ODE** solvers, such as Sundial.
- Do not write your own or copy from text books!

Copyright © 2007 Ge Baolai

Concluding Remarks

Features	MATLAB	Fortran
Data types	Typeless. Objects, double only.	Typed. Five intrinsic types, capable of defining new object types.
Data abstraction	As simple as mathematical representations.	As simple as mathematical representations. Array operations have nearly the same forms as MATLAB.
Loops	Slow.	Fast
Functions/Procedures		
Numeric polymorphism	Yes	Yes
Variable argument list	Yes	Yes
Problem solving	Self-contained	Requires external libraries.
Graphics	Yes	No
Parallel processing		
Multithreading	Multithreading in linear algebra and element-wise operations is now supported in the latest release.	Data parallelism has long been handled by compiler, via **OpenMP**, or can use **POSIX threads** explicitly.
Distributed processing	Distributed objects, parallel processing has appeared, but in its initial stage	Explicit data and task parallelism can be achieved using OpenMP and MPI.
… …	… …	… …

References

About Fortran and MATLAB

[1] Michael Metcalf and John Reid, "***FORTRAN 90/95 Explained***", 2nd edition, Oxford University Press, New York, 2002.

[2] **JTC1/SC22** – The international standardization subcommittee for programming languages (http://www.open-std.org/jtc1/sc22/).

[3] ***MATLAB 7 Programming***, MathWorks, 2007.

Some interesting readings

[1] NA Digest V. 7 #3, #4, "***MATLAB vs. Fortran in introductory numerical analysis courses***", (http://www.cs.utexas.edu/users/flame/FORTRANvsMATLAB/).

[2] Marcos Rico and Manuel Baselga, "***30 Years of Research in Animal Breading: APL versus Matlab and Fortran***", APL 2002 Proceedings.

[3] Cleve Moler's Corner at MathWorks, "***The Origins of MATLAB***", MATLAB News & Notes, December 2004. Also available in video.

Copyright © 2007 Ge Baolai

Weekly Online Seminars
Every Monday
Via
AccessGrid

SHARCNET
Scientific and Technical Computing
www.sharcnet.ca

Categories

- New User Training
- Research Topics
- Special Topics on Scientific and High Performance Computing

SHARCNET Literacy Series

Scientific and High Performance Computing