A quick guide to Fortran

Sergiy Bubin

Department of Physics
Nazarbayev University

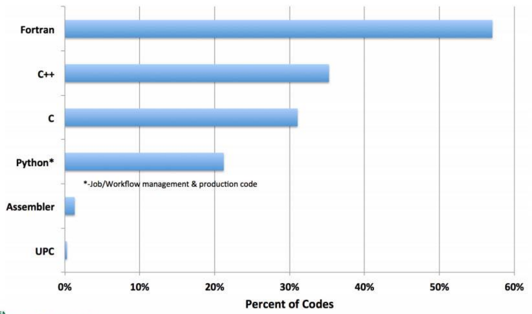
History of Fortran

- One of the oldest general purpose high-level computer languages
- First developed in 1957 at IBM in the era of punch cards
- Fortran name originated from an acronym - FORmula TRANslation
- Fortran evolution:
  - Fortran 57
  - Fortran II
  - Fortran IV
  - Fortran 66
  - Fortran 77
  - Fortran 90
  - Fortran 2000
  - Fortran 2003
  - Fortran 2008
  - Fortran 2015

Why Fortran?

Languages Used at NERSC

- Here data are collected from all NERSC projects
- If by machine hours used, Fortran is even more popular: 23 out of 36 top codes primarily use Fortran


Languages Used at NERSC 2015
(Taken from allocation request form. Sums to >100% because codes use multiple languages)

Why Fortran?

One of my favorite jokes of all times. It is often attributed to different people (Seymour Cray, John Backus, Edsger Dijkstra) so I do not know who was the the original author. It appeared in the end of 1970s, upon standardization of Fortran 77.

Q: What will the scientific programming language of the year 2000 look like?

A: Nobody knows, but its name will be Fortran.

Why Fortran?

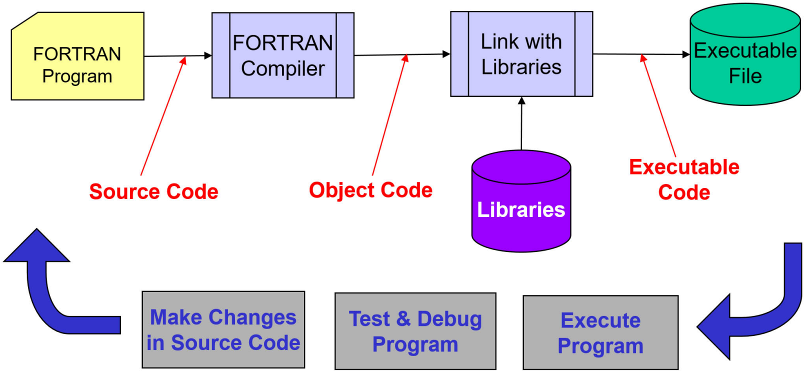
- Designed for scientific and engineering applications that involve heavy number crunching
- Large portion of existing scientific/engineering software is coded in Fortran
- Speed of computations - as a rule Fortan compilers generate the fastest code
- Huge legacy: mathematical functions/subroutines/algorithms developed by the scientific community over half century
- Actively developed and supported by major hardware vendors: Intel, HP, IBM, Cray, Fujitsu, Sun, AMD, etc.
- Free and open source versions are available on essentially any platform

Essential features

- Designed for scientific and engineering applications that involve heavy number crunching
- Array-oriented language - convenient support of arrays and array operations
- Compilers can generate highly optimized code
- Lots of available numerical/math libraries (both free and commercial ones), e.g. Intel MKL, IMSL, NAG
- Starting with Fortran 90 many contemporary programming constructs have been included
- Fortran is a compiled language
- Interoperable with other languages (in particular C)
- Code portability

Building a fortran program

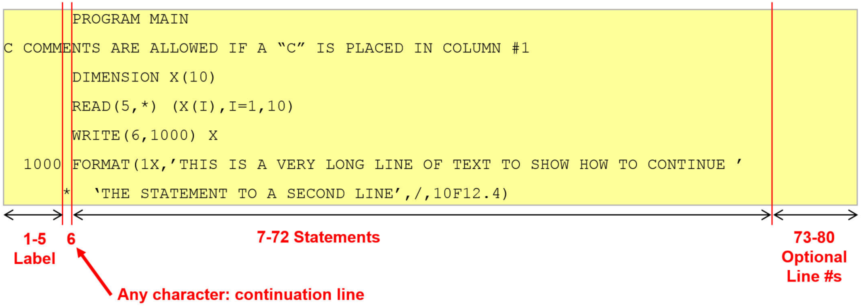
A typical framework for developing a Fortran program looks as follows



Picture: AE6382 Georgia Tech

“Old” vs “New” Fortran

Fortran standards before Fortran 90 require a fixed format of the code



```
      PROGRAM MAIN
C COMMENTS ARE ALLOWED IF A “C” IS PLACED IN COLUMN #1
      DIMENSION X(10)
      READ(5,*) (X(I),I=1,10)
      WRITE(6,1000) X
 1000 FORMAT(1X,’THIS IS A VERY LONG LINE OF TEXT TO SHOW HOW TO CONTINUE ’
     *   ‘THE STATEMENT TO A SECOND LINE’,/,10F12.4)
```

“Old” vs “New” Fortran

Fortran 90 relaxes this constraint

```
program main
!Comments are marked with an exclamation mark
                 !Placement of statements is essentially arbitrary
dimension x(10)
read(5,*) (X(i),i=1,10)
write (*,'(1x,a,10f12.4)') 'THIS IS A VERY LONG LINE OF TEXT TO SHOW HOW TO CONTINUE' &
'THE STATEMENT TO A SECOND LINE',x
```

Note: Fortran does not care about the case (upper/lower), so a variable a is the same as A. A statement write(*,*) a is the same as WRITE(*,*)

“Old” vs “New” Fortran

- Compilers retain backward compatibility, e.g. a Fortran program written according to an older standard should (and generally is) be compilable with newer compilers
- The general convention (not a requirement) is that fixed format source files have an extention .f, while the free format source files have an extention .f90

myprog.f myprog.f90

General structure of a Fortran program

- A Fortran programs consist of a main program (can have an arbitrary name) and may contain one or more subprograms (subroutines, functions)
- Declaration of variables/data, subroutines, and functions may be put in a separate program unit – a module
- The entire code may be split into multiple source files (convenient for large projects)

A basic structure of a main program:

```
program myprog
< declaration of variables and data >
< program body >
end program myprog

< definition of subroutines/functions >
```

gfortran

- gfortran is a modern, free, open source version of Fortran developed by the software community undeg GNU general public license
- Available on almost any platform
- High quality and fast (although may generate a code that runs is a little bit slower than that by some harware vendors and/or commercial compilers)

Fortran data types

Basic data types

- integer – integer numbers
- real – floating point numbers
- double precision – floating point numbers
- character(n) – strings with up to n characters
- logical – logical variable that takes on values .true. or .false.
- complex – complex numbers

Fortran data types

Integers and reals can specify the kind (essentially the number of bytes that they use)

- real(4) - single precision real numbers (about 7-8 decimal figures)
- real(8),double precision - double precision real numbers (about 15-16 decimal figures)
- complex(4) - single precision complex numbers
- complex(8) - double precision complex numbers
- integer(4) - integers ranging from -2147483648 to $+2147483647$
- integer(8) - integers ranging from -9223372036854775808 to $+922337203685477580$

The default is integer(4) and real(4)

Fortran data types

One needs to be extra careful with arithmetic operations in Fortran

- a=2/3 will give 0 even if a is declared as real
- a=2.0/3 will convert 8 to single precision then will divide 2.0 by 3.0 with single precision, then it will assign the (single precision) result to a. If a is declared as a double precision, you will essentially have only half (7-8) accurate digits in it. For double precision the proper syntax would be a=2.0D0/3 or a=2.0_8/3

Fortran data types

One needs to be extra careful with arithmetic operations in Fortran

- a=2/3 will give 0 even if a is declared as real
- a=2.0/3 will convert 8 to single precision then will divide 2.0 by 3.0 with single precision, then it will assign the (single precision) result to a. If a is declared as a double precision, you will essentially have only half (7-8) accurate digits in it. For double precision the proper syntax would be a=2.0D0/3 or a=2.0_8/3

Arrays

Arrays of any type must be declared, e.g.

- `real(8), dimension(20,30) :: a,b` – declares two static arrays (a and b) of size 20x30 whose elements are double precision numbers
- `integer(4)    a(10), b(20), c(30,30), d(10,10,10)` – declares three static arrays. Array a is a vector with 10 elements. Array b is a vector with 20 elements. Array c is a 30x30 matrix. Array d is a 10x10x10 3D array. All these arrays are of type integer(4).

Referencing an element of an array:

- `x=a(15,15)` – assigns variable x the value stored in element 15,15 of matrix a

Allocatable Arrays

When the size of data is not known it advance (or if there are other reasons), it is possible to allocate/deallocate arrays as necessary

```
program myprog
real(8),dimension(:,:) :: a
real(8),dimension(:) :: b
allocate(a(10,10),b(20))
< ... do something else ... >
deallocate(a,b)
allocate (b(100))
< ... do something else ... >
deallocate(b)
end program
```

Default types

- By default, an implicit type is assumed depending on the first letter of the variable name: A-H, O-Z define real variables I-N define integer variables
- We can use the implicit statement: implicit real (A-Z) makes all variables real if not declared
- implicit character(2) (W) makes variables starting with W be 2-character strings
- A good habit is to force explicit type declarations by putting implicit none right in the beginning of your program or module (i.e. after program myprogramname statement)

Parameters

Constants can be declared as follows

```
real(8),parameter :: pi=3.141592653589793D+00
real(8),parameter :: halfpi=3.141592653589793D+00/2
```

Execution control - if statement

```
if (a>2) b=3

if ((a==1).or.(a==2)) then
  b=3
else
  b=4
endif

if (a>10) then
  b=1
elseif (a<5) then
  b=2
else
  b=3
endif
```

Loops

```
do i=1,10
  < ... do something ... >
enddo

do i=100,1,-5
  < ... do something ... >
enddo

do while ((x>0).and.(y==5))
  < ... do something ... >
enddo
```

Functions and subroutines

```
program main
real(8) x,y,z
  x=10.0d0;  y=20.0d0
  z=myfunc(x)
  call mysub(y,z)
end program main

function myfunc(a)
real(8) a,myfunc
  myfunc=a*a
end function myfunc

subroutine mysub(a,b)
real(8) a,b
  write(*,*) a,b
end subroutine mysub
```

Input and output statements

- write(*,*) a,b – writes into the default device (screen) the values of two variables a and b using default format
- write(1,*) a – writes the value of variable a into device 1 (usually a file that needs to be open prior to that)
- write(1,'(a,i3,a,d13.6)') 'measurement number ',j,' yielded ',y – writes four variables (a string, an integer number, a string, and a double precision number into device 1 according to the specified format (3 total digits for the integer j, 13 total symbols + 6 decimal figures for the float number y))

Pretty much the same rules apply to read statement

Opening and closing files

- open statement is used to make file available to read and write
- Several files can be opened at the same time if necessary (each referenced by a device number)
- Files can be of either binary form (fast read, compact storage, but unreadable by humans) or ASCII format (readable text)
- Positioning and reading in a file is sequential
- close statement is used to close access to a file

Example of writing a matrix into an ASCII file

```
program myprog
integer,parameter :: n=5
real(8) A(n,n)
integer i,j
call random_number(A)
open(1,file='myfile.dat')
do i=1,n
  do j=1,n
    write(1,'(1x,d23.16)',advance='no') A(i,j)
  enddo
  write(1,*) enddo
close(1)
end program myprog
```

Compiling a simple Fortran program with gfortran

- Write a program (e.g. file myprog.f90) in any editor of your choice
- Open a terminal and go to the directory where file myprog.f90 is located
- Type `gfortran myprog.f90 -o myprog`
- After that (if no error occurs) a binary file called myprog will be generated and placed in the same directory
- You can now run this binary file in the terminal by typing `./myprog`

LAPACK

LAPACK (Linear Algebra PACKage) is a widely used numerical software package written in Fortran. It contains a large number of subroutines to solve various problems that involve matrices - systems of linear equations, eigenvalue problems, singular value decompositions, etc. It makes use of BLAS (Basic Linear Algebra Subprograms) - highly tuned sets of subprograms that are available for most hardware platforms. It emerged in 1990s from the famous LINPACK and EISPACK packages and is being maintained/developed by numerical mathematicians in several National Labs and Universities.

- Used as a component in most scientific software (due to the ubiquity of linear algebra problems in numerical computations)
- Exploits the architechture of moderns computers (e.g. cache)
- Very efficient for general purposes
- Some vendors adapt it for multi-core CPUs

LAPACK

- Guide and (non-optimized, reference) source code is available at
  http://www.netlib.org/lapack
- Comments explaining all arguments are provided in the source of each subroutine

Linking with precompiled LAPACK library when using gfortran

- Write a program (e.g. file myprog.f90) that calls a LAPACK subroutine
- Open a terminal and go to the directory where file myprog.f90 is located
- Type `gfortran -llapack myprog.f90 -o myprog`
- After that (if no error occurs) a binary file called myprog will be generated and placed in the same directory
- You can now run this binary file in the terminal by typing `./myprog`

References

A huge number of Fortran manuals/guides/reference books/youtube videos/presentation slides/tutorials exist. Feel free to google. Examples of the books are:

- R. Davis, A. Rea, D. Tsaptsinos, *Introduction to Fortran 90*
- J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, J. L. Wagener, *Fortran 90 Handbook*