WVU Summer Hands-on Introduction to HPC

Guillermo Avendano-Franco

June 8, 2017

Contents

Schedule

Day 1 - June 12, 2017	
09:00 - 10:00	Login into remote systems (SSH, PuTTY and tmux)
10:00 - 11:00	Command Line Interface
11:00 - 12:00	Text Editors (nano, emacs and vi)
12:00 - 13:00	Lunch Break
13:00 - 13:30	Using Environmental Modules (module)
13:30 - 15:30	Torque and Moab (qsub, qstat, qdel, checkjob)
15:30 - 16:00	Transfering files (rsync and globus)

Day 2 - June 13, 2017	
09:00 - 10:00	Shell Scripting (including grep)
10:00 - 11:00	Python Scripting
11:00 - 12:00	Using pip and virtualenv
12:00 - 13:00	Lunch Break
13:00 - 14:00	Plotting (gnuplot and matplotlib)
14:00 - 15:00	Building Software (example with fftw)
15:00 - 16:00	Creating Environmental Modules

Day 3 - June 14, 2017	
09:00 - 10:00	Advanced Scripting (regular expressions)
10:00 - 11:00	Programming in C, Fortran and Python I
11:00 - 12:00	Programming in C, Fortran and Python II
12:00 - 13:00	Lunch Break
13:00 - 14:00	Parallel Programming (OpenMP)
14:00 - 15:00	Parallel Programming (MPI)
15:00 - 15:30	Test Driven Development (Python nose)
15:30 - 16:00	Version Control with Git

1 Linux, Command Line and HPC Environment (Newcomer)

1.1 Logging in to Mountaineer or Spruce

Currently WVU has two clusters for HPC, mountaineer and spruce. You can access them using SSH. SSH provides a secure channel over an unsecured network such as internet. Both Linux and macOS commonly include the SSH client by default. On Windows machines you can use a free application called PuTTY.

To connect to Mountaineer use:

```
ssh <username>@mountaineer.hpc.wvu.edu
```

For Spruce

```
ssh <username>@spruce.hpc.wvu.edu
```

Once you enter on the system, you can start typing commands. You can open several connections simultaneously. Each connection is independent of each other.

Power users can benefit from a terminal multiplexer such as tmux. tmux allows users to keep several virtual windows and panels open from a single connection. It offers also preserve the terminal status in case of disconnection from the server.

To use tmux, first connect to the server and execute the command

```
tmux
```

You can create new virtual windows with `CTRL-b c`, you move between windows with `CTRL-b n` and `CTRL-b p`. You can detach from your multiplexed terminals with `CTRL-b d`.

If for some reason you lost the connection to the server or you detached from the mulpiplexer all that you have to do to reconnect is to execute the command:

```
tmux a
```

There are many options for using tmux, see the cheat cheat for some of them.

1.2 Basic Commands to learn (Top 10) and some more

When you interact with a HPC cluster your interaction is basically by executing commands on a terminal and editing text files. For newcomers using command lines could be a frustrating experience knowing that there are literally hundreds of commands. Certainly there are manuals for most of those commands, but they are of no use if you do not know which is the command you need to use for each situation. The good news is that you can do a lot of things with just a bunch of them and you can learn others in due time.

This is a selection of the 10 most essential commands you need to learn.

1.2.1 ls

List all the files in a directory. Linux as many Operating Systems organize files in files and directories (also called folders).

```
$ ls
file0a  file0b  folder1  folder2 link0a  link2a
```

Some terminal offer color output so you can differentiate normal files from folders. You can make the difference more clear with this

```
$ ls -aCF
./  ../  file0a  file0b  folder1/  folder2/ link0a@  link2a@
```

You will see a two extra directories "." and "..". Those are special folders that refer to the current folder and the folder up in the tree. Directories have the suffix "/". Symbolic links, kind of shortcuts to other files or directories are indicated with the symbol "@".

Another option to get more information about the files in the system is:

```
$ ls -al
total 36
drwxrwxr-x.  4 gufranco users     86 May 30 12:16 .
drwxr-xr-x. 82 gufranco users  12288 May 30 12:05 ..
-rw-rw-r--.  1 gufranco users      0 May 30 12:08 file0a
-rw-rw-r--.  1 gufranco users      0 May 30 12:08 file0b
drwxrwxr-x.  2 gufranco users     32 May 30 12:07 folder1
drwxrwxr-x.  2 gufranco users     32 May 30 12:07 folder2
lrwxrwxrwx.  1 gufranco users      6 May 30 12:16 link0a ->
   file0a
lrwxrwxrwx.  1 gufranco users     14 May 30 12:16 link2a ->
   folder2/file2a
```

Those characters on the first column indicate the permissions. The first character will be "d" for directories, "l" for symbolic links and "-" for normal files. The next 3 characters are the permissions for "read", "write" and "execute" for the owner. The next 3 are for the group, and the final 3 are for others. The meaning of "execute" for a file indicates that the file could be a script or binary executable. For a directory it means that you can see its contents.

1.2.2 cp

This command copies the contents of one file into another file. For example

```
$ cp file0b file0c
```

1.2.3 rm

This command deletes the contents of one file. For example

```
$ rm file0c
```

There is no such thing like a trash folder on a HPC system. Deleting a file should be consider an irreversible operation.

Recursive deletes can be done with

```
$ rm -rf folder_to_delete
```

Be extremely cautious deleting files recursively. You cannot damage the system as the files that you do not own you cannot delete. However, you can delete all your files forever.

1.2.4 mv

This command moves a files from one directory to another. It also can be used to rename files or directories.

```
$ mv file0b file0c
```

1.2.5 pwd

It is easy to get lost when you move in complex directory structures. pwd will tell you the current directory.

```
$ pwd
/home/gufranco/Dropbox/SummerHandsOn
```

1.2.6 cd

This command moves you to the directory indicated as an argument, if no argument is given, it returns to your home directory.

```
$ cd folder1
```

1.2.7 cat and tac

When you want to see the contents of a text file, the command cat displays the contents on the screen. It is also useful when you want to concatenate the contents of several files.

```
$ cat INCAR
system    =  LiAu
PREC       =  High
NELMIN     =  8
NELM       =  100
EDIFF      =  1E-07
...
```

To concatenate files you need to use the symbol ">" indicating that you want to redirect the output of a command into a file

```
$ cat file1 file2 file3 > file_all
```

The command tac shows the files in reverse starting from the last line back to the first one.

1.2.8 more and less

Sometimes text files, as those created as product of simulations are too large to be seen in one screen, the command ”more” shows the files one screen at a time. The command `"less"` offers more functionality and should be the tool of choice to see large text files.

```
$ less OUTCAR
```

1.2.9 ln

This command allow to create links between files. Used wisely could help you save time when traveling frequently to deep directories. By default it creates hard links. Hard links are like copies, but they make references to the same place in disk. Symbolic links are better in many cases because you can cross file systems and partitions. To create a symbolic link

```
$ ln -s file1 link_to_file1
```

1.2.10 grep

The grep command extract from its input the lines containing a specified string or regular expression. It is a powerful command for extracting specific information from large files. Consider for example

```
$ grep TOTEN OUTCAR
  free energy    TOTEN  =        68.29101273 eV
  free energy    TOTEN  =       -13.46870926 eV
  free energy    TOTEN  =       -18.78141268 eV
  ...
```

Regular expressions offers ways to specified text strings that could vary in several ways and allow commands such as grep to extract those strings efficiently. We will see more about regular expressions in third chapter.

1.2.11 More commands

The 10 commands above, will give you enough tools to move files around and travel the directory tree. There are more commands summarized

Output of entire files	
cat	Concatenate and write files
tac	Concatenate and write files in reverse
nl	Number lines and write files
od	Write files in octal or other formats
base64	Transform data into printable data

Formatting file contents	
fmt	Reformat paragraph text
numfmt	Reformat numbers
pr	Paginate or columnate files for printing
fold	Wrap input lines to fit in specified width

Output of parts of files	
head	Output the first part of files
tail	Output the last part of files
split	Split a file into fixed-size pieces
csplit	Split a file into context-determined pieces

Summarizing files	
wc	Print newline, word, and byte counts
sum	Print checksum and block counts
cksum	Print CRC checksum and byte counts
md5sum	Print or check MD5 digests
sha1sum	Print or check SHA-1 digests
sha2 utilities	Print or check SHA-2 digests

Operating on sorted files	
sort	Sort text files
shuf	Shuffle text files
uniq	Uniquify files
comm	Compare two sorted files line by line
ptx	Produce a permuted index of file contents
tsort	Topological sort

Operating on fields	
cut	Print selected parts of lines
paste	Merge lines of files
join	Join lines on a common field

Operating on characters	
tr	Translate, squeeze, and/or delete characters
expand	Convert tabs to spaces
unexpand	Convert spaces to tabs

Directory listing	
ls	List directory contents
dir	Briefly list directory contents
vdir	Verbosely list directory contents
dircolors	Color setup for 'ls'

Basic operations	
cp	Copy files and directories
dd	Convert and copy a file
install	Copy files and set attributes
mv	Move (rename) files
rm	Remove files or directories
shred	Remove files more securely

Special file types	
link	Make a hard link via the link syscall
ln	Make links between files
mkdir	Make directories
mkfifo	Make FIFOs (named pipes)
mknod	Make block or character special files
readlink	Print value of a symlink or canonical file name
rmdir	Remove empty directories
unlink	Remove files via unlink syscall

Changing file attributes	
chown	Change file owner and group
chgrp	Change group ownership
chmod	Change access permissions
touch	Change file timestamps

Disk usage	
df	Report file system disk space usage
du	Estimate file space usage
stat	Report file or file system status
sync	Synchronize data on disk with memory
truncate	Shrink or extend the size of a file

Printing text	
echo	Print a line of text
printf	Format and print data
yes	Print a string until interrupted

Conditions	
false	Do nothing, unsuccessfully
true	Do nothing, successfully
test	Check file types and compare values
expr	Evaluate expressions
tee	Redirect output to multiple files or processes

File name manipulation	
basename	Strip directory and suffix from a file name
dirname	Strip last file name component
pathchk	Check file name validity and portability
mktemp	Create temporary file or directory
realpath	Print resolved file names

Working context	
pwd	Print working directory
stty	Print or change terminal characteristics
printenv	Print all or some environment variables
tty	Print file name of terminal on standard input

User information	
id	Print user identity
logname	Print current login name
whoami	Print effective user ID
groups	Print group names a user is in
users	Print login names of users currently logged in
who	Print who is currently logged in

System context	
arch	Print machine hardware name
date	Print or set system date and time
nproc	Print the number of processors
uname	Print system information
hostname	Print or set system name
hostid	Print numeric host identifier
uptime	Print system uptime and load

Modified command	
chroot	Run a command with a different root directory
env	Run a command in a modified environment
nice	Run a command with modified niceness
nohup	Run a command immune to hangups
stdbuf	Run a command with modified I/O buffering
timeout	Run a command with a time limit

Process control	
kill	Sending a signal to processes

Delaying	
sleep	Delay for a specified time

Numeric operations	
factor	Print prime factors
seq	Print numeric sequences

1.3 Text Editors

There are several terminal-based editors available on our clusters. We will focus our attention to three of them: nano, emacs and vim. Your choice of an editor depends mostly on how much functionality do you want from your editor, how many fingers do you want to use for a given command and the learning curve to master it. For HPC users the editor is an important choice. Most of your time you are on the terminal executing commands or editing files, being those input files, submission scripts or the output of your calculations.

Lets review those three editors to give you the opportunity to have an informative choice.

1.3.1 Nano

Nano is a small, free and friendly editor with commands that usually manage using the Control (CTRL) combined with some other key.

You can start editing a file using a command line like this

nano myfile.f90

There are several commands available, the list below comes from the help text. When you see the symbol "^" it means to press the Control (CTRL) key, the symbol "M-" is called Meta, but in most keyboards is identified with the (Alt) key.

```
^G  (F1)            Display the help text
^X  (F2)            Close the current file buffer / Exit from
   nano
^O  (F3)            Write the current file to disk
^J  (F4)            Justify the current paragraph

^R  (F5)            Insert another file into the current one
^W  (F6)            Search for a string or a regular expression
^Y  (F7)            Move to the previous screen
^V  (F8)            Move to the next screen

^K  (F9)            Cut the current line and store it in the
   cutbuffer
^U  (F10)           Uncut from the cutbuffer into the current
   line
^C  (F11)           Display the position of the cursor
^T  (F12)           Invoke the spell checker, if available
```

The most basic usage is to edit a file, and exit from the editor with CTRL-X. Nano ask you if you want to save the file, you answer ”Y” and offers you a name. Simply press ENTER and your file is saved.

1.3.2 Emacs

Emacs is an extensible, customizable, open-source text editor. Together with Vi/Vim is one the most widely used editors in Linux environments. There are a big number of commands, customizations and extra modules that can be integrated with Emacs. We will just go briefly covering the basics.

The number of commands for Emacs is large, here the basic list of commands for editing, moving and searching text.

Entering Emacs

```
emacs <filename>
```

Leaving Emacs

```
suspend Emacs (or iconify it under X) C-z
exit Emacs permanently C-x C-c
```

Files

```
read a file into Emacs C-x C-f
save a file back to disk C-x C-s
save all files C-x s
```

```
insert contents of another file into this buffer C-x i
replace this file with the file you really want C-x C-v
write buffer to a specified file C-x C-w
toggle read-only status of buffer C-x C-q
```

Incremental Search

```
search forward C-s
search backward C-r
regular expression search C-M-s
reverse regular expression search C-M-r
select previous search string M-p
select next later search string M-n
exit incremental search RET
undo effect of last character DEL
abort current search C-g
Use C-s or C-r again to repeat the search in either direction.
   If
Emacs is still searching, C-g cancels only the part not
  matched.
```

Motion

```
entity to move over backward forward
character C-b C-f
word M-b M-f
line C-p C-n
go to line beginning (or end) C-a C-e
sentence M-a M-e
paragraph M-{ M-}
page C-x [ C-x ]
sexp C-M-b C-M-f
function C-M-a C-M-e
go to buffer beginning (or end) M-< M->
scroll to next screen C-v
scroll to previous screen M-v
scroll left C-x <
scroll right C-x >
scroll current line to center, top, bottom C-l
goto line M-g g
goto char M-g c
back to indentation M-m
```

Killing and Deleting

```
entity to kill backward forward
character (delete, not kill) DEL C-d
word M-DEL M-d
line (to end of) M-0 C-k C-k
```

```
sentence C-x DEL M-k
sexp M-- C-M-k C-M-k
kill region C-w
copy region to kill ring M-w
kill through next occurrence of char M-z char
yank back last thing killed C-y
replace last yank with previous kill M-y
```

Marking

```
set mark here C-@ or C-SPC
exchange point and mark C-x C-x
set mark arg words away M-@
mark paragraph M-h
mark page C-x C-p
mark sexp C-M-@
mark function C-M-h
mark entire buffer C-x h
```

Query Replace

```
interactively replace a text string M-%
using regular expressions M-x query-replace-regexp
Valid responses in query-replace mode are

replace this one, go on to next SPC or y
replace this one, don t move ,
skip to next without replacing DEL or n
replace all remaining matches !
back up to the previous match ^
exit query-replace RET
enter recursive edit (C-M-c to exit) C-r
```

Formatting

```
indent current line (mode-dependent) TAB
indent region (mode-dependent) C-M-\
indent sexp (mode-dependent) C-M-q
indent region rigidly arg columns C-x TAB
indent for comment M-;
insert newline after point C-o
move rest of line vertically down C-M-o
delete blank lines around point C-x C-o
join line with previous (with arg, next) M-^
delete all white space around point M-\
put exactly one space at point M-SPC
fill paragraph M-q
set fill column to arg C-x f
set prefix each line starts with C-x .
set face M-o
```

Case Change

```
uppercase word M-u
lowercase word M-l
capitalize word M-c
uppercase region C-x C-u
lowercase region C-x C-l
```

1.3.3 Vi/Vim

The third editor widely supported on Linux systems is ”vi”. Over the years since its creation, vi became the *de-facto* standard Unix editor. The Single UNIX Specification specifies vi, so every conforming system must have it.

vi is a modal editor: it operates in either insert mode (where typed text becomes part of the document) or normal mode (where keystrokes are interpreted as commands that control the edit session). For example, typing i while in normal mode switches the editor to insert mode, but typing i again at this point places an ”i” character in the document. From insert mode, pressing ESC switches the editor back to normal mode.

Vim is an improved version of the original vi, it offers

Here is a summary of the main commands used on vi. On Spruce when using ”vi” you are actually using ”vim”.

To Start vi

To use vi on a file, type in vi filename. If the file named filename exists, then the first page (or screen) of the file will be displayed; if the file does not exist, then an empty file and screen are created into which you may enter text.

```
vi filename  edit filename starting at line 1
vi -r filename recover filename that was being edited when
  system crashed
```

To Exit vi

Usually the new or modified file is saved when you leave vi. However, it is also possible to quit vi without saving the file. Note: The cursor moves to bottom of screen whenever a colon (:) is typed. This type of command is completed by hitting the ¡Return¿ (or ¡Enter¿) key.

```
:x<Return> quit vi, writing out modified file to file named in
   original invocation
:wq<Return>  quit vi, writing out modified file to file named
  in original invocation
:q<Return> quit (or exit) vi
:q!<Return>  quit vi even though latest changes have not been
  saved for this vi call
```

Moving the Cursor

Unlike many of the PC and MacIntosh editors, the mouse does not move the cursor within the vi editor screen (or window). You must use the the key commands listed below. On some UNIX platforms, the arrow keys may be used as well; however, since vi was designed with the Qwerty keyboard (containing no arrow keys) in mind, the arrow keys sometimes produce strange effects in vi and should be avoided. If you go back and forth between a PC environment and a UNIX environment, you may find that this dissimilarity in methods for cursor movement is the most frustrating difference between the two. In the table below, the symbol "^" before a letter means that the `<CTRL>` key should be held down while the letter key is pressed.

```
j or <Return> [or down-arrow]     move cursor down one line
k [or up-arrow]              move cursor up one line
h or <Backspace> [or left-arrow]  move cursor left one
  character
l or <Space> [or right-arrow]     move cursor right one
  character
0 (zero)                     move cursor to start of current
  line (the one with the cursor)
$                              move cursor to end of current
  line
w                              move cursor to beginning of next
   word
b                              move cursor back to beginning of
   preceding word
:0<Return> or 1G             move cursor to first line in file
:n<Return> or nG             move cursor to line n
:$<Return> or G              move cursor to last line in file
```

Screen Manipulation

The following commands allow the vi editor screen (or window) to move up or down several lines and to be refreshed.

```
^f move forward one screen
^b move backward one screen
^d move down (forward) one half screen
^u move up (back) one half screen
^l redraws the screen
^r redraws the screen, removing deleted lines
```

Adding, Changing, and Deleting Text

This command acts like a toggle, undoing and redoing your most recent action. You cannot go back more than one step.

```
 u UNDO WHATEVER YOU JUST DID; a simple toggle
```

Inserting or Adding Text

The following commands allow you to insert and add text. Each of these commands puts the vi editor into insert mode; thus, the ¡Esc¿ key must be pressed to terminate the entry of text and to put the vi editor back into command mode.

```
i  insert text before cursor, until <Esc> hit
I  insert text at beginning of current line, until <Esc> hit
a  append text after cursor, until <Esc> hit
A  append text to end of current line, until <Esc> hit
o  open and put text in a new line below current line, until <
   Esc> hit
O  open and put text in a new line above current line, until <
   Esc> hit
```

Changing Text

The following commands allow you to modify text.

```
r  replace single character under cursor (no <Esc> needed)
R  replace characters, starting with current cursor position,
   until <Esc> hit
cw change the current word with new text, starting with the
   character under cursor, until <Esc> hit
cNw  change N words beginning with character under cursor,
   until <Esc> hit; e.g., c5w changes 5 words
C  change (replace) the characters in the current line, until
   <Esc> hit
cc change (replace) the entire current line, stopping when <
   Esc> is hit
Ncc or cNc change (replace) the next N lines, starting with
   the current line, stopping when <Esc> is hit
```

Deleting Text

The following commands allow you to delete text.

```
x  delete single character under cursor
Nx delete N characters, starting with character under cursor
dw delete the single word beginning with character under
   cursor
dNw  delete N words beginning with character under cursor; e.g
   ., d5w deletes 5 words
D  delete the remainder of the line, starting with current
   cursor position
dd delete entire current line
Ndd  delete N lines, beginning with the current line; e.g., 5
   dd deletes 5 lines
dNd    same as Ndd
```

Cutting and Pasting Text

The following commands allow you to copy and paste text.

```
yy      copy (yank, cut) the current line into the buffer
Nyy     copy (yank, cut) the next N lines, including the
  current line, into the buffer
yNy     same as Nyy
p       put (paste) the line(s) in the buffer into the text
  after the current line
```

Searching Text

A common occurrence in text editing is to replace one word or phase by another. To locate instances of particular sets of characters (or strings), use the following commands.

```
/string  search forward for occurrence of string in text
?string  search backward for occurrence of string in text
n  move to next occurrence of search string
N  move to next occurrence of search string in opposite
  direction
```

Determining Line Numbers

Being able to determine the line number of the current line or the total number of lines in the file being edited is sometimes useful.

```
:.=  returns line number of current line at bottom of screen
:= returns the total number of lines at bottom of screen
^g provides the current line number, along with the total
  number of lines, in the file at the bottom of the screen
```

Saving and Reading Files

These commands permit you to input and output files other than the named file with which you are currently working.

```
:r filename<Return>          read file named filename and
  insert after current line (the line with cursor)
:w<Return>                  write current contents to file
  named in original vi call
:w newfile<Return>          write current contents to a new
  file named newfile
:12,35w smallfile<Return>  write the contents of the lines
  numbered 12 through 35 to a new file named smallfile
:w! prevfile<Return>         write current contents over a pre
  -existing file named prevfile
```

1.4 Environmental Modules

1.5 Using Torque and Moab

1.5.1 Job submission, monitoring, debugging and optimization

1.5.2 Chaining Job with dependencies

1.5.3 Executing many jobs at one, job arrays

1.5.4 Parallel jobs (MPI)

1.5.5 Managing the 3 HPC variables (cores, memory and time)

1.6 Transferring files between systems

2 Scientific Workflows (Building, HPC Running, and post-processing)

2.1 Shell Scripting

2.2 Python Scripting

2.3 Using Python pip and virtualenv

2.4 Plotting (gnuplot, xmgrace and matplotlib)

2.5 Building/installing software

Sometimes, the modules available on the system are not enough or you need to compile the code by yourself to get some extra functionality not present on the current modules.

For this tutorial we will go on the whole process of compiling a code by yourself. We have selected fftw a well known library for computing Fast Fourier Transforms.

First, we go to the webpage of the FFTW code `http://www.fftw.org`

Before downloading the code, create a directory on your home folder suitable for compiling codes, for example `"$HOME/local/src"` and go into such directory

Go to downloads and copy the link to the code that we will download and copy the link.

On the terminal execute:

```
wget http://www.fftw.org/fftw-3.3.6-pl2.tar.gz
```

Now that you have downloaded the code uncompress it using the command line

```
tar -zxvf fftw-3.3.6-pl2.tar.gz
```

Now go into that directory. There is usually a file `README` or `INSTALL`. Those files will give you instructions on how to compile the code.

It is a good idea to create a directory for building the code. Here we will use `build_gcc`. Go into that directory and execute:

```
../configure --help
```

You will see all the options available for configure the code. System administrators are usually conservative when choosing options for compilation, usually shifting the preference towards stability rather than performance. Consider for example the case where we want the long-double precision library rather than the original double precision. The configuration line will be like this:

```
../configure --prefix=$HOME/local --enable-long-double
```

The next step is compile the code with

```
make
```

It is always good practice to test the compilation. Good software comes with tests that compare results with known results.

```
make check
```

Finally, the installation is done with:

```
make install
```

Now, we have fftw for long-double precision compiled and installed at `$HOME/local`. Check by yourself the existence of folders such as lib and include, they contain both the libraries and headers needed to compile other programs using the library you just compiled.

We have a small program to test the FFT library we just compiled. The code is as follows:

```
#include <stdio.h>
#include <math.h>
#include <fftw3.h>

#define NUM_POINTS 10000
#define REAL 0
#define IMAG 1

void create_input(fftwl_complex* signal) {
  /* The input is a sum of several cosines and sines with
   different frequencies
   * and amplitudes
   */
  int i;

  printf("Creating a signal with precision LONG DOUBLE (sizeof
   =%lu Bytes)\n", sizeof(long double));

  for (i = 0; i < NUM_POINTS; ++i) {
    long double theta = (long double)i / (long double)
   NUM_POINTS * 2 * M_PI;

    signal[i][REAL] = 1.0 * cos(10.0 * theta) +
      2.0 * cos(20.0 * theta) +
      3.0 * cos(30.0 * theta) +
      4.0 * cos(40.0 * theta) +
      5.0 * cos(50.0 * theta);

    signal[i][IMAG] = 1.0 * sin(10.0 * theta) +
      2.0 * sin(20.0 * theta) +
      3.0 * sin(30.0 * theta) +
      4.0 * sin(40.0 * theta) +
      5.0 * sin(50.0 * theta);
  }
}

void print_magnitude(fftwl_complex* result, FILE *fp) {
  int i;
  for (i = 0; i < NUM_POINTS; ++i) {
    long double mag = sqrt(result[i][REAL] * result[i][REAL] +
         result[i][IMAG] * result[i][IMAG]);
    fprintf(fp,"%34.25Le %34.25Le %34.25Le\n", result[i][REAL],
    result[i][IMAG], mag);
  }
}

int main() {
  FILE *fp;
  fftwl_complex signal[NUM_POINTS];
  fftwl_complex result[NUM_POINTS];

  fftwl_plan plan = fftwl_plan_dft_1d(NUM_POINTS,
              signal,
              result,
              FFTW_FORWARD,
              FFTW_ESTIMATE);

  create_input(signal);
  printf("Saving input signal...\n");
  fp = fopen("Input_FFT.dat", "w");
  print_magnitude(signal, fp);
  fclose(fp);

  fftwl_execute(plan);

  printf("Saving FFT from signal...\n");
  fp = fopen("Output_FFT.dat", "w");
  print_magnitude(result, fp);
  fclose(fp);

  fftwl_destroy_plan(plan);
  return 0;
}
```

To compile this code you have to be very explicit on the locations of the libraries and headers because they are no included in the environmental variables that gcc uses to search for them. The compilation line will be:

```
gcc example_fftl.c -I$HOME/local/include -L$HOME/local/lib -
   lfftw3l -lm
```

When output is not declared like above, the executable is a file called `a.out`. We have the advantage that we produce a static library for the long-double precision version of FFTW.

The library is `libfftw3l.a`. The big advantage of static libraries is that the application can be certain that all its libraries are present and that they are the correct version. Being static for FFTW, our example simply runs with:

```
./a.out
```

You can check library dependencies executing

```
ldd ./a.out
```

Once executed you will have 2 files: `Input_FFT.dat` and `Output_FFT.dat`, those files contain the original signal and its Fourier-Transform function. We create a small python script to help you visualize both functions. The functions are in complex space, so you are drawing Real and Imaginary parts and the magnitude of the signal.

In the next section we will see how we can create modules that will facilitate compilation and execution of binaries that need these libraries.

2.5.1 Exercises:

- Compile the FFTW in their single precision (float) and double precision versions. You can use the same place `$HOME/local` as prefix for your installing the libraries. It is always good idea to clean the build directory before trying to compile a new version.
- Compile FFTW enabling the build of shared libraries. Try to compile the same code using them. Check dependencies with `ldd` and notice how the execution is not longer possible without explictly adding the environmental variable `LD_LIBRARY_PATH` pointing to the location of the libraries.

2.6 Creating Environmental Modules

Environment Modules (EM) provides a way for the dynamic modification of a user's environment via modulefiles. The Environment Modules package is a tool that simplify shell initialization and lets users easily modify their environment during the session. To achieve its goal, EM uses files called modules located on special locations, you can load and unload modules, changing the environment variables.

For this tutorial we will create a private repository for your own modules and we will create one module for the library we just create in our previous section.

The firs step is decide a place where we we locate the modules. For this tutorial we will use `$HOME/local/modules`. The location is arbitrary as soon as you can write in that directory. The first step is to create that directory.

The next step is to setup the variable `MODULEPATH` on your `.bashrc` pointing to the place where you will add your modules. Open you favorite text editor and edit your `$HOME/.bashrc`. You can add this at the very end of the file. Assuming you use bash:

```
export MODULEPATH=$HOME/local/modules:$MODULEPATH
```

In order to make effective the changes you need to "source" the file. incorporating the changes in your current session.

```
source $HOME/.bashrc
```

Now it is time to create your first module.

```
#%Module1.0#########################################
## Fast Fourier Transform Library
##

module-whatis "Name: fftw"
module-whatis "Version: 3.3.6"
module-whatis "Category: C subroutine library"
module-whatis "Description: Library for computing the discrete
   Fourier transform (DFT)"
module-whatis "URL: http://fftw.org/"

set  prefix     <ENTER_YOUR_HOME_DIR_HERE>/local

# This is used during compilation for searching for libraries
prepend-path  LIBRARY_PATH        ${prefix}/lib

# This is used during execution for searching for libraries
prepend-path  LD_LIBRARY_PATH     ${prefix}/lib

# This is used during compilation for searching headers (*.h
   and *.mod)
prepend-path  CPATH               ${prefix}/include

# These two are usual places where man pages and info pages are
    located
prepend-path  INFOPATH            ${prefix}/share/info:
prepend-path  MANPATH             ${prefix}/share/man

# This is a search path for searching for executables
prepend-path  PATH                ${prefix}/bin

# This is a helper tool used when compiling applications and
   libraries.
# It helps you insert the correct compiler options on the
   command line
prepend-path  PKG_CONFIG_PATH     ${prefix}/lib/pkgconfig
```

3 A glimpse on advanced topics

3.1 Advanced Bash/Python Scripting

Here we will cover a few more advanced topics not covered on our introductory scripting session.

3.1.1 Regular Expressions with python

Consider the following challenge. We have the output from a simulation with some data that we would like to process, the problem now is that the data is not on a single like, so a simple grep will not work. The data we want to parse looks like this:

```
...
     14        7.7300      0.00000
     15        7.9145      0.00000
     16        8.7421      0.00000

 k-point 115 :       0.4444    0.3636    0.4286
  band No.  band energies     occupation
      1      -6.7076      1.00000
      2      -6.6256      1.00000
      3      -3.8932      1.00000
      4      -3.8031      1.00000
      5       0.1344      1.00000
      6       0.4871      1.00000
      7       0.7520      1.00000
      8       1.1131      1.00000
      9       3.2272      1.00000
     10       3.3574      1.00000
     11       7.4689      0.00000
     12       7.4905      0.00000
     13       7.7325      0.00000
     14       7.9343      0.00000
     15       8.3742      0.00000
     16       8.7648      0.00000

 k-point 116 :       0.0000    0.4545    0.4286
  band No.  band energies     occupation
      1      -7.0118      1.00000
      2      -6.8668      1.00000
      3      -4.4179      1.00000
...
```

This is quite complex set of data and we would like to take the different elements in such a way that we can manipulate them later on.

There are several ways of solving this problem, for example, knowing that each block of data starts with "k-point" and spans 17 rows. Such task could be done using just grep

```
grep -A 17 k-point OUTCAR
```

The argument "-A 17" will tell grep to show 17 rows after each occurrence of the line k-point. We extract the line of information that we need but still is just a piece of text that is not so simple to manipulate.

With python we can achieve this task with just 4 lines, using the so called regular expressions, a way to explain a computer that we want extract text with some format by indicating the kind of data that we expect on the text.

This following script will extract the pieces still as text but, we will work on the conversion to text later.

```
import re
rf = open('OUTCAR')
data = rf.read()
kp = re.findall('k-point([\d\s]*):([\d\s.]*)band[\.\s\w]*
   occupation([\s\d:\-\.]*)\n\n', data)
```

The most cryptic part of this small script is understanding the meaning of all those symbols used as arguments for the findall function. Lets start with a simpler version of the findall line an we will understand it piece by piece.

Using IPython lets start with executing the first 3 lines and we will explore the findall function step by step

```
import re
rf = open('OUTCAR')
data = rf.read()
```

Now, we start exploring this line

```
re.findall('k-point[\d\s]*:', data)
```

The output will look like this:

```
['k-point  1 :',
 'k-point  2 :',
 'k-point  3 :',
 'k-point  4 :',
```

The line in findall can be read like this: Search for text that start with "k-point" followed by 0 or more (that is the meaning of '*') groups of characters (what is enclose by "[" and "]") that can be either numbers "\d" or characters that looks like spaces "\s"

Now, if we just need the number, we can enclose the information that findall will return by enclosing it in parenthesis, like this:

```
re.findall('k-point([\d\s]*):', data)
```

At this point could be interesting to show how we can convert the list of strings returned by findall into actual numbers. This could be done like this:

```
[int(x) for x in re.findall('k-point([\d\s]*):', data)]
```

Now lets move forward and get the next piece of information, the three numbers after colon, the numbers before the word 'band'

```
re.findall('k-point([\d\s]*):([\d\s.]*)band', data)
```

As you can see we are getting more information this time

```
[('   1 ', '        0.0000    0.0000    0.0000\n  '),
 ('   2 ', '        0.1111    0.0000    0.0000\n  '),
 ('   3 ', '        0.2222    0.0000    0.0000\n  '),
 ('   4 ', '        0.3333    0.0000    0.0000\n  '),
 ('   5 ', '        0.4444    0.0000    0.0000\n  '),
 ('   6 ', '        0.0000    0.0909    0.0000\n  '),
 ('   7 ', '        0.1111    0.0909    0.0000\n  '),
 ('   8 ', '        0.2222    0.0909    0.0000\n  '),
...
```

The output is a list of tuples, each tuple consisting of two strings. There is just one extra character on the regular expression, dot "." is added to cover the existence of that character in the 3 numbers after colon. In regular expressions "dot" is used to match any character except a newline. Inside the "[]" special characters lose their special meaning, so "dot" here means just a ".".

Now we can go to our final version of the findall function

```
re.findall('k-point([\d\s]*):([\d\s.]*)band[\s\w.]*occupation
   ([\s\d:.\-]*)\n\n', data)
```

The meaning of all this cryptic code become far more clear now, the only notice here is that the character minus "-" needs still to be escaped like "\-" because it has a meaning for ranges inside "[]". We close the regular expression with a double `\n\n`, indicating that each block is separated by a double newline.

Out final task is to convert the output from findall into actual numbers such that we can manipulate them for whatever purpose we need.

Lets do first a more simple exercise by storing correctly the k-point number and the three numbers after colon, they are the positions but their actual meaning is not important here.

```
ret=[]
for ikp in kp:
    entry={}
    entry['number']= int(ikp[0])
    entry['position']= [float(x) for x in ikp[1].split()]
    entry['values']= len(ikp[2].split())
    ret.append(entry)
```

What we are doing here is creating a list called **ret** and for each element in our list kp we will create a python dictionary, converting the elements from the tuple into numbers, the first one will be integer, the second one is a set of three floating point numbers and for the third one we will just split the string into words and count the elements.

```
[{'number': 1, 'position': [0.0, 0.0, 0.0], 'values': 48},
 {'number': 2, 'position': [0.1111, 0.0, 0.0], 'values': 48},
 {'number': 3, 'position': [0.2222, 0.0, 0.0], 'values': 48},
 {'number': 4, 'position': [0.3333, 0.0, 0.0], 'values': 48},
 {'number': 5, 'position': [0.4444, 0.0, 0.0], 'values': 48},
 {'number': 6, 'position': [0.0, 0.0909, 0.0], 'values': 48},
...
```

For the position we use a list comprehension, a syntactic construct available in some programming languages for creating a list based on existing lists.

The conversion of values is a bit more elaborated. First, notice that the final element contain 49 elements due to a final string with several dashes. We would like to extract the numbers that are really relevant the floating point numbers. Lets consider just the final element from kp

```
In [44]: kp[-1]
Out[44]:
(' 120 ',
 '       0.4444     0.4545     0.4286\n  ',
 ' \n      1      -6.6226       1.00000\n      2      -6.5937
     1.00000\n      3      -3.7536       1.00000\n      4
   -3.7218      1.00000\n      5      -0.0498      1.00000\n
     6       0.0803      1.00000\n      7       0.5565
   1.00000\n      8       0.6896      1.00000\n      9
   3.3146      1.00000\n     10       3.3603      1.00000\n
   11       7.6585      0.00000\n     12       7.6740
   0.00000\n     13       7.9721      0.00000\n     14
   8.0356      0.00000\n     15       8.8014      0.00000\n
   16       8.9382      0.00000\n\n\n
   ---------------------------------------------------------------------------------
   n')
```

Using Numpy we can easily get the information converted ready easily. Consider this line

```
import numpy
np.array(kp[-1][2].split()[:48], dtype=float).reshape(-1,3)
```

This line can be readed like this. Take the last element in kp (kp[-1]). Now take the third element of the tuple (kp[-1][2]). Split the string in words and make a list with the first 48 words encountered

```
kp[-1][2].split()[:48]
```

The final step is to convert those 48 strings into numbers as floating point numbers and reshape the whole array in 3 columns.

The final version of this part of the script will look like this:

```
ret=[]
for ikp in kp:
    entry={}
    entry['number']= int(ikp[0])
    entry['position']= [float(x) for x in ikp[1].split()]
    entry['values']= np.array(ikp[2].split()[:48], dtype=float)
   .reshape(-1,3)
    ret.append(entry)
```

For reasons that will become clearer later we would like to keep everything as simple lists of numbers rather than numpy arrays. So we will serialize the numpy array into a list of lists

```
ret=[]
for ikp in kp:
    entry={}
    entry['number']= int(ikp[0])
    entry['position']= [float(x) for x in ikp[1].split()]
    entry['values']= np.array(ikp[2].split()[:48], dtype=float)
   .reshape(-1,3).tolist()
    ret.append(entry)
```

It is time for us to save the data that we parse in something that allow us to recover later. There are several ways to store python objects into files. One way is using JSON, another is using pickle

Right now, the variable ret is a list of dictionaries where each of them contains either single numbers, lists or lists of lists. We can store that in a JSON file such that we can recover that information easily.

JSON is a lightweight data interchange format inspired by JavaScript object literal syntax. The JSON module in python offers convenient functions to convert simple variables such as list and dictionaries into strings that could be stored in text files such that their contents could be easily retrieved.

Try first executing something like this

```
import json
json.dumps(ret)
```

Not easy to read for a human but that long string can be easily understood by a computer to recover the information you stored in it. Try this version for something clearer to read

```
import json
json.dumps(ret, sort_keys=True, indent=4, separators=(',', ':
   '))
```

Lets now store ret into a file and read it again to test we can recover the file.

```
wf = open('k-points.json','w')
dp = json.dump(ret, wf, sort_keys=True, indent=4, separators
   =(',', ': '))
wf.close()
```

Now lets test recovering the data from the file.

```
rf2=open('k-points.json')
json.load(rf2)
```

Finally, lets summarize all that we learn with this example. The whole script will be listed here:

```python
import re
import numpy as np
import json

rf = open('OUTCAR')
data = rf.read()

# Parsing of the data
kp=re.findall('k-point([\d\s]*):([\d\s.]*)band[\s\w.]*
    occupation([\s\d:.\-]*)\n\n', data)

# Giving structure to the data
ret=[]
for ikp in kp:
    entry={}
    entry['number']= int(ikp[0])
    entry['position']= [float(x) for x in ikp[1].split()]
    entry['values']= np.array(ikp[2].split()[:48], dtype=float)
    .reshape(-1,3).tolist()
    ret.append(entry)

# Storing the results into a JSON file
wf = open('k-points.json','w')
dp = json.dump(ret, wf, sort_keys=True, indent=4, separators=('
    ,', ': '))
wf.close()
```

3.2 Programming in C, Fortran and Python

Learn a new programming language takes time and goes beyond we can pretend here. It is not only the actual knowledge of the syntax and grammar of the language is also an understanding of the expressiveness of each language for describing operations to a computer. There are 3 full featured programming languages in use today for scientific computing. Fortran, C and Python.

There are more specialized languages, such as R, Julia. But we will try to keep the things simple here by showing how the same task is programmed in the 3 languages we have selected. See for example 3.2 for a comparison on the performance of those different languages.

I am taking these examples from http://rosettacode.org. To give you a flavor of what is the feeling writing code in those 3 languages I have selected 2 tasks and showing how the solution is express in those languages.

3.2.1 Sieve of Eratosthenes

The Sieve of Eratosthenes is a simple algorithm that finds the prime numbers up to a given integer.

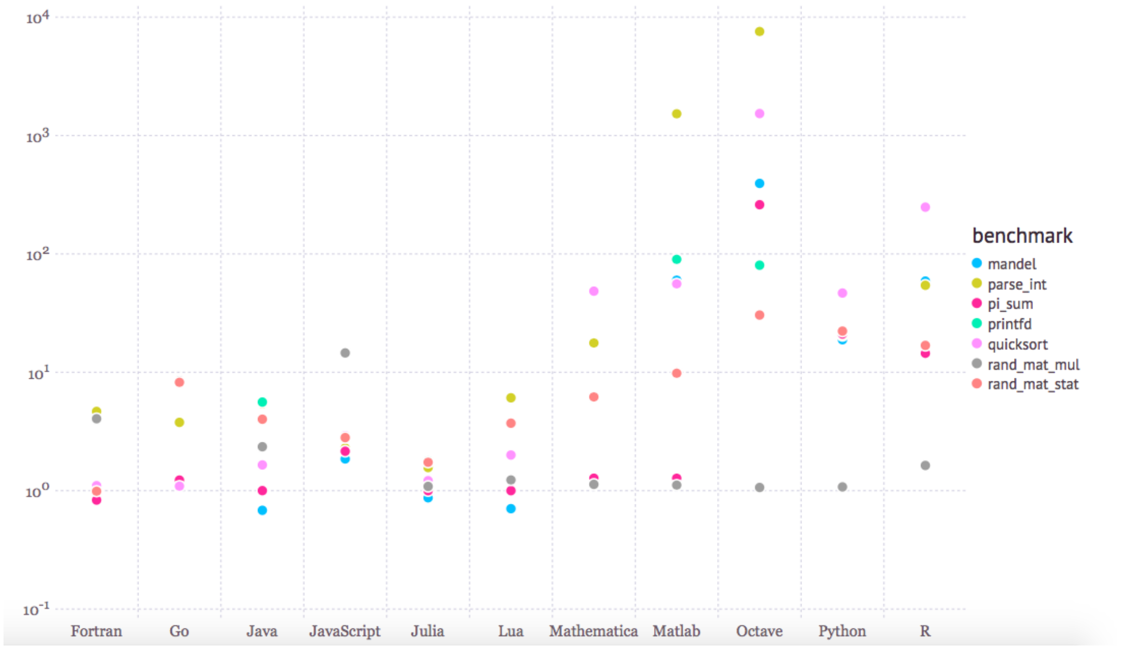

Figure 3.1: Comparison of performance for several computing languages. Benchmark times relative to C (smaller is better, C performance = 1.0). Source: https://julialang.org/benchmarks/.

Lets start with the implementation in C. I am selecting not the most optimized version, but the simplest implementation for pedagogical purposes. The idea is to get a flavor of the language.

```
#include <stdio.h>
#include <stdlib.h>

void sieve(int *, int);

int main(int argc, char *argv[])
{
  int *array, n;

  if ( argc != 2 ) /* argc should be 2 for correct execution */
    {
      /* We print argv[0] assuming it is the program name */
      printf( "usage: %s max_number\n", argv[0] );
    }
  else
    {
      n=atoi(argv[1]);
      array =(int *)malloc(sizeof(int));
      sieve(array,n);
    }
  return 0;
}

void sieve(int *a, int n)
{
  int i=0, j=0;

  for(i=2; i<=n; i++) {
    a[i] = 1;
  }

  for(i=2; i<=n; i++) {
    printf("\ni:%d", i);
    if(a[i] == 1) {
      for(j=i; (i*j)<=n; j++) {
  printf ("\nj:%d", j);
  printf("\nBefore a[%d*%d]: %d", i, j, a[i*j]);
  a[(i*j)] = 0;
  printf("\nAfter a[%d*%d]: %d", i, j, a[i*j]);
      }
    }
  }

  printf("\nPrimes numbers from 1 to %d are : ", n);
  for(i=2; i<=n; i++) {
    if(a[i] == 1)
      printf("%d, ", i);
  }
  printf("\n\n");
}
```

This example shows the basic elements from the c language, the creation of variables, loops and conditionals. The inclusion of libraries and the printing on screen.
You can compile this code using the code at

```
Day3_AdvancedTopics/2.Programming
```

```
gcc sieve.c -o sieve
```

and execute like this

```
./sieve 100
```

Now lets consider the Fortran version of the same problem.

```
module str2int_mod
contains

  elemental subroutine str2int(str,int,stat)
    implicit none
    ! Arguments
    character(len=*),intent(in) :: str
    integer,intent(out)         :: int
    integer,intent(out)         :: stat

    read(str,*,iostat=stat)  int
  end subroutine str2int

end module

program sieve

  use str2int_mod
  implicit none

  integer :: i, stat, i_max=0
  logical, dimension(:), allocatable :: is_prime
  character(len=32) :: arg

  i = 0
  do
    call get_command_argument(i, arg)
    if (len_trim(arg) == 0) exit

    i = i+1
    if ( i == 2 ) then
       call str2int(trim(arg), i_max, stat)
       write(*,*) "Sieve for prime numbers up to", i_max
    end if

  end do

  if (i_max .lt. 1) then
     write (*,*) "Enter the maximum number to search for primes
 "
     call exit(1)
  end if

  allocate(is_prime(i_max))

  is_prime = .true.
  is_prime (1) = .false.
  do i = 2, int (sqrt (real (i_max)))
    if (is_prime (i)) is_prime (i * i : i_max : i) = .false.
  end do
  do i = 1, i_max
```

```
    if (is_prime (i)) write (*, '(i0, 1x)', advance = 'no') i
  end do
  write (*, *)

end program sieve
```

You can notice the particular differences of this language compared with C, working with arrays is in general easier with Fortran.

You can compile this code using the code at

```
Day3_AdvancedTopics/2.Programming
```

```
gfortran sieve.f90 -o sieve
```

and execute like this

```
./sieve 100
```

Finally, this is the version of the Sieve written in python

```
#!/usr/bin/env python

from __future__ import print_function
import sys

def primes_upto(limit):
    is_prime = [False] * 2 + [True] * (limit - 1)
    for n in range(int(limit**0.5 + 1.5)): # stop at ``sqrt(
   limit)``
        if is_prime[n]:
            for i in range(n*n, limit+1, n):
                is_prime[i] = False
    return [i for i, prime in enumerate(is_prime) if prime]

if __name__=='__main__':

    if len(sys.argv)==1:
        print("Enter the maximum number to search for primes")
        sys.exit(1)
    limit = int(sys.argv[1])
    primes = primes_upto(limit)
    for i in primes:
        print(i, end=' ')
    print()
```

Python is an interpreted language so you do not need to compile it, instead directly execute the code at:

```
Day3_AdvancedTopics/2.Programming
```

using the command line:

```
python sieve.py 100
```

3.3 Introduction to Parallel Programming

Parallel programming is essential in High-Performance Computing. Computers nowadays are not increasing speed as they use to years ago. Instead, they increase the number of cores. Modern HPC clusters are now build from several nodes, with several processors each and with several cores each processor. Those processing capabilities are complemented by adding GPU and Co-processors such as Xeon Phi.

For this tutorial we will consider two popular alternatives for parallel computing, both OpenMP and MPI offers ways of execute calculations concurrently on several cores. An application can run on a computer cluster using both OpenMP and Message Passing Interface (MPI), such that OpenMP is used for parallelism within a (multi-core) node while MPI is used for parallelism between nodes.

We will explore those two kinds of parallel programming alternatives with a few examples each.

3.4 Parallel Programming with OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran.

The basic idea is to write special comments called “pragmas” that will be interpreted by the compiler when you compile the code with some special argument. In most cases the code can compile just fine without the extra argument and it will work as a serial code, using just one core.

Lets start with the usual hello program. This is the implementation in C

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]) {

  int nthreads, tid;

  /* Fork a team of threads with each thread having a private
   tid variable */
#pragma omp parallel private(tid)
  {

    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    /* Only master thread does this */
    if (tid == 0)
      {
  nthreads = omp_get_num_threads();
  printf("Number of threads = %d\n", nthreads);
      }
  }  /* All threads join master thread and terminate */

}
```

You compile it with the command

```
gcc -fopenmp omp_hello.c -o hello
```

The version in Fortran is:

```
program hello

  integer nthreads, tid, omp_get_num_threads,
   omp_get_thread_num

  !fork a team of threads with each thread having a private tid
    variable
  !$omp parallel private(tid)

  !obtain and print thread id
  tid = omp_get_thread_num()
  print *, 'Hello world from thread = ', tid

  !only master thread does this
  if (tid .eq. 0) then
     nthreads = omp_get_num_threads()
     print *, 'Number of threads = ', nthreads
  end if

  !all threads join master thread and disband
  !$omp end parallel

end program hello
```

You compile it with the command

```
gcc -fopenmp omp_hello.f90 -o hello
```

When you execute the program you will see messages comming from the different threads, all the program runs on the same machine but it creates threads to concurrently execute the section enclosed by the `"#pragma"` or `"!$omp"` blocks.

You can control the number of threads using

```
export OMP_NUM_THREADS=3
```

3.5 Parallel Programming with MPI

3.6 Test Driven Development

3.7 Version control with Git

The tutorial about Git will be based on the repository created at:

```
https://github.com/guilleaf/TutorialGitAutotools
```