Introduction to HPC Cluster (day2)

First Summer School in HPC and AI
7th - 8th June 2021 Free University of Bozen

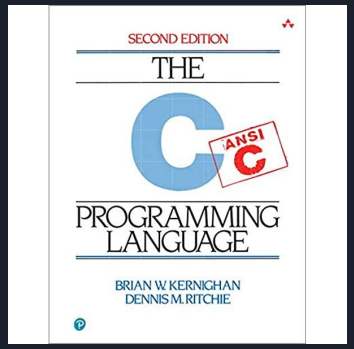
Marco Cianfriglia <m.cianfriglia@iac.cnr.it>

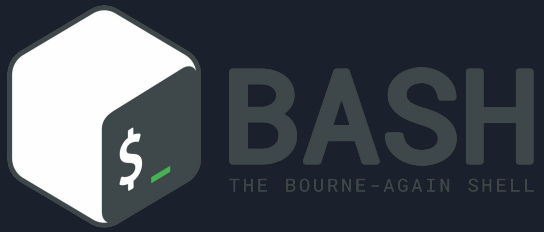
Agenda

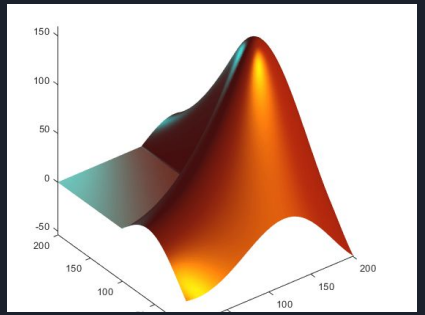
- Development tools, libraries, etc
- Schedulers and batch systems
- Hands-on

Development tools, libraries, etc..

Many languages, libraries ..







Compiled languages

- The source code is processed by special programs, called compilers, that generate machine code
- Low level languages, like C, are usually compiled
- The machine code generated is usually very efficient but not portable (i.e. you may need to recompile it in order to run on a different system)
- The compilation process usually executes several steps:
  - lexical analysis (also known as prepocessing)
  - syntax analysis
  - code generation

Interpreted languages

- The source code is parsed by a program, called interpreter, that executes the source code instructions directly
- The interpreter may use several strategies to parse the source code that may cause the translation of it in an intermediate representation, sometimes called bytecode
- Several examples:
  - Python
  - Matlab
  - R

Compiled vs Interpreted

Which is the right choice ? it depends

- Compiled languages, like C, usually performs better but they require good expertise
- Interpreted languages usually are easier to write and some of them, like Python, nowadays use wrapper to many C libraries for some computational intensive tasks

Makefile
(credits to D. A. Gaitros)

- What is ***make***?: The tool is designed to allow programmers to efficiently compile large complex programs with many components easily.
- You can place the commands to compile a program in a Unix script but this will cause ALL modules to be compiled every time.
- The ***make*** utility allows us to only compile those that have changed and the modules that depend upon them

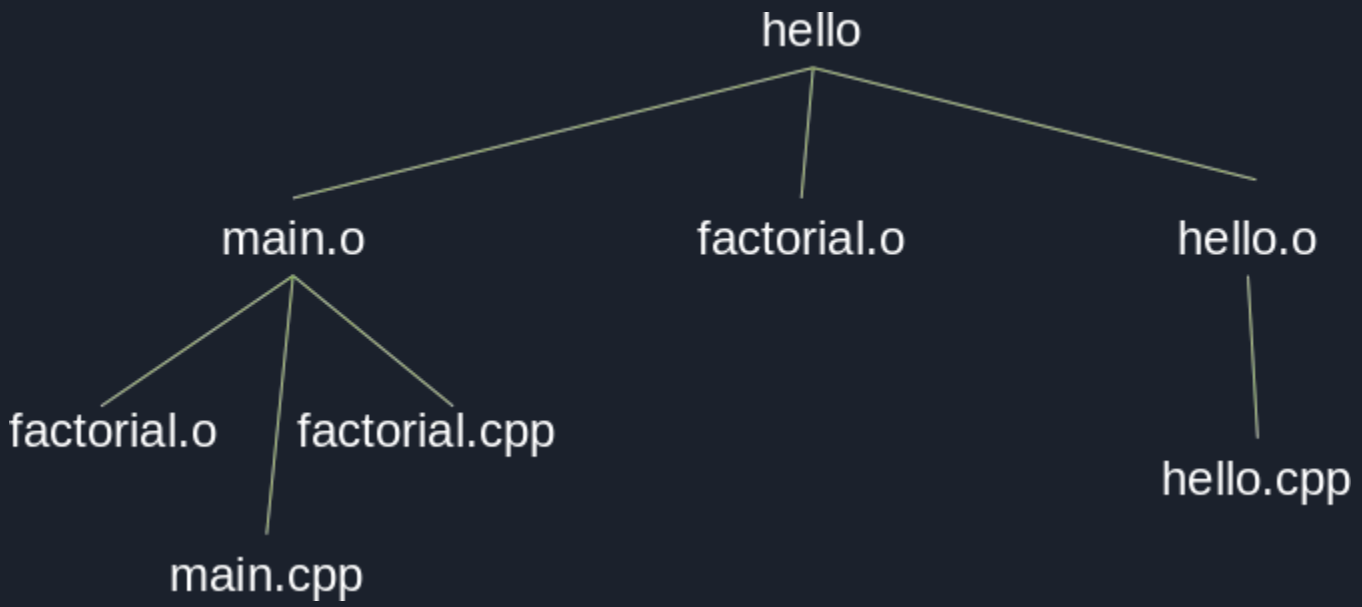
Makefile: how does it work? (credits to D. A. Gaitros)

- In Unix, when you run ***make*** it search for a file called *makefile* or *Makefile*
- A makefile contains a series of directives that tell the ***make*** utility how to compile your program and in what order.
- Each file will be associated with a list of other files by which it is dependent. This is called a *dependency line*.
- If any of the associated files have been recently modified, the ***make*** utility will execute a directive command just below the *dependency line*.

Makefile: simple make (credits to D. A. Gaitros)

```
hello: main.o factorial.o hello.o
	g++ main.o factorial.o hello -o hello


main.o: main.cpp
	g++ -c main.cpp

factorial.o: factorial.cpp
	g++ -c factorial.cpp

hello.o: hello.cpp
	g++ -c hello.cpp

clean:
	rm -rf *.o hello
```

Makefile: Tree
(credits to D. A. Gaitros)



Makefile: components (credits to D. A. Gaitros)

- Comments
- Dependency lines
- Shell lines
- Rules
- Inference rules

Makefile: components (credits to D. A. Gaitros)

- A ***comment*** is indicated by the character "#", and can start anywhere
- The lines with ":" are called ***dependency lines***
  - on the left are the ***dependencies***
  - on the right are the ***sources*** needed to make the dependency
  - ***make*** is recursive: it checks all dependencies are not out-of-date before completing building process
  - **The order of dependencies is important!**
- ***Shell lines*** follow the dependency lines.
  - They tell make how to build the target
  - A target may have more than one shell line
  - Each shell line must be preceded by a ***tab***
  - After each shell line is executed, make checks the ***exit status.***
    - if 0, ok
    - otherwise make will stop and display an error

Makefile: components (credits to D. A. Gaitros)

- A ***rule*** tell make when and how to make a file. The format is :
  - A rule must ha a **dependency line** and may have and **action or shell line** after it
  - The action/shell line is executed if the dependency line is ***out of date***
  - Example:

```
hello.o: hello.cpp          #dependency line
        g++ -c hello.cpp    #shell line
```

Rule

- ***Inference rules*** are a method of **generalizing** the build process
  - It is a sort of **wildcard** rule, % character is used as wildcard
  - Example:

```
%.o : %.c
        $(CC) $(FLAGS) -c $(SOURCE)
#All .o files have dependencies of all .c files of the same name
```

Makefile: components (credits to D. A. Gaitros)

Macros

- Basically it is a shorthand or alias used in the makefile
- A string is associated with another usually larger string
- Inside the file, to **expand** a macro, you have to place the string inside of $( ).
- The whole thing is expanded during execution of the ***make*** utility.

Example
HOME=/home/project
SOURCE=$(HOME)/src
CFLAGS= -O3 -g -lpthread -lm

Makefile built-in variables

- They refer to specific parts of rules

```
eval.o: eval.c eval.h
        gcc -c eval.c
```

- $@ - the target rule name (eval.o)
- $< -the first dependency name (eval.c)
- $^ - the names of all the dependencies (eval.c eval.h)
- $? - The names of all the dependencies that are newer than the target

Exercise

- Copy the directory /home/clusterusers/sc/Day_2/exercises on your home
- Go to ~/exercises/C/simple_hello
- Print the content of Makefile
- Run make and then execute simple_hello
- Run make again. What do you expect it to do?
- Open with vi the file simple_hello.c, save it (:w) and then quit (:q)
- Run make. What do you expect this time?

Exercise Solution

- Copy the directory /home/clusterusers/sc/Day_2/exercises on your home
  - ***cp -r /home/clusterusers/sc/Day_2/exercise ~/***
- Go to ~/exercises/C/simple_hello
  - ***cd ~/exercise/C/simple_hello***
- Print the content of Makefile
  - ***cat Makefile***
- Run make and then execute simple_hello
  - make && ./simple_hello
- Run make again. What do you expect it to do?
  - ***make; It will do nothing***
- Open with vi the file simple_hello.c, save it (:w) and then quit (:q)

- Run make. What do you expect this time?
  - ***It will compile the file again***

module

A user can customize her/his shell environment by using the ***module*** command.
It requires one of the following **subcommand**

- **apropos/keyword** <*string*>: show all modulefiles that contain the searched string within their information
- **avail**: print the list of available modules
- **add**/**load** <*modulefile*>: load the specified modulefile in the user shell environment
- **list**: print all the loaded modulefiles
- **rm**/**unload** <*modulefile*>: remove the specified modulefiles from the user shell environment
- **purge**: unload all the modulefiles.

Why module?

- It is easy for users to customize their environment, no need to configure and build packages
- Multiple versions of packages and libraries can be available on the system
- On a multi-user system, it reduces the proliferation of the same library installed locally by many users
- The modules are installed and managed by system administrators
- Once used on HPC system, guarantees all nodes have the same environemt

Exercise

1. List all the available modulefiles
2. Search for the modulefile that match the following string “4.1”
3. Unload, if necessary, all the previously loaded modulefiles
4. Add the modulefile obtained from the search (step 1)
5. Verify the selected modulefile has been successfully loaded

Exercise Solution

1. List all the available modulefiles
   - ***module avail***
2. Search for the modulefile that match the following string “4.1”
   - ***module apropos 4.1***
   - ***module keyword 4.1***
3. Unload, if necessary, all the previously loaded modulefiles
   - ***module purge***
4. Add the modulefile obtained from the search (step 1)
   - ***module load openmpi-4.1.1***
5. Verify the selected modulefile has been successfully loaded
   - ***module list***

Exercise

1. Check which version of python is available on the login-node
2. Use vim to write the following line into the file testversion.py:
   print “Python test”
3. Use module to load python version 2
4. Run the testversion.py script. Does it work?
5. Use module to switch to python 3.7.7
6. Run the testversion.py script. Does it work?

Exercise Solution

1. Check which version of python is available on the login-node
   - ***module avail***
2. Write the following line to the file testversion.py:
   print “Python test”

3. Use module to load python version 2
   - ***module load python-2.7.18***
4. Run the testversion.py script. Does it work?
   - ***python testversion.py; Yes***
5. Use module to switch to python 3.7.7
   - ***module switch python-2.7.18 python-3.7.7***
6. Run the testversion.py script. Does it work?
   - ***python testversion.py; No, python3 requires ()***

Schedulers and batch system

Slurm
Simple Linux Utility for Resource Management

- It is an open-source, fault-tolerant, and highly scalable, cluster management and job scheduling system for large and small Linux clusters
- It assigns exclusive and non-exclusive access to cluster resources (nodes) to users for the requested time so they can use the resources
- It provides a framework for starting, executing and monitoring work (jobs) on the allocated nodes
- It manages the queue of pending work by mediating the contention to the resources

Slurm commands

- sinfo
- salloc
- srun
- sacct

- sbatch
- scontrol
- scancel
- …. and more

Here a good news: all these commands support (more or less) the same options

sinfo

Print information about Slurm nodes and partitions

Example: print name and number of cpus for each node belonging to sc partition

sinfo -p sc -o %N,%c # -p <partition>
-o output-format %N =nodes name, %c number of cpu

Exercise: print name, hostname, size of memory, for the node ‘hpcazu2’ (hint see man sinfo)

Before starting

- Please remember that the resources for the summer school are shared among all of you
- Do not allocate too much resources as it will negatively impact on all of us
- Default resources allocation:
  2 nodes, 1 cpu-per-node, 100MB of memory-per-cpu
- On many HPC systems you will get some resource *budget* for your project. Usually, it is not a good idea over-allocate resources

Before starting

- On HPC system, usually users may request to run jobs interactively or batch
- Interactively jobs are useful to
  - check the environment is properly configured
  - check the output of (small) runs and verify everything is working
  - perform very small tasks
  - exercise
- Once a user is (quite) confident everything is working properly, it is a good time to start the real computation, and submit the batch tasks

salloc

- It is used to obtain a Slurm job allocation (a set of nodes)
- It then allows user to execute a command on the allocated nodes
- Once the command is finished, it releases the resource allocation
- It has many options to customize your job allocation request (see man salloc)
  We will mainly use the following (valid also for the other commands)
  - -A (--account=) sc-users
  - -p (--partition=) sc
  - -j (--job-name)
  - --mem-per-cpu=100M
  - -N (--nodes=) 2

srun

- Run a parallel job on a cluster managed by Slurm
- If necessary, it will first create a resource allocation in which to run the parallel job

Example

srun -N2 -A sc-user -p sc --mem-per-cpu=100MB --time=00:01:00 hostname
#It will execute the ***hostname*** command on two nodes -

Try it!!

sacct

It allows users to display accounting data for all the jobs invoked with Slurm

```
gst_mcianfriglia@slurm-ctrl:~$sacct
       JobID    JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
53286        submit_ho+         sc   sc-users          2  COMPLETED      0:0
53286.batch       batch              sc-users          1  COMPLETED      0:0
53286.0      hostname.+              sc-users          2  COMPLETED      0:0
53287                 t1         sc   sc-users          1  COMPLETED      0:0
53287.batch       batch              sc-users          1  COMPLETED      0:0
53287.0          matlab              sc-users          1  COMPLETED      0:0
53288        submit_mm+         sc   sc-users          1 CANCELLED+      0:0
53288.batch       batch              sc-users          1  CANCELLED     0:15
53288.0          python              sc-users          1  CANCELLED     0:15
53289        submit_mm+         sc   sc-users          1    TIMEOUT      0:0
53289.batch       batch              sc-users          1 OUT_OF_ME+    0:125
53289.0          python              sc-users          1 OUT_OF_ME+    0:125
```

sbatch

- It allows to submit a batch script to Slurm
- Immediately after the script is successfully transferred to the Slurm controller and assigned a Slurm Job id, it returns the job id
  - Once a script is successfully submitted, it may stay in the pending jobs queue waiting for the requested resources to become available
- The stdout and stderr will be redirect by default on the file slurm-%j.out, where %j indicates the job-id.
- An sbatch script may contain some options in the form of #SBATCH directives
  - Example #SBATCH -A sc-users

sbatch script example

```
#!/bin/bash
#submit simple_hello

#SBATCH -A sc-users
#SBATCH --partition=sc
#SBATCH -N 2
#SBATCH --mem-per-cpu=10M
#SBATCH -J simple_hello


srun ./simple_hello “Hello Everybody”
```

```
#!/bin/bash
#submit simple_hello.py

#SBATCH -A sc-users
#SBATCH --partition=sc
#SBATCH -N 2
#SBATCH --mem-per-cpu=10M
#SBATCH -J simple_hello_python
module purge
module load python-3.8.2

srun python simple_hello.py
```

squeue

- ***squeue***: shows information about jobs located in the Slurm scheduling queue

```
gst_mcianfriglia@slurm-ctrl:~/BASH$squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
             53290        sc hostname gst_mcia PD       0:00      3 (Resources)
             53292        sc hostname gst_mcia PD       0:00      3 (Resources)
             53293        sc hostname gst_mcia PD       0:00      3 (Resources)
             53294        sc hostname gst_mcia PD       0:00      3 (Resources)
```

- The ST column reports the Job state:
  PD = Pending; R = Running; CA = Cancelled; CF = Configuring;
  CG = Completing; CD = Completed; F = Failed; TO = TimeOut;
  NF = NodeFailure; RV = Revoked; SE = Special Exit State

scontrol and scancel

- ***scontrol*** is a powerful tool to view and modify Slurm configuration, including
  - job, node, partition, etc
- it supports many commands, most of them are reserved to root accounts
- A standard user may use it to check its jobs status
  - scontrol show job <jobid
- ***scancel***: it is used to signal or cancel jobs:
  - Ex. ***scancel 53290 53294***

Exercise

1. Go to ~/exercise/BASH/hostname and submit a job using submit_hostname.sh script

2. Go to ~/exercise/BASH/list_remote_dir. Open the file list_remote_dir.sh and follow the instructions. Once you are ready, use submit_list-remote-dir.sh to submit a job

3. Go to ~/exercise/BASH/sort-list. Complete the submit_sort-list.sh script and submit the job

4. Go to ~/exercise/C/simple_hello
    a. If simple_hello exist, remove it
    b. Load the modulefile gcc-6.5.0 on your environment
    c. make
    d. Make sure the submit_simple_hello.c contains will use the same environment when run it

Exercise

1. Go to ~/exercise/C/compute_pi
   a. Load the modulefile gcc-6.5.0 on your environment
   b. make
   c. Open the file submit_compute_pi_host.sh and finish to write it

2. Go to ~/exercise/Python/testversion
   a. Write the sumbit_testversion.sh script

3. Go to ~/exercise/Matlab/mm and submit a job using the submit_mm.sh script
   Please take a look of the script content

4. Go to ~/exercise/MPI/simple_hello and submit a job using the submit_mpi_hello.sh

Thank you for the attention

Feel free to contact me for any question/issue