Universitat Pompeu Fabra Barcelona

Seminar 1

Ricard Borrell Pol[†], **Sergi Laut Turón**[‡]

Summary. In this Seminar we will see the basic tools to work with a remote cluster. We will work with an example of a code to find prime numbers to test some of the tools and evaluate performance.

1. Environment

In this Section we present the environment we will be using throughout the course as well as some useful tools that will help you for the lab sessions.

1.1. Cluster

For this course we will work with a remote cluster.

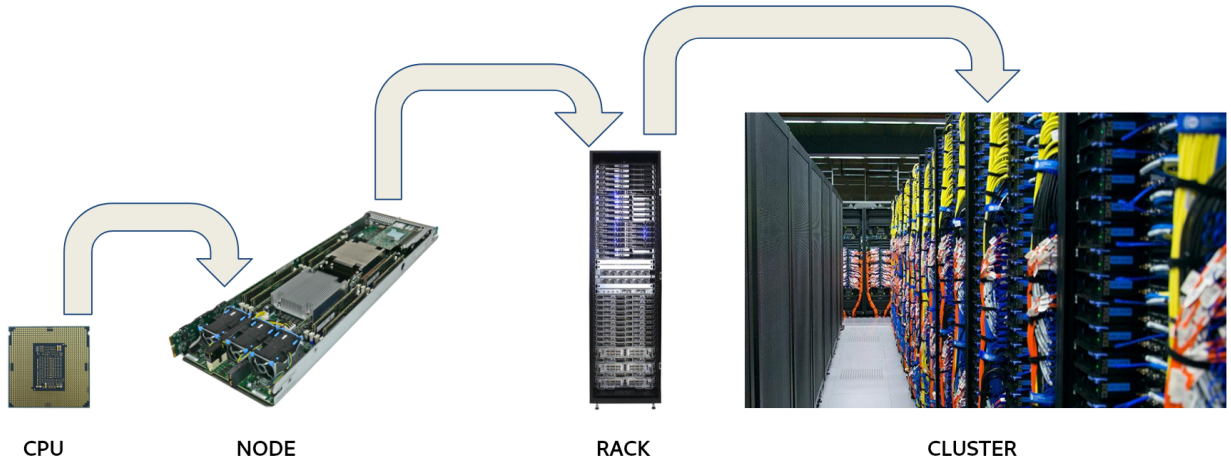


Figure 1. Cluster structure.

A cluster is essentially a group of computers connected through an interconnection network aimed to solve a common task by working together. They can be viewed as a single system. It is composed of several racks interconnected by a fast network. Each rack consists of nodes that may have CPUs or GPUs. The fast interconnection network allows all nodes to communicate with each other.

Clusters are used for computing different types problems and nodes are shared among different jobs and users.

In Barcelona we have the MareNostrum IV supercomputer with a total peak performance of 11.15 PFlops, consisting of 48 racks with 72 nodes each. In each node there are 2x Intel Xeon Platinum 8160 with 24 cores

[†] ricard.borrell@upf.edu.
[‡] sergi.laut@upf.edu.

operating at 2.1 GHz. For this course we will use the Amazon Web Services (AWS) cluster.

1.2. Amazon Virtual Private Cluster (VPC)

Amazon Web Services (AWS) provides a wide range of resources and configurations. In this course we will be using AWS resources with an Amazon Virtual Private Cluster (VPC). VPC is a configuration of AWS instances configured in a network of resources that provides a virtual cluster.

The configuration that we will be using during this session and the first 2 labs is set up with the following machines:

- 1 login node (Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz): the login node is not used for computation, but to send submit jobs to the computational nodes. The login node shares the file system with the cluster. Therefore, we will be able to retrieve results calculated in the clusters from there.
- 4 computational nodes (AMD EPYC 7R32): these four compute nodes are used for computation,the programs will run here and will save the results in the shared file system.

After 30' of inactivity the cluster will shutdown. When requesting it to get back to activity it will take more than 10' to be active again. The idea of having a cluster for this course is to give you the opportunity of having an experience with a real cluster that is employed in modern research and industrial centres. The technologies used throughout the course are the same used in supercomputing centers and high performance applications.

IMPORTANT NOTE:

- Because the lab assignments will be tackling different computer architectures, we will work with different cluster configurations through the course, so the cluster configuration and login IP will change. Most of the concepts that you will see here will work on all of them, but aspects such as the parameters of the configuration can change in future lab sessions. Moreover, your password will also change when the cluster changes, but we will provide the new one to you.

1.3. Remote connection

Clusters are usually accessed remotely, as they may be located far away. We will use SSH to connect to our cluster. Secure Shell (SSH) is a network protocol that allows to connect two systems securely through an unsecured network. It is commonly used to remotely connect and login to other systems and execute commands.

Depending on your operating system there are different ways to connect to a remote server.

1.3.1. LINUX

To establish an SSH connection open a terminal and run the following command:

```
$ ssh UNIS@10.49.0.109
```

Where UNIS is your UPF id (e.g u123456) and the password will be provided to you by the teachers. The first time you have logged in you will be requested to change your password. After entering your password, you are connected to the login node and all the commands used in this terminal will be executed in the cluster.

Once connected to the cluster you may need to send or receive files to or from the cluster. You can do it with the SCP command, which allows to transfer data between different hosts using SSH. In this documentation you will find how to use the commands depending on your situation. Note SCP commands have to be executed from your local computer. These two commands allow to receive and send, files between a remote host and a local host.

```
$ scp your_username@remotehost.edu:foobar.txt /some/local/directory
```

To copy data from the cluster to the local machine.

```
$ scp foobar.txt your_username@remotehost.edu:/some/remote/directory
```

To copy data from the local machine to the cluster.

A graphical drag-and-drop tool to perform data transfers is FileZilla. You can find the client for Linux here. FileZilla is also available for Windows and macOS. In any case when configuring it use port 22, which is used for SSH connections.

1.3.2. WINDOWS

Windows users can download PuTTY to connect via SSH to the cluster. PuTTY is a free open source software that allows to remotely connect to other servers.

To connect to our cluster simply configure it with the host IP, 10.49.0.109 and press Open. A terminal will open. Introduce your UNIS as your id and the password that will be provided to you by the teachers. The first time you have logged in you will be requested to change your password. After entering your password, you are connected to the login node and all the commands used in this terminal will be executed in the cluster.

To move data between your local system and the remote login node you can use WinSCP, FileZilla or MobaXterm among other options.

Windows users can also install a Ubuntu terminal and follow the Linux commands.

1.3.3. MACOS

To establish an SSH connection open a terminal and run the following command:

```
$ ssh UNIS@10.49.0.109
```

Where UNIS is your UPF id (e.g u123456) and the password will be provided to you by the teachers. The first time you have logged in you will be requested to change your password. After entering your password, you are connected to the login node and all the commands used in this terminal will be executed in the cluster.

Once connected to the cluster you may need to send or receive files to or from the cluster. You can do it with the SCP command, which allows to transfer data between different hosts using SSH. In this documentation you will find how to use the commands depending on your situation. Note SCP commands have to be executed from your local computer.

A graphical drag-and-drop tool to perform data transfers is FileZilla. You can find the client for macOS here.

1.4. FileZilla

Drag and drop tools are easy to use and allow to perform many actions that you will commonly require when working with a cluster. Here we explain the basics of FileZilla, which is a common tool for the different operating systems. Note this is not necessary if you already use another method to connect to the cluster.

Download Filezilla from the provided link. Once opened, Filezilla will look as in Figure 3. The configuration is straight forward:

- Set as host the private IP address of the cluster, 10.49.0.109.
- Use your UPF username and password.
- Set 22 as port and click Quickconnect.
- You will be prompted with a password remembering query and to trust the host you are connecting (accept it).

Once successfully connected you will have access to your local filesystem (in blue) and to your remote filesystem (in red) inside the cluster. To move data from one system to another simply drag and drop the folder or file you want onto the folder you want it to be in.

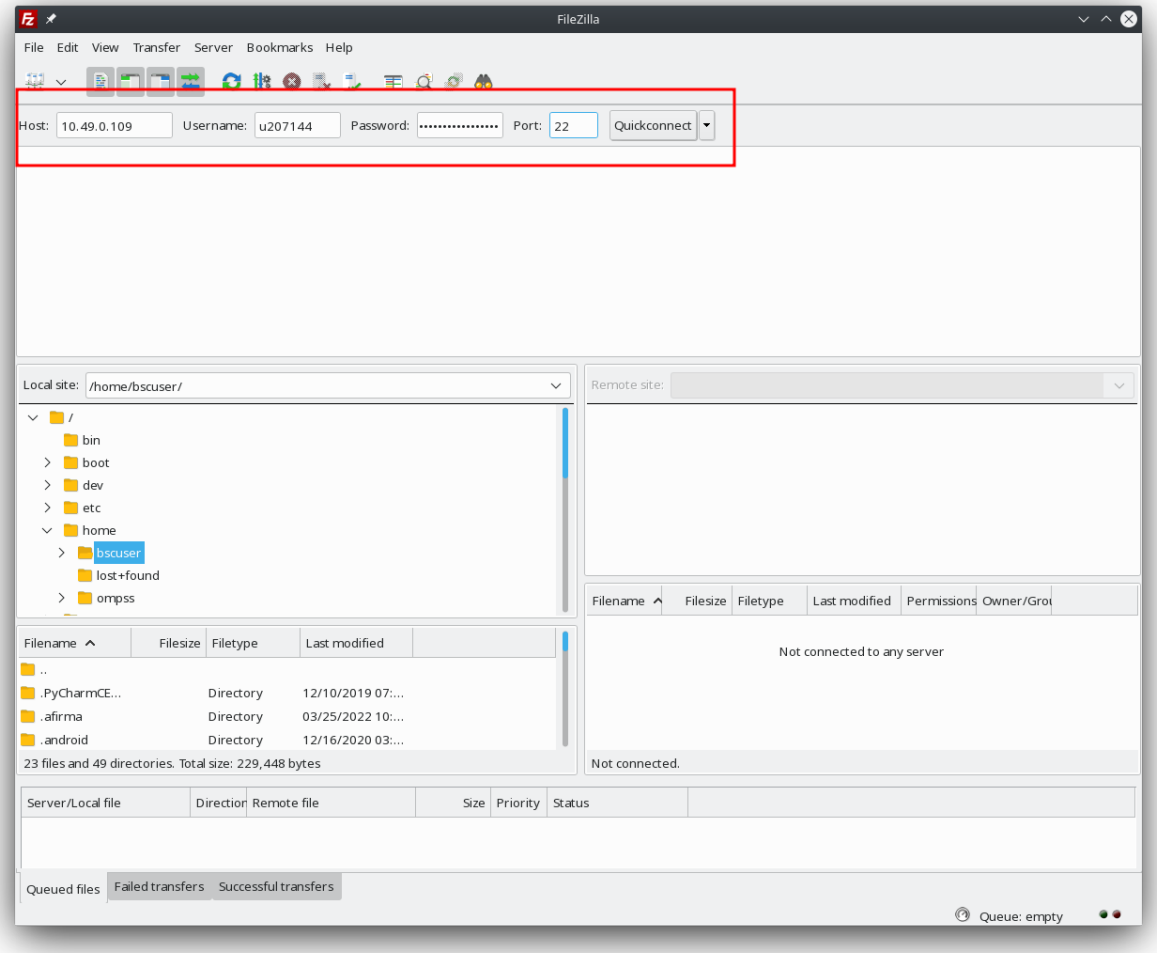


Figure 2. FileZilla window. Inside the red rectangle you have to input the configuration.

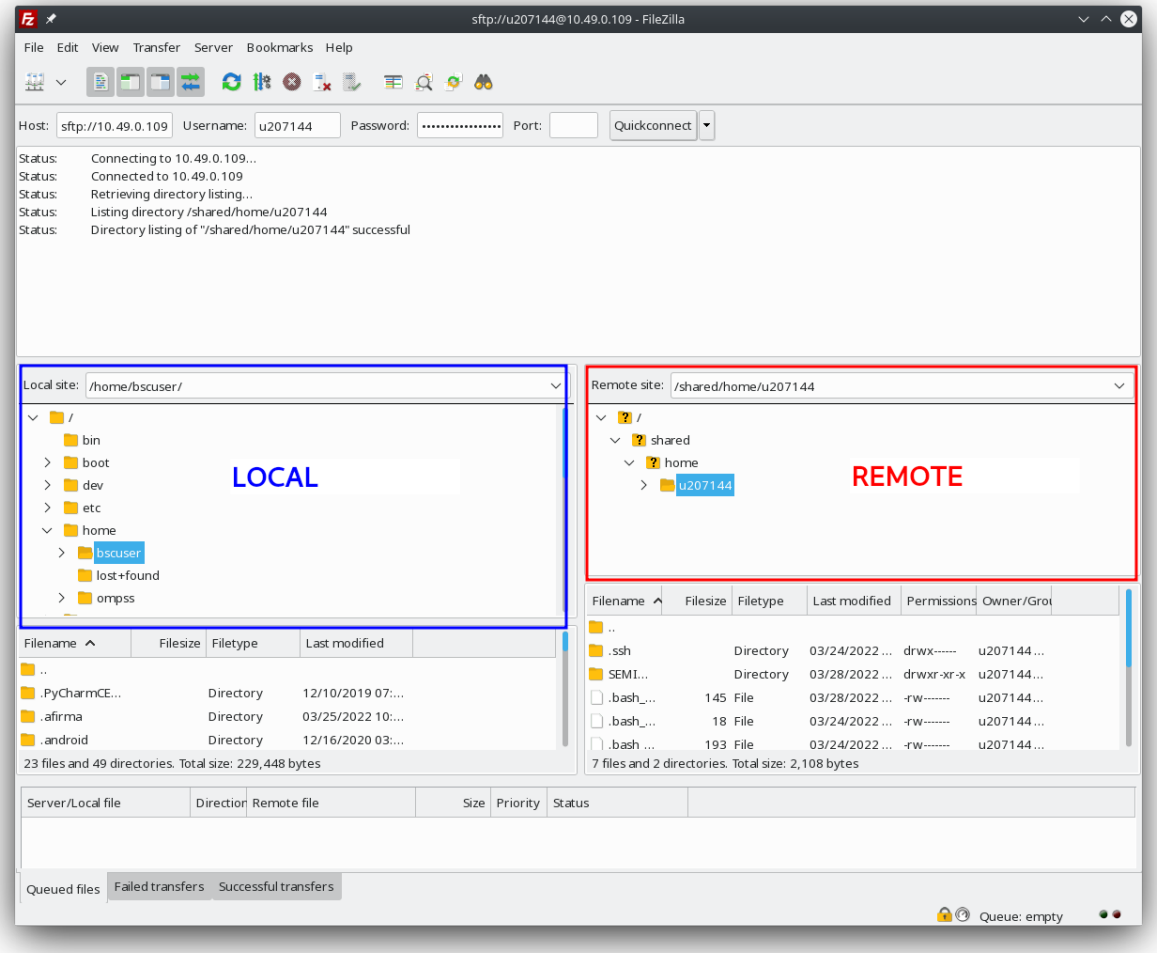
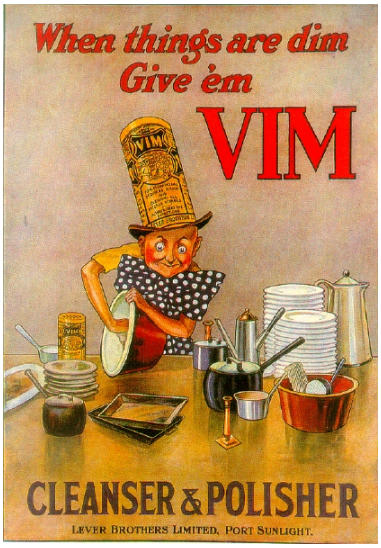


Figure 3. FileZilla window after connection.

1.5. Text editor - VIM/Nano

While graphical and drag-and-drop tools are simple to use and allow to edit your files locally, every time a you make a local change you have to send your files to the cluster, and when you have to evaluate results get them to your computer. Another option is to read and edit your files directly in the cluster using the command line editors VIM or Nano. Both options allow to modify any file and save changes through the terminal. To open a file with VIM use in the terminal:

```
$ vim name_of_file
$ vi name_of_file
```



Here we will show some of the basic commands to use VIM in its several editing modes:

- Insert mode: Accessed by pressing *i* in the normal mode. It allows to edit the file by directly writing. To exit the mode press *ESC*.
- Normal or command mode: This is the default mode when a file is opened It can be accessed from any other mode pressing key *ESC*. It is used for editor commands such as opening and closing files, saving changes, find and replace, copy and paste, navigate a file, among many others. While in this mode you can use the arrows to move through the file, but it can not be directly edited. Some of the most used commands are:
  - *:q* to close the file. *:q!* if there are changes you do not want to save.
  - *:w* to save changes in a file. *:wq* combines both commands.
  - *:sav file* to save the file with a new name.
  - *G* to move to the end of the file
  - *gg* to move to the beginning of the file.
  - *:80* move to line 80.
  - *y* to copy selected text.
  - *p* to paste the selected text.
  - *dd* to cut a line.
  - *yy* to copy current line.
  - *u* to undo last change.
  - *ctrl+r* to redo last change.
  - */foo* searches the next appearance of the string *foo* in the file starting from the place the cursor is to the bottom.
  - *?foo* searches the next appearance of the string *foo* in the file starting from the place the cursor is to the top.
  - * searches the text under the cursor.
  - *:%s/foo/bar/gic* replaces all appearances of *foo* with *bar*. Final keys *g*, *i*, *c* are optional to define a global file change (if it is not defined the changes apply only to the line appearances), to define case-insensitivity of the changes and to ask for confirmation, respectively.
- Visual mode: It allows to select a chunk of text. It is accessed from the normal mode by pressing:
  - *v* for the simple visual mode. Once here move through the file using arrows to select text.
  - *V* for the visual mode to select full rows.
  - *ctrl+v* for the visual mode to select blocks of text.

  While in the visual mode you can use normal mode commands to copy, cut, delete and paste the selected text.

1.6. Some useful terminal commands

Here we list some useful commands that will help when using a terminal:

- *pwd* - It stands for Print Working Directory and prints the current directory.
- *mkdir dir_name* - It stands for Make Directory. Creates a new folder in the current directory. The option *-p* allows to create nested folders.
- *cd dir_name* - It stands for Change Directory and is used to move between folders. Use *cd name_of_folder* to move inside a folder and *cd ..* to move to the parent directory.
- *touch file_name* - Creates a file.
- *ls* - It stands for List. Lists all the contents in the curren directory. Flag *-a* lists hidden directories. Flag *-l* gives extra information. These flags can be used together as *-la*.
- *clear* - Clears terminal.
- *mv foo bar/* - It stands for Move. Allows to move files and folders from one directory to another. In this example moves file or folder *foo* inside folder *bar*. It also allows to rename files and folders. *mv foo bar* renames file or folder *foo* to *bar*. To use recursively, to make the command also move directories and files inside a folder, use flag *-r*.
- *cp foo bar* - It stands for Copy. It allows to copy a file or folder. In this case it makes a copy of *foo* in *bar*. To use recursively use flag *-r*.
- *rm name_of_file* - It stands for Remove. Deletes **PERMANENTLY** a file or empty folder. To use recursively use flag *-r*. To force use flag *-f*.
- *lscpu* - Shows information of the CPU.
- *du* - It stands for disk usage. Returns the size of files and folders in the current directory.
- *cat name_of_file* - It stands for Concatenate. It prints the content of a file to *stdout*.
- *man name_of_command* - It stands for Manual. It prints the user manual for a command.
- *echo* prints a text or the content of a variable. E. g. *echo $PATH*, *echo $SLURM_JOB_NUM_NODES*.

Exercise 1 — SSH

1 Connect to the cluster using one of the shown methods and change your password.

2 Where are we connected after introducing the SSH command? To a compute node or the login node?

The login node.

3 Do you know what command checks how many users are connected?

w or *who*.

4 Run the command *lscpu*. What does it return?

It returns useful information about the CPU, such as the architecture, the model, clock frequency or cache memory.

5 Do the following:

- Create a folder in your local computer with a file. Leave it empty.
- Upload it to the cluster using *scp* commands or with a graphical tool.
- Add some text using VIM. Try some of the describe modes and commands.
- Download the folder to your local computer using *scp* commands or with a graphical tool.

1.7. Working with a cluster

As you have already seen in the subject, Operating Systems, a process is the active entity that represents a program in execution, and thread is a basic unit of CPU utilization, it comprises a thread ID, a program counter, a register set, and a stack and is a lightweight version of a process. A process can have several threads. In the case of a quad-core we can have a single process with 4 threads and each thread can be using a single core independently. These concepts are very important for this course and will be used through the development of all the assignments.

A cluster is shared among many users. In our case, the cluster is shared by around 130 users. The resources are limited and, therefore, we need a way to isolate them for a certain amount of time in case users request them.

1.7.1. Batch Jobs - SLURM

We will use a **queueing system** that will allow us to declare, mainly, the instructions that we want to run, the amount of resources that we want to use and the execution time. We specify all the instructions in what is called a **job** or a **batch file**. To specify what we are going to do we declare the instructions or directives in the batch file, and then we submit our jobs to the queueing system. For example, we could create a batch file *example.sh* and write into it the following:

```
#!/bin/bash
```

```
#SBATCH --output=example_%j.out
#SBATCH --error=example_%j.err
#SBATCH --time=00:00:05

hostname
./main
```

This declared job configuration sets a maximum running time of 5 seconds and it will print the host name and execute a binary with the path "./main". The errors in the execution will be printed in the *.err* file, while the standard output will be printed in the *.out* file.

The name of the queueing system that we will use is SLURM (Simple Linux Utility for Resources Management). It uses #SBATCH lines as structured comments to specify parameters.

The *sbatch* syntax is the following:

```
#SBATCH --directive=value
```

Some of the directives that you will need for your jobs are the following:

- To specify amount of time requested.

```
#SBATCH --time=DD-HH:MM:SS
```

- To specify the number of nodes requested.

```
#SBATCH --nodes=number
```

- To specify the number of processes to start. Note for SLURM process=task. Maximum of 1 task per core.

```
#SBATCH --ntasks=number
```

- To specify the amount of cores per process. Note for SLURM CPU=core.

```
#SBATCH --cpus-per-task=number
```

- To specify the number of processes in a node.

```
#SBATCH --tasks-per-node=number
```

For example, if we had to execute a code with two processes with 4 cores each we would edit the batch file with the following:

```
#!/bin/bash

#SBATCH --output=example_%j.out
#SBATCH --error=example_%j.err
#SBATCH --time=00:00:05
```

```
#SBATCH --ntasks=2
#SBATCH --cpus-per-task=4

export OMP_NUM_THREADS=4

mpirun -np 2 ./my_code
```

Finally, to submit a job to the queuing system:

```
$ sbatch my_job.sh
```

This gives you the ID of the submitted job, e.g. “Submitted batch job 123”.

To check the status of all the jobs in the cluster run the command:

```
$ squeue
```

To check only the status of yours run:

```
$ squeue -u USER
```

You can find more information about SLURM directives here and also in the UPF guide.

Exercise 2 — SLURM

1 Create a batch job configuration with 4 cores, 1 task and 4 threads per task. The output must be the following:

```
Number of cores: 4
Number of tasks: 1
Number of cores per task: 4
```

#!/bin/bash #SBATCH –job-name=job

#SBATCH –output=job.out

#SBATCH –error=job.err

#SBATCH –ntasks=1

#SBATCH –cpus-per-task=4

#SBATCH –time=00:00:01

echo Number of cores: $SLURM_CPUS_PER_TASK

echo Number of tasks: $SLURM_NTASKS

echo Number of cores per task: $SLURM_CPUS_PER_TASK

2 Create a batch job configuration with 4 cores, 4 task and 1 threads per task. The output must be the following:

```
Number of cores: 4
Number of tasks: 4
Number of cores per task: 1
```

#!/bin/bash #SBATCH –job-name=job

#SBATCH –output=job.out

#SBATCH –error=job.err

#SBATCH –ntasks=4

#SBATCH –cpus-per-task=1

#SBATCH –time=00:00:01

echo Number of cores: $SLURM_CPUS_ON_NODE

echo Number of tasks: $SLURM_NTASKS

echo Number of cores per task: $SLURM_CPUS_PER_TASK

3 Create a batch job configuration with 2 nodes, 4 cores per node, 2 task per node and 2 threads per task. The output must be the following:

```
Number of nodes: 2
Number of cores: 8
Number of tasks: 4
Number of cores per task: 2
```

#!/bin/bash #SBATCH –job-name=job

#SBATCH –output=job.out

#SBATCH –error=job.err

#SBATCH –nodes=2

#SBATCH –ntasks=4

#SBATCH –cpus-per-task=2

#SBATCH –time=00:00:01

echo Number of nodes: $SLURM_JOB_NUM_NODES

echo Number of cores: $(($SLURM_JOB_NUM_NODES * $SLURM_CPUS_ON_NODE))

echo Number of tasks: $SLURM_NTASKS

echo Number of cores per task: $SLURM_CPUS_PER_TASK

Hint: echo $SLURM_JOB_NUM_NODES, $SLURM_CPUS_ON_NODE, $SLURM_NTASKS and $SLURM_CPUS_PER_TASK.

1.7.2. MODULES

An environment variable is a variable set in the user session. Environment variables are used to configure and interact with the system. These can be listed running the following commands

```
$ printenv
$ env
```

In a cluster, what happens if we have users who want to use two different versions of the same software and both programs have the same name? For instance, imagine that one user wants to use the GCC version 4.7 and another one needs to use GCC version 7.1 and both binaries are named "gcc". To solve this issue we use modules. Software packages can be used by loading modules that will include the specific package that we need and will load and unload information into environment variables as PATH or LD_LIBRARY_PATH. At this point of the course we will not use it but you will do later on. The available modules in the cluster are the ones you see in the picture above and can be listed only in the compute nodes. To list them you can run the following job configuration:

```
#!/bin/bash

#SBATCH --time=00:00:10

module avail
```

This will provide a list with the available modules in the compute nodes such as this one:

To load any module from the obtained list to use in your code add in your batch files the following before the execution line:

```
$ module load name_of_module
```

To remove a loaded module use:

```
$ module purge name_of_module
```

To obtain a list of all the loaded modules:

```
$ module list
```

1.7.3. COMPILING

Compiling is the process of converting a code readable by humans to a machine code that can be executed by a computer. The compiler is the program that does the process.

```
------------------------------ /shared/easybuild/modules/all ------------------------------
   Autoconf/2.69-GCCcore-8.3.0             binutils/2.32                                (D)
   Automake/1.16.1-GCCcore-8.3.0           bzip2/1.0.8-foss-2019b
   Autotools/20180311-GCCcore-8.3.0        bzip2/1.0.8-GCCcore-8.3.0                    (D)
   Bison/3.0.4                             cuDNN/7.6.5.32-CUDA-9.0.176
   Bison/3.3.2-GCCcore-8.3.0               cuDNN/7.6.5.32-foss-2019b-CUDA-10.1.243      (D)
   Bison/3.3.2                      (D)    flex/2.6.4-GCCcore-8.3.0
   CUDA/9.0.176                            flex/2.6.4                                   (D)
   CUDA/10.1.243-foss-2019b                foss/2019b
   CUDA/10.2.89                     (D)    gettext/0.19.8.1
   EasyBuild/4.1.1                         gompi/2019b
   EasyBuild/4.1.2                  (D)    help2man/1.47.4
   FFTW/3.3.8-gompi-2019b                  help2man/1.47.8-GCCcore-8.3.0                (D)
   GCC/8.3.0                               hwloc/1.11.12-GCCcore-8.3.0
   GCCcore/8.3.0                           libffi/3.2.1-GCCcore-8.3.0
   GMP/6.1.2-GCCcore-8.3.0                 libpciaccess/0.14-GCCcore-8.3.0
   M4/1.4.17                               libreadline/8.0-GCCcore-8.3.0
   M4/1.4.18-GCCcore-8.3.0                 libtool/2.4.6-GCCcore-8.3.0
   M4/1.4.18                        (D)    libxml2/2.9.9-GCCcore-8.3.0
   OpenBLAS/0.3.7-GCC-8.3.0                ncurses/6.0
   OpenMPI/3.1.4-GCC-8.3.0                 ncurses/6.1-GCCcore-8.3.0                    (D)
   PGI/19.10-GCC-8.3.0-2.32                numactl/2.0.12-GCCcore-8.3.0
   Python/3.7.4-foss-2019b                 numpy/1.17.5-foss-2019b-Python-3.7.4
   SQLite/3.29.0-GCCcore-8.3.0             scipy/1.3.3-foss-2019b-Python-3.7.4
   ScaLAPACK/2.0.2-gompi-2019b             xorg-macros/1.19.2-GCCcore-8.3.0
   Tcl/8.6.9-GCCcore-8.3.0                 zlib/1.2.11-GCCcore-8.3.0
   XZ/5.2.4-GCCcore-8.3.0                  zlib/1.2.11                                  (D)
   binutils/2.32-GCCcore-8.3.0
```

Figure 4. Available module list.

We will use the GCC compiler.

```
gcc main.c
```

This command compiles the C code in the file *main.c* to an executable file. Since no name for the output file is specified the default will be *a.out*. Flags are used to introduce options to the compiler. Some useful flags are *-Wall*, which produces warnings in case there some parts of the code can be improved, *-o*, to name the output file or *-l* to link libraries. By default, the libraries are searched in the directories */usr/local/lib/* and */usr/lib/*, so if a code requires external libraries their paths have to be introduced with the flag *-L*, for the path to the library, and *-I* for the path to the header file, so the compiler finds and links them. For example,

```
gcc -Wall main.c -lm -o output
```

compiles the file *main.c* into a file named *output*. During compilation time warnings will be shown, if there are any. Also, the MATH library will be linked (*-lm*). The compiler searches for external functions from left to right, so if the file *main.c* requires a function located in the MATH library we have to add *-lm* after *main.c*, otherwise we will have a compilation error.

1.7.4. MAKEFILE

Compilation lines can get quite large for some codes. The *make* utility is the way to automatize compilations. It requires a *Makefile* that defines a set of tasks to be done. Inside this file we define the *recipe* to create a *target* with some *prerequisites*. The basic syntax is the following:

```
target: prerequisites
<tab> recipe
```

The target does not need to be a file, it can just be the name for the recipe. These are called *phony* targets.

In the *Makefile* we can define variables, paths, flags and everything we need for a full compilation. You can use this example as a base:

```
COMPILER = gcc
CFLAGS   = -Wall
LIBS     = -lm
OBJ      = main.c
TARGET   = a.out

.PHONY: all clean

all:
    $(COMPILER) $(CFLAGS) (OBJ).c $(LIBS) -o $(TARGET)
clean:
    rm $(OBJ)
```

To execute the *Makefile* the command *make* has to be run in the directory the *Makefile* is found. Running *make clean* executes the phony target *clean* which deletes the output file.

Remember to compile in the compute nodes using a SLURM job to avoid blocking the login node.

1.8. Cluster workflow

As a summary for this section here we show a list of steps that you will have to commonly follow:

1. Connect to the cluster and login.
2. Send your files to the cluster.
3. Navigate to the directory where your files are.
4. Submit a job to compile and run your code.
5. Optionally, check its status.
6. Once the job is finished transfer output data to your computer and evaluate it.
7. Use **logout** command in the cluster to finish the SSH connection.

2. Example

In this Section we will test the cluster and all the explained tools with a provided code. You can find the code in "/Seminars".

Exercise 3 — Sieve

1 You will have to configure a job, modify a *Makefile* and submit it to the cluster. The aim of our task is to have 12 different tasks to calculate different prime numbers. We will use the sieve of Erathostanes algorithm to calculate the total of prime numbers smaller than a random number. For this, copy the folder "Seminar_1" from the folder "/Seminars" to your home directory. You will find an implementation of the algorithm and an incomplete *Makefile* and job script. Modify the *Makefile* and job script to achieve the following (the program does not need any arguments to run):

- Copy the folder "Seminar_1" from the folder "/Seminars" to your home directory.
- Complete the *Makefile* so it uses the compiler GCC. Define the *TARGET* of the *Makefile*, which is the output of the compilation or the executable file.

```
COMPILER = gcc
CFLAGS = -Wall -O3
LIBS = -lm
OBJ = sieve
TARGET = a.out
.PHONY: all
all:
        $(COMPILER) $(CFLAGS) $(OBJ).c $(LIBS) -o $(TARGET)
.PHONY: clean
clean:
        rm $(OBJ)
```

- Complete the job file to have 1 task computing prime numbers and a maximum of 10 seconds. Remember to add the execution line to the job file. No arguments are required to execute the code.

```
#!/bin/bash
#SBATCH –job-name=sieve
#SBATCH –output=sieve_%j.out
#SBATCH –error=sieve_%j.err
#SBATCH –cpus-per-task=1
#SBATCH –ntasks=1
#SBATCH –nodes=1
#SBATCH –time=00:00:10

make || exit 1 # Exit if make fails
./a.out
```

- Submit the job and take a look at the output files.
- Modify the job file to run 12 tasks computing prime numbers. Submit the new job. The output should be similar to this.

```
gcc -Wall -O3 sieve.c -lm -o a.out
PID 31787: Calculating primes for 3595
PID 31787: There are 503 primes smaller than 3595
PID 31788: Calculating primes for 3143
PID 31788: There are 446 primes smaller than 3143
PID 31792: Calculating primes for 6079
PID 31792: There are 793 primes smaller than 6079
PID 31793: Calculating primes for 4084
PID 31793: There are 562 primes smaller than 4084
PID 31796: Calculating primes for 2132
PID 31796: There are 321 primes smaller than 2132
PID 31786: Calculating primes for 9632
PID 31786: There are 1190 primes smaller than 9632
PID 31794: Calculating primes for 6628
PID 31794: There are 855 primes smaller than 6628
PID 31797: Calculating primes for 5421
PID 31797: There are 716 primes smaller than 5421
PID 31795: Calculating primes for 7044
PID 31795: There are 906 primes smaller than 7044
PID 31789: Calculating primes for 4084
PID 31789: There are 562 primes smaller than 4084
PID 31790: Calculating primes for 7727
PID 31790: There are 981 primes smaller than 7727
PID 31791: Calculating primes for 511
PID 31791: There are 97 primes smaller than 511
```

```
#!/bin/bash
#SBATCH –job-name=sieve
#SBATCH –output=sieve_%j.out
#SBATCH –error=sieve_%j.err
#SBATCH –cpus-per-task=1
#SBATCH –ntasks=12
#SBATCH –nodes=1
#SBATCH –time=00:00:10

make || exit 1 # Exit if make fails
srun ./a.out
```

Do we need several CPUs per task?

No, we only need 1 CPU per task and 12 tasks.

- What does *srun* do? How is it useful for our problem?

  It distributes the code among the available resources. If we do not use it then only one instance of sieve will be executed. With srun there will be 12 processes running.