HPC Laboratory Assignment
Lab 1: Parallel Execution Environment

J.R Herrero and M.A. Senar

Spring 2023

Contents

Session 1

Experimental setup

The objective of this first session is to familiarise yourself with the hardware and software environment that you will be using during the semester to do all laboratory assignments in HPC. From your PC/terminal booted with Linux[1] you will access `boada`, a multiprocessor server located at the Computer Architecture Department at UPC. To connect to it you will have to establish a connection using the secure shell command: `"ssh -X hpc1XYZ@boada.ac.upc.edu"`, being `hpc1XYZ` the username assigned to you. Option `-X` is necessary in order to forward the X11 data and be able to open remote windows in your local desktop[2]. You should change the password for your account using `"ssh -t hpc1XYZ@boada.ac.upc.edu passwd"`[3].

Once you are logged in you will find yourself in any of the interactive nodes:`boada-6` to `boada-8`, where you can execute interactive jobs and from where you can submit execution jobs to the rest of the nodes in the machine. In fact, `boada` is composed of several nodes (named `boada-1` to `boada-15`), equipped with five different processor generations, as shown in the following table:

Node name	Processor generation	Interactive	Partition
`boada-1` to 4	Intel Xeon E5645	No	execution2
`boada-6` to 8	Intel Xeon E5-2609 v4	Yes	interactive
`boada-9`	Intel Xeon E5-1620 v4 + Nvidia K40c	No	cuda9
`boada-10`	Intel Xeon Silver 4314 + 4 x Nvidia GeForce RTX 3080	No	cuda
`boada-11` to 14	Intel Xeon Silver 4210R	No	execution
`boada-15`	Intel Xeon Silver 4210R + ASUS AI CRL-G116U-P3DF	No	iacard

However, in this course you are going to use only nodes `boada-6` to `boada-8` interactively and nodes `boada-11` to `boada-14` through the `execution` queue, as explained in the next subsection. The rest of the nodes have restricted access and HPC users are not allowed to send jobs to their corresponding queues.

All nodes have access to a shared NAS (*Network-Attached Storage*) disk; you can access it through `/scratch/nas/1/hpcXXYY` (in fact this is your *home directory*, check by typing `pwd` in the command line). In addition, each node in `boada` has its own local disk which can be used to store temporary files non visible to other nodes; you can access it through `/scratch/1/hpcXXYY`.

We will post all necessary files to do each laboratory assignment in `/scratch/nas/1/hpc0/sessions`. For the session today, you need to extract the files from file `lab1.tar.gz` **from** that location by uncompress it **into** your home directory in `boada`.

From the the root of your home directory unpack the files with the following command line: `"tar -zxvf /scratch/nas/1/hpc0/sessions/lab1.tar.gz"`.

[1] You can also access from your laptop, booted with Linux, Windows or MacOS X, if a secure shell client is installed.

[2] Option `-Y` if you are connecting from a MacOS X laptop with XQuartz. With Windows, if you use putty (https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html) then you will also need xming: (https://wiki.centos.org/HowTos/Xming). Alternatively, you can use MobaXterm (https://mobaxterm.mobatek.net/download.html) which has everything integrated.

[3] The `passwd` command will be executed in `boada`. After entering the old password correctly twice you will be asked for your new password also twice.

In order to set up all environment variables you have to process the `environment.bash` file now available in your home directory with `"source ~/environment.bash"`. **Note:** since you have to do this every time you login in the account or open a new console window, we recommend that you add this command line in the `.bashrc` file in your home directory, a file that is executed every time a new session is initiated. If you have unpacked the lab1.tar.gz files in your home directory then this file has already been added automatically to your home directory and it will be used from your next connection to `boada`.

In case you need to transfer files from `boada` to your local machine (laptop or desktop in laboratory room), or viceversa, you have to use the secure copy `scp` command. For example if you type the following command `"scp hpc1XYZ@boada.ac.upc.edu:lab1/pi/pi_seq.c ."` in your local machine you will be copying the source file `pi_seq.c` located in directory `lab1/pi` of your home directory in `boada` to the current directory, represented with the `"."`, in the local machine, with the same name.

1.1 Node architecture and memory

The first thing you will do is to investigate the architecture of the available nodes in `boada`. Run `sbatch submit-arch.sh` command in directory `lab1/arch`. This command will enqueue `submit-arch.sh` script. More detail about *queueing* executions below. This script will execute the `lscpu` and `lstopo` commands in order to obtain information about the hardware in one of the nodes of `execution` queue (`boada-11 to 14`). The execution of these two commands using `submit-arch.sh` script generates three files, where `number` may be $11, 12, 13$ or 14: 1) `lscpu-boada-number`, 2) `lstopo-boada-number`, and `map-boada-number.fig`. You can use the `xfig` command to visualise the output file generated (`map-boada-number.fig`) and export to a different format (PDF or JPG, for example) using `File` $\rightarrow$ `Export` in order to include it in your deliverable for this laboratory assignment[4]. Those files will help you to figure out:

- the number of sockets, cores per socket and threads per core in a specific node;
- the amount of main memory in a specific node, and each NUMAnode;
- the cache memory hierarchy (L1, L2 and L3), private or shared to each core/socket.

Hint: Observe which is the number of `Thread(s) per core` returned by the `lscpu` command. If the number is 2 this means that the hardware in each core keeps the context of two threads of execution and can pick up instructions from any of the 2 threads, though not simultaneously. This is called `Hyper-Threading` for Intel processors. Although `lscpu` reports having 40 CPUs (cores), in reality there are only 20. Even when `Hyper-Threading` is enabled and the system can take instructions quickly from 40 threads, there are only 20 cores available!

Fill in the table in the "Deliverable" section with the main characteristics of the node. Draw the architecture of the node based on the information generated by the tools above. the `"--of fig map.fig"` option for `lstopo` can be very useful for that purpose. then you can use the `xfig` command to visualise the output file generated (`map.fig`) and export to a different format (pdf or jpg, for example) using `file` $\rightarrow$ `export`[5].

[4] In the `boada` Linux distribution you can use `xpdf` to open pdf files and `display` to visualise graphics files. You can also use the `"fig2dev -L pdf map.fig map.pdf"` command to convert from `.fig` to `.pdf`; look for alternative output graphic languages by typing `"man fig2dev"`.

[5] in the boada linux distribution you can use `xpdf` to visualise pdf files and `display` to for other graphic formats. You can also use the `"fig2dev -l pdf map.fig map.pdf"` command to convert from `.fig` to `.pdf`; look for alternative output graphic languages by typing `"man fig2dev"`.

1.2 Execution modes: interactive vs queued

There are two ways to execute your programs in boada:

1. via a queueing system (in one of the nodes `boada-11` to `boada-14`);
2. interactively (in any of the login nodes `boada-6` to `boada-8`). In this case, the system limits the number of cores to be used in parallel executions to two.

It is mandatory to use option 1 when you want to execute scripts that require several processors in a node, ensuring that your job is executed in isolation (and therefore reporting reliable performance results) and to avoid adding additional load to the interactive node accessed by all users; the execution starts as soon as a node is available. When using option 2 your execution starts immediately but will share resources with other programs and interactive jobs, not ensuring representative timing results. Usually, scripts for both options (`submit-xxxx.sh` and `run-xxxx.sh`, respectively) will be provided:

- Queueing a job for execution: `"sbatch [-p partition] ./submit-xxxx.sh"` . Additional parameters may be specified, if needed by the script, after the script name. If you do not specify the name of the partition with `"-p partition"` your script will run on the `execution` partition by default. Use `"squeue"` to ask the system about the status of your job submission. You can use `"scancel"` followed by the job identifier to remove a job from the queueing system. Note that partition names associated to each node name are shown in the last column of the table above. After the execution in an available node associated to the specified partition, in addition to the files being generated by the script, two additional files will be created. Their name will have the script name followed by an `".e"` and an `".o"` and the job identifier. They will contain the messages sent to the `standard error` and `standard output` respectively during the execution of the job. You should check them to be sure results make sense.
- Interactive execution: `./run-xxxx.sh`. Additional parameters may be specified after the script name. Jobs interactively executed have a short time limit to be executed.

1.3 Serial compilation and execution

Next you will get familiar with the compilation and execution steps for both sequential and parallel applications. For this, we will use some codes which compute digits of number Pi. As a reference, Figure 1.1 shows a python code which computes the area under one quadrant of the unit circle, i.e. a circle of radius equal to 1, defined by the curve $y = \sqrt{1 - x^2}$.
The execution time for this code on `boada-1` took 522.1 seconds to compute the estimate 3.14159266367805 (actual value is 3.141592653589793). As you can see, it took a long time while it only commputed 7 correct digits of Pi. We aim at estimating digits of Pi much faster. For that we are going to use some C codes which we will parallelize. You are going to use a very simple code, `pi_seq.c`, which you can find inside the `lab1/OpenMP/pi` directory. `pi_seq.c` performs the computation of the Pi number by computing the integral of the equation in Figure 1.2. This time the algorithm is using the derivative of the arctangent function of x, which is equal to 1 divided by $(1 + x^2)$ which is another way to estimate Pi. Again, the equation can be solved by computing the area defined by the function, which at its turn can be approximated by dividing the area into small rectangles and adding up its area. This is one of the many ways to estimate digits of number Pi. Both algorithms above multiply by 4 because the functions compute $\pi/4$. Figure 1.3 shows a simplified version of the code you have in `pi_seq.c`. Variable `num_steps` defines the number of rectangles, whose area is computed in each iteration of the `i` loop.

Figure 1.4 shows the compilation and execution flow for a sequential program. We will always compile programs to generate binary executable files through a `Makefile`, with multiple targets that specify the rules to compile each program version. In this course we are going to use `gcc` (the C front–end from the *GNU Compiler Collection*) or `icc` (the C front–end from the *Intel Compilers* collection)[6]. You can type `"gcc -v"` or `"icc -v"` to know about which specific version of the compiler you are using. And define variable `CC` in the `Makefile` to use one or another.

[6]Note that `icc` however is not found by default and will only become available if the environment variable PATH is properly defined. When necessary, we will add it to our environment configuration file. Only then it'll become available once `"source ~/environment.bash"` is done, either manually, or automatically at login if it is specified in file `~/.bashrc`.

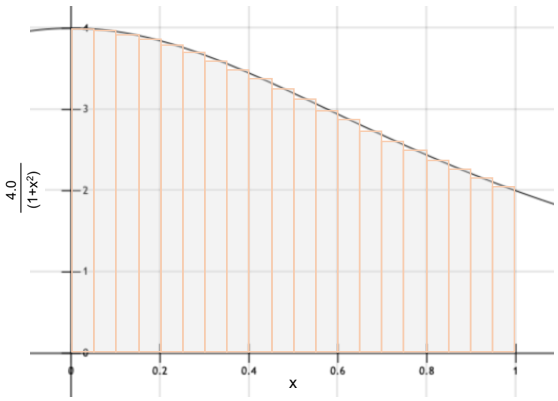
```
def midpoint_riemann_sum():
    N = 1000000000

    delta_x = (1 - 0) / N
    x = 0
    pi = 0
    start = time.time()

    while x < (1-delta_x):
        x_next = x + delta_x
        x_mid = (x_next + x) / 2
        f_x = math.sqrt(1 - math.pow(x_mid, 2))
        pi += f_x * delta_x
        x += delta_x

    pi = 4 * pi
    end = time.time()
    print(end - start)
    print("Estimate: " + str(pi))
    print("Actual:   " + str(math.pi))
```

Figure 1.1: Python code for computing Pi



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Figure 1.2: One way to compute Pi

```
static long num_steps = 100000;
void main () {
    double x, pi, step, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (long int i=0; i<num_steps; ++i) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

Figure 1.3: Serial code for Pi

In the following steps you will compile `pi_seq.c` using the `Makefile` and execute it interactively and through the queueing system, with the appropriate timing commands to measure its execution time:

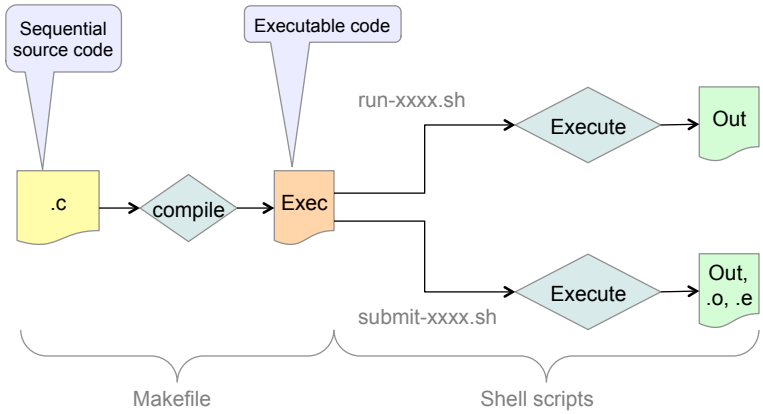


Figure 1.4: Compilation and execution flow for sequential program.

1. Open the `Makefile` file, identify the `target` you have to use to compile the sequential code. Observe how the compiler is invoked. Execute the command line `"make target_identified"` in order to generate the binary executable file. Alternatively, you can just type `"make"` which will compile all programs.

2. Interactively execute the binary generated to compute number pi sequentially by doing 1.000.000.000 iterations using the `run-seq.sh` script (`"run-seq.sh pi_seq 1000000000"`) which returns the user and system CPU time, the `elapsed` time, also called `wall` time, and the % of CPU used (using the GNU time program in `/usr/bin/time`). In addition, the program itself also reports the elapsed execution time using `gettimeofday`. Look at the source code and identify the function invocations and data structures required to measure execution time. Note that CPU time is the quantity of processor time taken by the process. This does not indicate duration. Elapsed time represents the total duration of the execution. If we exploit parallelism, elapsed time can be less than the total CPU time because several cores are used to do the work and it can get done faster.

3. Submit the execution to the queueing system using the `"sbatch submit-seq.sh"` command and use `"squeue"` to see that your script is running. Look at `submit-seq.sh` script and the results generated (the standard output and error of the script and the `time-pi_seq-boada{11-14}` file).

1.4 Compilation and execution of OpenMP programs

In this section we are going to use `OpenMP`, the standard for parallel programming using shared-memory, to express parallelism in the C programming language. Note that `OpenMP` will be explained in detail in the second part of the course, and the goal now is to have a first contact and observe that we can speed-up the execution of our programs. Therefore, in this section we will see how to compile and execute parallel programs in `OpenMP` and we will observe the performance improvements. Figure 1.5 shows the compilation and execution flow for an OpenMP program. The main difference with the flow shown in Figure 1.4 is that now the `Makefile` will include the appropriate compilation flag to enable `OpenMP`.

1.4.1 Compiling OpenMP programs

1. In the same `lab1/OpenMP/pi` directory you will find an `OpenMP` version of the code for doing the computation of pi in parallel (`pi_omp.c`). Compile the `OpenMP` code using the appropriate target in the `Makefile`). What is the compiler telling you? Is the compiler issuing a warning or an error message? Is the compiler generating an executable file? The compilation run smoothly because we used flag `-fopenmp`, but if you remove the `-fopenmp` flag and force the recompilation you will see the difference.

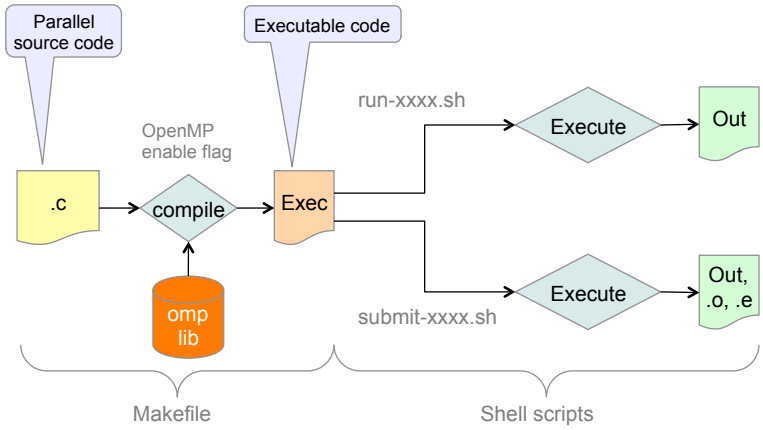


Figure 1.5: Compilation and execution flow for OpenMP.

1.4.2 Executing OpenMP programs

1. Interactively execute the `OpenMP` code with 8 threads (processors) and same number of iterations (1.000.000.000) using the `run-omp.sh` script ("`run-omp.sh pi_omp 1000000000 8`"). What is the `time` command telling you about the user and system CPU time, the elapsed time, and the % of CPU used? Take a look at the script to discover how do we specify the number of threads to use in `OpenMP`. Note that, since any login node `boada-[678]` is shared by many users and is reserved for interactive usage, it has been configured so that our processes only have 2 cores available for their execution. Thus, we will see no improvement due to the parallelization.

2. Use `submit-omp.sh` script to queue the execution of the `OpenMP` code ("`sbatch submit-omp.sh`") and measure the CPU time, elapsed time and % of CPU when executing the `OpenMP` program using 8 threads in isolation. Do you see a major difference between interactive and queued execution?

1.4.3 Strong vs. weak scalability

Finally in this section you are going to explore the scalability of the `pi_omp.c` code when varying the number of threads used to execute the parallel code. To evaluate the scalability the ratio between the sequential and the parallel execution times will be computed. Two different scenarios will be considered: *strong* and *weak* scalability.

- In *strong* scalability the number of threads is changed with a fixed problem size. In this case parallelism is used to reduce the execution time of your program.
- In *weak* scalability the problem size is proportional to the number of threads. In this case parallelism is used to increase the problem size for which your program is executed.

We provide you with two scripts, `submit-strong-omp.sh` and `submit-weak-omp.sh`, which should be submitted to the queueing system with the `sbatch` command. The scripts execute the parallel code using from 1 (`np_NMIN`) to 20 (`np_NMAX`) threads. The problem size for strong scalability is 1.000.000.000 iterations; for weak scalability, the initial problem size is 100.000.000 which grows proportionally with the number of threads. The execution **will take some time** because several executions are done for each test (in order to get a minimum time), please be patient! And, very important, do NOT execute interactively but, instead, **submit them for execution in the `execution` queue**. As a result each script generates several files, including a plot (a file with a name ending in `.ps` in Postscript format) showing the resulting parallel execution time and speed–up or parallel efficiency. Visualise[7] the plots generated and reason about how the speed–up changes with the number of threads in the two scenarios.

Optional: Change the value for `np_NMAX` in the `submit-strong-omp.sh` from 20 to 40 and execute again. Can you explain the behaviour observed in the scalability plot? Even when `Hyper-Threading` is enabled and the system can take instructions quickly from 40 threads, there are only 20 cores available!

[7] In the boada Linux distribution you can use the ghostscript `gs` command to visualise Postscript files or convert the files to PDF using the `ps2pdf` command and use `xpdf` to visualise PDF files.

1.5 Programming, Compiling and Executing MPI programs

1.5.1 Learning MPI

In this section we are going to use MPI, the standard for parallel programming using distributed-memory systems, to express parallelism in the C programming language. Contrary to the previous section, in this part of the course we need to understand how to develop a code using MPI. We introduce MPI in the videos available at Aula-ESCI. Please, watch them first and do the associated questionnaires. Next, you will find a set of examples in directory `lab1/MPI/Tutorial`. Use them to test different functionalities offered by MPI. They come from `https://www.jics.utk.edu/mpi-tutorial` at the University of Tennessee. You can find explanations there. In most subdirectories there is a file with name ended in `.c.soln` with the solution so that you can check your solution against a good one. You can disregard anything related to PBS, which is another queueing system different from SLURM[8], the one used at `boada`. Note that the execution of a parallel code can change from execution to execution. Consequently, the output being printed can reflect different from one execution to another, and can overlap messages written from different MPI processes. Sometimes it's necessary to execute several times in order to understand the behaviour. In summary, please whatch the videos on MPI and try to understand the code and behaviour of the programs in the Tutorial, at least for the following programs (and number of MPI processes within parenthesis): `hello1 (4), hello2 (4), pical (4), pingpong (2), deadlock (2), deadlockf (2), collectives (4)`. Remember that you have to compile the program typing `make` within the corresping directory; and execute them with `mpirun -np P MPIEXECFILE`, where P stands for the number of MPI processes that we want to create, and MPIEXECFILE has to be replaced with the name of the MPI executable file generated after the compilation. Also, it's important to use the on-line manual to understand the MPI calls. For instance, when studying the `deadlock.c` file you might be interested in reading about some new MPI primitive (`man MPI_Ssend`). Additionally, please inspect the document about *MPI's send modes* at `Aula-ESCI`. In the report:

- Explain the *deadlock* problem and how it can be fixed;
- Show the code excerpt related to the *gather* and *scatter* collective operations. Also, show the output of their execution, commenting it.

1.5.2 Strong scalability with processes and threads

Next, we will test ourselves how to compile and execute parallel programs in MPI, or a combination of both MPI and OpenMP. To do so, follow similar same steps as for OpenMP in the previous section, but working with the files in directory `lab1/MPI/pi`. The code calculates number Pi with a different algorithm based on the so called Monte Carlo method. Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The idea is to simulate random (x, y) points in a 2-D plane with domain as a square of side 1 unit. Imagine a circle inside the same domain with same diameter and inscribed into the square. We then calculate the ratio of number of points that lied inside the circle and total number of generated points. To higher the number of samples, the better the precision obtained.

We can get a plot of the execution time and speed–up that is obtained when varying the number of MPI processes (strong scalability) by submitting a job to the `execution` queue with `sbatch submit-strong-mpi.sh`). In addition, we want to observe the combination of MPI and OpenMP. For that execute `sbatch submit-mpi2-omp.sh`) and study the results presented in the output file created after the execution finishes. Note that since the test uses 2 MPI processes the execution implies using an even number of threads between 2 and 40 even when the output presents the results for 1 to 20 threads: there are between 1 to 20 threads per MPI process; with 2 processes this results in having twice as many threads.

We hope that, with this experiments, you realize that we can speed up the code by parallelizing a code with OpenMP, with MPI, or a combination of both to use several nodes and several cores per node!

[8] You will see SLURM in the 2^{nd} part of this course.

1.5.3 Write your own `MPI` code

Finally, test your understanding by creating an `MPI` program of your own which includes calls to several `MPI` primitives. Here's your opportunity to be creative! It's not necessary that the code is really useful, but the code should include some message exchanges with both 1) point to point primitives and 2) at least one collective communication primitive.

For the latter, notice that all processes have to call a collective primitive in the very same way in the program; and the way by which we can specify which of the processes has to behave differently is through one of the parameters. Oftentimes such process is called the *master* or *root* process in the literature.

In the introductory video and in multiple Makefiles within the `MPI` subtree of directories provided within lab1 you can see how to compile C programs extended with `MPI` with `mpicc`. But, if you prefer to code the `MPI` program of your own using C++ you can compile it with `mpic++`. If you prefer Fortran then you can use `mpif77`, `mpiF90` or `mpifort` depending on the flavor of Fortran.

For instance, if my `MPI` program is `hello2.c`, we can compile with "`mpicc -o hello2 hello2.c`". Since the code you will produce is likely to run in a very short time you can execute it interactively with `mpirun`. For instance, we can run it with 4 processes with "`mpirun -np 4 hello2`".

Note: if your code involves a lot of CPU time then you have to create a shell script and launch it to execution in the queuing system with the `sbatch` command. If that was the case, you can get inspiration from the multiple `submit-xxxx.sh` files within `lab1`. However, this is NOT the goal and we neither expect you to develop a complex code here nor one that implies a high computational cost.

Please, test your code running it with several processes. **In your report**, you should show the code and explain briefly your implementation and its functionality, clarifying what your program is supposed to do, and how to compile and run it, possibly complemented with the output that it produces.

Deliverable

After the last session for this laboratory assignment, and before starting the next one, you will have to deliver two files:

1. A report in `PDF` format (other formats will not be accepted) containing the answers to the questions stated at the end of this document;
2. For question 5, you also have to hand in the source file of your C code extended with MPI.

Your professor will open the assignment at the moodle aula ESCI website and set the appropriate delivery dates for the delivery. **Individual submission** through the moodle aula ESCI website.

Important: In the front cover of the document, please clearly state your name and surname, the identifier of the group (username `hpc1XYZ`), title of the assignment, date, academic course/semester, ... and any other information you consider necessary.

As part of the document, you can include any code fragment, figure or plot you need to support your explanations. In case you need to transfer files from boada to your local machine (laptop or desktop in laboratory room), or viceversa, you can use the secure copy `scp` command. For example `"scp hpc1XYZ@boada.ac.upc.edu:lab1/foobar.txt local/directory/."` executed `locally in your machine` to copy file `foo.txt` **from** inside directory `lab1` in your home directory of `boada` (the remote system) **to** directory `local/directory/` in your local machine with the same name.

Node architecture and memory

1. Complete the following table with the relevant architectural characteristics of the different node types available in `boada`:

	`boada-11` **to** `boada-14`
Number of sockets per node	
Number of cores per socket	
Number of threads per core	
Maximum core frequency	
L1-I cache size (per-core)	
L1-D cache size (per-core)	
L2 cache size (per-core)	
Last-level cache size (per-socket)	
Main memory size (per socket)	
Main memory size (per node)	

2. Include in the document the architectural diagram for one of the nodes `boada-11` to `boada-14`.

Timing sequential and parallel executions

For each of the following sections show the plots or tables required. But, in addition, please **provide some reasoning about the results obtained**. For instance, comment whether the scalability is good or not, and why.

3. Plot the execution time and speed–up that is obtained when varying the number of threads (*strong scalability*) by submitting the jobs to the `execution` queue (section 1.4.3). It you did the `optional part`, show the resulting plot and comment the reason for this behaviour. Show the parallel efficiency obtained when running the *weak scaling* test. Explain what strong and weak scalability refer to, exemplifying your explanation with the plots that you present.

4. Plot the execution time and speed–up that is obtained when varying the number of `MPI` processes from 1 to 20 (strong scalability) by submitting the jobs to the `execution` queue (section 1.5.2). In addition, show in a table the elapsed execution time when executed with 2 `MPI` processes when varying the number of threads from 1 to 20 [9]. Which was the original sequential time?; the time with 20 `MPI` processes?; and the time with 2 `MPI` processes each using 20 `OpenMP` threads? You can retrieve such information for 1 and 20 `MPI` processes from file `elapsed.txt`; and for 2 `MPI` processes each with 20 threads within the output file created after the execution of `sbatch submit-mpi2-omp.sh`.

`MPI`

Understanding `MPI` codes

- Explain the *deadlock* problem and how it can be fixed;
- Show the code excerpt related to the `MPI_Gather` and `MPI_Scatter` collective operations. Also, show the output of their execution in program `collectives` (only the output of these two operations, not the whole output of the program), explaining it briefly.

Write your own `MPI` code

5. Create an `MPI` program of your own which includes calls to several `MPI` primitives. Here's your opportunity to be creative! The code should include some message exchanges with both 1) point to point primitives and 2) at least one collective communication primitive. Test it with several processes. In your report, you should show the code and explain briefly your implementation and its functionality.

[9] Note that for 2 `MPI` processes the scalability test implies using an even number of threads between 2 and 40.