CAP-GIA (FIB) Laboratory Assignment

Lab 6: Getting started with a supercomputing environment

Josep Lluís Berral and Jordi Torres

Fall 2024

1. Hands-on exercise description

Supercomputers represent the leading edge in high performance computer technology, a crucial component in the progress of artificial intelligence. This laboratory assignment provides an approach to the practical environment in which the laboratory part of this course will be developed, giving the student hands-on experience to discover some of the basic building blocks of a supercomputer and its system software stack.

At the end of this hands-on session, the student will be familiar with the execution and storage environment and proficient in using the SLURM batch scheduling system, a crucial element for obtaining resources on the Marenostrum supercomputer.

Important note: Beware of "copy & paste" as symbols such as commas or dashes can be incorrectly copied from this document to the console. If a command/code doesn't work correctly, write it directly!

Content

Laboratory practice instructions

- For this laboratory assignment, we are distributed in pairs (groups of 2 people). It is recommended that both group members always use the same account for all the lab sessions.
- Each hands-on may include theoretical content to supplement the material covered in the lectures, guiding students in completing the tasks into which the practical is divided. It is recommended that all students have read the instructions for the practical before the corresponding laboratory session.
- In laboratory assignments, we will find tasks that must be done in strict order. While waiting for the results of the executions, we can move on to the next one. It is recommended to take digital notes of intermediate steps during execution, as well as the results of each task.
- Once the hands-on is completed, each group must submit their work to the corresponding inbox on the 'Racó de la FIB' intranet. Delivery to the RACO@FIB intranet box will only be **made by one of the group's two members.**
- The submission must include a **report** that details the results of each task, as these will be used to present the overall results of the hands-on activity. As previously noted, one group, selected at random, will be chosen to present the results of their tasks to the class. For each submission to the 'Racó de la FIB', the .zip file must include not only the report **(in PDF** format**)** but also the relevant code files (.c, .sh, .txt, etc.) and any required outputs as specified in the hands-on. The report should contain a cover clearly stating the number of the practical, the name of the group members, the e-mail of the group members, and the date.
- You may consult any internet source or use generative AI tools to complement or expand the information related to this hands-on. However, it is important to remember that AI assistants themselves, like ChatGPT, warn that they can make mistakes, and users should verify critical information. If the results of the hands-on assignment contain errors due to not thoroughly reviewing the output of these tools when used, it **will be critical** and will significantly reduce the likelihood of passing the evaluation of the hands-on.

2. Marenostrum 5 supercomputer

MareNostrum 5 is a pre-exascale EuroHPC supercomputer hosted at BSC-CNS. The system is supplied by Bull SAS, combining Bull Sequana XH3000 and Lenovo ThinkSystem architectures, and it has a total peak computational power of 314PFlops. It consists of two computer partitions with the shared file system:

- MARENOSTRUM 5 GPP: General Purpose partition based on Intel Sapphire Rapids, 6408 nodes + 72 HBM nodes (45 PFlops peak)
- MARENOSTRUM 5 ACC: Accelerated Partition based on Intel Sapphire Rapids and Nvidia Hopper GPUs, 1120 nodes with 4 Hopper GPUs each one (230 PFlops peak)

Your BSC credentials as a student will allow you to access both partitions, but you need to log in to different login nodes. You can access both BSC supercomputing resources from your local terminal using the secure shell (ssh).

Once you are logged in either MN5 partition, you will find yourself in an interactive node where you can execute interactive jobs (small jobs) and submit execution jobs to the rest of the nodes of the partition chosen using the SLURM batch scheduling system.

The Marenostrum file system is based on the IBM General Parallel File System (GPFS), a high-performance shared-disk file system providing fast, reliable data access from all cluster nodes to a global filesystem. GPFS allows parallel applications simultaneous access to a set of files (even a single file) from any node with the GPFS file system mounted while providing high control over all file system operations.

From your MN5 account, you can use the text editors Vim or Nano. If you are new to text editing and need something easy to use now, Nano is a good choice. If you have experience with Vim, it's better. Both use a command line interface and are available on Marenostrum nodes.[1]

[1] If you want to mount the supercomputer's file system on your local computer and use your local editor, you can do so. However, you must do it on your own, without teacher support, as it is not strictly necessary and therefore not covered in this course.

3. Software environment

Typically, users initialize their environment when they log in by setting environment information for every application they reference during the session. The *Environment Modules*[2] package is a tool that simplifies shell initialization and lets users easily modify their environment during the session with modulefiles. Each modulefile contains the information needed to configure the shell for an application or a compilation. Modules can be loaded and unloaded dynamically in a clean fashion. All popular shells are supported, including bash, ksh, zsh, sh, csh, tcsh, as well as some scripting languages such as perl.

Modules can be invoked in two ways: by name alone or by name and version. Invoking them by name implies loading the default module version. This is usually the most recent version tested to be stable (recommended) or the only version available.

```
module load intel
```

Invoking by version loads the version specified by the application. As of this writing, the previous command and the following one load the same module.

```
module load intel/2023.2.0
```

The most important commands for modules are these:

- *module list* shows all the loaded modules
- *module avail* shows all the modules the user is able to load
- *module purge* removes all the loaded modules
- *module load* <*modulename*> loads the necessary environment variables for the selected modulefile
- *module unload* <*modulename*> removes all environment changes made by module load command
- *module switch* <*oldmodule*> \<*newmodule*> unloads the first module and loads the second module
- *module show* <*modulename*>

You can run "module help" anytime to check the command's usage and options[3].

The latest Intel compilers provide the best possible optimizations for the architecture of MN5. For this reason, by default, when starting a new session on the system, the basic modules for the Intel suite will be automatically loaded.

[2] http://modules.sourceforge.net/

[3] Each Marenostrum 5 partition have dedicated directories for module files: /apps/GPP/modulefiles and /apps/ACC/modulefiles respectively.

Task 1

This Task covers the primary usage of the module environment. Do the following activities:

1. **Check the loaded modules in your environment now using the correct command for modules. Describe the modules that the system has loaded and made available**
2. **Unload all modules**
3. **Load module intel/2023.2.0**
4. **Discover information about the specific Intel module version 2023.2.0, including details on environment variables to be set, changes to the PATH, and other adjustments that will be applied when this module is loaded. Hint: "module show".**

4. Space area for file storage

GPFS filesystems

The IBM General Parallel File System (GPFS) is a high-performance shared-disk file system that provides fast, reliable data access from all cluster nodes to a global filesystem. GPFS allows parallel applications simultaneous access to a set of files (even a single file) from any node that has the GPFS file system mounted while providing high control over all file system operations. In addition, GPFS can read or write large blocks of data in a single I/O operation, thereby minimizing overhead.

These are the GPFS filesystems accessible on the machine from all nodes:

- /apps: This file system hosts applications and libraries that are pre-installed on the machine and new builds. Browse the directories to discover applications available for general use.
- /gpfs/home: This file system contains the home directories for all users, and upon login, you will automatically start in your home directory. Each user has their designated home directory to store personally developed sources and personal data. A default quota restricts the data stored in individual home directories. Running jobs directly from this filesystem is strongly discouraged. Executing your jobs in your group's /gpfs/projects or /gpfs/scratchdirectories is recommended.
- /gpfs/scratch: Each user will also be allocated a directory per group within /gpfs/scratch/<GROUP>. Its purpose is to store temporary files your jobs generate during their execution. A group-based quota will be enforced based on the allocated space.

Accessing from/to the outside Marenostrum 5

The login nodes are the only nodes accessible from external networks, and no connections from the cluster to the outside world are permitted for security reasons. All file transfers from/to the outside must be executed from your local machine and not within the cluster.

Examples:

Copying files or directories from MareNostrum 5 to an external machine:

```
mylaptop$> scp -r {username}@transfer1.bsc.es:"MN5_SOURCE_dir"
"mylaptop_DEST_dir"
```

Copying files or directories from an external machine to MareNostrum 5:

```
mylaptop$> scp -r "mylaptop_SOURCE_dir"
{username}@transfer1.bsc.es:"MN5_DEST_dir"
```

5. Serial compilation and execution in C language

C is a general-purpose programming language developed in the early 1970s and is still quite popular because it is so powerful for creating applications, operating systems, and utilities. Due to its simplicity, efficiency, and portability, C has influenced the development of many other modern programming languages. In this section, we will introduce how to use it on the Marenostrum supercomputer and review some exciting, more relevant features to this course.

C compilers

As you learned in previous courses, you always compile programs to generate binary executables in C language. In the MN5 cluster, you can find different C/C++ compilers. All invocations of the C or C++ compilers follow these suffix conventions for input files:

```
.C, .cc, .cpp, or .cxx -> C++ source file.
.c -> C source file
.i -> preprocessed C source file
.so -> shared object file
.o -> object file for ld command
.s -> assembler source file
```

Below, you can see a "Hello World" program written in C:

```
$ more hello.c
#include <stdio.h>
int main(){
    printf("Hello world!\n");
}
```

It can be compiled with

```
$ icx hello.c -o hello
```

Then, it can be executed in the login node with (the program output will be):

```
$ ./hello
Hello world!
```

Task 2

Write a Hello World program in C that prompts "Hello World from UPC", compile it, and run it in a MN5 login node. Add your code and the execution output in the lab report.

Intel icx compiler

The icx and gcc compilers are available on MN5 platform to compile C and C++ programs.

gcc, or the GNU Compiler Collection, is a versatile and widely used open-source compiler that supports a broad range of languages and platforms. It is known for its robustness, extensive language support, and portability across various operating systems.

icx is a next-generation C/C++ compiler developed by Intel as part of its suite of development tools, providing advanced optimization features and support for the latest Intel architectures (including support for the latest **instruction sets like AVX-512**). This compiler enables different levels of optimizations using flags in the compilation process.

While gcc is preferred for its general-purpose capabilities and cross-platform compatibility, icx offers performance advantages on Intel processors due to its architecture-specific optimizations.

Task 3

Add an explanation in your lab report about the " AVX-512 instruction set" present in MN5 Intel Sapphire.

Using the test.c program (GitHub available[4]), we can measure the performance of the icx compiler using different flags.

```
/* sample.c */
```

[4] https://github.com/jorditorresBCN/Marenostrum5/blob/main/test.c

```
/* Example from PATC course @ BSC*/

#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>

#define SIZE 8000
typedef double matrix[SIZE][SIZE];
matrix m1, m2;
struct timeval start_time, end_time;

static void foo (void) {
  int i,j;
  for (i = 0;i < SIZE; ++i)
    for (j = 0; j < SIZE; ++j)
       m1[j][i] = 8.0/m2[j][i];
}

void check_matrix() {

   int i, j;
   for(i=0;i<SIZE;i++) {
     for(j=0;j<SIZE;j++) {
       if(m1[i][j]!=3.2)
      printf("Check failed\n");
      exit(1);
       }
        //printf("\n");
     }
}

void print_times()
{
   int total_usecs;
   float total_time, total_flops;
   total_usecs = (end_time.tv_sec-start_time.tv_sec) * 1000000 +
         (end_time.tv_usec-start_time.tv_usec);
   printf(" %.2f mSec \n", ((float) total_usecs) / 1000.0);
   total_time = ((float) total_usecs) / 1000000.0;
}

int main () {
  int i,j;
```

```
   for(i=0;i<SIZE;i++) {
     for(j=0;j<SIZE;j++) {
      m2[i][j]=2.5;
      }
    }
   gettimeofday(&start_time, NULL);
    for (i = 0; i < 10; ++i) foo();

   gettimeofday(&end_time, NULL);
   print_times();
   check_matrix();
}
```

This program simply initializes a large matrix with a specific value, performs a computational task on the matrix and measures the time taken for this Task.

You can use the following shell script downloadable from github[5] (compilation-icx.sh):

```
icx -O0 test.c -o test
echo "icx -O0"
./test
icx -O1 test.c -o test
echo "icx -O1"
./test
icx -O2 test.c -o test
echo "icx -O2"
./test
icx -O3 test.c -o test
echo "icx -O3"
./test
icx -O3 -xhost test.c -o test
echo "icx -O3 -xhost "
./test
```

[5] https://github.com/jorditorresBCN/Marenostrum5/blob/main/compilation-icx.sh

Task 4

Discuss the comparison of the execution times for each compilation while also explaining the flags used and their exact meanings.

As we can see, compilation is a very important step that we must consider when aiming to achieve the best performance in terms of time.

C language basics

In this course, we assume the student has some basic knowledge of this language. Here, you can find a review of some essential language C features relevant to future laboratory assignments.

pointers and addresses

C language has powerful pointer capabilities, allowing direct memory manipulation and efficient data structure implementation. For this purpose, it uses the symbols "*" and "&". Using "*" in front of a variable declaration indicates that the variable is a pointer. For example:

```
int *ptr;
```

In this declaration, "ptr" is a pointer to an integer. Pointers are variables that store memory addresses, allowing the programmer to indirectly access the value at that address in memory.

The "&" symbol is used as the address-of operator. It allows the programmer to obtain the memory address of a variable in memory. For example:

```
int x = 100;
int *ptr = &x;
```

Here, "&x" returns the memory address of the variable "x", which is then stored in the pointer variable "ptr." When you use the "*" symbol before a pointer variable, it is called the "dereference" operator. It allows the program to access the value stored at the memory address held by the pointer. For example:

```
int x = 100;
int *ptr = &x;
int value = *ptr;
```

Here, "ptr" points to the memory address of x, and the "value" will be 100, the value stored at the memory address pointed by "ptr".

One-dimension array matrix to simulate two-dimension arrays in C

C programmers frequently use one-dimension arrays to simulate two-dimension arrays by performing some manual index calculations. This is because, in C language, matrices are flattened into linear memory. C language uses row-major order, where rows are stored one

after the other in memory. To show that, inspect the code in C that initializes two 4x4 matrices h_a and h_b with the value 1, performs matrix multiplication of h_a and h_b, and prints the resulting matrix h_c.

Below is the Code Listing from the Seq_MatMul.c program demonstrating a simple practical case of using a one-dimensional array to simulate two-dimensional arrays in C.

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define N 4

double h_a[N][N], h_b[N][N], h_c[N][N];

void cleanMatrices(int nrows, int ncols, double *M) {
    for (int i = 0; i < nrows; i ++) {
        for (int j = 0; j < ncols; j++){
            M[i*ncols+j] = 0;
    }
    }
}

void iniMatrices(int nrows, int ncols, double *M) {
    for (int i = 0; i < nrows; i ++) {
        for (int j = 0; j < ncols; j++){
            M[i*ncols+j] = 1;
    }
    }
}

void printMatrix(int nrows, int ncols, double *M) {
    printf("\n ******** PRINT MATRIX ******** \n");
    for (int i = 0; i < nrows; i ++) {
        for (int j = 0; j < ncols; j++){
            printf("| %f |", M[i*ncols+j]);
        }
    printf("\n");
    }
    printf("\n ******** END ******* \n");
}

int main() {
```

```
    iniMatrices(N, N, (double *) h_a);
    iniMatrices(N, N, (double *) h_b);
    cleanMatrices(N, N, (double *) h_c);
    printf("Matrix A\n");
    printMatrix(N, N, (double *) h_a);
    printf("Matrix B\n");
    printMatrix(N, N, (double *) h_b);

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
                h_c[i][j] += h_a[i][k] * h_b[k][j];
            }
        }
    }

    printMatrix(N, N, (double *) h_c);

    return 0;
}
```

Below are some explanations of the code for those who are new to programming in the C language.

- The #include directive tells the C preprocessor to insert the contents of a specified header file into the source code before the actual compilation begins.
- The for loop is a control flow statement that allows you to execute a block of code repeatedly based on a condition.
- printf is a function used for formatted output in C. It allows you to print data to the standard output (usually the console).
- The #define directive is a preprocessor directive that defines a macro. Macros are essentially constants or short code snippets that are replaced with their values during the pre-processing phase.
- A function is a block of code that performs a specific task and can be called from other parts of the program. In C, functions have a return type, a name, optional parameters (arguments), and a body enclosed in curly braces {}. When a function is called, control is transferred to that function's body, and when the function finishes its execution, control returns to the calling point. In C, when a function is declared as void as the return type, it means the function does not return any value.
- The main function is a special function in C. It serves as the entry point of a C program. When you run a C program, the execution starts from the main function, and the program continues to execute statements sequentially until it reaches the end of the

main function. The main function is a mandatory component of every C program, and its presence is required for the program to be compiled and executed.

Task 5

Write a matrix multiplication program based on the previous code that multiplies two matrices 8x8. Compile it and run it in your login node. Add your code and the execution output in the lab report.

Analyze and explain in the lab report how a matrix declared as two-dimensional can be updated using the index `[i*ncols+j]`, assuming that it is a 1D array.

---

6. Batch scheduling system

As we already introduced, the SLURM batch scheduling system is a crucial element for obtaining resources on the Marenostrum supercomputer because the interactive login nodes are used only for editing, compiling, preparing, and submitting batch executions in the Marenostrum 5 supercomputer.

Submit to queues

It is mandatory to use the batch queuing system when you want to execute scripts that require several processors in a node to avoid adding additional load to the interactive node accessed by all users.

SLURM is the utility used for batch processing support, so all jobs must be run through it. SLURM stands for "Simple Linux Utility for Resource Management." It is an open-source, highly scalable, and widely used workload manager designed for High Performance Computing clusters and supercomputers. SLURM provides a robust and efficient system for scheduling and managing jobs across many compute nodes, ensuring that computational resources are utilized optimally and fairly among users and applications.

A job is the execution unit for SLURM, and it is defined by a text script file containing a set of directives describing the job's requirements and the commands to execute. Jobs submission to the queue system must be made through the SLURM commands. To submit a job, you must use the command:

```
$> sbatch {job_script}
```

To show all the submitted jobs (the flag "-starts " is used to know the estimated time to be executed):

```
$> squeue -starts
```

To cancel a job:

```
$> scancel {job_id}
```

For more information:

```
$man sbatch
```

Several queues are present in the machines, and users may access different queues. Queues have, unlike limits regarding the number of cores and duration for the jobs. To check the limits for the queues, you can do the following:

```
$> bsc_queues
```

For GPP partition, standard queues (QoS) names and limits are as follows:

Queue	**Max. number of nodes (cores)**	**Wallclock**	***Slurm QoS name***
Debug	32 (3,584)	2h	*gp_debug*
Interactive	1 (**32**)	2h	*gp_interactive*
Training	32 (3584)	24h	*gp_training*

For ACC partition, standard queues (QoS) names and limits are as follows:

Queue	**Max. number of nodes (cores)**	**Wallclock**	***Slurm QoS name***
Debug	8 (640)	2h	*acc_debug*
Interactive	1 (**40**)	2h	*acc_interactive*
Training	4 (320)	24h	*acc_training*

Additionally, special/extra queues can be provided for extended or larger executions, subject to demonstrating the scalability and performance of the application, taking into account factors such as demand and the current workload of the machine, among other conditions.

Job directives

The text file that defines a job must contain a series of directives to inform the batch queuing system about the characteristics of the job. These directives appear as comments in this text file that represent a job script and must conform to the *sbatch* syntaxes:

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

Additionally, the job script may contain a set of commands to execute. You may find the most common directives below.

Mandatory job directives

Some directives are mandatory for any SLURM job description. Let's start by introducing these directives.

Request the queue for the job:

This parameter is mandatory when submitting jobs in the current MareNostrum 5 configuration.

```
#SBATCH --qos={QoS_queue_name}
```

For example, the queue ' gp_debug' is intended for small test executions in the GPP partition.

Sometimes, node reservations can be granted for executions where only a set of accounts can run jobs. Useful for courses as ours at UPC. You can set the reservation name where you will allocate your jobs (assuming that your account has access to that reservation):

```
#SBATCH --reservation={reservation_name}
```

Specifying the queue allows you to send jobs from any login node to any partition (ACC or GPP).

Set the limit of wall clock time

Another mandatory parameter sets a time limit for the total runtime of the job:

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

You must set it to a value greater than the real execution time for your application and smaller than the time limits granted to the user. Notice that your job will be killed after the time has passed.

Set the Unix group account

Marenostrum 5 users will have a unique username and an associated secondary group with resource allocation required to manage your projects' data and jobs. Then, when submitting a job, it is mandatory to specify this Unix group (account).

```
#SBATCH --account {account}
```

To list the currently available accounts, you can use the command line:

```
source bsc_project list
```

The system will generate specific error messages if an attempt is made to submit a job without specifying a Slurm account, time, and/or queue (QoS):

Task 6

Create a SLURM bash script file (*) to submit, with the command sbatch, the previous 'compilation-icx.sh' shells script program using SLURM. Where do you find the standard output of your execution?. Include your slurm file and explain the results in the lab report.

(*) You can start with the following file, where the first line indicates the script should be interpreted and executed using the Bash shell.

https://github.com/jorditorresBCN/Marenostrum5/blob/main/compilation-icx.slurm file

```
#!/bin/bash

#SBATCH -t <indicate the time>
#SBATCH --account <indicate the Unixgroup/Account>
#SBATCH --qos <indicate the Queue>

module purge
module load intel
./compilation-icx.sh
```

Useful job directives

It is often useful to specify the working directory of your job in the SLURM command file, which is different from the directory from which the job is launched. It is also very helpful to set the name of the file to collect the standard output (stdout) of the job and to set the name of the file to collect the standard error output (stderr) of the job. Additionally, including the unique job ID in SLURM in the filenames is very useful. Let's look at the flags to achieve this."

Set the working directory of your job:

```
#SBATCH -D pathname
```

or

```
#SBATCH --chdir=pathname
```

This queue is where the job will run. If not specified, it is the current working directory when the job was submitted.

Set the name of the file to collect the standard output (stdout) of the job:

```
#SBATCH --output=file
```

Set the name of the file to collect the standard error output (stderr) of the job:

```
#SBATCH --error=file
```

Use of unique job id in SLURM

In SLURM, the "%j" is a special parameter used in job scripts to represent the job ID.

When you submit a job to SLURM, it assigns a unique job ID to that particular job. The "%j" parameter is a placeholder that allows you to dynamically access and use this job ID within your job script. We usually use "%j" in the output and error filenames:

```
#SBATCH --output=output_%j.out
#SBATCH --error=output_%j.err
```

"%j" will be replaced by SLURM with the actual job ID assigned to your job. So, if your job is assigned the ID "123456," the output file will be named "output_123456.out." Using the job ID in this way can help track and organize job-related files, logs, or other information specific to individual job executions.

Basic job directives for reserving computing resources

Below, we briefly present the basic directives for reserving the computing resources required for batch jobs. Throughout the various hands-on sessions, these directives will be explored in more detail.

Request an exclusive use of a compute node without sharing the resources with other users:

```
#SBATCH --exclusive
```

Set the number of requested nodes:

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

Or:

```
#SBATCH -N number
```

This parameter will enforce the exclusivity of the nodes in case you request more than one.

Set the number of processes to start:

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

Set the number of threads each process would open:

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

The number of cores assigned to the job will be the ntasks number * cpus_per_task number.

Set the number of tasks assigned to a node:

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

Set the number of tasks assigned to a socket:

```
#SBATCH --ntasks-per-socket=number
```

sbatch examples

Below, you can find three basic examples of SLURM script files.

1) Example for a sequential job:

```
#!/bin/bash
#SBATCH --job-name="serial_job"
#SBATCH --chdir=.
#SBATCH --output=serial_%j.out
#SBATCH --error=serial_%j.err
#SBATCH --ntasks=1
#SBATCH --account <indicate the Unixgroup/Account>
#SBATCH --qos <indicate the Queue>
#SBATCH --time=00:02:00

./serial_binary
```

This SLURM script defines a job named "serial_job" that runs a binary named "serial_binary" in the current directory. The job will be executed on a single core and is limited to a maximum runtime of 2 minutes. The standard output and error messages from the job will be saved in separate files with unique names based on the job ID. The first line indicates the script should be interpreted and executed using the Bash shell.

2) Examples of a parallel job running a pure OpenMP job on one MN4 node using 48 cores:

```
#!/bin/bash
#SBATCH --job-name=omp
#SBATCH --chdir=.
#SBATCH --output=omp_%j.out
#SBATCH --error=omp_%j.err
#SBATCH --cpus-per-task=48
#SBATCH --ntasks=1
#SBATCH --account <indicate the Unixgroup/Account>
#SBATCH --qos <indicate the Queue>
#SBATCH --time=00:10:00

./openmp_binary
```

This SLURM script defines a job named "omp" that runs an OpenMP parallel program named "openmp_binary" in the current directory. The job will be executed with a time limit of 10 minutes and will use 48 CPUs for the computation.

3) Example of running on two MN5 nodes using a pure MPI job:

```
#!/bin/bash
#SBATCH --job-name=mpi
#SBATCH --output=mpi_%j.out
#SBATCH --error=mpi_%j.err
#SBATCH --ntasks=96
```

```
#SBATCH --account <indicate the Unixgroup/Account>
#SBATCH --qos <indicate the Queue>
#SBATCH --time=00:10:00


./mpi_binary
```

This SLURM script defines a job named "mpi" that runs an MPI parallel program named "mpi_binary." The job will be executed with 96 tasks.

Task 7

As a warm-up using SLURM, create a new bash script to execute compilation-icx.sh **according to the following suggestions to execute :**

- **The name of the job is HelloWorldCompilation**
- **Start one process**
- **Max time to be executed 5 min.**
- **The name for collecting the standard output is output_%J.out The name to collect the standard error is error_%J.err**
- **Use the node exclusively.**

Once this script is done, it has to be submitted using the command sbatch. Include the script, standard output file, standard error file, and your comments in the lab report.

More details about SLURM in Marenostrum 5 can be found on the support system pages: https://www.bsc.es/supportkc/docs/MareNostrum5/slurm

Welcome to the Marenostrum 5

7. (Optional) Brief about authentication by SSH public keys

Here are the steps to connect to your Marenostrum account using SSH without typing the password every time. Assuming you have OpenSSH installed on your Linux laptop, follow these commands:

1. Generate SSH keys on your laptop:

- Open a terminal on your Linux laptop.
- Run the following command to generate your SSH keys:

```
ssh-keygen -t rsa
```

- It will prompt you to specify the location and name of the key file. You can leave the default values by pressing "Enter". You can also add a passphrase to protect your key, but in this case, we will not use one to avoid typing it every time.

2. Copy the public key to your remote account:

- Run the following command to copy your public key to the remote server:

```
ssh-copy-id nctxxxxx@glogin1.bsc.es
```

- Replace "`nctxxxxx`" with your username on the Marenostrum server.
- It will ask for your password to authenticate the SSH connection and copy the public key to the remote server.

3. Passwordless authentication:

- Once you have copied your public key to the remote server, you should be able to connect without entering a password.
- To log in to your remote account, run the following command:

```
ssh nctxxxxx@glogin1.bsc.es
```

- Again, replace "nctxxxxx" with your username. If everything is set up correctly, you should log in to your remote account without being prompted for a password.

From this point on, you should be able to connect to your Marenostrum Linux account using SSH without typing the password every time. Your private key is stored on your laptop, and the public key is on the remote server, enabling passwordless authentication. Make sure to keep your keys secure and follow proper security practices to protect your systems.

8. (Optional) Brief about Linux commands

Below is a brief manual of some main Linux commands for beginners. These commands will help you navigate, manage files, and perform common tasks in a Linux terminal.

List files and directories in the current directory:

```
ls
```

Print the current working directory (i.e., your current location in the file system):

```
pwd
```

Change directory. Use it to navigate to a different directory:

```
cd <directory_path>
```

Create a new directory:

```
mkdir <directory_name>
```

Remove an empty directory:

```
rmdir <directory_name>
```

Remove files or directories. Be cautious as it is a permanent action:

```
rm <file_name>
rm -r <directory_name>
# Use -r for recursive deletion of directories and their contents.
```

Copy files or directories from one location to another.

```
cp <source_file> <destination_path>
cp -r <source_directory> <destination_path>
# Use -r for recursive copying of directories and their contents.
```

Move or rename files or directories:

```
mv <source> <destination>
```

Display the beginning lines of a file:

```
head <file_name>
```

Display the last lines of a file:

```
tail <file_name>
```

Search for a pattern in a file or output:

```
grep <pattern> <file_name>
```

Access the manual pages for a command to get more information:

```
man <command_name>
```

Display information about active processes:

```
ps aux
```

Terminate a process using its PID (Process ID):

```
kill <PID>
```