Universitat Pompeu Fabra Barcelona

Lab 2

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

Instructions

The second Lab session consists of 5 problems. You have to compile your answers to the exercises in a report. This lab assignment has a weight of 15% of the total grade of the subject and the deadline for submission is May 20th at 23:59.

Each group will have to submit a compressed file named lab2_TxGyy.zip, where TxGyy is your group identifier. A .tar or a .tgz file is also accepted (e.g. lab2_T1G1.zip or lab2_T2G21.zip). The compressed file has to contain five folders and all the requested files with the following structure.

```
lab2_TxGyy.zip ─┬─ 1_hello_world ─┬─ helloworld.c
                │                 ├─ Makefile
                │                 ├─ job.cmd
                │                 └─ outputs
                ├─ 2_mpi_dotp ──┬─ mpi_dotp.c
                │               ├─ Makefile
                │               ├─ job.cmd
                │               └─ outputs
                ├─ 3_mv_prod ──┬─ mv_dotp.c
                │              ├─ Makefile
                │              ├─ job.cmd
                │              └─ outputs
                ├─ 4_broadcast ─┬─ matrix.c
                │               ├─ Makefile
                │               ├─ job.cmd
                │               └─ outputs
                ├─ 5_game_of_life ─┬─ game_of_life.c
                │                  ├─ aux.c
                │                  ├─ aux.h
                │                  ├─ Makefile
                │                  ├─ job.cmd
                │                  ├─ life1.bin
                │                  ├─ life2.bin
                │                  ├─ bitmaps
                │                  └─ outputs
                └─ report.pdf
```

A sample file named lab2_TxGyy.zip containing the reference codes has been published in the Aula Global. To ease the process we provide sample job files and Makefiles. You may need to modify the execution lines in the job scripts to perform your tests and the batch lines to request more cores or tasks. The Makefiles should

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

not be modified unless the exercise says so. Focus on the code and the work requested for each exercise. In the case the compilation fails, the job will finish and it will not try to run the binary.

For any arising question, please, post it in the Lab class forum in the Aula Global. However, do not post your code in the Forum. For other questions regarding the assignment that you consider that cannot be posted in the Forum (e.g. personal matters or code), please, contact the responsible of the lab **sergi.laut@upf.edu**.

Criteria

The codes will be tested and evaluated on the same cluster where you will be working. The maximum grade on each part will only be given to these exercises that solve in the most specific way and that tackle all the functionalities and work requested. All the following criteria will be applied while reviewing your labs in the cluster.

Exercises that will not be evaluated:

- A code that does not compile.
- Code giving wrong results.

Exercises with penalty:

- A code with warnings in the compilation.
- lab2_TxGyy.zip delivered files not structured as described previously.

1. Hello world (20%)

In this exercise we will work with MPI Communicators and Groups, executing a code with 16 processes. You will have to write a code in C with 4 phases. In each phase sub-communicators are generated in a different way. The output of your code has to be similar to the following (note that the ordering of prints within each phase is not important):

```
PHASE 1

Hi, I'm rank 0. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 1. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 2. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 3. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 4. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 5. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 6. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 7. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 8. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 9. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 10. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 11. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 12. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 13. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 14. My communicator is MPI_COMM_WORLD and has a size of 16 processes.
Hi, I'm rank 15. My communicator is MPI_COMM_WORLD and has a size of 16 processes.

PHASE 2

Hi, I was rank 0 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 0 in communicator SPLIT_COMM_0 which has 4 processes.
Hi, I was rank 1 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 1 in communicator SPLIT_COMM_0 which has 4 processes.
Hi, I was rank 2 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 2 in communicator SPLIT_COMM_0 which has 4 processes.
Hi, I was rank 3 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 3 in communicator SPLIT_COMM_0 which has 4 processes.
Hi, I was rank 4 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 0 in communicator SPLIT_COMM_1 which has 4 processes.
Hi, I was rank 5 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 1 in communicator SPLIT_COMM_1 which has 4 processes.
Hi, I was rank 6 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 2 in communicator SPLIT_COMM_1 which has 4 processes.
Hi, I was rank 7 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 3 in communicator SPLIT_COMM_1 which has 4 processes.
Hi, I was rank 8 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 0 in communicator SPLIT_COMM_2 which has 4 processes.
Hi, I was rank 9 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 1 in communicator SPLIT_COMM_2 which has 4 processes.
Hi, I was rank 10 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 2 in communicator SPLIT_COMM_2 which has 4 processes.
Hi, I was rank 11 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 3 in communicator SPLIT_COMM_2 which has 4 processes.
Hi, I was rank 12 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 0 in communicator SPLIT_COMM_3 which has 4 processes.
Hi, I was rank 13 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 1 in communicator SPLIT_COMM_3 which has 4 processes.
Hi, I was rank 14 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 2 in communicator SPLIT_COMM_3 which has 4 processes.
Hi, I was rank 15 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 3 in communicator SPLIT_COMM_3 which has 4 processes.

PHASE 3

Hi, I was rank 0 in communicator SPLIT_COMM_0 which had 4 processes. Now I'm rank 0 in communicator EVEN_COMM which has 8 processes.
Hi, I was rank 2 in communicator SPLIT_COMM_0 which had 4 processes. Now I'm rank 1 in communicator EVEN_COMM which has 8 processes.
Hi, I was rank 0 in communicator SPLIT_COMM_1 which had 4 processes. Now I'm rank 2 in communicator EVEN_COMM which has 8 processes.
Hi, I was rank 2 in communicator SPLIT_COMM_1 which had 4 processes. Now I'm rank 3 in communicator EVEN_COMM which has 8 processes.
Hi, I was rank 0 in communicator SPLIT_COMM_2 which had 4 processes. Now I'm rank 4 in communicator EVEN_COMM which has 8 processes.
Hi, I was rank 2 in communicator SPLIT_COMM_2 which had 4 processes. Now I'm rank 5 in communicator EVEN_COMM which has 8 processes.
Hi, I was rank 0 in communicator SPLIT_COMM_3 which had 4 processes. Now I'm rank 6 in communicator EVEN_COMM which has 8 processes.
Hi, I was rank 2 in communicator SPLIT_COMM_3 which had 4 processes. Now I'm rank 7 in communicator EVEN_COMM which has 8 processes.

PHASE 4

Hi, I was rank 1 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 0 in communicator ODD_COMM which has 8 processes.
Hi, I was rank 3 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 1 in communicator ODD_COMM which has 8 processes.
Hi, I was rank 5 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 2 in communicator ODD_COMM which has 8 processes.
Hi, I was rank 7 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 3 in communicator ODD_COMM which has 8 processes.
Hi, I was rank 9 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 4 in communicator ODD_COMM which has 8 processes.
Hi, I was rank 11 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 5 in communicator ODD_COMM which has 8 processes.
Hi, I was rank 13 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 6 in communicator ODD_COMM which has 8 processes.
Hi, I was rank 15 in communicator MPI_COMM_WORLD which had 16 processes. Now I'm rank 7 in communicator ODD_COMM which has 8 processes.
```

Exercise 1 — MPI Communicators and Groups. 20%

1. PHASE 1: Using the communicator MPI_COMM_WORLD print the rank of each process, the name of the communicator and the total amount of processes.

2. PHASE 2: Split the communicator MPI_COMM_WORLD into 4 sub-communicators with 4 processes each. Name each sub-communicator differently. For each process, print its rank in the MPI_COMM_WORLD communicator and in the new one.

3. PHASE 3: Create a group of processes containing the even ranks. Create a communicator for the group called EVEN_COMM. For each process, print its rank in the PHASE 2 communicator and in the new communicator.

4. PHASE 4: Create the group of processes in MPI_COMM_WORLD that are not in EVEN_COMM using MPI Group functionalities. Create a communicator for this new group called ODD_COMM. Print the rank of each process in the MPI_COMM_WORLD and in ODD_COMM.

Note: you can set the name of a communicator with the function **MPI_Comm_set_name (MPI_Comm, char*)**. You can retrieve the name of a communicator with the function **MPI_Comm_get_name (MPI_Comm, char*, int*)**.

Note: the order of the prints does not have to be such as the one in the figure, but phases have to be separated.

2. Dot product (20%)

In this exercise we will create the hybrid OpenMP/MPI version of the dot product operation. The dot product is an algebraic operation that performs a vector-vector operation as described in the following formula:

$$y = \vec{p} \cdot \vec{q} \tag{1}$$

Where:

- $\vec{p}$ and $\vec{q}$ are vectors.
- y is a scalar.

In a shared memory implementation of the dot product all threads have access to all the components of the multiplying vectors. However, in a distributed memory system data is split between processes.

In this exercise, we will implement the parallel read of a binary file using MPI I/O tools and then further parallelize the dot product code implemented in Lab1.

Exercise 2 — **MPI I/O and global communications. 20%**

1 Create a file named *mpi_dotp.c* in the folder *2_mpi_dotp*. This code does not need to get any arguments.

2 Write a function in C that reads a binary file in parallel. The function must have the following structure:

```
double *par_read(char *in_file, int *p_size, int rank, int nprocs);
```

Where *in_file* is the name of the file you open, *p_size* is a pointer to an integer that is modified in the function to store the size of the part of the array read by the process, *rank* is the identifier of the process and *nprocs* is the total amount of processes. The function returns a pointer to the data that has been obtained from the binary file.

3 Use the function to read the files in the paths *"/shared/Labs/Lab_2/array_p.bin"* and *"/shared/Labs/Lab_2/array_q.bin"*. The binary files store doubles.

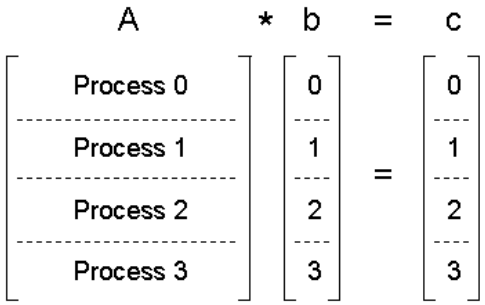
4 Parallelize the dot product using MPI. Combine it with the shared memory version (lab1) to obtain a hybrid version (differently than in lab1 in this case the dot product is performed in arrays of doubles) .

5 Submit a job executing the code and obtain results for 1 process with 1, 2, 4, 8 and 16 threads (you can use the sample slurm script dotp_job.cmd as a base for your job files). Submit a job executing the code and obtain results for 1, 2, 4, 8 and 16 processes with 1 thread each. Which case scales better? Plot the strong speedup for both cases.

6 Submit jobs executing the code and obtain results for various combination of processes and threads: 2-12 and 4-6. Which is the best case?

3. Matrix Vector Product (20%)

The Matrix Vector product is a common operation in scientific codes. When working with distributed memory systems matrices and vectors are split into smaller parts each stored by a one process.



In the figure above we see the schematic of a matrix vector multiplication using 4 processes. Note each process has a part of the matrix A, of the multiplying vector, b, and of the solution c. However, in the multiplication process each process requires to have the full vector b.

Exercise 3 — **MPI I/O and global communications. 20%**

1 Create a file named *mv_prod.c* in the folder *3_mv_prod*. This code does not need to get any arguments.

2 Use your previously implemented parallel read function to read the files in paths *"/shared/Labs/Lab_2/matrix.bin"* and *"/shared/Labs/Lab_2/matrix_vector.bin"*.

matrix.bin stores the entries in double precision in a square matrix. The entries are ordered row by row. Since it will be stored in a simple array you can use the following definition to access its values.

```
#define ind(i, j, nx) (((i)*(nx)) + (j))
```

Where i is its row index, j is its column index and nx is the size of the matrix.

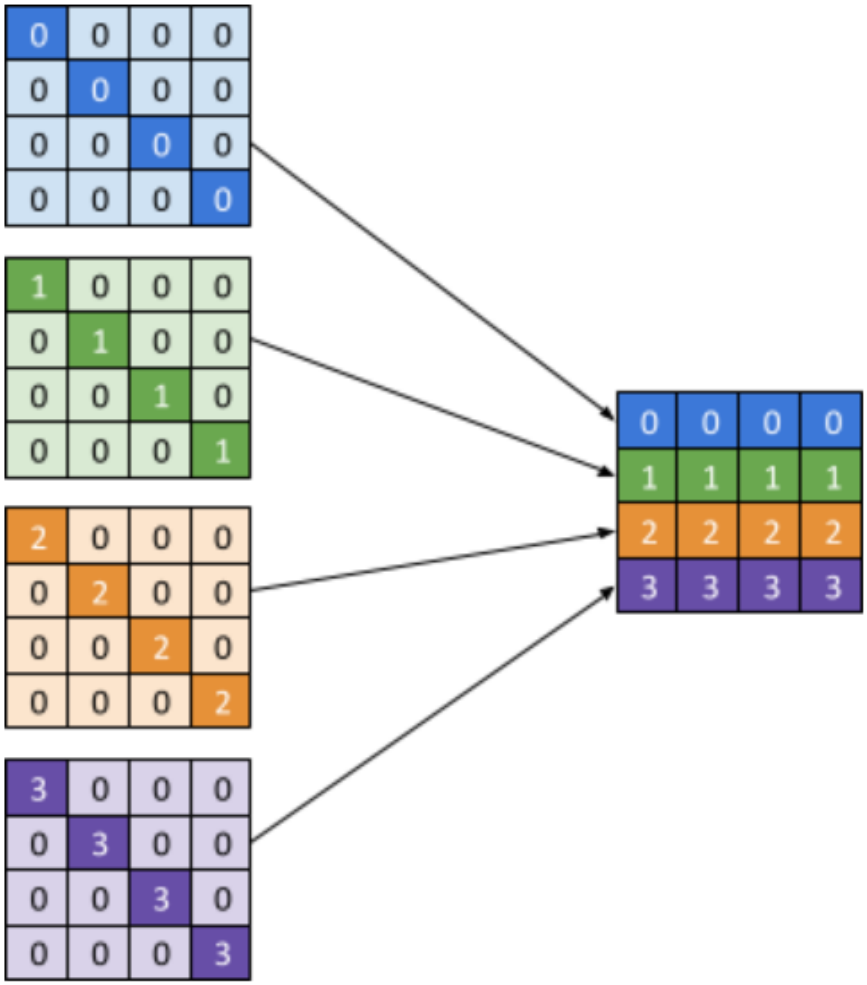
matrix_vector.bin stores the multiplying vector.

3 Implement an MPI version of the matrix vector product.

4 Submit a job executing the code and obtain results for 1, 2, 4, 8 and 16 processes (you can use the sample slurm script dotp_job.cmd as a base for your job files). Plot the strong speedup.

4. Broadcast (20%)

In this exercise we will work with data types and collective routines. The goal of this exercise is to have a process receive data from all others with a derived data type.



Exercise 4 — Data types and global communications. 20%

1 Write a program *matrix.c* that performs the following steps. Each process fills a NxN matrix (where N is the total number of processes) with 0s, except for the diagonal elements that are initialized with the process' rank identifier. Write the following routine to realize this step:

```
int *new_matrix (int size, int rank)
```

Process 0 prints its initialized matrix.

2 Next, each process sends to process 0 an array with all elements of its diagonal using a collective routine.

Process 0 overwrites the array sent by process "i" on the i-th row of its local matrix. Finally, process 0 prints its final matrix. To print the initial and final matrix, implement the following routine:

```
void print_matrix (int *matrix, int size)
```

In order to send a diagonal, a vector data type should be created and set for reading the diagonal elements of a matrix with the right offset and stride.

For example, your program output with 4 processes must be equal to this one:

```
Initial Matrix (rank 0)
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0

Final matrix (rank 0)
0 0 0 0
1 1 1 1
2 2 2 2
3 3 3 3
```

3 Try it with 4 and 8 processes.

5. Game of Life (20%)

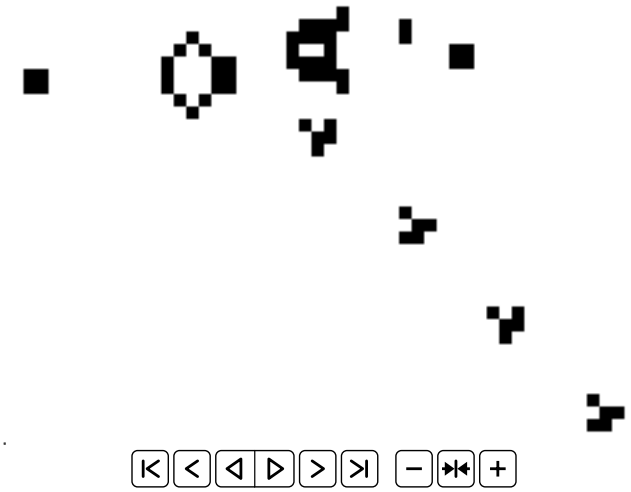
The Game of Life is a cellular automaton proposed by the mathematician John Horton Conway in 1970 that is played on a 2D square grid. It is a zero-player game, what means that it only depends on an initial configuration which may be set by a human and requires no other interactions. One only observes its evolution.

Each square (or "cell") on the grid can be either alive or dead, and they evolve according to the following rules:

- Any live cell with fewer than two live neighbours dies (referred to as underpopulation).
- Any live cell with more than three live neighbours dies (referred to as overpopulation).
- Any live cell with two or three live neighbours lives, unchanged, to the next generation.
- Any dead cell with exactly three live neighbours comes to life.

The goal of the game is to find patterns that evolve in interesting ways – something that people have now been doing for over 50 years. It is Turing complete and can simulate a universal constructor or any other Turing machine.

If your PDF reader allows it you can see the evolution of a sample Game of Life here.



In our case we will have a square domain with a randomized initial population in a binary file. We will split the domain by rows and each process will store a bunch of them. In each iteration each process will have to send to its lower neighbour its lowest row (except process P-1), and to its upper neighbour its highest row (except process 0) to compute its next generation.

Follow the steps in exercise 5. Add each step in *game_of_life.c* where it is indicated.

Exercise 5 — Point to point communications. 20%

1. Modify your binary file reading function so it reads integers and add it to *game_of_life.c*. The Variable *local_entries* has to store the matrix entries in each process.
2. Define the neighbours processes of each process. Each process has 2 neighbours, *next* and *prev*.
3. Make rank 0 print a bitmap with the initial configuration. Rank 0 has to gather other processes local matrix. You can use the function *bitmap(full_matrix, local_entries, row_size, nprocs, iteration)*.
4. Communicate rows to neighbours. *upper_send* row has to be sent to *prev* and *lower_send* to *next*. The corresponding processes need to post the receive orders.
5. Make rank 0 print a bitmap of the final configuration.
6. Provide the initial and final bitmaps for *life1.bin* and *life2.bin* with 1 and 8 processes.