Instructions for submitting

Each group has to submit a compressed file named **lab2_TxGyy.zip**, where *TxGyy* is your group identifier. A .tar or a .tgz files are also accepted (e.g. lab2_T1G1.zip or lab2_T2G21.zip). The compressed file has to contain three folders (ex1, ex2 and ex3) and all the requested files with the following structure (no job.sh or Makefiles need to be included).

```
lab2_TxGyy.zip ─┬─ ex1──── matrix.c
                ├─ ex2─┬─ pi.c
                │      ├─ speedup.py
                │      └─ speedup.png
                └─ ex3──── count.c
```

A sample file named *lab2_TxGyy.zip* containing the reference codes has been published in the Aula Global. This lab assignment has a weight of **15%** of the total grade of the subject and the deadline for this assignment is **May 21th at 23:59**.

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 cannot be posted in the Forum (e.g. personal matters or code), please contact the responsible of the lab **xavier.saez@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 the 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.

Previous considerations

The cluster maintains the configuration from OpenMP laboratory, therefore the IP is **10.49.0.95** and your password has not changed.

Remember that the compilation cannot be done in the login node and that it has to be submitted to the compute nodes. To ease this process we have included batch files as well as Makefiles in the file **lab2_TxGyy.zip.**

To ease the process, all the job files and Makefiles are in the file lab2_TxGyy.zip prepared to compile the code and run it, and you do not have to modify them. Just 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.

Notice

- Remember that it is totally forbidden to copy in these exercises. It is understood that there may be communication between the students during the realization of the activity, but the delivery of these exercises has to be differentiated from the rest.

- Therefore, deliveries that contain an identical part regarding deliveries of other groups will be considered copies and all the people involved (without the link between them being relevant) will suspend the activity delivered.

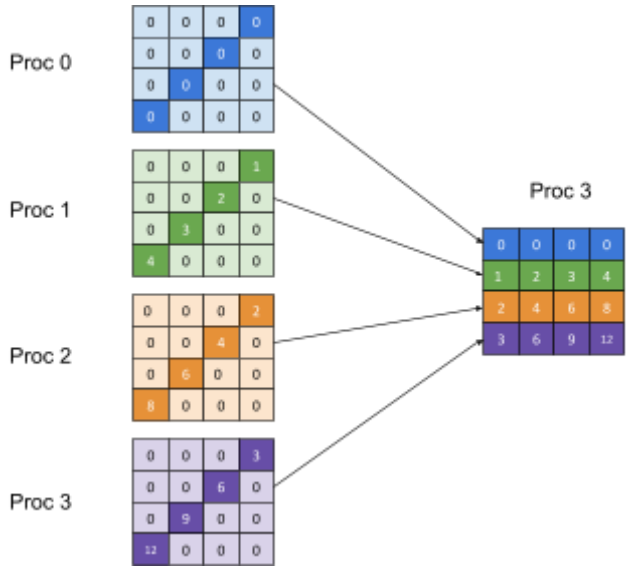
Exercise 1: Broadcast (30%)

Write a program **`matrix.c`** that performs the following steps.

Each process fills a NxN matrix with 0s, where N is the total number of processes, except for the anti-diagonal elements that are initialized with the process' rank identifier multiplied by the row number. Write the following routine to realize this step:

```
void init_matrix (int *matrix, int size, int rank)
```

Process N-1 prints its initialized matrix. Next, each process sends to process N-1 an array with all elements of its anti-diagonal using a **collective routine**. Process N-1 creates a new matrix to store the arrays received from all processes, where the i-th row of this new matrix is the received array from the process "i". Finally, process N-1 prints the result matrix:



To print the initial and result matrix, implement the following routine:

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

In order to send the anti-diagonal, a **vector data type** should be created and set for reading the anti-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:

Output job4.sh

```
Initial Matrix (rank 3)
0        0       0       3
0        0       6       0
0        9       0       0
12       0       0       0

Final matrix (rank 3)
0        0       0       0
1        2       3       4
2        4       6       8
3        6       9       12
```

Exercise 2: Pi (35%)

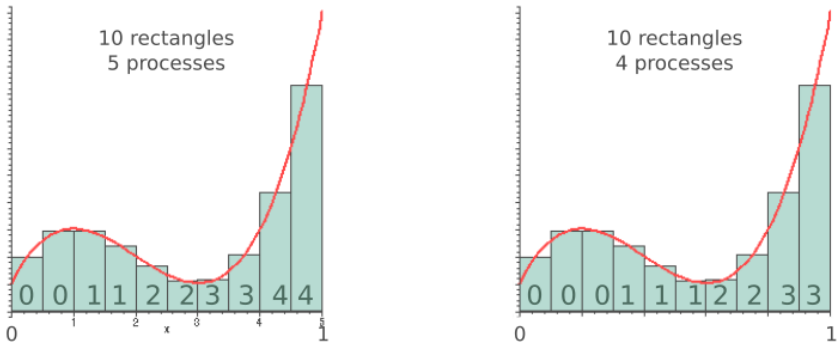
Write a program **`pi.c`** that computes the Pi number by integrating the following mathematical expression:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$$

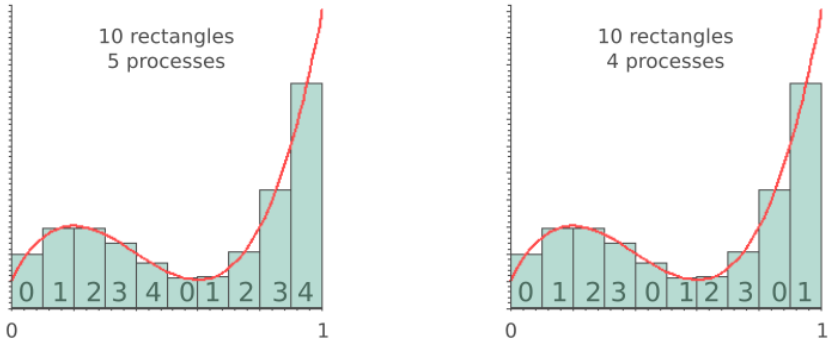
In order to solve this integral, we are going to apply the rectangle method that consists of calculating the area under the curve. The way to parallelize the area calculation is by dividing this area into rectangles with the same width and distributing them among the processes.

For this exercise, we are going to implement two different ways to distribute the rectangles among the processes. Look at the following example with 4 processes to understand them:

a) Continuous distribution



b) Distribution with gaps



Firstly, the program will compute the Pi number with the **continuous distribution**. To do this distribution, each node will receive the total number of rectangles to use and it will decide which rectangles it has to compute. Note that in the midpoint approximation, the height of each rectangle is the f (x) value in the

middle of the subdivision. Once all nodes have completed their work, all the rectangle areas will be accumulated by process 0 and this one will print the final result and the total elapsed time:

```
double pi_cont_dist (long nrect, int rank, int nprocs)
```

Next, the program will repeat these steps with the distribution with gaps. And again, process 0 will print the final result and the total elapsed time:

```
double pi_gaps_dist (long nrect, int rank, int nprocs)
```

An example of the output with 10 000 000 rectangles using 4 processes is:

Output job4.sh

```
rect. = 100000000

Pi    = 3.1415926535896825
Time continuous distr. = 0.0907

Pi    = 3.1415926535902168
Time distr. with gaps = 0.0913
```

Once parallelized we want to see the results for 1, 2, 4, 8 and 16 processes. For this, there are the following batch files with the required configuration. Submit them individually to get the result for a specific number of processes:

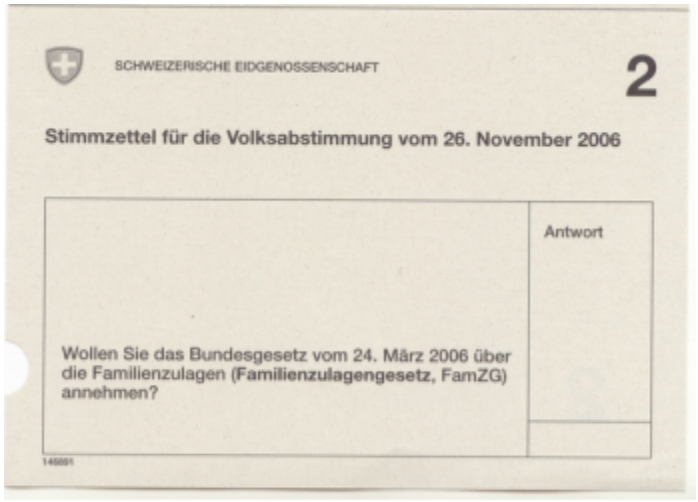
job1.sh: output files named ex2_1pr_%j.
job2.sh: output files named ex2_2pr_%j.
job4.sh: output files named ex2_4pr_%j.
job8.sh: output files named ex2_8pr_%j.
job16.sh: output files named ex2_16pr_%j.

Plot the speedup, and save the “speedup.py” file and the speedup image to deliver them.

Note: Use double type to do the computations.

Exercise 3: Polling (35%)

Switzerland's voting system is unique among European nations. Approximately four times a year, voting occurs over various initiatives and referendums, where policies related to social issues (e.g. welfare, healthcare, and drug policy), public infrastructure (e.g. public transport and construction projects) and environmental issues (e.g. environment and nature protection), economics, public finances (including taxes), etc... are directly voted by people.



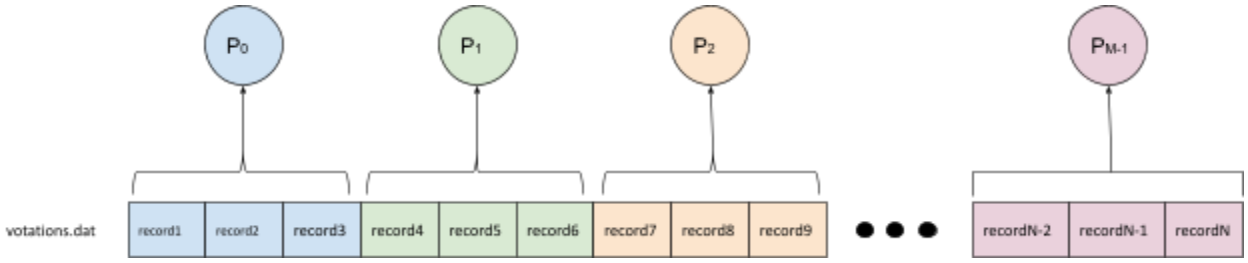
Referendum ballet. (wikipedia)

We are going to develop a small program named `count.c` that reads in parallel a file named `votations.dat` with the results from each votation table and proposed initiative. Each record in the `votations.dat` file follows this structure:

```
typedef struct {
    int idTable;
    int idQuestion;
    int yes;
    int no;
 }  TRecord;
```

Where the votation table (**idTable**) and the voted initiative (**idQuestion**) are identified by a code that is an integer number. The record also contains the number of votes in that votation table for the initiative (**yes**) and against the initiative (**no**).

The `count.c` program should distribute the records of the `votations.dat` file among the processes as the following figure shows:



Each process calculates the number of votes for and against each initiative and prints how many records it has processed. Finally, process 0 will gather the results from all processes, and it will show the total percentage of each answer per initiative.

For example, the output with 4 processes must be equal to this one (discarding rows ordering):

Output job4.sh

```
Proc   0. Counted votes = 12342580
Proc   1. Counted votes = 12494495
Proc   2. Counted votes = 12667760
Proc   3. Counted votes = 12416870
------------------------------------------------------------
Question 0: yes: 52.5% (5238511) no: 47.5% (4745830)
Question 1: yes: 23.4% (2334342) no: 76.6% (7649999)
Question 2: yes: 65.5% (6543912) no: 34.5% (3440429)
Question 3: yes: 7.4% (743497) no: 92.6% (9240844)
Question 4: yes: 45.5% (4539760) no: 54.5% (5444581)
------------------------------------------------------------
Total votes = 49921705
```

Another example is the output with 8 processes, which must be equal to this one (discarding rows ordering):

Output job8.sh

```
Proc   0. Counted votes = 6237925
Proc   1. Counted votes = 6104655
Proc   2. Counted votes = 6216260
Proc   4. Counted votes = 6413480
Proc   5. Counted votes = 6254280
Proc   6. Counted votes = 6250785
Proc   7. Counted votes = 6166085
Proc   3. Counted votes = 6278235
------------------------------------------------------------
Question 0: yes: 52.5% (5238511) no: 47.5% (4745830)
Question 1: yes: 23.4% (2334342) no: 76.6% (7649999)
Question 2: yes: 65.5% (6543912) no: 34.5% (3440429)
Question 3: yes: 7.4% (743497) no: 92.6% (9240844)
Question 4: yes: 45.5% (4539760) no: 54.5% (5444581)
------------------------------------------------------------
Total votes = 49921705
```