Parallel Computing
Lab 1

In this lab you will implement a histogram in MPI.

What is the problem definition?

Your program reads a series of p integers from a file then shows a histogram of n bins of these numbers. The output of your program is not a picture of a histogram just the values of each bin. Assume that the largest floating-point number will not be larger than 100 (one hundred).
For example, suppose we have six numbers (37, 11, 17, 62, 72, 79) and four bins. This means:

bin[0] will be responsible for numbers in the range [0, 25[
bin[1] will be responsible for numbers in the range [25, 50[
bin[2] will be responsible for numbers in the range [50, 75[
bin[3] will be responsible for numbers in the range [75, 100[

The output of your program on the screen must be:
bin[0] = 2
bin[1] = 1
bin[2] = 2
bin[3] = 1

The range of numbers assigned to each bin depends on (100.00 / number of bins). If the 100 is not divisible by the number of bins, then the last bin is the one with the shortest range. For example if the number of bins = 7, then ceil(100/7) = 15. This means bin[0] will be responsible for the range [0,15[, bin[1] for [15,30[, …bin[6] for [90,100]

What is the input?

Your program receives two inputs: number of bins and a filename that contains the floating point numbers. The command line with mpiexec will specify the number of processes.
The input file starts with an int, which is the number of floating points available, then followed by the floating point numbers. So, the numbers of the above example can be in a file that looks like this:

6 7 1 31 57 72 77

There is one space between each two numbers. The first number is the total number of integers. Here we have 6 integers.

We are providing you with a file random-num.c to generate the integers file. Feel free to use that file, compile and run it to generate the integers.

Assume your program is called *histogram*, the command line is expected to be:

mpiexec -n x ./histogram b filename
Where:
b: is the number of bins, $0 < b <= 50$
x: number of processes, $0 < t <= 50$
filename: the name of the text file that contains the floating point numbers

What is output?

You program must print on screen, the bins and their counts, as shown in the example above.

How will you solve this?

Each process will do the following:
Part 1: No communication needed and no MPI work done.

- Open the file and read the number of elements: x.
- Dynamically allocate an array of x integers.
- Read the integers from the file and into the array.
- Dynamically allocate an array of b integers initialized to 0. These are the bins.

Part 2: MPI starts and parallel processing.

- Each process, based on its rank, calculates the starting and ending points in the array of integers where it will process. Each process will be responsible of subset of the elements in the array not all the array.
- Each process will fill its own local histogram based on the subset of the numbers it has processed.

Part 3: Combine the results.

- Combine all the local histogram into one main histogram at process 0.
- Process 0 prints the histogram on the screen.
- End of MPI
- End of program

For example: Suppose we have the following:

- 3 bins
- a file with 10 integers: {1 4 8 7 3 5 1 8 9 6 7}
- 3 processes

Then

- 3 bins → bin[0] range [0, 34[, bin[1] range [34, 68[, and bin[2] range [68, 100].
- 3 processes: process 0 will be responsible for array elements from element 0 to 2, process 1 from 3 to 5, and process 2 from elements 6 to 9. As you can see, we have a small work imbalance here because 10 is not divisible by the number of processes.

After you are done implementing your program, you need to do the following experiments, analyze them, and include them in your report, as indicated below.

You compile your code with:

mpicc -Wall -std=c99 -o histogram netID.c
(netID.c is your program where netID is your own netID)
Do not forget to do the steps needed to compile/run MPI programs on crunchy machines as stated on the course website. If you don't do it, the compilation and execution may not work.

The report

You need to do the following experiments. Assume the number of bins is fixed to 10.

Experiment 1: Assume the number of bins is fixed to 10. We will use the *time* command of Linux. We need to see the speedup for different problem sizes. Draw a table as follows and for each entry calculate speedup = (time with 1 process / time with x processes). The speedup can be < 1 in case there is a slowdown. This is likely to happen for small problem sizes.

# integers→	**1,000**	**10,000**	**100,000**	**1000,000**	**10,000,000**
1	1	1	1	1	1
2					
3					
4					

- What happens to the speedup as the problem size increases? Why do you think that happens?
- What happens to the speedup as the number of processes increases with the same problem size? Why do you think that happens?

Experiment 2: With the same numbers of experiment 1 above, draw the same table but with efficiency = speedup/#processes

- What is your conclusion, based on the numbers you measures, about the efficiency as the problem size increases for the same number of processes? and as the number of processes increases for the same problem size?

Important:

- Do all the experiments on the same machine (crunchy1 or crunchy 3, etc ..). Repeat each measurement 3-5 times and take the average to get more precise numbers and avoid fluctuations.

Regarding compilation

- Name your source file: netID.c (where netID is your netID number). Add as many comments as you can to your code. This will help us give partial credit in case your program does not compile.
- Name your report netID.pdf (where netID is your netID number).
- Do the compilation and execution on crunchyx (x = 1, 3, 5, or 6) machines.

What to submit?

Add the source code netID.c as well as the pdf file that contains your results to a zip file named: lastname.firstname.zip
Where lastname is your last name, and firstname is your first name.

How to submit? Through the assignment sections of Brightspace.

Have Fun!