Multicore Processors: Architecture & Programming
Lab 1

In this first lab you will implement a histogram in OpenMP.

What is the problem definition?

Your program reads a series of p floating point numbers 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 100.00 (one hundred).
For example, suppose we have six numbers (37.7, 11.2, 17.7, 62.1, 72.4, 79.1) 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).

What is the input?

Your program receives three inputs: number of bins, number of threads and a filename that contains the floating point number.
That 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 0.7 11.2 17.7 12.1 2.4 7.1
There is a space between each two numbers.

We are providing you with a file random-num.c to generate the floating point file. Feel free to use that file, compile and run it to generate the floating point numbers, or build your own file generator.

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

./histogram b t filename
Where:
b: is the number of bins, $0 < b <= 50$
t: number of threads, $0 < t <= 100$
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?

You will implement three versions of the histogram:

- sequential version
- ver1: parallel version where each thread is responsible for a bin
- ver2: parallel version where each thread is responsible for a subset of the numbers

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

The report

You need to do the following experiments. Assume the number of bins is fixed to 20.

Experiment 1: To see the effect of the sequential version on the overall performance. Here, for each run, you need to measure two things:
- The *real* part of the *time* command
- The parallel part of your code, that does *not* include: reading the file, allocating and populating the array, and write the histogram on the screen. You can measure this using the clock() API and translate this to seconds afterwards, or use omp_get_wtime();

Draw a graph where the x-axis is a problem size (i.e. number of floating points): 1000, 10000, 100000, 1000000, 10000000, and 100000000. The y-axis is a percentage that goes from 0 to 100%. For each x value, draw two bar graphs, one for each parallel version, showing the parallel part as percentage of the total execution time. That is: (time of parallel part) / (total time from the time command). Assume 10 threads.

Experiment 2: Using the time of the parallel part, we will see the speedup we get for different number of threads. The x-axis here is the number of threads: 1, 2, 3, 4, and 5. If the problem size is divisible by the number of threads than a thread may have a bit more/less work than the others. The y-axis is the speedup relative to sequential (remember that OpenMP with one thread is a bit different than a purely sequential code). The input is a file of 1000,000 numbers and 50 bins. For each thread number, put two bar-graphs: one for each parallel version.

Experiment 3: In this experiment, we will analyze the efficiency. Using numbers from the above two experiments, and do any needed experiments in addition, fill-out the following table with the *efficiency (speedup relative to 1 thread / # threads).* Columns are problem size and rows are number of threads. We will use the timing of the parallel part only and the version of the solution of this lab not the version of lab 1. Use ver2.

	1,000	10,000	100,000	1000,000	10,000,000
1	1	1	1	1	1
2					
3					
4					
5					

Analysis: Given the above experiments, what can you say about:

- The fraction of the sequential part from the overall execution time as the problem size increases.
- The speedup as the number of threads increases, both for the version of lab 1 and the version of this lab. Justify your conclusion.
- Did the efficiency experiment produce the results you expected? Justify.

Important:

- Be careful when measuring the time inside your code that some functions return double precision floating point not single precision. Keep the double precision.
- 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: netIDver1.c (where netID is your netID number) and netIDver2.c, and netIDseq.c where ver1 and ver2 are the two versions of the parallel code and the third file is the sequential code. Add as many comments as you can. This will help us give partial credit in case your program does not compile.
- Do the compilation and execution on crunchyx (x = 1, 3, 5, or 6) machines.
- Use the latest version of gcc on crunchy by typing: module load gcc-9.2 (or higher)
- Compile with: gcc -fopenmp -Wall -std=c99 -o histogram netID.c

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!