ME759
High Performance Computing for Engineering Applications

The goals of this assignment are as follows:

- Understanding how to coordinate threads in their execution
- Understanding how to simultaneously use vectorization and OpenMP to compound the benefits of ILP and multi-threaded parallelism

1. Assume that you have a picture that has dimensions 1800 by 1200; i.e., it has 1800×1200 pixels. This image is essentially a 2D matrix; each of the 1800 rows has 1200 entries in it. The camera used to take the picture has seven colors it can capture. Your task is to generate a color histogram, which reports how many times each color i , $0 \leq i \leq 6$, shows up in the picture (for simplicity, assume the colors are labeled 0 through 6).

Your solution should run like this

```
>> ./problem1 N
```

with an understanding that there is a file in the current directory called `picture.inp`, which holds the image. The image is simply a random collection of integer values 0, 1, 2, 3, 4, 5, and 6. These integers are stored in `picture.inp` in a one-integer-per-line fashion. The value N represents the number of OpenMP threads you choose to use to run your code.

Given a picture, you have to produce output (sent to the `stdout`; i.e., the terminal) that has the following format:

- The first 7 lines of output have each one integer per line. On line i, $0 \leq i \leq 6$, you report how many times color i has been found in the picture.
- On line eight you report the value of N
- On the ninth line you report the shortest amount of time in millisecond required to finish your histogram. The shortest amount of time is obtained by computing in a loop the histogram for 10 times. Read once the input image and then run 10 histogram generation steps. Do not include the read stage/data setup in your timing.

Remarks:

- Provide a PDF file called `problem1.pdf` that shows a scaling analysis: how the amount of time (in milliseconds) to compute the histogram changes with the value of N. For this problem, $N = 1,2,4,6,8,10,14,16,20$.
- In this pdf comment on the behavior of your solution after $N > 6$. Kudos to those who can continue to accelerate the code for $N > 6$.

2. In HW03, we explored how vectorization can be applied to a memory-bound problem. Here, we will explore how it can be used to speed up a compute-bound problem. In the code provided in `problem2.cpp`, you are provided with a version of the integral computation problem from HW04 that uses both explicit vectorization in the form of intrinsics and parallelization via OpenMP. Aside from using vector intrinsics, we must also use a special math library as the built-in math function (in C and C++) are not vectorizable because they are contained in a separate object file (If you aren't familiar with how shared object files work, that's ok- the details are not important. Just know that the math functions are serial). We use the open source GROMACS library to carry out the vectorized math functions. (NOTE: the GROMACS library is good, but it is not identical to the algorithms used by the C or C++ standard libraries to calculate the math functions).

Your solution should run like this

```
>> ./problem2 N
```

Part A: Your code should produce three outputs (one per line): the number of threads N, the **minimum** wall-time over 10 iterations, and the value of the integral. I also recommend outputting the percent error to a file (you don't need to turn this in). For this problem, run your code for N=1 to 40 (inclusive) on the `slurm_shortgpu` partition on Euler. Plot this scaling analysis in a PDF file called `problem2a.pdf`. On this same figure, overplot the scaling analysis using your code from HW04. The line for this problem should be in blue, and the line from your code should be in red. Including a legend is preferred, but not required.

On the forum, discuss the speedup (if any) you see using the provided vectorized code. Also discuss the errors you calculated. The calculations in the provided code are being done in double precision. Recall that the machine epsilon for double precision is about 1e-16 and for single precision is 1e-7. Compare the calculated errors to the machine epsilons. Discuss possible sources of the errors (hint: it's not due to truncation or rounding).

Part B: Recall from the lectures that a superscalar architecture can execute multiple instructions simultaneously. This is referred to as Instruction Level Parallelism (ILP). It therefore follows that we may be able to increase the performance of our code by increasing the ILP. One common technique for increasing ILP in tight loops is called "loop unrolling." Suppose we want to calculate the sum of an array.

```
for (int i=0; i<N; i++)
    sum += x[i];
```

This loop has very low arithmetic intensity (we are doing one add per load). We can increase the arithmetic intensity by exploiting ILP.

```
for (int i=0; i<N; i+=2) {
    sum += x[i];
    sum += x[i + 1];
}
```

Note that this requires N to be a multiple of 2. Because we are doing two sums per loop iteration *and* they are independent, the CPU can do both adds simultaneously. This has the possibility of increasing performance through ILP.

Apply loop unrolling to both the vectorized version of the integration as well as your version from HW04. Perform the same scaling analysis as in Part A in a file called `problem2b.pdf`.