ECE 759
High Performance Computing for Engineering Applications
Assignment 3
Due Friday 10/18/2024 at 23:59 PM

Submit responses to all tasks which don't specify a file name to Canvas in a file called `assignment3.pdf`. Submit all plots (if any) on Canvas too. Do not zip your Canvas submission.

All *source files* should be submitted in the `HW03` subdirectory on the `main` branch of your homework `git` repo with no subdirectories. Your `HW03` folder should contain `task1.cpp`, `matmul.cpp`, `task2.cpp`, `convolution.cpp`, `task3.cpp`, and `msort.cpp`.

All commands or code must work on *Euler* without loading additional modules unless specified otherwise. A program may behave differently on your computer, so be sure to test on *Euler* before you submit. Note that this assignment is relevant to OpenMP, so the following line needs to be added to your `slurm` script:

- `#SBATCH --cpus-per-task=20` (or `-c 20` for short) should be added, which requests one node with 20 virtual cores (note the slight misnomer – Linux refers to virtual cores as cpus). The maximum number of threads required in this assignment is 20, so you should not ask for more than 20 cores.

Please submit clean code. Consider using a formatter like clang-format.

* Before you begin, copy the provided files from `Assignments/HW03` directory of the ECE 759 Resource Repo. Do not change any of the provided files since these files will be overwritten with clean, reference copies when grading.

Specify your GitHub link here:
https://github.com/ronpower7/repo759/tree/main/HW03

Problem 1. In HW02 task3, you have implemented several different ways to carry out matrix multiplication in sequential computing. In this task, you have to write a function called `mmul`, which takes the `mmul2` function in HW02 and parallelize it with OpenMP.

a) Implement the function `mmul` in a file called `matmul.cpp`, the parallel version of the `mmul2` function in HW02 task3 with the prototype defined as in `matmul.h`.

b) Write a program `task1.cpp` that will accomplish the following:

- Create and fill with `float` type numbers the square matrices `A` and `B` (with the data range and format specified in the description of HW02 task3; if the range is not explicitly given then you should populate the matrices however you like). The dimension of `A` and `B` should be `n`×`n` where `n` is the first command line argument, see below.
- Compute the matrix multiplication $C = AB$ using your parallel implementation with `t` threads, where `t` is the second command line argument, see below.
- Print the first element of the resulting C array.
- Print the last element of the resulting C array.
- Print the time taken to run the `mmul` function in *milliseconds*.
- Compile: `g++ task1.cpp matmul.cpp -Wall -O3 -std=c++17 -o task1 -fopenmp`
- Run (where `n` is a positive integer, `t` is an integer in the range [1, 20]; make sure you use `Slurm`):
  `./task1 n t`
- Example expected output:
  ```
  1.0
  1376.5
  3.21
  ```

c) On *Euler*, via `Slurm` do the following:

- Run `task1` for value `n` = 1024, and value `t` = $1, 2, \cdots, 20$. Generate a plot called `task1.pdf` which plots time taken by your `mmul` function vs. `t` in linear-linear scale. Feel free to share the plot on Piazza.

Problem 2. In HW02 task2, you've implemented the 2D convolution for sequential execution. In this task, you will use OpenMP to parallelize your previous implementation.

a) Implement in a file called `convolution.cpp` the parallel version of `convolve` function with the prototype specified in `convolution.h`.

b) Write a program `task2.cpp` that will accomplish the following:

- Create and fill with `float`-type numbers an `n`×`n` square matrix `image` (with the data range and format specified in the description of HW02 task2), where `n` is the first command line argument, see below.
- Create a 3×3 `mask` matrix (with the data range and format specified in the description of HW02 task2).
- Apply the `mask` matrix to the `image` using your `convolve` function with `t` threads where `t` is the second command line argument, see below.
- Print the first element of the resulting `output` array.
- Print the last element of the resulting `output` array.
- Print the time taken to run the `convolve` function in *milliseconds*.
- Compile: `g++ task2.cpp convolution.cpp -Wall -O3 -std=c++17 -o task2 -fopenmp`
- Run (where `n` is a positive integer, `t` is an integer in the range [1, 20]; make sure you use `Slurm`):
  `.task2 n t`
- Example expected output:
  `2.0`
  `137.5`
  `3.21`

c) On *Euler* via `Slurm`:

- Run `task2` for `n` = 1024, and `t` = $1, 2, \cdots, 20$. Generate a figure called `task2.pdf` which plots the time taken by your `convolve` function vs. `t` in linear-linear scale. Feel free to share the plot on Piazza.
- Discuss your observations from the plot. Explain to what extent the increase in the number of threads improves the performance, and why the run time does not show significant decrease after reaching a certain number of threads.

Ans: After the thread number reaches more than 10 , there is not significant decrease of the run time. Here the outermost for loop is parallelized , so the worklaod distribution per thread is almost equally distributed and as the n for the outer loop is 1024 ..so initailly the rate of change of workload per thread is very high as compared to that when thread number gradually incrases the value towards 20. and if the number (1024) is exactly divisible by no of threads , then it attains good performance while thread number is close to the highest value.

Problem 3. Implement a parallel merge sort[1] using OpenMP **tasks**.

a) Implement in a file called `msort.cpp` the parallel merge sort algorithm with the prototype specified in `msort.h`. You may add other functions in your `msort.cpp` program, but you should not change the `msort.h` file. Your `msort` function should take an array of integers and return the sorted results in place in ascending order. For instance, after calling `msort` function, the original `arr` = [3, 7, 10, 2, 1, 3] would become `arr` = [1, 2, 3, 3, 7, 10].

b) Write a program `task3.cpp` that will accomplish the following:
   - Create and fill with random `int` type numbers in the range [-1000, 1000] an array `arr` with length `n`, where `n` is the first command line argument, see below.
   - Apply your `msort` function to the `arr`. Set number of threads to `t`, which is the second command line argument, see below.
   - Print the first element of the resulting `arr` array.
   - Print the last element of the resulting `arr` array.
   - Print the time taken to run the `msort` function in *milliseconds*.
   - Compile: `g++ task3.cpp msort.cpp -Wall -O3 -std=c++17 -o task3 -fopenmp`
   - Run (where `n` is a positive integer, `t` is an integer in the range [1, 20], `ts` is the threshold as the lower limit to make recursive calls in order to avoid the overhead of recursion/task scheduling when the input array has small size; under this limit, a serial sorting algorithm without recursion calls will be used):
     `./task3 n t ts`
   - Example expected output:
     `1`
     `513`
     `3.21`

c) On *Euler* via `Slurm`:
   - Run `task3` for value `n` $= 10^6$, value `t` $= 8$, and value `ts` $= 2^1, 2^2, \cdots, 2^{10}$. Generate a plot called `task3_ts.pdf` which plots the time taken by your `msort` function vs. `ts` in linear–log scale. Feel free to share the plot on Piazza.
   - Run `task3` for value `n` $= 10^6$, value `t` $= 1, 2, \cdots, 20$, and `ts` equals to the value that yields the best performance as you found in the plot of time vs. `ts`. Generate a plot called `task3_t.pdf` which plots time taken by your `msort` function vs. `t` in linear–linear scale. Feel free to share the plot on Piazza.

[1] See the Parallel merge sort section in this link as a reference of the pseudo code.