ME 759
High Performance Computing for Engineering Applications
Assignment 8
Due Thursday 3/26/2020 at 9:00 PM

Submit responses to all tasks which don't specify a file name to Canvas in a file called assignment8.txt, docx, pdf, rtf, odt (choose one of the formats). Also, all plots should be submitted in Canvas. All *source files* should be submitted in the `HW08` subdirectory on the `master` branch of your homework git repo with no subdirectories.

All commands or code must work on *Euler* without loading additional modules unless specified otherwise. They may behave differently on your computer, so be sure to test on Euler before you submit. Note that this assignment is relevant to OpenMP and not related to GPU computing, so the following changes need to be made to your `slurm` script:

- `#SBATCH --gres=gpu:1` should be removed since GPU is not required in this assignment.
- `#SBATCH --nodes=1 --cpus-per-task=20` (or `-N 1 -c 20` for short) should be added, which requests one node with 20 cores (note the slight misnomer). 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 `HW08` of the ME759-2020 repo.

1. In HW02 task3, you have implemented several different ways to carry out matrix multiplication in sequential computing. In this task, you will take the `mmul` function in HW02 that yields the best performance and parallelize it with OpenMP.

   a) Implement in a file called `matmul.cpp` the parallel version of the best-performing `mmul` 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` (stored as 1D arrays in the order required by the best-performed `mmul` method, more details can be found in the description of HW02 task3). 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 -o task1 -fopenmp`
   - Run (where `n` is a positive integer, `t` is an integer in the range [1, 20]):
     `./task1 n t`
   - Example expected output:
     ```
     1.0
     1376.5
     3.21
     ```

   c) On an Euler *compute node*:

   - 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.

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` (stored as a 1D array in row-major order), where `n` is the first command line argument, see below.
   - Create a 3×3 `mask` matrix (stored in 1D in row-major order) of whatever mask values you like.
   - 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 -o task2 -fopenmp`
   - Run (where `n` is a positive integer, `t` is an integer in the range [1, 20]):
     `./task2 n t`
   - Example expected output:
     ```
     2.0
     137.5
     3.21
     ```

   c) On an Euler *compute node*:

   - 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.
   - 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.

   d) (**Optional**, Extra credits, 10 pts) There are ways to further improve the performance when using more threads (i.e. `t` > 10). For instance, the *data affinity* topic covered in class would be a good starting point to look into this.

   - Submit in a file `task2_op.cpp` that includes all necessary changes for improved scalability (there should be no need to change your `Convolve` function). Explain (if any) changes to the environment variables along with your discussions in c).
   - On an Euler *compute node*, run `task2_op` for value `n` = 1024, and value `t` = $1, 2, \cdots, 20$. Overlay this plot on the pattern you achieved in c) and save in a file `task2_op.pdf`.
   - Discuss your observations from the plot.

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 however you like with `int` type numbers 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 -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 an Euler *compute node*:

   - 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-linear scale.
   - 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.

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