

ME 759
High Performance Computing for Engineering Applications
Assignment 9
Due Thursday 11/09/2023 at 9:00 PM

Submit responses to all tasks which don't specify a file name to Canvas in a file called assignment9.{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 HW09 subdirectory on the main branch of your homework git repo with no subdirectories. Your HW09 folder should have task1.cpp, cluster.cpp (new version that avoids false sharing), task2.cpp, montecarlo.cpp, task3.cpp and if you attempt the bonus question, task4.cpp.

All commands or code must work on *Euler* without loading additional modules unless specified otherwise. Commands/code may behave differently on your computer, so be sure to test on *Euler* before you submit. For the OpenMP related tasks, i.e. Task 1 and Task 2, the following specifications need to be included in your `slurm` script:

- `#SBATCH --cpus-per-task=10` (or `-c 10` for short).

For Task 3 (MPI related), on top of the `slurm` script used for OpenMP, the following changes need to be made to your `slurm` script:

- Remove `#SBATCH --cpus-per-task=10` (or `-c 10` for short).
- Add `#SBATCH --ntasks-per-node=2`. This specification is only for this assignment, since only two processes are needed for task 3.

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

* Before you begin, copy the provided files from HW09 of the ME759 Resource Repo. Do not change any of the provided files since these files will be overwritten with clean, reference copies when grading.

Problem 1. In this task, you are given a function that displays a false sharing issue. Specifically, in your code, each thread will calculate the sum of the distances between a “center point” associated with this thread and a chunk of entries of a large reference array `arr` of size `n`. The function declaration is in `cluster.h`; its definition is in `cluster.cpp`. Do not modify the header file or add any additional functions to it. You will need to use the techniques discussed in class (Lecture 23) to fix the false sharing issues and assess any impact of performance (false sharing vs. no false sharing). To that end:

- Modify the current `cluster.cpp` file to solve the false sharing issue. In this particular problem, you are allowed to modify the file `cluster.cpp`, even though it is a provided file.
- Write a program `task1.cpp` that will accomplish the following:

- Create and fill with `float`-type random numbers an array `arr` of length `n`, where `n` is the first command line argument, see below. The range for these random floating point numbers is `[0, n]`.
- Sort `arr` (`std::sort` will do).
- Create and fill an array `centers` of length `t`, where `t` is the number of threads and it's the second command line argument, see below. Assuming that `n` is always a multiple of `2 * t` (your code does not need to account for the cases where `n` is not a multiple of `2 * t`), the entries in the array `centers` should be (as floating point numbers):

$$\left[\frac{n}{2t}, \frac{3n}{2t}, \dots, \frac{(2t-1)n}{2t} \right].$$

- Create and initialize with `zeros` an array `dist`s of length `t`¹, where `t` is the number of threads and it's the second command line argument, see below.
- Call the `cluster` function and save the output distances to the `dist`s array.
- Calculate the `maximum distance` in the `dist`s array.

¹Note that if you choose to use padding to avoid false sharing, the size of arrays `centers` and `dist`s can be changed accordingly.

- Print the maximum distance.
- Print the partition ID (the thread number) that has the maximum distance.
- Print the time taken to run the `cluster` function in *milliseconds*.
- Compile: `g++ task1.cpp cluster.cpp -Wall -O3 -std=c++17 -o task1 -fopenmp`
- Run (where `n` is a positive integer, `t` is an integer in the range $[1, 10]$):
`./task1 n t`
- Example expected output:
`32516`
`2`
`0.032`

A more concrete example (also given in the `cluster.h` file):

Input: `arr = [0, 1, 3, 4, 6, 6, 7, 8]`, `n = 8`, `t = 2`.

Expected `centers`: `[2, 6]`.

Expected `dists`: `[6, 3]`, where

$$6 = |0 - 2| + |1 - 2| + |3 - 2| + |4 - 2|,$$

$$3 = |6 - 6| + |6 - 6| + |7 - 6| + |8 - 6|.$$

Your code needs to output on the first line `6`, on the second line `0`, and on the third line the time taken in milliseconds.

c) Using `Slurm` on *Euler*:

- Run `task1` for value `n = 5040000`, and value `t = 1, 2, \dots, 10`. Generate a plot called `task1.pdf` which plots time taken by your `cluster` function vs. `t` in linear-linear scale. If there are irregular spikes in the run time, run the `cluster` function for 10 times and use the average time for plotting.

Problem 2. In this task, you will implement an estimation of the number π using the Monte Carlo Method¹. The idea is to generate n random floats in the range $[-r, r]$, where r is the radius of a circle, and count the number of floats that reside in the circle (call it `incircle`), then use $4 * \text{incircle} / n$ as the estimation of π . When n becomes very large, this estimation could be fairly accurate. Upon implementing this method with `omp for`, you will also compare the performance difference when the `simd` directive is added.

a) Implement in a file called `montecarlo.cpp` with the prototype specified in `montecarlo.h` the function that accomplishes the Monte Carlo Method.

b) Your program `task2.cpp` should accomplish the following:

- Create and fill with float-type random numbers an array `x` of length `n`, where `n` is the first command line argument, see below. The values in `x` should be drawn from the range $[-r, r]$, where `r` is the circle radius and it can be set to `1.0`.
- Create and fill with float-type random numbers an array `y` of length `n`, where `n` is the first command line argument, see below. The values in the array `y` should be drawn from the range $[-r, r]$, where `r` is the circle radius and it can be set to `1.0`.
- Call the `montecarlo` function that returns the number of points that reside in the circle.
- Print the estimated π .
- Print the time taken to run the `montecarlo` function in *milliseconds*.
- Compile²: `g++ task2.cpp montecarlo.cpp -Wall -O3 -std=c++17 -o task2 -fopenmp -fno-tree-vectorize -march=native -fopt-info-vec`
- Run (where `n` is a positive integer, `t` is an integer in the range $[1, 10]$):
`./task2 n t`
- Example expected output:
`3.1416`
`0.352`

c) Using `Slurm` on *Euler*:

- Run `task2` for `n = 106`, and `t = 1, 2, \dots, 10`. Generate a figure called `task2.pdf` which includes two patterns:
 - The time taken by your `montecarlo` function **without the `simd`** directive vs. `t` in linear-linear scale.
 - The time taken by your `montecarlo` function **with the `simd`** directive vs. `t` in linear-linear scale.

If there are irregular spikes in the run time, run the `montecarlo` function for 10 times and use the average time for plotting.

¹See the details about Monte Carlo Method in [this link](#). It also has an explanation about π estimation in the second paragraph of its Overview section

²Some notes about the compiler flags: `-fno-tree-vectorize` helps to avoid auto-vectorization that does not come from the OpenMP `simd` directive; `-march=native` helps to map to the vector size available on the hardware; `-fopt-info-vec` outputs a message when vectorization is achieved.

Problem 3. Write an `MPI program` to quantify the communication latency and bandwidth between two MPI processes (executed on the same node) using `MPI_Send` and `MPI_Recv`.

a) Write a program `task3.cpp` that will accomplish the following:

- Create and fill `two buffer arrays` with `float-type numbers` however you like that represent the messages to be sent or received. Both arrays should have `length n`, where `n` is the first command line argument, see below.
- Initialize the `variables necessary for` the function call of `MPI_Send` and `MPI_Recv`.
- Implement `the procedure` listed below for the communication between two processes:

```
if (rank == 0) {  
    // start timing t0  
    MPI_Send(...);  
    MPI_Recv(...);  
    // end timing t0  
} else if (rank == 1) {  
    // start timing t1  
    MPI_Recv(...);  
    MPI_Send(...);  
    // end timing t1  
}
```

- `Print the time` taken for both processes to complete (`t0+t1`) in *milliseconds*. You may need to send the timing result from one rank to the other to get the total time.
- Compile¹: `mpicxx task3.cpp -Wall -O3 -o task3`
- Run² on an *Euler compute node* (meaning in a `slurm` script with proper headers), where `n` is a positive integer:
`srun -n 2 task3 n`
- Example expected output:
`3.45`

b) You will have to do the following:

- Using `Slurm` on *Euler*, run `task3` for value `n = 21, 22, ..., 225`. Generate a plot called `task3.pdf` which plots the time taken by the communication between two processes vs. `n` in `log-log scale`.
- Based on the timing results, roughly estimate the `latency` and `bandwidth` of this point-to-point communication and discuss `how well` these estimations match with the hardware specifics.

¹Use `module load mpi/mpich/4.0.2`

²Use `module load mpi/mpich/4.0.2`, here `srun` is equivalent to `mpirun` in the context of running an MPI application with Slurm. The `-n` flag specifies number of processes.

Problem 4. (Bonus: 20pts) **NOTE: This is a bonus problem and worth 20 extra points. The maximum score of the assignment remains 100. If you score above a 100, your score will be rounded down to 100.** In this task you are given a piece of code that implements a 2D convolution of a `float`-type square `image` matrix of size $n \times n$ with a 3×3 `mask` matrix. The purpose of this task is for you to understand that you can do computation on the GPU even through OpenMP. The function declaration is in `convolve.h`; its definition is in `convolve.cpp`. Do not modify the header file or add any additional functions to it. In this task, you will also **not modify** the `convolve.cpp` file. Instead, you will write a new file `task4.cpp` that will accomplish the following:

- Create and fill with `float`-type numbers an $n \times n$ 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).
- Use the provided `convolve` function to apply the `mask` to the `image` matrix.
- Print the time taken to run the `convolve` function in *milliseconds*.
- Compile: `clang++ -O3 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda -Xopenmp-target=nvptx64-nvidia-cuda --offload-arch=sm_61 -march=native -o task4 convolve.cpp task4.cpp`. Note: Ensure you have `cuda module 11.8.0` loaded. Additionally, this command is only expected to work on *Euler* and not on your local machine (unless you modify the compile command to work with the compute capability of your GPU)
- Run (where `n` is a positive integer; make sure you use `Slurm`):
`./task2 n`.
 Additionally, make sure that you request a GPU using the `#SBATCH --gres=gpu:1` flag in your `slurm` script.
- Example expected output:
`3.21`

On euler via `Slurm`:

- Run `task4` for `n = 512, 1024, 2048` and `4096`. Generate a figure called `task4.pdf` which plots the time taken by your `convolve` function vs. `n` in linear-linear scale. Feel free to share the plot on Piazza.
- Explain the `#pragma`'s within the `convolve` function in the `convolve.cpp` file. What does it help accomplish? What in your opinion is the purpose of the `omp target enter`, `omp target teams` and `omp target exit` directive? Explain what `map` does.
- In the compile command, explain the function of the flags `--offload-arch` and `-fopenmp-targets`.