ME 759
High Performance Computing for Engineering Applications
Assignment 10
Due Thursday 12/1/2022 at 9:00 PM

Submit responses to all tasks which don't specify a file name to Canvas in a file called assignment10.{txt, docx, pdf, rtf, odt} (choose one of the formats). All plots should be submitted in Canvas. All *source files* should be submitted in the `HW10` subdirectory on the `main` branch of your homework git repo with no subdirectories. `HW10` should include `task1.cpp`, `optimize.cpp`, `task2.cpp`, `task2_pure_omp.cpp` and `reduce.cpp`.

All commands or code must work on *Euler* without loading additional modules unless specified otherwise. The executables may behave differently on your computer, so be sure to test on Euler before you submit. For the ILP task; i.e. Task 1, you will not need to use multiple cores, thus, asking for 1 node and 1 core (`-N 1 -c 1`) would be sufficient. For the hybrid OpenMP+MPI task, i.e. Task 2, the following specifications need to be included in your `slurm` script:

- `#SBATCH --nodes=2 --cpus-per-task=20 --ntasks-per-node=1`

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

Before you begin, copy the provided files from `HW10` of the ME759 Resource Repo. These provided files will be overwritten with clean, reference copies when grading.

---

1. (50pts) In this task, you will explore the optimizations using ILP (instruction level parallelism) based on the code examples given in Lecture 26. Some macros and utils functions are defined in the provided file `optimize.h` with the same naming fashion as the code examples in the lecture slides. You will need to accomplish the following:

   a) Write five optimization functions in `optimize.cpp` that each (either represents the baseline, or) uses a different technique to capitalize on ILP as follows:
   - `optimize1` will be the same as `reduce4` function in slide 20.
   - `optimize2` will be the same as `unroll2a_reduce` function in slide 31.
   - `optimize3` will be the same as `unroll2aa_reduce` function in slide 33.
   - `optimize4` will be the same as `unroll2a_reduce` function in slide 36.
   - `optimize5` will be similar to `reduce4`, but with $K = 3$ and $L = 3$, where K and L are the parameters defined in slide 39.

   b) Write a program `task1.cpp` that will accomplish the following:
   - Create and fill a `vec v` of length `n` with `data_t` type values generated any way you like (with this freedom, it is your responsibility to prevent data overflow); `n` is the first command line argument of this script.
   - Do the following for each `optimizeX` function:
     - Call your `optimizeX` function to get the result of `OP` operations and save it in `dest`.
     - Print the result of `dest`.
     - Print the time taken to run the `optimizeX` function in *milliseconds*.
   - Compile[1]: `g++ task1.cpp optimize.cpp -Wall -O3 -std=c++17 -o task1 -fno-tree-vectorize`
   - Run on a Euler compute node with a `Slurm` script (where `n` is a positive integer): `./task1 n`
   - Example expected output:
     ```
     3125
     0.706
     3125
     0.710
     3125
     0.353
     3125
     0.354
     ```

[1] Please compile with `g++` of version at least `10.2`. If you are on Euler, you can safely use the default version of `g++`.

```
3125
0.236
```

c) On an Euler *compute node*:

- Run `task1` for `n` $= 10^6$, with the settings of `data_t`, `OP`, and `IDENT`, and the pdf files naming conventions mentioned in Table 1. Each pdf should plot the time taken by all five of your `optimizeX` functions and one additional data point (as the sixth data point) from SIMD version of `optimize1`[2] vs. `X` in linear–linear scale, where `X` $=$ `1,...,6`. Run these `optimizeX` functions 10 times each, and use the average time for plotting.
- Note for `optimize.h` file: You can change the definition of the macros and `typedef` in `optimize.h` to run tests for plotting, but your code should not depend on any changes in the provided `optimize.h` file in order to compile and run. When we grade, `optimize.h` will still be overwritten by a clean copy.

Table 1: Setting of macros for each file.

	`data_t`	`OP`	`IDENT`
`task11.pdf`	`int`	`+`	`0`
`task12.pdf`	`int`	`*`	`1`
`task13.pdf`	`float`	`+`	`0.f`
`task14.pdf`	`float`	`*`	`1.f`

[2]data point `X=6` should come from the result of `optimize1` when compiled with the following command instead:
`g++ task1.cpp optimize.cpp -Wall -O3 -std=c++17 -o task1 -march=native -fopt-info-vec -ffast-math`

2. (50pts) In this task, you will implement a parallel reduction (summation of an array) using a hybrid OpenMP+MPI implementation. You will use OpenMP to speed up the reduction, and use two MPI processes that each runs on one node to execute the `reduce` function to add further parallelism. Figure 1 demonstrates the expected work flow of your program.

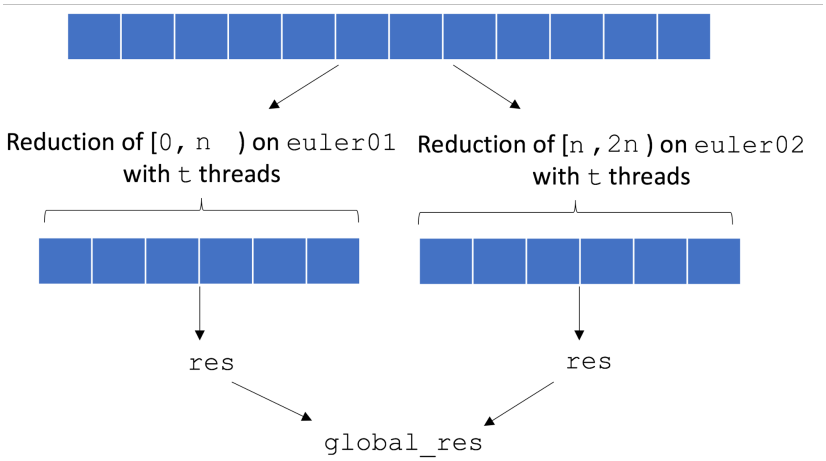

Figure 1: Schematic for the execution of the reduction program.

a) Implement in a file called `reduce.cpp` using the prototype specified in `reduce.h` the function that employs OpenMP to speed up the reduction as much as possible (i.e., use a `simd` directive).

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

- Create and fill with `float`-type random numbers in the range [-1.0, 1.0] an array `arr` of length `n`, where `n` is the first command line argument. Note that `n` is *half* of the length of the array that we are doing reduction on, refer to Figure 1.
- Initialize necessary variables for the MPI environment.
- Set the number of OpenMP threads as `t`, where `t` is the second command line argument.
- Call the `reduce` function and save the result in each MPI process's local `res` as indicated in Figure 1.
- Use `MPI_Reduce` to combine the local results and get the `global_res`.
- Print the `global_res` from one process.
- Print the time taken for the entire reduction process (including the call to `reduce` function and `MPI_Reduce`) in *milliseconds*[1].
- Compile[2]: `mpicxx task2.cpp reduce.cpp -Wall -O3 -o task2 -fopenmp -fno-tree-vectorize -march=native -fopt-info-vec`
- Run[3] on an Euler compute node using a `Slurm` script (where `n` is a positive integer, `t` is an integer in the range [1, 20]):
  `srun -n 2 --cpu-bind=none ./task2 n t`
- Example expected output:
  `3562.7`
  `0.352`

c) Write another simple test program called `task2_pure_omp.cpp` that uses `t` threads, and calls the `reduce` function to do a reduction on an array of size `n`. The initialization of this array is similar to that in `task2.cpp`.

[1] This time is the "absolute" time. You will start timing when the first process calls the `reduce` function (you may add `MPI_Barrier` before timing starts to make sure that the two processes approximately start at the same time) and end timing when `MPI_Reduce` is finished. Do not time each process separately like in HW09.

[2] Use `module load mpi/mpich/4.0.2`. Please compile with `g++` of version at least `10.2`. If you are on Euler, you can safely use the default version of `g++`.

[3] Here `srun` is equivalent to `mpirun` in the context of running an MPI application with Slurm. You can add `export OMP_DISPLAY_AFFINITY=true` to your Slurm script to check the mapping between OpenMP threads and the physical cores. This information might help you navigate the issues if you found that more threads could lead to worse performance.

- To compile: `g++ task2_pure_omp.cpp reduce.cpp -Wall -O3 -o task2_pure_omp -fopenmp -fno-tree-vectorize -march=native -fopt-info-vec`
- To run on an Euler compute node using a `Slurm` script (where `n` is a positive integer, `t` is an integer in the range [1, 20]):
  `./task2 n t`

d) On an Euler *compute node*:

- Run `task2` for `n` $= 10^7$, and `t` $= 1, 2, \cdots, 20$. Run `task2_pure_omp` for `n` $= 2 \times 10^7$, and `t` $= 1, 2, \cdots, 20$. Generate a plot called `task2.pdf` that includes the run time of the reduction process for the two programs vs. `t` in linear–linear scale.
- Choose a value `t` from the plot `task2.pdf` where both OpenMP+MPI and pure OpenMP can achieve good performance. Use this `t` value to run `task2` and `task2_pure_omp` for `n` $= 2, 2^2, ..., 2^{26}$ (this `n` is used for OpenMP+MPI). Note that you should control the input arguments such that array size for pure OpenMP is always two times the array size for OpenMP+MPI (so in total, they process the same number of elements, hence an apple-to-apple comparison). Generate a plot called `task2_comp.pdf` that includes the run time of the reduction process for the two programs vs. `n` in log–log scale.
- When does one method outperform the other? Which method would you choose for a smaller size array (i.e., several kilobytes) and which method would you use to reduce a very large array (i.e., several gigabytes), and why?