ME 759
High Performance Computing for Engineering Applications
Assignment 10
Due Friday 4/10/2020 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). Also, all plots should be submitted in Canvas. All *source files* should be submitted in the `HW10` 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. 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 ME759-2020 repo.

1. In this task, you will explore the optimizations using ILP (instruction level parallelism) based on the code examples given in Lecture 26 slides 5-35. 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 that each (either represents the baseline, or) uses a different technique to achieve ILP as the following:
      - `optimize1` will be the same as `combine4` function in slide 9.
      - `optimize2` will be the same as `unroll2a_combine` function in slide 20.
      - `optimize3` will be the same as `unroll2aa_combine` function in slide 22.
      - `optimize4` will be the same as `unroll2a_combine` function in slide 25.
      - `optimize5` will be similar to `optimize4`, but with $K = 3$ and $L = 3$, where K and L are the parameters defined in slide 28 and 29.

   b) Write a program `task1.cpp` that will accomplish the following:
      - Create and fill with `data_t` type numbers however you like a `vec v` of length `n` where `n` is the first command line argument, see below.
      - Call your `optimizeX` functions to get the results 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: `g++ task1.cpp optimize.cpp -Wall -O3 -o task1 -fno-tree-vectorize`
      - Run (where `n` is a positive integer):
        `./task1 n`
      - Example expected output:
        ```
        3125 //from optimize1
        0.706 //from optimize1
        3125 //from optimize2
        0.710 //from optimize2
        3125 //from optimize3
        0.353 //from optimize3
        3125 //from optimize4
        0.354 //from optimize4
        3125 //from optimize5
        0.236 //from optimize5
        ```

   c) On an Euler *compute node*:

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

- Run `task1` for value `n` = 10^6, with the settings of `data_t`, `OP`, and `IDENT`, and the naming of pdf files referring to Table 1. Each pdf should plot the time taken by all five of your `optimizeX` functions and one additional data point from SIMD version of `optimize1`[1] vs. `X` in linear-linear scale, where `X` = `1,...,6`. Run the `optimizeX` function for 10 times and use the average time for plotting.
- Note for `optimize.h` file: You can change the definition of macros in `optimize.h` file 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.

[1]data point `X=6` should come from the result of `optimize1` when compiled with command:

```
g++ task1.cpp optimize.cpp -Wall -O3 -o task1 -march=native -fopt-info-vec
```

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

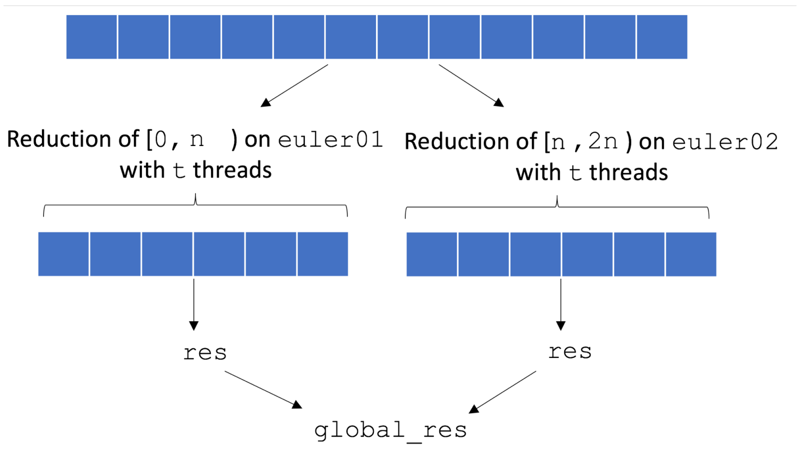
Reduction of [0, n) on euler01
with t threads
Reduction of [n , 2n) on euler02
with t threads
res
res
global_res

Figure 1: Schematic for the execution of reduction program.

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

b) Your program `task2.cpp` should accomplish the following:
   - Create and fill with `float`-type numbers however you like an array `arr` of length `n`, where `n` is the first command line argument, see below. Note that `n` is half of the length of the array that we are doing reduction on.
   - Initialize necessary variables for MPI environment.
   - Set the number of OpenMP threads as `t`, where `t` is the second command line argument, see below.
   - 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 (where `n` is a positive integer, `t` is an integer in the range [1, 20]):
     `mpirun -np 2 --bind-to none ./task2 n t`
   - Example expected output:
     ```
     3562.7
     0.352
     ```

c) On an Euler *compute node*:
   - Run `task2` for `n` $= 10^6$, and `t` $= 1, 2, \cdots, 20$. Generate a plot called `task2.pdf` that includes the run time of your program (the second output of your program) vs. `t` in linear-linear scale.
   - (**Optional**, extra credit 10 points) Compare the timing you received from two MPI processes running on two nodes with pure OpenMP implementation that runs

[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 from 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/openmpi`. Some notes about **optional** inspection into the execution if you are interested in understanding the performance better: You can compile with `gcc/9.2.0` (`module load gcc/9.2.0`), and add `export OMP_DISPLAY_AFFINITY=true` to check the mapping between OpenMP threads and the physical cores.

on one node. Make a plot of run time vs. `t` in linear-linear scale with these two patterns in `task2_op.pdf`. Submit your code for timing the pure OpenMP implementation as `task2_op.cpp` that should take the input arguments in the same way as `task2.cpp`. Note that here `n` should be 2×10^6 to compare with previous results. Discuss the differences between the two and the optimal choice of achieving parallelism/reducing run time for arrays of different sizes.