DUE DATE: **By the end of your lab session**

CSE-443/543: High Performance Computing

Exercise #11

Max Points: 20

You should save/rename this document using the naming convention MUid.docx (example: `raodm.docx`).

Objective: The objective of this exercise is to:

- Review basic of parallelizing a C++ program with OpenMP
- Learn running OpenMP programs on the cluster using SLURM scripts

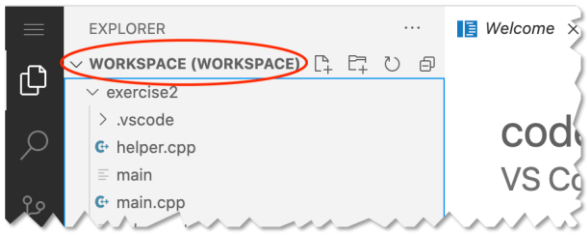
Fill in answers to all of the questions. For almost all the questions you can simply copy-paste appropriate text from the shell/output window into this document. You may discuss the questions with your instructor.

Name: **Maciej Wozniak**

Task #1: Setting up NetBeans project

Estimated time to complete: 5 minutes

1. Log into OSC's OnDemand portal via https://ondemand.osc.edu/. Login with your OSC id and password.
2. Startup a VS-Code server and connect to VS-Code. Ensure you switch to your workspace. Your VS-Code window should appear as shown in the adjacent screenshot.



3. Next, create a new VS-Code project in the following manner:
   a. Start a new terminal in VS-Code
   b. In the VS-Code terminal use the following commands:

```
$ # First change to your workspace directory
$ cd ~/cse443
$ # Use ls to check if workspace.code-workspace file is in pwd
$ # Next copy the basic template for a C++ project
$ cp -r /fs/ess/PMIU0184/cse443/templates/openmp exercise11
$ # Copy the starter code for this exercise
$ cp /fs/ess/PMIU0184/cse443/exercises/exercise11/* exercise11
```

4. Add the newly created project to your VS-Code workspace to ease studying the source code.

5. Compile the program and ensure it compiles successfully in `NetBeans` first.

6. Next, review the supplied `ex11_slurm.sh` script. This is a pretty standard job script, except for the setting of the number of OpenMP threads. Pay attention to that detail.

7. Finally practice compiling the program from the command line as shown below. Note that the command-line is very similar to the standard one used, except for the addition of the `-fopenmp` flag (to engage OpenMP -- even though the program does not use OpenMP it still includes `omp.h`):

```
g++ -g -Wall -std=c++17 -O3 -fopenmp main.cpp -o exercise11
```

8. Measure and record the runtime of the serial version of the program using the supplied `qsub_ex11.sh` script in the following manner:

```
$ sbatch --tasks-per-node=1 ex11_slurm.sh
```

   Once the job is complete (takes < 2 minutes of runtime once the job starts), view the output file to ensure the output from the program is:

```
Total number of factors: 1903
```

Task #2: Parallelize the program using OpenMP

Estimated time to complete: 15 minutes

Background: Data parallel applications are essentially synchronization free (aka embarrassingly parallel) applications that are easy to parallelize using multiple threads. In a data parallel application, each thread operates on an independent sub-set of data. Many useful and widely used applications in data processing, image processing, etc. fall into this category.

Exercise: The objective of this part of the exercise is to convert the given C++ program into a suitable data parallel version, in the following manner:

1. Revise the `getFactors` method to operate as in a data parallel fashion to use multiple threads to compute the factors of the list of numbers in `numList`. This is merely a copy-paste from lecture slides -- but ensure that you understand how the example works.

2. Now submit a parallel job using 1 compute node and 4 cores using the supplied `qsub_ex9.sh` script in the following manner:

```
$ sbatch --tasks-per-node=4 ex11_slurm.sh
```

   Once the job is complete (takes < 2 minutes of runtime once the job starts), view the output file to ensure the output is exactly the same as the serial version of the program

3. Submit separate jobs to measure runtime for 2 and 6 cores by suitably modifying the above `sbatch` command.

4. Once the jobs have completed, use the timing information from the jobs to complete the table below with the average elapsed time and average %CPU when using different number of threads. Recollect that Speedup *S* is defined as: *S* = Ts ÷ Tp, where Ts is serial runtime and Tp is parallel runtime. Note that in theory, under ideal conditions, given *n* threads it is desirable to attain a speedup of *n*. Consequently, the theoretical runtime with *n* threads is computed using the formula $T_{THEORY} = Ts \div n$

Threads (n)	**Theoretical/ Expected elapsed time (sec)** $T_{THEORY} = Ts \div n$	**Avg. Observed User Time (sec)**	**Avg. Observed Elapsed time (sec)**	**Avg. Observed %CPU**	**Observed Speedup** $S = Ts \div Tp$
1 (**Ts**)	22.02	22.1	22.01	99%	1.0×
2	11.01	31.09	15.76	197%	1.4
4	5.5	31.02	8.16	380%	2.75
6	3.66	31.66	6.63	479%	3.32

Validation of Observations:

- The %CPU should be approximately 100**n*, where *n* is the number of threads used. This may not be the case for 6-cores because there isn't sufficient work.

- Average user time (which is the total time that would be required to run the same set of instructions in the program is independent of number of cores used) should be about the same, but with a slight increase as the number of threads is increased. The slight increase is due to additional operations necessary to create, manage, and destroy threads.

- The Elapsed time (or wall-clock time or runtime) should decrease proportionally to the number of threads used and the %CPU allocated by the OS for the job/process.

- In this specific example (that does not have much synchronization), the observed speedup should be proportional to the number of threads used (and the %CPU allocated by the OS for the job/process) except for the 6-core case. Note that synchronization overheads will degrade speed-up.

Submit files to Canvas

Upload the following files to Canvas:

1. The source file that you modified in this exercise.
2. Upload this MS-Word document (duly filled with the necessary information) saved as PDF using the naming convention **`MUid.pdf`**.