DUE BY: **By the end of your lab session.**

High Performance Computing
Exercise #14
Max Points: 20

Objective: The objective of this exercise is to:
- Develop an OpenMP-based program
- Explore the effectiveness of different scheduling strategies for `parallel for`.

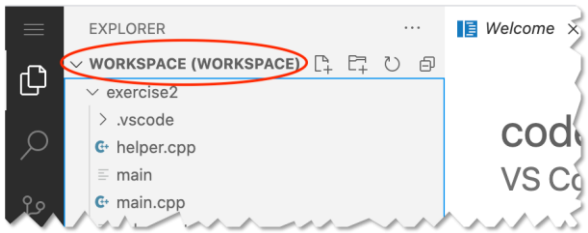
Submission: Save this MS-Word document using the naming convention *MUid*`_Exercise14.docx` prior to proceeding with this exercise. Upload the following at the end of the lab exercise:
1. This MS-Word document saved as a PDF file using the file naming convention *MUid*`_Exercise14.docx`.
2. Program developed in part #1 of the exercise.

You may discuss the questions with your instructor.

Name: **Maciej Wozniak**

Part #0: Setting up VS-Code 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 exercise14
$ # Copy the starter code for this exercise
$ cp /fs/ess/PMIU0184/cse443/exercises/exercise14/* exercise14
```

DUE BY: **By the end of your lab session.**

Part #1: Simplify a program using OpenMP-`for` [10 points]

Estimated time to complete: 20 minutes

Background: OpenMP provides a convenient mechanism to readily parallelize a given `for`-loop in a C/C++ program using the `#pragma omp parallel for` construct. Recollect that OpenMP essentially uses multiple threads as the underlying operating system level primitive for enabling multiprocessing expressing using the Fork-Join programming model. Accordingly, the OpenMP `parallel for` directive automatically splits the iterations of the loop between a given numbers of threads. Neither the number of iterations nor the number of threads need be known at compile time. The OpenMP runtime dynamically determines the necessary information and accomplishes parallelization.

Exercise: The objective of this part of the exercise is to simplify a given program using OpenMP-for in the following manner:

1. Add the newly created project to your VS-Code workspace to ease studying the source code.
2. Study the method `findNumberWithMaxPrimeFactorCount`. This method returns a `std::pair` as result which contains the number that has the highest number of prime factors. For example, given the numbers `{5, 10, 11}`, this method returns `<10, 2>` as 10 has 2 prime factors, namely 2 and 5.
3. Now streamline/simplify this method by appropriately modifying it to use OpenMP for construct. Yes you will need to add a `#pragma omp for` directive before the for-loop and suitably simplify/clean-up the source code.

Testing

First do a functional testing from VS-Code terminal as shown below. Note: The run time of the program is just 1 or 2 seconds and if your find your program running longer than 5 seconds you need to abort/stop it.

```
$ OMP_NUM_THREADS=2 ./exercise14 20
```

Expected output:

The expected 1 line output from the program is show below. The output should be the same immaterial of the number of threads used.

```
Number with most number of prime factors: 1012484, prime factor count = 4
```

1. Copy-paste the complete output from two runs using 1 thread in the spaces shown below.

Outputs from 2 different runs with OMP_NUM_THREADS=1
Number with most number of prime factors: 1012484, prime factor count = 4
Number with most number of prime factors: 1012484, prime factor count = 4

2. Copy-paste the output from two runs using 2 threads in the spaces shown below.

Outputs from 2 different runs with OMP_NUM_THREADS=2
`Number with most number of prime factors: 1012484, prime factor count = 4`
`Number with most number of prime factors: 1012484, prime factor count = 4`

3. Copy-paste the output from two runs using 4 threads in the spaces shown below.

Outputs from 2 different runs with OMP_NUM_THREADS=4
`Number with most number of prime factors: 1012484, prime factor count = 4`
`Number with most number of prime factors: 1012484, prime factor count = 4`

Output verification:
The output from your program should be exactly the same immaterial of the number of threads used. If the outputs are different then there is something incorrect about your program and you need to resolve this issue prior to proceeding with the remainder of this exercise. If you are unsure about your solution request your instructor to verify the solution prior to proceeding with rest of this exercise.

Part #2: Explore effect of scheduling options in `parallel for` [10 points]

Estimated time to complete: 20 minutes

Background: The OpenMP `parallel for` directive automatically splits the iterations of the loop between a given numbers of threads. Neither the number of iterations nor the number of threads need be known at compile time. The OpenMP runtime dynamically determines the necessary information and accomplishes parallelization. However, depending on the nature of data processing, the default scheduling mechanism of OpenMP may not provide the best performance and different scheduling strategies need to be explored to identify appropriate settings for a given problem.

Note: In this exercise the numbers are sorted in `main` function. Consequently, some threads get small numbers to work with and finish quickly while thread(s) with large numbers take longer time to finish causing the overall program to run slower -- that is the work is not properly load balanced causing sub-par performance.

Exercise: The objective of this exercise is to explore the effectiveness of various scheduling (namely: `static` and `dynamic`) strategies provided by OpenMP parallel for. The procedure for this experiment is as follows:

DUE BY: **By the end of your lab session.**

1. Modify the program developed in Part #1 to include the `runtime` scheduling option for OpenMP-for so that different scheduling strategies can be easily explored without having to recompile the program. Example from lecture slides shown below:

```
#pragma omp parallel for schedule(runtime)
```

2. Compile the program from a terminal to enable optimizations as shown below:

```
$ g++ -g -Wall -std=c++17 -O3 -fopenmp exercise14.cpp -o exercise14
```

3. Edit the supplied `ex14_slurm.sh` script to specify the type of scheduling to `static` by setting the environment variable `OMP_SCHEDULE` in the supplied `ex14_slurm.sh` script.

4. Submit 1 PBS job to run using 4 threads and record data in table below:

```
$ sbatch --tasks-per-node=1 ex14_slurm.sh
$ sbatch --tasks-per-node=2 ex14_slurm.sh
$ sbatch --tasks-per-node=4 ex14_slurm.sh
```

Observations for `static` scheduling:

Num	#Thr	Average of 5 observations	
		Elap. Time (sec)	%CPU
10000	1	39.5581	99.7
	2	39.2857	99.8
	4	19.8506	199.8

5. Next, edit the supplied `ex14_slurm.sh` script and specify the type of scheduling to **`dynamic`** by setting the environment variable `OMP_SCHEDULE` in the supplied `ex14_slurm.sh` script.

6. Submit 1 PBS job to run using 4 threads and record data in table below:

```
$ qsub -l "nodes=1:ppn=1" ex14_slurm.sh
$ qsub -l "nodes=1:ppn=2" ex14_slurm.sh
$ qsub -l "nodes=1:ppn=4" ex14_slurm.sh
```

Observations for `dynamic` scheduling:

Num	#Thr	Average of 5 observations	
		Elap. Time (sec)	%CPU
10000	1	39.58001	99.7%
	2	39.5492	99.9%
	4	19.830387137	199.6

DUE BY:	**By the end of your lab session.**

Inferences:
Using the above data draw inferences about the performance differences between `static` and `dynamic` scheduling. Discuss if-and-why one of the scheduling strategies is providing better performance than the other (based on the application).

In my case changing scheduling to dynamic, did not have any effect. Neither on %CPU or elapse time

```
export OMP_SCHEDULE=dynamic
```

What's interesting time using ppn 1 and 2 is exactly the same (+- 2 sec) but with using 4 ppn, it drops almost by half. Another thing is that even though we specify ppn=2, our job is still using one.

Part 3: Submit files to Canvas

Upload just the following files to Canvas:

1. This MS-Word document saved with the convention *`MUid_Exercise14`*`.docx`.
2. Program you developed in Part #1 of this exercise.