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

CSE-443/543: High Performance Computing
Exercise #12
Max Points: 20

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

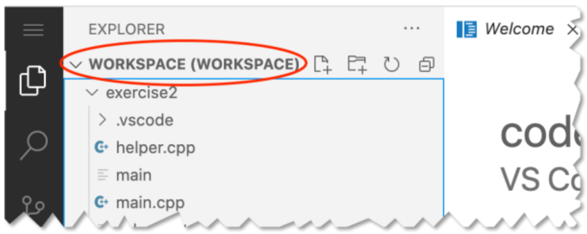
Objective: The objective of this exercise is to:

- Review basic of parallelizing a C++ program with OpenMP
- Understand issue with race conditions arising due to shared variables
- Fixing race conditions using critical sections
- Understand overhead of race conditions

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 are expected refer to LinuxEnvironment.pdf document available in Handouts folder off Niihka. 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 exercise12
$ # Copy the starter code for this exercise
$ cp /fs/ess/PMIU0184/cse443/exercises/exercise12/* exercise12
```

4. Add the newly created project to your VS-Code workspace to ease studying the source code.
5. Briefly review the program, specifically the `countBits` method. The `countBits` method is computing sum of number of bits in a given list of numbers. Multiple threads

share the `bitcount` variable in this method. Consequently reading/writing or using/changing the shared `bitcount` variable (without a critical section) results in race conditions in the program.

6. Compile the program and ensure it compiles. You will use the executable generated by `VSCode` for running the program in the next steps.

7. Next, review the supplied `ex12_slurm.sh`. This is a pretty standard SLURM job script, except for the setting of the number of OpenMP threads. Pay attention to that detail. In addition, ensure that the name of the executable in the script is consistent with your executable name.

8. Run the supplied the program using the supplied SLURM script in the following manner:
   ```
   $ sbatch --tasks-per-node=1 ex12_slurm.sh
   ```

   Once the job is complete (takes < 1 minute of runtime once the job starts), view the output file to ensure the output from each of the 5 runs of the program is (since this run uses only 1 thread, there cannot be any race conditions):
   ```
    Bit count: 632223552
   ```

Task #2: Reproducing race conditions

Estimated time to complete: 10 minutes

Background: Race conditions are semantic errors (or bugs) in a multithreaded program. Race conditions occur due to incorrect or inconsistent use of shared variables in a program. Similar to most semantic errors, race conditions manifest themselves in the form of incorrect/inconsistent outputs each time the program is run. It is important to that in this context shared variables refer to primitive or non-primitive data types on which final operations are performed. So even though a `std::vector` may be shared between multiple threads, if each thread modifies/uses a different entry in the `std::vector` then a race condition does not occur.

Exercise: The objective of this part of the exercise is to observe how race conditions typically manifest themselves:

1. Bear in mind that race conditions manifest only when multiple threads are used. Run the supplied the program using multiple threads in the following manner:
   ```
   $ sbatch --tasks-per-node=4 ex12_slurm.sh
   ```

2. Once the job is complete (takes < 1 minute of runtime once the job starts), view the output file and copy-paste the 5 1-line outputs from the program in the space below:
   ```
    Bit count: 162636976
   Bit count: 158055888
   Bit count: 165974800
   Bit count: 165974800
   ```

```
Bit count: 162636976
```

Important observations to note:

- Unlike the 1 thread case, you should notice that the %CPU is more than 100%. This is indicative that multiple threads ran to produce the output. Note that in some cases even though may threads may run the %CPU can be < 100%
- You should observe that the outputs from the program are not consistent with the 1 threaded output. This is due to race conditions in the program! This is how race conditions manifest -- each time the program is run you may get a different/incorrect result.

Task #3: Resolving race conditions using a critical section

Estimated time to complete: 15 minutes

Background: There are 2 common strategies for resolving race conditions:

1. Remove sharing of variables by using independent values for each thread as in a data parallel application. This is typically the preferred way to resolve race conditions.
2. Use critical sections to serialize access to shared variables. Critical sections are essentially result in a single-threaded part in a program and therefore degrade performance. Hence, they must be used sparingly in program to ensure good performance and efficiency.

Exercise: The objective of this part of the exercise is to eliminate a race condition using a critical section:

1. Add a suitable critical section to your program as shown below:
```
#pragma omp critical
{
    bitcount += popcount(numList[idx]);
}
```
2. Bear in mind that race conditions manifest only when multiple threads are used. Run the supplied the program using multiple threads in the following manner:
```
$ sbatch --tasks-per-node=4 ex12_slurm.sh
```
3. Once the job is complete (takes < 1 minute of runtime once the job starts), view the output file and copy-paste the 5 1-line outputs from the program in the space below:
```
 Bit count: 632223552
Bit count: 632223552
Bit count: 632223552
Bit count: 632223552
Bit count: 632223552

CPU% for each of them is ~370% so we are using multiple cores.
```

Important observations to note:

- Unlike the 1 thread case, you should notice that the %CPU is more than 100%. This is indicative that multiple threads ran to produce the output.
- You should observe that the outputs from all 5 runs of the program are now consistent with the 1 threaded output! This indicates that the critical section successfully resolved the race condition.

! Note that performance of the program using a critical section is terrible! Even though the program used 4 threads the elapsed time of the program has not improved (in fact elapsed time will degrade). This is a clear indication that the program is not efficiently multithreaded.

Think about how this program can be restructured as a data parallel application (by making `bitcount` into a `std::vector`) to improve performance (this could be an Exam #2 question)

Submit files to Canvas

Upload the following files to Canvas:

1. Upload this MS-Word document (duly filled with the necessary information) saved as PDF using the naming convention **`MUid.pdf`**.
2. Upload your version of `exercise12.cpp` that you revised in this exercise.