**CSE-443/543: High Performance Computing**

**Lab #06**

Max Points: 50

**Objective**: The objective of this exercise is to:

* Build experience with compiling code at the command line and constructing batch shell scripts to run a parallel program
* Build experience with compiling and running parallel code using OpenMP.
* Build with the concepts of parallel program performance discussed in class
* Gain experience with using distinct (explicit) tasks to parallelize a program using OpenMP

Fill in answers to all of the questions in this document and the corresponding spreadsheet. You may discuss the questions with your instructor. Note: The total execution time for all of the timing runs in this lab should be less than 15 minutes.

# Background

In this lab you are given the source code for a serial C++ program called workhorse.cpp. You will be modifying this code to run in parallel using explicit tasks. Recall from [Session 15](https://docs.google.com/presentation/d/1joljndt2xdQdMX3y2pZL8CUW_JahAWKmyVh-jAiZ36U/edit?usp=sharing) that explicit tasks make use of the #pragma omp task directive to define the distinct tasks and that a #pragma omp taskwait directive can be used to synchronize the threads so they all wait until each task has completed. This is important when the tasks take significantly different amounts of time to run as will be the case with workhorse.cpp.

One thing to note about OpenMP directives. If you issue a directive before a method or function, the directive will be in force inside the method. So when parallelizing workhorse\_explicit.cpp you can put the #pragma omp parallel and #pragma omp single directives in the main function right before you invoke workhorse().

The workhorse(long int n) function in workhorse.cpp consists of 16 for loops, some of which depend on the output of other loops. When constructing explicit tasks for this program you should **not** try to parallelize these loops with #pragma parallel for directives, **nor** should you combine the loops. Keep them as separate loops, grouping dependent pairs of loops into explicit tasks and synchronizing the tasks at the end of the function before returning the result and exiting. **You should end up with 8 explicit tasks if you are doing this correctly.**

The workhorse program takes a single command line argument which is the length of the for loops in the workhorse() function. You should make sure that the serial code is working correctly before proceeding with parallelizing it. As a test case, if you run workhorse with loops of length 1,000,000, the program should finish in less than three seconds and return a value of 382604.567386. You should use this as the test case for while developing your parallel code. **Note that for your final production timing runs you will use loops of length 40,000,000.**

After you parallelize this code you will perform timing runs for different numbers of threads (1 to 8), and you will be determining the parallel speedup ratio Sn, the parallel efficiency En, and the parallel cost Pn in this lab. Refer to the [Session 12](https://docs.google.com/presentation/d/1TfLybeHFBKUirSD97Zrtkv8KE24tz7Tf5FQOhV1u1Zw/edit?usp=sharing) presentation if you need a refresher in how to calculate these quantities.

# Experiment

1. Download the file workhorse.cpp from Canvas.
2. Create a serial executable called workhorse by compiling workhorse.cpp with our standard options: -O3 -std=c++17 -Wall
3. Run workhorse for loops of 1,000,000 to ensure that you are getting the correct answer 382604.567386. This run should take less than 3 seconds.

./workhorse 1000000

1. Run and time your workhorse program twice for loops of length 40,000,000. Enter your timing data (Elapsed, and CPU = User + System, expressed in seconds without a ‘:’) into the Google spreadsheet. You should know how to do this by now.
2. Make a copy of workhorse.cpp called workhorse\_explicit.cpp.
3. Modify workhorse\_explicit.cpp to run in parallel with OpenMP by creating explicit tasks inside the workhorse() function.
   1. Use these compiler options: -O3 -std=c++17 -Wall -fopenmp
   2. Use the test case of loops of length 1,000,000 to ensure that you are getting the correct answer 382604.567386.
4. When you have successfully parallelized the code, run workhorse\_explicit for 1 through 8 threads, performing two runs for each value of OMP\_NUM\_THREADS. Enter your timing data (Elapsed, and CPU = User + System, expressed in seconds without a ‘:’) into the Google spreadsheet. Remember to run it for loops of length 40,000,000 for these production timing runs

./workhorse\_explicit 40000000

If your code is working correctly you should get the result 15303951.9986 for loops of length 40,000,000. Your single-threaded execution time should be a bit more than 100 seconds when you run workhorse or workhorse\_explicit (with one thread) for loops of length 40,000,000.

# Apparatus (platform for experiment)

The first experiment documented in this report was conducted on the following platform (fill in the Details column using information determined in your shell script):

|  |  |
| --- | --- |
| Component | Details |
| CPU Model | Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz |
| Main Memory (RAM) size | MemTotal: 8056076 kB |

# Observations – Lab06

Enter the raw timing data from your runs into the Google spreadsheet.

# Analysis – Lab06

Compute the requested quantities and enter them into the spreadsheet provided.

# Submit files to Canvas

When you complete the lab, download this document from Google Drive as a Microsoft Word (.docx) file with the naming convention Lab06 - MUid.docx (example: Lab06 - ferrenam.docx).  You should save the corresponding Google spreadsheet file as a Microsoft Excel (.xlsx) file with the naming convention Lab06 – Results MUid.xlsx (example: Lab06 - Results - ferrenam.xlsx)

Then, submit the following files to Canvas:

1. The Microsoft Word file you downloaded from Google Drive.
2. The Microsoft Excel file you downloaded from Google Drive.
3. Your workhorse\_explicit.cpp source code.