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

**Lab #07**

Max Points: 56

**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 using OpenMP to parallelize for loops in a program
* Build your understanding of microbenchmarks, in particular comparing the performance of two different methods of parallelizing a program (explicit tasks versus parallel for)

Fill in answers to all of the questions in this document and the corresponding spreadsheet. You may discuss the questions with your instructor and the TA.

# Background

In Lab 06, you were given the source code for a serial C++ program called workhorse.cpp and you used the concept of explicit tasks in OpenMP to parallelize it, creating a program called workhorse\_explicit.cpp. In this Lab, you will start with similar serial code but you will use the OpenMP API to parallelize for loops in the workhorse method.

In [Session 13](https://docs.google.com/presentation/d/1RT1xEPLP39Twt0HNJdAPOsE-qFT61eaoYi_pWSGS5NM/edit?usp=sharing) and [Session 14](https://docs.google.com/presentation/d/1SpCuXU5fMxLFpYcxQRL8hr1dZIHOozertCkE7O7qsNA/edit?usp=sharing) we talked about how to parallelize for loops in programs. You may want to review the slides and the prep work for these sessions to refresh your memory. In particular, you should know how to identify and eliminate race conditions inside a for loop.

As in Lab 06, if you run the program with a command-line argument of 1,000,000 (without the commas, of course) you should get the result 382604.567386. Use this sized loop for your development and debugging. When the code is working correctly, do your timing runs with a command-line argument of 40,000,000 (again, without the commas). You should get the result 15303951.9986 as you did in Lab 06.

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

./workhorse\_for 1000000

1. Perform two timing runs of workhorse\_for using a command line argument of 40000000 and capturing the elapsed time and total CPU (system + user), expressed in seconds without a ‘:’. Enter these results in your Lab07 spreadsheet.
2. Make a copy of workhorse\_for.cpp called workhorse\_for\_omp.cpp.
3. Modify workhorse\_for\_omp.cpp to run in parallel with OpenMP by parallelizing the for loops 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\_for\_omp for 1 through 8 threads, performing two runs for each value of OMP\_NUM\_THREADS. Enter your timing data (CPU = User + System, and Elapsed, expressed in seconds without a ‘:’) into the Lab07 Google spreadsheet. Remember to run it for loops of length 40,000,000 for these production timing runs

./workhorse\_for 40000000

If your code is working correctly you should get the result 15303951.9986 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 |  |
| Main Memory (RAM) size |  |

# Observations – Lab07

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

# Analysis – Lab07

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

# Discussion – Lab07

You should have found that the serial program workhorse\_for took less time to run than the workhorse program from Lab06. Explain why.

|  |
| --- |
| The primary reason workhorse\_for took less time is because of the number of for loops it’s using. Workhorse uses a loop for every single variable that is being incremented, which is not efficient in the slightest. workhorse\_for uses only two loops, one for all the variables that do not depend on each other first, then one for the variables that depend on the calculations from the previous loop. |

In the space below you should discuss your results, including at least the items below. A fully correct answer will include both quantitative measures (absolute times and parallel performance metrics) as well as a qualitative description of them.

* Comparing the parallel performance measures from explicit tasks (Lab 06) and parallelizing the for loop (Lab 07). Do you see any anomalous behavior as you increase the number of threads? Explain the difference in performance between the two parallelization approaches used in Labs 06 and 07. (I.e. why do you think they’re different?)

|  |
| --- |
| The explicit tasks performance was almost entirely worse across the board than the parallel for performance. Parallel speedup, parallel efficiency, and parallel cost were all significantly better for the parallel for loop. The only instance where this was not the case was when the parallel for was used with 5 threads. It’s parallel speedup and parallel efficiency were worse than when used with 4 threads, and worse than the explicit task performance with the same number of threads, although this did turn around when 6 threads were used. The difference in performance here can be partly attributed to the difference in the base code, as well as the complexity of the multi-threading that is being done. With the explicit tasks code, we are giving it much more to work on in terms of how many parts of the code can be worked on by different threads. In the parallel for code, it is simplified a bit, with only two chunks of code that can be worked on by a thread at any given time. |

# 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 Lab07 - MUid.docx (example: Lab07 - ferrenam.docx).  You should save the corresponding Google spreadsheet file as a Microsoft Excel (.xlsx) file with the naming convention Lab07 – Results MUid.xlsx (example: Lab07 - 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\_for\_omp.cpp source code.