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

**Lab #08**

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 07, 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 the same serial code but you will use the OpenMP API to parallelize for loops in the workhorse method.

In [Session 13](https://drive.google.com/open?id=1Amm--1DeYO-HlrFe6os2egSfEupog6YUV2F9cZpPJ-Q) and [Session 14](https://drive.google.com/open?id=1FlzpiInWbAlK3VPbM2-AfjJQOZOcqIaqoDczkWNXjus) 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.

Note that it is possible to parallelize each for loop individually, but as we saw in earlier sessions, a for loop has overhead that results in branching which can degrade the performance of the program. (If you need a refresher, please refer to the [Session 08](https://drive.google.com/open?id=1vPsOZcI6_Du-rYDB7AF94qC1j5JRCaMSJTAOGgFw3eY) presentation.) To minimize branching, you should use the minimum number of loops possible. In this case, because of data dependencies between the loops, you will need to have two of them. Consolidating the loops has a measurable difference, even for the serial code. In my testing I found that the code with consolidated loops, and therefore fewer branches, ran 15 seconds faster than the original serial code.

As in Lab 07, if you run the program with a command-line argument of 1,000,000 (without the commas, of course) you should get the result 382600.8746. 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 15303951.59 as you did in Lab 07.

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://drive.google.com/open?id=1noK7mMx14QScNMZ7DtF-wdjsF5g6YFvMOoNkeM7mqQY) presentation if you need a refresher in how to calculate these quantities.

# Experiment

1. Download the file workhorse.cpp from Canvas, or use your copy from last session if you didn’t modify it.
2. As in Lab 07, create a serial executable called workhorse by compiling workhorse.cpp with our standard options: -O3 -std=c++14 -Wall
3. Run workhorse for loops of 1,000,000 to ensure that you are getting the correct answer 382600.8746.

./workhorse 1000000

1. Modify the workhorse.cpp program to consolidate the for loops in the workhorse method. You should end up with two loops. You may want to make a backup copy of the original so you can refer back to it.
2. Repeat steps 2 and 3 above, checking that you are getting the same result as with the original workhorse.cpp program.
3. Perform two timing runs of the modified workhorse.cpp program, 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 Lab08 spreadsheet.
4. Make a copy of workhorse.cpp called workhorse\_for.cpp.
5. Modify workhorse\_for.cpp to run in parallel with OpenMP by parallelizing the for loops inside the workhorse() method.
   1. Use these compiler options: -O3 -std=c++14 -Wall -fopenmp
   2. Use the test case of loops of length 1,000,000 to ensure that you are getting the correct answer 382600.8746.
6. When you have successfully parallelized the code, run workhorse\_for 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 Lab08 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 15303951.59 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 | 8071532 kB |

# Observations – Lab08

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

# Analysis – Lab08

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

# Discussion – Lab08

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 07) and parallelizing the for loop (Lab 08). 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 07 and 08. (I.e. why do you think they’re different?)
* Comparing the serial and 1-thread performance of the original workhorse, workhorse\_explicit, workhorse with the for loops consolidated, and workhorse\_for. Does anything seem out of the ordinary?

|  |
| --- |
| 1. Parallelizing the for loop, as is the case for Lab08, compared to creating and running tasks, as is the case for Lab07, demonstrates much different results. Parallelizing the for loop lead to a much higher parallel speedup value, particularly as the number of threads increased. As a result, the parallel efficiency also went up, with a value of close to 0.47 for the “task method”, and 0.61 for the “for loop” method for the 8-threaded test. One particularly interesting note is that both methods had a parallel cost of about 1.61 for the 8-threaded run. As to why this is the case, I assume the time the master thread spends checking task completion (as is the case for the taskwait clause). In addition to that time cost, I would also vouch that the master thread would have to spend quite a bit of time creating the tasks, designating memory for those tasks, and handling additional overhead for splitting each for loop among many different threads. 2. The 1-thread performance of the parallel workhorse\_for implementation is faster than than the serial version of the same code. Both the version of the task variant (workhorse and workhorse\_explicit) run almost the same. It seems strange that the 1-thread performance of workhorse\_for would beat out the serial counterpart by close to 10 seconds, since the task variant did not show this kind of difference. My best guess is that somehow, OpenMP does some small additional optimization in the background for cases where a user is doing two subsequent for loops using the results from one into the next. That seems outlandish, but I don’t know of any other reason why the explicitly 1-thread OMP program would run FASTER rather than slower unless there is an issue in the compiler. |

# 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 Lab08 - MUid.docx (example: Lab08 - ferrenam.docx).  You should save the corresponding Google spreadsheet file as a Microsoft Excel (.xlsx) file with the naming convention Lab08 – Results MUid.xlsx (example: Lab08 - 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.cpp source code.