CS450: Introduction to Parallel Programming

Assignment 5: Programming with OpenMP

Due: July 13th, 2022, 11:59 PM, MST

Preliminaries

You are expected to do your own work on all homework assignments. You may (and are encouraged to) engage in discussions with your classmates regarding the assignments, but specific details of a solution, including the solution itself, must always be your own work. See the academic dishonesty policy in the course syllabus.

Submission Instructions

You should turn in an electronic archive (.zip, .tar., .tgz, etc.). The archive must contain a single top-level directory called CS450_aX_NAME, where "NAME" is your NAU username and "X" is the assignment number (e.g., CS450_a5_ab1234). Inside that directory you should have all your code (no binaries and other compiled code) and requested files, named exactly as specified in the questions below. In the event that I cannot compile your code, you may (or may not) receive an e-mail from me shortly after the assignment deadline. This depends on the nature of the compilation errors. If you do not promptly reply to the e-mail then you may receive a 0 on some of the programming components of the assignment. Because I want to avoid compilation problems, it is crucial that you use the software described in Assignment 0. Assignments need to be turned in via BBLearn.

Turn in a single pdf document that outlines the results of each question. If you were not able to solve a problem, please provide a brief write up (and screenshots as appropriate) that describes what you tried and why you think it does not work (or why you think it should work). You must provide this brief write up for each programming question in the assignment.

This pdf should be independent of the source code archive, but feel free to include a copy in the top level of that archive as well.

Make sure that you configured your virtual machine to use multiple cores. Otherwise, you will not be able to see any speedup.

Overview

You just started your dream job as a software developer at NASA. A rather strange scientist asks you to parallelize a piece of code, but you do not know what the code does. You ask your boss (not the scientist) if the program has at least been optimized, and your boss replies that they do not know what "optimized code" means. Thus, you are stuck parallelizing the code as it is given to you. The code is `crazy_scientist.cpp`. Your job is to parallelize it as described in the questions below. NASA is counting on you, and you start to wonder if this is your dream job after all...

Grading

All questions are weighted equally.

Validation of All Questions

You are responsible for ensuring that your program is correct. There are no testing scripts for this assignment. You must determine how to ensure program correctness.

Table 1: The average response time in seconds for each question, and load imbalance for Q2-Q4. Below, t_1 and t_2 refer to thread 1 and thread 2, respectively. All response times have been averaged over 10 time trials, each using separate executions of the program.

Question	Total Time	Individual Thread Response Times and Load Imbalance
1	_____	N/A
2	_____	Time t_1: _____ Time t_2: _____ Load Imbalace: _____
3	_____	Time t_1: _____ Time t_2: _____ Load Imbalace: _____
4	_____	Time t_1: _____ Time t_2: _____ Load Imbalace: _____

Master Table

For all questions (Questions 1–4 below), fill out a master table with the information, formatting, and caption, as shown in Table 1.

Question 1

Using OpenMP, write a parallel version of this program, called `crazy_scientist_v1_NAME.cpp` so that 2 threads are used. In this question, make it so that the first thread computes the top part of the array (i.e., rows 0 to 24) and the second thread computes the bottom part of the array (i.e., rows 25 to 49).

Your program must report the overall execution time as an average of 10 trials by running your program 10 times, and manually averaging the results (hereafter referred to as time trials). Use the `omp_get_wtime()` function as follows:

```
#include <omp.h>

int main(int argc, char **argv) {

  double tstart=omp_get_wtime();
  ... //code to time
  double tend=omp_get_wtime();

  double elapsed = tend - tstart;

  printf("Elapsed time: %f seconds\n",elapsed);

 }
```

Example Output:

```
Some of the '.' are removed below:

g++ crazy_scientist_v1.cpp -o crazy_scientist_v1 -fopenmp
./crazy_scientist_v1

.....................................................
.....................................................
.....................................................
.....................................................
Elapsed time: 19.306 seconds
```

Question 2

Modify the program from Question #1, naming it `crazy_scientist_v2_NAME.cpp` so that it computes (and outputs) the load imbalance. The load imbalance is the absolute value of the difference between the completion times of the two threads. The execution time of each thread should also be printed.
Example Output:

```
Some of the '.' are removed below:

g++ crazy_scientist_v2.c -o crazy_scientist_v2 -fopenmp
./crazy_scientist_v2

..........................................................
..........................................................
..........................................................
..........................................................
Total Time (sanity check): 19.667371
Time Thread1: 2.390382
Time Thread2: 19.667208
Load imbalance: 17.276826
```

In the above run one thread ran for 19.67 s and the other for 2.39 s, for a high load imbalance of 17.28 s. The Total Time above timed everything just to double check that time times for each thread make sense. The total time should be a little bit more than the time of the thread that executes the longest.

Time trial your code 10 times and in the pdf file report on the the average load imbalance and the average execution time (i.e., the average execution time of the slowest thread).

Hint: This requires a little bit of creativity given the rigidity of OpenMP. Using separate `#pragma omp parallel` and `#pragma omp for` directives. The `nowait` option for `#pragma omp for` likely comes in handy.

Question 3

Modify the program, naming it `crazy_scientist_v3_NAME.cpp` so that both threads compute the rows of the matrix on demand. That is, when a thread completes processing a row, it starts processing the next available row.

Time trial your code 10 times and in your pdf file report on the average load imbalance and the average execution time. Any improvements compared to the version in Question #2? Explain.

Bonus Question 4

Create a new program called `crazy_scientist_v4_NAME.cpp` that implements Question #3 using **Pthreads**. Only `omp_get_wtime()` may be used from the OpenMP library. You may modify the program to use global variables. Also, after each thread finishes, it should output the number of iterations it processed.
Example Output:

```
Some of the '.' are removed below:
g++ crazy_scientist_v4.c -o crazy_scientist_v4 -pthread -fopenmp
..........................................................
..........................................................
..........................................................
..........................................................
```

```
Number of iterations for thread: 25
Number of iterations for thread: 25
Time Thread1: <time1>
Time Thread2: <time2>
Load imbalance: <|time1-time2|>
```

Time trial your code 10 times and in your pdf file report on the average load imbalance and the average execution time. Any improvements compared to Question #2 or Question #3? Explain.