ECE 565 - Fall 2018
Assignment #4
Parallel Programming with OpenMP

Important General Instructions

1. **This is an individual assignment. All work is expected to be your own.**
2. This assignment consists of 2 sub-problems. In each sub-problem, you will parallelize existing code using OpenMP and conduct performance experiments that will be written up into a report.
3. **Performance experiments:** Your performance experiments *must* be done using an 8-core VM (running Ubuntu Linux) that you will have access to for assignment #4 and #5. You should run your performance experiments when you have exclusive access to the VM. Instructions will be posted on Piazza for reserving exclusive access.
4. **Program development:** Your code development, testing, debugging, etc. may be performed either on your own machine, the Intel server that you were given access to previously (kraken.egr.duke.edu or leviathan.egr.duke.edu), or the 8-core Ubuntu VM if no other student has reserved exclusive access during that time.
5. Both the Intel servers and the 8-core Ubuntu VMs have an OpenMP compiler (GCC) as well as various development and profiling tools installed.
6. Note, as you work through these sub-problems, that **bugs in parallel code are often timing dependent,** i.e. incorrect parallel code may (quite often) result in correct execution due to the absence of certain timing conditions in which the bugs can manifest. Therefore, manual code correctness analysis is very important, and **your code will be graded based on code correctness rather than on output correctness from particular test runs**.
7. Please follow the spec carefully and turn in everything that is asked. **Be sure to start early and make steady progress.** Due to the need to share parallel machines, your experience and learning with this project will not be very satisfying if you wait until the last minute. You may not be able to do as good a job as you'd like at the last minute if the machines are all heavily occupied.
8. Parallel programming requires significant design effort although the amount of code you will write will not be extremely large. You should not underestimate the effort and time to get to the right solution. The best way to do well in this project is to start early.
9. As you work through the two sub-problems of this assignment you will create a writeup (named **report.pdf**). You will submit this report as well as modified code as described in the submission instructions. **Be sure to submit all source code files and report.pdf inside of a hw4.zip file exactly as described or points will be deducted.**

First Step – You should take care of this ASAP

The first step you should take is to set up your 8-core Ubuntu VM access. To gain access, you will need to create an SSH private-public key pair (if you do not already have one). Here are the steps to create these keys. **These steps should be done from the home directory of the machine you will use to 'ssh' to the 8-core Ubuntu VM.**

1. Create your SSH Public key by following the steps described here (skip this step if you already have an SSH Public key):

   https://git-scm.com/book/en/v2/Git-on-the-Server-Generating-Your-SSH-Public-Key

2. Send Brian an email with the contents of your Public key (you can just copy/paste the key into the text of the email).

3. Brian will email you back the machine name and user name you will use to access your 8-core Ubuntu VM. At this point, you will be able to ssh into the VM with no password required (as long as the private key is in your ~/.ssh/ directory). We'll have ~10 students on each VM that our class has access to.

Problem 1: Histogram

Download the histogram.tgz file and extract it with the following command: ‘tar xzvf histogram.tgz’. In the included source code, focus on histogram() function. In the loop nest inside that function, you will notice a statement that has a loop-carried dependence, depending on the content of the image’s pixels across different iterations:

```
histo[image->content[i][j]]++;
```

Your task is to parallelize this loop nest using OpenMP. Do not parallelize the outer ‘for m’ loop. That is only included so that the histogram generation is done a number of times. You should parallelize the ‘for i’ and/or ‘for j’ loops (you can experiment to see what parallelization works best). The test prints out histogram results which you can use to ensure your parallel program is correct (the ‘diff’ Linux command will be useful for this). The code also prints the runtime (in seconds) that you should report.

In order to remove the parallelization limitation imposed by the loop-carried dependence, implement you should implement 3 different approaches:

1. Use an array of locks. For help in how to declare and use OpenMP locks, refer to the OpenMP standards document (e.g. omp_init_lock(), omp_set_lock(), omp_unset_lock()). Put this version in a code file named histo_locks.c and edit the makefile to generate an additional executable named ‘histo_locks’.
2. Use atomic operation to protect the update to the histogram element, instead of anarray of locks. Before implementing your solution, read about #pragma omp atomic in the OpenMP standards document. Put this version in a code file named histo_atomic.c and edit the makefile to generate an additional executable named ‘histo_atomic.
3. Find your own creative solution that does not rely on atomic operations or locks. Put this version in a code file named histo_creative.c and edit the makefile to generate an additional executable named ‘histo_creative’.

For each of these 3 versions:

a) Run the program and verify your results against the sequential version.
b) Measure and record the execution time reported by the program when running with 2, 4, and 8 threads. Then, plot the speedup curve by taking the ratio of the sequential execution time to the parallel execution time.
c) Write your observations about which version performs best, and discuss why you think that is the case. Also, describe how your own creative solution removes the need for using locks and atomic operations while still producing correct results.

The histogram program is executed, for example, as “./histogram <input_file>”. There are two image input files included in the Histogram/ directory. The first one is uiuc.pgm, a small input image that can be used for testing your code. The second one is uiuc-large.pgm, a large input image that should be used for running your performance experiments. **Due to the size of uiuc-large.pgm, do *not* include it along with your final files that you submit.**

Part 2: AMG

Download the amgmk.tgz file and extract it with the following command: ‘tar xzf amgmk.tgz’. This code is a microkernel that represents important portions of an algebraic multigrid solver application. This microbenchmark exercises 3 computations: 1) compressed sparse row (CSR) matrix vector multiplication, 2) algebraic multigrid (AMG) mesh relaxation, and a simple a * X + Y vector operation.

The provided makefile will compile the baseline sequential code. The benchmark can be run with no arguments (i.e. ./AMGMk) after compilation. The output will print a measured Wall time for each of the 3 computations, labeled “MATVEC”, “Relax”, and “Axpy”. The benchmark also self-checks results. An “error” message will be printed for each computation where an incorrect result is produced.

Your task is to use OpenMP to parallelize the code for each of these 3 computations to improve performance as much as possible compared to the baseline sequential code. You may also alter / restructure the baseline C code if needed to implement your OpenMP parallelization.

You should include the following items in your report.pdf writeup: 1) A clear description of what changes you make to the code (e.g. describe your OpenMP directives and any code changes you make). You can refer to source files, line numbers, etc. and show code snippets. 2) You should summarize your baseline sequential vs. optimized parallel performance across 1, 2, 4, and 8 threads. You will include your final parallelized code in directory within your hw4.zip submission. This directory should be named amgmk/

Account Usage Guide

1. How to login to the machines

As described earlier, you will be running your programs on a shared memory system. Only SSH access is permitted. If you are using Windows, you will need an SSH client such as PuTTy (free software). If you are using a UNIX/Linux machine, you can login using the command below (depending on which machine you are assigned):

```
ssh -k -X <user_id>@<machine>.duke.edu
```

To copy files from or to your accounts, you can use a secure copy program such as scp.

2. How to obtain the program

You can download the tar file containing the benchmark you are supposed to work on from the class webpage. To un-tar the file, use:

```
tar xzvf <file>.tgz
```

3. How to compile a program

You will use the gcc compiler which provides support for compiling OpenMP programs.

To compile code with OpenMP directives:

```
$ gcc –fopenmp –o < outputfile > <source_file>
```

Where <outputfile> is the output file name (executable) and <sourcefile> is your .c file.
The -fopenmp option informs the compiler that the program is a shared memory program with OpenMP directives.

4. How to run a program

If you want to run an OpenMP program named “test” with 4 threads you can type:

```
$ OMP_NUM_THREADS=4 ./test > result.out
```

5. You need to Submit

You will submit this project as a single zip file named hw4.zip to your Sakai drop box. You are responsible to submit the following contents in the zip file:

1) The report document (report.pdf) that addresses all parts described above.
2) 2 sub-directories (named histo/ and amgmk/) with your source code for the parts described above.