Performance Measurement and Parallel Computation
Project 6 – CS 3339 – Spring 2019

Due Date per TRACS. Friday before 11:55pm, late until Saturday noon -10pts, after that no submissions accepted.

PROBLEM STATEMENT

In this project, you will explore several aspects of performance and parallel execution by running several programs while measuring execution time. Please be aware these programs will necessarily tax the system, in some cases significantly. When running on zeus or eros please be considerate of others and do not execute too many times when the system is in use by multiple users. Also please do not allow batch files to run for more than a few minutes. Any console outputs inside the timed calculation loops will slow the execution significantly and should be avoided except for debugging very small problem sizes. If you are running this on a local machine such as your laptop you may experience malloc failures and should probably save work from other applications first.

ASSIGNMENT SPECIFICS

Submit a brief 2 – 3 page written report in pdf format with a summary of your findings and answers to the questions that follow along with a tar file of your code.

1) Processor/Cache Evaluation (15 points)

A) On the system you will be using for this project build a table of the number of cores, threads per core, L1 instruction cache size, L1 data cache size, L2 cache size, and CPU frequency(s) (e.g. base, turbo, max, etc)

On GNU/Linux systems you can execute the command lscpu or view the proc/cpustats file. On Windows/OSX please research and list the application or method you use.

Download the code MatrixOps.cpp from TRACS.
Read through the code including the timing measurement.

B) Complete the missing code segments to calculate the size of the array and do a column first traversal. Compile with

```
$ g++ -O3 MatrixOps.cpp -o matrixOps -std=c++11
```

Once completed test the timing with various nxn-sized matrices. For n < 500 the timing will likely be very fast, for n > 20000 the memory allocation may fail. For longer running processes you can check your array size calculations using pid or task manager.

C) Record the timings for row first and column first for the smallest meaningful size (that doesn't generate weird times, the largest size that will allocate the array, and at least two in-between points). What difference do you see on row first versus column first? Make a table in your report with the data.

D) Try to capture the effect of a warm vs cold cache, meaning that on the first pass the score is lower than subsequent passes due to the number of cache misses on the first pass through memory. Some possibilities: loop several times in succession with separate timing records, use malloc and free to move array locations, experiment with different size arrays (of course you probably don't want a single pass to bust the cache). Explain what you tried and why plus any results that show the effect (screen shots are great).

2) Pthreads (15 points)

Pthreads (POSIX threads) can be thought of as lightweight processes. They provide a way to execute multiple instruction streams in parallel without the overhead of separate processes. "When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes." From the overview here: https://computing.llnl.gov/tutorials/pthreads/

Download the serial dotproduct code from here:
https://computing.llnl.gov/tutorials/pthreads/samples/dotprod_serial.c

Compile with gcc: `$ gcc dotprod_serial.c -o serial`

Download the pthread version with mutex locks from here:
https://computing.llnl.gov/tutorials/pthreads/samples/dotprod_mutex.c

Compile with gcc and pthread flag: `$ gcc -pthread dotprod_mutex.c -o pthreads`

Read through the code, run both programs several times, and examine the results.

Edit dotprod_mutex.c and comment out the 2 lock and printf lines:

```
//pthread_mutex_lock (&mutexsum);
dotstr.sum += mysum;
//printf("Thread %ld did %d to %d:  mysum=%f global…
//pthread_mutex_unlock (&mutexsum);
pthread_exit((void*) 0);
```

Build a simple shell script to execute and append the sums to a temp file.

```
$ vim loop
```

Create the following script

```
echo "Starting pthread dotproduct without mutex lock" > temp.out
#!/bin/bash
for i in {0..10000}
do
   ./pthreads >> temp.out
done  //note corrected typo in original, should be lower case
```

```
$ chmod +x loop
```


Run the script and look for miscalculations in temp.out. It will take a minute or so to execute.

```
$ ./loop
$ grep -v 'Sum =  400000.000000' temp.out
```

A) Document your results in the report. What is the frequency of error? What is the erroneous value(s)? Why?

Remove the comments added on the lock lines (but not the printf – too much output), recompile, rerun the script file and the grep command. Did the lock actually work?

B) Add timing measurement to the serial and pthreads version of the code. Increase the vector length to 4 million elements for the serial case and 1 million elements for the pthreads version. You do not want to include the setup overhead for either case but you do want the timer to start before the creation of the pthreads and stop after the results have been recombined.

Document your findings in your report. Delete temp.out when you are done.

3) OpenMP (10 points)

Download the *dot_product_openmp.c* source code from
https://people.sc.fsu.edu/~jburkardt/c_src/openmp/openmp.html

Compile and execute the code on zeus or your local machine (for which you have provided the processor info in section 1.)

On zeus the following syntax can be used `$ gcc -fopenmp -lm dot_product.c -o dp`
[Note: -lm is need to properly link the math library, if you get an error retype the command]

A) Submit a screenshot of the output and also compute the speedup and efficiency for each of the four vector sizes. Speedup $S_p = T_1/T_p$ where T_1 is the execution time for a sequential algorithm and T_p is the execution time for a parallel algorithm. Efficiency is the achieved fraction of total potential parallel processing gain.

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

B) Regarding the following two lines of code within the test02 method. Note: this is actually only 2 lines of code the backslash \ is used as a continuation character.

```
# pragma omp parallel \
  shared ( n, x, y ) \
  private ( i )

# pragma omp for reduction ( + : xdoty )
```

Why is the variable i required to be private while n, x, y can be shared? What does the second line do?

SUBMISSION INSTRUCTIONS

Submit your writeup in pdf format along with a tar file containing the updated MatrixOps.cpp file and the timing instrumented version of dotprod_mutex.c to TRACS.

All project files are to be submitted using TRACS. Please follow the submission instructions here:
http://tracsfacts.its.txstate.edu/trainingvideos/submitassignment/submitassignment.htm
Note that files are only submitted if TRACS indicates a successful submission.

You may submit your file(s) as many times as you'd like before the deadline. Only the last submission will be graded. ***TRACS will not allow submission after the deadline***, so I strongly recommend that you don't come down to the final seconds of the assignment window. Late assignments will not be accepted.

ACADEMIC HONESTY (excerpt from course syllabus)

You are expected to adhere to the Texas State University Honor Code which is described here
http://www.txstate.edu/honorcodecouncil/Academic-Integrity.html

All assignments and exams must be done individually, unless otherwise noted in the assignment handout. Turning in an exam or assignment that is not entirely your own work is cheating and will not be tolerated.

Group discussion about course content is NOT cheating and is in fact strongly encouraged! You are welcome to study together and to discuss information and concepts covered in class, as well as to offer (or receive) help with debugging or understanding course concepts to (or from) other students. However, this cooperation should never involve students possessing a copy of work done by another student, including solutions from previous semesters, other course sections, the Internet, or any other sources.

Turning in an assignment any part of which is copied from the Internet, another student's work, or any other non-approved source will result in a 0 grade on the assignment and will be reported to the Texas State Honor Code Council. Should one student copy from another, both the student who copied work and the student who gave material to be copied will receive 0s and be reported to the Honor Code Council. You should never grant anyone access to your files or email your programs to anyone (other than the instructor)!

