CSCI 551 – Numerical and Parallel Programming: Exercise #2

Grid and Matrix Processing with Shared Memory Threading

DUE: As indicated on Canvas

Please thoroughly read Week-3 and Week-4 notes and bring your questions to discussion. You should expect to spend about 1 hour per 10 points, and complete spending 5 hours per week. This leaves you 4 hours per week for reading, coming to office hours, and taking quiz #1.

You will need an ECC system (***ecc-linux.csuchico.edu***) account or access to a native Linux system that is multi-core to work on this assignment. Most students can just SSH to the ECC system using MobaXterm, Cyberduck, or Putty or whatever your favorite SSH tool is, but please obtain an account one if you don't already have (https://www.csuchico.edu/itss/index.shtml). Note that you can download the starter code from the class website as a gzip at http://www.ecst.csuchico.edu/~sbsiewert/csci551/code/, which is the last distribution. For analysis and design work, you will find the LLNL POSIX threads tutorials useful as well as use of "man -k pthread" on your Linux system.

All code submissions must be accompanied by a Makefile or similar build automation with documentation on how to build with a "single command" from the command line on either ECC server (ecc-linux.csuchico.edu) or a Raspberry Pi or Jetson Linux system. Report your results, clearly answering each of the questions below with analysis, example output, code snippets, and/or mathematical analysis and solutions (answers clearly indicated by boxing them).

ECC Cluster node use policies – Based on your birthday and year, if your year of birth is even, use "o244" nodes and if you birth year is odd, use "o251" nodes. For POSIX shared memory threading (single node use), login to the node # that is the same as your birthday – e.g. for me, odd year, 14th day of month, so I would use "ssh o251-14". This should help distribute the load as we get into problems that are more CPU, I/O, and memory intensive.

Exercise #2 Objectives:

- Learn about basic Pthread programming by example – thread creation, thread join, and an introduction to concepts of thread safe functions.
- Learn about programs that are harder to make parallel and how to time parallel and sequential sections of code.
- Learn basic vector and matrix operations and how to code them, but also how to check your work using math solver tools (e.g., https://matrix.reshish.com/ and MATLAB). ***Note that you must do work requested as original work, but you are encouraged to use tools to check and verify your work.***

1) [30 points] Download the basic Pthread example C code from our class website (csci551/code/simplethread.zip, csci551/code/sumnums.zip, csci551/code/incdecthread.zip), and make sure it builds and runs for you, observe what it does, and then complete the following. Note that the sum of a sequence of numbers (0, 1, 2, 3, 4, …, n) counting by 1, is known to be n(n+1)/2, so you can check your work based on properties of arithmetic progression.

   a) [10 pts] For simplethread, where ***n=number range to sum up***, ***m=number of threads (workers)***, modify the pthread.c code and comment it well and analyze output and results carefully:
      1. Run your program with 8 threads where all threads sum 1…n, where "n" is the (thread # (0…7) +1) x 100, so that you sum to 100, 200, 300, …, 800 for each thread accordingly. Check with n(n+1)/2 formula to make sure each is correct (5050, 20100, 45150, …, 320400).
      2. Document and explain any difference you see in output over multiple runs (at least 3).

   b) [10 pts] Modify sumnums **Pthread** code so that it uses 10 threads to sum the numbers between 1…1,000,000 in 10 sub-ranges of 100,000 and sum the sums at the end such that it computes the expected value of n(n+1)/2, which is 500000500000. and comment your code well. Note whether the threads always complete in the same order, and if not, explain why, and whether this is necessary for correctness.

   c) [10 pts] Create an **OpenMP** version of sumnums Pthread example code and compare run time and order of summing for 10 threads to sum the numbers between 1…1,000,000 in 10 sub-ranges of 100,000 and sum the sums at the end such that it computes the expected value of n(n+1)/2, which is 500000500000. Can you verify that the subranges are correct with a formula?

2) [30 points] **Finding Number of Primes on Interval and Prime Factors for a Large Semi-Prime**: For this exercise, you may refer to the following **Pthread** example code (csci551/code/Eratosthenes-Simple/) and ***use it if you wish***, but you must modify it to meet the exercise objectives. You may design and implement code any way you wish (***using OpenMP for speed-up***), to ***find the largest prime numbers you can between 0… 1 billion and the total number of primes on this interval (keeping a list) as a step toward large semi-prime factoring***. Please confirm your run time and exit based on POSIX clock_gettime using the MONOTONIC RAW clock and print out start and end time in floating point seconds. It is not possible to use a simple square root approximation since you must count the number of primes (and keep a list) as well as find the largest on the interval (e.g., this code is not sufficient - https://www.geeksforgeeks.org/nearest-prime-less-given-number-n/). Compute the semi-prime number (64 bit unsigned) that is the product of the two largest primes

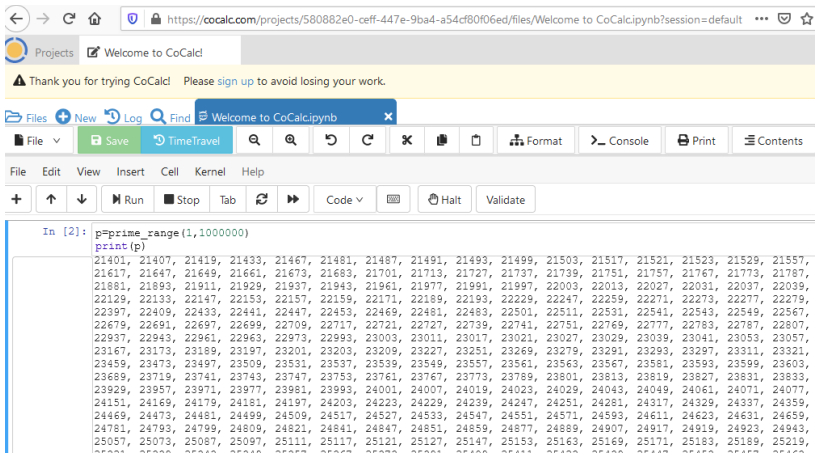
between 0…1 billion. According to this list of primes, here is an ***example*** using 2 of the larger prime numbers less than 1 billion (32-bit) to compute a large semi-prime (64-bit) by multiplying these 2 large primes together: **999999491 X 999999733 = 999,999,224,000,135,903.**

Your code should be able to find any randomly selected set of primes P_1 and P_2 that compose SP with the equation $SP=P_1 \times P_2$ for any semi-prime < $(1,000,000,000)^2$. E.g., if the primes were each randomly selected from the list and used to compute any possible SP. Here's a list of some semi-primes and known prime factors to test your program:

1. 35: factored into 7, 5
2. 376,223: factored into 439 & 857
3. 4,006,336,753: factored into 46,411 & 86,323
4. 406,615,978,649: factored into 470,303 & 864,583
5. 4,154,092,115,820,191: factored into 47,868,193 & 86,781,887
6. 418,155,269,059,864,129: factored into 481,346,903 & 868,719,143

Verify that you can find these 2 large primes < 1 billion and compute the expected semi-prime. The strength of encryption comes from the trap-door mathematical characteristic based on how hard it can be to find the two primes that are factors of a large semi-prime (e.g., a 128-bit semi-prime composed of any two 64-bit range primes).

The sieve of Eratosthenes and sieve functions in general are the only fully reliable way to compute the number of primes (keeping a list without generating false positives) up to the largest on an interval, and ***requires lots of memory for tracking***, even for a bitmap version, so you may research and use an improved algorithm, or just work on making the common example run faster or both. Comment your code well and report on your methods to speed up the code and verify that your largest prime is correct. ***Be careful with array sizes and check return codes if you use malloc*** to make sure memory allocation does not have errors. Note that ***cluster nodes usually have more free memory than ECC*** (e.g., check with "free -h"). You should verify your results with a second source (e.g., https://www.sagemath.org/, https://cocalc.com and https://t5k.org/howmany.html) or similar tool or authoritative list.



MATLAB verification (free download for CSU – see notes):

```
>> p=primes(1000)

p =

  Columns 1 through 19

     2     3     5     7    11    13    17    19    23    29    31    37    41    43    47    53    59    61    67

  Columns 20 through 38

    71    73    79    83    89    97   101   103   107   109   113   127   131   137   139   149   151   157   163
```

Other prime number list generation starter code that may be helpful include: csci551/code/erast_compare.zip, csci551/code/Eratosthenes-sieve-primes.zip, and https://github.com/kimwalisch/primesieve/wiki/Segmented-sieve-of-Eratosthenes.

3) [40 points] For this exercise, you must write C or C++ code that will rotate every image 90 degrees right or left for each playing card in a standard deck of cards (cards_3x4_ppm.zip for color, and cards_3x4_pgm.zip for grayscale) or halt after running for 3 seconds (please confirm your run time with POSIX clock_gettime) using a matrix rotation of your own implementation for the 3:4 aspect ratio cards provided. Spades and clubs should be rotated right and hearts and diamonds rotated left. You can put the deck in /tmp and write it back to /tmp/cards_rotated and then zip up your results along with description of how far you get – if you complete the whole deck in less than 3 seconds, provide the time it took to process the whole deck. Time must include time to load and store the images. Your code should work for any rotation of 90 degrees, clockwise, counterclockwise (90, 180, 270, 360), but you only need to time the rotate right/left case for the 4 suits. You can use a PGM (grayscale) set of cards to simplify the problem or PPM (color). ***Use OpenMP to speed-up your code to see if the sped-up version can complete in less than 3 seconds. Provide a comparison of your OpenMP threaded to single thread version in terms of total execution time required.***

Recall that in Exercise #1 the PSF convolution code has an example of how to open and read a PPM (header and RGB binary data) as well as how to write one out. A PGM is similar, but has only grayscale binary data, and is therefore simpler. Documentation on PPM and PGM

can be found here - https://en.wikipedia.org/wiki/Netpbm You are welcome to use any code you wish to read in the playing cards as PGM or PPM, but this simply requires reading the header and then reading the data based on the header resolution (in this case, all playing cards have the same resolution).

Overall, provide a well-documented professional report of your findings, output, and tests so that it is easy for a colleague (or instructor) to understand what you have done, what worked, what did not and why (even if you cannot complete to your satisfaction). Provide clear instructions on how to run your programs, including command line arguments required and screenshots demonstrating use and test cases you used to verify your parallel and sequential programs. For all parallel programs please show you have completed the triple challenge (still works, has speed-up, can scale-up and if applicable, can scale-out).

Include any design files or log files, C/C++ source code you write (or modify) and Makefiles needed to build your code. I will look at your report first, so it must be well written and clearly address each problem providing clear and concise responses and example results (e.g. summary analysis and clearly boxed mathematical answers) to receive credit, but I will look at your log files, code and test results as well if I have questions.

Report file MUST be separate from the ZIP file with code and other supporting materials.

Rubric for Scoring for scale 0…10

Score	Description of reporting and code quality
0	No answer, no work done
1	Attempted and some work provided, incomplete, does not build, no Makefile
2	Attempted and partial work provided, but unclear, Makefile, but builds and runs with errors
3	Attempted and some work provided, but unclear, build warnings, runs with no apparent error, but not correct or does not terminate
4	Attempted and more work provided, but unclear, build warnings, runs with no apparent error, but not correct or does not terminate
5	Attempted and most work provided, but unclear, build warnings, runs with no apparent error, but not correct or does not terminate
6	Complete answer, but does not answer question well and code build and run has warnings and does not provide expected results
7	Complete, mostly correct, average answer to questions, with code that builds and runs with average code quality and overall answer clarity
8	Good, easy to understand and clear answer to questions, with easy to read code that builds and runs with no warnings (or errors), completes without error and provides a credible result
9	Great, easy to understand and insightful answer to questions, with easy to read code that builds and runs cleanly, completes without error, and provides an excellent result
10	Most complete and correct - best answer and code given in the current class

Grading Checklist for Rubric

[30 points] Simple POSIX threading code:

Problem	Score 0…10	Comments
A		
B		
C		
TOTAL		

[30 points] Finding the largest prime and number of primes on a large interval as well as expected semi-prime product of 2 largest primes < 1 billion:

Problem	Score 0…10	Comments
Speed-up		
Timing		
Verification		
TOTAL		

[40 points] 90 degree matrix rotation:

Problem	Score 0…10	Comments
Algorithm		
Code quality		
Speed-up		
Timing & Verification		
TOTAL		