ECE 451-566 Parallel and Distributed Computing, Rutgers University, Fall 2020.

Instructor: Maria Striki

Total Points: **50 + 50 = 100**

Issue Date: 11-11-2020 (Wednesday) Due Date: 11-24-2020 (Tue) at 22.00

Programming Project 2:

When you run your programs, you may compare them against other nodes in your group, if the specifications of other machines (nodes) are different.

Problem 1 (50 points):

Part 1 (6 pts): Write a sequential version of **Quicksort OR Bitonic Sort.** Construct your own random input of serial array. You must experiment with array sizes starting from O(1000), O(1,000,000), up to 50,000,000 elements (as long as your machine does not crush!). Measure (time) the performance of your implementation across all array sizes and report your observations. If you have time you may work on both sorting algorithms for extra credit.

Part 2 (18 pts) : Then, write parallel versions of **Quicksort OR Bitonic Sort** using Posix Pthreads. Investigate if you need to apply any of the first and simple synchronization mechanisms discussed in class. If so, for which sorting algorithm, and for which steps?

Part 3 (12 pts): Research and find which of the sequential version sorting algorithms lend themselves easily to parallel execution. If there are some that do not, explain why and check if they can be "tweaked" so that they produce related versions that are easy to parallelize. If so, please pick one such version, describe it, and implement it using Pthreads. Generate and use the same input arrays for the same sizes as before (using the same random generator every time).

NOTE: Vary the number of threads used (from 2 to max number before your system crashes (whatever this is), measure (time) the performance of your algorithm and report the results.

Part 4: Experimental Analysis (8 pts):

If you implemented both: Which method performs best when implemented in parallel? Which method performs best sequentially?

If you implemented both or one: How does the parallel performance of your method compares with the sequential? Do your findings agree with what you would expect from analyzing the behavior of your program?

If you implemented both or one: How does the performance of your parallel implementations change as you increase the number of threads? Why?

Part 5: Theoretical Complexity Analysis (6 pts): Conduct your own analysis or do some research and report:

a) Which of the two algorithms when run sequentially has the best performance in order of complexity?

b) Which of the two algorithms when run in parallel has the best performance in order of complexity? Why, how?

c) Do you results agree with the experimental results and analysis conducted in **Part1-Part4**? If not, or if not entirely, could you justify your answer with a few distinct arguments?

Solution:

Problem 2 (50 points): Implement a simple **password cracker** using either ISPC or Pthreads.

Implement a password which is between 1-8 ascii characters (at least try to do so (depending on your hardware resource capabilities) by gradually increasing the character length which can be only alphanumerical at the beginning). You will then use one of the **popular hash functions** to convert your password to a hash. You are expected to do adequate research on this topic and list your results in a detailed report. We assume that the attacker (you-the group-the programmer) captures only the hash of your password and no other information.

Before you start coding and tackling the actual problem, please attempt to understand it by conducting some research in order to answer the following questions:

Part 1: Theoretical Preparation and Investigation (10 pts)

1) What is a hash function? What are its properties? Name and describe the most popular hash functions.
2) What is a rainbow table (w.r.t. hashing)?
3) Whenever the attacker gets a hashed password, can he directly know what the hash function is? Or can he somehow retrieve or investigate the hash and obtain more information?
4) How can the attacker eventually obtain the hash function so that he can brute-force or use a rainbow table to try any combination of characters through the hash in order to obtain the same hash value?
5) When to use a Dictionary attack vs. a Rainbow Table attack? What are the resource requirements for each type of attack? Which uses more storage? Which requires more pre-computation? Which requires more analysis time? (per-hash vs. batch cracking)

Part 2: Coding and Experimentation (40 points): You are to implement your own password cracker based on the concepts investigated above. You are provided the range of the password length, which is to be between 1-8 characters. Your group should brute force the password, i.e., take every possible combination of words within the mentioned length, convert it into a hash and compare it with the provided hash. For ease of the implementation here, no matter what your findings were in Part 1, you will assume that both the victim and the attacker are aware and use the **same hash function** (you choose which one). All groups in addition to your own imaginary passwords, **also use the following** to compare against all groups and all varying implementations: **bv37qi#f**.

This task can be easily parallelized. You first generate the hashes of various works, then attempt to match them with the obtained hash from the victim, via your dedicated hardware (what is the ideal HW to be used here?).

You may **explore various and diverse strategies for your problem decomposition and assignment** with the end goal to achieve **optimal workload utilization** which – under circumstances – leads to fastest times of password cracking. **A major portion of your grade will be slated for efficient decomposition and assignment strategy.**

Deliverables and write-up:

1) Code for sequential implementation **(8 pts)**
2) Code for parallelized implementation: extend your code to utilize **any number of Pthreads or ISPC tasks**, partitioning the computation accordingly (increase Pthreads/ISPC tasks from 2, 3, 4, 5, …, 16, … , 32, …., 128, …. 512, … whatever your system can sustain). **(18 pts)**
3) Use your code to run experiments: Produce graphs of speedup comparing parallel to sequential implementation, and also speed up of distinct parallel implementations alone, as a function of the number of cores and threads/tasks used. **(7 pts)**
4) For the distinct parallel implementations, how does speedup increase or decrease w.r.t. the number of threads/tasks (linearly, as a convex, abrupt/steep curves, …)? What do you observe? Interpret your results and provide adequate arguments. **Rational interpretation of this part will gain your group a lot of points (out of the 40)**, so please analyze and think about your results carefully. **(7 pts).**

IMPORTANT NOTE 1: Also experiment with a number of distinct password characters and report the difference in the timing results.

IMPORTANT NOTE 2: All groups in addition to your own imaginary passwords, **also use the following** to compare against all groups and all varying implementations: **bv37qi#f**. Apply a number of popular hashes and obtain a number of versions of the password. And you take it from there…

ADDITIONAL/EXTRA: For the curious: you may also conduct some search on the web, find and utilize an existing software tool for password cracking. Does your tool lend itself easily to parallel execution? You are expected to do some research and find one that does. Use it to conduct the same experiment. Compare results and discuss them in your report.

Solution:

Output and Grading:

You are encouraged and welcome to run your code on the orbit testbed in WINLAB. Use your ORBIT account per group. Time slots are reserved everyday normally between 20.00-12.00am but also randomly earlier in the day. Pthread compiler and libraries come with the gcc compiler which is by default included in "mariasfirstimage". Also, you have already used ISPC compiler using the same image, so I hope by now everyone is familiarized with using ISPC through ORBIT. Of course, you may use your own machines or even labs with powerful cores if you have access to.

What to Submit: Please submit through sakai a folder including: 1) all your code, 2) a readme file on how exactly you run your code, 3) all your results, 4) a detailed report that addresses all questions regarding the two problems and gives a sufficient description of your findings.

Due Date: Please submit everything via sakai by Tuesday**, Nov 24th at 10.00 pm**.

Late submissions policy: you may submit late, past the due date, but there will be considerable penalty for every day you delay your submission.

Grading: You get full grade if you adequately address and justify every question, and of course if you provide correct code that is running and corresponds to the questions of the problems.

Groups working solo: You are asked to work only on one of the two problems, but must use Pthreads.

Maria Striki