ECE 451-566 Parallel and Distributed Computing, Rutgers University, Fall 2018.

Instructor: Maria Striki TA: George Chantzialexiou

Total Points: 60 + 40 + 25 = 125

Issue Date: 10-27-2018 (Saturday) Due Date: 11-13-2018 (Tuesday) 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.

Part 1 (35 points): Write a sequential version of **BubbleSort, MergeSort, and QuickSort**. Construct your own random input of serial array. You may experiment with array sizes starting from O(1000) to O(1,000,000), perhaps even up to 30,000,000 elements. Measure (time) the performance of your implementation across all array sizes and report your observation. Then, write parallel versions of Bubble Sort, Merge Sort, and Bucket Sort using Pthreads. Do your own 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. If those that are difficult to parallelize can be "tweaked" so that they produce related versions that are easy to parallelize, please pick one such version, describe it, and implement it in a parallel manner using Pthreads.

Use the same input arrays. Vary your number of threads for parallelization, measure (time) the performance of your algorithm and report the results. Describe and justify the algorithm for parallelization you will be using.

Which method performs best when implemented in parallel? Do your findings agree with what you would expect from analyzing the behavior of your program? Which method performs best sequentially? Justify your answers. Also, how does the performance of your parallel implementations change as you increase the number of threads? Why?

Theoretical Complexity Analysis: Conduct your own analysis or do some research and report: i) a) Which of the three algorithms when run sequentially has the best performance in order of complexity?

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

Do you results agree with the analysis above? Why yes, why not?

Optional Step – **Extra Credit** (for those that are familiar with these systems): You may measure the speedup of your parallel program on an increasing number of CPUs: 1, 4, 8, 12, 16 if you utilize 2, 3, or all 4 nodes of the grid you have at your disposal. You may do so by

communicating portions of your code to other machines (using sockets or other communication primitives you have a good command of, e.g., MPI).

Part 2 (35 points): Implement a parallel version of the Minimum Spanning Tree: Prim's Algorithm by using Pthreads. A sequential version of the algorithm is illustrated.

A minimum spanning tree (MST) for a weighted undirected graph is a spanning tree with minimum weight. As an example, the minimum length of cable necessary to connect a set of computers in a network can be determined by finding the MST of the undirected graph containing all the possible connections. If G is not connected it cannot have a spanning tree, but it probably has a spanning forest.

Prim's algorithm for finding an MST is a greedy algorithm. It begins by selecting an arbitrary starting vertex. It then grows the minimum spanning tree by choosing a new vertex and edge that are guaranteed to be in a spanning tree of minimum cost. The algorithm continues until all the vertices have been selected.

Let G = (V, E, w) be the weighted undirected graph for which the minimum spanning tree should be found, and let matrix: *A* = a[i, j] be its weighted adjacency matrix.

Initialize matrix *A* with "reasonable random" variables so that you ensure that your graph of N vertices remains connected. How will you ensure that? Think about it before you start. You should work with a very large matrix A of the same dimensions as those you used for Part 1.

```
1.    procedure PRIM_MST(V, E, w, r)
2.    begin
3.        V_T := {r};
4.        d[r] := 0;
5.        for all v ∈ (V − V_T) do
6.            if edge (r, v) exists set d[v] := w(r, v);
7.            else set d[v] := ∞;
8.        while V_T ≠ V do
9.        begin
10.           find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
11.           V_T := V_T ∪ {u};
12.           for all v ∈ (V − V_T) do
13.               d[v] := min{d[v], w(u, v)};
14.       endwhile
15.   end PRIM_MST
```

Prim's sequential minimum spanning tree algorithm.

The algorithm uses set V_T to hold the vertices of the MST during its construction. It also uses an array d[1..n] in which, for each vertex *u* in (V- V_T), d[*u*] holds the weight of the edge with the least weight from any vertex in V_T to vertex *u*. initially, V_T contains an arbitrary vertex *r* that becomes the root of the MST. Furthermore, d[*r*] = 0, and for all *u* such that *u* in (V- V_T), d[*u*] = w(r, u) if such an edge exists, otherwise d[u] = Inf. During each iteration, a new vertex *u* is added to V_T such that d[*v*] = min {d[*u*]|*u* in (V- V_T)}. After this vertex is added, all values of d[*u*] such that *u* in (V- V_T) are updated because there may now be an edge with a smaller weight between vertex *u* and the newly added vertex *v*. The algorithm terminates when V = V_T.

Implement this algorithm first sequentially, compute its complexity both analytically (there are many documents on sequential Prim that derive complexity in O(N)) and experimentally, using your C based execution.

Then, search for the optimal in your opinion way to parallelize this program. A word of caution: this algorithm is not easy to parallelize. Justify why. Try to do your own research to find an acceptable algorithm for parallelization or come up with your own. Beware of what synchronization primitives you select to use. Justify which and why (if any). Measure the speedup of your parallel program on 4 CPUs. You may experiment by modifying the number of threads to be used at certain steps of implementation and again report your results. Compute once again the theoretical complexity of this parallelized version in O(N) and compare it with the execution time of your own program.

Optional Step – **Extra Credit** (for those that are familiar with these systems): You may measure the speedup of your parallel program on an increasing number of CPUs: 1, 4, 8, 12, 16 if you utilize 2, 3, or all 4 nodes of the grid you have at your disposal. You may do so by communicating portions of your code to other machines (using sockets or other communication primitives you have a good command of, e.g., MPI).

Part 3 (30 points): Implement a simple **password cracker** using Pthreads.

Below I am quoting the same guidelines I have posted for ispc. This is the same question, but this time you are going to use Pthreads instead of ispc and compare your password cracker performance to this of Project 1.

You are going to make a password which is between 1-8 ascii characters (at least try to do so 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. So, let's assume that the attacker (you-the group-the programmer) is exposed only to the hash of

your password and to no other information. Another question to answer: what is a hash function? What are the properties of a hash function? When the attacker gets a hashed password can he directly know what the hash function is? Or can he retrieve or investigate the hash? How does the attacker eventually obtain the hash so that the attacker can brute-force or even use a rainbow table to try any combination of characters through the hash in order to obtain the same hash value? So, you are asked to do exactly the same. You are to implement your own password cracker based on the rules above. You are provided the range of the password length, which is to be between 1-8 characters. Provided this information, your group should brute force the password, i.e., take every possible combination of words between the mentioned length, convert it into a hash and compare it with the provided hash. Yet a couple of extra questions to answer: i) When to use a Dictionary attack vs. a Rainbow Table attack? Ii) 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)

This task can be easily parallelized. You are expected to use and exploit exactly the concepts worked upon in Project 1 Part 1 and Part 2 in the best possible way to provide the fastest times for your password cracker. And you may select various and diverse ways of your problem decomposition and assignment with the end goal to achieve optimal workload utilization which may of course – under circumstances – which do you believe? – lead also to fastest times of password cracking. To facilitate your understanding on the concept of assignment, let's consider a scenario for which thread 1 checks only combinations of length 1, thread 2 does that for 2 letter words, thread 3 for 3 letter words etc. So, you first generate the hashes and then matching them in bulk via your dedicated hardware (what type of HW in particular should you be using here?).

Extend your code to utilize any number of threads, partitioning the computation accordingly. In your write-up, produce a graph of speedup compared to a sequential implementation (which you need to implement as well) as a function of the number of cores and threads used. Is speedup linear in the number of cores used? In the number of threads used? Please justify why yes or why not.

You may also conduct some search on the web and utilize in addition 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. Conduct a similar experiment as the one with your own password cracker. Compare the results and discuss them in your report.

Note: Also experiment with a number of different password characters and report the difference in the timing results.

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…

You are encouraged 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 incuded in "mariasfirstimage".

Output and Grading:

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.

Deadline: Please submit everything via sakai by **Tuesday Nov 13, 10.00 pm**.

Not Allowed: Any sort of plagiarism, either among colleagues or from web resources. There is a great benefit to work within your own group and do as much as you can. This is the only way to learn and get something useful out of this class!

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.

GOOD LUCK

Maria Striki