CMSC 180 Laboratory Research Problem 2

Jarem Thimoty M. Arias — CMSC 180 T-5L

I. Introduction

This Research Problem serves as an introduction to Parallel Programming through the use of the same base algorithm that was encountered in the previous Research Problem which was the Pearson Correlation Coefficient Algorithm. The PThreads library in C was used to implement the multi threading for both columnar and row submatrices.

March 3, 2024

A. Research Question 1

What do you think is the complexity of solving the Pearson Correlation Coefficient of the columns in an n x n square matrix X and an n x 1 vector y when using n concurrent processors? The obvious processor assignment is one column of M and the vector y for each processor.

Since there are n concurrent processors, that each will compute for a single column in the n x n matrix, it will most likely be of a complexity of O(n) since each column will finish in a linear fashion at roughly the same time.

B. Research Question 2

What do you think is the complexity of solving the Pearson Correlation Coefficient of the columns in an n x n square matrix X and an n x 1 vector y when using n/2 concurrent processors (what is the obvious processor assignment here)? What about with n/4 concurrent processors (i.e., processor assignment)? What about with n/8 concurrent processors? What about with n/m concurrent processors, where n is greater than m? Is the processor assignment still obvious at n/m concurrent processors?

Since complexity does not care about attached coefficients such as 2 or 1/2, the complexity remains at O(n). However, there will be a noticeable decrease in speed as the number of processors get lower (This is in a theoretical sense). In the case of n/m processors where n is greater than m, the assignment of processors might be a bit less clear as the tasks may not be evenly distributed, some threads may take up more tasks than others to compensate.

II. Objectives

1) Identification of parts of an algorithm that can be performed concurrently
2) Distribution of data amongst processes/threads
3) Assignment of each piece into different processes that will run in parallel
4) Managing data sharing between processes
5) Synchronization of of processor execution to guarantee consistency

III. Methodology

The code from the previous Exercise was reused and added on to be able to add the Functionalities required for both Research Activity 1 and Research Activity 2.

The PThreads library from C was used to be able to transform the program into a threaded version. Instead of removing the serial functionality, it was kept and the input was modified to both choose the number of threads and the type of algorithm (Serial or Parallel) that was to be used.

The Columnar division of the submatrices of the multi-threaded program (Research Activity 1) was easy to code, all that was needed was to turn a few structures public so that all the threads could access the data needed to compute the entire 2d matrix. Since there were no overlapping data that the threads needed to access, there were no need for locks on the data. The division of the data was done through giving each thread a corresponding structure that contained the ranges of data they were responsible for.

For the row version of the submatrices, it was a bit more complex, but it will be explained during the Results chapter.

The device that ran the results for this paper had been changed from the previous Research Problem and as such the results needed from the previous Research Problem was ran again on the machine with the following specs:

1) CPU: AMD Ryzen 5 4500U
2) Ram: 16GB Generic RAM running @2666 mhz

IV. Results and Discussion

n	t	Run 1	Run 2	Run 3	Average
25000	Serial	20.3896	20.75016	20.71051	20.61675667
25000	1	20.40524	20.42882	20.42679	20.42028333
25000	2	10.0351	10.06189	10.04772	10.04823667
25000	4	8.81355	8.80635	8.92724	8.849046667
25000	8	9.48265	9.52775	9.49049	9.500296667
25000	16	9.53948	9.74418	9.68244	9.655366667
25000	32	9.51933	9.45914	9.50961	9.496026667
25000	64	9.7252	9.56718	9.57223	9.621536667

TABLE I
Graph of runtime (Columnar Multithreading)

As can be seen in the table, the serial program is similar in speed to t=1, then t=2 cuts the runtime in half, and the fastest run belongs to t=4. The following t then plateaus at around 9 seconds. This may be due to the number of threads saturating the amount of cores that the device running the program and the extra threads become a hindrance instead of helping the runtime past t=4.

A. Research Question 3

Why do you think that t = 1 will be a little bit higher than the average that was obtained in LRP01?

I had to do another run of my previous Laboratory 1 Exercise as I had made changes to the timing system that may have an effect on its runtime and I also was not using the same machine that I used for the results in the first paper.

To answer the question, unfortunately my run did not show a difference between the serial program and t=1. But t=1 may be higher due to the added processes needed to initiate threads.

B. Research Activity 2

With lab02, repeat the activities in LRP01 for n = 30,000 and n = 40,000. Do you think you can now achieve n = 50,000 and even n = 100,000? Try it to see if you can. If you were able to do so, why do you think you can now do it? If not yet, why do you think you still can not?

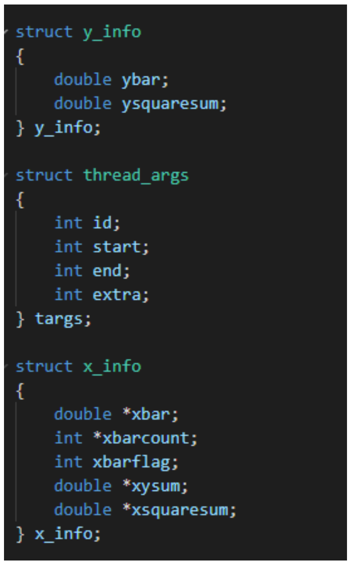
At the moment, I am still currently unable to run past n=45000 as the device I am using does not have enough memory to run it past that amount of n. If further changes to the program are made, mainly to the space taken up by the n x n matrix, it is possible that I may be able to run it in the future.

C. Research Question 4

In step (4) in number 1 above, explain what will happen if we divide X into n x n/t but n/t x n instead? How are we going to do it so that the same answer can be arrived at?

If we were to divide the submatrices by row instead of by column, there would be a lot of changes needed so that the same answer could be arrived at.

The biggest change is the additional data structures that would be needed to accommodate the fact that multiple threads will be computing for the same value in a column.



```
struct y_info
{
    double ybar;
    double ysquaresum;
} y_info;

struct thread_args
{
    int id;
    int start;
    int end;
    int extra;
} targs;

struct x_info
{
    double *xbar;
    int *xbarcount;
    int xbarflag;
    double *xysum;
    double *xsquaresum;
} x_info;
```

Fig. 1. Additional structures for row multithreading

The values regarding the y array are computed before the threads are initialized so that it can be used by the threads immediately. However the problem lies with the values regarding the x per column. Since each thread does not have full access to the entire column, the final Pearson Correlation Coefficient cannot be immediately solved by each thread, instead all threads must pass through a specific column before the computation can continue.

This is further complicated by the fact that the threads have to access the same element in the x_info structure. An array of mutex locks corresponding to a column was used so that the threads can prevent race conditions and data corruption. Each thread also has to wait for the main thread to compute for the final values of x_info before it can proceed to computing the other needed variables of the computation in my implementation of the program.

All these implementations above to make the by row multi-threading work however causes the program to slow down by a significant amount due to the waiting caused by the mutex lock and waiting of both the main thread and the computing threads.

n	t	Run 1	Run 2	Run 3	Average
25000	1	28.02866	27.97929	27.97352	27.99382333
25000	2	26.58652	20.86399	15.27446	20.90832333
25000	4	12.32912	14.59419	11.85961	12.92764
25000	8	7.76933	7.76796	7.90713	7.814806667
25000	16	11.79895	11.58976	12.19432	11.86101
25000	32	15.56794	14.84974	13.89462	14.77076667
25000	64	13.25279	16.88026	16.43599	15.52301333

TABLE II
GRAPH OF RUNTIME (ROW MULTITHREADING)

The results of the row multithreading are somewhat similar to the columnar multithreading, however it can be seen that it is slower and has much more variance when it comes to the run time due to several factors. The slowest run is that of t=1, it then speeds up until t=8, which probably saturated the number of cores in the device I was using to run it. It then slowed down until it reached a plateau of around 15 seconds. The

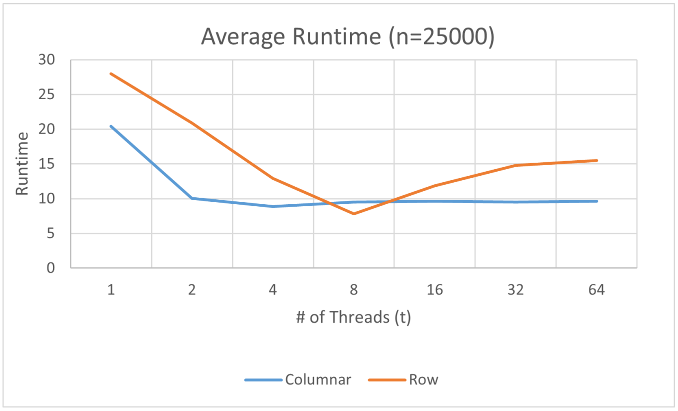


Fig. 2. graph of runtime results n=25000

Looking at the graph, it can be seen that the average runtime decreases as t increases up to a certain point for the columnar approach. n=t is most likely not possible to reach due to memory constraints and the fact that it may be detrimental to add more threads since it will add more processes and the amount of speed it provides is negligible or even would make it worse. The Row multithreading follows a similar curve, albeit slower overall but it has the fastest peak of around 7 seconds at t=8.

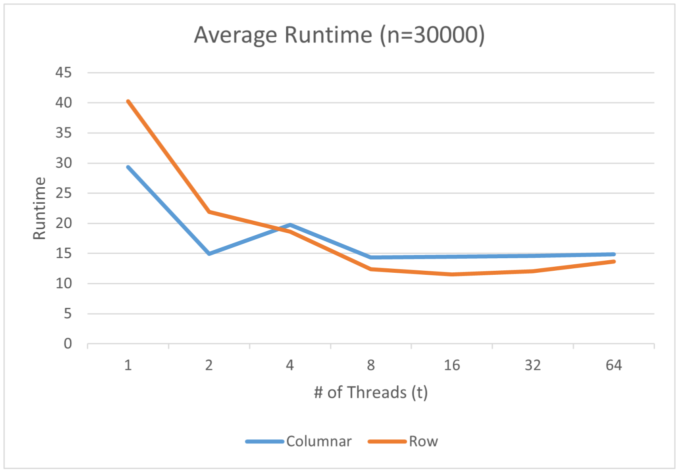


Fig. 3. graph of runtime results n=30000

As expected, the overall average runtime of the code when n=30000 is higher compared to the previous run of n=25000. In these results, the row multithreading is slower at the start compared to the columnar run, however the row multithreading overtakes the columnar one in speed starting at t=4 albeit not by a significant amount. The general form of the graph also follows the previous graph, the runtime decreasing up to a certain t when the program saturates the resources of the device running it.

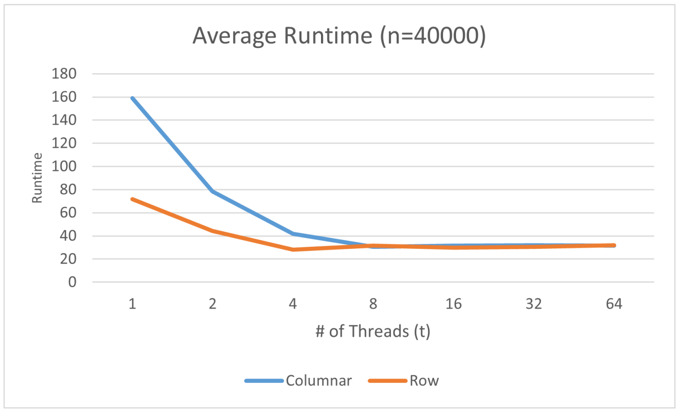


Fig. 4. graph of runtime results n=40000

At n=40000, the graph follows a similar form albeit with some major differences. Unlike the previous graphs, the row multithreading is significantly faster at the beginning compared to the columnar version. An explanation for this phenomena may be the possibility that the way the row multithreading is coded may be taking advantage of the row major preference of C. However, as t increases, the speeds of the two methods are the same.