Runtime-efficient Threaded Pearson Correlation Coefficient

Elvin Marc M. Bautista

ABSTRACT

This paper presents the execution time of a C programming language function that computes the Pearson Correlation Coefficient of a Matrix and a Vector. The entire program utilizes multithreading to perform concurrent computation for the submatrices of the matrix. I examined the algorithm's time complexity and explored how the program would behave when the matrix is divided into t submatrices, considering both row-based and column-based partitioning strategies.

Keywords

Pearson Correlation Coefficient, Matrix, Vectors, C Programming Language, Time Complexity, Threading, Threaded Programs

1. INTRODUCTION

In the realm of computing, a thread represents a singular sequential flow of execution encapsulated within a process. Thread is sometimes referred to as lightweight process due to its similarities to an actual process[1]. Since threads generally perform faster than a process, utilization of these is an effective way to improve the performance of an algorithm or a program. A program's runtime to compute for a value that involves a large amount of data can be improved by making it into a threaded program, where each thread will compute a part instead of making the single process compute. A threaded program utilizes multiple threads that can perform concurrent operations, these threads can access the same data so data integrity is not an issue when utilizing multiple threads to make an algorithm or a program perform faster. The Pearson Correlation Coefficient is used to measure the relationship between two variables. This is often denoted as r, the formula for calculating the Pearson Correlation Coefficient is given by

$$r = \frac{n\sum(xy) - (\sum x)(\sum y)}{\sqrt{(n\sum x^2 - (\sum x)^2)(n\sum y^2 - (\sum y)^2)}}$$

where r represents the Pearson Correlation Coefficient, x and y are data points and, n is the number of data points. The algorithm will compute the Pearson Correlation Coefficient of two given vectors. To improve the performance of computing the Pearson Correlation Coefficient between a matrix and a vector, utilization of multiple threads that will compute the coefficient for every submatrices of the matrix can be implemented. The vectors x will come from the columns of the given submatrix. Finally, after all the threads finish its computation, the result of each thread will be stored in a vector. This division of matrix will make the entire program run faster that doing it serially.

This research aims to identify the time complexity of the threaded program that will compute for the Pearson Correlation Coefficient of a vector from an $n \times n$ matrix with an $n \times 1$ vector when using different number of processors. Additionally, the research explore the relationship of the elapsed time of a single threaded computation of the Pearson Correlation Coefficient and the threaded version of the program. Finally, the research investigates the feasibility and performance implications of employing row partitioning of the matrix as an alternative to the column-based division.

2. METHODOLOGY

2.1. Threaded Pearson Correlation Coefficient Calculation Algorithm

The program was implemented in C Programming Language. Below is a pseudocode of the program starting from the thread creation.

```
for j from 0 to num_threads - 1 do
    arguments[j].mat := submatrices[j]
    arguments[j].vecY := vecY
    arguments[j].row := n

    if j == num_threads - 1 and
    num_threads != 1 then
        arguments[j].col := cols +
    excess_column
    else
        arguments[j].col := cols

    #create threads and pass the
    pearson_cor_thread function
```

```
end for

#wait all the threads to finish

index := 0
for i from 0 to num_threads - 1 do
    vecX[i] := arguments[i].result
    column := cols
    if i == num_threads - 1 and
    num_threads != 1 then
        column += excess_column
    end if

    for j from 0 to column - 1 do
        finalR[index] := vecX[i][j]
        index++
    end for
end for
```

Listing 1: Threaded Pearson Correlation Coefficient Program

The algorithm and code structure for computing the Pearson Correlation Coefficient remain unchanged from the original implementation (LRP01). However, two significant modifications were introduced to enhance performance and facilitate concurrent computation. Firstly, the matrix is partitioned into submatrices with dimensions of $n \times n/t$, where t represents the number of desired threads. This division allows for parallelization of the computation, thereby potentially reducing the overall runtime.

The elapsed time for the calculation is measured using gettimeofday() function under the sys/time.h library. The first call of gettimeofday() is stored in the start_time. After saving the time, the implementation of the presented pseudocode is executed. After the execution, the gettimeofday() is called again, the value is stored at end_time. The difference of the end_time and start_time is then printed in the console.

The threads are created using p_threads. The program will create a t number of threads, these threads will execute the function to compute the Pearson Correlation Coefficient of the given submatrix and vector. After all the threads have finished, the result of all the threads will be stored in a single vector to replicate the behavior of the serial Pearson Correlation Coefficient algorithm.

2.2. System Specifications

The experiments were conducted on a Laptop Computer with the following specifications

- Processor: Intel(R) Core(TM) i5-10300H @2.50GHz, 2496Mhz, 4 Cores, 8 Logical Processors
- Memory: 16 GB DDR4 RAM, 2933MHz
- Storage: 256 GB SSD; 500 GB SSD
- Operating System: Windows 10
- Compiler: GCC version 9.4.0

3. RESULTS AND DISCUSSION

The program was run three times for each given t where $n = 25,000$, and the average elapsed time of those runs were recorded. The t used for each run of the program is increasing, starting from 2^0 up to 2^6. The recorded and averaged elapsed time of the program started with 1 thread.

Since using threads to compute the Pearson Correlation Coefficient of a matrix and vector will improve the performance, it is worth noting that the time complexity of this algorithm will change. From the original complexity of $O(n^2)$ presented in LRP01, if the algorithm will use n concurrent processors where each processor is assigned a single column from the original matrix, the program should have a complexity $O(n)$ since instead of going through n columns in the pearson correlation function, it will just run 1 time for the column assigned, therefore the loop will just run n times giving it a time complexity of $O(n)$.

Since the number of threads can be specified in this specific program if the threads or concurrent processors used are $n/2$ or even $n/8$ the program's time complexity will still be $O(n^2)$. The processor assignment at n/m concurrent processor is not apparent since it can have different variations that may result in varying running times. Being said that, the actual elapsed time of computation may vary but the time complexity of the program when using m concurrent processors where $n >> m$ will still be $O(n^2)$.

n	t	**Time Elapsed**		
		Run 1	**Run 2**	**Run 3**
25,000	1	36.108564	37.324175	36.151223
25,000	2	17.780378	18.355845	18.663757
25,000	4	11.371611	11.8761	12.214553
25,000	8	10.205692	10.753359	10.881947
25,000	16	10.584404	10.370843	10.313081
25,000	32	8.557063	9.559363	8.872483
25,000	64	7.642162	7.043031	7.209701

Table 1: Program Run Time

The table shows the elapsed time of the program given a value of t with $n = 25,000$. It shows the run times for each execution of the program given a t, recorded in seconds.

n	t	**Average**
25,000	1	36.52798733
25,000	2	18.26666
25,000	4	11.82075467
25,000	8	10.613666
25,000	16	10.422776
25,000	32	8.996303
25,000	64	7.298298

Table 2: Average Program Runtime

The table shows the average of the 3 runs of the program given t. The average elapsed time of the threaded program when using 1 thread is close to the average of the serial program presented in LRP01 but is slightly higher. This is due to the thread creation that adds more time before the completion of the computation. Additionally, copying the result of the threaded pearson correlation function into a single vector adds processing time. Since these processes are needed to finish before taking the end time of the program, the serial program with no threads will perform faster than the threaded program when only a single thread is utilized.

n	t	**Time Elapsed**			**Average**
		Run 1	**Run 2**	**Run 3**	
30,000	1	65.27711	67.144987	66.148919	66.19033867
30,000	2	29.914099	29.681085	30.243617	29.946267
30,000	4	19.469606	20.342522	19.4096	19.740576
30,000	8	15.326602	16.010352	15.948165	15.76170633
30,000	16	15.080807	15.435942	15.549475	15.355408
30,000	32	14.168716	14.49489	14.25455	14.306052
30,000	64	11.522644	11.817581	12.086405	11.80887667

Table 3: Program Run Time and Average $n = 30,000$

n	t	**Time Elapsed**			**Average**
		Run 1	**Run 2**	**Run 3**	
40,000	1	167.101241	166.870838	144.640555	159.5375447
40,000	2	78.261832	77.879559	78.202537	78.11464267
40,000	4	37.675013	36.555502	37.169577	36.8625395
40,000	8	32.251218	32.485674	29.839035	31.525309
40,000	16	30.612469	31.996027	31.460362	31.356286
40,000	32	27.71712	28.691923	27.626579	28.011874
40,000	64	24.388241	25.261706	24.572146	24.74069767

Table 4: Program Run Time and Average $n = 40,000$

The tables above show the run time of the program and its average given t when $n = 30,000$ and $n = 40,000$. Similar to what happened in the LRP01, the machine was able handle up to $n = 50,000$. Additionally, since the memory of the machine remained the same, the program was not able to handle $n = 100,000$. The OOM killer terminated the process, meaning it was not able to allocate enough memory for the program[2]. Graphing the average run time of the program given t, where $n = 25,00$, this graph will be produced.

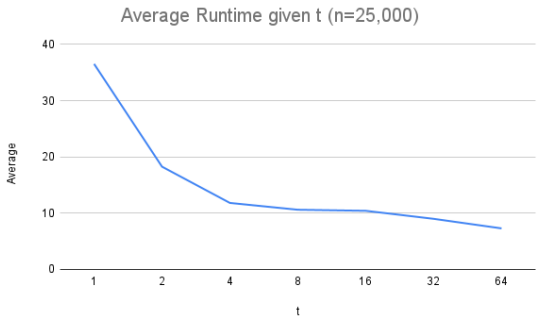

Figure 1: Average Run Time Graph

It is noticeable that as the number of threads increases, the runtime of the program decreases. It can also be observed in the graph that there is a steep decrease in the average runtime of the program from $t = 1$ to $t = 2$. Additionally, the decrease in runtime is almost half when the program utilizes 4 threads from 2 threads. As the number of threads used in the program gets closer to 64, the line in the graph almost becomes flat, indicating a little improvement on the performance of the program. With that being said, utilizing more threads such as $t = n$ threads might not be as effective as using 4 or 8 threads. In this context and with the machine used for experiments, it might not be possible to have n threads since each threads have their own stack to store local variables[3] it would take up a lot memory and the entire process will eventually be stopped, similar to what happens when the process takes up a large amount of memory when calculating for a large matrix. The approach used in the results above was a column based partitioning, where the matrix is divided into an $n \times n/t$. Another approach is to divide the matrix into submatrices with the dimension $n/t \times n$. To achieve the same result using this approach, the matrix can be transposed before dividing it into submatrices. The result of the row based partitioning can be seen below.

n	t	**Time Elapsed**			**Average**
		Run 1	**Run 2**	**Run 3**	
25,000	1	6.202618	6.250535	6.141056	6.198069667
25,000	2	3.402246	3.473677	3.463179	3.446367333
25,000	4	1.908669	1.936638	1.940958	1.928755
25,000	8	1.308315	1.653906	1.585621	1.515947333
25,000	16	1.222652	1.242717	1.328808	1.264725667
25,000	32	1.266445	1.213006	1.51758	1.332343667
25,000	64	1.239891	1.266317	1.385684	1.297297333

Table 5: Program Run Time and Average of $n/t \times n$ Submatrices $n = 25,000$

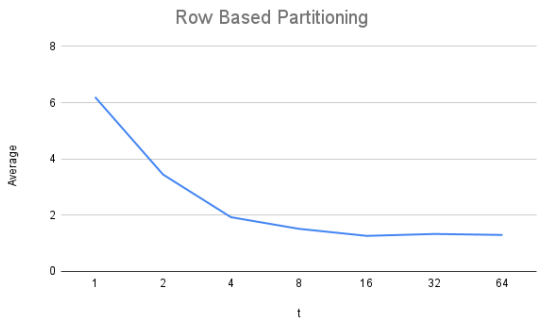

Figure 2: Average Run Time Graph of $n/t \times n$ Submatrices

It is noticeable that this approach performs significantly faster compared to the column-based computation. As seen in the graph, the process completed in approximately 6 seconds when using a single thread, which is already faster than the completion time observed with 64 threads in the column-based partition approach. Despite the substantial performance difference, the trend observed in this graph resembles that of the column-based approach. This indicates that the performance improvement of the program diminishes as the number of threads increases, regardless of the approach used. The significant increase in performance of the row-based computation is due to the way C stores data. Arrays in C are stored in row-major order, meaning that elements in the same row are stored contiguously in memory. This results in to a faster accessing of values due to better memory locality[4].

In conclusion, the study proved that parallelization of a serial program can significantly improve its performance. This was showed in the threaded approach in solving the Pearson Correlation Coefficient of a Matrix and a Vector. The diminishing improvement of performance as the number of threads increases has been observed throughout the experiments. Additionally, the time complexity of the function to solve the coefficient when using varying number of threads have been explored. Lastly, the comparison between the performance of row-based and column-based partitioning showed that row-based accessing of values performs better compared to column-based accessing, at least for programs implemented in C.

References

[1] GfG, "Multithreading in c," GeeksforGeeks, Jan. 06, 2023. https://www.geeksforgeeks.org/multithreading-in-c/

[2] "Linux Out of Memory Killer - Knowledge Base," Neo4j Graph Data Platform. [Online]. Available: https://neo4j.com/developer/kb/linux-out-of-memory-killer/

[3] "Operating systems: threads." https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

[4] K. D. Cooper and L. Torczon, "Code shape," in Elsevier eBooks, 2012, pp. 331–403. doi: 10.1016/b978-0-12-088478-0.00007-4.