Runtime-efficient Threaded Pearson Correlation Coefficient

John Rommel B. Octavo

***Abstract*—The study investigates the efficiency of multithreaded programming in computing the Pearson correlation coefficient of a matrix and a vector. The performance of the serial program was compared to a multithreaded program that utilizes both the column-wise and the row-wise matrix division. A performance increase of 1.76% is observed when it utilizes the column-wise matrix division, while 46% when it uses the row-wise matrix division. When utilizing row-wise matrix division, the program can run up to $t = n$ thread. The program that utilizes the column-wise matrix division struggles to run when threads are increased. It is also observed that the average runtime of a multithreaded program is influenced by how the program accesses the memory. Moreover, the program can only run at $n = 40000$ maximum because of the hardware limitations.**

***Index Terms*—Pearson correlation coefficient, multithreading, matrix computation, POSIX thread**

I. Introduction

THE computation of the Pearson correlation coefficients of a matrix and a vector can be done using different algorithms and techniques. One of the most common ways is through multithreaded programming. In multithreading, threads are used to simultaneously calculate the independent parts of the whole computational process [1].

Multithreading executes the independent computations in a computer program using threads [2]. As the data increases in a multithreaded program, the computational complexity also increases. However, the number of independent computations can increase the efficiency of the multithreaded program since it uses threads to simultaneously compute the independent calculations [3]. This is particularly pertinent when dealing with large matrices and vectors that have independent computations, as in the case of computing Pearson correlation coefficients.

The Pearson Correlation Coefficient, often denoted as r, is a statistical measure that quantifies the strength and direction of the linear relationship between two variables [4]. It is a dimensionless value that ranges from -1 to 1, where $r = 1$ indicates a perfect positive linear relationship, $r = -1$ indicates a perfect negative linear relationship, and $r = 0$ indicates no linear relationship between the variables.

In this research, we will use the formula for the Pearson Correlation Coefficient shown in Equation 1. It consists of an $m \times n$ matrix X with m rows and n columns, a $m \times 1$ vector y, and a $1 \times n$ vector r that represents the Pearson Correlation Coefficient of the columns X and y.

$$r(j) = \frac{m[X_j y] - [X]_j[y]}{[\sqrt{m[X^2]_j - ([X])^2} \cdot \sqrt{m[y^2] - ([y])^2}} \quad (1)$$

where,

$$[X]_j = \sum_{i=1}^{m} X(i,j) = X(1,j) + X(2,j) + \cdots + X(m,j),$$

$$[X^2]_j = \sum_{i=1}^{m} X(i,j)^2 = X(1,j)^2 + X(2,j)^2 + \cdots + X(m,j)^2,$$

$$[y] = \sum_{i=1}^{m} y(i) = y(1) + y(2) + \cdots + y(m),$$

$$[y^2] = \sum_{i=1}^{m} y(i)^2 = y(1)^2 + y(2)^2 + \cdots + y(m)^2,$$

$$[X_j y] = \sum_{i=1}^{m} X(i,j)y(i) = X(1,j)y(1) + \cdots + X(m,j)y(m).$$

The computation of Pearson correlation coefficients for large matrices presents computational challenges especially when serial programming is used. In an attempt to boost the efficiency of the computation, multithreading was used in this paper. It is important to investigate the efficiency of computer algorithms in handling large-scale data computations. As the dimensions of the data increase, so does the computational complexity. This necessitates efficient algorithms and computational resources to handle such tasks effectively.

In this paper, the computation of the Pearson correlation coefficient of a matrix and a vector will be done through multithreaded programming. Threading was used to compute the independent calculations in the matrix calculation. Empirical investigation through experiments was conducted to assess the efficiency of the threaded computation for varying matrix sizes.

A. Statement of the Problem

Multithreaded programming employs threading to simultaneously calculate independent computations [1]. Existing research indicates that the computational complexity of multithreaded programming is influenced by the number of independent computational calculations in a program. However, there remains a need to comprehensively investigate how employing multithreaded programming in the computation of the Pearson correlation coefficient of a matrix and a vector impacts certain factors that affect the computational complexity. Moreover, thorough experiments can extract essential findings that can be used to optimize the computation.

B. Significance of the Study

This research aimed to investigate how the computation of the Pearson correlation coefficient of a matrix and a vector

using multithreaded programming impacts the program's overall computational complexity. It is beneficial in optimizing computational processes, and crafting an efficient algorithm design, and can be used for further studies.

Optimization of the computational process in solving the Pearson correlation coefficient of a matrix and a vector can be achieved through experiments. This study may give substantial details on how the computational process can be optimized given the restraints of threaded serial programming in solving the problem.

Future research and studies may benefit from the results of this study. This may contribute to the development of efficient algorithms, theories, or programs that may compute the Pearson correlation coefficient of a matrix and a vector effectively.

C. Objectives of the Study

This study aims to assess the performance of the algorithm that computes the Pearson correlation coefficient of a matrix and a vector. It specifically focuses on the use of serial programming in the development of this algorithm.

The specific objectives of this research study are the following:

1) To develop a multithreaded program that will compute the Pearson correlation coefficient of a matrix and a vector,
2) To understand the relationship between matrix data size, program's computational time, and number of threads, and;
3) To differentiate the performance between serial programs and multithreaded programs in the computation.

D. Scope and Limitations of the Study

This study aims to investigate the efficiency of multithreaded programming techniques in computing the Pearson correlation coefficient vector of a matrix and a vector. Computational analysis, algorithmic optimization, empirical evaluation, and comparative analysis of different program outputs are used to achieve the goal of this research. Computational analysis is conducted to measure the performance of the multithreaded programming approach across a varying number of threads.

However, this research is only limited to the investigation of a computer program that computes the Pearson correlation coefficient vector of a matrix and a vector. This means that its findings are limited to specific computation as it cannot assume other computation tasks or domains. The hardware and software resources also play a huge role in the outcome or results of the computation. Variations in data characteristics and computational environments may influence the applicability of the findings to different scenarios. Furthermore, while the study includes computational complexity analysis and algorithmic optimizations, it may not comprehensively cover all its aspects or strategies.

Despite these limitations, the study aims to provide valuable guidance for algorithm design, optimization strategies, and future research directions in computational optimization in serial programming.

E. Date and Place of the Study

The study will be conducted in the 2nd semester of the academic year 2023-2024 at the Institute of Computer Science, University of the Philippines Los Baños, Laguna, Philippines.

II. Review of Related Literature

In the modern age of computation, hardware, and software architecture design have evolved and is rapidly improving. This development paves the way for revolutionizing the computation process of computers. Complex calculations can now be handled by computers and memory constraints are mostly solved by continuous innovations [5].

Serial computing is dependent on data transmission through the hardware [6]. The development of the hardware positively affects the efficiency of serial computation. Despite this advancement, serial computation still faces limitations primarily on hardware constraints. Hardware development may somehow reach its limitations and thereby stunt the efficiency of serial programming [7].

To solve this dilemma, a new computational process is introduced. Modern hardware and software architecture design of computers can now handle multiple processes at a time. This computational technique is called parallel computing. It is the simultaneous execution of independent processes using multiple computing resources. Parallel computing divides the problem into discrete parts that can be solved concurrently and each of these divisions is further broken down into a series of instructions [8].

In the study about the significance of parallel computation over serial computation, the researchers emphasized the advantage of parallel computation over serial computation in terms of processing time. The researchers affirm that parallel computation reduces a significant amount of execution time when it is applied to several algorithms. However, they also cited that there are factors to achieve an ideal processing performance in parallel computation. This includes synchronization, fault tolerance, and distribution [9].

In a 1994 study on serial and parallel computing, the researcher explains that parallel computing is not generally better than serial programming [10]. While both methods have drawbacks, the researcher believes that there are factors to consider when it comes to achieving optimal results. The study particularly cites resource restrictions as a factor that affects computational processing.

One common technique in the parallel approach to programming is multithreading. The Central Processing Unit of a computer provides multiple concurrent threads of execution [11]. Threads are used to simultaneously execute independent computations in a program [12]. This approach in the computation may boost the efficiency of the program. However, there are still factors that affect the program's overall performance and efficiency.

This study deals with the computation of the Pearson correlation coefficient of a matrix and a vector using a multithreaded program. The performance of the program based on the number of threads used was analyzed.

III. Methodology

A. Development Tools

The experiments for the computation of the Pearson correlation coefficient of a matrix and a vector will be conducted on a computer with the following specifications:

- Operating System: Windows 11 Home Single Language
- Processor: Intel® Core™ i5-1035G1 CPU @ 1.00GHz 1.19 GHz
- Memory: 32 GB DDR3 L

The computer program was developed on Visual Studio Code, and it uses the C programming language. The C programming language was used since compiled language has better performance in terms of threaded computation.

B. Complexity of Computation

Determining the theoretical complexity of computing the Pearson correlation coefficient of a matrix and a vector is an essential step in developing an algorithm and conducting experiments. These theoretical calculations provided the data that is the basis of the experiments.

In the calculation of its complexity, the sum of each row of X, the sum of each square element in each row of X, and the sum of each element in each row of X and y all have an $O(n^2)$ complexity. Both the sum of each row of X and the sum of each square element in each row of X require iterating over each element in each row once. The sum of each element in each row of X and y requires iterating over each element in each row of X and once over the elements in y.

Moreover, the sum of elements of y, the sum of each square element of y, and the other arithmetic operations in the Pearson correlation coefficient all have an $O(n)$ complexity. This is because it requires iterating over each element once and the complexity of the arithmetic operations is linear.

Analyzing each part of the calculation of the Pearson correlation coefficient of a matrix and a vector, the overall complexity of this calculation is $O(n2)$. Table 1 shows the detail of the computation's complexity.

TABLE I: Computation Complexity of the Program

Operation	**Complexity**
Sum of each row of X	$O(n^2)$
Sum of elements of y	$O(n)$
Sum of each square element in each row of X	$O(n^2)$
Sum of each square element in y	$O(n)$
Sum of each element in each row of X and y	$O(n^2)$
Arithmetic operations in Pearson Correlation Coefficient	$O(n)$
Overall Complexity of the Program	$O(n^2)$

In multithreaded programming, if the computation of one column M and the vector y from an $n \times n$ matrix and a vector y is assigned to n processors, the overall complexity remains $O(n^2)$. Likewise, even when fractions of n concurrent processors are used in the computation of the Pearson correlation coefficient, the complexity is always $O(n^2)$. Table 2 shows the complexity of the program using different numbers of concurrent processors.

The simultaneous computation of independent calculations in the program will still require the computation of the $pearson\_cor$ function which has $O(n^2)$ complexity. The time complexity of the program remains the same even when different numbers of concurrent processors are used in the computation.

TABLE II: Complexity of the Program in Different Numbers of Concurrent Processors

Number of Processors	Complexity
n	$O(n^2)$
$n/2$	$O\left(\frac{n^2}{2}\right) = O(n^2)$
$n/4$	$O\left(\frac{n^2}{4}\right) = O(n^2)$
$n/8$	$O\left(\frac{n^2}{8}\right) = O(n^2)$
$n >> m$	$O\left(\frac{n^2}{m}\right) = O(n^2)$

C. Calculation of the Program's Elapsed Time

In the calculation of the program's elapsed time, the $sys/time.h$ header was used in the program. This system-specific approach in the calculation of the program's elapsed time provides precision and finer granularity in the calculation.

In each variation of the number of threads t in a 25000×25000 matrix and a 25000×1 vector, the time elapsed was calculated in three runs. The average time of these three runs was noted. Listing 1 shows the code of how the elapsed time is calculated in the program.

Listing 1: Calculation of the Program's Elapsed Time

```
// Measure time before creating threads
gettimeofday(&begin, NULL);

for(int k = 0; k<t; k++){
    pthread_create(&threads[k], NULL,
        pearson_cor, (void*)&thread_data[k
        ]);
}

// Wait for threads to finish
for (int i = 0; i < t; i++) {
    pthread_join(threads[i], NULL);
}

// Recreate the correct r from the output
    of each threaded pearson_cor
for(int k=0; k<t; k++){
    for (int i = 0; i < t; i++) {
        for (int j = 0; j < (k == t - 1 ?
            n - (n / t) * k : n / t); j
           ++) {
            v[i * (n / t) + j] =
                thread_data[i].result[j];
        }
    }
}

// Measure time after creating threads
gettimeofday(&end, NULL);
```

```
timeTaken =  (end.tv_sec - begin.tv_sec)
    + ((end.tv_usec - begin.tv_usec)
    /1000000.0);
printf("Time taken: %f seconds\n",
    timeTaken);
```

D. Threading and Argument Structure

In the implementation of threads, the POSIX Thread library or the $pthread.h$ library was used. Since the $pearson\_cor$ function has multiple arguments, the library requires a structure to handle these arguments. Listing 2 shows the structure of the data as an argument to the $pearson\_cor$ function.

Listing 2: Argument Structure

```
typedef struct {
    float **submatrix;
    float *y;
    int rows;
    int col;
    int thread_id;
    float* result;
} ThreadData;
```

In Listing 2, the submatrix is the local copy of the matrices derived from the column-wise or row-wise division of the matrix X. The y pointer points to the original copy of the vector y. On the other hand, the variables $rows$, col, and $thread\_id$, store the row number, column number, and thread number respectively. Lastly, the result pointer points to an allocated memory where the partial result of the calculation will be stored at the end of the computation.

E. The Column-Wise Division of Matrix

The randomly generated matrix X is transposed first before the matrix column-wise division. The transpose function ensures that the result of the column-wise computation is correct. The $n \times n$ matrix X is subdivided into the column-wise $n \times n/t$ submatrices, where n is the number of rows and t is the number of threads. Both n and t are user input. In the case where n/t has a remainder, the last thread gets the remaining $n \times 1$ vector submatrix. Listing 3 shows the column-wise division of the matrix.

Listing 3: The $n \times n/t$ Division of the Matrix

```
transposeMatrix(X, n);

for(int k=0; k<t; k++){
    thread_data[k].submatrix = (float**)
       malloc(n * sizeof(float*));

    if (thread_data[k].submatrix == NULL)
        {
        printf("Memory allocation failed
           .\n");
        return 1;
    }
    for (int i = 0; i < n; i++) {
        thread_data[k].submatrix[i] = (
           float*)malloc(n * sizeof(float
           ));
        if (thread_data[k].submatrix[i]
           == NULL) {
            printf("Memory allocation
               failed.\n");
            return 1;
        }
    }

    int start_col = (n * k) / t;
    int end_col = (n * (k + 1)) / t;
    if (k == t - 1) {
        end_col = n;
    }

    for (int i = 0; i < n; i++) {
         for (int j = start_col; j <
            end_col; j++) {
            thread_data[k].submatrix[i][j
                - start_col] = X[i][j];
        }
    }

    thread_data[k].y = y;
    thread_data[k].rows = n;
    thread_data[k].col = (k == t - 1 ? n
       - (n / t) * k : n / t);
    thread_data[k].thread_id = k;

    // Allocate memory for the result
    thread_data[k].result = (float*)
       malloc(thread_data[k].col * sizeof
       (float));
    if (thread_data[k].result== NULL) {
        printf("Memory allocation failed
           .\n");
        exit(EXIT_FAILURE);
    }
}
```

F. The Row-Wise Division of Matrix

The $n \times n$ matrix X is subdivided into the row-wise $n/t \times n$ submatrices, where n is the number of rows and t is the number of threads. Both n and t are user input. In the case where n/t is has a remainder, the last thread gets the remaining $1 \times n$ vector submatrix. Unlike the column-wise division of the matrix, the row-wise division need not be transposed. Listing 4 shows the row-wise division of the matrix.

Listing 4: The $n/t \times n$ Division of the Matrix

```
for(int k = 0; k < t; k++){
```

```
    int rows_per_thread = n / t;
    if (k == t - 1) {
        rows_per_thread += n % t; // Add
            the remainder to the last
            thread
    }

    thread_data[k].submatrix = (float**)
        malloc(rows_per_thread * sizeof(
        float*));
    if (thread_data[k].submatrix == NULL)
         {
        printf("Memory allocation failed
            .\n");
        return 1;
    }
    for (int i = 0; i < rows_per_thread;
        i++) {
        thread_data[k].submatrix[i] = (
            float*)malloc(n * sizeof(float
            ));
        if (thread_data[k].submatrix[i]
            == NULL) {
            printf("Memory allocation
                failed.\n");
            return 1;
        }
        // Copy rows to the submatrix
        int source_row = k * (n / t) + i;
            // Calculate source row in
            the original matrix
        for (int j = 0; j < n; j++) {
            thread_data[k].submatrix[i][j
                ] = X[source_row][j];
        }
    }

    thread_data[k].y = y;
    thread_data[k].rows = rows_per_thread
        ; // Each thread handles a
        fraction of the rows
    thread_data[k].col = n; // Each
        submatrix has all the columns of
        the original matrix
    thread_data[k].thread_id = k;

    // Allocate memory for the result
    thread_data[k].result = (float*)
        malloc(n * sizeof(float));
    if (thread_data[k].result == NULL) {
        printf("Memory allocation failed
            .\n");
        exit(EXIT_FAILURE);
    }
}
```

G. The $pearson\_cor$ Function

Listing 5 shows the $pearson\_cor$ function of the multi-threaded program. The $isColumnWise$ variable serves as a checker for the submatrix. The value of 1 indicates that the submatrix is column-wise, otherwise it is row-wise. This variable changes the flow of the calculation to correctly compute the Pearson correlation coefficient for both the column-wise and the row-wise submatrices.

The $math.h$ library is also used in computation, especially in exponentiations and square roots. Moreover, the result of the calculation is stored in the result variable which will be recreated to get the $1 \times n$ vector r that contains the answers.

Listing 5: The pearson_cor Function

```
void* pearson_cor(void* arg) {
    ThreadData* data = (ThreadData*)arg;

    int isColumnWise = (data->col < data
        ->rows) ? 1 : -1;

    // Calculate Pearson Correlation
        Coefficients for the assigned
        columns
    for (int j = 0; j < ((isColumnWise ==
         1) ? data->col : data->rows); j
        ++) {
        float sum_X = 0.0, sum_X_sq =
            0.0, sum_y = 0.0, sum_y_sq =
            0.0, sum_Xy = 0.0;

        for (int i = 0; i < ((
            isColumnWise == 1) ? data->
            rows : data->col); i++) {
            float submatrix_element = (
                isColumnWise == 1) ? data
                ->submatrix[i][j] : data->
                submatrix[j][i];
            sum_X += submatrix_element;
            sum_X_sq += pow(
                submatrix_element, 2);
            sum_y += data->y[i];
            sum_y_sq += pow(data->y[i],
                2);
            sum_Xy += submatrix_element *
                 data->y[i];
        }

        float numerator = ((isColumnWise
            == 1) ? data->rows : data->col
            ) * sum_Xy - sum_X * sum_y;
        float denominator = sqrt((((
            isColumnWise == 1) ? data->
            rows : data->col) * sum_X_sq -
             pow(sum_X, 2)) * (((
            isColumnWise == 1) ? data->
            rows : data->col) * sum_y_sq -
             pow(sum_y, 2)));
```

```
        if (denominator != 0) {
            data->result[j] = numerator /
                denominator;
        } else {
            data->result[j] = 0; // If
                denominator is zero, set
                correlation coefficient to
                 zero
        }
    }

    pthread_exit(NULL);
}
```

H. Thread Creation, Joining, and Vector Recreation

Listing 6 shows the code of how threads are created and joined. The for loop serves as the method in creating and joining t number of threads. The thread function is the $pearson\_cor$ function while the arguments are stored in the $thread\_data$ array. In the thread join part, the program will wait for all threads to finish the calculation. After all threads are finished, the $1 \times n$ vector r will be recreated based on the result variable from the $thread\_data$ array.

Listing 6: Thread Creation and Join label

```
for(int k = 0; k<t; k++){
        pthread_create(&threads[k], NULL,
            pearson_cor, (void*)&
           thread_data[k]);
    }

// Wait for threads to finish
for (int i = 0; i < t; i++) {
    pthread_join(threads[i], NULL);
}

// Recreate the correct r from the output
    of each threaded pearson_cor
for(int k=0; k<t; k++){
    for (int i = 0; i < t; i++) {
        for (int j = 0; j < (k == t - 1 ?
            n - (n / t) * k : n / t); j
           ++) {
            v[i * (n / t) + j] =
               thread_data[i].result[j];
        }
    }
}
```

IV. Results and Discussion

A. Serial Program

To be able to assess the comparison between a serial and multithreaded program, the runtime of the serial program in varying data sizes is noted. Unlike the multithreaded program, the serial program doesn't use any threads to simultaneously compute for independent calculations. This means that the Pearson correlation coefficient of a matrix and a vector is computed sequentially. Table 3 shows the runtime of the serial program in varying data sizes.

TABLE III: Runtime of the Serial Program

n	Time Elapsed (seconds)			Average Runtime
	Run 1	Run 2	Run 3	
100	0.001024	0.001057	0.001015	0.001032
200	0.003526	0.003209	0.001505	0.002746667
300	0.002994	0.004021	0.004949	0.003988
400	0.005648	0.005444	0.006045	0.005712333
500	0.008366	0.008621	0.00974	0.008909
600	0.012041	0.012663	0.012381	0.012361667
700	0.016579	0.016867	0.016701	0.016715667
800	0.023015	0.025229	0.023438	0.023894
900	0.033973	0.030321	0.030419	0.031571
1000	0.036794	0.039971	0.038585	0.03845
2000	0.142442	0.138284	0.142915	0.141213667
4000	0.543644	0.543115	0.596527	0.561095333
8000	2.242916	2.228568	2.316354	2.262612667
16000	9.653118	9.600131	9.921436	9.724895
20000	14.471427	14.665314	15.012296	14.71634567
25000	25.19988	25.16249	26.06134	25.47457
30000	33.977622	35.474561	34.463082	34.63842167
40000	60.520226	61.025746	61.633254	61.059742

The serial program can only be run on $n = 40000$ with an average runtime of 61.0597 seconds. It is also worth noting that the runtime increases as the data increases. In $n = 25000$, the serial program achieves a runtime of 25.4746 seconds. This result will be compared to the performance of the multithreaded program.

B. Column-Wise Matrix Division

In the column-wise matrix division, the running time for $n = 25000$ in varying thread sizes is noted in Table 4. When one thread is used, the average runtime is 29.5155 seconds. This running time is slightly higher than in serial programs. This can be explained by the additional cost of using thread in the program.

It can also be observed that at $t = 2$ and $t = 4$, the average runtime increases. This can be explained by the cache misses caused by accessing non-contiguous data blocks. When accessing data in a $n \times n/t$ column-wise matrix, the memory access is prone to cache misses. This may explain why the performance in a column-wise threaded program is statistically the same as using the serial program.

TABLE IV: Runtime of the Multithreaded Program (Column-Wise Matrix Division)

n	t	Time Elapsed (seconds)			Average Runtime
		Run 1	Run 2	Run 3	
25000	1	28.978937	29.902496	29.665039	29.51549
25000	2	54.302822	55.187176	55.407127	54.96571
25000	4	35.114246	36.796902	36.611572	36.17424
25000	8	24.774239	24.918467	25.707228	25.13331
25000	16	24.480013	25.545019	25.539494	25.18818
25000	32	24.368092	25.058079	25.654882	25.02702
25000	64	25.135925	25.070208	25.226494	25.14421

At t = 8, the runtime of the column-wise threaded program stops improving. Compared to the serial program, the perfor-

mance only increases by approximately 1.76%. This means that using threads does not always improve the performance of a program. The programmer must also be wary of how the memory is accessed. In this case, the memory access of a column-wise matrix is not efficient. Figure 1 shows the graph of the threads versus the average runtime of the program that uses the column-wise matrix division.

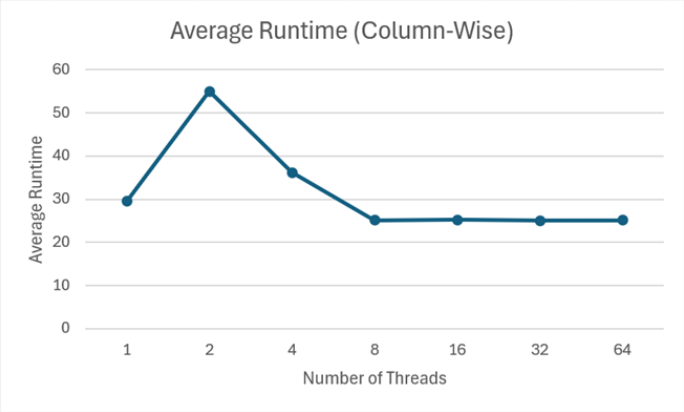

Fig. 1: Threads Vs. Average Runtime (Column-Wise)

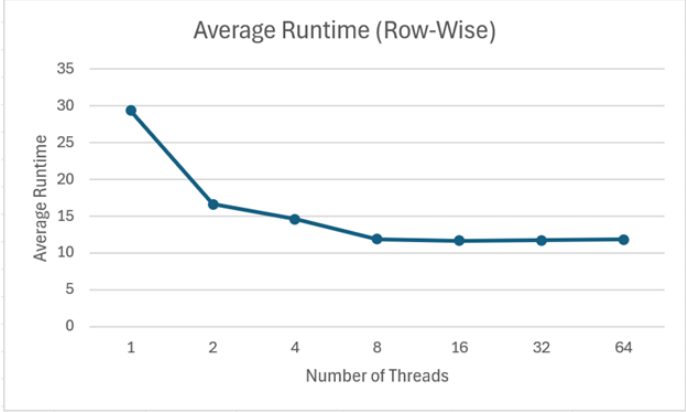

Fig. 2: Threads Vs. Average Runtime (Row-Wise)

C. Row-Wise Matrix Division

Unlike in a column-wise matrix, a $n/t \times n$ row-wise matrix accesses a contiguous block of memory in the computation. This memory access maximizes the cache hits thereby making it efficient. Because of this, it is expected that the runtime of the program that uses row-wise matrix division is better than in using column-wise matrix division.

The runtime of the multithreaded program that uses row-wise matrix division is noted in Table 5. The result presented in the table is better than in using column-wise matrix division. At $t = 1$, the average runtime is slightly higher than the average runtime of the serial program at $n = 25000$. As mentioned previously, this phenomenon is explained by the additional cost caused by the usage of threads in the program.

TABLE V: Runtime of the Multithreaded Program (Row-Wise Matrix Division)

n	t	Time Elapsed (seconds)			Average Runtime
		Run 1	Run 2	Run 3	
25000	1	28.946037	29.389626	29.720388	29.352017
25000	2	16.422895	16.719954	16.671743	16.604864
25000	4	14.305547	14.838234	14.649723	14.59783467
25000	8	11.814496	11.873459	11.886233	11.85806267
25000	16	11.934187	11.30075	11.717256	11.650731
25000	32	11.788196	11.734202	11.608797	11.71039833
25000	64	11.767674	11.864824	11.916784	11.84976067

It is observed in the table that as the threads increase, the runtime of the program decreases. This means that using threads in a program that utilizes row-wise matrix division makes it more efficient compared to both the serial program and the threaded program that utilizes column-wise matrix division.

At $t = 8$, similar observations on column-wise matrix division were noted. At this point, the program stops improving. Based on performance, using a row-wise matrix division in a threaded program results in an increase in performance by 46%. This means that using threads may only be effective in increasing the performance of a program if the memory access is efficient. Figure 2 shows the graph of the threads versus the average runtime of the program that uses the row-wise matrix division.

D. Maximum Handled Data

In the serial program, the highest data that the program can handle is $n = 40000$. This is also true in the multithreaded program. Table 6 shows the performance of the multithreaded program either using column-wise or row-wise matrix division. It can be observed that even if the average runtime is better on $n = 30000$ and $n = 40000$, the program cannot run when data $n = 50000$.

The maximum handled data in a program may be explained by the hardware limitation by which the program is running. Since both the serial and multithreaded program stopped working at $n = 50000$ it is safe to assume that the machine can't handle data beyond this point.

TABLE VI: Runtime of the Multithreaded Program (Row-Wise Matrix Division)

Division	n	t	Time Elapsed (seconds)			Average
			Run 1	Run 2	Run 3	
Col-Wise	30000	8	38.81	43.96	44.14	42.30
Col-Wise	40000	8	66.76	68.31	67.33	67.47
Col-Wise	50000	8	-	-	-	-
Col-Wise	100000	8	-	-	-	-
Row-Wise	30000	8	15.59	16.62	15.83	16.01
Row-Wise	40000	8	27.50	28.22	28.89	28.20
Row-Wise	50000	8	-	-	-	-
Row-Wise	100000	8	-	-	-	-

E. Maximum Handled Thread

In measuring the maximum threads, both the column-wise and row-wise matrix divisions were tested. On using the column-wise matrix division, the multithreaded program cannot handle even $n/8$ threads in a data $n = 25000$. However, when the program uses the row-wise matrix division, it can run on the maximum threads of $t = n$.

Since column-wise matrix division is not efficient in terms of memory access, it may be the reason why the program can't run on more threads. The creation of more threads entails an increase in memory usage. Moreover, column-wise matrix division results in frequent cache misses which may be the reason why the program cannot run even in $n/8$ threads. Table 7 shows the result of analyzing the maximum threads that the program can handle.

TABLE VII: Runtime of the Multithreaded Program (Row-Wise Matrix Division)

Division	n	t	Time Elapsed (seconds)			Average
			Run 1	Run 2	Run 3	
Col-Wise	25000	8	-	-	-	-
Col-Wise	25000	8	-	-	-	-
Col-Wise	25000	8	-	-	-	-
Col-Wise	25000	8	-	-	-	-
Row-Wise	25000	8	46.14	41.68	46.67	44.83
Row-Wise	25000	8	23.62	23.74	22.75	23.37
Row-Wise	25000	8	14.53	14.02	13.37	13.94
Row-Wise	25000	8	12.65	13.80	10.83	12.43-

V. Conclusion

In this study, a multithreaded program was developed in C programming language to solve the Pearson correlation coefficient of a matrix and a vector. The performance of the computation of the serial program was compared to the performance of the multithreaded program. Two methods in matrix division were implemented. Firstly, the column-wise matrix division, and secondly, the row-wise matrix division.

In the performance of the multithreaded program that uses the column-wise matrix division, the average running time at $n = 25000$ only improves at approximately 1.76%. This poor performance can be linked to non-contiguous memory access of column-wise matrix that results in frequent cache misses. On the other hand, there is an observed 46% increase in performance when the multithreaded program uses row-wise matrix division. This increase in performance is linked to the efficient memory access that results in frequent cache hits.

The program can only run up to data $n = 40000$ because of hardware limitations. Moreover, when the program uses the row-wise matrix division, it can achieve the maximum number of threads $t = n$ while in the column-wise matrix, the program struggles to run on $n/8$ threads.

References

[1] P. Kirvan, "What is multithreading?," *WhatIs*, https://www.techtarget.com/whatis/definition/multithreading, Accessed March. 14, 2024.

[2] R. Buyya, C. Vecchiola, and S. T. Selvi, "Chapter 6 - Concurrent Computing: Thread Programming," in *Mastering Cloud Computing*, Rajkumar Buyya, Christian Vecchiola, and S. Thamarai Selvi (eds.), Morgan Kaufmann, Boston, pp. 171-210, 2013. ISBN: 978-0-12-411454-8. DOI: https://doi.org/10.1016/B978-0-12-411454-8.00006-1.

[3] T. Liu, C. Curtsinger, and E. D. Berger, "Dthreads: efficient deterministic multithreading," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, Association for Computing Machinery, New York, NY, USA, pp. 327–336, 2011. ISBN: 9781450309776. DOI: https://doi.org/10.1145/2043556.2043587. Available: https://doi.org/10.1145/2043556.2043587.

[4] D. L. Hahs-Vaughn, "Foundational methods: Descriptive statistics: Bivariate and Multivariate Data (correlations, associations)," *International Encyclopedia of Education(Fourth Edition)*, pp. 734–750, 2023. Available: https://doi.org/10.1016/b978-0-12-818630-5.10084-3.

[5] N. C. Thompson and S. Spanuth, "The decline of computers as a general purpose technology," vol. 64, pp. 64-72, 2021. doi: 10.1145/3430936.

[6] B. Barney, "Introduction to Parallel and Distributed Computing," Retrieved from Pace University, 2020. Available: https://csis.pace.edu/~marchese/SE765/L0/L0b.htm.

[7] A. R. Brodtkorb, T. R. Hagen, C. Schulz, and G. Hasle, "GPU computing in discrete optimization. Part I: Introduction to the GPU," *EURO Journal on Transportation and Logistics*, vol. 2, no. 1-2, pp. 129-157, 2013. Available: https://doi.org/10.1007/s13676-013-0025-1.

[8] M. Kumar, S. Jain, and Snehalata, "Parallel Processing using multiple CPU," *International Journal of Engineering and Technical Research (IJETR)*, vol. 2, no. 10, pp. 247-250, 2014.

[9] S. Rastogi and H. Zaheer, "Significance of parallel computation over serial computation," in *International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, 2016, pp. 2307-2310. doi: 10.1109/ICEEOT.2016.7755106.

[10] R. Kolisch, "Serial and parallel resource-constrained project scheduling," *European Journal of Operational Research*, vol. 90, pp. 320-333, 1996.

[11] J. Laudon, A. Gupta, and M. Horowitz, "Interleaving: a multithreading technique targeting multiprocessors and workstations," *SIGPLAN Not.*, vol. 29, no. 11, pp. 308–318, Nov. 1994. ISSN: 0362-1340. DOI: https://doi.org/10.1145/195470.195576. Available: https://doi.org/10.1145/195470.195576.

[12] R. Bianchini and B.-H. Lim, "Evaluating the Performance of Multithreading and Prefetching in Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 83-97, 1996. ISSN: 0743-7315. DOI: https://doi.org/10.1006/jpdc.1996.0109. Available: https://www.sciencedirect.com/science/article/pii/S0743731596901094.

John Rommel B. Octavo is currently pursuing a Bachelor of Science degree in Computer Science at the University of the Philippines Los Baños. His hobbies include cycling, playing piano and recorder flute, watching movies, reading religious literature, and programming. He dreams to be a software developer and fulfill his priestly vocation in the future.