Core-affine Threaded Computation of the Pearson Correlation Coefficient

John Rommel B. Octavo

***Abstract*—The study investigates the efficiency of a core-affined threaded program in computing the Pearson correlation coefficient of a matrix and a vector. The performance of both the serial and multithreaded programs was compared to the core-affined threaded program. It utilizes both the column-wise and the row-wise matrix division.**

Results show that in utilizing the column-wise matrix division, the core-affined threaded program is observed to have a decrease in performance of 24.36% compared to the serial program and 12.36% compared to the multithreaded program. In utilizing the row-wise matrix division, the program is observed to increase in performance by 36% compared to the serial program. However, when it is compared to the multithreaded program, the performance decreases by approximately 26.84%.

When utilizing row-wise matrix division, the program can run up to $t = n$ thread. On the other hand, when column-wise matrix division is used, the program struggles to run when threads are increased. It is also observed that the average runtime of the core-affined threaded program is influenced by how the program accesses the memory and how the CPU cores are utilized. Moreover, the program can only run at $n = 40000$ maximum because of the hardware limitations.

***Index Terms*—Pearson correlation coefficient, core-affine, multithreading, matrix computation, CPU core, 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 core-affined threaded computation. In multithreading, threads are used to simultaneously calculate the independent parts of the whole computational process [1]. Moreover, these threads are assigned to specific Central Processing Unit (CPU) cores to aim for better performance through parallel computation.

Multithreading executes the independent computations in a computer program using threads [2]. CPU cores serve as independent machines for parallel computation or execution. As the data increases in a core-affined threaded program, the computational complexity also increases. However, the number of independent computations can increase the efficiency of the core-affined threaded 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. Aside from the threads, the number of available and usable CPU cores can affect the overall performance of the core-affined threaded program.

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.

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

where,

$$\begin{aligned}
[X]_j &= \sum_{i=1}^{m} X(i,j) = X(1,j) + X(2,j) + \cdots + X(m,j), \\
[X^2]_j &= \sum_{i=1}^{m} X(i,j)^2 = X(1,j)^2 + X(2,j)^2 + \cdots + X(m,j)^2, \\
[y] &= \sum_{i=1}^{m} y(i) = y(1) + y(2) + \cdots + y(m), \\
[y^2] &= \sum_{i=1}^{m} y(i)^2 = y(1)^2 + y(2)^2 + \cdots + y(m)^2, \\
[X_j y] &= \sum_{i=1}^{m} X(i,j)y(i) = X(1,j)y(1) + \cdots + X(m,j)y(m).
\end{aligned}$$

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, core-affine threading 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 calculation of the Pearson correlation coefficient of a matrix and a vector will be done through a core-affined threaded program. Threading was used to compute the independent calculations in the matrix calculation. Moreover, these threads will be assigned to specific CPU cores of the machine to achieve parallel computation. Empirical investigation through experiments was conducted to assess the efficiency of the threaded computation for varying matrix sizes.

A. Statement of the Problem

Existing research indicates that the computational complexity of core-affined threaded programming is influenced by the number of independent computational calculations in a program and usable CPU cores. However, there remains a need to comprehensively investigate how employing core-affined threaded programming impacts certain factors that affect 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 core-affined threaded 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.

This research may also guide the formulation of efficient algorithm design for computing the Pearson correlation coefficient of a matrix and a vector. This efficient algorithm design may optimize the computational process given the restraints of core-affined threaded programming.

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 core-affined threaded programming in the development of this algorithm.

The specific objectives of this research study are the following:

1) To develop a core-affined threaded 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, number of threads, and number of usable CPU cores and;
3) To differentiate the performance between serial programs, multithreaded programs, and core-affined threaded programs in the computation.

D. Scope and Limitations of the Study

This study aims to investigate the efficiency of core-affined threaded 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 core-affined threaded 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 computation is the core-affine threaded programming. 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]. These threads are assigned to specific CPU cores in the machine. 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 core-affined threaded 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
- Physical CPU Cores: 4 Cores
- Logical CPU Cores: 8 Cores

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 $O(n^2)$. Table 1 shows the detail of the computation's complexity.

TABLE 1: 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 core-affine threaded 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.

TABLE 2: Complexity of the Program in Different Numbers of Concurrent Processors

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

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.

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);
```

```
// ... Other Codes Here ...

// 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 core-affine 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 and Core Assignment

In the core assignment, the machine has four physical cores and eight logical cores. The program will only use $n-1$ physical cores or three cores. Table 3 shows the logical core assignment in the machine. Since the program uses three physical cores, the created threads will only be assigned to logical core 0, logical core 2, or logical core 4. This assignment will ensure that the threads will only be assigned either to physical core 0, physical core 1, or physical core 2.

TABLE 3: Logical CPU Core Assignment

Physical CPU Core	Logical CPU Core
Core 0	Core 0 Core 1
Core 1	Core 2 Core 3
Core 2	Core 4 Core 5
Core 3	Core 6 Core 7

Listing 6 shows the code of how threads are created, assigned to a specific core, and joined. The for loop serves as the method in creating, assigning to specific core, and joining t number of threads. The $core\_ctr$ variable is responsible for the core assignment of the created threads. This ensures that the created threads will be assigned to either of the $n-1$ physical CPU cores of the machine.

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 Core Assignment label

```
for(int k=0; k<t; k++){
    if(core_ctr == 6){
        core_ctr = 0;
    }
    //Assign thread to a specific core
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(core_ctr, &cpuset);

    pthread_create(&threads[k], NULL,
       pearson_cor, (void*)&thread_data[k
       ]);
    pthread_setaffinity_np(threads[k],
       sizeof(cpu_set_t), &cpuset);

    core_ctr += 2;
}

// 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. 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 33.67862 seconds. This running time is slightly higher than in the serial program and the multithreaded program. The average running time for $n=25000$ in the serial program is 25.47457 seconds. In the multithreaded program, the running time using one thread is 29.51549 seconds. The slightly higher running time of the core-affine threaded program can be explained by the additional cost of using threads and the core assignment in the program.

It can also be observed that starting at $t=2$, 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 core-affine threaded program is poor compared to both the serial and multithreaded programs.

At $t=8$, the runtime of the column-wise threaded program stabilizes. Compared to the serial and multithreaded programs, the performance of the core-affine threaded program decreases

TABLE 4: Runtime of the Core-affine Threaded Program (Column-Wise Matrix Division)

n	t	Time Elapsed (seconds)			Average Runtime
		Run 1	Run 2	Run 3	
25000	1	32.548016	33.554707	34.933132	33.67862
25000	2	64.34594	63.666122	64.492371	64.16814
25000	4	63.275234	63.929565	65.023186	64.076
25000	8	56.082291	54.847855	55.728592	55.55291
25000	16	55.582771	54.97324	55.785248	55.44709
25000	32	51.190826	51.237225	50.744125	51.05739
25000	64	51.805489	50.196568	51.147701	51.04992

TABLE 5: Runtime of the Core-affine Threaded Program (Row-Wise Matrix Division)

n	t	Time Elapsed (seconds)			Average Runtime
		Run 1	Run 2	Run 3	
25000	1	30.566978	32.936749	32.685162	32.062963
25000	2	22.096247	21.724886	22.71212	22.177751
25000	4	21.962973	21.967112	22.078159	22.002748
25000	8	20.002848	17.729488	19.627289	19.119875
25000	16	19.087904	19.2775	17.27207	18.54582467
25000	32	16.630026	17.449362	18.960457	17.67994833
25000	64	16.086391	16.458574	16.045546	16.196837

by approximately 24.36% and 12.36% respectively based on the average runtime at $n = 25000$. This means that using core-affine threaded computation 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.

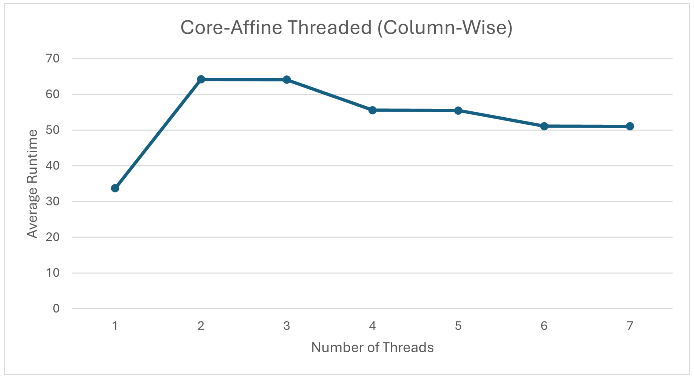

Fig. 1: Threads Vs. Average Runtime (Column-Wise)

B. 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 core-affine threaded 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 and their assignment to specific CPU cores.

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

Based on performance, using a row-wise matrix division in a core-affine threaded program results in an increase in performance by 36% for the serial program. However, compared to the multithreaded program, there is a decrease of approximately 26.84% in performance. The reason for the decrease in performance lies in the core assignment. In the multithreaded program, all eight logical CPU cores are utilized for the computation. On the other hand, only three logical CPU cores are used in the core-affine threaded program.

This means that using the core-affine threaded program may only be effective in increasing the performance of a program if the memory access is efficient. Moreover, efficient utilization of CPU cores is also a factor that increases the program's performance. Figure 2 shows the graph of the threads versus the average runtime of the program that uses the row-wise matrix division.

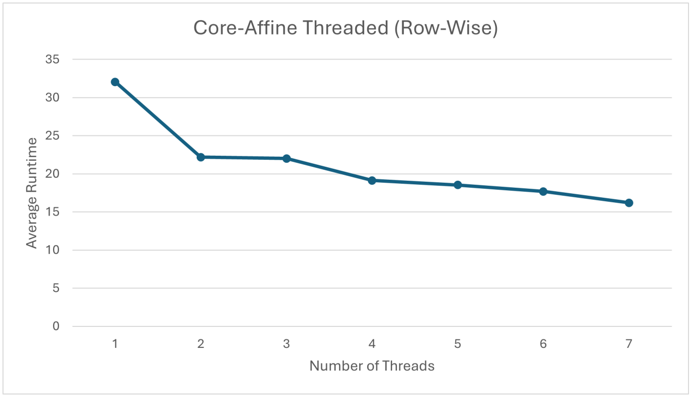

Fig. 2: Threads Vs. Average Runtime (Row-Wise)

C. Maximum Handled Data

In the serial program, the highest data that the program can handle is $n = 40000$. This is also true in the core-affine threaded program. Table 6 shows the performance of the core-affine threaded 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, multithreaded, and core-affine threaded program stopped working at $n = 50000$ it is safe to assume that the machine can't handle data starting at this point.

TABLE 6: Program's Maximum Data

Division	n	t	Time Elapsed (seconds)			Average
			Run 1	Run 2	Run 3	
Col-Wise	30000	8	107.10	106.65	107.27	107.01
Col-Wise	40000	8	162.03	130.46	132.29	141.59
Col-Wise	50000	8	-	-	-	-
Col-Wise	100000	8	-	-	-	-
Row-Wise	30000	8	23.25	25.46	25.61	24.77
Row-Wise	40000	8	44.53	44.56	45.03	44.71
Row-Wise	50000	8	-	-	-	-
Row-Wise	100000	8	-	-	-	-

D. 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 core-affine threaded 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 number of threads that the program can handle.

TABLE 7: Program's Maximum Thread

Division	n	t	Time Elapsed (seconds)			Average
			Run 1	Run 2	Run 3	
Col-Wise	25000	n	-	-	-	-
Col-Wise	25000	n/2	-	-	-	-
Col-Wise	25000	n/4	-	-	-	-
Col-Wise	25000	n/8	-	-	-	-
Row-Wise	25000	n	60.44	56.73	60.99	59.39
Row-Wise	25000	n/2	35.07	34.85	34.39	34.77
Row-Wise	25000	n/4	26.06	24.03	26.64	25.58
Row-Wise	25000	n/8	26.68	19.80	25.22	23.90

V. Conclusion

In this study, a core-affine threaded 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 both the serial and multithreaded programs was compared to the performance of the core-affine threaded 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 core-affine threaded program that uses the column-wise matrix division, the performance decreases at approximately 24.36% compared to the serial program and 12.36% compared to the multithreaded program. 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, with the use of row-wise matrix division, there is an observed 36% increase in performance compared to the serial program. However, when it is compared to the row-wise multithreaded program, the performance decreases by 26.84%. This varying result in terms of performance is linked to both the efficient memory access and core assignment.

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 even 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.