Computing the Pearson Correlation Coefficient of a Matrix and a Vector

John Rommel B. Octavo

***Abstract*—The study investigates the efficiency of serial programming in computing the Pearson correlation coefficient of a matrix and a vector. The experimentation focuses on the complexity and runtime performance of the developed algorithm. This program utilizes a serial computational technique with a theoretical complexity of $O(n^2)$. The experiments across varying data sizes observed that the runtime of the program is directly proportional to the data size and computed complexity. Algorithmic optimization and hardware upgrades are suggested to achieve a lower runtime of the developed serial program.**

***Index Terms*—Pearson correlation coefficient, serial computing, computational complexity, optimization**

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 serial programming. In serial programming, the efficiency of algorithms and computational tasks heavily influences overall performance [1].

Serial programming sequentially executes computer programs in a single processing unit [2]. As the data increases in serial computation, the time required for computation also increases [1]. This is particularly pertinent when dealing with large matrices and vectors, 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 [3]. 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. 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 Pearson correlation coefficient of a matrix and a vector is computed through serial programming. Empirical investigation through experiments was used to assess the efficiency of the computation for varying data sizes.

A. Statement of the Problem

Serial programming employs sequential techniques in computation [2]. Existing research indicates that the computational complexity of serial programming is influenced by the scale of the data being processed [1]. However, there remains a need to comprehensively investigate how employing serial 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 serial programming impacts certain factors that affect its computational complexity. It is beneficial in optimizing computational processes, crafting an efficient algorithm design, and it 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 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 serial 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 serial programming in the development of this algorithm.

The specific objectives of this research study are the following:

1) To develop a computer 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 complexity of the program, and;
3) To discuss ways to optimize the computer program considering the constraints of serial programming.

D. Scope and Limitations of the Study

This study aims to investigate the efficiency of serial 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 serial programming approach across varying data sizes. Algorithmic optimization strategies are also explored to enhance computational efficiency to improve runtime performance.

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 finding is 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

Serial computing is the sequential execution of instructions on a processor [2]. This computational technique can only handle one instruction at a time which means that other instructions need to wait first before it finally executes.

In the early days of computing, serial computing was used for computation. Hardware and software designs at that time had a simple architecture that could handle simple and straightforward calculations on a computer [4]. Early models of computers only had a single processor. These hardware limitations only allow one instruction to be executed at a time.

Moreover, early computers have shared memory which makes it difficult to execute more than one process [5]. Because of these limitations, early software development is hampered by these constraints. Early programs are written for serial computation to run on a computer with a single processor, to divide the problem into discrete series of instructions, and to execute instructions sequentially [6].

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 [7].

Serial computing is dependent on data transmission through the hardware [8]. 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 [9].

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 [10].

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 [11].

Despite the introduction of parallel computing, the serial computational technique is still used particularly in optimization and algorithm design. There are some limitations in the parallelization of programs that still require the use of serial computing.

In the study about the serial and parallel implementation of the Needleman-Wunsch algorithm, researchers encountered a problem in adapting their algorithm to parallel computing because of the data dependency [12]. This data dependency issues require the algorithm to run sequential execution of instructions. This leads the researchers to propose a new optimized algorithm that both handles parallel and serial computational techniques.

Another study was conducted to differentiate serial and parallel computing using MATLAB. In this study, given a 100×100 matrix, the researcher analyzes the execution time of both the Normal Algorithm and the Fox Algorithm. The researchers found that although serial computation has its limitations, the running time for serial processing is faster than in parallel computing. The study concludes that serial processing is still the best way to compute data until technology finally achieves true parallel processing [13].

In a 1994 study on serial and parallel computing, the researcher explains that parallel computing is not generally better than serial programming [14]. 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.

This study deals with solving the Pearson correlation coefficient of a matrix and a vector using serial computing. While this research does not implement parallel computation to compare the execution time, different techniques were implemented to optimize the serial program.

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 Python programming language. In the creation of a matrix and a vector, random package was used in the experiments. The Python programming language was used since it was the better choice for scientific experimentation, especially in data analytics.

B. Pseudocode of the Program

The computer program that computes the Pearson correlation coefficient of a matrix and a vector employs serial computing. It runs on a single processing and does not use threads in the program. Listing 1 shows the pseudocode for the function $pearson\_cor$. This function accepts an X matrix, y vector, m for the number of rows, and n for the number of columns. The pseudocode is the basis of the program development.

Listing 1: Program Pseudocode

```
func pearson_cor(X as matrix, y as vector
    , m as integer, n as integer) as
    vector
begin
        define v(n) as vector;
        for i:=1 to n do
        begin
                v(i) := 0
                for j:=1 to m do
                begin
                        v(i) := {see
                            equation 1};
                end;
        end;
        pearson_cor:=v;
end;
```

C. 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. These theoretical complexity calculations were compared to the actual processing time of the developed program.

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.

D. Calculation of the Program's Process Time

The Time package in Python is used to calculate the program's process time. The time before the function call and the time after calling the function were determined. The time elapsed of the program was calculated by the difference between these two.

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 each variation of the data size n in a $n \times n$ matrix and a $n \times 1$ vector, the time elapsed was calculated in three runs. The average time of these three runs was also noted and compared to the complexity in Table 1.

The n data size is increased to determine the limit by which the computer can handle the calculation. Optimization of the algorithm is done to improve the calculation and decrease the processing time.

IV. Results and Discussion

A. Serial Program

In the development of the $pearson\_cor$ function, serial computing is achieved by not using Python libraries that employ threads or similar parallel computing approaches. Listing 2 shows the Python code of the $pearson\_cor$ function that solves for the Pearson correlation coefficient of a matrix and a vector.

Listing 2: pearson_cor Function

```
def pearson_cor(X, y, m, n):
    v = [0] * n

    # Calculate sums
    sum_X = [sum(row) for row in X]
    sum_y = sum(y)
    sum_X_squared = [sum(item ** 2 for
        item in row) for row in X]
    sum_y_squared = sum(val ** 2 for val
        in y)

    x_mul_y = []

    for row in X:
        temp = []
        for i in range(n):
            temp.append(row[i]*y[i])
        x_mul_y.append(temp)

    sum_Xy = [sum(row) for row in x_mul_y
        ]

    # Calculate Pearson Correlation
        Coefficient vector
    for j in range(n):
        numerator = m * sum_Xy[j] - sum_X
            [j] * sum_y
        denominator = ((m * sum_X_squared
            [j] - sum_X[j] ** 2) * (m *
            sum_y_squared - sum_y ** 2))
            ** 0.5

        v[j] = numerator / denominator

    return v
```

This algorithm is patterned on the pseudocode in Listing 1 which has a computational complexity of $O(n^2)$. The complexity of the $pearson\_cor$ function depends on the operation that deals with traversing the values of the matrix. These operations involved nested looping that can take up much of the processing time.

In the worst-case scenario, the n x n matrix is huge and the computational complexity is $O(m \times n)$ or $O(n^2)$ where m and n are the number of rows and columns respectively. The $O(n^2)$ complexity can also be applied on the average-case scenario especially when the inputs in the matrix are randomly distributed. On the other hand, the best-case scenario would be that the matrix and other input parameters in the function are very small. The best-case complexity would be almost instant or $O(1)$.

To be able to call the $pearson\_cor$ function, the mxn matrix, $n \times 1$ vector, m rows, and n columns must be passed as parameters to the function. The main program achieves this by creating a matrix and a vector with random values. The size of the matrix and the vector depend on the program input of the user. Listing 3 shows how the main program creates a randomly generated matrix and vector to be passed as parameters to the $pearson\_cor$ function.

Listing 3: pearson_cor Function

```
def main():
    n = int(input("Enter the size of the
        square matrix and the length of
        the vector: "))

    X = [[random.randint(1, 100) for _ in
         range(n)] for _ in range(n)]
    y = [random.randint(1, 100) for _ in
        range(n)]


    time_before = time.time()
    v = pearson_cor(X, y, n, n)
    time_after = time.time()

    time_elapsed = time_after -
        time_before

    print("Time elapsed:", time_elapsed,
        "seconds")
```

The random library is used to generate the $m \times n$ matrix and $n \times 1$ vector. On the other hand, the time Python library is used to compute the processing time of the $pearson\_cor$

function in calculating the Pearson correlation coefficient of a matrix and a vector. Finally, the time elapsed is printed to check on the processing time of the $pearson\_cor$ function.

B. Data Size Experiment

In order to study the performance of the $pearson\_cor$ function in varied data sizes, program execution experiments were conducted. The processing time in each data size and the average in three runs were noted. This will be compared against the theoretical computation of the program's complexity.

Table 2 shows the result of the experiment. It is observed that the increasing data size n yields higher runtime and complexity. The runtime is computed in seconds and the complexity is computed based on the theoretical computation of the complexity which is $O(n^2)$.

TABLE II: Time Elapsed in Different Data Sizes

n	Average Runtime (Seconds)	Complexity (n^2)
100	0.00619	10000
200	0.013343333	40000
300	0.01901	90000
400	0.03123	160000
500	0.041346667	250000
600	0.06766	360000
700	0.08233	490000
800	0.107667333	640000
900	0.138113333	810000
1000	0.17419	1000000
2000	0.72601	4000000
4000	2.76367	16000000
8000	11.57646667	64000000
16000	46.74364	256000000
20000	103.7705867	400000000

The experiment result in Table 2 shows up to $n = 20,000$ data size with a complexity of $400,000,000$. The data size was further increased to $n = 100,000$ and $n = 10,000,001$ to test the limit of the computational power of the computer. However, the IDE flags a memory error that signifies that the computation can't be completed due to memory constraints. This means that the computation cannot proceed because there was no available memory space for the extensive calculation. In this case, to make the computation possible for larger data sizes greater than $n = 20000$ the computer must free up memory space or upgrade the memory capacity of the computer. Figure 1 and 2 shows the error message for $n = 100,000$ and $n = 10,000,001$ data sizes respectively.

```
PS C:\Users\User> python -u "c:\Users\User\Documents\My Projects\CMSC 180\lab01.py"
Enter the size of the square matrix and the length of the vector: 100000
Traceback (most recent call last):
  File "c:\Users\User\Documents\My Projects\CMSC 180\lab01.py", line 94, in <module>
    main()
  File "c:\Users\User\Documents\My Projects\CMSC 180\lab01.py", line 52, in main
    X = [[random.randint(1, 10) for _ in range(n)] for _ in range(n)]
          ^^^^^^^^^^^^^^^^^^^^
MemoryError
PS C:\Users\User>
```

Fig. 1: Error Message for $n = 100,000$ Data Size

C. Data Size, Running Time, and Complexity

The data presented in Table 2 are plotted on a superimposed line graph. The line graphs for data size n versus average runtime and data size n versus complexity are shown in Figure 3. In plotting the complexity, the values of the calculated n^2 were multiplied by $1.0 \times e^{-7}$ to compute the theoretical runtime of the program in seconds. By doing so, the form of the two line graphs for the average runtime and complexity can be compared in a single plot.

```
PS C:\Users\User\Documents\My Projects\CMSC 180> python -u "c:\Users\User\Documents\My Projects\CMSC 180\lab01.py"
Enter the size of the square matrix and the length of the vector: 10000001
Traceback (most recent call last):
  File "c:\Users\User\Documents\My Projects\CMSC 180\lab01.py", line 59, in <module>
    main()
  File "c:\Users\User\Documents\My Projects\CMSC 180\lab01.py", line 35, in main
    X = [[random.randint(1, 100) for _ in range(n)] for _ in range(n)]
          ^^^^^^^^^^^^^^^^^^^^^
MemoryError
PS C:\Users\User\Documents\My Projects\CMSC 180>
```

Fig. 2: Error Message for $n = 10,000,001$ Data Size

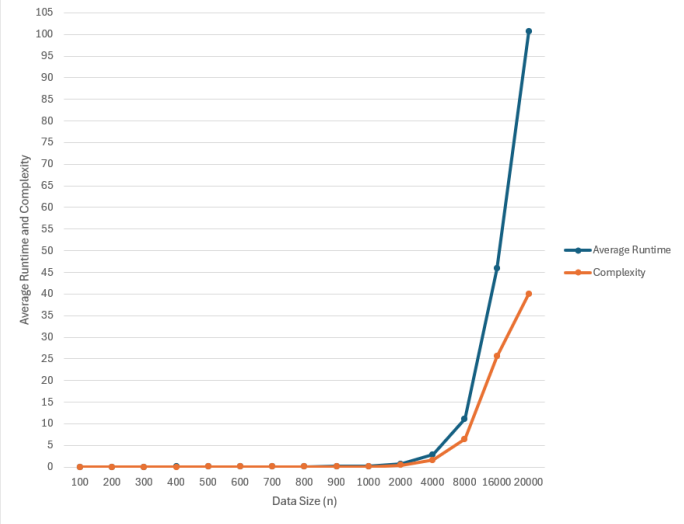

Fig. 3: Data Size versus Average Runtime and Complexity

Analyzing the plot of the data in the experiment, the line graph of the data size versus average runtime has the same form as the line graph of data size versus complexity. This means that the matrix data size, the program's computational time, and the complexity of the program have a direct proportionality relationship. The average time and complexity have little difference up to $n = 2000$. However, starting on $n = 4000$, the two line graphs were observed to have an upward trend and have bigger differences compared to when the data size n is 2000 and below.

Despite the growing difference starting at $n = 4000$, the two line graphs follow the same parabolic form. This means that the average runtime of the program given n data size follows the theoretical computation for the complexity of the program. Differences in the result of the average runtime that can be seen starting at $n = 4000$ are believed to be influenced by the computer limitations. These limitations affect the average runtime of the program. However, the essential result of the experiment lies in the discovery that both line graphs for the average runtime and complexity follow the same parabolic form. This result would prove that the program has indeed an $O(n^2)$ computational complexity.

D. Optimization of the Serial Program

The average runtime of the serial program is affected by its algorithm design and the constraints or limitations of the computer. In the algorithm design, the average runtime can be lowered by optimizing the algorithm. However, since the average-case complexity of the serial program is $O(n^2)$ and the program does not use threads or similar parallel computing techniques, optimizing the algorithm has little effect in lowering the runtime of the program.

Another solution to achieve a lower runtime of the program is by upgrading the hardware components of the computer. Upgrading the computational speed of the computer and allocating more memory can significantly lower the average runtime of the program. Specifically, dedicating a more powerful CPU or GPU to the serial computation of the Pearson correlation coefficient of a matrix and a vector can improve the program's processing speed.

V. Conclusion

In this study, a serial program was developed to solve the Pearson correlation coefficient of a matrix and a vector. The algorithm's theoretical complexity is computed as $O(n^2)$ and was proven by plotting the average runtime of the serial program against the data size n. It was also observed in the experiments that the data size, average runtime, and complexity of the serial program have a direct proportionality relationship. In the discussion about the optimization to achieve lower runtime, the study suggests optimizing the algorithm of the program and upgrading the computational speed of the computer.

References

[1] P. Anurag, A. V. Chauhan, A. Yadav, and M. K. Yadav, "Performance Analysis of Serial Computing Vs. Parallel Computing in MPI Environment," *International Journal of Advanced Research in Science, Communication and Technology*, vol. 2, no. 7, 2022, pp. 1–1. Available: https://doi.org/10.48175/568.

[2] Lithmee, "What is the difference between serial and parallel processing in computer architecture," Pediaa.Com, 2019. Available: https://pediaa.com/what-is-the-difference-between-serial-and-parallel-processing-in-computer-

[3] 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.

[4] R. E. Smith, "A Historical Overview of Computer Architecture," *Annals of the History of Computing*, vol. 10, pp. 277-303, 1989.

[5] D. R. Cheriton, "Problem-oriented shared memory revisited," in *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring (EW 5)*, 1992, pp. 1-4. Available: https://doi.org/10.1145/506378.506392.

[6] P. Durgaprasad, "Parallel Computing: High Performance," *International Journal of Emerging Technology and Advanced Engineering*, vol. 1, no. 2, 2011.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] Y. S. Lee, Y. S. Kim, and R. L. Uy, "Serial and parallel implementation of Needleman-Wunsch algorithm," *International Journal of Advances in Intelligent Informatics*, vol. 6, pp. 97-108, 2020.

[13] A. B. Rathod, R. Khadse, and M. F. Bagwan, "Serial Computing vs. Parallel Computing: A Comparative Study using MATLAB," *International Journal of Computer Science and Mobile Computing*, vol. 3, no. 5, pp. 815-820, 2014.

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

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.