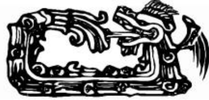


www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 16(2), April 2025, 191-199. ISSN: 2007-1558.
https://doi.org/10.61467/2007.1558.2025.v16i2.1081

Computational Efficiency in Mathematical Algorithms: A Study of Linear vs. Parallel Programming in the Context of Image Processing

Eric León Olivares[1], Luis Carlos Márquez Strociak[3], Mayra Lorena González Mosqueda[2], Karla Martínez Tapia[2], Salvador Martínez Pagola[1], Eric Simancas-Acevedo[4]

[1] Tecnológico Nacional de México/Instituto Tecnológico de Pachuca – Departamento de Sistemas y Computación

[2] Tecnológico Nacional de México/Instituto Tecnológico de Pachuca – Estudiante del Programa de Ingeniería en Sistemas Computacionales

[3] Tecnológico Nacional de México/Instituto Tecnológico de Pachuca – Departamento de Ciencias Económico Administrativas

[4] Universiadad Politécnica de Pachuca/Dirección de Investigación, Innovación y Posgrado – P.E de la Maestría en Tecnologías de la Información y Comunicaciones

E-mails: eric.lo@pachuca.tecnm.mx, l23200286@pachuca.tecnm.mx, mayra.gm@pachuca.tecnm.mx, karla.mt@pachuca.tecnm.mx, salvador.mp@pachuca.tecnm.mx, ericsimancas@upp.edu.mx

Abstract. The implementation of mathematical algorithms plays a fundamental role in computational efficiency. Sequential programming, which processes instructions in a linear manner, often struggles with large data volumes due to its inherent limitations. In contrast, parallel programming distributes tasks across multiple cores, significantly reducing processing times and improving overall performance. This paper presents a comparative analysis of both approaches and their relevance in Systems Engineering, where computational optimization is critical. To this end, we implement and evaluate the Sobel algorithm—commonly used for edge detection in images—in both sequential and parallel modes. The implementation is carried out in Python, leveraging the NumPy, OpenCV, and Multiprocessing libraries. This study analyzes the conditions under which parallelization enhances performance and identifies scenarios where process overhead may negate its benefits, thus establishing fundamental criteria for applying these techniques to solve mathematical problems in engineering. The source code is available on GitHub at: [GitHub Repository].

Keywords: Parallel programming, Sequential programming, Sobel filter, Computational efficiency, Image processing.

Article Info
Received Jan 26, 2025
Accepted Mar 11, 2025

1 Introduction

The execution methodology of mathematical algorithms directly impacts computational efficiency, particularly in operations involving large-scale calculations. Historically, sequential programming—also known as linear programming—has been the predominant approach, executing operations step by step in a linear fashion. However, this paradigm is not always optimal, especially when high performance is required for processing vast amounts of data and complex operations.

This paper conducts a comparative analysis between sequential and parallel programming, evaluating their respective advantages and limitations in mathematical problems related to image processing. The study explores the applicability of parallelism, determining its suitability based on the specific context and nature of the problem.

© Editorial Académica Dragón Azteca S. de R.L. de C.V. (EDITADA.ORG), All rights reserved.

With advancements in hardware technology and increasing demand for computational optimization, parallel programming has emerged as a fundamental solution. Its ability to distribute tasks among multiple processing cores significantly reduces execution times, an aspect especially relevant in applications such as image processing using mathematical algorithms. In this context, intensive operations such as matrix manipulation and other complex calculations inherent to image processing can greatly benefit from concurrency techniques.

For the practical implementation, Python was selected due to its extensive ecosystem of scientific computing and image processing libraries. Specifically, the following tools are utilized:

- NumPy (numpy): For efficient creation, manipulation, and mathematical operations on arrays and matrices.
- OpenCV (cv2): For image loading and processing. Notably, although OpenCV includes its own implementation of the Sobel filter, a custom version of the algorithm was developed in this work. This choice was made for didactic and optimization reasons, as it allows illustrating the parallelization process from its basics and provides more precise control over the internal logic of the operation.
- Time (time): For measuring execution times with high precision.
- Multiprocessing (multiprocessing): To leverage parallel execution by distributing tasks across multiple processing cores.

This set of tools facilitates the implementation and evaluation of the algorithms in both their sequential and parallel versions, enabling a detailed analysis of their performance and efficiency.

The case study presented in this paper focuses on the implementation and analysis of the Sobel algorithm for edge detection in images. The primary objective is to evaluate the impact of parallel programming compared to its sequential execution, measuring processing times and determining whether task division into multiple processes leads to a significant performance improvement.

To achieve this, an experiment was designed in which both versions of the algorithm—parallel and sequential—are executed under controlled conditions, using the same input image and hardware for execution. In the parallel version, different configurations were tested, varying the number of processing segments to identify the point at which parallelization reaches its maximum efficiency and when the benefits start to decline due to the overhead associated with process management.

Through this comparative analysis, the study aims to validate the feasibility of using parallel programming in these types of mathematical problems, establishing clear criteria to determine when it is advantageous to implement these techniques and in which cases sequential programming remains a viable option. This approach provides a solid foundation for decision-making in algorithm optimization for engineering and image processing applications.

The remainder of this paper is structured as follows:

1. State of the Art – A review of the evolution of parallel programming, its role in high-performance computing, and the available tools for implementation.
2. Methodology – Details on the experimental setup, algorithm implementation, execution environment, and performance measurement techniques.
3. Results – A comparative analysis of execution times, illustrated through tables and graphs.
4. Discussion – Interpretation of findings, identification of optimal parallel configurations, and an assessment of computational overhead.
5. Conclusions – Key insights into the benefits and limitations of parallel programming, along with recommendations for future research.

The complete source code, along with execution and testing instructions, is available at: [GitHub Repository].

2 State of the Art

Parallel programming has undergone significant advancements in recent decades, driven by the increase in computing power and the growing demand for computational optimization in various fields of science and engineering.

2.1 Evolution of Parallel Programming

Traditionally, algorithm execution was carried out sequentially, processing instructions one after another on a single CPU core. However, with the emergence of multicore and manycore architectures, it has become possible to divide and distribute workloads among multiple processing units, enabling simultaneous execution and significantly improving efficiency. This development has been a milestone in solving large-scale and complex problems, optimizing both execution time and resource utilization.

2.2 Impact of High-Performance Computing

The increase in transistor density, as described in Moore's Law [Moore, 2021], allowed circuit miniaturization and higher processor speeds. However, physical limitations in clock frequency scaling led to the adoption of multi-core processors, giving rise to multicore and later manycore systems, where hundreds or even thousands of cores can operate in parallel.

In this context, High-Performance Computing (HPC) emerged as a key area for developing parallel systems, enabling the simulation and modeling of complex phenomena in science, engineering, and big data analysis. HPC relies on supercomputers, computing clusters, and heterogeneous platforms (CPU + GPU) to process large volumes of information in reduced time. This type of computing has been essential in fields such as computational physics, big data analytics, and artificial intelligence.

2.3 Tools for Implementing Parallelism

To facilitate parallelism implementation, several programming tools and standards have been developed to efficiently distribute workloads:

- OpenMP (Open Multi-Processing): Provides a shared-memory model via directives (pragmas) that parallelize code execution in C, C++, and Fortran, optimizing the use of multiple threads in CPUs [Pinho, 2023].
- MPI (Message Passing Interface): Designed for distributed-memory environments where each computing node has its own memory space and communicates via message passing, enabling task execution on supercomputers and clusters [Gropp et al., 2021].
- CUDA (Compute Unified Device Architecture): Developed by NVIDIA, allows massively parallel execution on GPUs by extending C, C++, and Fortran, enabling computations across hundreds or thousands of GPU cores [Nickolls et al., 2021].
- OpenCL (Open Computing Language): Provides an open standard for parallel programming on CPUs, GPUs, and other heterogeneous architectures, allowing multiple types of hardware to be utilized within a single system [Stone et al., 2023].

Python in Parallel Computing

As computing has transcended academia and high-performance computing (HPC), accessible tools for implementing parallelism in industrial and scientific applications have become essential. In this context, Python has become one of the most prominent options in scientific computing, not only due to its intuitive syntax and ease of learning but also because of its extensive ecosystem of optimized libraries. Its popularity has grown exponentially, driven by several key factors:

- Readability and ease of learning: Python allows for clear and simple syntax, making it easier to implement parallel models without the complexity of low-level languages.
- Robust ecosystem of libraries: Various libraries such as NumPy, SciPy, and Pandas are designed to efficiently handle data structures, facilitating numerical computations in arrays and matrices [Oliphant, 2020].

- Native parallelization capabilities: Multiprocessing and Threading modules enable parallel execution via processes and threads, respectively, while tools like Dask and Ray extend these capabilities to distributed computing environments [Zaharia et al., 2016].
- Compatibility with modern hardware: Python integrates with GPUs and heterogeneous architecture through libraries like CUDA and OpenCL, maximizing efficiency in high-performance computations.
- Scalability and flexibility: Python enables experiments on personal computers as well as supercomputers, making it viable for research and development across various disciplines.

Applications in Image Processing

In the field of image processing, the OpenCV library has been widely used to optimize operations using SIMD (Single Instruction, Multiple Data) techniques and multithreading [OpenCV Development Team, 2023]. Its ease of use and compatibility with Python have made it a key tool in this domain, enabling efficient execution of computer vision tasks with high computational efficiency.

3 Methodology

The proposed methodology aims to compare the performance of sequential and parallel programming when applying the Sobel filter in image processing. The following steps and evaluation criteria are detailed:

1. **Selection of the Test Image**
    - A grayscale image is selected (or converted into this format) that is large enough to highlight performance differences.
    - Optionally, a blur filter is applied to reduce noise, ensuring consistent initial processing conditions.
2. **Implementation of the Algorithms**
    - Sequential version: The Sobel algorithm is executed linearly over the image, processing each pixel one after another without task division.
    - Parallel version: The image is divided into segments, and the workload is distributed among multiple processes. For this experiment, multiple executions will be performed while varying the number of segments (2, 4, 6, 8, 12) to evaluate performance changes as parallelism increases.
3. **Execution Environment Configuration**
    - **Language:** Python 3, using libraries such as NumPy, OpenCV, and multiprocessing.
    - **Hardware:** The number of available CPU cores, RAM memory, and operating system are specified.
    - **Library Versions:** The versions of NumPy, OpenCV, and Python are documented to ensure reproducibility.
4. **Execution Time Measurement**
    - Before applying each version **(sequential or parallel)**, the start time is recorded using the **time()** function.

- After processing is completed, the end time is recorded, and the total execution time is calculated.
- To reduce variability, each experiment will be executed multiple times **(e.g., 3 or 5 iterations)**, and the average execution time will be taken.

5. **Comparison and Analysis of Results**
   - Comparative tables will be created to display the average execution time for each version. The columns will include:
     - **Number of segments** (1 for sequential, 2, 4, 6, 8, 12 for parallel execution).
     - **Average execution time.**
     - **Standard deviation or range of variation** (if deemed necessary).
   - To better visualize differences, bar charts will be created comparing sequential execution with different parallel configurations.
   - The analysis will focus on identifying:
     - An inflection point where increasing the number of segments **no longer provides significant improvements** (or may even cause overhead due to process management).
     - The reduction in execution time achieved by the parallel version compared to the sequential version.

4 Discussion of Results

The reasons why a specific number of segments optimizes or does not optimize performance will be interpreted (e.g., process overhead, image size, CPU capacity, among others).

The scalability of the parallel approach and its applicability in high-demand environments will be considered

This approach makes it possible to quantify the performance improvement provided by parallel programming compared to sequential programming, establishing a correlation between the degree of parallelism (number of segments) and the effective reduction in computation times. This analysis will not only allow for a precise interpretation of the results but will also lay the foundation for informed decision-making in Systems Engineering projects, where optimization and efficiency are key priorities.

4.1 Considerations of Benchmarking Technology

To ensure a fair comparison between the sequential and parallel approaches, multiple tests were conducted for each configuration, and average values were calculated to reduce the influence of external factors on the results.

- In the sequential execution, each image was processed 10 times, and the times obtained were averaged to minimize fluctuations caused by the operating system load or variations in resource availability.
- In parallel execution, 5 tests were performed for each number of segments (2, 4, 6, 8, 12) on both images, ensuring that the results reflect a real trend in the impact of parallelism rather than relying on a single isolated measurement.

Additionally, to avoid unnecessary deviations, the tests were carried out with the least possible number of background processes. Most programs were disabled, and the measurements were performed directly from the Windows Terminal, without interference from other running applications. This approach ensured more **precise and representative values**, reflecting only the algorithm's performance under optimal execution conditions.

4.2 Preliminary Analysis of Results

To evaluate the impact of parallelism, execution time measurements were performed while varying the number of segments in the parallel version: 2, 4, 6, 8, and 12 segments. The sequential version, which does not divide the image, is considered the baseline case with a single segment. It is expected that as the number of segments increases, execution time will decrease up to a certain optimal point, beyond which process management overhead may prevent further improvements. For the tests, images 1 and 2 were used, as shown below:

Image 1

Image 2

5 Results

In Table 1, the measured execution times for each configuration are presented. These results will help identify trends and determine the extent to which parallelization is effective in optimizing the Sobel gradient computation.

Table 1. Results of the Sequential Test

Image	Average Time's	Standard Deviation's
Image 1	2.424875	0.014845
Image 2	1.086225	0.01059

Tables 2 and 3 present the results of executing the Sobel algorithm using more than one segment, which involves applying parallelization techniques for Image 1 and Image 2, respectively.

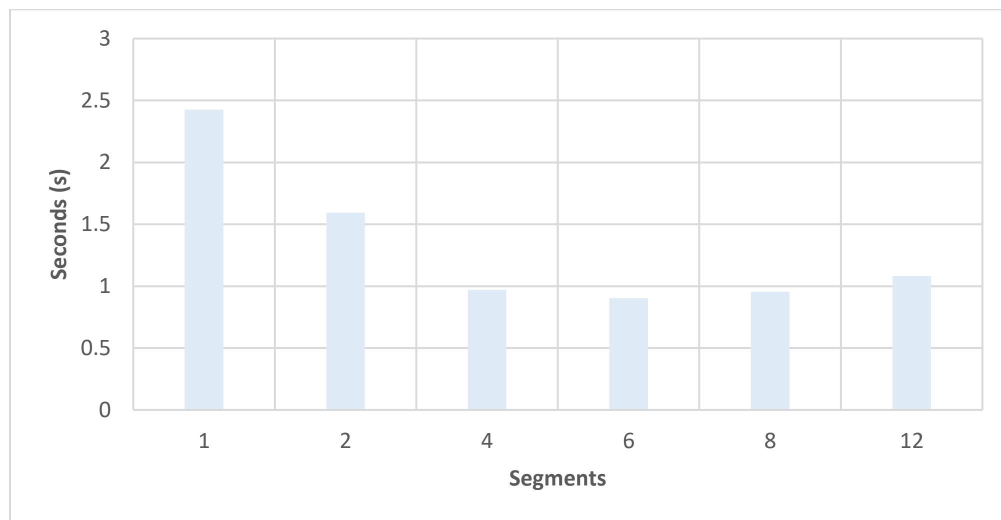
Table 2. Results of the Sequential Test

Number of Segments	Average Time's	Standard Deviation's
2	1.592117	0.010056
4	0.97128	0.008603
6	0.90326	0.024059
8	0.95552	0.012606
12	1.08192	0.017005

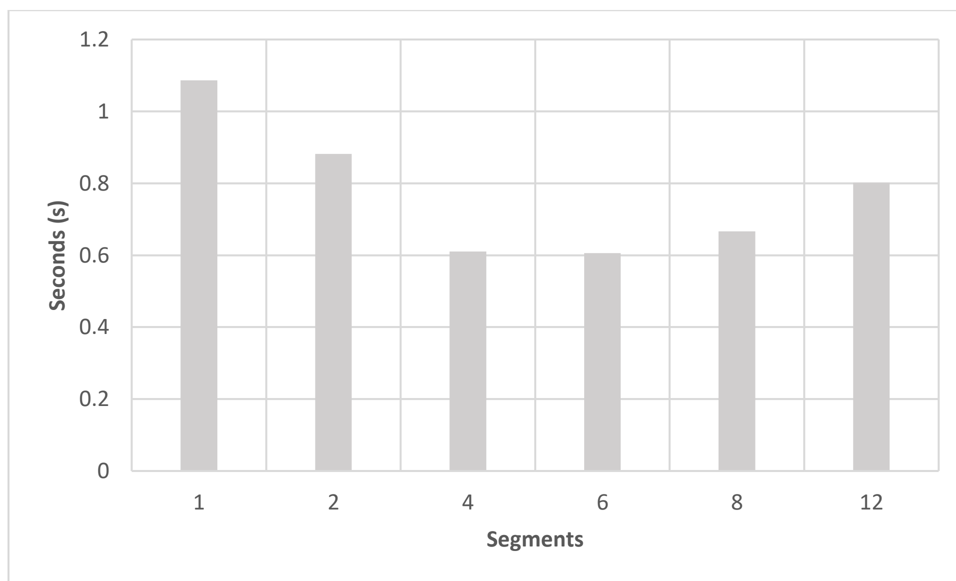
Table 3. Results for image 2

Number of Segments	Average Time's	Standard Deviation's
2	0.88188	0.009833
4	0.610543	0.003744
6	0.6059	0.013864
8	0.66664	0.012268
12	0.80176	0.012116

Below, Graph 1 and Graph 2 visually represent the benchmarking results using bar charts. These graphical representations facilitate the interpretation and comparative analysis of the collected data, highlighting the trends and impact of parallel execution across different segment configurations.



Graph 1. Results for Image 1



Graph 2. Results for Image 2

6 Discussion

The obtained results clearly reflect the impact of parallelism on the performance of image processing using the Sobel filter. It is evident that, in general, parallel execution significantly reduces processing times compared to sequential execution. However, certain limits on performance improvement are also observed, especially when the number of segments exceeds an optimal threshold.

Comparison between sequential and parallel execution

The average times obtained for sequential execution show that processing Image 1 took an average of 2.4248 seconds, while Image 2 took 1.0862 seconds. These differences can be attributed to variations in resolution and image content, which affect the number of pixels to process.

In parallel execution, using 2 to 12 segments allowed for a reduction in processing times for both images. However, it was found that performance improves up to a certain point, after which the benefits of parallelism begin to diminish or even degrade due to the overhead in process management.

Performance analysis based on the number of segments

For Image 1, the best time reduction was achieved with 6 segments, where processing took 0.9032 seconds, resulting in a 62.7% improvement compared to sequential execution. However, when increasing to 8 and 12 segments, the time increased again, indicating that system overhead and process synchronization begin to affect parallelism efficiency.

For Image 2, the shortest duration was achieved with 6 segments, with an average time of 0.6059 seconds, representing a 44.2% reduction compared to sequential execution. Similar to what was observed in Image 1, increasing to 8 and 12 segments caused the time to increase progressively, suggesting that the gain from parallelism is limited by the cost of coordination between processes.

Impact of the standard deviation

The standard deviations obtained in both cases are low, indicating that execution times are consistent and stable in each configuration. However, for Image 1 with 6 segments, a higher deviation (0.0240 s) was observed, suggesting some variability in execution times due to factors such as shared memory handling and core allocation by the operating system.

Interpretation of the results

The obtained data supports the hypothesis that parallelism improves performance in image processing with the Sobel filter but also shows that there is an inflection point where the benefits of parallelism decrease. This is because, although dividing the image into segments reduces computation time, the management and synchronization of multiple processes introduce overhead that, after a certain number of segments, negates the improvements achieved.

These findings are consistent with the principles of scalability in parallel programming, where increasing the number of processes does not guarantee indefinite performance improvement. In this case, the results suggest that, in a 6-core and 12-thread environment, dividing the load into 6 segments is an optimal option for improving efficiency without incurring synchronization overhead.

7 Conclusions

This study on computational efficiency in mathematical algorithms, focused on the implementation of the Sobel filter, has allowed for a comparison between the performance of sequential and parallel programming in the context of image processing. The results obtained demonstrate that parallelism is a powerful tool for optimizing intensive calculations, significantly reducing execution times compared to the sequential approach. However, it has also been shown that an increase in the number of processes does not always guarantee continuous improvement, due to the overhead associated with task management and synchronization between processes.

It was determined that using 6 segments on a system with 6 cores and 12 threads provided the best balance between performance and efficiency, achieving shorter execution times without incurring additional costs from process management. Beyond this point, a degradation in performance was observed, suggesting that an optimal configuration must consider both the available hardware resources and the specific nature of the task being processed.

From a broader perspective, this study underscores the importance of parallelism in the field of Systems Engineering, where efficient data processing is a critical factor in applications ranging from computer vision to numerical simulation and artificial intelligence. The adoption of parallel methodologies not only improves scalability and hardware utilization but also becomes a necessity in environments where computing speed is crucial for real-time decision-making.

For future research, it is recommended to explore more complex problems within image processing, such as the implementation of more sophisticated filters or the application of convolutional neural networks (CNNs) in edge detection and image segmentation tasks. Additionally, it would be of interest to expand this analysis to heterogeneous architectures, combining CPU and GPU, to assess the impact of parallelism on platforms with massive computing capabilities. Finally, extending these methodologies to other areas of engineering, such as physical model simulation or optimization of machine learning algorithms, could open new opportunities to improve computational efficiency in real-world applications.

This work makes it clear that, while parallel programming represents the future of high-performance computing, its implementation must be carefully planned to avoid inefficiencies. The design of optimized algorithms tailored to the available resources will be a key element in the evolution of modern computing and in solving increasingly complex problems in the field of Systems Engineering.

8 References

1. Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.
2. Dagum, L., & Menon, R. (1998). OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, *5*(1), 46–55.
3. Gropp, W., Lusk, E., & Skjellum, A. [2021]. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press.
4. OpenCV Development Team. [2023]. *OpenCV: Open Source Computer Vision Library*. Retrieved from https://opencv.org
5. Gropp, W., Lusk, E., & Skjellum, A. (1999). *Using MPI: Portable Programming with the Message Passing Interface*. MIT Press.
6. Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, *38*(8).
7. Moore, G. E. [2021]. *Cramming more components onto integrated circuits*. *Electronics*, *38*[8], 114–117.
8. Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. *ACM Queue*, *6*(2), 40–53.
9. Walt, S. van der, Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, *13*, 22–30.
10. Stone, J. E., Gohara, D., & Shi, G. [2023]. *OpenCL: A parallel programming standard for heterogeneous computing systems*. *IEEE Design & Test of Computers*, *12*[3], 6–18.
11. Zaharia, M., Xin, R., Wendell, P., Das, T., Armbrust, M., & Stoica, I. [2016]. *Apache Spark: A Unified Engine for Big Data Processing*. *Communications of the ACM*, *59*[11], 56–65.
12. Strociak, L. (2025). Github Repository. *Sobel-Algorithm: Sobel algorithm to detect forms from images*. https://github.com/lcmarquez2005/Sobel-Algorithm