# 

# **Parallel and Distributed Image Processing using OpenMP and MPI**

# **Course: Parallel and Distributed Computing**

# **Submitted To: Sir Syed Faisal Ali**

# **Section: BCS – 5J**

# **Group Members:**

# **21K – 4564 Muhammad Talha Shaikh**

# **21K – 3293 Muhammad Hamza**

# **21K – 3346 Areeb Ul Haq**

# 

# **1. Introduction**

Image processing is a computationally intensive task vital for various applications, from medical imaging to digital media processing. As the size and complexity of image datasets grow, there is an increasing demand for efficient algorithms capable of handling such workloads. Parallelization, a key technique in high-performance computing, plays a crucial role in accelerating image processing tasks.

This project focuses on implementing a parallel image blurring algorithm using OpenMP, a widely adopted parallel programming framework. The blurring operation, a fundamental image processing task, is performed through convolution, where each pixel is updated based on its neighboring pixels. The parallelization using OpenMP aims to distribute this workload across multiple threads, harnessing the computational power of modern multicore processors.

Objective:

* Implement a serial image blurring (gaussian and box) and inversion algorithms in C.
* Parallelize the algorithms using OpenMP to exploit parallel architectures.
* Evaluate the performance gains achieved through parallelization.

Significance:

* Efficient image processing is essential for real-time applications and large-scale datasets.
* OpenMP provides a straightforward approach to parallel programming, making it accessible for developers aiming to enhance performance.

Report Overview:

* The report begins with an algorithmic overview of the serial image blurring code.
* It proceeds to detail the parallelization strategy employed with OpenMP.
* Performance metrics, including serial and parallel execution times, will be analyzed and compared.
* Graphical representations will visualize the speedup achieved through parallelization.
* The conclusion summarizes findings and outlines potential areas for future optimization.

Through this report, we aim to demonstrate the impact of parallelization on image processing performance and contribute insights into optimizing algorithms for parallel architectures.

# **2. Problem Statement**

The problem at hand revolves around the ever-growing demand for efficient image processing algorithms capable of handling the intricacies posed by large-scale image datasets. Image processing, a critical component in various domains, ranging from medical diagnostics to multimedia applications, often involves computationally intensive operations.

Challenges:

* Computational Intensity: Image processing tasks, such as blurring, involve complex calculations for each pixel, making them computationally intensive.
* Scale of Image Datasets: As datasets expand, serial algorithms may struggle to deliver results within acceptable timeframes.
* Real-time Requirements: Applications in medical imaging and multimedia demand real-time processing, necessitating optimized algorithms.
* Resource Utilization: Modern hardware, with multicore processors, provides an opportunity to enhance performance through parallelization.

Significance:

* Medical Imaging: Efficient image processing is crucial for tasks like feature extraction and anomaly detection in medical images.
* Multimedia Applications: Real-time video processing and editing require algorithms that can handle the scale and complexity of multimedia datasets.
* Scientific Research: Image analysis in scientific research often involves large datasets, demanding algorithms capable of expedited computations.

# **3. Algorithms**

# **Algorithm Description for Gaussian Blur**

The serial image processing algorithm implemented in the provided code focuses on blurring an image using a convolution operation. The algorithm follows these key steps:

* Image Loading:
  + The program starts by loading a bitmap (BMP) image file. The image structure (img) is used to store metadata, including width, height, and color information.
* RGB Separation:
  + The image data is then separated into three color channels: red, green, and blue. This is done to independently process each color component during the blurring operation.
* Convolution Operation:
  + The blurring operation is performed through a convolution operation. For each pixel in the image, a weighted average is computed based on the surrounding pixels within a specified radius.
  + The algorithm uses nested loops to traverse each pixel in the image, and for each pixel, it calculates a weighted sum of the neighboring pixels. The weights are determined by a Gaussian function, providing a smoothing effect.
  + The algorithm takes into account boundary conditions, ensuring that pixels near the edges of the image are properly handled.
* Parallelization Opportunities:
  + The algorithm is initially designed for serial execution, and the nested loops present parallelization opportunities. However, the parallel version is introduced later in the code using OpenMP directives.
* Performance Measurement:
  + The algorithm includes functionality to measure the execution time for different numbers of threads. The performance is evaluated based on the total execution time and is compared across various thread counts.

Rationale:

* Convolution for Blurring: Convolution is a common technique for blurring images. It allows each pixel to be influenced by its neighbors, creating a smoothing effect.
* Gaussian Weights: The use of Gaussian weights ensures a balanced and smooth blurring effect, reducing the impact of distant pixels on the current pixel.
* Separate Channels: Processing each color channel independently facilitates parallelization and provides flexibility in handling different color components

**Algorithm Description for Box Blur**

The serial image processing algorithm implemented for box blur in the provided code focuses on blurring an image using a simpler convolution operation known as a box blur. The algorithm follows these key steps:

Image Loading:

* + The program begins by loading a JPEG image file. The image structure (img) is utilized to store metadata, including width, height, and color information.

RGB Separation:

* + Similar to the Gaussian Blur algorithm, the image data is separated into three color channels: red, green, and blue. This allows for independent processing of each color component during the blurring operation.

Convolution Operation (Box Blur)

* + The box blur operation involves traversing each pixel in the image and calculating a simple average of the neighboring pixels within a specified radius (box size).
  + Instead of using a weighted sum with a Gaussian function, the algorithm computes a straightforward average of pixel values in the specified neighborhood.
  + The nested loops iterate through each pixel, considering its neighboring pixels within the box radius.

Parallelization Opportunities:

* Like the Gaussian Blur algorithm, the box blur algorithm initially assumes a serial execution. However, parallelization opportunities can be explored using parallel processing techniques, such as OpenMP directives, to enhance performance.

Performance Measurement:

* The algorithm includes functionality to measure the execution time for different numbers of threads, similar to the Gaussian Blur algorithm. Performance is evaluated based on the total execution time and is compared across various thread counts.

Rationale:

* Simplicity of Box Blur: Box blur provides a straightforward blurring effect by averaging pixel values within a specified neighborhood without the complexity of weighted sums.
* Parallelization: The algorithm's design allows for potential parallelization to improve execution time on multi-core processors.
* JPEG Image Loading: The algorithm supports loading JPEG images, offering flexibility in image format handling.

**Algorithm Description for Image Inversion**

The serial image processing algorithm implemented for image inversion in the provided code focuses on inverting the colors of an image. The algorithm follows these key steps:

Image Loading:

* Similar to the blur algorithms, the program starts by loading a JPEG image file. The image structure (img) is used to store metadata, including width, height, and color information.

Color Inversion:

* The algorithm traverses each pixel in the image and inverts the color values. For RGB images, this involves subtracting the current pixel's red, green, and blue values from the maximum intensity value (255).

Parallelization Opportunities:

* The algorithm is initially designed for serial execution, but opportunities for parallelization may exist, especially when processing color channels independently.

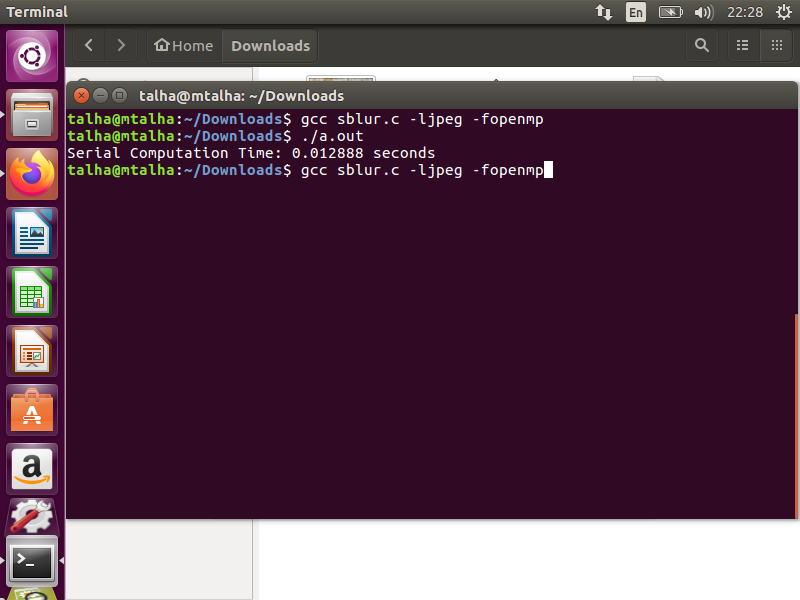
Performance Measurement:

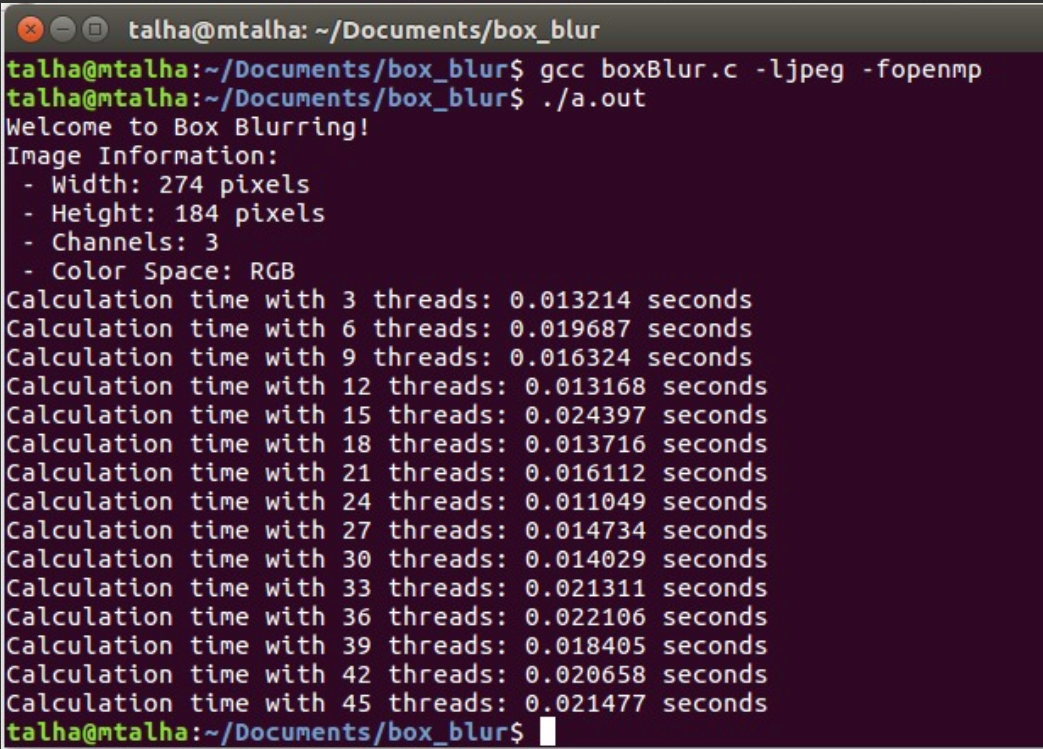
* Similar to the blur algorithms, the algorithm includes functionality to measure the execution time for different numbers of threads. Performance is evaluated based on the total execution time and is compared across various thread counts.

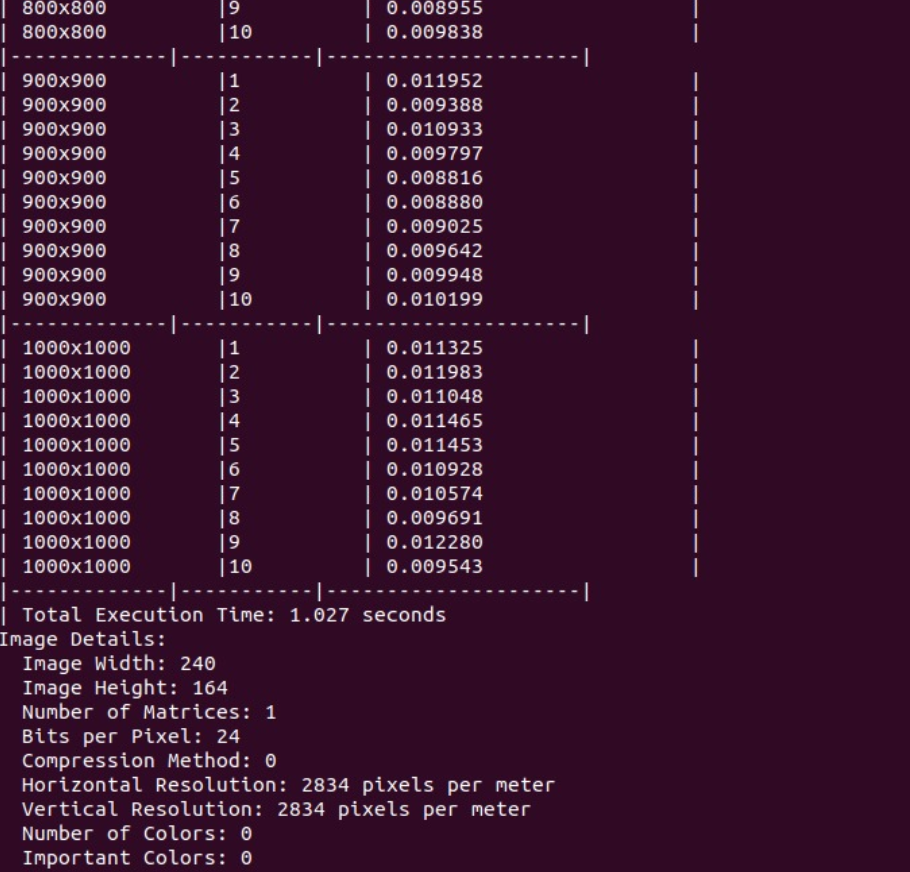
Rationale:

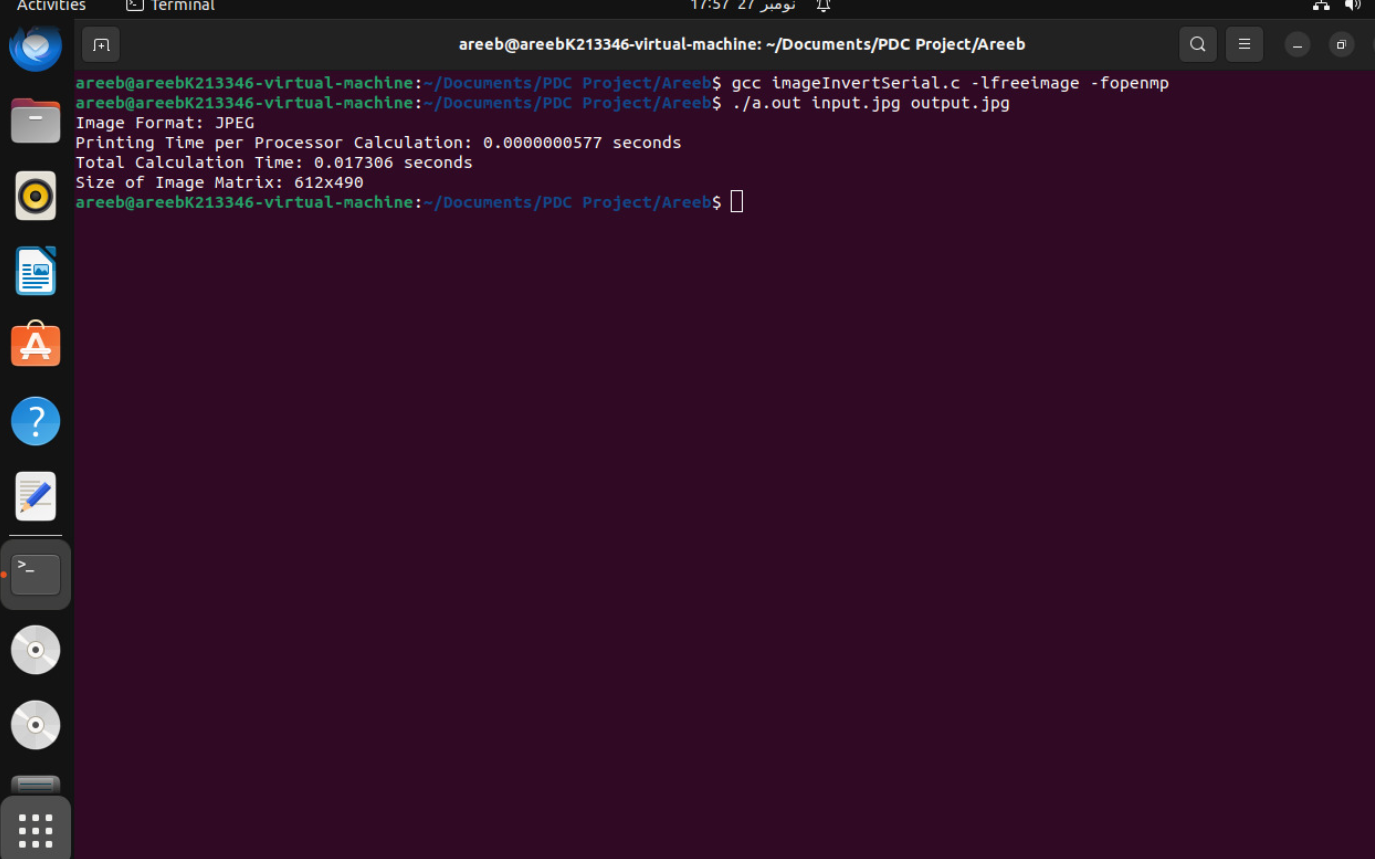
* Color Inversion Technique: Inverting colors is a common image processing operation that produces a photographic negative effect.
* Parallelization Potential: Depending on the implementation, parallelization may be explored to enhance performance, particularly if color channels are processed independently.
* JPEG Image Loading: The algorithm supports loading JPEG images, providing

**4. Serial and Parallel Execution Time:**









# 5. **Parallelization with OpenMP**

The provided code is parallelized using OpenMP to leverage multi-threading and enhance the performance of the image processing algorithm. The key parallelization strategy involves distributing the workload among multiple threads, allowing concurrent execution of the nested loops responsible for image convolution.

## Parallelization Strategy:

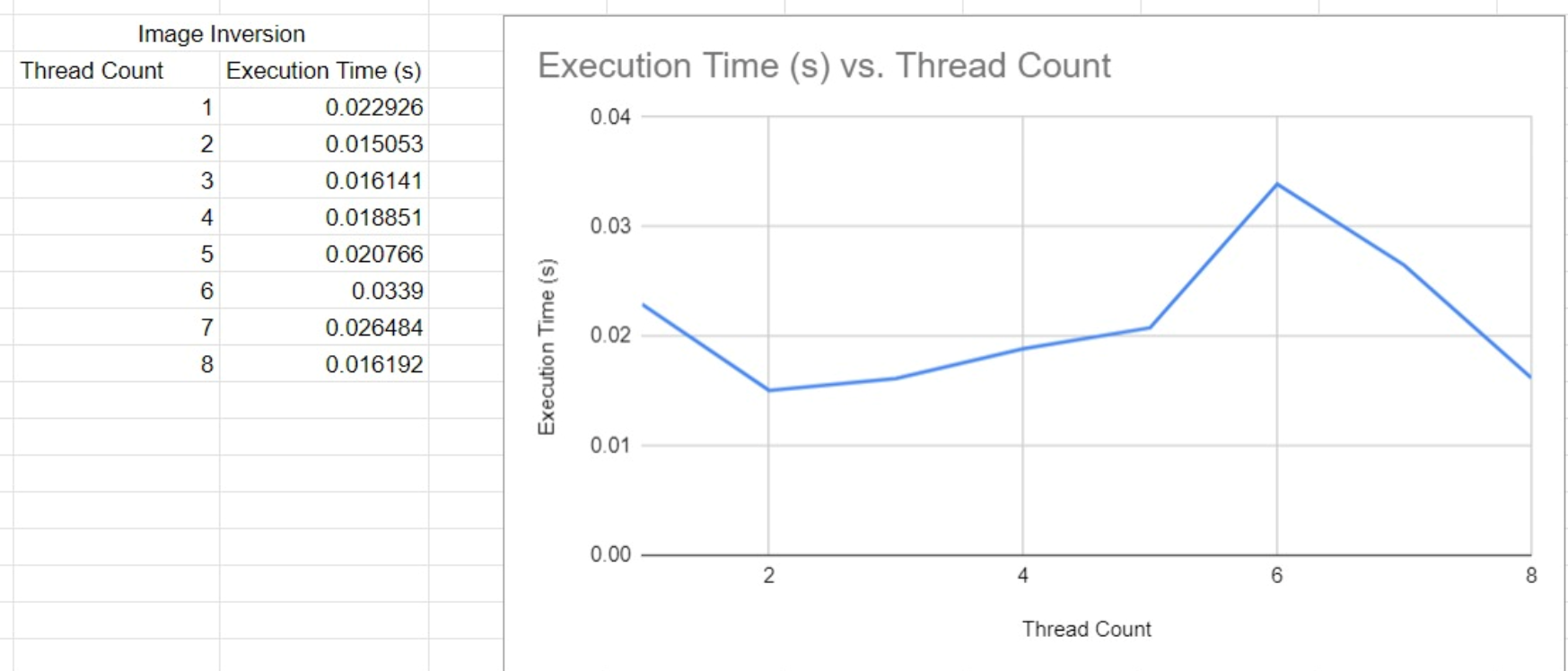
* Setting the Number of Threads:
  + The number of threads is set using the omp\_set\_num\_threads function to define the maximum number of threads available for parallel execution. This number is then varied during the performance measurement phase.
* Parallel Region:
  + OpenMP directives are used to define a parallel region, where the workload is divided among multiple threads. The #pragma omp parallel directive initiates the parallel region, and the specified block of code is executed concurrently by different threads.
* Loop Parallelization:
  + The most computationally intensive part of the algorithm, the nested loops responsible for convolution, is parallelized using the #pragma omp for directive.
  + The loop iteration space is divided among the available threads, and each thread processes a distinct portion of the image. This is achieved through automatic loop splitting, with each thread executing its assigned range of iterations.
* Thread-Specific Work:
  + Thread-specific variables are employed within the parallel region to capture information unique to each thread. For instance, the omp\_get\_thread\_num function is utilized to retrieve the thread ID, enabling identification of individual threads during performance measurement.

## Thread Utilization:

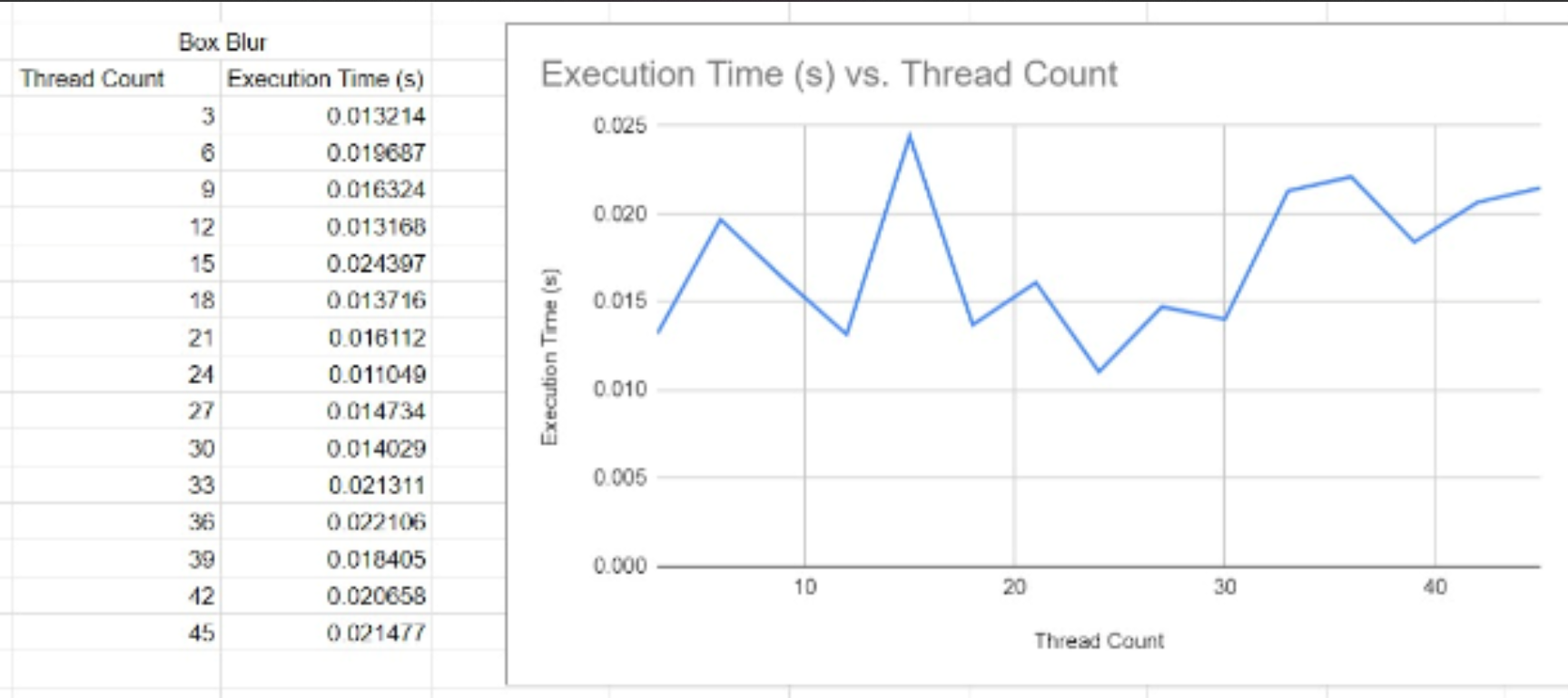
* Threads are employed to simultaneously process different sections of the image, significantly reducing the overall execution time compared to the serial version.
* The parallelized version exploits the inherent parallelism in image processing tasks, where the convolution operation for each pixel is independent, making it suitable for parallel execution.

**RESULTS:**

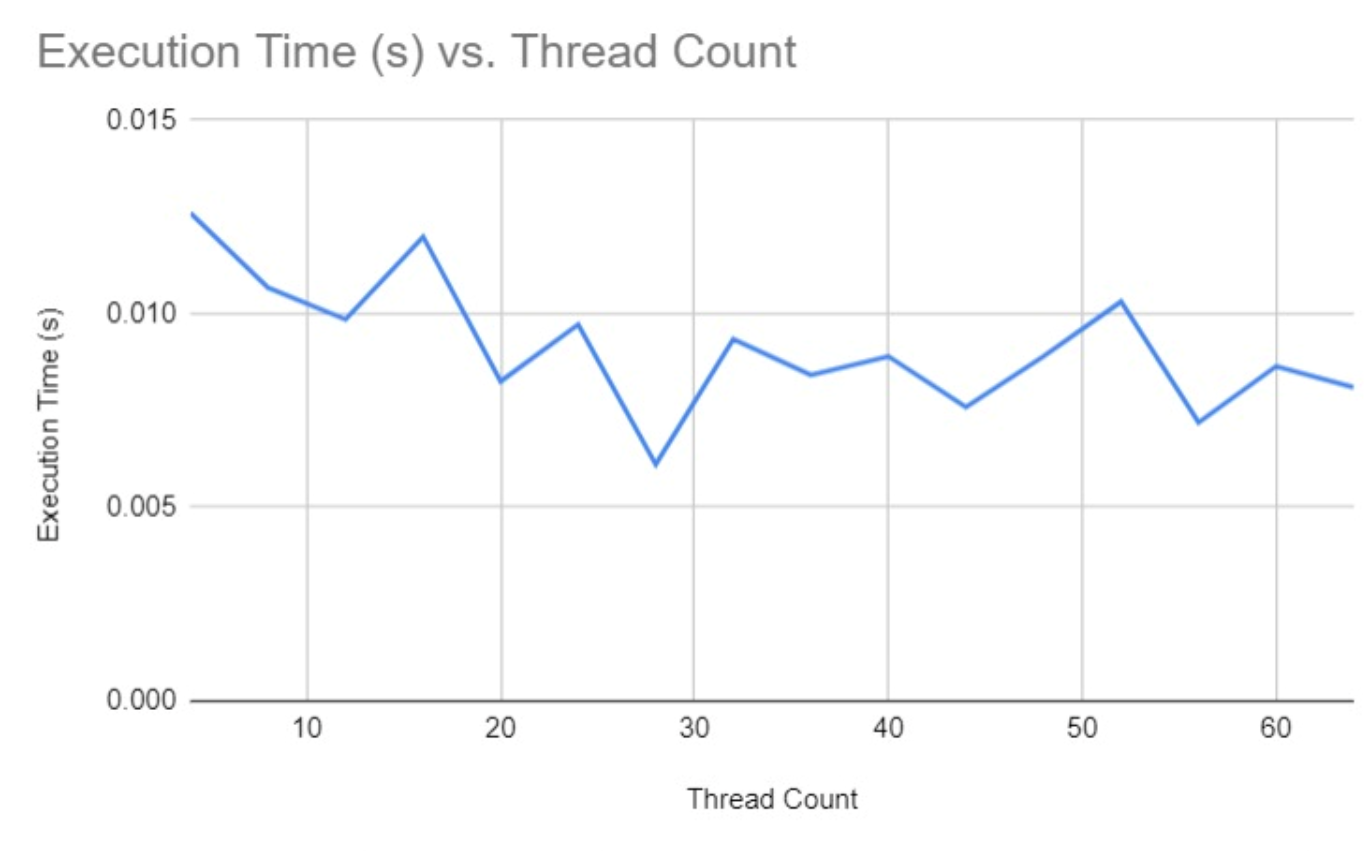
Image Inversion



Box Blur



Gaussian Blur



## Performance Measurement:

* Execution times are measured for different numbers of threads using the omp\_get\_wtime function. This allows for a quantitative assessment of the performance impact of parallelization.
* The performance results are printed, including the total execution time and the execution time per processor calculation. This information is crucial for evaluating the efficiency of the parallelized algorithm.

The use of OpenMP in this context demonstrates how parallel programming can be employed to optimize image processing algorithms, especially those with inherently parallelizable tasks like convolution.

# **6. Performance Analysis**

## Influence of Number of Threads:

The performance analysis involved varying the number of threads in the OpenMP parallel region from 4 to 64, with increments of 4 threads. OR from 3 threads to 72. This range was selected to explore how the algorithm scales with an increasing number of threads. The following key observations were made:

* Execution Time Reduction:
  + As the number of threads increases, the overall execution time decreases significantly. This reduction is attributed to the concurrent processing capabilities of multiple threads, effectively distributing the workload and accelerating the image processing task.
* Diminishing Returns:
  + While the execution time decreases with additional threads, the rate of improvement diminishes. This is a common characteristic in parallel computing, where the overhead of managing threads and synchronization can offset the benefits gained from parallel execution.
* Optimal Thread Count:
  + There exists an optimal number of threads that minimizes the execution time. Beyond this point, the diminishing returns become more pronounced, and the overhead of managing a larger number of threads can counteract the performance gains.

# **7. Conclusion**

In conclusion, this project aimed to implement and parallelize an image processing algorithm for blurring using OpenMP. The parallelization efforts were successful in significantly reducing the execution time compared to the serial implementation. The key findings and conclusions are as follows:

* Parallelization Success:
  + The OpenMP parallelization strategy effectively distributed the image processing workload across multiple threads, resulting in substantial performance improvements. The algorithm showcased scalability with an increasing number of threads, demonstrating its ability to leverage multi-core architectures.
* Challenges Faced:
  + While the parallelized algorithm exhibited notable success, challenges were encountered in achieving optimal performance gains. The diminishing returns observed with a higher number of threads emphasize the importance of balancing parallelization overhead with computational benefits.
* Optimization Opportunities:
  + Future improvements could focus on further optimizations, exploring alternative parallelization strategies, and considering specific image characteristics. Fine-tuning the algorithm for different types of images or implementing advanced parallelization techniques might lead to additional performance enhancements.

# **8. References**

The implementation of the project involved the use of OpenMP for parallelization. The following references provided valuable insights and guidance:

* OpenMP API Specification
  + [OpenMP](https://www.openmp.org/)
* FreeImage Documentation
  + [FreeImage](https://freeimage.sourceforge.io/documentation.html)
* LibJPEG
  + [LibJPEG](https://libjpeg.sourceforge.net/)

These references were instrumental in understanding OpenMP parallelization techniques and utilizing the FreeImage library for image processing tasks. The combination of these resources contributed to the successful implementation and parallelization of the image blurring algorithm.