International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 08 Issue: 09 | Sep 2021 www.irjet.net p-ISSN: 2395-0072

# A Comparative Analysis of Parallelisation Using OpenMP and Pymp for Image Convolution

**Awani Kendurkar[1], Mohith J[2]**

*[1]School of Information Technology and Engineering, Vellore Institute of Technology, Tamil Nadu, India.*
*[2]School of Computer Science and Engineering, Vellore Institute of Technology, Tamil Nadu, India.*

---

**Abstract -** *The introduction of multi-core processing into the realm of digital image processing has opened up avenues for faster execution of computationally intensive processes, like image convolutions. However, not all interfaces that provide multi-threading work in the same way. In this paper, we aim to study and parallelise two fundamental algorithms of digital image processing: Otsu segmentation and Sobel edge detection. Otsu's algorithm is a popular method that segments the pixels of an image into either foreground or background. Sobel filter, on the other hand, is widely used for edge detection. It classifies the image pixels into either edge or non-edge pixels and produces an output image that emphasises the edges. We use OpenMP and Pymp, an interface that aims at bringing OpenMP-like functionality to Python for parallelisation. We use nine images of increasing pixel sizes to perform convolution and compare OpenMP and Pymp. We also visualise our findings and use three performance metrics for comparison.*

***Key Words*: Digital Image Processing; Image convolution; OpenMP; Otsu segmentation; Parallel Programming; Pymp; Sobel edge detection**

## 1. INTRODUCTION

Parallel computing is becoming increasingly ubiquitous, especially since the advent of multi-core workstations. It is the simultaneous use of multiple computer resources to achieve a computationally intensive task. In simpler words, the task at hand is broken down into smaller sub-parts that can be performed individually, and then the results are combined upon completion. The parallel processing paradigm involves instantaneous utilisation of computer hardware to overcome memory constraints or to reduce the execution time effectively. Imagining a world without parallel processors today is next to impossible; all our smartphones and laptops are equipped with multi-processors. Thanks to these, most digital tasks are now accomplished in less than microseconds. Rapid advancement in this field has led to the invention of new programming languages and frameworks to parallelise the tasks that were previously coded sequentially. Being a new domain of study that has evolved massively in the past decade, there is a heightened curiosity to explore different tools and find out which tool is best for what application.

Parallel programming employs one of these three architectures: shared, distributed or hybrid memory architecture. Shared memory design refers to computers that use multiple processors but common memory resources. On the other hand, in distributed memory design, each processor has a designated memory. These are all usually connected over a network. Hybrid memory design amalgamates both shared and distributed memory types of design; in fact, most distributed networks are technically hybrid.

One of the most commonly used programming interfaces for parallelisation is OpenMP (Open Multi-Processing). It supports cross-platform shared memory parallel programming in languages like C, C++ and Fortran by using compiler directives, library routines and environment variables that govern run-time behaviour. Python enthusiasts have long tried to achieve parallelisation by multi-threading, but this is prohibited by GIL (global interpreter lock). Pymp is one such tool that brings OpenMP-like functionality to Python. While many other programming interfaces are extant in facilitating multi-processing, this paper focuses on OpenMP and Pymp. We choose OpenMP and Pymp because of their similar architecture and compare their performances when exposed to similar applications.

While there is a varying degree of potential in parallel processing applications, image convolution provides an inherent existence of parallelism. It is a critical operation in image processing, widely used for sharpening, smoothing, and edge detection. The fact that it is highly computationally intensive calls for parallel utilisation of computer resources. Its idea is linked to matrix multiplication, which is computationally costly, particularly in two-dimensional (2-D) convolution. The kernel may be split in some filters, such as Gaussian and Sobel, and 2-D convolution can be done as two 1-D convolutions, which is more economical [1]. On top of that, image quality is getting better each day, and high-resolution images need to be processed and stored on a daily basis for various applications. It would be apt to say that its relevance has been increasing exponentially in the last couple of years, its applications ranging from medicine to defence purposes. Segmentation and texture analysis are widely used for cancer and other disorder identifications. Digital image processing has been consistently used in military and security applications such as small target identification and tracking, missile guidance, vehicle navigation, broad area surveillance, and automatic/aided

target recognition. Many intelligent home systems or smart city systems employ image processing algorithms for intrusion detection or other purposes.

The Otsu image segmentation algorithm returns a single intensity threshold, determined by minimising intra-class intensity variance, which effectively separates pixels into two segments: foreground and background. The Sobel filter is an algorithm that creates an output image out of the input image, emphasising the edges by convolving the image with a small, separable, and integer-valued filter in the horizontal and vertical directions. These two algorithms were chosen as the focus of this paper for parallelisation since they are basic omnipresent algorithms in image processing. They also happen to possess inherent parallelism, which can be exploited to compare the performances of OpenMP and Pymp.

This work aims to compare the time taken to execute by OpenMP and Pymp when Otsu segmentation and Sobel edge detection algorithms are implemented. This comparison is made by calculating various performance metrics like speedup, efficiency and performance. Meaningful insights derived from graphical interpretations are analysed and discussed further.

This paper is organised as follows. Section 2 reviews related works on parallelisation in general, and more specifically, in image convolution. Existing literature on OpenMP, Pymp, Otsu thresholding and Sobel filter is reviewed as well. Section 3 elaborates on the parallel programming models that our experiments will be employing. Section 4 discusses the experimental setup, hardware and software specifications. Section 5 explains the performance metrics using which we will be comparing OpenMP and Pymp. Section 6 presents the experiment results and then discusses inferences. Finally, the conclusion is outlined in Section 7.

## 2. RELATED WORKS

An extensive and exhaustive review of existing literature on parallelisation reveals that many applications and implementation methods are extant. Since the concept of parallelism depends on various factors like the choice of compiler, computing language and even the choice of algorithm or problem statement, it is imperative to explore the impact these choices can have on a particular implementation.

### 2.1 Parallelisation of Image Convolution Algorithms

[2] elaborates convolution using procedural programming paradigm and object-oriented programming languages. Procedural programming produced better results compared to object-oriented in C++ implementations. The experimental results also show that Java has a shorter response time when compared to C++'s object-oriented implementation when parallelised. However, when compared to their sequential versions, object-oriented C++ had a better response time. The authors attribute this finding to better thread utilisation in Java. Furthermore, the parallelisation that was combined with concurrency was outperformed by pure parallelism.

[1] used compilers like ICC, GCC and LLVM to carry out 2D convolution on both single-core and multi-core. The implementations were done on OpenCV, OpenMP and Compilers Automatic vectorisation. The experimental results were obtained using three different sets of kernels of varying sizes. They suggest that the performance of GCC has much more significant improvement when compared to other compilers, with LLVM having no improvement between single-core and multi-core results.

[3] tries to make comparisons between image level and operational level parallelism. In operational level parallelism, different images are given to different cores, whereas in image level parallelism, the processing of one image is shared among different cores. Various image convolution techniques were applied to the image to make observations for this. The results favoured image-level parallelism as it is found to have higher speedups than operational level parallelism.

The performance of OpenCL implementations is comparatively less than OpenMP by a factor of two as per experimental observations in [4]. This finding can be attributed to very few overhead costs associated with OpenMP. In their experiments, GPRM implementation achieved the best performance for the largest images, and this is due to the fixed overhead costs associated with the model. Due to the same reason, the GPRM model is not suitable for small images. Even from [5], the findings suggest that OpenMP is the best parallel computing framework when computing resources like cores and memory are sufficient. MPI should be considered while dealing with moderate data size problems and for computationally very intensive problems. However, MapReduce could be the best framework to use when the dataset size is vast and if the computation does not require iterative processing. However, MapReduce's structure is less flexible relative to MPI. Hence, it is better to choose MPI when the program is being implemented in a distributed and parallel manner. The fundamental difference is using multiple cores for parallelising in MPI, while CUDA uses multiple accelerating units (GPUs), as mentioned in [6]. It also discusses the existence of hybrid models that reduce the execution time by a significant amount.

In experiments conducted in [7], parallel models have been implemented in Java, based on thread, message passing, and their hybrid. Threading proved to be an efficient method among the other two when tested on small datasets. However, when the dataset size is doubled, message passing gives the best performance. The hybrid model was slow in these experiments and did not show any significant performance improvement.

Another meaningful discussion that should be considered is programming productivity. In [8], the experiments were conducted using different analytical software tools to determine the programming productivity. The parallelisation

of code to some extent depends on the human factor too. However, the obtained results suggest that OpenCL requires more effort than OpenMP and OpenACC. With OpenMP being the one that requires minimal effort for programming. While this is true, discussions in [9] suggest that Phoenix++ needs less programming efforts than OpenMP, especially in application-oriented implementations. However, OpenMP remains unbeatable when it comes to performance and has significant speedup over Phoenix++.

## 2.2 OpenMP Implementations

[10] gives a relatively simple implementation of parallelism in image processing. It is implemented using OpenMP, and grey-scale images of size 3200 X 3200 are used. A speedup of 2.348 is achieved for a dual-core processor. A quad-core server is also used to experiment, and it attains a speedup of 4.452. The effect of increasing the number of threads used is analysed, and findings indicate that the computation time decreases when the number of threads is increased from 2 to 4. However, the subsequent increase in the number of threads causes the computation time to increase.

[11] proposes a parallel scheme for implementing a general FFT based algorithm for 2D convolutions. It further discusses the various costs involved, like computation, communication and load-balancing costs. An analysis of various levels of parallelism reveals the choice of exploiting the natural decomposition of the larger image into smaller sub-images. The use of IDL, OpenMP, and VSIPL gives an almost linearly increasing speedup as the number of processors increases.

[12] presents a practical approach to implement a parallel two-dimensional least mean square (TDLMS) filter in the spatial domain using OpenMP and C++. It is an adaptive image filter based on TDLMS, which means it is pixel-wise dependent and high computing power demanding. The primary approach is to split a given image into equal sub-blocks, each processed by the parallel running threads. The performance is examined using the speedup metric, the ratio of filtering durations for sequential and parallel implementations, respectively. The parallel implementation is realised using 2, 4 and 6 cores on images with sizes 512 x 512, 1024 x 1024 and 2048 x 2048. The speedup obtained is close to 1.9, 3.9 and 5.8, respectively, with sequential implementation using single-core.

[13] compares the performances of OpenMP and OpenCL in four different algorithms: matrix multiplication, n-queens problem, image convolution and string reversal. The speedup in the case of OpenMP remains constant when matrix multiplication is done with increasing matrix order, whereas that of OpenCL increases. Coming to the common problem of n-queens, an increased speedup in OpenCL when the number of queens is increased is observed, while the OpenMP implementation was not considered. OpenMP is better suited to image convolution than OpenCL, owing to background processes like kernel creation (seq/MP = 10.2, seq/CL = 0.53). String reversal requires OpenMP to run in its critical condition, which results in no performance improvement. Moreover, OpenCL is also unsuitable since it takes more time than sequential processing. [13] concludes that OpenCL ranks first performance-wise, followed by OpenMP and sequential processing.

[14] explores the performance enhancement offered by parallel processing using two tools: TBB (Threading Building Blocks) and OpenMP. The choice of algorithm is cubic convolution interpolation, and both dual-core and quad-core processors are used for parallel implementations. Two experiments conducted reveal that in dual-core processors, the speedups offered by TBB and OpenMP are 1.646 and 1.99, respectively, whereas, in the case of quad-core processors, the speedups are 3.289 and 3.807, respectively. It can be concluded that OpenMP is more suitable in the application of cubic convolution interpolation.

[15] uses parallel computation in a different scenario. A sequential calculation process focusing on the IR signature of an aerial object caused by reflection and radiation from the Earth's surface and the atmosphere is infeasible. This work presents parallelism to calculate the reflection of background radiation incidents from different directions in each spectral wavelength. It is implemented using OpenMP, OpenACC and CUDA, and their performances are compared. The speedup achieved with OpenMP is low for fewer threads but increases as the number of threads increases. OpenACC gives a speedup of approximately 140, whereas a naive implementation of CUDA gives a speedup of 295. Other implementations of CUDA may drive the speedup to as high as 426.

A performance evaluation is presented in [16] using time of execution, speedup and efficiency as performance metrics. The implementation is done for the matrix multiplication algorithm on a dual-core processor using two threads. From a range of 10 to 5000 for the matrix size, the speedup ranges from 0.168 to 2.224. Furthermore, the findings suggest that parallelism is only effective when the matrix size is greater than 50 X 50.

## 2.3 Pymp Implementations

[17] proposes a parallel algorithm for median filtering that uses Python and a multi-core processor architecture with the help of the pymp library. It aims to decrease time consumption using shared memory and multi-threading. The findings indicate that Python takes less time than C for median filtering when single-core is used. However, multi-core gives different results; for lower image sizes (512 x 512, 1024 x 1024), C performs better, but Python becomes more effective as the image size increases. Furthermore, Python gives an increasing speedup of 1.422, 2.465 and 4.279 when a quad-core processor is used for image size 512 x 512, 1024 x 1024 and 2048 x 2048, respectively. Pymp can have various applications, one of which can be in artificial intelligence, as is shown in [18]. This work uses parallel computation for face recognition and employs Pymp for a multi-processing library.

## 2.4 Otsu Implementations

Segmentation is an essential technique in digital image processing that bifurcates pixels into sets of different pixels.

Otsu is one such segmentation algorithm that identifies two sets of pixels; background and foreground pixels. Otsu uses image thresholding to transform the digital grey level in binary form as per [19]. Unlike K-means, where thresholding is localised, Otsu uses global thresholding to classify the backgrounds and foregrounds.

Otsu Canny edge detection, discussed in [20], presents the optimised Otsu segmentation that effectively classifies pixels as either edge or non-edge pixels. It has three more stages to be undergone before the thresholding, which helps build an effective model compared to its traditional implementation. The parallel implementation done using Hadoop in this paper reveals increasing speedup when parallelised for bigger image sizes.

[21] presents an in-depth review of Otsu's method for image thresholding. The method is discussed in detail, followed by a review of other literature. The flow is segmented into four phases: input of 2D image, conversion to grayscale, then histogram and finally thresholding.

[22] emphasises the importance of using a clean dataset while exposing it to Otsu binary segmentation. The paper discusses the parallelised implementation of Otsu using GPU, which is programmed in CUDA. It discusses many performance metrics, like F-measure, peak-signal-to-noise ratio, negative-rate metric and information-to-noise difference.

[23] makes use of OpenMP to write multi-threaded applications for image segmentation in plant species classification. The Canny-Edge detector and Otsu thresholding methods are used and compared for their efficiency in a quad-core processor. The implementation of parallelism in both methods is discussed. Experiments reveal an increasing speedup for both implementations when the number of cores is increased. It is important to note that Canny-Edge implementation of coarse parallelism gives more speedup than Otsu's coarse-grain.

## 2.5 Sobel Implementations

The Sobel operator is a fundamental, efficient and popular image convolution technique for edge detection. The operator carries out its function by calculating a gradient of the 2D grayscale image fed. As described in [24], the operator has two convolution masks that determine the horizontal and vertical gradients, Gx and Gy, respectively, at each point. These kernels are just 90-degree rotations of each other. The approximate absolute gradient is determined by combining the horizontal and vertical gradients.

[25] mentions the advantage Sobel operator has over other edge detectors as it highlights the edges by making them thicker and brighter. The filter is also less sensitive to noise. The paper presents another critical observation that the time taken for parallel execution, MPI in this particular implementation, reduces drastically until the number of processors increases. However, after a threshold of three processors, the reduction in execution time is less significant. The authors suggest that this can be attributed to using a dual-core machine, and the threshold would be high if the experiment is conducted on a machine with more than two cores.

[26] discusses a novel approach of using a 5 x 5 kernel rather than the usual 3 x 3 kernel used while applying a Sobel operator. The implementation was done in OpenCL using both CPU and GPU to calculate speed ups. The speedup achieved by the 5 x 5 kernel is almost the same as the 3 x 3 kernel, and the paper states that further research should be carried out to get optimised results. Although the speedups were almost identical, the output images contained relatively better edges than outputs collected with the 3 x 3 kernel.

Sobel operator, primarily as an edge detector, can have various applications, one of which can be in facial recognition, as is shown in [27]. This work shows that the filter helps determine the edge orientation, structure, and cancelling noise in identifying facial images.

# 3. PARALLEL PROGRAMMING MODELS

## 3.1 OpenMP

OpenMP is a popular parallel programming library with compiler directives based on the SMP (shared memory processors) model. It uses the concept of threads to achieve parallelism. The main thread, also called the master thread, gets forked into multiple threads, called slave threads, when it encounters a parallel block in the program. OpenMP is flexible and helps programmers develop applications with ease because of its scalability. It is versatile and can be run on various operating systems, like Windows, macOS, Linux and Solaris, and various compilers, including GCC, Intel, IBM XL, and LLVM/Clang. OpenMP supports both fine-grained as well as coarse-grained parallelisation.

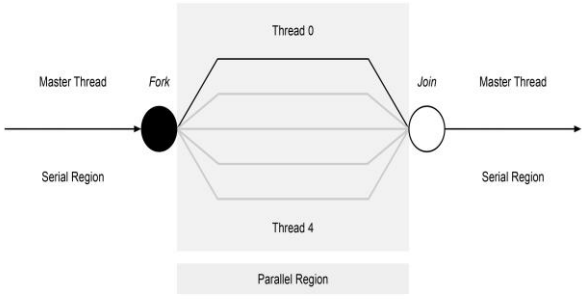


**Fig – 1**: OpenMP master-thread architecture

The number of threads created depends upon the cores available in the system. The programmer can also set the number of threads manually. OpenMP has the feature of sharing variables among threads. All variables are given three primary data-sharing attributes; shared, private and default. These features in OpenMP tackle the problem of race conditions in parallel programming. Besides this, it can also synchronise among threads. The major synchronisations available are critical, atomic, ordered, barrier and no-wait. Loops can be parallelised with the help of work-sharing

constructs. It eases out considerable work for the programmer as it is straightforward and only requires specifying a few parameters.

### 3.2 Pymp

Pymp is an OpenMP-like tool in the Python programming language that helps in parallelisation. Programmers encounter the constraint of GIL, Global Interpreter Lock, that disallows carrying out multi-processing in Python using threads. Even though tools like Ctypes and Cython exist, which can run computationally intensive code outside Python language, they primarily bypass Python to achieve multi-processing through threads. Pymp provides a way to circumvent GIL to perform threaded multi-processing in Python. To be specific, the operating system's forking method is used by Pymp to circumvent GIL. It greatly assists in achieving significantly less overhead costs. Also, the results are achieved in the expected semantics. It has a similar master-slave architecture like OpenMP, as shown in Figure 1, and the forking happens when the compiler encounters a parallelisable block of code. Pymp has a unique feature of conditional parallelism, which helps deactivate parallelisation regardless of other settings with the constructor in its parallel region. Unlike OpenMP, when child processes are forked, the memories are referenced, not shared. This characteristic helps in keeping the process overheads low. The only downside Pymp poses is that the library can only operate on fork supported systems.

It can achieve OpenMP-like parallelisation by using pymp.range and pymp.xrange statements. Like in OpenMP, one can manage the aspects of the code by using environment variables. It also has the flexibility of deciding the number of threads for the parallelisable blocks. The threads in Pymp are divided into a producer-consumer pattern, with the main thread being the producer and the rest of the threads acting as the consumer threads.

## 4. EXPERIMENTAL SETUP

An extensive and exhaustive review of existing literature on parallelisation reveals that many applications and implementation methods are extant. Since the concept of parallelism depends on various factors like the choice of compiler, computing language and even the choice of algorithm or problem statement, it is imperative to explore the impact these choices can have on a particular implementation.

### 4.1 Algorithms Used

The algorithm used in the experiment for Otsu segmentation is presented in Algorithm 1.

**ALGORITHM 1: OTSU SEGMENTATION ALGORITHM**

1 Input image
2 Generate a histogram using 2D array
3 Generate probability density using 2D array
4 Generate Ω and μ
5 Maximise inter-class variance
6 Determine optimum threshold value
7 Convert pixels with values more than threshold to white, otherwise black
8 Output image

The algorithm used in the experiment for Sobel edge detection is presented in Algorithm 2.

**ALGORITHM 2: SOBEL EDGE DETECTION ALGORITHM**

1 Input image
2 Read the image into a 2D array
3 Apply Kernel 1 on each pixel to get $G_x$
4 Apply Kernel 2 on each pixel to get $G_y$
5 Computer absolute value of each pixel in $G_x$
6 Computer absolute value of each pixel in $G_y$
7 Determine threshold value using Manhattan distance
8 Convert pixels with values more than threshold to black, otherwise white
9 Output image

Usually, the threshold value is determined by Euclidean distance formula as shown by formula (1)

$$|G| = \sqrt{G_x^2 + G_y^2} \quad (1)$$

But to achieve better performance, we will be compromising with the Manhattan distance formula as shown in formula (2).

$$|G| = |G_x| + |G_y| \quad (2)$$

For the Sobel edge detection, the two 3x3 kernels used are represented in Figures 1 and 2.

| | | |
|---|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

**Fig – 2**: Kernel 1

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

**Fig – 3:** Kernel 2

For the Sobel edge detection, the two 3x3 kernels used are represented in Figures 2 and 3.

### 4.2 Dataset Used

The input images for this experiment were picked with careful consideration, so they have both foreground and background. This helps in distinctly pursuing the detection and segmentation work carried out. All images that are being used in this experiment are in .pgm format. Images of different sizes are obtained in the .jpg format and then converted to .pgm format. All images are converted to make

the computations with the image easier as it creates a 2D grayscale image. The different image sizes that are exposed to the image processing programs are 100 x 100, 250 x 250, 500 x 500, 1000 x 1000, 2000 x 2000, 2500 x 2500, 3000 x 3000, 4000 x 4000 and 5000 x 5000.

### 4.3 Hardware and Software Specifications

All the experiments in this study were carried out in a single machine with Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz microprocessor. The RAM capacity was 4GB, and experiments were carried out in Ubuntu Linux 64-bit operating system. The compiler used was GCC compiler to run C and OpenMP programs as its performance improvements are higher than other C compilers like ICC and LLVM. For Python programs, Python version 3.6.9 was used. In order to implement parallel programming in Python, the pymp-pypi package was installed using pip install.

## 5. PERFORMANCE METRICS

In order to evaluate and compare the results obtained from our experiments, we will be using three performance metrics used in parallel computing. They are presented in this section. These metrics help in making inferences about the parallelism achieved.

### 5.1 SpeedUp

This performance metric provides us the relative benefit of parallelising a problem. It is defined as the ratio of time taken to execute a program sequentially to the time taken to execute the same program parallelly. The speedup formula is presented by formula (3).

$$Speedup = \frac{Sequential\ Time}{Parallel\ Time} \quad (3)$$

### 5.2 Efficiency

This performance metric illustrates the utilisation of the processor. It is defined as the ratio of speedup achieved for solving a problem to the number of processors used to solve the problem. The efficiency formula is presented by the formula (4).

$$Efficiency = \frac{Sequential\ Time}{No.of\ Processors \times Parallel\ Time} \quad (4)$$

### 5.3 Improvement

This performance metric gives us the knowledge of the relative improvement in performance achieved by solving a problem in parallel rather than solving the same problem sequentially. It is defined as the ratio of the difference in execution time of parallel and sequential implementations of a problem to the sequential execution time of the problem. The improvement formula is presented by the formula (5).

$$Improvement = \frac{Sequential\ Time - Parallel\ Time}{Sequential\ Time} \quad (5)$$

## 6. RESULTS AND DISCUSSIONS

The obtained results are presented and visualised to get meaningful inferences from them. This section will outline the readings from the experiments carried out and discuss some essential observations of our study.

### 6.1 Experimental Results

These experimental results are based on the execution time of both algorithms in C and Python. The results presented are for 8 scales of different image sizes ranging from 100 x 100 to 5000 x 5000. Table 1 represents the execution time for parallel and sequential implementations in C and Python for Otsu segmentation.

**Table – 1:** Execution time for Otsu segmentation in C and Python

| Size of Image | C Sequential | OpenMP Parallel | Python Sequential | Pymp Parallel |
|---|---|---|---|---|
| 100 X 100 | 0.0004 | 0.0005 | 0.0599 | 0.1109 |
| 250 X 250 | 0.0010 | 0.0015 | 0.3569 | 0.2186 |
| 500 X 500 | 0.0018 | 0.0269 | 1.4108 | 0.6552 |
| 1000 X 1000 | 0.0062 | 0.0330 | 5.6340 | 2.8196 |
| 2000 X 2000 | 0.0213 | 0.0590 | 22.9136 | 10.2911 |
| 2500 X 2500 | 0.0318 | 0.0765 | 35.3827 | 17.5532 |
| 3000 X 3000 | 0.0458 | 0.0934 | 51.7487 | 24.5988 |
| 4000 X 4000 | 0.0775 | 0.1693 | 104.3175 | 49.5334 |
| 5000 X 5000 | 0.1177 | 0.2015 | 121.6839 | 62.8480 |

**Table – 2:** Execution time for Sobel edge detection in C and Python

| Size of Image | C Sequential | OpenMP Parallel | Python Sequential | Pymp Parallel |
|---|---|---|---|---|
| 100 X 100 | 0.0016 | 0.0013 | 0.9770 | 0.5123 |
| 250 X 250 | 0.0018 | 0.0016 | 5.9509 | 1.8923 |
| 500 X 500 | 0.0044 | 0.0026 | 29.3704 | 7.9721 |
| 1000 X 1000 | 0.0147 | 0.0073 | 125.8100 | 38.9696 |
| 2000 X 2000 | 0.0571 | 0.0245 | 458.0815 | 181.7700 |
| 2500 X 2500 | 0.0877 | 0.0407 | 717.8653 | 280.2159 |
| 3000 X 3000 | 0.1274 | 0.0523 | 1173.1765 | 444.1182 |
| 4000 X 4000 | 0.2187 | 0.0899 | 1831.8895 | 737.4789 |
| 5000 X 5000 | 0.3433 | 0.1442 | 3035.8576 | 1166.3124 |

Table 2 represents the execution time for parallel and sequential implementations in C and Python for Sobel edge detection.

**Fig – 4:** Input image

**Fig – 5:** After Otsu segmentation

We have also presented a sample of the results of the computation carried out. Figure 4 represents the input image of 3000 x 3000 that is exposed to image convolution. Figure 5 represents the image after exposure to Otsu segmentation. Figure 6 represents the image after applying a Sobel edge detector.

In order to get insights from the obtained results, we visualised the results in the form of line graphs. The line graphs are created for execution time vs image sizes. Chart 1 shows the visualisation for Otsu segmentation using C, Chart 2 shows the same for Otsu segmentation using Python, Chart 3 shows Sobel edge detection using C and Chart 4 shows Sobel edge detection using Python.

**Fig – 6:** After Sobel edge detection

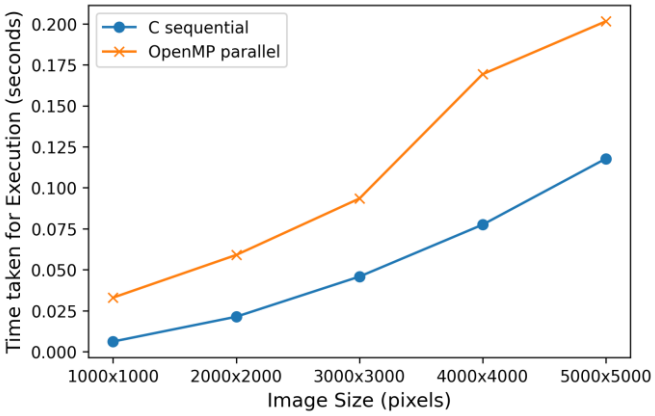


**Chart – 1:** Graph of execution time for Otsu segmentation using C

## 6.2 Calculation of performance metrics

To understand the level of parallelisation achieved by our experiment, we have calculated the values of performance metrics discussed in section 5. Table 3 illustrates the three performance metrics; speedup, efficiency and improvement for Otsu segmentation implemented in OpenMP and Pymp.

Table 4 illustrates the performance metrics speedup, efficiency and improvement for Sobel edge detection implemented in OpenMP and Pymp.

Charts 5 and 6 show a visualisation of the information presented in Table 4 for better perception.

**Table – 3:** Speedup, efficiency and improvement for Otsu segmentation implemented in OpenMP and Pymp

| Size of Images | OpenMP | | | Pymp | | |
|---|---|---|---|---|---|---|
| | *Speedup* | *Efficiency* | *Improvement* | *Speedup* | *Efficiency* | *Improvement* |
| 100 X 100 | 0.7712 | 0.1928 | -0.2967 | 0.5398 | 0.1350 | -0.8525 |
| 250 X 250 | 0.7036 | 0.1759 | -0.4214 | 1.6322 | 0.4080 | 0.3873 |
| 500 X 500 | 0.0666 | 0.0166 | -14.0257 | 2.1531 | 0.5383 | 0.5356 |
| 1000 X 1000 | 0.1883 | 0.0471 | -4.3119 | 1.9982 | 0.4995 | 0.4995 |
| 2000 X 2000 | 0.3616 | 0.0904 | -1.7653 | 2.2266 | 0.5566 | 0.5509 |
| 2500 X 2500 | 0.4155 | 0.1039 | -1.4067 | 2.0157 | 0.5039 | 0.5039 |
| 3000 X 3000 | 0.4904 | 0.1226 | -1.0391 | 2.1037 | 0.5259 | 0.5246 |
| 4000 X 4000 | 0.4578 | 0.1144 | -1.1846 | 2.1060 | 0.5265 | 0.5252 |
| 5000 X 5000 | 0.5843 | 0.1461 | -0.7115 | 1.9362 | 0.4840 | 0.4835 |

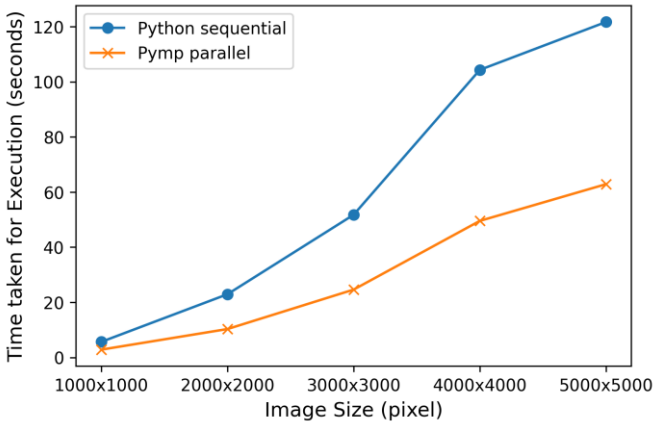


**Chart – 2:** Graph of execution time for Otsu segmentation using Python

The highest performance achieved in our experiment is a speedup of 3.6 by pymp while implementing Sobel edge detection.

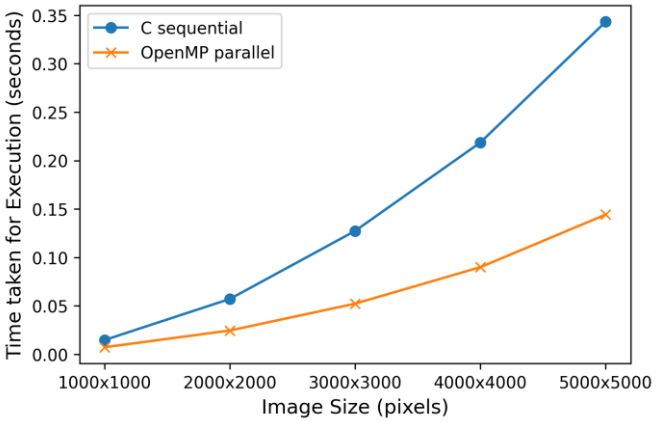


**Chart – 3:** Graph of execution time for Sobel edge detection using C

## 6.3 Observation and Inferences

This section provides the observations and key insights obtained from the experiments conducted. Table 5 presents a

**Table – 4:** Speedup, efficiency and improvement for Sobel edge detection implemented in OpenMP and Pymp

| Size of Images | OpenMP | | | Pymp | | |
|---|---|---|---|---|---|---|
| | *Speedup* | *Efficiency* | *Improvement* | *Speedup* | *Efficiency* | *Improvement* |
| 100 X 100 | 1.2676 | 0.3169 | 0.2111 | 1.9072 | 0.4768 | 0.4757 |
| 250 X 250 | 1.1271 | 0.2818 | 0.1128 | 3.1448 | 0.7862 | 0.6820 |
| 500 X 500 | 1.7208 | 0.4302 | 0.4189 | 3.6842 | 0.9210 | 0.7286 |
| 1000 X 1000 | 2.0215 | 0.5054 | 0.5053 | 3.2284 | 0.8071 | 0.6903 |
| 2000 X 2000 | 2.3288 | 0.5822 | 0.5706 | 2.5201 | 0.6300 | 0.6032 |
| 2500 X 2500 | 2.1555 | 0.5389 | 0.5361 | 2.5618 | 0.6405 | 0.6097 |
| 3000 X 3000 | 2.4371 | 0.6093 | 0.5897 | 2.6416 | 0.6604 | 0.6214 |
| 4000 X 4000 | 2.4314 | 0.6079 | 0.5887 | 2.4840 | 0.6210 | 0.5974 |
| 5000 X 5000 | 2.3810 | 0.5952 | 0.5800 | 2.6030 | 0.6507 | 0.6158 |

consolidated view of performance across both the image convolution techniques by calculating mean values of all the image sizes.

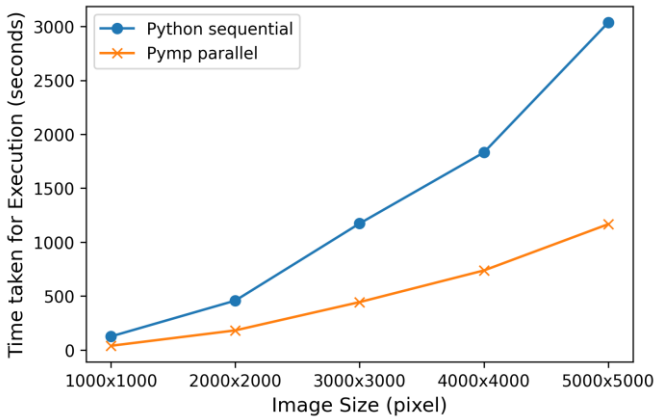


**Chart – 4:** Graph of execution time for Sobel edge detection using Python

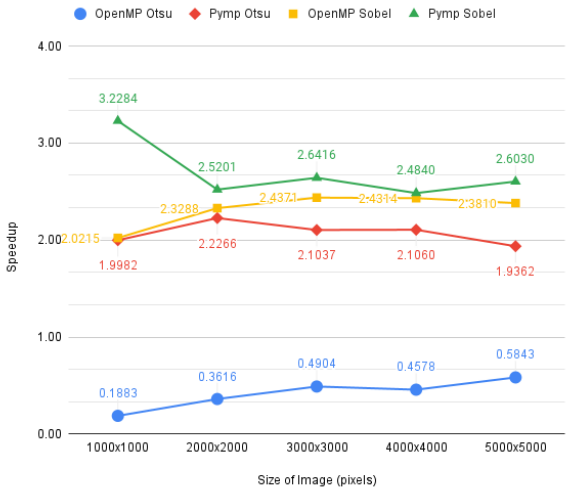


**Chart – 5:** Graph for speedup comparison

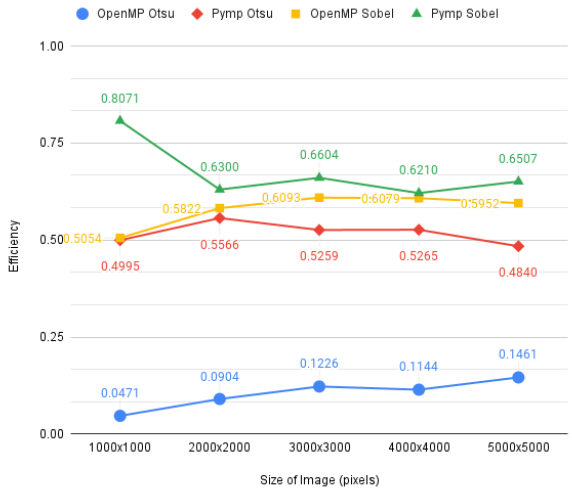


**Chart – 6:** Graph for efficiency comparison

Graphs in the Chart 7 for Otsu segmentation and Chart 8 for Sobel Edge Detection present the findings from Table 5 for better comparison.

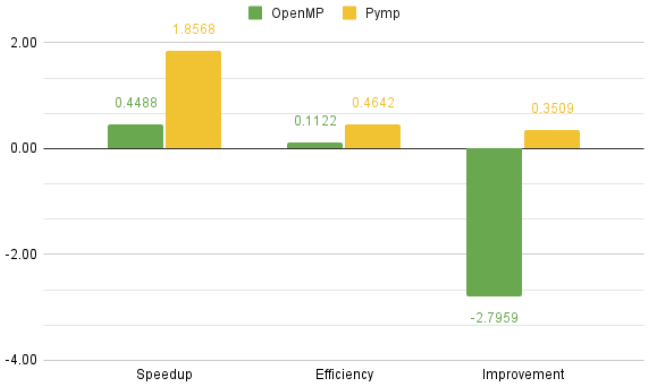


**Chart – 7:** Mean performance metric of all image sizes for Otsu segmentation

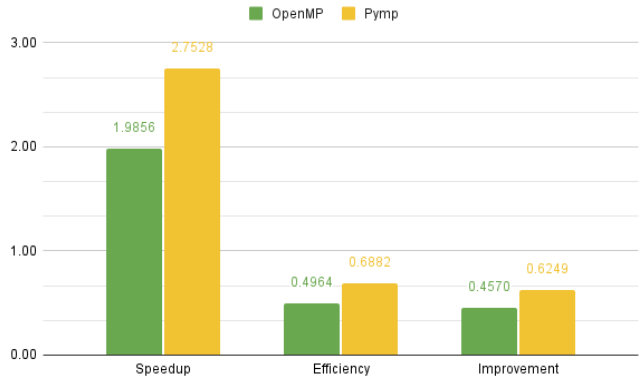


**Chart – 8:** Mean performance metric of all image sizes for Sobel edge detection

The first and foremost inference that can be made is that the execution time for both image convolution techniques is significantly less while it is implemented in C when compared to Python. This phenomenon can be attributed to the fact that Python is a very high-level programming language and takes more execution time for the same task. Although the time taken by Python is more, we can observe that the performance of Python Pymp is significantly better than OpenMP. Pymp can bring down the computation time to a greater extent when parallelised than OpenMP. Thus we can summarise that Python Pymp parallelises tasks better than OpenMP.

Another important observation to be discussed is that Otsu segmentation's sequential implementation in C takes less time than its parallel implementation, as shown in Figure 6. Because of this, Otsu's OpenMP speedup is below one, and the improvement is negative. This could be due to overhead costs associated with it. Otsu implementation requires creating and maintaining more threads, and hence it is prone to higher overhead costs.

Even the readings for the 100 x 100 image in Otsu segmentation done with Python had sequential time lesser than parallel time. The problem was not big enough for Pymp

to make some merit over its sequential counterpart. The time involved in thread creation and management took longer than the actual parallelisation.

Furthermore, we can interpret from Charts 1, 2 and 3 that the amount of parallelisation achieved increases with the increase in image size. In other words, more time is being saved by parallelising a bigger image than parallelising a smaller one.

### 6.4 Future Scope

As discussed in the observations and inferences, the time taken for sequential execution of Otsu segmentation was less than that for parallel execution. OpenMP sometimes provides the disadvantages of high overheads. A potential area of study would be to clearly identify the problem associated with it and develop an algorithm for Otsu segmentation optimised for parallel execution. This research could also be extended to other parallel programming frameworks using GPU. In our future works, we aim to extend the study to CUDA and OpenCL-based implementations and study their behaviour on more image convolution operations. We also aim at expanding our study with images of higher resolution, more than 5000 x 5000.

## 7. CONCLUSION

In recent years, parallel programming has become a vital part of digital image processing, owing to multiple advantages like faster execution and utilising all the computer resources available. Many tools and frameworks are now available for enthusiasts trying to speed up their problem-solving tasks, out of which we have studied and compared two for image convolution: OpenMP and Pymp. These interfaces were compared for two algorithms, Otsu segmentation and Sobel edge detection. We carried out experiments for nine images of different sizes and tabulated the time taken to execute sequentially and parallelly. The readings were visualised using graphs to generate meaningful insights. We observed that, generally, implementation in Python took a longer time than C. It can be because Python is a high-level language and requires more time to break down into low-level code blocks. Both OpenMP and Pymp are prone to overheads when exposed to computationally less challenging tasks or a small dataset. Pymp in Python performs better parallelisation of a problem than OpenMP. On average, Pymp exhibited a speedup of about 2.75 for Otsu segmentation and about 1.86 for Sobel edge detection. In contrast, OpenMP was only able to achieve a speedup of 1.98 for Otsu segmentation. In Sobel edge detection, the speedup was less than one since sequential implementation ran faster, owing to the lack of any overheads. In the future, we aim at extending our study to more parallel programming tools and would also like to explore the effect of these tools on more complex image convolution techniques.

## REFERENCES

[1] Amiri, Hossein, and Asadollah Shahbahrami. "High performance implementation of 2-D convolution using AVX2." 2017 19th International Symposium on Computer Architecture and Digital Systems (CADS). IEEE, 2017.

[2] Da Penha, Dulcinéia O., et al. "Performance evaluation of programming paradigms and languages using multithreading on digital image processing." Proceedings of the 4th WSEAS International Conference on Applied Mathematics and Computer Science. 2005.

[3] Zade, Saurabh, et al. "Performance Analysis of Parallel Image Processing Operations." 2020 International Conference on Communication and Signal Processing (ICCSP). IEEE, 2020.

[4] Tousimojarad, Ashkan, Wim Vanderbauwhede, and W. Paul Cockshott. "2D Image Convolution using Three Parallel Programming Models on the Xeon Phi." arXiv preprint arXiv:1711.09791 (2017).

[5] Kang, Sol Ji, Sang Yeon Lee, and Keon Myung Lee. "Performance comparison of OpenMP, MPI, and MapReduce in practical problems." Advances in Multimedia 2015 (2015).

[6] Rastogi, Shubhangi, and Hira Zaheer. "Significance of Parallel Computation over Serial Computation Using OpenMP, MPI, and CUDA." Quality, IT and Business Operations. Springer, Singapore, 2018. 359-367.

[7] Jamaluddin, Muhammad & Ismail, Azlan & Rashid, Amir & Omar Takleh, Talha Takleh. (2019). "Performance Comparison of Java based Parallel Programming Models." Indonesian Journal of Electrical Engineering and Computer Science. 16. 1577-1583. 10.11591/ijeecs.v16.i3.pp1577-1583.

[8] Memeti, Suejb, et al. "Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption." Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing. 2017.

[9] Arif, Mahwish, and Hans Vandierendonck. "A case study of openmp applied to map/reduce-style computations." International Workshop on OpenMP. Springer, Cham, 2015.

[10] Xionggang, Tu, and Chen Jun. "Parallel image processing with OpenMP." 2010 2nd IEEE International Conference on Information Management and Engineering. IEEE, 2010.

[11] Kepner, Jeremy. "A multi-threaded fast convolver for dynamically parallel image filtering." Journal of Parallel and Distributed Computing 63.3 (2003): 360-372.

[12] Akgün, Devrim. "A practical parallel implementation for TDLMS image filter on multi-core processor." Journal of Real-Time Image Processing 13.2 (2017): 249-260.

[13] Thouti, Krishnahari, and S. R. Sathe. "Comparison of OpenMP & OpenCL parallel processing technologies." arXiv preprint arXiv:1211.2038 (2012).

[14] Liu, Ying, and Fuxiang Gao. "Parallel implementations of image processing algorithms on multi-core." 2010 Fourth International Conference on Genetic and Evolutionary Computing. IEEE, 2010.

[15] Guo, Xing, et al. "Parallel computation of aerial target reflection of background infrared radiation:

Performance comparison of OpenMP, OpenACC, and CUDA implementations." IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 9.4 (2016): 1653-1662.

[16] Dash, Yajnaseni, Sanjay Kumar, and V. K. Patle. "Evaluation of performance on OPEN MP parallel platform based on problem size." International Journal of Modern Education and Computer Science 8.6 (2016): 35.

[17] Sharmila, B. S., and Narasimha Kaulgud. "Comparison of time complexity in median filtering on multi-core architecture." 2017 3rd International Conference on Advances in Computing, Communication & Automation (ICACCA)(Fall). IEEE, 2017.

[18] Balachandran, Bashini, et al. "Parallel Computer For Face Recognition Using Artificial Intelligence." 2019 14th International Conference on Computer Engineering and Systems (ICCES). IEEE, 2019.

[19] Kumar, Arpan, and Anamika Tiwari. "A Comparative Study of Otsu Thresholding and K-means Algorithm of Image Segmentation." Int. J. Eng. Technol. Res 9 (2019): 2454-4698.

[20] Singh, Brij Mohan, et al. "Parallel implementation of Otsu's binarization approach on GPU." International Journal of Computer Applications 32.2 (2011): 16-21.

[21] Bangare, Sunil L., et al. "Reviewing otsu's method for image thresholding." International Journal of Applied Engineering Research 10.9 (2015): 21777-21783.

[22] Cao, Jianfang, et al. "Implementing a parallel image edge detection algorithm based on the Otsu-canny operator on the Hadoop platform." Computational intelligence and neuroscience 2018 (2018).

[23] Rahman, M. Nordin A., et al. "Image segmentation using openmp and its application in plant species classification." International Journal of Software Engineering and Its Applications 9.5 (2015): 135-144.

[24] Fredi, Hana Ben, et al. "Parallel implementation of Sobel filter using CUDA." 2017 International Conference on Control, Automation and Diagnosis (ICCAD). IEEE, 2017.

[25] Tartory, Haneen, and Mohammed ALDasht. "Parallelization of Sobel Edge Detection Algorithm." (2012).

[26] Sanida, Theodora, Argyrios Sideris, and Minas Dasygenis. "A Heterogeneous Implementation of the Sobel Edge Detection Filter Using OpenCL." 2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST). IEEE, 2020.

[27] Sharma, Achal, and Shilpa Jaswal. "Analysis of sobel edge detection technique for face recognition." International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) Volume 4 (2015).