Scientific Research Publishing

Journal of Computer and Communications, 2024, 12, 1-10
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Parallel Image Processing: Taking Grayscale Conversion Using OpenMP as an Example

Bayan AlHumaidan*, Shahad Alghofaily, Maitha Al Qhahtani, Sara Oudah, Naya Nagy

Department of Computer Science, College of Computer Science & Information Technology, Imam Abdulrahman Bin Faisal University, Dammam, Saudi Arabia
Email: *2200001837@iau.edu.sa, 2190002020@iau.edu.sa, 2190004208@iau.edu.sa, 2200006100@iau.edu.sa, nmnagy@iau.edu.sa

How to cite this paper: AlHumaidan, B., Alghofaily, S., Al Qhahtani, M., Oudah, S. and Nagy, N. (2024) Parallel Image Processing: Taking Grayscale Conversion Using OpenMP as an Example. *Journal of Computer and Communications*, **12**, 1-10.
https://doi.org/10.4236/jcc.2024.122001

Received: December 8, 2023
Accepted: February 3, 2024
Published: February 6, 2024

Abstract

In recent years, the widespread adoption of parallel computing, especially in multi-core processors and high-performance computing environments, ushered in a new era of efficiency and speed. This trend was particularly noteworthy in the field of image processing, which witnessed significant advancements. This parallel computing project explored the field of parallel image processing, with a focus on the grayscale conversion of colorful images. Our approach involved integrating OpenMP into our framework for parallelization to execute a critical image processing task: grayscale conversion. By using OpenMP, we strategically enhanced the overall performance of the conversion process by distributing the workload across multiple threads. The primary objectives of our project revolved around optimizing computation time and improving overall efficiency, particularly in the task of grayscale conversion of colorful images. Utilizing OpenMP for concurrent processing across multiple cores significantly reduced execution times through the effective distribution of tasks among these cores. The speedup values for various image sizes highlighted the efficacy of parallel processing, especially for large images. However, a detailed examination revealed a potential decline in parallelization efficiency with an increasing number of cores. This underscored the importance of a carefully optimized parallelization strategy, considering factors like load balancing and minimizing communication overhead. Despite challenges, the overall scalability and efficiency achieved with parallel image processing underscored OpenMP's effectiveness in accelerating image manipulation tasks.

Keywords

Parallel Computing, Image Processing, OpenMP, Parallel Programming, High Performance Computing, GPU (Graphic Processing Unit)

1. Introduction

In recent years, parallel computing has become increasingly spreading, particularly in the context of multi-core processors and high-performance computing environments, like multi-core workstations. The concept of parallelism involves the simultaneous execution of multiple tasks, allowing for the acceleration of computation and the efficient use of available resources. In simpler terms, this approach involves breaking down the given task into smaller sub-parts that can be independently processed, and the results are then combined upon completion. This rise in parallel computing is particularly significant in the field of image processing, which has experienced remarkable advancements in recent years. Image processing, at its core, involves the application of algorithms and techniques to manipulate and analyze visual data and encompasses a diversity of tasks. Among these tasks, the conversion of colorful images to grayscale stands out as a classic operation that plays a central role in various domains, such as medical imaging and computer vision. The growing demand for real-time image processing has led to the rise of parallel computing as a crucial facilitator, which offers the potential to significantly improve the speed and efficiency of image processing with large datasets and complex algorithms [1].

One of the popular approaches to parallelism is OpenMP (Open Multi-Processing) which is one of the most widely and powerful Application Programming Interfaces (APIs) for parallelization. It enables cross-platform parallel shared memory programming in languages such as C, Fortran [1], and C++, through the use of environment variables, library routines, and compiler directives, all of which control the runtime behavior of the parallelized processes. OpenMP allows developers to harness the power of multi-core processors by dividing a task into parallel threads that can be executed concurrently. OpenMP has the benefits of being powerful and well-suited for modern processor architectures and C/C++ and Fortran compilers as well as various operating systems, such as, Linux, Microsoft Windows, and Apple Macintosh OS X, and most importantly being incredibly easy to understand and use. Click or tap here to enter text.

This parallel project embarks on the exploration of parallel image processing, specifically focusing on the conversion of colorful images to grayscale, a task foundational to various computer vision applications. By incorporating OpenMP into our framework, and employing parallelism in our project, we seek to optimize the performance of image processing algorithms, reducing computation time and enhancing efficiency as our goals.

This paper is organized as follows. Section 2 reviews related works on parallelization in general and in image processing, as well as knowledge gap. Existing Section 3 shows the implementation of image processing specifically focusing on the conversion of colorful images to grayscale with and without the race condition, and how a number of cores could affect the performance. Specifically focusing on the conversion of colorful images to grayscale, Section 4 presents the experiment results and then discusses inferences. Finally, the conclusion is outlined in Section 5.

2. Literature Review

The field of parallel image processing has evolved rapidly, due to the increasing demand for efficient and high-speed image analysis in various applications. This section reviews key studies in this field.

The paper by Saxena *et al.* [2] reviews parallel image processing techniques and tools, outlining their benefits and limitations. It discusses the application of various tools like Java, Hadoop, OpenCV, GPUs, and CUDA in image processing. The paper highlights GPU's role in a heterogeneous co-processing model with CPUs, noting its power efficiency but also its high power usage and heat production. CUDA, designed for parallel computing and similar to C language, is limited to NVIDIA GPUs and doesn't fully support the C standard. Java's multithreading improves performance and concurrency but is complex to write. Hadoop scales well but struggles with merging multiple datasets. OpenCL, an open-source API, accelerates parallel programming but is challenging to learn. OpenCV, an Intel-developed image processing library, is open-source but difficult to master. The paper provides an overview of these tools, but it lacks detailed explanations of their limitations and doesn't specify the best application areas for each tool.

Highlighting a different tool, a paper by Slabaugh *et al.* [3] presented image processing applications in parallel using OpenMP (Open Multi-Processing). OpenMP is an Application Programming Interface (API) that enables writing multiprocessing programming. The paper provides an overview introduces OpenMP and demonstrates simple image processing tasks to showcase how easy it is to implement and how effective OpenMP can be. The paper covers various aspects of using OpenMP, such as loop-level parallelism, variable scope, and scheduling. It also explains that static scheduling, which divides work evenly among threads, is the default approach, but dynamic scheduling can be used for unbalanced loops where iterations have varying costs. It mentions two examples, image warping and mathematical binary morphology, which showcase how OpenMP can drastically reduce processing time in image processing applications. All examples used "parallel for" directive to iterate over the pixels of the image to perform specific operation. One limitation of the paper is that it only presents simple image processing operations and does not provide examples of more complex applications that could be built on similar principles. Additionally, the paper does not discuss the potential synchronization issues that can arise when using OpenMP for parallel image processing. Overall, the paper gives a useful introduction about using OpenMP in image processing.

Expanding on the theme of OpenMP, a comprehensive comparison study done by Kendurkar [1] compares between OpenMP and Pymp. Pymp is a Python module that provides support for parallel programming, it is similar to OpenMP but is designed specifically for Python. The authors explain the parallel programming models used in the experiments, including shared memory parallelism and task-based parallelism. Also, they discussed the performance metrics, which include speedup, efficiency, and performance. The results show that Pymp outperforms OpenMP in terms of speedup and efficiency. The paper also provides graphical interpretations of the results and discusses the implications of the findings. Finally, it was concluding with a summary of the study and its contributions. The paper suggests that Pymp is a promising parallel programming model for image convolution and that future research could explore extending the study to other parallel programming frameworks using GPU. The authors also suggest expanding the study with images of higher resolution.

Shifting the focus to another aspect, the study titled "Performance Analysis of Image Segmentation Using Parallel Processing" [4] focuses on Fuzzy C Means (FCM) based segmentation, a clustering method using fuzzy logic for handling ambiguity in image groups. The research developed both sequential and parallel simulation models to assess the FCM algorithm, considering factors like time consumption, efficiency, and the Minimum Mean Square Error (MMSE). The findings demonstrate that MATLAB's parallel computing capabilities can significantly accelerate image processing on multi-core platforms without compromising image quality. The paper compares sequential and parallel execution times, MMSE values, and efficiency percentages. It concludes by underscoring the advantages of parallel methods in image segmentation, particularly in improving efficiency and processing time for real-time applications.

Introducing a novel perspective, a study by Iqbal *et al.* [5] addresses the challenge of improving the efficiency of image processing computations. It proposes a method that integrates Digital Signal Processor (DSP) resources, an overlap segment technique, and parallel processing to boost the efficiency of image processing computations. DSPs are specialized microprocessors for efficiently processing digital signals in real-time. The study's approach divides image frames into sections to solve filtering issues and employs parallel processing for simultaneous computations. It aims to enhance accuracy and speed in image processing, particularly for real-time or high-performance tasks. The research explores strategies like parallel computing and increasing processor clock frequency using VLSI technology, addressing power and thermal issues. The study assesses filter types, segment sizes, overlap factors, and Mean Squared Error (MSE), along with Peak Signal-to-Noise Ratio (PSNR) for evaluating image quality. It concludes that computation time can be significantly reduced with these techniques, highlighting the benefits of parallel processing and overlap segment methods in digital image processing.

In conclusion, this literature review encapsulates the significant advancements in parallel image processing, highlighting diverse methodologies and tools from Saxena *et al.*'s broad overview of parallel processing technologies to the detailed analyses of OpenMP and Pymp's applications in image processing by Slabaugh *et al.* and Kendurkar. The exploration of Fuzzy C Means for image segmentation and Iqbal *et al.*'s innovative use of DSP resources and overlap segment techniques further emphasize the field's dynamic nature. Collectively, these studies underline the growing efficiency and sophistication in image processing techniques, pointing to a future where parallel processing plays a pivotal role in tackling more complex and real-time image analysis challenges.

3. Grayscale Conversion

One of the useful uses of image processing in many disciplines is the conversion of color images to grayscale images. When publishing a picture in publication organizations, color is more expensive than grayscale. For low-cost edition books, color photos have therefore been transformed to grayscale images in order to lower the cost of printing. Similar to how normal individuals see color pictures, color-deficient viewers need high-quality grayscale images in order to understand the content. Similarly, different applications for image processing need to converse to reduce computational requirements, focus on specific features like texture and shape, and prepare images for a variety of advanced processing tasks.

Further understanding of the color image is necessary in order to convert it to a grayscale image. The mix of Red, Green, and Blue (RGB) colors makes up each pixel color in a picture. The RGB color values are represented in three dimensions XYZ, given by the hue, chroma, and brightness characteristics. A color image's quality is determined by how many bits the digital device can handle to represent a given color.

The basic color image is represented by 8 bits, the high color image is represented using 16 bits, the true color image is represented by 24 bits, and the deep color image is represented by 32 bits. The number of bits decides the maximum number of different colors supported by the digital device. If each Red, Green, and Blue occupies 8 bits then the combination of RGB occupies 24 bits and supports 16,777,216 different colors. The 24-bit represents the color of a pixel in the color image. The grayscale image is represented by luminance using an 8-bit value. The luminance of a pixel value of a grayscale image ranges from 0 to 255. The conversion of a color image into a grayscale image is converting the RGB values (24-bit) into grayscale values (8-bit) [6].

Implementing the Grayscale Conversion

Our code performs an image processing task known as a grayscale conversion and uses OpenMP to parallelize the conversion process, distributing the work among multiple threads to enhance performance.

The code is inspired by and modified in order to implement OpenMP parallelization.

First is the image reading, the code opens an image file and reads its header to obtain essential information about the image, such as its width, height, and color depth.

Then, the memory is allocated to hold pixel data for the image, specifically for a buffer based on the image's stride. The stride represents the number of bytes in a row of pixels in the image, accounting for alignment and padding.

The parallel grayscale conversion section as illustrated in **Figure 1**.

- For each row of pixels in the image, it reads a row of pixel data into the buffer.
- For each pixel in the row, it extracts the red, green, and blue color components of the pixel.

Then, calculating the grayscale value of the pixel using a weighted sum formula which is:

gray = 0.11 * *blue* + 0.59 * *green* + 0.3 * *red* [6].

Subsequently, assigning this calculated grayscale value to the red, green, and blue components of the pixel effectively converts it to grayscale. And we accumulate the grayscale value for computing the average later.

After processing each row of pixels, it will write the modified pixel data (grayscale) back to the output file. Once all rows have been processed, the allocated memory is freed, and the input and output files are closed. Finally, the code computes the average grayscale value by dividing the total accumulated grayscale values by the total number of pixels in the image.

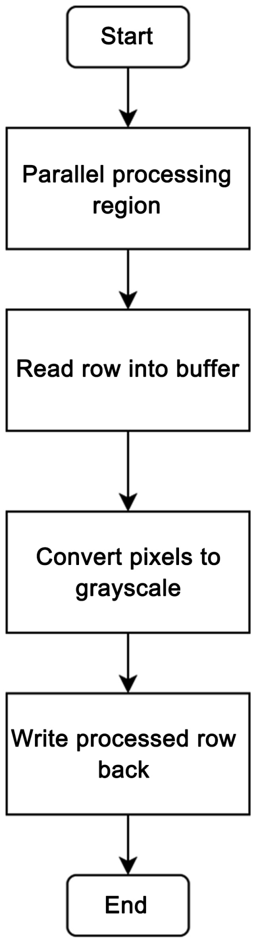

Figure 1. Flowchart of parallel grayscale conversion process.

This algorithm iterates through each pixel of the image, converts it to grayscale using specific weighted values for each color component, and writes the modified data to a new file, resulting in a grayscale representation of the original image.

4. Results and Discussions

Experimental Results

In this section, we will discuss the experimental results based on the implementation of algorithm execution time in C. They cover four scales of different image sizes ranging from 100 × 100 to 1000 × 1000. And experimental results based on number of cores from 1 to 8.

Table 1 shows the execution time of OpenMP parallel and sequential based on the size of image and with the same number of core, so we realize that when we use OpenMP, we had speedup the execution time. However, after calculating the speedup in **Table 2**, we notice the speedup was increasing when the size of image increased. When the size of image was 100 × 100 the speedup was 1.1623 and when we reached to 1000 × 1000 size of image it gave us the highest speedup with 3.1810. Therefore, the size of the image increase (**Figure 2**), more parallelization is obtained. Since the data in each thread is greater, then we save more time than parallelizing a smaller one.

In **Table 3**, we examined the performance of the provided code, and we calculated the speedup and the efficiency in a variety of thread numbers, from one to eight threads. The time is calculated before the parallel region and stopped after the parallel block. Each thread configuration aimed to reduce execution time by leveraging parallel processing. The single-threaded scenario, where no benefit of parallelization, had the slowest execution time and speedup the efficiency is set as the baseline due to sequential execution. We saw an observable improvement

Table 1. Size of image and speedup table.

Size of image	Speedup
100 × 100	1.1623
250 × 250	1.522077
500 × 500	2.23088
1000 × 1000	3.1810

Table 2. C sequential vs OpenMP execution time table.

Size of image	C sequential execution time	OpenMP parallel execution time
100 × 100	0.001919	0.001651
250 × 250	0.009376	0.006160
500 × 500	0.034543	0.015484
1000 × 1000	0.136276	0.04284

Table 3. Speedup and efficiency for number of threads table.

Number of threads	Speedup	Efficiency
1	1	1
2	1.4018	0.700835
4	1.779160	0.4447390
8	2.3331547	0.2916108

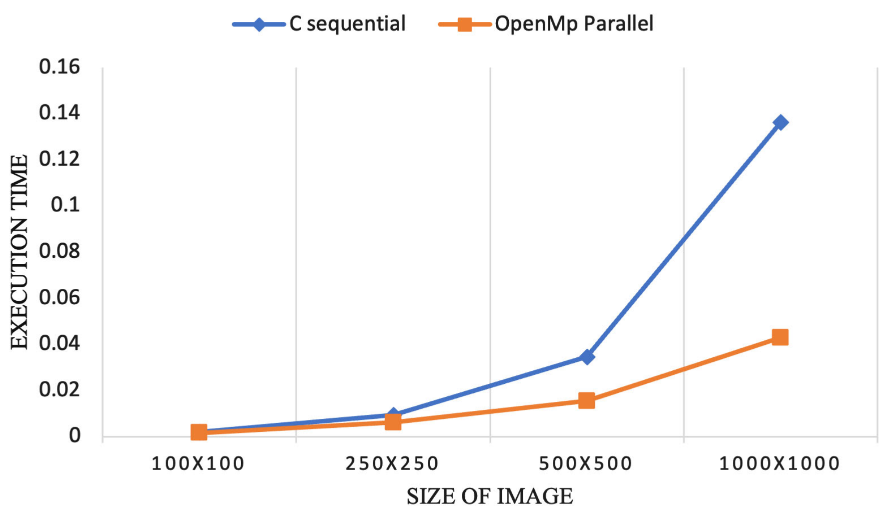

Figure 2. C sequential vs OpenMP parallel execution time in term of image size.

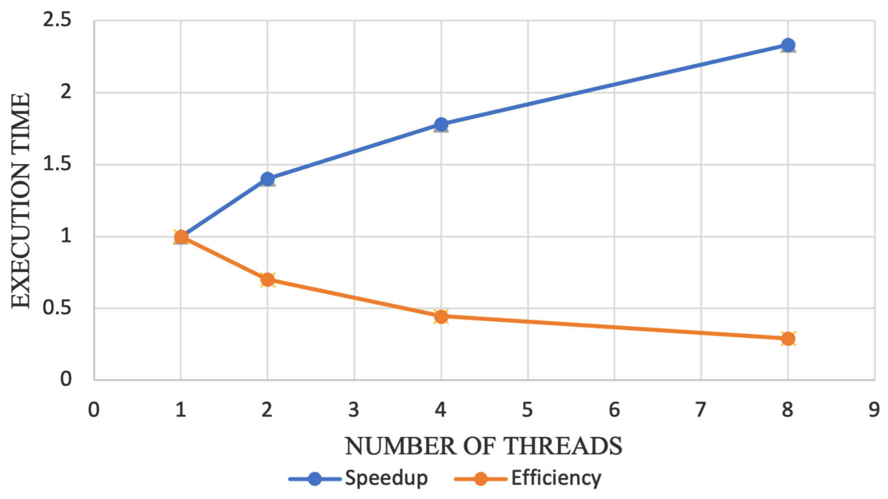

Figure 3. Speedup and efficiency based on number of cores.

in execution time and speedup as we moved to two threads and introduced minimal parallelization and the efficiency decreases to 0.700835, indicating an additional work or inefficiency introduced in the parallelization process, though this improvement was not perfect because of inherent overhead and thread dependencies, especially with smaller image sizes. Four threads introduced moderate parallelization, further dividing the image into segments for simultaneous processing and improving execution time and increased the speedup, particularly for large images but the drops to 0.4447390. Finally, the fastest execution time and the highest speedup were achieved by using eight threads to maximize parallelization because we utilize all cores that we have. We noticed the efficiency decreases in cores 4 and 8, indicating potential issues like communication issues or uneven workload distribution, indicating that adding more cores may not yield the expected speedup as is shown in **Figure 3**.

5. Conclusion

In summary, using OpenMP for parallel image processing, especially for grayscale conversion, offers substantial performance improvements. As OpenMP's capabilities enable concurrent processing of multiple cores, it significantly reduces execution times by distributing tasks across multiple cores. According to the presented speedup values for various image sizes, the parallel approach proves especially useful when working with large images. However, as the table illustrates, parallelization efficiency may decrease with an increase in number of cores. This highlights the importance of carefully optimizing the parallelization strategy, considering factors such as load balancing and minimizing communication overhead. Despite these challenges, the overall scalability and efficiency achieved through parallel image processing underscore OpenMP's effectiveness in accelerating image manipulation tasks, with the potential for even greater gains through meticulous tuning and optimization. This demonstrates the significance of carefully balancing load and reducing communication overhead while optimizing the parallelization strategy. Even with these difficulties, OpenMP is a powerful tool for speeding up image manipulation tasks due to its overall scalability and efficiency through parallel image processing. With careful tuning and optimization, however, even bigger gains may be possible.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Kendurkar, A. (2021) A Comparative Analysis of Parallelisation Using OpenMP and Pymp for Image Convolution. *International Research Journal of Engineering and Technology*, **8**, 54-64. https://www.irjet.net/

[2] Saxena, S., Sharma, S. and Sharma, N. (2016) Parallel Image Processing Techniques, Benefits and Limitations. *Research Journal of Applied Sciences*, *Engineering and Technology*, **12**, 223-238. https://doi.org/10.19026/rjaset.12.2324

[3] Slabaugh, G., Boyes, R. and Yang, X. (2010) Multicore Image Processing with OpenMP [Applications Corner]. *IEEE Signal Processing Magazine*, **27**, 134-138. https://doi.org/10.1109/MSP.2009.935452

[4] Gupta, S. and Rahman, M. (2015) Performance Analysis of Image Segmentation Using Parallel Processing. *International Journal of Innovative Research in Computer Science & Technology* (*IJIRCST*), **3**, 57-63.

[5] Iqbal, M. and Raghuwanshi, S. (2013) Analysis of Digital Image Processing with Parallel and Overlap Segment Technique. *International Journal of Engineering Research and Technology* (*IJERT*), **2**, 2116-2121. https://www.ijert.org/

[6] Saravanan, C. (2010) Color Image to Grayscale Image Conversion. 2010 *Second International Conference on Computer Engineering and Applications*, Bali, 19-21 March 2010, 196-199. https://doi.org/10.1109/ICCEA.2010.192