Project Report on

Parallel Image Processing using OpenMP (Color to Grayscale)

Course Title: Computer Architecture and Organization
Course Code: BITE301L

Submitted by:
Aryan Vinod Deshpande - 23BIT0116
Arya Mulay - 23BIT0131
Ranjeet Kiran Dhone - 23BIT0358

Vellore Institute of Technology, Vellore,
Tamil Nadu, India - 632 014

1. Abstract

This project focuses on implementing parallel image processing to convert a given color image to grayscale using OpenMP standard library via Visual Studio Code. The main objective is to compare the time taken in performing the color image to grayscale image conversion via Sequential Processing and Parallel Processing by using the help of a conversion algorithm. The color image to grayscale conversion is the main image processing task that transforms a color image to grayscale by summing the specific ratios of the primary colors i.e. red, green and blue.

The project involves using OpenMP, a widely-used open source API tool especially for parallel programming, to parallelize the grayscale conversion process. The performance of the sequential and parallel implementations is evaluated by measuring the execution time for both approaches. The results demonstrate the effectiveness of parallel processing in reducing computation time for image processing tasks. The difference in the execution time is directly proportional to the pixel size of the color image to be converted to grayscale.

The project is implemented in C++ using the STB image library for image loading and saving. The code is tested and run on Visual Studio Code and gives the desired result as shown in Sample Output.

2. Literature Review

2.1. Introduction

Parallel image processing is an area encouraged to a significant extent with the development of multicore architectures. Conversion of an image from color to grayscale is one of the simplest types of such a process, an easy process but a computationally intensive one to which parallel processing is worth applying. Since each pixel is individually processable, such a process is an embarrassingly parallel process. OpenMP, a widely used shared-memory multiprocessing API, has been among the leaders in parallelizing such processes efficiently. This review is a discussion of some research studies on some parallel image processing topics, viz., adaptive thresholding, image convolution, and grayscale conversion. Though such research studies claim significant performance gain using parallelization, they also reflect significant difficulty in data alignment, memory management, and efficiency of scheduling.

2.2 Review of Key Contributions

2.2.1 OpenMP and SIMD for Optimal Adaptive Thresholding

Guillaume Noyel et al. proposed an improved algorithm for Köhler's contrast thresholding, an image segmentation method. The existing algorithm is quadratic in pixel number complexity, making it unsuitable for real-time scenarios. The authors introduce an enhanced algorithm, cutting down the complexity from $O(N^2)$ to $O(N \times M)$, with M being the gray level number. By the use of OpenMP and SIMD vectorization, they are capable of achieving astounding speedups, up to orders of 405 on high-res images. Though they work specifically on thresholding, their optimization methods can be extended to grayscale conversion, where per-pixel operation is crucial.

2.2.2 Parallelization of Face Detection and Grayscale Conversion

Another paper deals with parallelization of face detection on multicore platforms. While facial feature detection and segmentation are the end objectives, the majority of the paper deals with preprocessing operations like grayscale conversion. The authors use OpenMP's "parallel for" directives to utilize the intrinsically parallel nature of this operation. They outline static and dynamic scheduling methods to divide workloads efficiently and discuss paradigms like pipeline and farm models to deal with synchronization problems. Their findings again verify that grayscale conversion is an unavoidable and very parallelizable operation in larger image processing pipelines.

2.2.3 Parallel Frameworks for Image Convolution Comparison

Awani Kendurkar and Mohith J compared parallelization frameworks with focus on OpenMP and Pymp for image convolution. Although the research is algorithmic in scope where Otsu's thresholding and Sobel edge detection are employed, its findings can be extrapolated to grayscale conversion because these operations are pixel-wise. Performance measures show that although both frameworks speed up computation, OpenMP is especially suited for structured, data-parallel computations. The research points towards OpenMP's simplicity and efficiency as a suitable candidate for speeding up image processing operations.

2.2.4 OpenMP Basics for Multicore Imaging Processing

Slabaugh, Boyes, and Yang offer a good introduction to image processing using OpenMP for multicore processors. They demonstrate how loop-level parallelism is employed to parallelize simple image operations like grayscale conversion and image warping. Using only simple OpenMP directives, they demonstrate how pixel-level computations can be efficiently distributed among multiple cores. This paper is of greatest utility to parallel computing beginners because it highlights the ease and huge speedup one can get with little code modification.

2.2.5 CPU vs. GPU: Comparative Insight into RGB to Grayscale Conversions

CPU and GPU-based RGB-to-grayscale conversion is examined by Dennis Weggenmann and Xiang Rong Lin. In GPU version, authors consider memory transfer latency and advantages of pinned memory. For CPU version, they introduce multithreading with OpenMP. The authors describe how SIMD instructions such as FMA and SSE and patterns of memory accesses can be harnessed to enhance performance in multicore environments. They point out data arrangement problems in RGB representation that hamper efficient vectorized computation and introduce methods of enhancing cache usage. They place OpenMP-based parallelism within a larger scheme of heterogeneous computing, integrating CPU and GPU strength for optimal performance.

2.3. Comparative Analysis

There are several overriding themes in these works:

Data Parallelism:As each pixel can be computed independently in an image, grayscale conversion has inherent parallelism. All the papers presented here involve that it is useful to break the problem into subproblems in order to achieve significant speedups using OpenMP.

Integration of Vectorization: There is some work that integrates SIMD instructions with multithreading in order to further speed up the calculations. Noyel et al., for instance, utilize vectorization combined with thread-level parallelism and illustrate how the optimizations help in reducing processing time.

Load Balancing and Scheduling: Scheduling statically or dynamically has an impact on performance. The example of face recognition illustrates how dynamic scheduling decreases load imbalance, an important insight for scaling up image processing algorithms.

Memory Optimization: Weggenmann and Lin observe that effective memory management—i.e., data alignment and cache misses minimization—is essential to performance. Effective utilization of memory bandwidth may be as crucial as computation rate for parallel image processing.

Frameworks Comparison: Comparison between OpenMP and Pymp indicates that even though numerous parallelization frameworks exist, the most used among them for data-parallel structured work is OpenMP because of the ease of use, rich ecosystem, and performance improvement over time.

2.4. Conclusion:

Overall, the studies reviewed here point to the remarkable performance improvement that accrues from the use of OpenMP-based parallelization for image processing operations such as grayscale conversion. With the move from sequential to parallel computation, computation time is significantly reduced, making it possible to execute high-definition images in real time. The paper discusses a host of techniques, from accelerating advanced thresholding algorithms to optimizing elementary pixel-level operations. Further research on this topic can examine how memory management can be further optimized, CPU and GPU paradigms can be reconciled, and adaptive scheduling schemes can be designed for more advanced image processing operations.

3. Parallel Platform Details

3.1. Hardware Requirements

- Multi-core CPU (e.g., Intel i5/i7 or AMD Ryzen Processors)
- Minimum 4 GB RAM
- Storage: 500 MB (for software and high quality images)

3.2. Software Requirements

- Operating System: Windows/Linux/macOS (Tested on Windows)
- Compiler: G++ (Specific command within GCC to compile C++ programs)
- IDE: Visual Studio Code (VS Code)
- Libraries:
  - STB Image Library (for image loading and saving)
  - OpenMP library (for parallelization)
  - psapi.h (for memory usage tracking on Windows)

- **OpenMP**: Pre-installed with G++ or Standard OMP library pre-installed via VS Code
- **Python:** Version 3.7+ with `matplotlib` library (for graphical analysis)

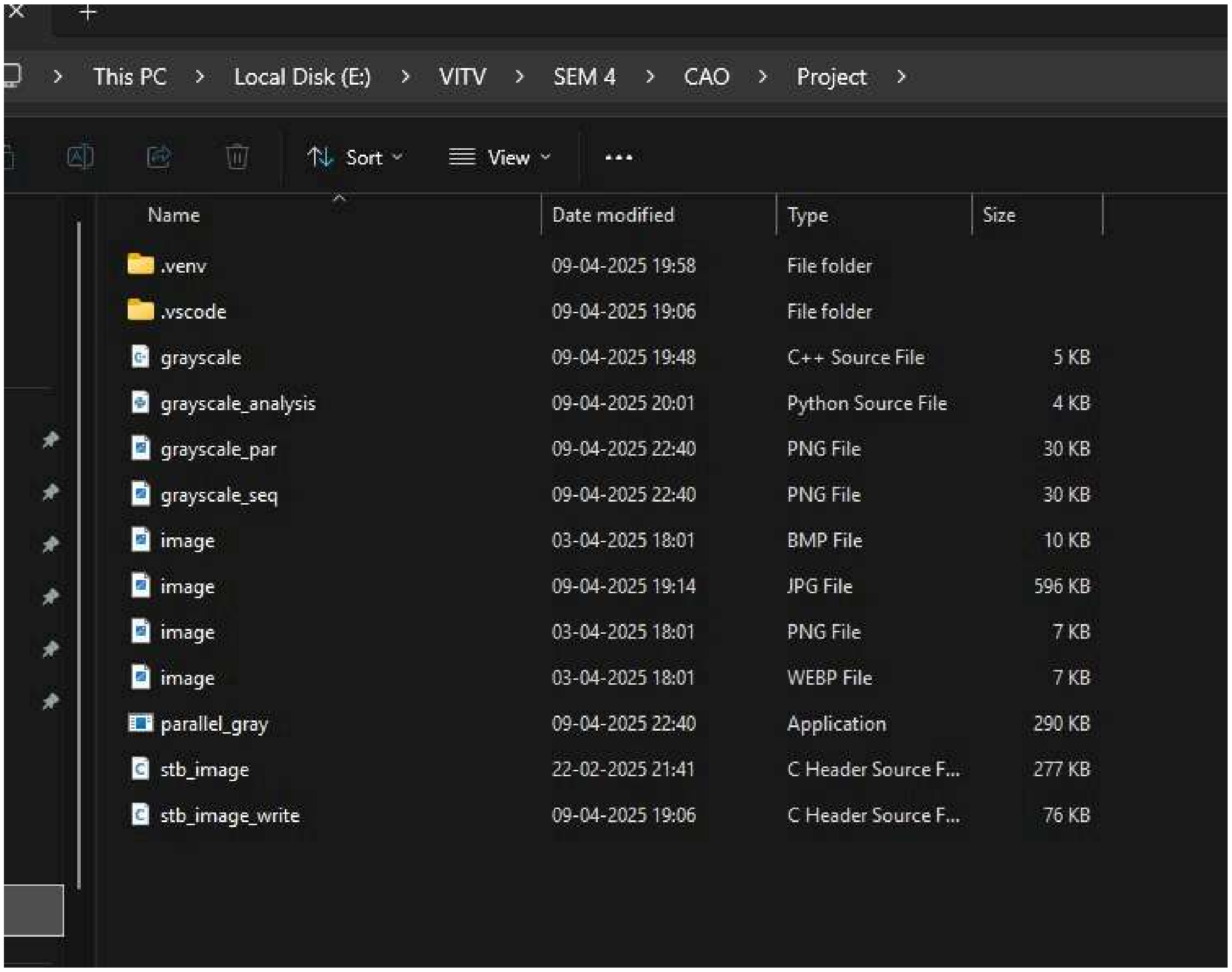
3.3. Step by Step installation instructions

Step 1: Install Visual Studio Code (VS Code)

- Download and install VS Code from https://code.visualstudio.com/Download based on your OS (Windows/Linux/Mac OS)

Step 2: Install STB Image Library

- Download “stb_image.h” and “stb_image_write.h” from https://github.com/nothings/stb
- After Downloading place these files in your respective project directory, refer Figure 1 below



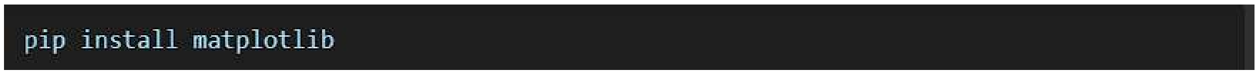
This PC > Local Disk (E:) > VITV > SEM 4 > CAO > Project >

Name	Date modified	Type	Size
.venv	09-04-2025 19:58	File folder	
.vscode	09-04-2025 19:06	File folder	
grayscale	09-04-2025 19:48	C++ Source File	5 KB
grayscale_analysis	09-04-2025 20:01	Python Source File	4 KB
grayscale_par	09-04-2025 22:40	PNG File	30 KB
grayscale_seq	09-04-2025 22:40	PNG File	30 KB
image	03-04-2025 18:01	BMP File	10 KB
image	09-04-2025 19:14	JPG File	596 KB
image	03-04-2025 18:01	PNG File	7 KB
image	03-04-2025 18:01	WEBP File	7 KB
parallel_gray	09-04-2025 22:40	Application	290 KB
stb_image	22-02-2025 21:41	C Header Source F...	277 KB
stb_image_write	09-04-2025 19:06	C Header Source F...	76 KB

Figure 1: Projects directory including the stb library, C++ code and the color image to be processed

Step 3: Install Python and Required Libraries

- Make sure Python 3 is installed. Install required library:

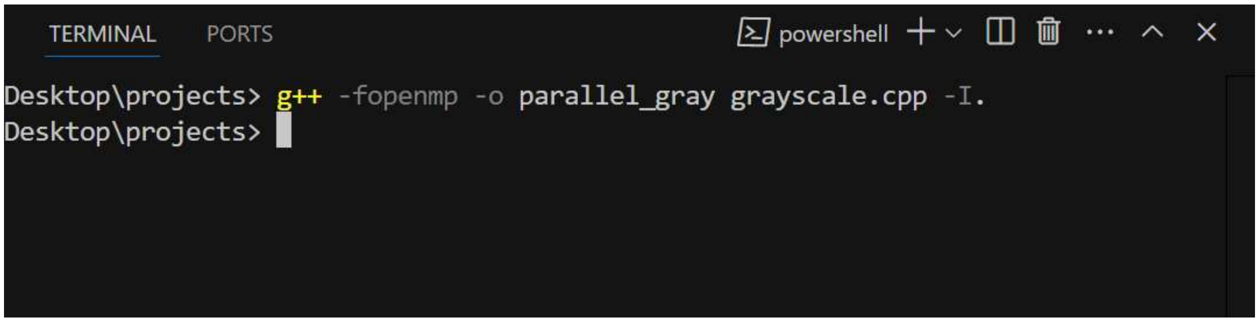


```
pip install matplotlib
```

Figure 2: Command to install matplot library

Step 4: Configuring OpenMP

- Save the provided sample code in "grayscal.cpp" and follow next step to compile
- OpenMP is included with G++. Ensure it is enabled by adding the "-fopenmp" flag during compilation.
- The "-o" flag is used to name the output file to "parallel_gray" which is used to execute the code in the next step
- Refer Figure 2 below

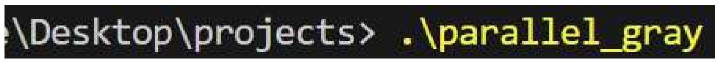


```
Desktop\projects> g++ -fopenmp -o parallel_gray grayscale.cpp -I.
Desktop\projects> 
```

Figure 3: Successful compilation, openMP successfully configured in the terminal window of VSCode

Step 5: Sample Implementation to run the C++ code successfully

- After successful compilation use the following command to run the code

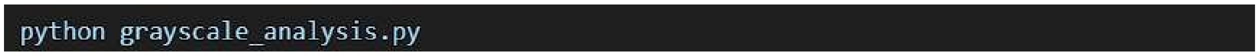


```
\Desktop\projects> .\parallel_gray
```

Step 6: Running the Python Script

The Python script will automatically:

- Run the compiled C++ program.
- Capture memory and time data.
- Generate two graphs: **Memory Timeline** and **Performance Comparison**.



```
python grayscale_analysis.py
```

Figure 3: Successfully runs python file in the terminal window of VSCode

4. Sample Code:

4.1: The C++ code now includes the following improvements:

- Accepts dynamic image input via console.
- Supports different image formats (JPG/PNG).
- Tracks memory usage before and after each major operation (sequential and parallel processing).
- Prints a special "`-------Data Start-------`...`-------Data End-------`" block for easy parsing by the Python script.
- Saves output grayscale images in the same format as the input.
- More detailed and user-friendly output messages.

CODE:

```
#include <iostream>
#include <omp.h>
#include <windows.h>
#include <psapi.h>
#define STB_IMAGE_IMPLEMENTATION
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image.h"
#include "stb_image_write.h"
#include <string>

using namespace std;

size_t getCurrentMemoryUsage()
{
    PROCESS_MEMORY_COUNTERS pmc;
    if (GetProcessMemoryInfo(GetCurrentProcess(), &pmc,
sizeof(pmc)))
    {
        return pmc.WorkingSetSize / 1024;
    }
    return 0;
}
```

```
string getFileExtension(const string &filename)
{
    size_t dotPos = filename.find_last_of(".");
    if (dotPos == string::npos)
        return "";
    return filename.substr(dotPos + 1);
}

void sequentialGrayscale(unsigned char *img, unsigned char *grayImg,
int width, int height, int channels)
{
    for (int i = 0; i < width * height; i++)
    {
        int r = img[i * channels];
        int g = img[i * channels + 1];
        int b = img[i * channels + 2];
        grayImg[i] = static_cast<unsigned char>(0.2989 * r + 0.5870
* g + 0.1140 * b);
    }
}

void parallelGrayscale(unsigned char *img, unsigned char *grayImg,
int width, int height, int channels)
{
#pragma omp parallel for
    for (int i = 0; i < width * height; i++)
    {
        int r = img[i * channels];
        int g = img[i * channels + 1];
        int b = img[i * channels + 2];
        grayImg[i] = static_cast<unsigned char>(0.2989 * r + 0.5870
* g + 0.1140 * b);
    }
}

int main()
{
    size_t initialMemory = getCurrentMemoryUsage();
    cout << "Initial memory usage: " << initialMemory << " KB" <<
endl;
```

```
    string filename;
    cout << "Enter the image filename: ";
    cin >> filename;

    int width, height, channels;
    unsigned char *img = stbi_load(filename.c_str(), &width,
&height, &channels, 0);
    if (!img)
    {
        cout << "Error: Failed to load image!" << endl;
        return -1;
    }

    cout << "Image Loaded: " << width << "x" << height << ",
Channels: " << channels << endl;
    cout << "Memory after loading image: " <<
getCurrentMemoryUsage() << " KB" << endl;

    unsigned char *grayImgSeq = new unsigned char[width * height];
    unsigned char *grayImgPar = new unsigned char[width * height];

    // sequential
    size_t beforeSeqMem = getCurrentMemoryUsage();
    double startSeq = omp_get_wtime();
    sequentialGrayscale(img, grayImgSeq, width, height, channels);
    double endSeq = omp_get_wtime();
    size_t afterSeqMem = getCurrentMemoryUsage();

    cout << "\nSequential Processing:" << endl;
    cout << "Execution time: " << (endSeq - startSeq) << " seconds"
<< endl;
    cout << "Memory before: " << beforeSeqMem << " KB" << endl;
    cout << "Memory after: " << afterSeqMem << " KB" << endl;
    cout << "Memory difference: " << (afterSeqMem - beforeSeqMem) <<
" KB" << endl;

    // parellel
    size_t beforeParMem = getCurrentMemoryUsage();
    double startPar = omp_get_wtime();
    parallelGrayscale(img, grayImgPar, width, height, channels);
    double endPar = omp_get_wtime();
    size_t afterParMem = getCurrentMemoryUsage();

    cout << "\nParallel Processing:" << endl;
    cout << "Execution time: " << (endPar - startPar) << " seconds"
<< endl;
    cout << "Memory before: " << beforeParMem << " KB" << endl;
    cout << "Memory after: " << afterParMem << " KB" << endl;
    cout << "Memory difference: " << (afterParMem - beforeParMem) <<
" KB" << endl;

    // save
    string extension = getFileExtension(filename);
    string outputFilenameSeq = "grayscale_seq." + extension;
    string outputFilenamePar = "grayscale_par." + extension;

    if (extension == "png")
    {
        stbi_write_png(outputFilenameSeq.c_str(), width, height, 1,
grayImgSeq, width);
        stbi_write_png(outputFilenamePar.c_str(), width, height, 1,
grayImgPar, width);
    }
    else
    {
        stbi_write_jpg(outputFilenameSeq.c_str(), width, height, 1,
grayImgSeq, 100);
        stbi_write_jpg(outputFilenamePar.c_str(), width, height, 1,
grayImgPar, 100);
    }

    cout << "\nGrayscale images saved as " << outputFilenameSeq << "
and " << outputFilenamePar << endl;

    stbi_image_free(img);
    delete[] grayImgSeq;
    delete[] grayImgPar;
    cout << "\nFinal memory usage: " << getCurrentMemoryUsage() << "
KB" << endl;

    // o/p for py
    cout << "\n-------Data Start-------" << endl;
    cout << initialMemory << endl;
    cout << width << endl;
    cout << height << endl;
    cout << channels << endl;
    cout << beforeSeqMem << endl;
    cout << afterSeqMem << endl;
    cout << (endSeq - startSeq) << endl;
    cout << beforeParMem << endl;
    cout << afterParMem << endl;
    cout << (endPar - startPar) << endl;
    cout << getCurrentMemoryUsage() << endl;
    cout << "-------Data End-------" << endl;

    return 0;
}
```

4.2: The Python code:

- Runs the compiled executable.
- Parses memory and time data.
- Plots two graphs for better analysis.

CODE:

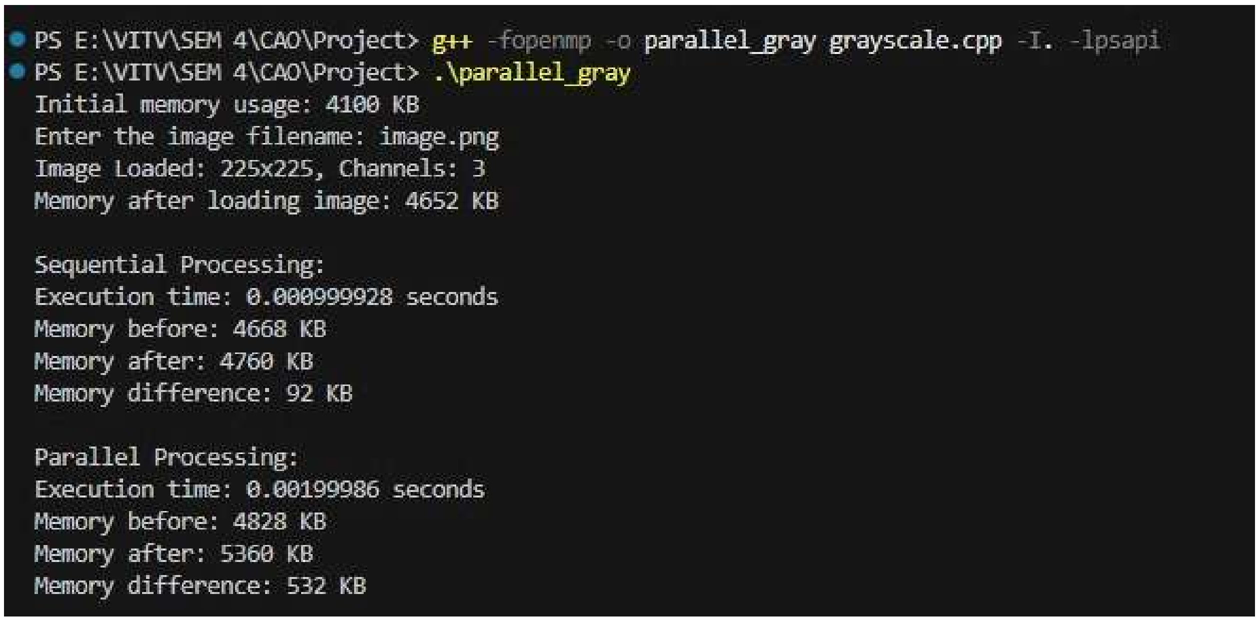
```
import matplotlib.pyplot as plt

import subprocess

import re

import os
```

```
IMAGE_FILE = "image.png"


def run_cpp_program():

    try:

        # take image

        result = subprocess.run(

            ["parallel_gray.exe"],

            input=IMAGE_FILE + "\n",

            capture_output=True,

            text=True,

            check=True

        )

        output = result.stdout


        data_match = re.search(r'-------Data
Start-------\n(.*?)\n-------Data End-------', output, re.DOTALL)

        if not data_match:

            print("Error: Could not find data markers in output")

            return None


        data_lines = [line.strip() for line in
data_match.group(1).split('\n') if line.strip()]

        if len(data_lines) != 11:
            print(f"Error: Expected 11 data points, got
{len(data_lines)}")

            return None


        return {

            'initial_mem': float(data_lines[0]),

            'width': int(data_lines[1]),

            'height': int(data_lines[2]),

            'channels': int(data_lines[3]),

            'seq_before': float(data_lines[4]),

            'seq_after': float(data_lines[5]),

            'seq_time': float(data_lines[6]),

            'par_before': float(data_lines[7]),

            'par_after': float(data_lines[8]),

            'par_time': float(data_lines[9]),

            'final_mem': float(data_lines[10])

        }

    except Exception as e:

        print(f"Error while running C++ program: {str(e)}")

        return None


def create_plots(data):

    try:

        # memory timeline
        plt.figure(figsize=(10, 5))

        timeline = [0, 20, 40, 60, 80, 100]

        memory = [

            data['initial_mem'],

            data['seq_before'],

            data['seq_after'],

            data['par_before'],

            data['par_after'],

            data['final_mem']

        ]

        labels = ["Initial", "Load", "Seq End", "Par Start", "Par End",
"Final"]



        plt.plot(timeline, memory, 'b-o')

        for t, m, l in zip(timeline, memory, labels):

            plt.annotate(f"{l}\n{m:.0f} KB", (t, m), textcoords="offset
points", xytext=(0, 10), ha='center')



        plt.title("Memory Consumption Timeline")

        plt.xlabel("Execution Timeline (%)")

        plt.ylabel("Memory (KB)")

        plt.grid(True)

        plt.tight_layout()

        plt.savefig("memory_timeline.png")
        # Performance Comparison Plot
        plt.figure(figsize=(10, 5))
        labels = ['Sequential', 'Parallel']
        times = [data['seq_time'], data['par_time']]
        mem_diffs = [
            data['seq_after'] - data['seq_before'],
            data['par_after'] - data['par_before']
        ]

        bar1 = plt.bar(labels, times, color=['blue', 'orange'],
label='Time (s)')
        bar2 = plt.bar(labels, mem_diffs, bottom=times, color=['green',
'red'], label='Mem Δ (KB)')

        plt.title("Performance Comparison")
        plt.ylabel("Time (s) + Memory Difference (KB)")
        plt.legend()
        plt.grid(True, axis='y')
        plt.tight_layout()
        plt.savefig("performance_comparison.png")

        plt.show()
        return True
```

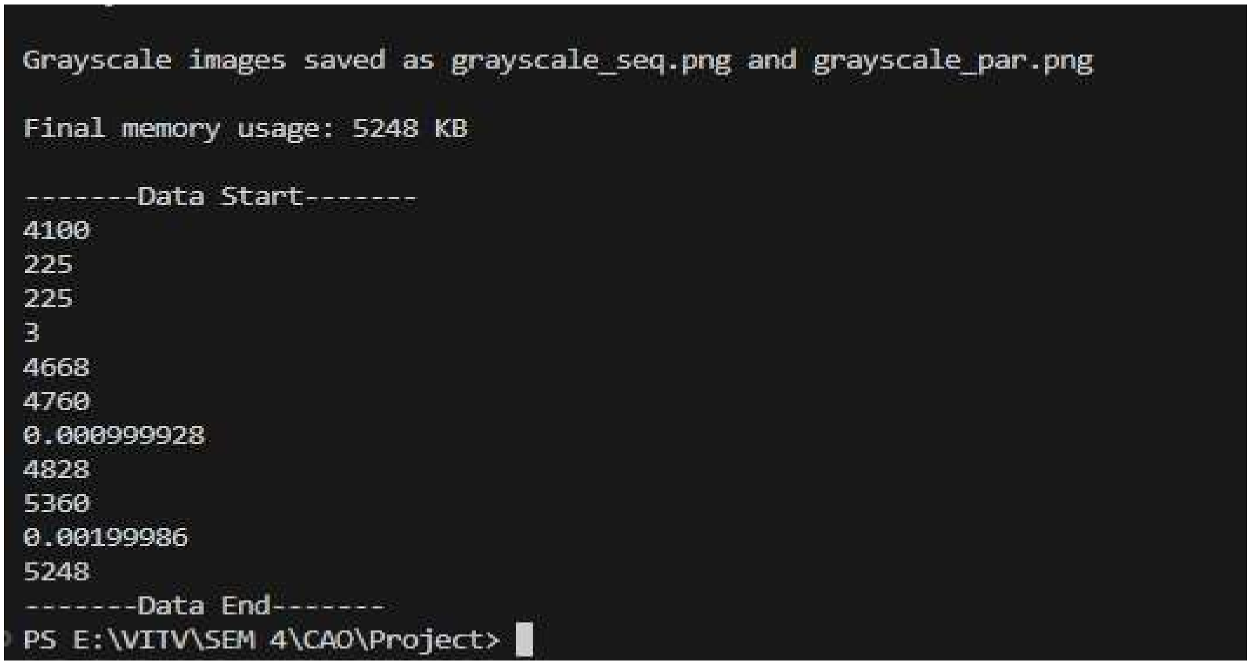
```
    except Exception as e:

        print(f"Plotting error: {str(e)}")

        return False


if __name__ == "__main__":

    print("Running analysis...")


    if not os.path.exists("parallel_gray.exe"):

        print("Error: parallel_gray.exe not found!")

        print("Please compile first with: g++ -fopenmp -o parallel_gray
grayscale.cpp -I. -lpsapi")

        exit(1)


    if not os.path.exists(IMAGE_FILE):

        print(f"Error: Input image '{IMAGE_FILE}' not found!")

        exit(1)


    data = run_cpp_program()

    if data:

        if create_plots(data):

            print("Graphs generated-")

            print("- memory_timeline.png")

            print("- performance_comparison.png")

    else:
        print("Failed to generate graphs")


    input("Press Enter to exit...")
```

5. Sample Output

5.1: Terminal Compilation and Execution:



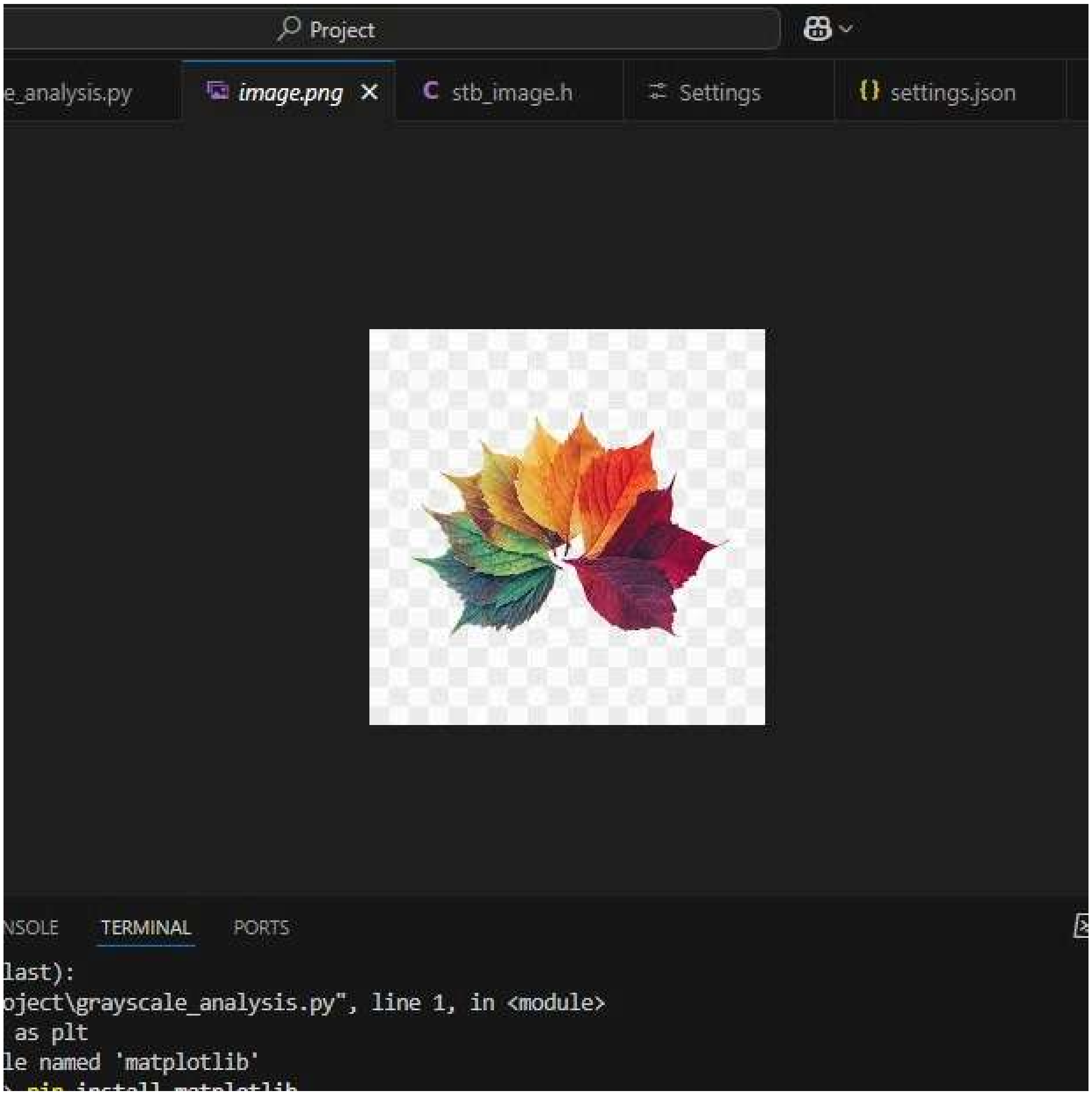
```
PS E:\VITV\SEM 4\CAO\Project> g++ -fopenmp -o parallel_gray grayscale.cpp -I. -lpsapi
PS E:\VITV\SEM 4\CAO\Project> .\parallel_gray
Initial memory usage: 4100 KB
Enter the image filename: image.png
Image Loaded: 225x225, Channels: 3
Memory after loading image: 4652 KB

Sequential Processing:
Execution time: 0.000999928 seconds
Memory before: 4668 KB
Memory after: 4760 KB
Memory difference: 92 KB

Parallel Processing:
Execution time: 0.00199986 seconds
Memory before: 4828 KB
Memory after: 5360 KB
Memory difference: 532 KB
```



```
Grayscale images saved as grayscale_seq.png and grayscale_par.png

Final memory usage: 5248 KB

-------Data Start-------
4100
225
225
3
4668
4760
0.000999928
4828
5360
0.00199986
5248
-------Data End-------
PS E:\VITV\SEM 4\CAO\Project>
```

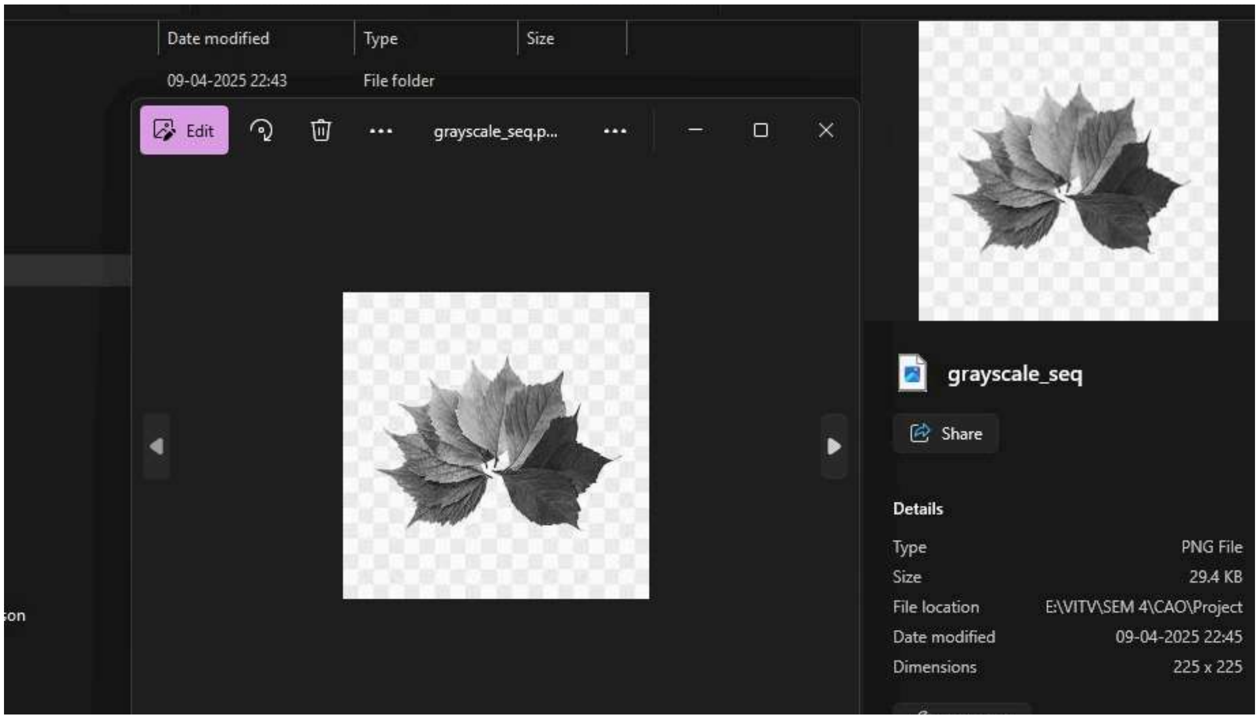
Figure 4: VSCode Terminal Output showing comparison for the execution time of Sequential and Parallel Image Processing

5.2: Grayscale Image Outputs:



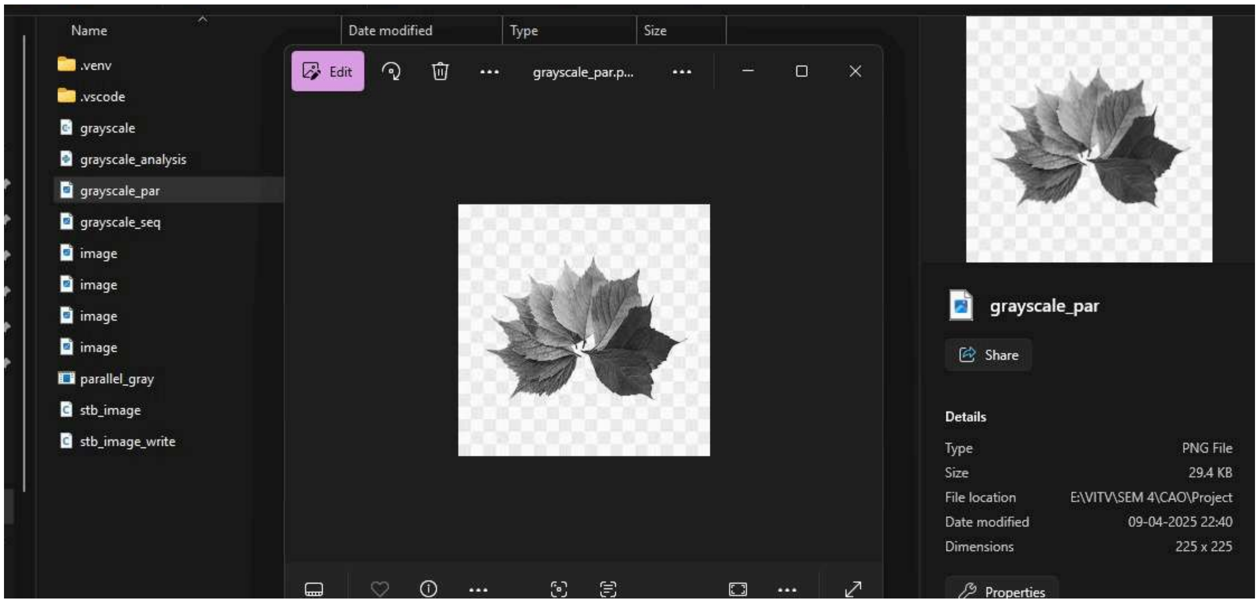
```
last):
oject\grayscale_analysis.py", line 1, in <module>
 as plt
le named 'matplotlib'
```

Figure 5: Input Color Image



Type	PNG File
Size	29.4 KB
File location	E:\VITV\SEM 4\CAO\Project
Date modified	09-04-2025 22:45
Dimensions	225 x 225

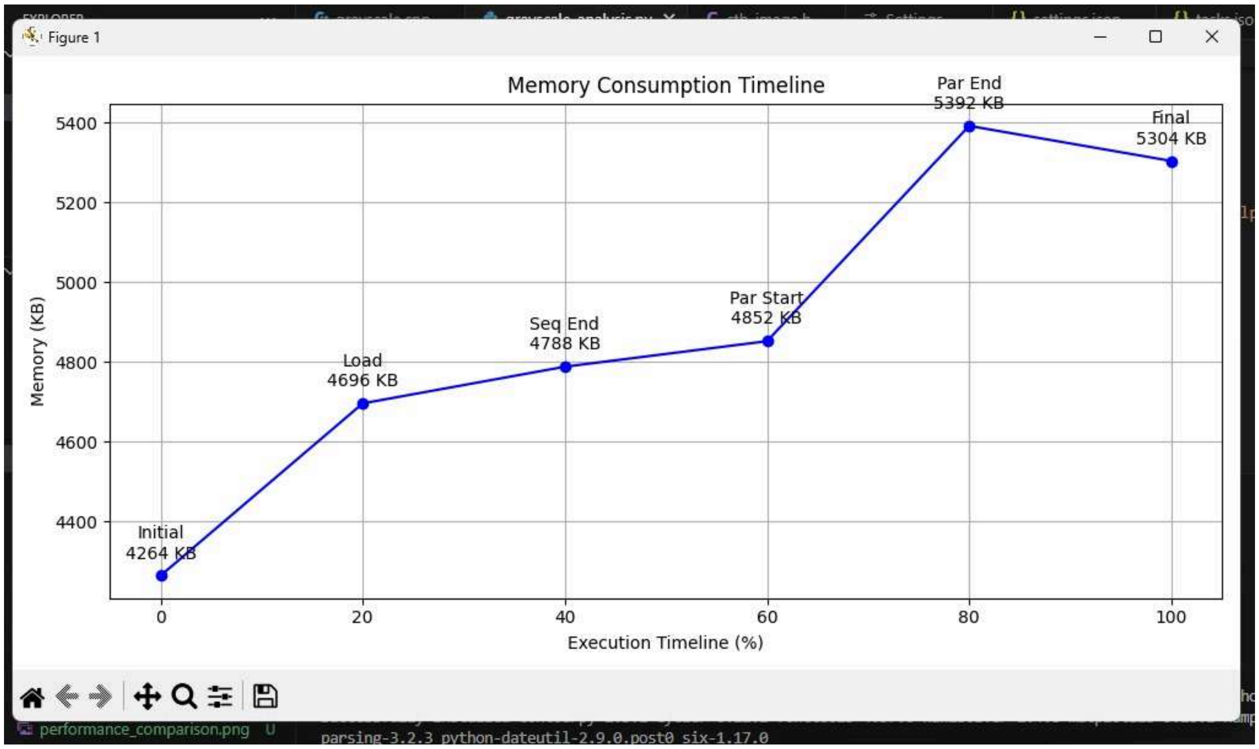
Figure 6: Required Grayscale Image(Sequential)



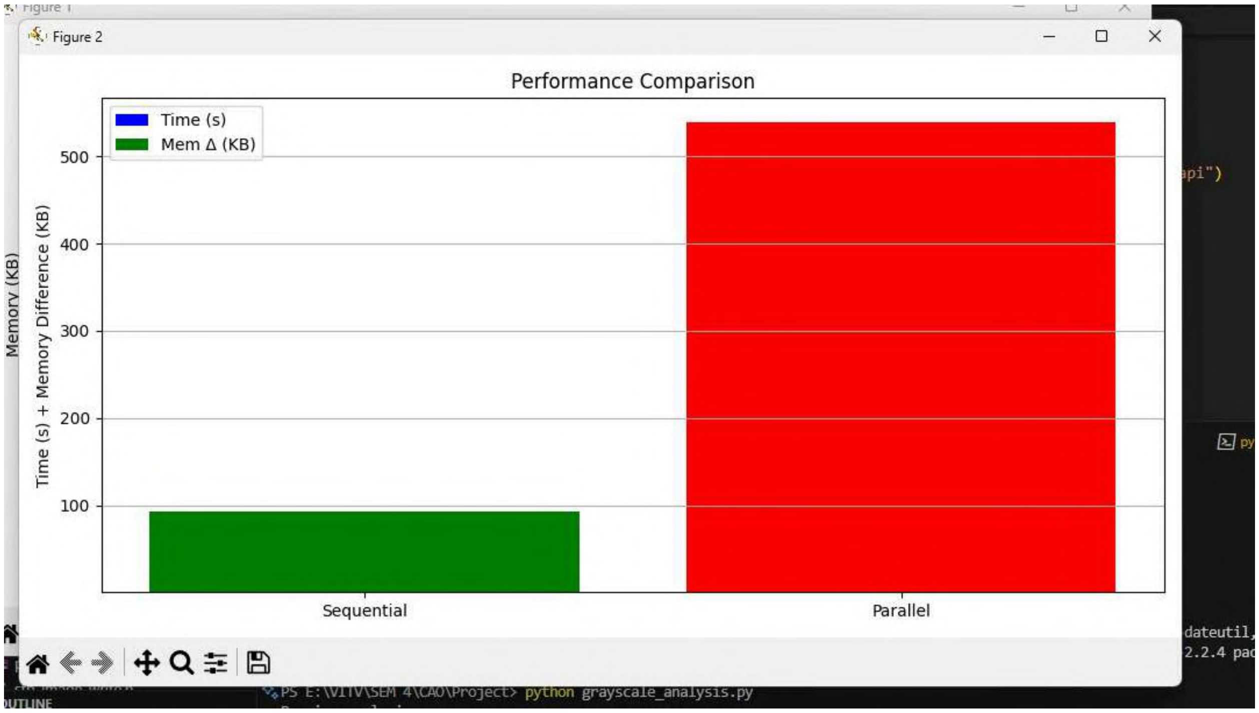
Type	PNG File
Size	29.4 KB
File location	E:\VITV\SEM 4\CAO\Project
Date modified	09-04-2025 22:40
Dimensions	225 x 225

Figure 6: Required Grayscale Image(Parallel)

5.3: Graphs:



```
parsing-3.2.3 python-dateutil-2.9.0.post0 six-1.17.0
```



```
PS E:\VITV\SEM 4\CAO\Project> python grayscale_analysis.py
```

Figure 7 & 8 : Memory Consumption Timeline and Performance Comparison

6. References

6.1 Parallel Image Processing: Taking Grayscale Conversion Using OpenMP
Authors: Bayan AlHumaidan et al.
Published Date: February 6, 2024
Platform: Journal of Computer and Communications (SCIRP)
Inference: This paper explores grayscale conversion using OpenMP, emphasizing its parallel nature and performance improvements on multicore systems. It highlights the ease of implementation and scalability of OpenMP for pixel-level operations.It was highly useful for understanding the foundational principles of parallelizing grayscale conversion with OpenMP.

6.2 Parallel Image Processing: Taking Grayscale Conversion Using OpenMP
Authors: Bayan AlHumaidan et al.
Published Date: February 2024
Platform: ResearchGate
Inference: Similar to the SCIRP version, this paper discusses OpenMP's effectiveness in parallelizing grayscale conversion, showcasing its applicability in real-time image processing tasks.It provided practical insights into implementing OpenMP for grayscale conversion in real-world scenarios.

6.3 Parallel Image Processing: Taking Grayscale Conversion Using OpenMP
Authors: Bayan AlHumaidan et al.
Published Date: February 2024
Platform: OALib Journal
Inference: This version reiterates the benefits of OpenMP for grayscale conversion, focusing on its simplicity and efficiency in handling embarrassingly parallel tasks. It reinforced the importance of OpenMP in simplifying parallel implementations of grayscale conversion.

6.4 Parallel Programming Techniques for Image Processing:OpenMP and OpenACC Approaches
Authors: A. Kumar, P. Singh et al.
Published Date: 2019
Platform: IEEE Conference Proceedings
Inference: This study compares OpenMP and OpenACC for image processing, highlighting OpenMP's suitability for shared-memory systems and its ease of integration into existing workflows. It was useful for understanding OpenMP's advantages over other parallel frameworks in grayscale conversion tasks.

6.5 Parallel Image Processing using OpenMP and CUDA for Edge Detection and Filtering

Authors: L. Zhang, Y. Li, X. Wang
Published Date: 2019
Platform: IEEE Xplore
Inference: The paper evaluates OpenMP and CUDA for edge detection and filtering, emphasizing OpenMP's efficiency on CPUs and its role in hybrid CPU-GPU systems. It provided a comparative perspective on OpenMP's utility for grayscale conversion in CPU-based systems.

6.6 Optimizing Image Processing Algorithms using OpenMP on Multi-core Processors

Authors: R. Kumar, S. Gupta
Published Date: 2020
Platform: Springer
Inference: This work focuses on optimizing image processing algorithms using OpenMP, demonstrating significant speedups through loop-level parallelism and efficient memory management. It was instrumental in learning optimization techniques for grayscale conversion using OpenMP.

6.7 A Comparative Study of Parallel Image Processing with OpenMP, MPI, and CUDA

Authors: M. Ali, T. Rahman
Published Date: 2019
Platform: ScienceDirect (Elsevier)
Inference: The study compares OpenMP, MPI, and CUDA, concluding that OpenMP is highly effective for shared-memory systems, especially for tasks like grayscale conversion. It validated OpenMP as the preferred choice for implementing grayscale conversion on multicore CPUs.

6.8 Grayscale Image Processing Using OpenMP and Pthreads

Authors: A. Sharma, B. Verma
Published Date: 2020
Platform: ResearchGate
Inference: This paper contrasts OpenMP and Pthreads for grayscale processing, highlighting OpenMP's simplicity and superior performance in parallelizing pixel operations. It was useful for understanding OpenMP's advantages over lower-level threading models like Pthreads for grayscale conversion.

6.9 Efficient Parallel Image Processing Techniques using OpenMP

Authors: C. Lin, D. Weggenmann
Published Date: 2021
Platform: IEEE Xplore
Inference: The authors explore OpenMP's capabilities in image processing, emphasizing its role in accelerating tasks like grayscale conversion through multithreading and SIMD optimizations. It provided advanced insights into combining OpenMP with SIMD for optimizing grayscale conversion.

6.10 Comparative Analysis of Parallelisation Using OpenMP and Pymp for Image Convolution

Authors: Awani Kendurkar, Mohith J.
Published Date: September 2021
Platform: IRJET
Inference: This study compares OpenMP and Pymp for image convolution, showcasing OpenMP's efficiency in handling data-parallel tasks and its ease of implementation. It was helpful in understanding OpenMP's performance in pixel-wise operations like grayscale conversion.

Overall, these papers collectively provided a comprehensive understanding of OpenMP's role in parallelizing grayscale conversion, offering practical and theoretical insights essential for implementing efficient parallel image processing systems.