HIGH PERFORMANCE COMPUTING
Speeding up Image Processing using OpenMP

Muhammad Sabihul Hasan
Dipartimento di Scienze e Tecnologie
muhammadsabihul.hasan001@studenti.uniparthenope.it
Matricola: 0120000308

***Abstract*—Image processing is a computationally intensive task where parallelization can provide significant benefits. This project explores the use of OpenMP, a parallel programming API for shared memory systems, to accelerate two common image processing operations - blurring and edge detection. The implemented solution reads a PPM image, applies the selected filters, and uses OpenMP directives to effectively parallelize the pixel-based computations. A performance evaluation compares the execution times of the serial and parallel implementations, with results showing that OpenMP can provide significant speedups. Additionally, the number of threads used is dynamically selected to optimize processing. The results highlight the effectiveness of OpenMP in improving computational efficiency, making it a valuable tool for real-time image processing applications.**

***Index Terms*—parallel programming, image processing, OpenMP**

I. Introduction - OpenMP

Historically, one of the main challenges of parallel computing has been the lack of a widely supported and easy-to-implement parallel programming model. As a result, multiple vendors have provided different models, often with varying degrees of sophistication and portability. Subsequently, software programmers have found it difficult to adapt applications to take advantage of advances in multicore hardware.

OpenMP aims to fill this gap by providing an industry-standard parallel programming API for shared-memory multiprocessors, including multicore processors. An Independent Vendor, The OpenMP Architecture Review Board (ARB), which includes most major computer manufacturers, oversees the OpenMP standard and approves new versions of the specification. Currently, most modern Fortran and C/C++ compilers and many operating systems, including Microsoft Windows, Linux, and Apple Macintosh OS X, support OpenMP. OpenMP version 1.0 was released in 1997. The latest version, 3.0, was released in 2008. For a list of standards, the full specification, and compiler reference documentation, see the official OpenMP website [1].

It is important to note that OpenMP is certainly not the only way to implement parallelism on multicore systems. There are other implementation models, such as CILK, Pthreads, and MPI [2] [3], which may be good choices, depending on the hardware, application, and programmer preference. In our experience, OpenMP has the advantages of being extremely easy to learn and implement, powerful, and well adapted to modern processor architectures.

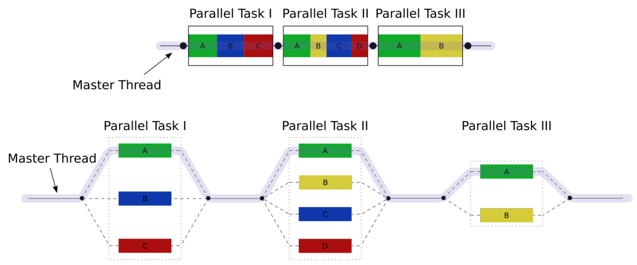

Fig. 1. Illustration of the fork–join model of parallel programming

II. Image Processing

Image processing is a fundamental area of computer science and engineering, with applications in medical imaging, remote sensing, computer vision, surveillance, and multimedia. It involves techniques for processing and analyzing digital images to extract meaningful information or improve visual quality. However, processing high-resolution images is computationally expensive as a result of the huge amount of data involved.

This project focuses on two fundamental image processing techniques: **Blurring** and **Edge Detection**, which are commonly used in computer vision and image enhancement. The following sections describe their theoretical background and computational implementation.

A. Blurring

Blurring is a technique that reduces image noise and detail by averaging pixel values within a small neighborhood. The process is often applied as a preprocessing step for tasks such as object detection and segmentation.

1) Theory: Blurring is accomplished by convolving an image with a smoothing kernel (a small weight matrix). One of the simplest forms is a box filter (average filter), where the new value of each pixel is the average of the surrounding pixels. Common blur kernels are:

$$K = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

Each pixel in the image is replaced by the weighted sum of itself and its neighbors. Larger kernels (e.g. 5×5 or Gaussian kernels) produce stronger blurring effects.

2) Implementation in parallel computing: Blurring involves reading the values of neighboring pixels, averaging them, and storing the new values. Since each pixel operation is independent (except for boundary conditions), it is suitable for parallelization. Using OpenMP, we can achieve cross-pixel parallelization by splitting the loop into multiple threads.

B. Edge Detection

Edge detection is a basic technique to identify the boundaries of objects in an image by detecting sudden changes in intensity. It is widely used in feature extraction, computer vision, and image segmentation.

1) Theory: Edge detection involves computing the intensity gradient of an image. A widely used method is the **Sobel** operator, which applies two convolution kernels:

Sobel X Kernel (Horizontal Edges)

$$G_x = \begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix}$$

Sobel Y Kernel (Vertical Edges)

$$G_y = \begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{bmatrix}$$

The final edge size for each pixel is calculated using the gradient size formula:

$$G = \sqrt{G_x^2 + G_y^2}$$

This results in pixels with sudden changes in intensity having a high response, thus highlighting the edges of the image.

2) Implementation in Parallel Computing: Since the edge gradients for each pixel are calculated independently of each other, OpenMP parallelization is suitable for this operation. The loop that iterates over the image pixels can be split into multiple threads, allowing for faster execution while maintaining accuracy.

III. Implementation

The implementation of the image processing filter is parallelized using C programming language and OpenMP. The program reads a PPM (P6) image, applies the selected filter (blur or edge detection), and writes the processed image to an output file. The following steps were followed during implementation:

1) Image Processing:

- The program uses **fscanf()** and **fread()** to read the **PPM (P6)** image format.
- The image is stored as a one-dimensional matrix in row-major order, where each pixel is represented by three consecutive bytes (R, G, B).
- After processing, the modified image is written back using **fwrite()**.

Listing 1. Functions for Reading and Writting the image

```
// Function to read a PPM image
Image readPPM(const char *filename) {
    FILE *fp = fopen(filename, "rb");
    if (!fp) {
        printf("Error:␣Cannot␣open␣file␣%
            s\n", filename);
        exit(1);
    }

    char format[3];
    int w, h, maxColor;
    fscanf(fp, "%s\n%d␣%d\n%d\n", format,
         &w, &h, &maxColor);

    if (strcmp(format, "P6") != 0 ||
       maxColor != MAX_COLOR) {
        printf("Error:␣Unsupported␣image␣
            format.\n");
        fclose(fp);
        exit(1);
    }

    Image img;
    img.width = w;
    img.height = h;
    img.data = (unsigned char *)malloc(3
        * w * h);
    fread(img.data, 3, w * h, fp);
    fclose(fp);

    return img;
}

// Function to write a PPM image
void writePPM(const char *filename, Image
     img) {
    FILE *fp = fopen(filename, "wb");
    if (!fp) {
        printf("Error:␣Cannot␣write␣to␣
            file␣%s\n", filename);
        exit(1);
    }

    fprintf(fp, "P6\n%d␣%d\n%d\n", img.
        width, img.height, MAX_COLOR);
    fwrite(img.data, 3, img.width * img.
        height, fp);
    fclose(fp);
}
```

2) Applying the filter using OpenMP:

- The computationally intensive loop that iterates over the pixels is parallelized using OpenMP directives.
- The **#pragma omp parallel for collapse(2) schedule(dynamic)** directive is used to split the rows and columns of the image into threads.
- The "if" clause allows for execution with or without OpenMP for comparison.
- Used **private()** for local variables in nested loops to avoid conflicts.

Listing 2. Functions for applying Blur and Edge Detection Filter

```
// Apply Blur Filter
void applyBlur(Image *img, int use_openmp
    ) {
    int w = img->width, h = img->height;
    unsigned char *temp = (unsigned char
        *)malloc(3 * w * h);

    #pragma omp parallel for collapse(2)
        schedule(dynamic) if(use_openmp)
    for (int y = 1; y < h - 1; y++) {
        for (int x = 1; x < w - 1; x++) {
            int sumR = 0, sumG = 0, sumB
                = 0;
            int count = 0;

            for (int dy = -1; dy <= 1; dy
                ++) {
                for (int dx = -1; dx <=
                    1; dx++) {
                    int idx = 3 * ((y +
                        dy) * w + (x + dx)
                        );
                    sumR += img->data[idx
                        ];
                    sumG += img->data[idx
                         + 1];
                    sumB += img->data[idx
                         + 2];
                    count++;
                }
            }

            int idx = 3 * (y * w + x);
            temp[idx] = sumR / count;
            temp[idx + 1] = sumG / count;
            temp[idx + 2] = sumB / count;
        }
    }

    memcpy(img->data, temp, 3 * w * h);
    free(temp);
}

// Apply Edge Detection Filter
void applyEdgeDetection(Image *img, int
    use_openmp) {
    int w = img->width, h = img->height;
    unsigned char *temp = (unsigned char
        *)malloc(3 * w * h);

    int Gx[3][3] = {{-1, 0, 1}, {-2, 0,
        2}, {-1, 0, 1}};
    int Gy[3][3] = {{-1, -2, -1}, {0, 0,
        0}, {1, 2, 1}};

    #pragma omp parallel for collapse(2)
        schedule(dynamic) if(use_openmp)
    for (int y = 1; y < h - 1; y++) {
        for (int x = 1; x < w - 1; x++) {
            int sumRx = 0, sumGx = 0,
                sumBx = 0;
            int sumRy = 0, sumGy = 0,
                sumBy = 0;

            for (int dy = -1; dy <= 1; dy
                ++) {
                for (int dx = -1; dx <=
                    1; dx++) {
                    int idx = 3 * ((y +
                        dy) * w + (x + dx)
                        );
                    int weightX = Gx[dy +
                         1][dx + 1];
                    int weightY = Gy[dy +
                         1][dx + 1];

                    sumRx += img->data[
                        idx] * weightX;
                    sumGx += img->data[
                        idx + 1] * weightX
                        ;
                    sumBx += img->data[
                        idx + 2] * weightX
                        ;

                    sumRy += img->data[
                                idx] * weightY;
                            sumGy += img->data[
                                idx + 1] * weightY
                                ;
                            sumBy += img->data[
                                idx + 2] * weightY
                                ;
                        }
                    }

                    int idx = 3 * (y * w + x);
                    temp[idx] = fmin(sqrt(sumRx *
                         sumRx + sumRy * sumRy),
                        255);
                    temp[idx + 1] = fmin(sqrt(
                        sumGx * sumGx + sumGy *
                        sumGy), 255);
                    temp[idx + 2] = fmin(sqrt(
                        sumBx * sumBx + sumBy *
                        sumBy), 255);
            }
        }

        memcpy(img->data, temp, 3 * w * h);
        free(temp);
}
```

3) Measuring execution time:

- Used **omp_get_wtime()** to measure the time required to apply each filter.
- Executed with and without OpenMP to compare performance.
- Printed the number of threads used.

IV. Performance Evaluation

To evaluate performance improvements due to parallelization using OpenMP, we tested the filters on 4 PPM sample images available on **https://filesamples.com/formats/ppm**. The execution time was recorded for serial and parallel executions.

A. Experimental Setup

1) System Specifications::

- Processor : Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz
- RAM : 8.0 GB
- Compiler: nvcc with OpenMP support (nvcc -Xcompiler ”-std=c99 -O3 -fopenmp”)

2) Filters Tested:

- Blur (3×3 Kernel)
- Edge Detection (Sobel Operator)

3) Measurements:

- Execution time without OpenMP (Serial)
- Execution time with OpenMP (Parallel)
- Speedup = Time without OpenMP (serial) / Time with OpenMP (parallel)

B. Results

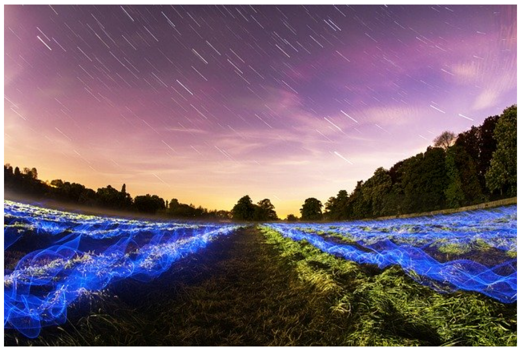

Fig. 2. Original Sample PPM Image

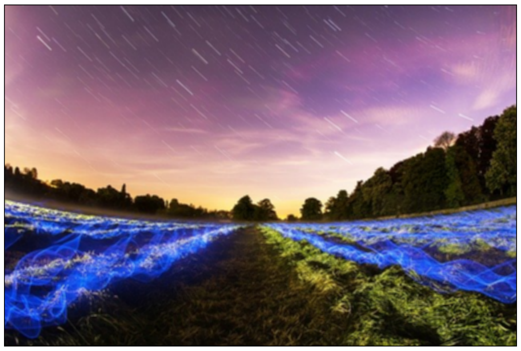

Fig. 3. Image after applying the Blur filter

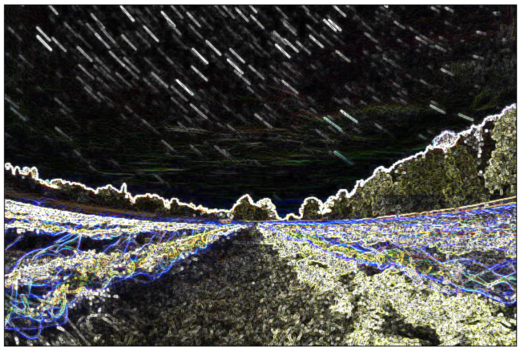

Fig. 4. Image after applying the Edge Detection filter

TABLE I
Table showing times recorded and the calculated speedup

Type of Filter	Size/pixels	Time/secs (without OpenMP)	Time/secs (with OpenMP)	Speedup Factor
Blur	272640	0.037711	0.015451	2.44x
	1091840	0.121991	0.050719	2.41x
	2457600	0.218103	0.104501	2.09x
	17915904	0.986432	0.763705	1.29x
Edge Detection	272640	0.067248	0.018162	3.70x
	1091840	0.195976	0.049349	3.97x
	2457600	0.345286	0.108304	3.19x
	17915904	1.772378	0.785251	2.26x

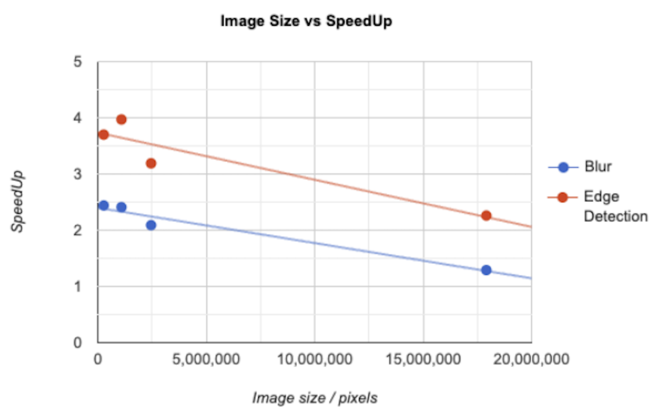

Fig. 5. Graph showing the affects of image size on overall speedup

C. *Observations*

Both filters are more than twice as fast due to OpenMP parallelization. Edge detection is more computationally intensive. It takes longer than blur because it involves more arithmetic operations per pixel. Despite this, OpenMP provides almost the better speedup as blur. 4 threads were chosen as the optimal number of threads since the performance gain must very small above 4 threads, suggesting a bottleneck due to memory access latency. Adding more threads than the number of physical cores (4) incurs hyperthreading overhead.

The **collapse(2)** directive helps distribute workload across both image rows and columns, improving load balancing. Dynamic scheduling **(schedule(dynamic))** prevents threads from being idle by dynamically assigning pixel blocks. Since edge detection involves varying computation per pixel, dynamic scheduling provided better performance than static scheduling. The graph's efficiency curve follows **Amdahl's Law**, where parallelization benefits diminish as some parts of the code remain sequential (I/O, memory copying)

V. Conclusion

This project successfully demonstrated how OpenMP can be used to accelerate image processing tasks, especially blurring and edge detection. By parallelizing computationally intensive loops using OpenMP directives, we achieved significant speedups compared to serial implementations. The results demonstrate that multicore processors can effectively handle large-scale image processing, making OpenMP a valuable tool for real-time applications. Performance analysis shows that:

- Parallel execution significantly reduces processing time, allowing filtering of high-resolution images.
- The edge detection filter takes longer than the blur filter in both serial and parallel modes because it involves more complex calculations for each pixel.
- Dynamic scheduling improves load balancing by ensuring that threads are efficiently utilized, especially in edge detection where different pixel regions have different computational loads.
- Memory bandwidth limitations hinder linear scalability, meaning that speedups above a certain threshold are limited by system constraints such as cache performance and data transfer rates.

Overall, the results confirm that OpenMP provides an effective way to utilize multicore processors for image processing tasks. While OpenMP significantly improves performance, additional optimizations are needed to minimize parallel overhead and optimize memory access patterns.

VI. Future Work

While OpenMP parallelization has provided substantial improvements, there are several areas where further optimization and advancement can be made. Here are some possible future directions:

A. *GPU Acceleration with CUDA*

Even with OpenMP, processors have inherent limitations when it comes to handling massively parallel tasks such as image processing. GPUs (Graphics Processing Units) are designed for highly parallel workloads and can provide 10x to 100x speedups over CPU-based OpenMP implementations. Implementing the same filter with CUDA can further improve efficiency by leveraging thousands of GPU cores.

B. *Optimizing Memory Access Patterns*

OpenMP performance is often limited by memory bandwidth because multiple threads compete for shared memory resources. Tiling techniques (blocking) can improve cache locality, thereby reducing memory access latency. Using SIMD (Single Instruction, Multiple Data) instructions in conjunction with OpenMP can further increase the speed of processing multiple pixels simultaneously.

C. *Try different parallelization strategies*

In addition to OpenMP, other parallel programming models such as MPI (Message Passing Interface) can be leveraged to distribute the workload across multiple machines in a cluster environment. Hybrid parallelization (OpenMP + CUDA) can be implemented to leverage CPUs and GPUs to perform different aspects of the image processing pipeline.

D. *Implement additional filters*

The current implementation focuses on blurring and edge detection, but OpenMP can be applied to more advanced filters such as:

- Gaussian blur (for smooth, low-noise filtering)
- Canny edge detection (for better edge detection performance)
- Laplacian operator (for detecting second-order derivatives of an image)
- Bilateral filtering (removes noise while preserving edges)

E. Real-time processing and multi-threaded pipelines

To process continuous real-time image streams (e.g. video processing), the system can be extended to a multi-threaded pipeline where different stages (reading, processing, and writing) run in parallel. This will allow near-instant processing of high-resolution images and videos.

REFERENCES

[1] http://www.openmp.org.

[2] H. Kim and R. Bond, "Multicore software technologies: A survey," IEEE Signal Processing Magazine, vol. 26, no. 6, pp. 80–89, 2009.

[3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, Parallel Programming in OpenMP. Morgan Kaufmann Publishers, first ed., 2001.

APPENDIX

The entire code and sample image files can be found in the following github repository: **https://github.com/m-sabihul-hasan/Image-Processing-with-OpenMP**