ME 766: High Performance Scientific Computing **July-Nov 2024**

Parallelization of Convolution Operation Using MPI and OpenMP - MPI Hybrid

Afnan Abdul Gafoor *Abhinav P T*

Abstract

Image convolution is a fundamental operation in image processing, used for tasks like blurring, sharpening, and edge detection. This study implements and evaluates Gaussian blur using three approaches: serial, MPI, and hybrid MPI + OpenMP. The objective was to analyze the performance of distributed memory (MPI) and hybrid shared-distributed memory (MPI + OpenMP) models against a serial baseline. Experiments reveal that the hybrid approach outperforms for fewer processes/threads, while MPI excels with higher process counts. Results demonstrate the potential of parallel programming techniques to optimize computationally intensive image processing tasks efficiently.
The codes are available at https://github.com/Afnnnan/ParallelConvolution.

1 Introduction

Image processing plays a pivotal role in a variety of applications such as computer vision, medical imaging, and remote sensing. One of the most fundamental operations in image processing is convolution, which is used for tasks like noise reduction, feature extraction, and blurring. Convolution operations, particularly the Gaussian blur, are essential for enhancing image quality by smoothing pixel values. However, performing convolution on large images can be computationally expensive, resulting in significant processing time and limiting the effectiveness of real-time applications.

To mitigate these challenges, parallel computing techniques are increasingly being employed. By distributing the computational workload, parallelization can significantly accelerate image processing tasks, making them more scalable and efficient. Two primary approaches to parallelization are the Message Passing Interface (MPI), which is suited for distributed memory systems, and OpenMP, which targets shared memory architectures. Both techniques have proven successful in parallelizing various computational tasks, but their effectiveness often depends on the size of the problem and the available computing resources.

While MPI has traditionally been used for high-performance computing on large clusters, OpenMP offers a more straightforward approach for parallelizing code on multi-core systems. Recently, hybrid approaches combining MPI and OpenMP have been explored to leverage the strengths of both models, particularly for large-scale problems that require both distributed and shared memory parallelism. The hybrid model allows for the efficient use of resources across multiple nodes while minimizing communication overhead between them.

In this work, we focus on the Gaussian blur operation as a case study for comparing the performance of three parallelization strategies: serial execution, MPI-based parallelization [1], and a hybrid MPI + OpenMP approach [2]. Our goal is to analyze the execution time, scalability, and efficiency of each strategy, and identify the conditions under which one model outperforms the others. Through detailed experimental analysis, we aim to provide insights into the advantages and trade-offs of hybrid parallelization for image processing tasks.

2 Convolution Algorithm

The convolution algorithm applied in this project is a fundamental image processing technique used to blur or smooth an image. The convolution process involves sliding a kernel (in this case, a Gaussian filter) over the image and performing element-wise multiplication between the kernel and the pixels it covers (Fig. 1). The result is then summed and placed in the output image. The steps of the convolution algorithm are explained below:

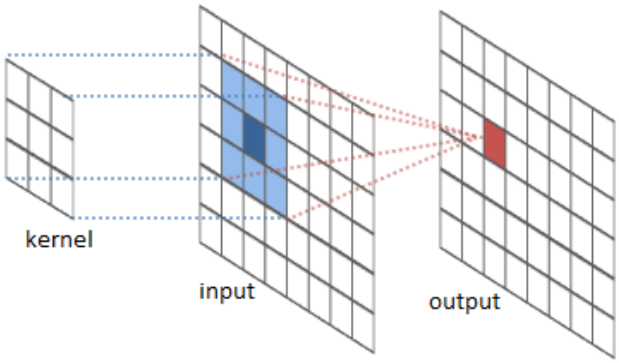

Figure 1: Convolution Process

1. **Input Image and Kernel:** The input image is represented as a 2D array of pixel values. The kernel, a small 2D array (such as 3x3 or 5x5), represents the filter used for convolution. In this project, we use a Gaussian kernel for blurring.
2. **Pixel Iteration:** The algorithm iterates over every pixel in the image, excluding the boundaries where the kernel would extend outside the image. For each pixel, the algorithm multiplies the kernel values with the corresponding pixels from the image.
3. **Weighted Sum Calculation:** For each pixel, the algorithm calculates a weighted sum of the neighboring pixels using the kernel. This sum is the convolution result for the current pixel and is placed in the output image.
4. **Edge Handling:** For edge pixels, where the kernel overlaps outside the image, padding or boundary constraints are used to handle these cases.
5. **Output Image:** After processing all the pixels, the result is stored in a new image, which represents the blurred output of the convolution.

This algorithm, although effective, is computationally intensive, especially for large images, as it requires nested loops to traverse all pixels and perform the convolution. The serial implementation processes each pixel independently, making it a good baseline but inefficient for large-scale applications.

3 Methodology

3.1 Serial Implementation

In this section, the image filtering process and convolution mechanism in the serial version of the program are explained in detail. The task involves applying a Gaussian blur filter to an image, where each pixel is

convolved with a 3x3 filter matrix. Below is a brief overview of how the serial code works:

- **Image Filtering Process:** The image is processed pixel by pixel, and a convolution filter is applied to each pixel. Specifically, a 3x3 Gaussian blur filter is used. The convolution operation calculates a weighted average of each pixel and its neighbors, resulting in a smoothing effect.
- **Convolution Mechanism:** The convolution operation involves a kernel (filter matrix), where each element of the kernel corresponds to a weight. For a 3x3 Gaussian filter, the weights are:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

  Each surrounding pixel value is multiplied by the corresponding kernel weight, and the sum of these values determines the new intensity of the pixel. The kernel is applied to all pixels in the image, ensuring the filtering effect is applied to the entire image.
- **Image Color Handling:** The code supports both grayscale and RGB image processing. For grayscale images, the convolution is applied directly to the pixel intensities. For RGB images, the convolution is applied separately to each of the red, green, and blue channels, ensuring that color information is preserved while the blur effect is applied.

The following code snippet provides a high-level overview of the implementation:

```
// Apply convolution for grayscale or RGB images
void convolute(uint8_t *src, uint8_t *dst, int row_from, int row_to,
               int col_from, int col_to, int width, int height,
               float **h, color_t imageType) {
    int i, j;
    if (imageType == GREY) {
        for (i = row_from; i <= row_to; i++)
            for (j = col_from; j <= col_to; j++)
                convolute_grey(src, dst, i, j, width+2, height, h);
    } else if (imageType == RGB) {
        for (i = row_from; i <= row_to; i++)
            for (j = col_from; j <= col_to; j++)
                convolute_rgb(src, dst, i, j*3, width*3+6, height, h);
    }
}
```

This function `convolute` is responsible for applying the convolution operation to the image, either in grayscale or RGB format, by iterating through the pixels and calling respective functions for each image type (`convolute_grey` or `convolute_rgb`).

3.2 MPI Implementation

In the MPI-based implementation of the parallel image processing task, several key steps are taken to efficiently divide the image and handle communication between different MPI processes. Below is an overview of the major steps involved:

- **Image Division:** The image is divided into rectangular blocks where each MPI process handles a specific region. This division ensures that the image can be processed in parallel across multiple nodes, making it suitable for both grayscale and RGB images.
- **Border Synchronization:** Each process communicates with its neighboring processes to exchange border pixel data. This is crucial for ensuring that the convolution operation, which depends on neighboring pixels, works accurately at the boundaries of the blocks.
- **Non-blocking Communication:** The implementation uses non-blocking MPI communication (e.g., `MPI_Isend` and `MPI_Irecv`) to send and receive the border data asynchronously, allowing computations to proceed without waiting for the communication to complete. This improves the efficiency of the parallelization.
- **Performance Optimization:** Several optimization strategies are used to minimize communication overhead, including intelligent division of the image into blocks, minimizing inter-process communication, and using MPI data types that efficiently handle the data being transferred.

Code Snippet: Below is a key section of the MPI implementation for image reading and communication.

```
MPI_File_open(MPI_COMM_WORLD, image, MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
if (imageType == GREY) {
    for (i = 1 ; i <= rows ; i++) {
        MPI_File_seek(fh, (start_row + i-1) * width + start_col, MPI_SEEK_SET);
        tmpbuf = offset(src, i, 1, cols+2);
        MPI_File_read(fh, tmpbuf, cols, MPI_BYTE, &status);
    }
} else if (imageType == RGB) {
    for (i = 1 ; i <= rows ; i++) {
        MPI_File_seek(fh, 3*(start_row + i-1) * width + 3*start_col, MPI_SEEK_SET);
        tmpbuf = offset(src, i, 3, cols*3+6);
        MPI_File_read(fh, tmpbuf, cols*3, MPI_BYTE, &status);
    }
}
MPI_File_close(&fh);
```

This section of the code demonstrates how image data is read in parallel across different processes. Each process reads its respective block of the image based on its rank and the division parameters. The image data is then processed by applying various filters (e.g., Gaussian blur, edge detection).

Key MPI Functions:

- `MPI_Bcast`: Broadcasts parameters (such as image dimensions and type) to all processes.
- `MPI_Isend` and `MPI_Irecv`: Non-blocking send and receive operations for border pixel data.
- `MPI_File_open`, `MPI_File_read`, `MPI_File_seek`: MPI file I/O operations for parallel reading of image data.

Communication Structure: The image blocks are exchanged between neighboring processes using non-blocking communication. For example, a process may send the northern border of its image block to its northern neighbor while simultaneously receiving data from the southern neighbor. This overlap between computation and communication minimizes idle time and maximizes the parallel efficiency of the program.

In the next step, the actual convolution is applied to the image blocks after border synchronization, which ensures that boundary conditions are correctly handled. This process is repeated for a specified number of loops, allowing multiple iterations of the filter to be applied.

3.3 Hybrid Implementation

In this implementation, MPI and OpenMP are used together to parallelize the image processing task. The image is divided across multiple processes using MPI, and each process parallelizes the convolution operation on its assigned region using OpenMP. Neighboring processes exchange boundary data to ensure correct computation.

```
// Convolution function (parallelized with OpenMP)
void convolute(uint8_t *src, uint8_t *dst, int rows, int cols, float **filter) {
    #pragma omp parallel for
    for (int i = 1; i < rows-1; i++) {
        for (int j = 1; j < cols-1; j++) {
            float sum = 0.0f;
            for (int k = -1; k <= 1; k++) {
                for (int l = -1; l <= 1; l++) {
                    sum += src[(i+k)*cols + (j+l)] * filter[k+1][l+1];
                }
            }
            dst[i*cols + j] = (uint8_t)sum;
        }
    }
}

// Main MPI function
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    // MPI communication setup and region distribution

    // Convolution computation for each process
    convolute(src, dst, rows, cols, filter);
    // MPI boundary data exchange between processes

    MPI_Finalize();
}
```

This code demonstrates how the convolution is parallelized with OpenMP within each process while MPI is used for distributing the work and handling communication between processes.

4 Results

The performance results for the complete implementations (both RGB and grayscale, for 2, 4, 6, and 8 processes) are provided in the appendix. Below are the performance results for the RGB implementation with 2 and 8 processes.

Table 1: Performance Results for 2 Processes (RGB)

N	Serial Time (s)	MPI Time (s)	Hybrid Time (s)
20	4.982	2.378417	0.920490
50	12.431	6.066608	2.273395
100	24.763	11.825304	4.515209
200	49.784	23.723422	9.088242
500	124.601	60.095191	22.488919

Table 2: Performance Results for 8 Processes (RGB)

N	Serial Time (s)	MPI Time (s)	Hybrid Time (s)
20	4.982	0.783944	0.923724
50	12.431	1.968385	2.342930
100	24.763	4.010541	4.636157
200	49.784	7.823434	9.355984
500	124.601	19.721030	23.184868

From the performance data, we can observe the following trends:

- The serial execution times increase significantly with the number of convolutional loops N, as expected.
- MPI and hybrid execution show notable improvements over the serial implementation, with hybrid execution (MPI + OpenMP) outperforming MPI alone for smaller N and fewer processes.
- At higher numbers of processes (e.g., 8 processes), MPI starts to outperform the hybrid model, suggesting that for larger-scale problems, the overhead of managing OpenMP threads in the hybrid model becomes a limiting factor.

From the speedup data (Figure 2), we can observe the following trends:

- For smaller numbers of processes (e.g., 2 processes), the hybrid approach (MPI + OpenMP) outperforms MPI alone, achieving a speedup of approximately 5.5 compared to MPI's speedup of around 2. This is because, for smaller problem sizes, the overhead of managing OpenMP threads is relatively low, and the parallelization within each process provides significant performance gains.
- For larger numbers of processes (e.g., 8 processes), MPI outperforms the hybrid model. MPI achieves a speedup of above 6, while the hybrid model achieves a speedup of about 5. This occurs because, at higher process counts, the communication overhead in the hybrid model increases due to the management of multiple threads, which reduces its scalability. MPI, being a purely distributed model, handles the communication more efficiently at this scale.

5 Hardware and OS Configuration

The experiments were performed on a 2021 MacBook Pro with the following configuration:

- Apple M1 Pro chip with 8-core CPU (6 performance cores and 2 efficiency cores)

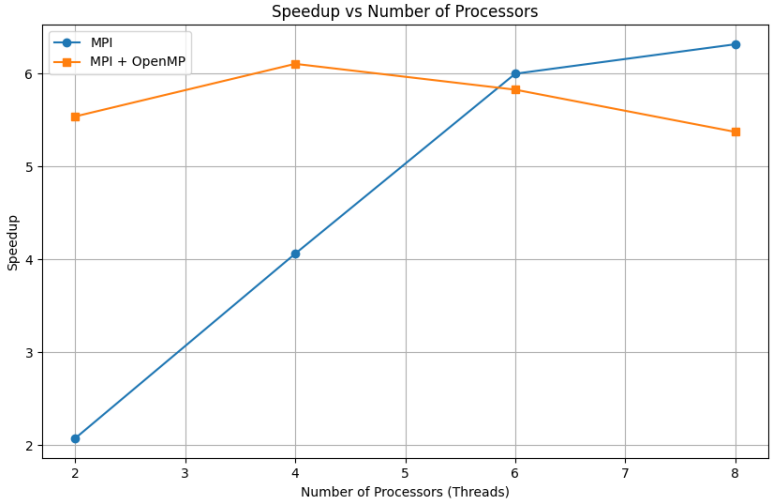

Figure 2: Speedup compared to the serial code for RGB with N = 500

- 16 GB unified memory
- macOS 15.1.1
- gcc 14.2.0
- Open MPI 5.0.5 (mpiexec)

6 Conclusion

In this project, we investigated the performance of hybrid parallelization using MPI and OpenMP for image convolution tasks, comparing it with serial and MPI-only implementations. Our findings revealed several key insights. As the problem size increases, the serial execution time grows significantly, underscoring the necessity of parallelization for handling larger datasets efficiently.

The hybrid model, combining MPI and OpenMP, showed substantial performance improvements over the serial implementation, particularly for smaller problem sizes and fewer processes. However, for larger problem sizes and higher process counts, the MPI-only implementation outperformed the hybrid model. This discrepancy can be attributed to the additional overhead incurred by OpenMP's thread management, which becomes more pronounced when the number of processes increases, thus reducing the performance benefits of parallelism in high-process scenarios.

Speedup results further confirmed this trend, with hybrid parallelism achieving greater speedup for smaller problem sizes, while MPI delivered better performance at larger problem sizes, especially with 8 processes. The hybrid approach demonstrated an initial advantage but was limited by the overhead at higher process numbers.

Overall, this project highlights the advantages and limitations of combining MPI and OpenMP for parallel computing. While hybrid models are effective for small to medium-sized problems, MPI alone may offer better performance for larger datasets and more processes. Future work could explore further op-

timizations to improve the scalability of hybrid models, enabling them to better leverage both MPI and OpenMP in a more balanced way.

References

[1] Jin Lu, Kaisheng Zhang, Ming Chen, and Ke Ma. Implementation of parallel convolution based on mpi. In *Proceedings of 2013 3rd International Conference on Computer Science and Network Technology*, pages 28–31, 2013.

[2] Osvaldo Mangual, Marvi Teixeira, Reynaldo Lopez, and Felix Nevarez. Hybrid mpi-openmp versus mpi implementations: A case study. 06 2014.

[3] Jim Ouris. Parallel convolution. `https://github.com/jimouris/parallel-convolution`, 2021. Accessed: 2024-11-29.

[4] Panagiotis Petropoulakis. Parallel convolution. `https://github.com/PetropoulakisPanagiotis/parallel-convolution`, 2021. Accessed: 2024-11-29.

A Appendix

Here we present the results of the convolution operation applied to both RGB and Grayscale images. The number of iterations, denoted as N, corresponds to the number of times the convolution filter is applied to the image. In this case, $N = 50$, meaning the filter is applied 50 times to the input images.

Figure 3: RGB image after applying Gaussian blur for $N = 50$ iterations.

Figure 4: Grayscale image after applying Gaussian blur for $N = 50$ iterations.

Table 3: Performance Results for Various Processes (RGB)

N	**Serial Time (s)**	**2 Processes**		**4 Processes**		**6 Processes**		**8 Processes**	
		MPI Time (s)	**Hybrid Time (s)**	**MPI Time (s)**	**Hybrid Time (s)**	**MPI Time (s)**	**Hybrid Time (s)**	**MPI Time (s)**	**Hybrid Time (s)**
20	4.982	2.378417	0.920490	1.236581	0.797058	0.845290	0.846247	0.783944	0.923724
50	12.431	6.066608	2.273395	3.055980	2.018847	2.078286	2.121881	1.968385	2.342930
100	24.763	11.825304	4.515209	6.194156	4.002539	4.112811	4.287725	4.010541	4.636157
200	49.784	23.723422	9.088242	12.183043	8.085214	8.148030	8.553585	7.823434	9.355984
500	124.601	60.095191	22.488919	30.673411	20.402480	20.763503	21.375393	19.721030	23.184868

Table 4: Performance Results for Various Processes (Grey)

N	**Serial Time (s)**	**2 Processes**		**4 Processes**		**6 Processes**		**8 Processes**	
		MPI Time (s)	**Hybrid Time (s)**	**MPI Time (s)**	**Hybrid Time (s)**	**MPI Time (s)**	**Hybrid Time (s)**	**MPI Time (s)**	**Hybrid Time (s)**
20	3.034	1.549288	0.564127	0.798311	0.506423	0.534291	0.534828	0.534619	0.615634
50	7.935	3.867279	1.396288	1.992304	1.246592	1.334239	1.329070	1.310875	1.526315
100	16.383	7.727510	2.850884	3.986903	2.469982	2.683687	2.741402	2.607915	3.066253
200	32.291	15.493539	5.617990	7.986798	5.052975	5.319520	5.422499	5.076195	6.218083
500	86.180	38.654856	14.047751	19.915830	12.500185	13.675643	13.515103	12.839587	15.371245