Optimizing Convolution with Naïve, OpenMP, and MPI Approaches

Justin Wenzel

Introduction to Convolution

Definition:

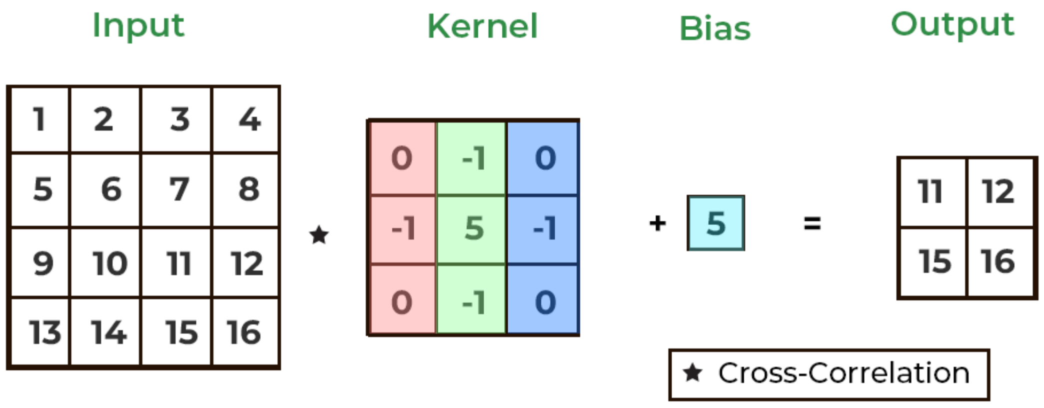
Applies a filter (kernel) to an image to extract features such as edges, patterns, or textures.

Applications:

- Edge detection
- Blurring/Edge Sharpening
- Medical Imaging
- Computer Vision (feature extraction, image classification)

Challenges:

- Convolution becomes computationally expensive for large images
- RGB images add complexity with multi-channel processing
- Large images often exceed memory/cache limits

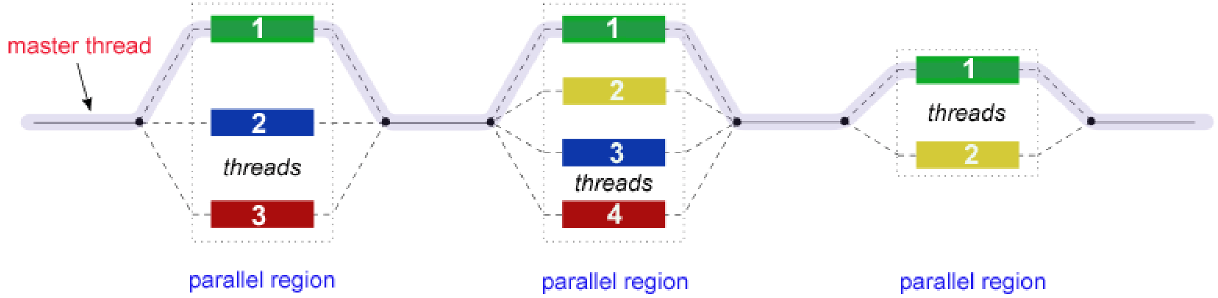


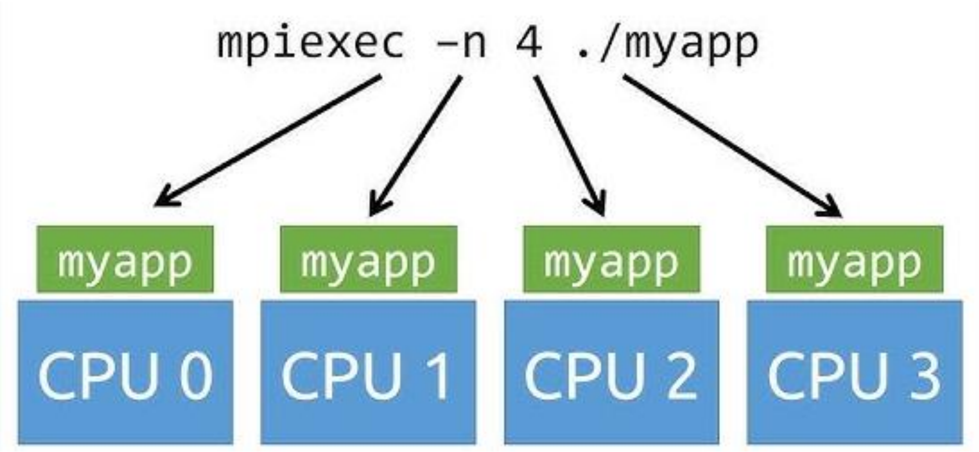
Objective and Motivation

Objective:	Speed up convolution for image processing by leveraging parallel computing
Motivation:	Convolution is an expensive and time-consuming operation to achieve real-time processing traditional methods are too slow

Parallel Computing for Convolution

Aspect	OpenMP	MPI
Memory	Shared Memory: All threads access could access the same memory	Distributed Memory: Each process has its own memory
Focus	Single-machine parallelization	Multi-machine/cluster parallelization
Scalability	Limited to available cores	Scales to multiple machines

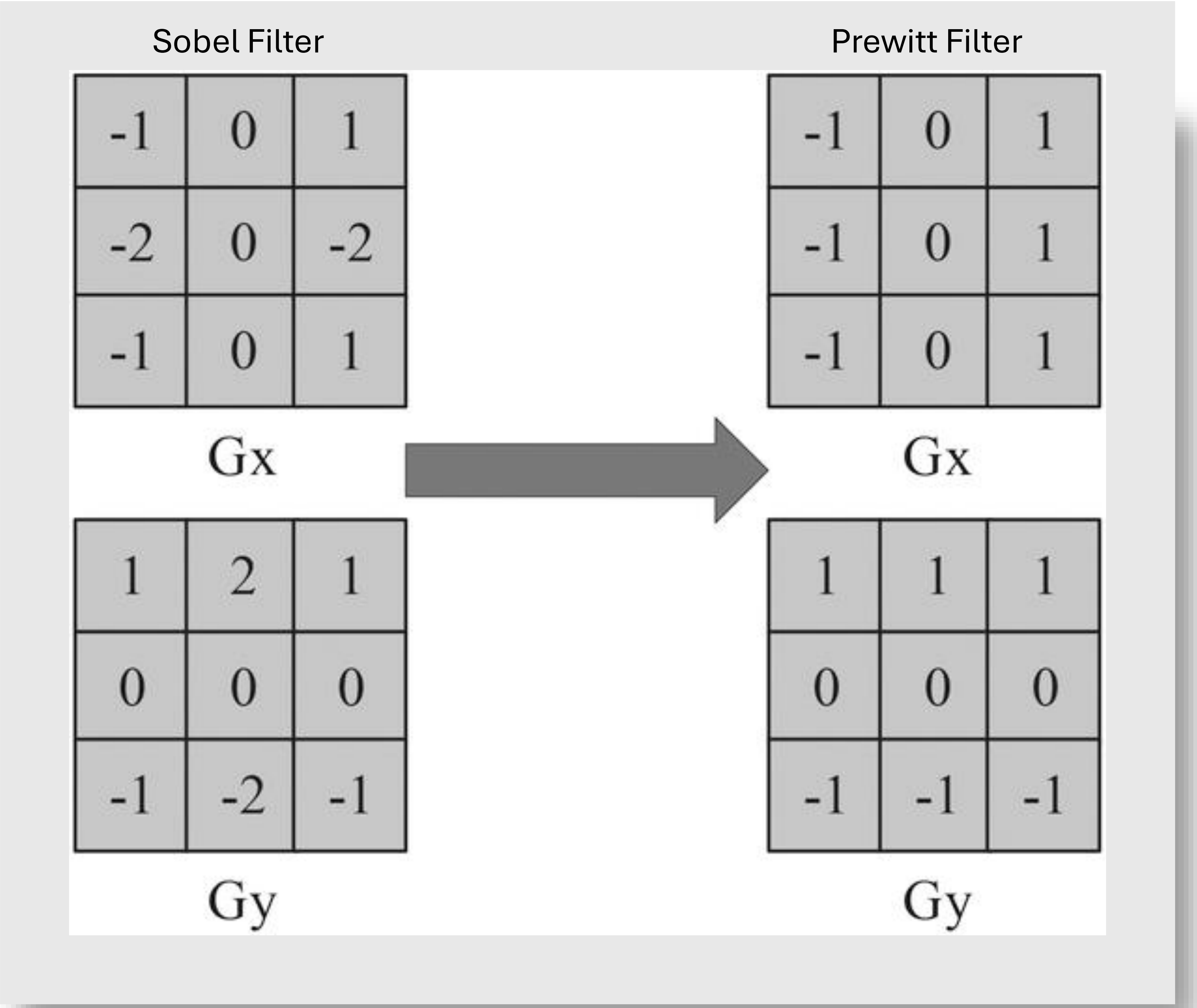




Sobel/Prewitt Filters

Edge detection filters are used to emphasize regions of high-intensity change

Sobel Filter	Prewitt Filter
Larger values	Simpler values
Smooth edge detection with noise suppression	Edge detection and more sensitive to noise



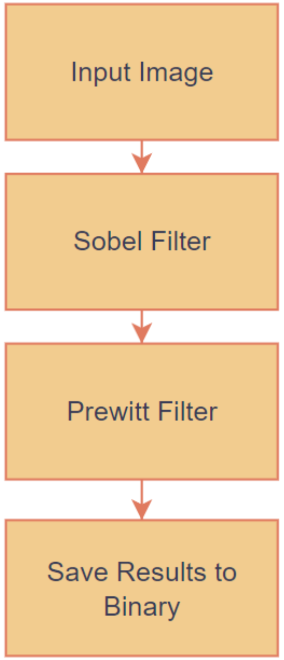
Naïve Convolution

Single-Threaded:

- Processes the filters sequentially, pixel by pixel
- Nested loops to apply filters

Implementation:

- Simpler with no dependencies on parallelization




```
// Sobel filter kernels for horizontal (Gx) and vertical (Gy) edge detection
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}};

// Iterate over each pixel, excluding the edge rows and columns
for (int y = 1; y < height - 1; y++) {
    for (int x = 1; x < width - 1; x++) {
        int sumX = 0, sumY = 0; // Initialize accumulators for Gx and Gy gradients

        // Compute convolution with the Sobel kernels (3x3 window)
        for (int i = -1; i <= 1; i++) {
            for (int j = -1; j <= 1; j++) {
                int idx = ((y + i) * width + (x + j)) * 3; // Flattened 3D index for pixel
                int intensity = (image[idx] + image[idx + 1] + image[idx + 2]) / 3; // Grayscale intensity

                // Accumulate weighted values from the Gx and Gy kernels
                sumX += intensity * Gx[i + 1][j + 1];
                sumY += intensity * Gy[i + 1][j + 1];
            }
        }

        // Clamp results and store in the output image
        int idx = (y * width + x) * 3; // Index of the current pixel in the output array
        output[idx] = abs(sumX) > 255 ? 255 : abs(sumX); // Red channel: Horizontal edges
        output[idx + 1] = 0;                             // Green channel: Unused
        output[idx + 2] = abs(sumY) > 255 ? 255 : abs(sumY); // Blue channel: Vertical edges
    }
}
```

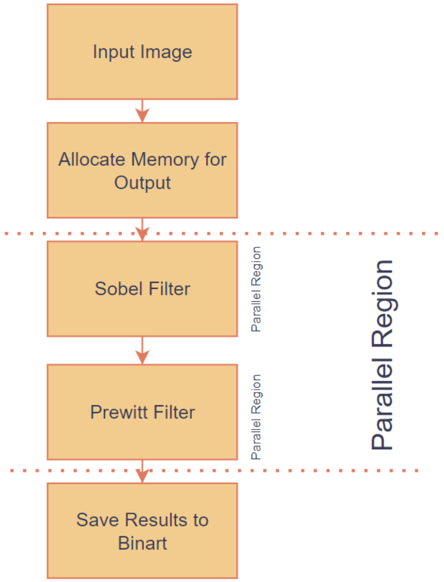
OpenMP Convolution

Shared Memory Parallelization:

- Split workload across multiple threads
- Each thread processes a portion of the image

Parallelized Sections:

- Sobel and Prewitt filters applied in parallel
- Loops are parallelized for convolution operation




```
// Apply Sobel and Prewitt filters in parallel using OpenMP sections
#pragma omp parallel sections
{
    #pragma omp section
    {
        apply_sobel_filter_color_coded(image, sobel_output, width, height);
    }

    #pragma omp section
    {
        apply_prewitt_filter_color_coded(image, prewitt_output, width, height);
    }
}
```

```
#pragma omp parallel for schedule(static)
for (int y = 1; y < height - 1; y++) {
    for (int x = 1; x < width - 1; x++) {
        int sumX = 0, sumY = 0;
        for (int i = -1; i <= 1; i++) {
            for (int j = -1; j <= 1; j++) {
                int idx = ((y + i) * width + (x + j)) * 3;
                int intensity = (image[idx] + image[idx + 1] + image[idx + 2]) / 3;
                sumX += intensity * Gx[i + 1][j + 1];
                sumY += intensity * Gy[i + 1][j + 1];
```

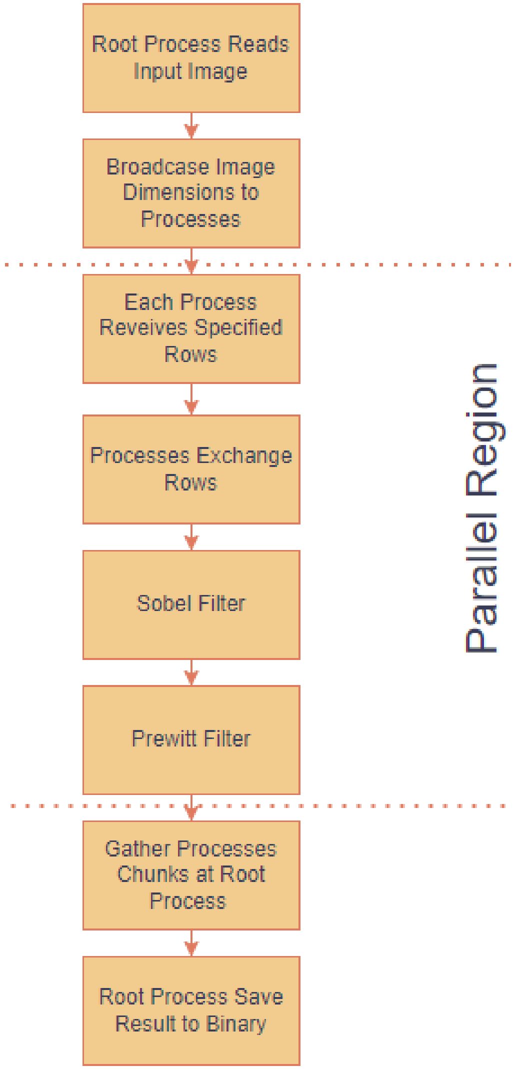
MPI Convolution

Parallelization:

- Image split into chunks of rows
- Each process computes convolution on its assigned chunk

Row Exchange:

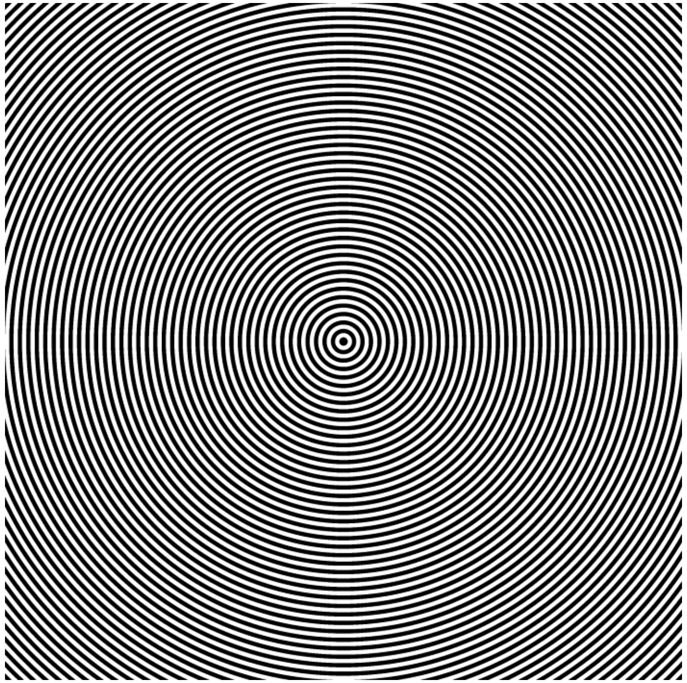
- Neighboring processes exchange top and bottom rows to ensure convolution for edge pixels



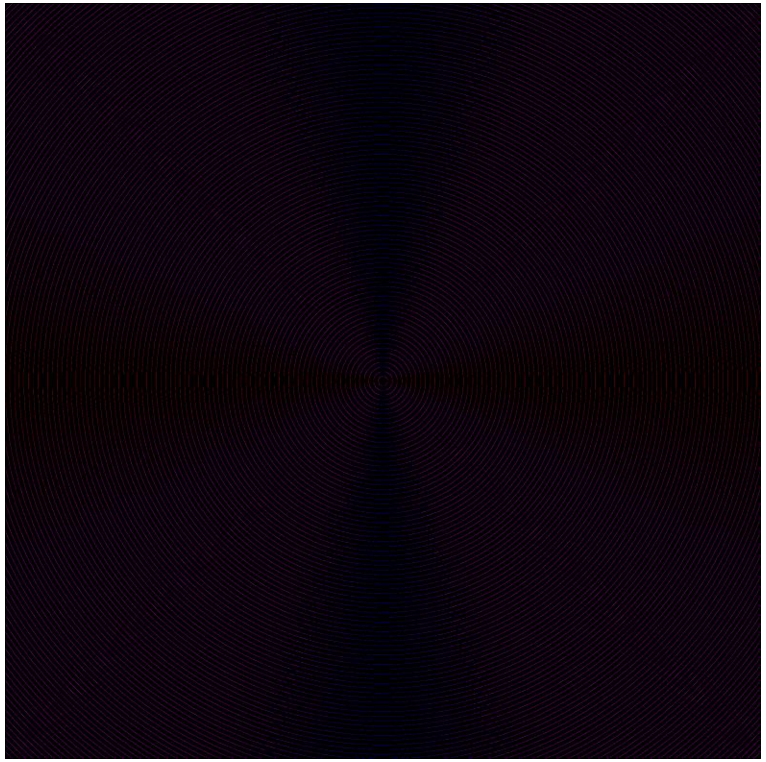

```
// Exchange halo rows
if (rank > 0) { // Send top row to previous rank and receive from it
    MPI_Sendrecv(&local_image[width * 3], width * 3, MPI_UINT8_T, rank - 1, 0,
                 local_image, width * 3, MPI_UINT8_T, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
if (rank < size - 1) { // Send bottom row to next rank and receive from it
    MPI_Sendrecv(&local_image[local_rows * width * 3], width * 3, MPI_UINT8_T, rank + 1, 0,
                 &local_image[(local_rows + 1) * width * 3], width * 3, MPI_UINT8_T, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

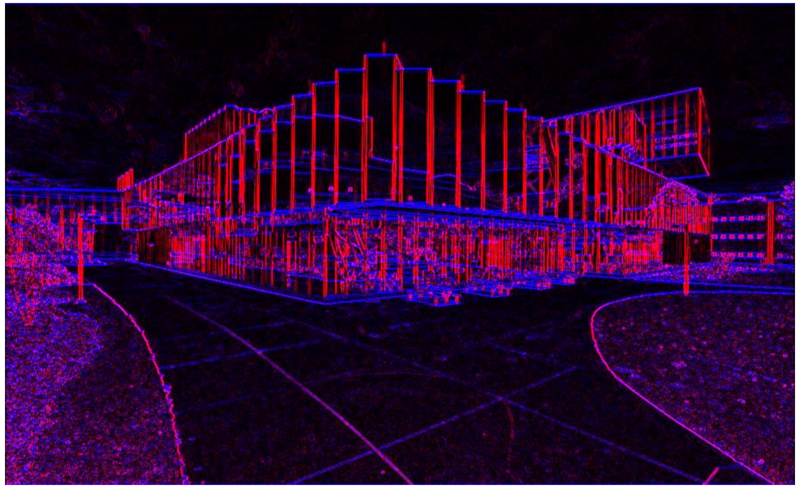
Experimental Images

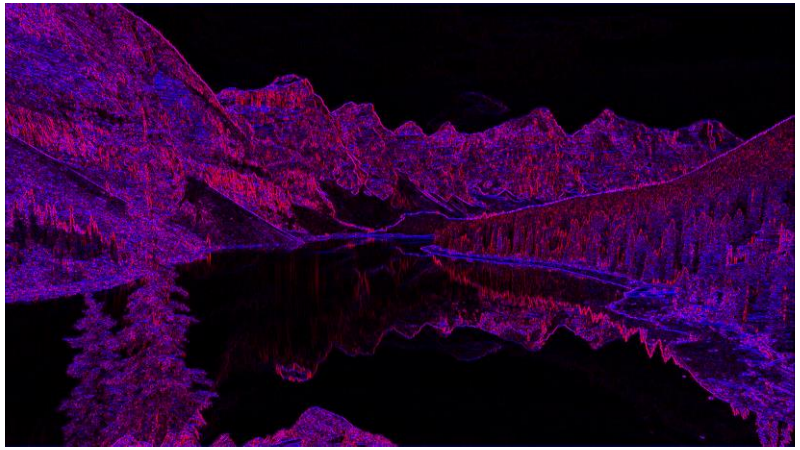
- Small Image: 900x550
- Medium Image:1920x1080
- Large Image: 4800x600
- Extra Large Image: 8000x8000



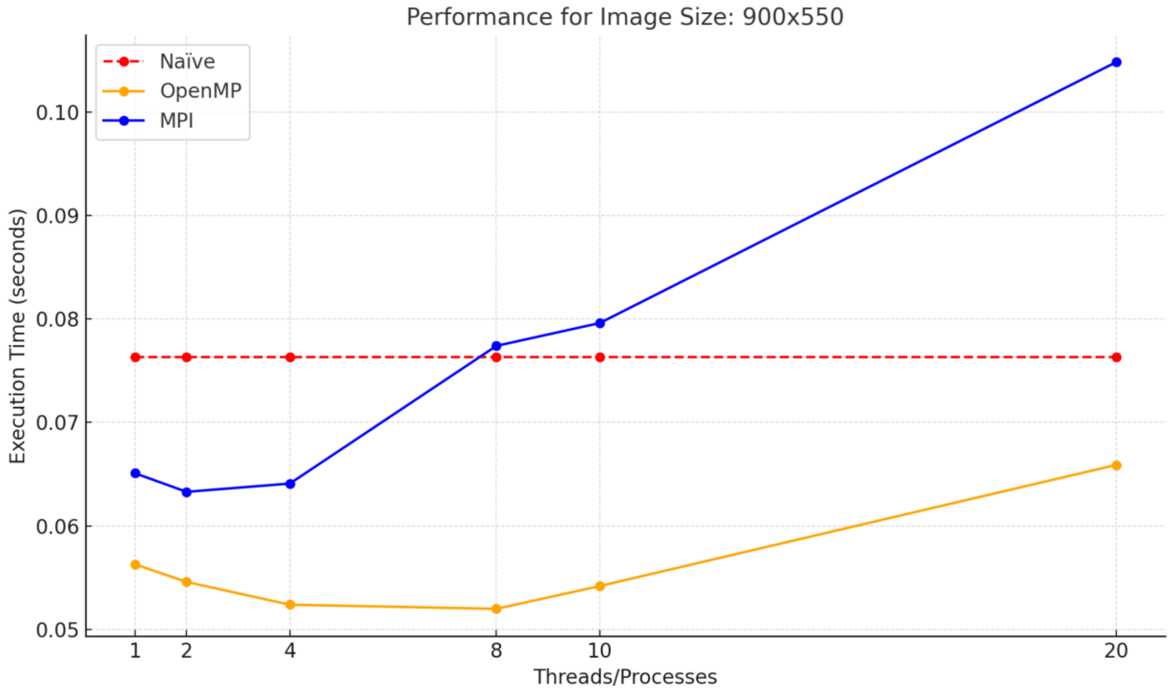
Output Images

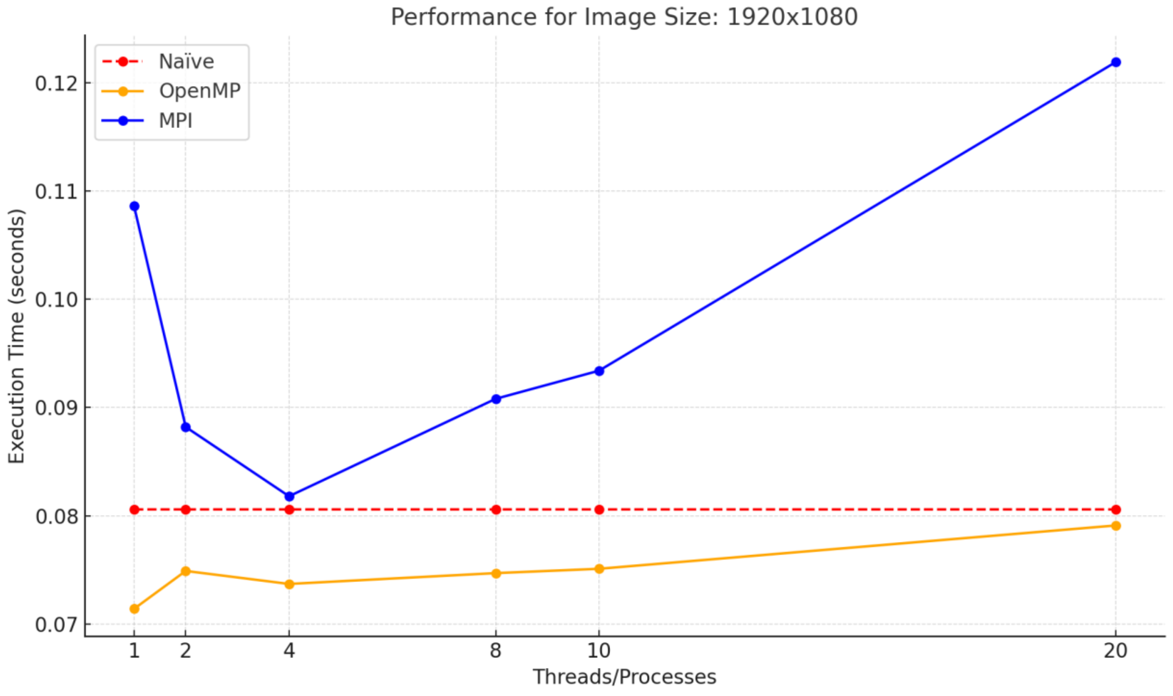




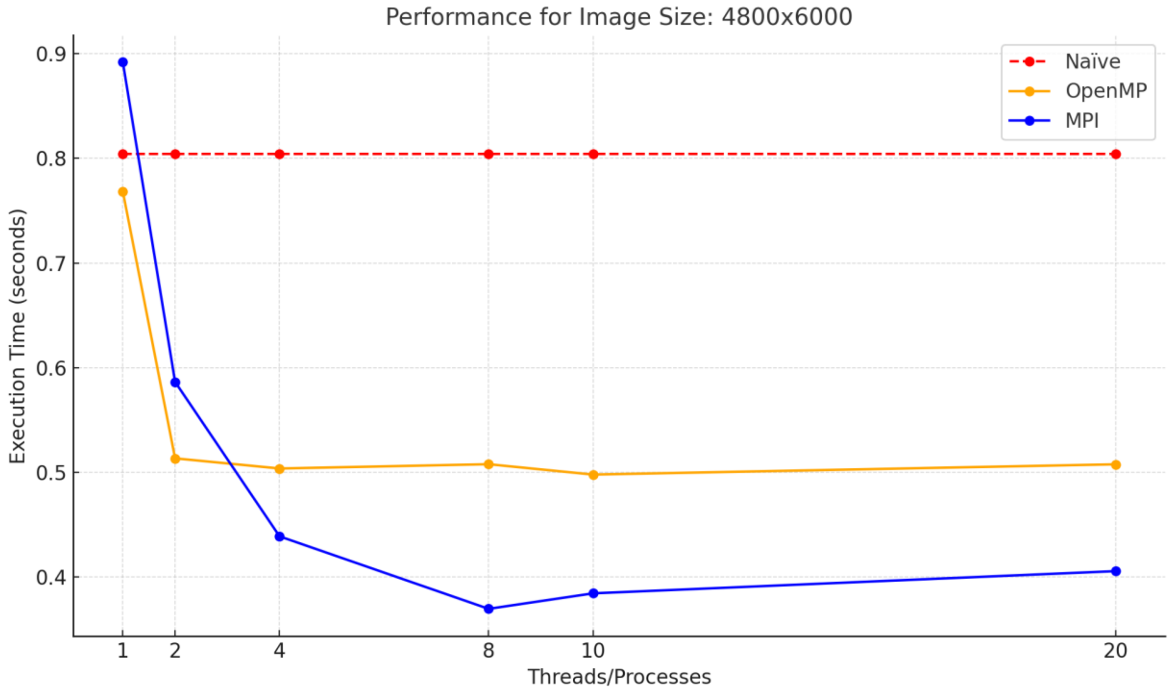


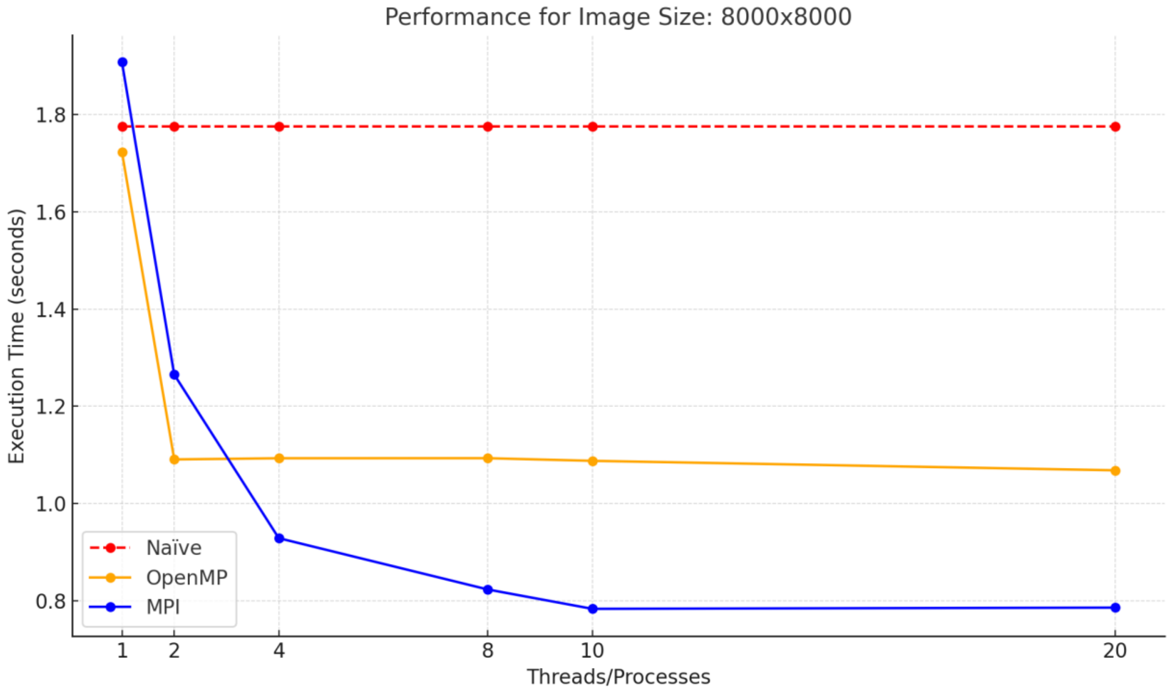
Smaller Image Timing Results





Larger Image Timing Results





Cluster Results

Metric	Local (32 threads)	Local (20 threads)	Cluster (32 threads)	Cluster (20 threads)
Total Execution Time	0.285738 s	0.324131 s	0.409255 s	0.395718 s
I/O Time	0.158111 s	0.177488 s	0.320754 s	0.309978 s
Computation Time	0.036545 s	0.058672 s	0.008695 s	0.007822 s
Communication Time	0.085594 s	0.080599 s	0.078735 s	0.077002 s

Performance Analysis

Small Images (900x550 and 1920x1080)	Large Images (4800x6000 and 8000x8000)
• OpenMP consistently outperforms Naïve and MPI • MPI shows overhead for small images and performs worse then OpenMP and Naïve • Smaller datasets most likely don’t outweigh the communication overhead for MPI	• MPI consistently outperforms Naïve and OpenMP as cores increase • OpenMP shows improvement over Naïve but plateaus as threads increase • MPI has a strong improvement on larger datasets allowing it to leverage the cores

Trade-offs

Feature/Aspect	Naive	OpenMP	MPI
Ease of Implementation	Simple, single-threaded.	Easy to use, small changes to existing code.	Complex setup with inter-process communication.
Performance	Slow for large images.	Scales well on single machines.	Scales better across distributed systems.
Scalability	Limited to one core (no parallelism).	Limited to a single machine's cores.	Works across multiple machines for large datasets.
Communication Overhead	None.	None (shared memory).	High due to inter-process communication.
Suitability for Small Data	Works, but slow.	Ideal for small to medium datasets.	Overhead may outweigh benefits for small data.
Suitability for Large Data	Extremely slow.	Limited by machine memory.	Best for large datasets distributed across nodes.

Conclusion

Summary of Findings:

- Convolution is computationally expensive but can be optimized using parallel computing
- OpenMP and MPI enable faster processing for different hardware setups

Takeaways:

- OpenMP is simple for a single machine and small datasets
- MPI is great for large datasets and multi-machine clusters

Future Areas:

- Combine OpenMP and MPI, exploring hybrid approaches