Convolution MPI Speedup

Justin Wenzel – Computer Engineering

Abstract:

Convolution operations such as applying edge detection filters for image processing, are computationally expensive and time consuming. Edge detection using convolution filters is a time consuming operation due to the large amount of data involved in the process, especially with higher-resolution images. The use of convolution is a growing field with increasing demand due to the spike of common interest in image processing, machine learning, and computer vision tasks. This study aims to highlight how parallel computing can be leveraged to speed up convolution operations on images. This is shown by applying two different edge detection filters onto a single image allowing us to explore the time it takes to apply two filters onto a single image and how each filter impacts the image. I tested different forms of parallel computing involving shared and distributed memory implementations, using OpenMP and MPI. By using different image sizes ranging from small to extra large I explored the tradeoffs of execution time, scalability, and overhead from each implementation. This report discusses the problem, approaches of parallelization, and performance evaluation of results and provides a insight into analyzing the optimizations that can be used for convolution tasks.

Problem:

Convolution is a fundamental operation involved in image processing. It has many growing application uses in image enhancement, edge detection, noise reduction, and feature extraction. The convolution operations are done by sliding a filter also known as a kernel, over an image and computing the pixel value at each location by using its neighboring pixel values. The filters being used in our tests include the Sobel and Prewitt filters which are used for edge detection both following a different implementation for the values in the filter. These filters create the opportunity to explore how each impact the resulting image and the time complexity it takes to apply multiple convolution operations in one pass of a program on an image. The filters work by calculating the amount of intensity variation in an image by determining the gradient magnitude in both horizontal and vertical directions, this isolates areas of sharp intensity which then corresponds to edges.

The convolution process is costly in computation because every pixel in an image goes through the operation involving the filter. For every resulting pixel the computation accesses the neighboring pixel values in the filter's region, then multiplies them by the corresponding values in the filter also known as kernel weights, and finally summing them up. The computational complexity of the image is directly proportional to the product of the image size and filter dimensions. If the images are of higher resolution such as 4800x6000 or 8000x8000, the number of

pixels involved will be huge. The increase in the size of the image increases the time for its processing significantly, making it difficult to achieve real-time processing using conventional sequential methods to perform the computation.

This operations is not being exacerbated by the increasing demands for processing high amount of data in real-world applications quickly, in areas such as medical imaging, video analysis, and real-time object detection. A simple sequential implementation would process and image filter by filter and pixel by pixel. This method is easy to implement but it becomes inefficient as the image size continues to grow and complexity of filters grows. This limitation creates a demand for optimization using parallel computing techniques in which the computations can be divided and performed simultaneously across threads or processes.

This work explores how parallel computing frameworks such as OpenMP and MPI can be used for accelerating the convolution process. OpenMP allows for multithreading on a single machine by dividing the workload across multiple CPU cores. MPI allows the computation to be split and distributed across multiple processes, potentially spanning multiple machines also. While splitting the work can be effective we must also account for other limitations such as overhead and synchronization problems. This report continues to explore how OpenMP and MPI address the computational complexity of image convolution with regards to various image sizes and hardware settings.

Solution:

For this project to investigate the use of parallel computing to significantly reduce execution time due to inefficiencies in sequential image convolution, especially for high-resolution images. I explore a solution that includes three approaches using different parallelization techniques in the solutions. The first approach is a simple Naïve sequential implementation, second and OpenMP shared memory approach and finally a MPI distributed memory approach.

The Naïve approach will represent the baseline for comparison, where convolution is performed pixel by pixel with a single thread. A sequential approach becomes unrealistic as the data continues to grow with the amount of computations involved. To continue with improving performance on a single machine OpenMP will be used where the work for calculating different resulting rows is split or shared between threads. It is expected that OpenMP will perform effectively on a single core system for middle sized images.

To address large inputs that might not always fit into memory and are still computationally expensive for a single machine I also propose to explore the approach of using MPI to split the work among processes and machines. Each process will operate on a portion of the image rows and communicate needed rows to one another to complete operations. This approach will distribute memory across multiple machines, hopefully overcoming the limits for large images expected to occur in a shared memory system.

These three solutions will show how parallel computing can be leveraged with both shared and distributed memory systems. Potentially reducing the computational bottlenecks and provide performance improvement for image convolution.

Implementation:

Naïve Implementation:

Naïve is the most straightforward of the three solutions for applying convolution. In this implementation convolution is applied serially over the whole image. Every pixel within the image, will be computed using convolution except the edges pixels because when applying a filter edge pixels do not have all their neighbors and are neglected for a final calculation. At each pixel location the neighboring pixels in relation with the 3x3 filter region will be accessed to compute the horizontal Gx, and vertical GY gradients using the Sobel and Prewitt kernels. This straight-forward implementation was implemented by nesting loops over the image dimensions and storing the resulting output from the convolution computation in an output array. This is a simple approach but can become inefficient for large images sizes as the number of computations or loop iterations increase.

OpenMP Implementation:

OpenMP is an optimization to the Naïve approach that runs the convolution operation in parallel using shared memory on a single core by leveraging multi-threading along the image rows. The "#pragma omp parallel for" directive divides the image rows between available threads, each thread computes the resulting pixel values for the rows assigned to it in parallel. OpenMP automatically creates and synchronizes the threads in this instance allowing for a reduced overhead in the program implementation for the developer while also maintaining high-performance execution. Static scheduling was chosen to balance the workload because each row requires nearly the same computation time, the work is evenly divided among threads with static. Since we are applying two filters, we are also able to parallelize the application of both filters by having the filters operate within separate "pragma omp sections" to run the two filters simultaneously in a multi-threading nested format. This implementation uses a shared memory system to parallelize the workload and efficiently distribute work row by row resulting in higher performance.

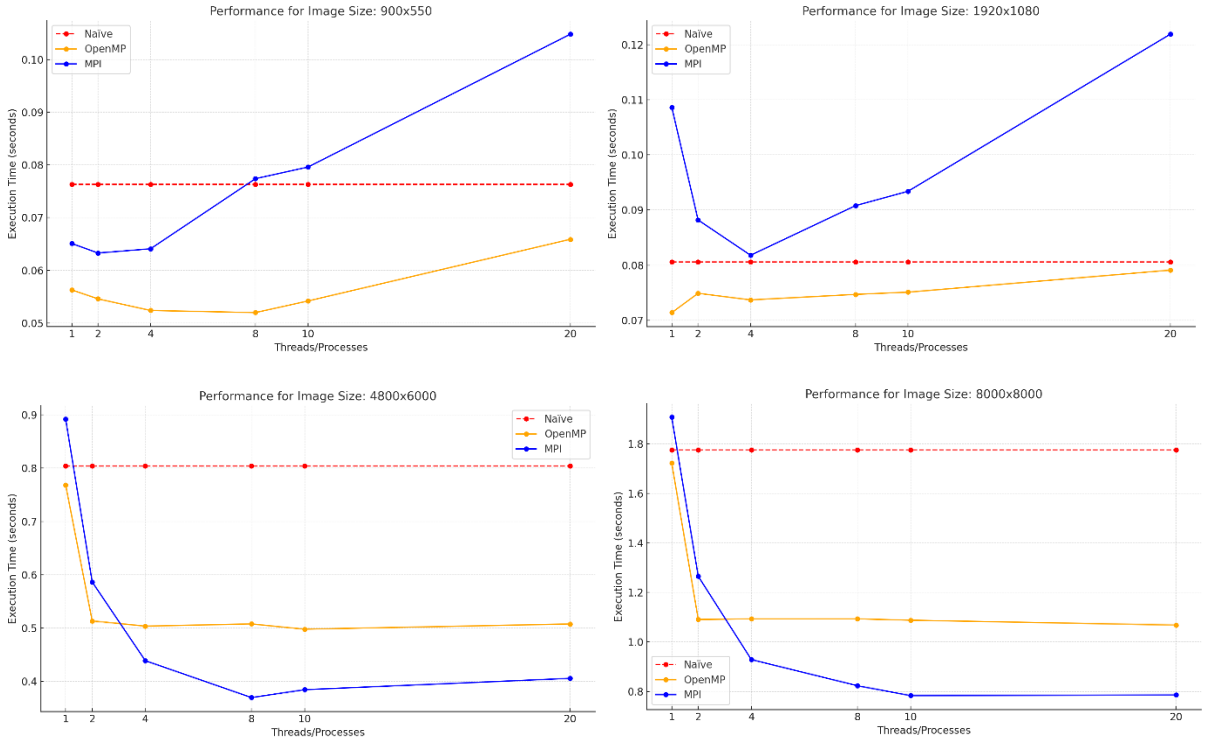
MPI Implementation:

MPI is the final optimization technique to extend parallelism across a distributed memory, this makes the environment more suitable for large datasets and potentially more scalable across multiple machines providing more memory to work with for the large data set. This approach divides the image into blocks of rows, with each process then handling a distinct block or portion of the image. The root process then scatters the blocks using the MPI_Scatterv function to ensure that each process gets the blocks it is responsible for computing. These different blocks are then

processed in parallel while also using the MPI_Sendrecv function which is used to exchanges the top and bottom rows of each block with the adjacent process as their neighboring rows overlap for the convolution operation to occur. This process ensures that every process has all the data from other processes needed to perform the convolution computation on edges of split block of rows.

Once data has been exchanged, each process executes the Sobel and Prewitt filters over their block of rows and exchanged rows from the adjacent processes. The processes then use MPI_Gatherv to send the resulting output back to the root process, where the final results are then reassembled into the final output image. MPI uses distributed memory parallelism, to enable the program to break down larger images that exceed the memory capacity of a single machine.

Results:



Execution Time Analysis:

The results provided above reflect how image size and different resource allocation for the parallelization including how allocating 1 to 20 threads/processes can affect the performance of the code. The smaller images in our tests which were900x550 and 1920x1080 show OpenMP outperforms the Naïve and MPI implementations consistently. This is believed to be due to the shared memory which reduces the communication found in MPI but also leverages parallelization which is not found in Naïve, this makes it an ideal option for smaller images. As MPI exceeds Naïve due to its complex communication overhead, managing a small image becomes a bottleneck.

The larger image results for size of 4800x6000 and 8000x8000 the results are flipped and favor the MPI implementation more as it shows the highest reduction of execution time as the number of processors available increase. This shows how the distributed memory of MPI can more easily distribute larger portions of the image across processes making the communication overhead more irrelevant. It can finally be noticed that the Naïve implementation stays static as it does not leverage parallelization and creates a baseline to identify how OpenMP is great for smaller images and MPI is better for larger images. The results also show the scalability limitations and impact that comes with communication and thread management overheads. The table below provides a summary of each implementation's trades offs.

Trade-offs Tables:

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.

Conclusion:

This project showed the capabilities of parallel computing for convolution on images using the three different implementation methods, Naïve, OpenMP, and MPI. OpenMP managed to show great speedup in execution for small and medium-sized datasets by leveraging shared memory parallelism while MPI showed results for improving execution time for larger datasets by distributing the workload across processes. The implementation shows how MPI was less efficient for smaller images due to its communication overhead.

The results also showed that the choice of a proper method depends on the dataset size for the workload and the hardware that is available. For single machines and moderate workloads OpenMP is preferred but for large datasets and workloads MPI's distributed memory is more favored. This opens an opportunity to explore hybrid approaches mixing both OpenMP and MPI for further performance optimizations that can address a range of diverse datasets sizes and architecture that is available.