Parallel Image Processing

Lorenzo Fasol
University of Trento
Trento, Italy
Student ID: 227561
lorenzo.fasol@studenti.unitn.it

Alessandro Iepure
University of Trento
Trento, Italy
Student ID: 228023
alessandro.iepure@studenti.unitn.it

Riccardo Minella
University of Trento
Trento, Italy
Student ID: 227326
riccardo.minella@studenti.unitn.it

***Abstract*—This report explores parallel image processing algorithms using shared and distributed memory paradigms, along with GPU acceleration. After developing a baseline serial implementation, the algorithm is parallelized using OpenMP, MPI, and CUDA, focusing on convolution-based image filtering for edge detection. Experiments are conducted on a heterogeneous HPC cluster, analyzing performance and scalability. Through theoretical analysis and practical experimentation, this report provides insights into the performance characteristics of parallel image processing algorithms and their scalability across different architectures.**

I. Introduction

Image processing, a fundamental task in computer vision and digital imaging, involves manipulating and enhancing digital images to extract useful information or to improve their visual quality. In many real-world applications, such as medical imaging, video processing, and remote sensing, the performance and efficiency of such algorithms are crucial for timely and accurate results. With the ever-increasing size and complexity of image data, there is a growing need for efficient parallel algorithms to handle the computational demands of this kinds of tasks.

The objective of this project was to explore and implement parallel image processing algorithms using shared and distributed memory paradigms and GPU accelerators. Specifically, we focused on parallelizing image filtering operations, which are commonly used for tasks such as noise reduction, edge detection, and image enhancement. We implemented and optimized a convolution-based image filtering algorithm to detect edges, which applies a filter kernel to each pixel in the input image to produce the corresponding pixel in the output image.

We began by developing a serial implementation of the algorithm and subsequently parallelized it using OpenMP for shared memory systems, MPI for distributed memory systems, and CUDA to harness the power of GPU cores. The parallel algorithms were designed to distribute the computational workload efficiently among available threads or processes, leveraging the capabilities of modern multi-core CPUs, distributed computing clusters and co-processors.

To ensure the correctness and effectiveness of the parallel implementations, appropriate boundary-handling strategies were employed to handle edge pixels during the convolution operation.
The project also analyzed performance and optimizations, including evaluating the speedup achieved by the parallel algorithms for different numbers of threads or processes.

Overall, this project aimed to provide a comprehensive understanding of parallel image filtering algorithms, their implementation, and their performance characteristics under different configurations. Through theoretical analysis and practical experimentation, we sought to gain insights into the scalability, efficiency, and adaptability of parallel image processing techniques.

II. Implementation

The baseline algorithm implemented starts by reading an input image from disk. The image is then processed by applying padding around the perimeter. This is done to ensure that the output image has the same dimensions as the input image after the convolution is computed. In our implementation, we have chosen to use zero-padding as the boundary-handling strategy. This decision was made based on several factors, including simplicity and computational efficiency. While other techniques such as mirror padding or wrapping around the image could also be considered, we opted for zero-padding due to its minimal impact on performance and its ability to preserve the overall structure of the image. By focusing on a single boundary-handling strategy, we can more effectively analyze and compare the performance and output quality of our parallel image filtering algorithms. After the convolution is computed, the output image is saved to disk and the elapsed time is printed. The latter considers only the convolution operation and not the time spent in I/O operations. This is also the performance metric we used to evaluate the speedup achieved by the different parallel algorithms. The convolution is based on a simple nested loop structure, where the filter kernel is applied to each pixel in the input image to produce the corresponding pixel in the output image. In our implementation, we have chosen to use a simple 3x3 filter kernel for edge detection.

The serial implementation of the algorithm does verbatim what has been described so far.

The shared memory parallel implementation uses OpenMP to distribute the computational workload among available threads. It does so by introducing parallel regions in the padding and convolution operations. The number of threads are set at runtime using the `OMP_NUM_THREADS` environment variable.

The distributed memory parallel implementation uses MPI to divide the computational workload among all available processes. The trick here is to divide the input image into equal-sized chunks and pad them using the neighbors data. To achieve the correct padding, the input image is first scattered to all processes, then each one sends and receives the top and bottom rows to and from its neighbors. On the reconstructed local image we apply the convolution as usual and the results are gathered at the end.

The GPU parallel implementation uses CUDA to harness the power of GPU cores. The kernel is stored on the GPU's constant memory and the input image is copied to the device memory after loading it from disk. Padding is achieved by copying the input image to a larger array on the device memory. This operation is done in a separate CUDA kernel from the convolution operation. The latter is computed in parallel by a grid of threads, where each thread computes the convolution of a single pixel in the output image. The results are then copied back to the host memory in order to be saved to disk. Execution times are measured via CUDA events for an accurate value.

III. Experiments and System Description

For our experiments, we conducted performance evaluations on the University's HPC cluster. Depending on the version of the algorithm executed and the resources required, the computing node(s) used changed. The cluster is composed of an heterogeneous set of nodes, each with different hardware. Multiple runs were performed for each version of the algorithm, and the results were averaged to obtain a more reliable estimate of the performance.

Software wise, all nodes in the cluster run CentOS Linux 7 and the code was compiled using the GNU Compiler Collection 9.1.2 (GCC) for the sequential and OpenMP implementations, MPICH 3.2 (MPICC) for MPI and the NVIDIA CUDA Compiler 11.3 (NVCC).

All tests were performed on the same input image, Fig. 1, with a resolution of 2560x1920 and 3 color channels per pixel. We chose this image because it is a standard test card used in television broadcasting and it contains a variety of patterns and colors that are useful for testing edge detection algorithms. The image was also chosen because it is relatively large, which allows us to better analyze the performance of our parallel algorithms. Image operations are performed with the STB Image library, which is a simple library to load and save images from files. The library is written in C and is able to load and save images in a variety of formats, including JPEG, PNG, BMP, and TGA. Our code works with any image as long as it is in one of the supported formats and - for the parallel versions only - the height of the image is a multiple of the number of processes or threads.

After experimenting with different block sizes in the CUDA version, we found that increasing the block size did not result in any significant performance improvement. This is likely because the input image is relatively small, and the overhead of launching a large number of threads outweighs any potential benefits. As a result, we decided to stick with a block size of 32 which is CUDA's maximum value in our setup.

Fig. 1. Philips PM5544 Test card, the input image for the experiments.

TABLE I
Results of the experiments - Execution times.

	Cores	**Time** [ms]	**Speedup** [×]	**Efficiency** [%]
Sequential	-	758	1.00	100
OpenMP	2	397	1.91	95.55
	4	200	3.79	94.75
	8	104	7.27	90.81
	16	56	13.62	85.10
	32	33	22.74	71.06
	64	31	24.72	38.62
MPI	2	415	1.83	91.33
	4	257	2.95	73.64
	8	151	5.02	62.75
	16	116	6.52	40.72
	32	100	7.58	23.69
	64	92	8.27	12.92
CUDA	-	10	75.80	-

IV. Results and Conclusions

OpenMP and MPI versions were tested multiple times each with a different amounts of threads and processes, respectively. On the other hand, the sequential and CUDA versions were tested only once. The results of three runs were averaged to obtain a more reliable estimate of the performance. The speedup was calculated as the ratio between the execution time of the sequential version and the execution time of the parallel version. The efficiency was calculated as the ratio between the speedup and the number of threads or processes. The results are shown in Table I.

The experimental results demonstrate the performance characteristics of different parallelization techniques for the image processing algorithm. The sequential implementation serves as the baseline. Assuming ideal conditions, parallel execution times should decrease linearly as the number of cores increase. However, our experiment demonstrates that this is not always the case. Analyzing the results on OpenMP, from 32 to 64

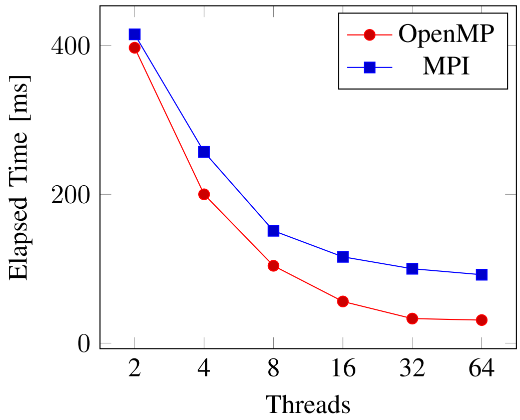

Fig. 2. Comparison graph by number of cores

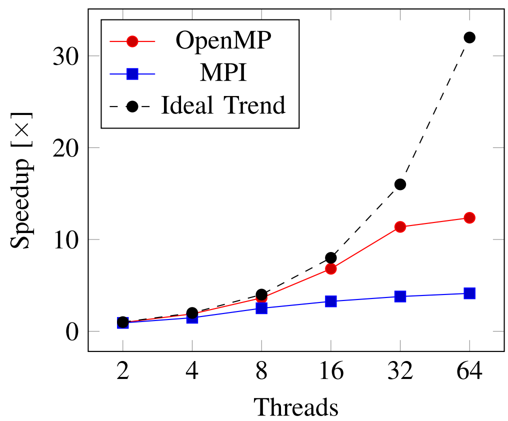

Fig. 3. Speedup graph by number of cores

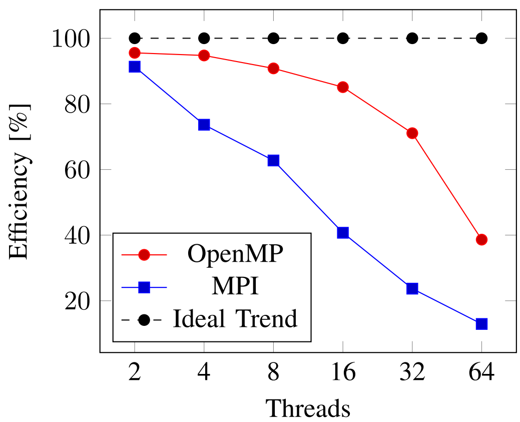

Fig. 4. Efficiency graph by number of cores

cores, the efficiency drops significantly as is observable in Fig. 4. Similarly, the MPI implementation shows a more pronounced drop between 8 and 64 processes. This is likely due to increased overhead and resource contention, which diminishes the returns in speedup. Discussing the speedup, OpenMP adheres to the ideal trend up to 32 cores before stabilizing. On the other hand, MPI exhibits limited speedup starting from 8 cores, remaining constant thereafter. This trend is also evident in Fig. 3. The CUDA implementation, achieves the highest speedup. However the resulting efficiency is not directly comparable to the one from the other CPU-based implementations in this experiment.

Given the techniques explored, architectures leveraging GPU acceleration, as demonstrated in the CUDA implementation, offer the highest speedup, making them particularly effective for image processing tasks.

Overall, our findings provide insights into the effectiveness and limitations of different parallelization techniques for image processing algorithms. Future research could explore optimization strategies to address scalability issues and further leverage GPU acceleration for improved efficiency. The experimental results underscore the importance of choosing the appropriate parallelization technique based on the characteristics of the algorithm and the underlying hardware architecture, enabling researchers and developers to make informed decisions to optimize performance and address computational challenges in image processing tasks.

V. Project Contributions and Reproducibility

This report is the result of a collaborative effort between the authors. All decisions were discussed, agreed on and developed together. Each author focussed mainly on:

- **Lorenzo Fasol**:
  - Sequential version
  - OpenMP version
  - CUDA version
- **Alessandro Iepure**:
  - Sequential version
  - MPI version
- **Riccardo Minella**:
  - Sequential version
  - OpenMP version
  - MPI version

The report was co-authored by all of us. All source codes for the various algorithm implementations, along with the scripts used for the experiments and other assets, are available in the project's GitHub repository at the following link https://github.com/aleiepure/Parallel-final. The repository also contains a PDF copy of this document and the LaTeX source code.