Parallel Computing using MPI, openMP, and CUDA

Yessin Moakher
Ecole Polytechnique
yessin.moakher@polytechnique.edu

Yuming Zhang
Ecole Polytechnique
yuming.zhang@polytechnique.edu

***Abstract*—Edge detection is crucial in image processing, with the Sobel filter being a common method for detecting intensity gradients. This project accelerates the Sobel filter for GIF images using OpenMP (shared-memory), MPI (distributed-memory), and CUDA (GPU) parallelization. We compare their performance across different image sizes and hardware configurations, highlighting their strengths and trade-offs. You can find the code in the github : https://github.com/fegounna/INF-560-Parallel-Computing**

***Index Terms*—HPC, openMP, MPI, CUDA**

I. Introduction

Edge detection is crucial in image processing, enabling applications in medical imaging, autonomous vehicles, and object recognition. It identifies boundaries between regions with different intensities, serving as a foundation for segmentation and feature extraction.

Among various methods, the Sobel operator is commonly used due to its simplicity and effectiveness. However, applying it to real-time high-resolution images processing is computationally demanding. To address this challenge, parallelization distributes the workload across multiple processing units.

This project report explores three parallelization strategies for Sobel edge detection:

1) OpenMP (Shared Memory): Uses multi-core CPUs for concurrent processing.

2) MPI (Distributed Memory): Spreads computation across multiple nodes in a cluster.

3) CUDA (GPU Parallelization): Leverages the massive parallelism of GPUs.

We evaluate these methods in terms of scalability, efficiency, and practical trade-offs across various image sizes and hardware setups.

A. Report overview

The remainder of this project report is organized as follows: Section 2 provides background information on the Sobel edge detection algorithm. Section 3 details our implementation …

II. Sobel edge detection algorithm

A. Image Filtering Pipeline

The edge detection process for each image of the animated GIF consists of three sequential steps: a) *Grayscale Conversion:*

Reduces the three-channel RGB image to a single intensity channel by averaging the RGB components.

b) *Blur Filter:* Blures top and bottom regions.

c) *Sobel Edge Detection:*

- Computes intensity gradients in horizontal (x) and vertical (y) directions using 3×3 Sobel kernels.
- Applies a thresholding operation: pixels with gradient magnitudes above a threshold are classified as edges (white), while others are non-edges (black).

III. Approach

A. Shared Memory System

In a shared memory system, all processing cores operate within a single address space where each memory location has a unique address. Since data stored in memory is accessible to all cores, synchronization mechanisms are required to ensure consistency.

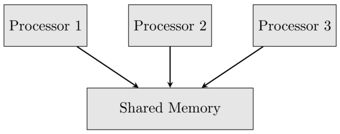
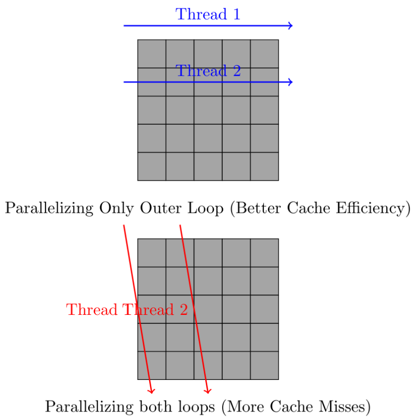


Fig. 1: Illustration of a Shared Memory System.

OpenMP is one of the most widely used parallel programming models for shared memory systems. It provides high-level constructs for parallelism, including work-sharing and synchronization mechanisms.

For each image of the animated gif, all operations inside the loops approximately take the same amount of time, it is straightforward to use schedule(static). The static schedule ensures balanced workload distribution, as iterations are divided equally among threads at compile time, avoiding the overhead of dynamic scheduling.

Another consideration when parallelizing code is the memory access pattern. In our tests, we found that parallelizing only the outer loop of a nested loop structure was more efficient than applying collapse(2) to parallelize both loops. The key reason for this is the memory access patterns. Since we are working with arrays, the collapse(2) strategy can lead to non-contiguous memory access for each thread. In contrast, parallelizing only the outer loop allows each thread to work on larger, contiguous blocks of memory. This improves data reuse within the cache and reduces cache misses.



Parallelizing Only Outer Loop (Better Cache Efficiency)

Parallelizing both loops (More Cache Misses)

Fig. 2: Cache Efficiency Comparison: Outer Loop vs Both Loops parallelization.

In our blur filter example, we parallelized the 3 loops acting separately on the top, bottom and middle part of the image with the nowait() directive to allow threads to proceed without waiting for others. This enables more parallelism and reduces the overall execution time.

It is important to note that not all loops require parallelization. For small loops, the overhead of creating threads may outweigh the benefits of parallel execution.

B. *Distributed System*

a) *Task Distribution Methods:*

In parallel computing environments using MPI, task distribution plays a crucial role in determining performance and scalability. One approach is for Rank 0 to handle all I/O operations by reading GIF files centrally and subsequently distributing the data or tasks to other processes. If Rank 0 is responsible for task distribution, an efficient scheduling strategy is required to minimize idle time across processes.

Our approach is the Master/Worker model, where Rank 0 acts as the master node, reading GIFs and creating work units that are then assigned to worker processes via MPI. A typical strategy involves an initial allocation of tasks to each worker, followed by dynamic scheduling: as workers complete their tasks, they send results to Rank 0 and request new tasks. This dynamic approach ensures that idle processes promptly receive new tasks, reducing overall execution time. This method is straightforward but suffers from scalability limitations as Rank 0 becomes an I/O bottleneck. When dealing with large file sizes or numerous GIF files, the memory usage and latency at Rank 0 can increase significantly, leading to inefficient resource utilization.

Efficient parallel processing of GIF images requires an adaptive task partitioning strategy to balance computational workload while minimizing memory overhead. Our algorithm is designed to generate processing tasks for GIF frames based on their size, leveraging both batch processing for small frames and block-wise decomposition for large frames. This approach optimizes cache locality and enhances parallel scalability.

The task creation process follows an adaptive partitioning strategy. Initially, the function retrieves the total pixel count and number of frames for the given GIF file. If the total number of pixels in the file is below Small frame threshold, no explicit tasks are generated, and the processing of such files is deferred to a batch-merging mechanism handled elsewhere. This avoids unnecessary task fragmentation for small GIFs, thereby improving processing efficiency.

For larger GIFs, the function iterates over all frames and determines their individual sizes. If a frame's pixel count exceeds the Large frame threshold, the frame is subdivided into multiple processing blocks along the row dimension to enhance parallelism. The number of blocks is computed based on the ratio of frame size to this threshold, ensuring that each block remains within an optimal processing size. The function dynamically determines the height of each block, ensuring load balancing across the blocks by distributing any remainder rows evenly. Each block is then encapsulated into a customized class structure, which includes the file index, frame index, starting row in the pixels array, width, and height.

TABLE I: Representative meta data for different GIF files. The dataset includes cases with many images but low pixel counts, single large images,etc.

GIF File	Frames	Total Pixels	Load Time (s)
"fire.gif"	33	59400	0.001043
"9815573.gif"	10	1405000	0.018764
"TimelyHugeGnu.gif"	19	4372920	0.045146
"giphy-3.gif"	5	10366920	0.09594
"Mandelbrot-large.gif"	1	69682912	0.405383

b) *Data Distribution Granularity:*

Per-File Task Allocation: The coarsest granularity is to allocate entire GIF files as individual tasks, where each MPI process processes a set of complete GIFs. This method minimizes communication overhead, as each task involves only one data transfer. It also facilitates intra-process parallelization using OpenMP. However, if GIFs vary significantly in frame count or resolution, this approach may lead to load imbalance: processes handling larger GIFs may take disproportionately longer than those assigned smaller GIFs.

Per-Frame Task Allocation: For GIFs with multiple frames, finer granularity can be achieved by distributing tasks at the frame level. Instead of assigning entire GIFs to processes, individual frames can be processed in parallel. For instance, a 10-frame GIF can be split into 10 independent tasks, reducing the impact of imbalanced file sizes. This method requires frame extraction, which can be handled by Rank 0 before distribution or by each process independently. The latter increases implementation complexity but reduces inter-process communication. The increased task count improves scheduling flexibility but also raises communication overhead.

Per-Pixel or Block Allocation: For extremely large images, further granularity can be achieved by dividing each frame into smaller regions, such as grid blocks, allowing multiple processes to process different sections of the same image in parallel. This method is particularly useful for high-resolution images that would otherwise be computationally expensive for a single process. However, excessive partitioning can lead to significant overhead in task management and data aggregation. Furthermore, the blur and Sobel filters involve convolutional computations, which make block partitioning particularly challenging.

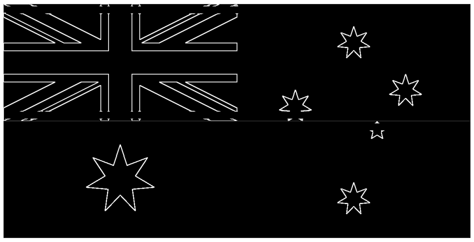


Fig. 3: The filtered image was also divided into blocks.

C. *A Hybrid Approach Based on MPI + OpenMP*

a) *Data Reading and Task Allocation:* To avoid a single-process I/O bottleneck, all MPI processes first read the metadata of the GIF files (number of frames, width, and height) in parallel. Once the metadata is retrieved, the master process dynamically partitions tasks based on the total number of frames and pixels across all GIFs:

- Direct Processing of Small GIFs: If a GIF contains a relatively small number of total pixels, GIF can be processed by a thread in Rank0.
- Frame-Level Task Partitioning for Multi-Frame GIFs: For GIFs with multiple frames, each frame is treated as an independent task. Different MPI processes handle different frames concurrently.
- Pixel Block Partitioning for Large Single-Frame GIFs: If a GIF contains only one frame but has a high resolution, the frame is divided into multiple pixel blocks, each processed independently by different MPI processes.

b) *Dynamic Thread Attribution:*

The master process generates a queue of tasks based on the metadata analysis and initially assigns tasks to all available worker processes. Upon completing a task, each worker sends its result to the master process and requests a new task. Within each MPI process, OpenMP threads are dynamically adjusted based on task size to optimize resource utilization.

c) *The Algorithm For Distributing Tasks:*

The MPI-based task distribution strategy in this implementation follows a Master/Worker paradigm to efficiently distribute the processing of GIF files across multiple computing nodes. The Master process (Rank 0) is responsible for reading input GIF files, using OpenMP parallelism, extracting metadata, generating computational tasks, and dynamically distributing them to Worker processes.

At initialization, Rank 0 loads all GIF metadata sequentially, determining the number of frames, their dimensions, and total pixel count per file. If a file's total pixel count falls below Small frame threshold, it is processed immediately by Rank 0 using an in-place sequential approach, bypassing MPI distribution. Otherwise, the metadata is used to generate a task queue via a function, where each frame or frame block (if exceeding Large frame threshold) is encapsulated as gif_task structure. This ensures that large frames are split into manageable processing blocks for optimal load balancing.

The dynamic task distribution mechanism begins with Rank 0 pre-assigning initial tasks to all available Workers. Each Worker receives a gif_task structure along with the corresponding pixel data via MPI_Isend. Upon completing a task, the Worker sends back the processed frame data to Rank 0 via MPI_Irecv and requests a new task. This process continues iteratively until all tasks are completed. If no further tasks remain, Rank 0 issues a termination signal to each Worker, signaling the end of computation.

This approach ensures efficient parallel execution by dynamically balancing workloads among Workers, preventing idle CPU time due to uneven task distribution. However, Rank 0 remains an inherent bottleneck for I/O and task scheduling, especially when handling large datasets. This limitation can make the performance worse than sequential method.

D. *GPU Parallelization*

For each image of the animated gif, the pipeline consists of transferring the input image to GPU memory, applying the sobel filter, and transferring the result back to host memory.

A straightforward GPU implementation assigns each thread to compute the gradient for a single pixel by accessing its neighboring pixels from global memory.

Each thread requires a 3×3 neighborhood to compute the Sobel gradient. Without optimization, multiple threads fetch the same pixels from global memory. This is highly inefficient because global memory access has high latency.

To mitigate this issue, tiling is employed. In this strategy, each thread block loads a small tile (subregion) of the image into shared memory before performing computations. Because shared memory is much faster than global memory, this approach allows multiple threads within the block to reuse the data efficiently. However, when applying the Sobel filter, each pixel requires access to a 3×3 neighborhood, meaning that simply loading a tile of the same size as the thread block is insufficient. To handle boundary conditions correctly, an additional halo region (extra rows and columns of pixels) must be loaded into shared memory.

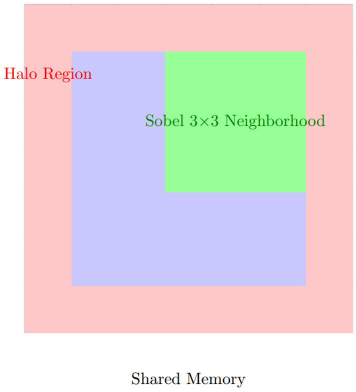


Figure 2: Illustration of the shared memory: the blue area represents the pixels corresponding to the block, the red area is the halo region, and the green area highlights a 3×3 neighborhood needed for the Sobel filter.

IV. Experimental Results

A. Experimental Setup

Workstation:

- Processor: Intel Core i7-13700K (16 cores, 32 threads)
- GPU: NVIDIA RTX A4000 (6144 CUDA cores)

Cluster Environment:

- Nodes: 4 homogeneous nodes

B. CUDA Performance: Naïve vs. Tiling Optimization

TABLE II: Average execution time for the Naïve and Tiled versions of the CUDA implementation of the third filter (Sobel), where tpb stands for threads per block.

Image Size	Naïve 32*32tpb(s)	Naïve 16*16tpb(s)	Tiled 32*32tpb(s)	Tiled 16*16tpb(s)
9933x7016	0.321±0.013	0.330±0.003	0.330±0.003	**0.302±0.023**

The difference is not significant for the Sobel filter. Since the Sobel filter is already fast on the CPU and significant performance gains are mainly observed when implementing the more computationally expensive blur filter on the GPU, we decided to use the Naïve implementation of the blur filter in CUDA for the next phases of the project. The blur filter only uses neighboring pixels for blurring the top and bottom parts of the image, so the tiling benefit is less significant.

C. Scalability Analysis

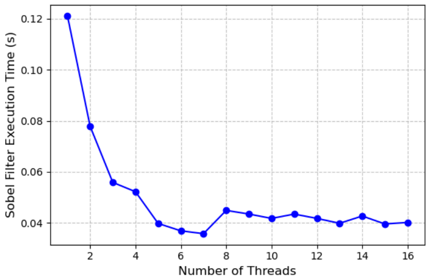


Fig. 4: Effect of Thread Count on Sobel Filter Execution Time with an 1484 x 913 image.

Figure illustrates the strong scaling behavior for the OpenMP and MPI implementations as a function of the number of cores (or processes). The OpenMP version shows nearly ideal scalability up to 8 cores on the workstation, while the MPI version exhibits diminishing returns beyond 20 cores due to increased communication overhead.

Figure 2 compares the execution time for the CUDA implementation as the image size increases. The curve demonstrates that the CUDA kernel overhead is amortized as the workload per pixel increases, resulting in nearly constant throughput for larger images.

D. Results on the Workstation

TABLE III: Execution times for the sequential, OpenMP, and CUDA implementations on the workstation for different image sizes.

Image Size	Sequential (s)	OpenMP (s)	CUDA (s)
1484 x 913	0.076471	0.026811	0.221143
1920 x 1080	0.468640	0.110511	0.184221
9932 x 7016	4.440013	1.207953	0.647956

In the OpenMP we used 7 threads and we only used cude for the blur filter.

E. Results on the hybrid model (MPI + OpenMP)

TABLE IV: Execution times on all inputs for different sources sizes.

node	cpu	thread	Execution time(s)
1	2	8	17.545
4	8	8	30.703
8	8	8	35.331

The execution results presented in Table IV indicate that the hybrid MPI + OpenMP approach does not exhibit strong scalability as the number of computational resources increases. While the single-node configuration (1 node, 2 CPUs, 8 threads) achieves the best performance, increasing the number of nodes to 4 and subsequently to 8 results in a significant increase in execution time. This performance degradation is primarily attributed to MPI communication overhead, dynamic task scheduling inefficiencies, and the additional memory allocation required for reconstructing the animated GIF format after receiving processed pixel data.

One primary factor contributing to this performance degradation is the overhead associated with the MPI-based master/ worker paradigm. As additional nodes are introduced, the frequency of inter-process communication increases due to dynamic task allocation and synchronization. The master process is responsible for managing task distribution, which introduces serialization effects that can limit scalability. Additionally, frequent MPI message exchanges introduce latency, which becomes increasingly significant as the number of worker nodes grows.

Beyond communication overhead, data transfer costs between nodes impose additional performance penalties. The hybrid model partitions large image frames across multiple MPI ranks, necessitating inter-node data exchange to assemble the final output. While OpenMP enables efficient intra-node parallelism, the necessity of redistributing received pixel data to reconstruct the GIF format introduces an additional memory allocation cost. Once a worker process completes the processing of a frame segment, it must allocate memory and reassemble pixel data into the animated GIF format before sending results to the master. This step incurs additional overhead, as memory reallocation and formatting operations involve non-trivial computational complexity, particularly for high-resolution images with multiple frames.

V. Discussion

The experimental results reveal several key findings:

- OpenMP: Provides an easy-to-implement, low-overhead solution for multi-core shared-memory systems.
- MPI: Suitable for distributed-memory environments, yet its performance is highly sensitive to the granularity of data partitioning. For small images, the communication overhead is too high; for larger images, MPI achieves a decent speedup, but not as high as the shared-memory approaches.
- CUDA: Demonstrates the best performance for large images, thanks to its ability to exploit thousands of parallel threads and high memory bandwidth.

The final model would be a decision tree based on image size and available resources to determine the most suitable method.