Image Filtering

Parallel and distributed algorithm

Zhuoyao Lin - Paul Woringer

Mar. 2023

Table of contents

Introduction

MPI, OpenMP, and CUDA are three commonly used parallel computing paradigms that are used to optimize the performance of high-performance computing applications. Each of these paradigms is designed to leverage the computational power of modern computer hardware in different ways.

MPI (Message Passing Interface) is a communication protocol that allows multiple processes to communicate with each other over a network. It is commonly used in distributed computing systems, such as clusters or grids, where the workload is divided among multiple nodes. MPI allows processes to exchange data and synchronize their actions, enabling them to work together towards a common goal (cf. section 1.1).

OpenMP (Open Multi-Processing) is a parallel programming API that is designed to enable parallelism within a shared memory system. It provides a set of compiler directives, library routines, and environment variables that allow the programmer to specify which parts of the code can be executed in parallel. OpenMP is particularly useful for exploiting parallelism in multi-core CPUs, which are common in modern computer systems (cf. section 1.2).

CUDA (Compute Unified Device Architecture) is a parallel computing platform that is designed to leverage the computational power of GPUs (Graphics Processing Units). It provides a programming model and API that allows developers to write simple code segments that can be executed in parallel on NVIDIA GPUs. CUDA is particularly useful for applications that require large amounts of parallel computation, such as machine learning, scientific simulations, and image processing.

The combination of these three parallel computing paradigms can be used to build high-performance computing applications that can leverage the computational power of modern computer hardware in multiple ways. In this project, we use MPI and a master node to distribute the work around to the other (slave) nodes. Then we use OpenMP to exploit parallelism within each node. Finally, CUDA is implemented to accelerate computations of the three filters applied (gray filter, blur, Sobel). This hybrid approach can lead to significant performance improvements and is becoming increasingly popular in scientific computing, machine learning, and other areas that require intensive computation.

Chapter 1

Parallelism Choices

The main source of parallelism in the image filtering task is data parallelism: we apply the same sequence of instructions to each image of the gif, and to each part of the image. However, since the output of one filter is the input of the next, and the output of one iteration of the blur filter is required for the next, there is a dependency between each task, and large grain control parallelism is not really an option in this case.

1.1 MPI

The image filtering process requires running three phases: gif loading, filtering and exporting. These three phases have to be done sequentially. Among them, filtering takes the longest time to process, and is the easiest to implement parallelism on. We therefore chose to focus on this task for our implementation of parallelism, and not worry about image loading and unloading, and the performance analysis done in the following sections only takes into account this phase.

The first source of parallelism we exploited is the simplest: the multiple images in a gif. In the sequential code, these images are processed one after the other. In our first improvement we used MPI to distribute the images of the gif across the different nodes of the network in a Round-Robin manner so that they are processed in parallel, making the most of available nodes.

Using this paradigm, we used the master node to first load the gif file, then oversee the scheduling of operations across the multiple nodes of the network, and finally export the result. The master node itself does not perform any filtering computation, which can be considered as a sacrifice of computational power, but it allows it to be fully available and responsive when the slave nodes need to be assigned a new task. Running this scheduling loop on the master node allows us to fully use the computational power of the others. We make use of asynchronous message passing functions (ISend, IRecv, ...) to communicate between nodes more efficiently.

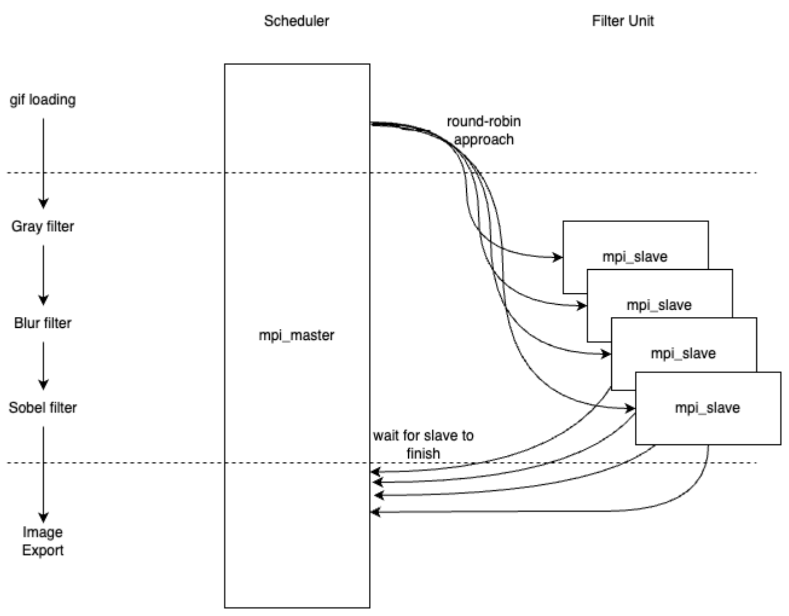

Figure 1.1: Diagram for MPI Distributed-memory model.

1.2 OpenMP

Our implementation of MPI distribute image to MPI processes. However, we can exploit more computational power by using intra-node parallelism. We can do so easily by exploiting the parallelism on the ‘for‘ loops that OpenMP allows. The following diagram illustrates the hierarchy of the network and the OpenMP threads in each node:

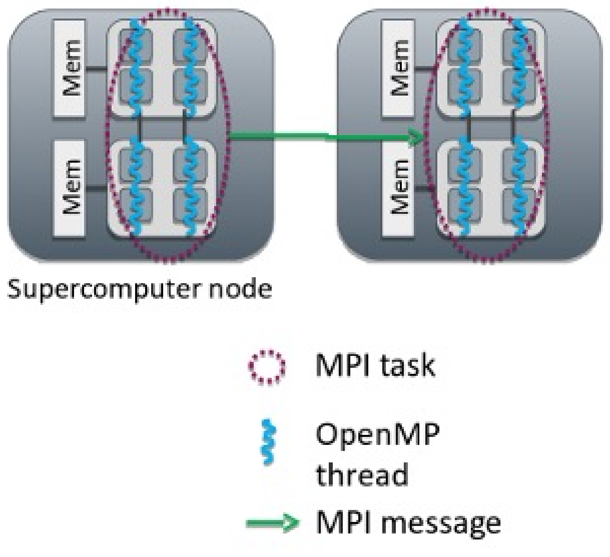

Figure 1.2: Diagram for combination model of OpenMP and MPI

1.3 CUDA

We have decided not to implement CUDA on top of MPI and OpenMP as CUDA is hard to debug and we want to keep it simple. Another reason for this decision is that, for most of the gif whose size is below 1MB, the MPI+OpenMP implementation could already achieve good performance. But for those whose size is greater than 1MB, even if using MPI+OpenMP could not bring significant improvement. Moreover, for small gif, having MPI, OpenMP and CUDA at the same will introduce a lot of operations, which might even increase the processing time.

We have implemented CUDA in the filtering process that take the longest time. Since the filter we apply can be decomposed in many simple mathematical operations, it is feasible to implement and execute these operations on GPU and achieve great performance improvements.

We use Multi-Kernel Model for CUDA. According to the Thread Hierarchy, memory is split into Grids and Blocks. Within a Block several threads are executing. Assume that we want one thread works for one pixel, then the total number of threads will be [number of images]*[width]*[height]. Then the number of grids should be number of threads divided by the number of block, while the number of block is determined by the maximum number of threads per block.

1.4 Image splitting

At this point, we noticed that our implementation showed great performance improvements for gifs with multiple images, but the improvements were not so great when the gif contained a single large image. We therefore focused on another level of data parallelism and decided to split the images into subparts, and distribute them across the slave nodes as we did previously with the multiple images of a gif. This is particularly relevant for gifs that contain a single image, but it can also be interesting for gifs that have less images than there are nodes available in the network.

However, while this would in the end allow us to use more of the computational power of the network, it also introduced new challenges. While the gray filter is applied to each pixel indepedently, the blur filter is a convolutional process that introduces a dependency from one cell to the next. This introduced the need for **ghost cells**. Since these ghost cells needed to be updated at each step of the blur filter, this introduced dependency between processes on different slave nodes. Another interdependency introduced is the fact that the verification of whether we have reached the threshold during the blurring process is now spread across multiple nodes.

We could solve this issue in one of two ways:

- We could split the blur pass task into sub tasks of one blur iteration each and use the master node scheduling to solve this task dependency issue
- Or we could introduce communications between slave nodes

We considered it a better practice in our case to communicate directly between nodes. We also chose to split the images vertically to make use of the use of adjacent memory cells in an array to make image splitting easier and more efficient. We believed that despite the fact that sometimes this led to having more ghost cells than vertical splits, it was beneficial in terms of robustness and efficiency.

Chapter 2

Performance analysis

It is important to consider multiple scenarios in our performance analysis:

- a gif with very low or very high resolution;
- a gif with a single image, a few, or many;
- a network with a single node, a few nodes, many nodes.

To perform these tests, the images we chose are as follow:

- giphy3.gif: a gif of 3 large images (5.9 MB total)
- Campusplan-Hausnr.gif: a gif of only 1 large image (1.3 MB total)
- fire.gif: a gif of 33 very small images (21 KB total)

2.1 Influence of the number of MPI ranks

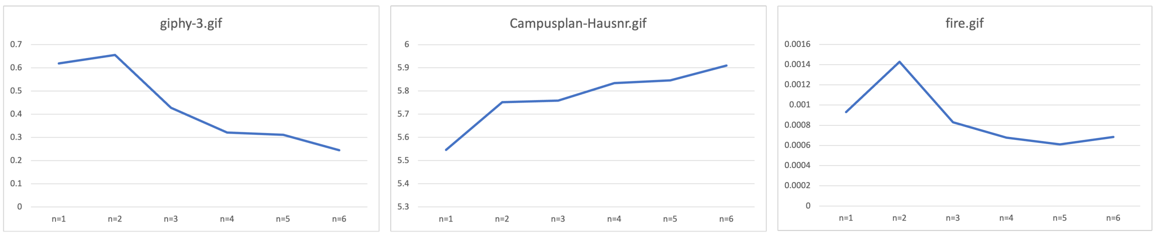

Figure 2.1: Time to process giphy3.gif (left), Campusplan-Hausnr.gif (center) and fire.gif (right) using MPI.

Note that in our implementation, n=1 MPI rank corresponds to the initial sequential code, and for n > 1 MPI ranks we have one master node in charge of scheduling and only n-1 slave nodes which perform the computations.

As can be expected, in every case going from n=1 to n=2 leads to an increase in execution time, because there is still only one node performing the computations, and the execution time also includes the time it takes to set up MPI communications and exchange the data between the master and the slave node. For Campusplan-Hausnr.gif, adding more nodes does not change anything, since splitting is not implemented and only one node can perform these computations.

However the execution time for giphy3.gif decreases as the number of slave nodes increases from 1 to 3 (= nb images), as the computations are more spread across the network. However, as we add even more active nodes, they are not used during the execution because there are no more images to distribute, and the execution time no longer improves.

Some improvement is also observed for fire.gif, but it is not very significant, as processing each image is almost as fast as sending its data through the network.

2.2 Influence of number of OpenMP threads

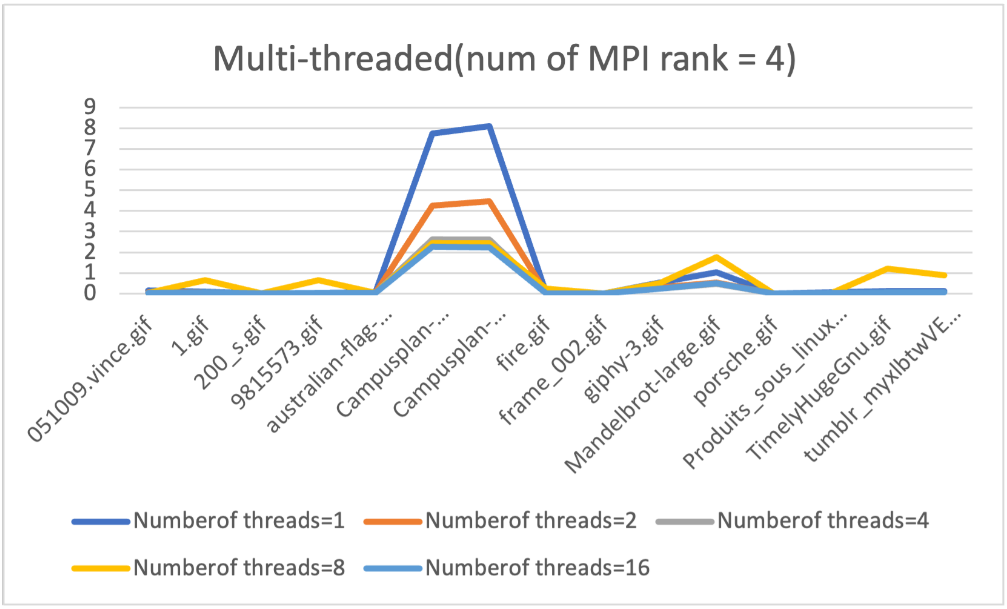

Figure 2.2: Influence of thread number

The figure 2.2 illustrates how the number of threads influences the runtime of the image filtering process. We note a consistent increase in performance as the number of threads increases (with an exception for threadnum=8 which we could not really explain). We only tested up to 16 as that was the maximum number of threads allowed per node in the network.

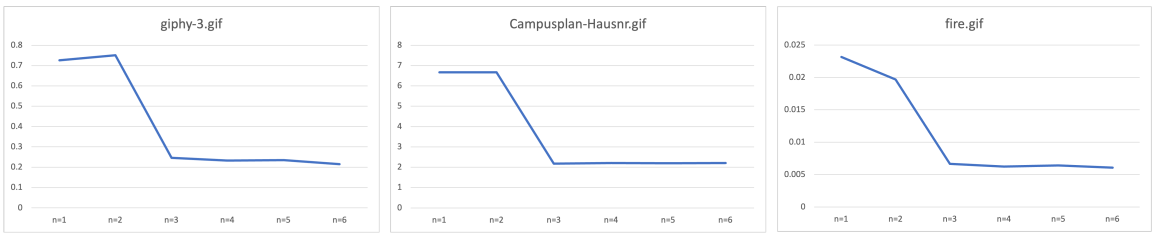

Figure 2.3: Time to process giphy3.gif (left), Campusplan-Hausnr.gif (center) and fire.gif (right) using MPI + OpenMP.

We can see that having multiple MPI ranks and OpenMP activated significantly increases the performance of the algorithm compared to the sequential code (n=1). Note that for n larger than 3 there are no longer significant improvements in execution time for any of the three gifs, which is coherent with what we explained in the previous section.

2.3 Influence of CUDA

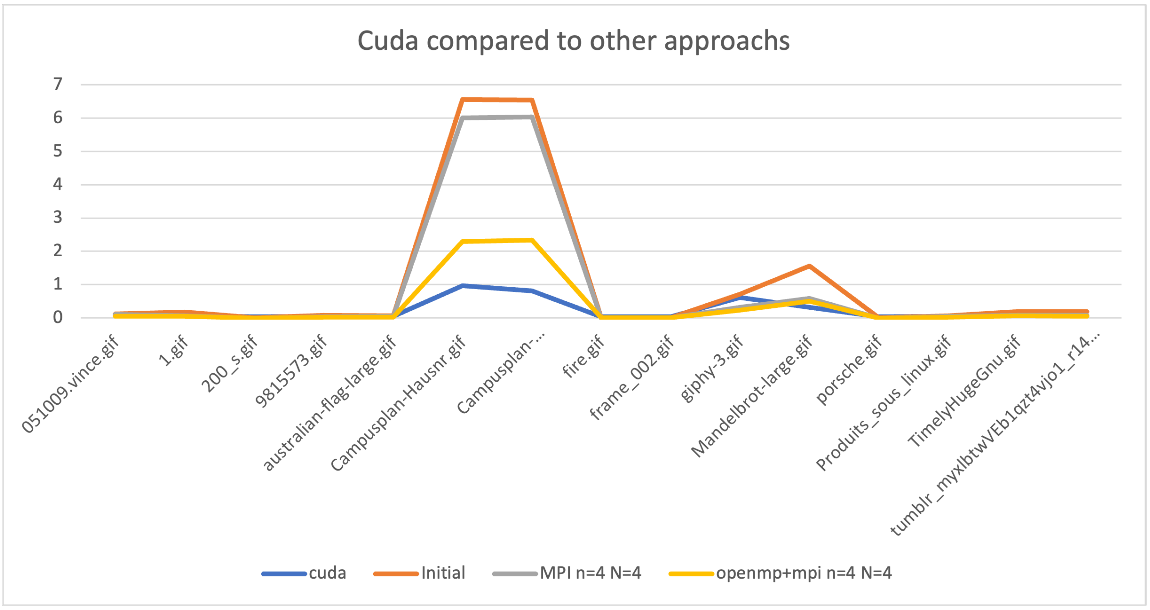

Figure 2.4: Compare CUDA and other approaches in total execution time

We note here that using intra-node parallelism (OpenMP or CUDA) is particularly efficient for gifs with large images, like Campusplan-Mobilitaetsbeschraenkte.gif or Mandelbrot-large.gif. This is particularly true in comparison to pure MPI, which cannot exploit the full capacities of the network in the absence of image splitting. The figure 2.5 below also illustrates the fact MPI brings significant improvements for gifs with multiple images (like Mandelbrot-large.gif and giphy-3.gif), but not for gifs with a single image.

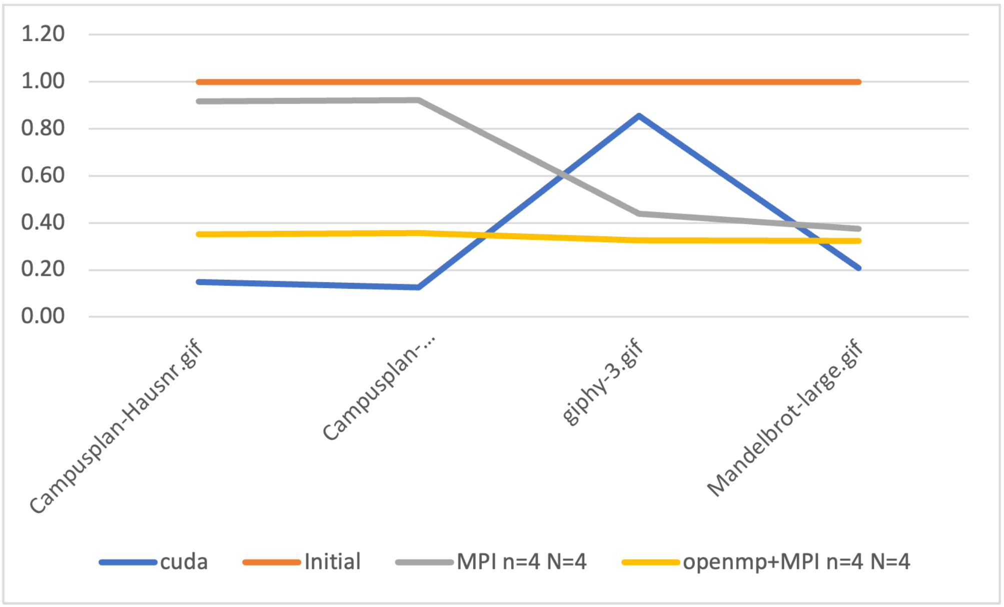

Figure 2.5: Compare CUDA and other approaches for gif with few images (rescaled in proportion to the runtime of the initial sequential code)

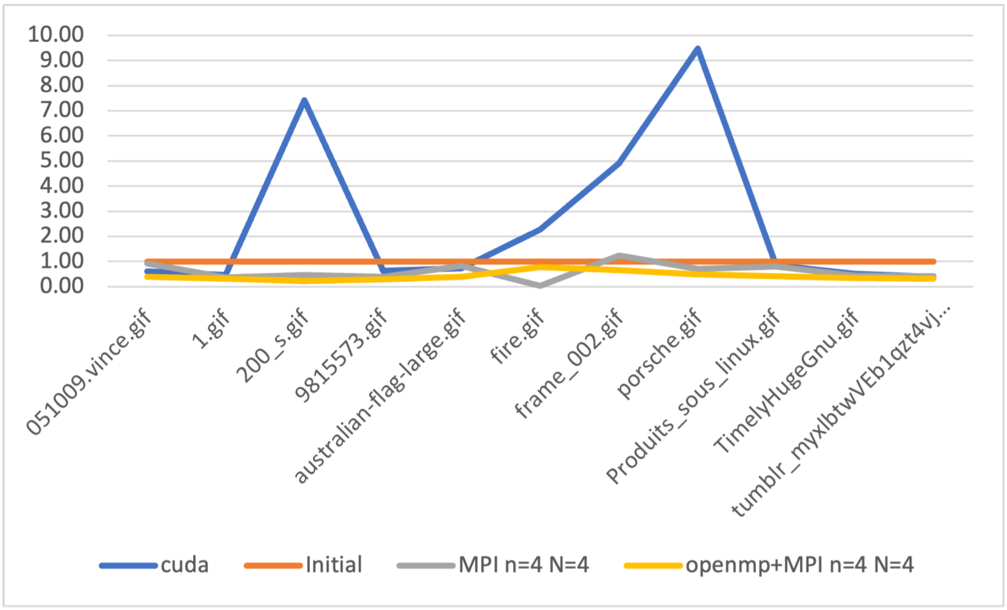

Figure 2.6: Compare CUDA and other approaches for gif with more than 4 images (rescaled in proportion to the runtime of the initial sequential code)

In the figure 2.6 above, we note that in the case of gifs with many smaller images, the OpenMP + MPI paradigm consistently brings a significant runtime improvement. However while pure CUDA can sometimes yield big improvements, in some cases this implementation is much slower. This is because of the latency of communications between the CPU and the GPU which can sometimes be longer than the improvement that this additional level of parallelism can bring for small images.

Chapter 3

Further improvements

One way we could improve our performances with MPI would be to also exploit the computational power of the master node. Currently if for example we have two active nodes, only the slave node does the work, and the master node is just overseeing scheduling. But if we were able to have the master node do both its scheduling task and to contribute to the image filtering process computations, we could significantly increase our performances with MPI for a low number of MPI ranks. For larger networks, the master node would be pretty busy handling the scheduling and could probably not afford to also apply filters itself.

Another way to improve the efficiency of the MPI approach would be to reduce the number of communications across the system and their dimensions. This would significantly reduce the delay caused by the limited bandwidth in the case of large images and improve the execution time. We can reduce amount of data communicated between nodes by:

- Loading the gif in each node and performing the final export directly
- Optimizing scheduling so that a node is reassigned to the same image for the next filtering pass

Choosing nodes optimally based on real-life architecture and physical distances between nodes (latency and bandwidth) could also bring slight improvements to our performances.

Choosing parameters (nb of threads, nb of MPI ranks, use CUDA or not) dynamically based on dimensions of the input gif and the structure of the network would also be ideal but it goes far beyond the scope of our project. In this study we only focused on the characteristics of the network at Polytechnique and the performance of our algorithms in this system.