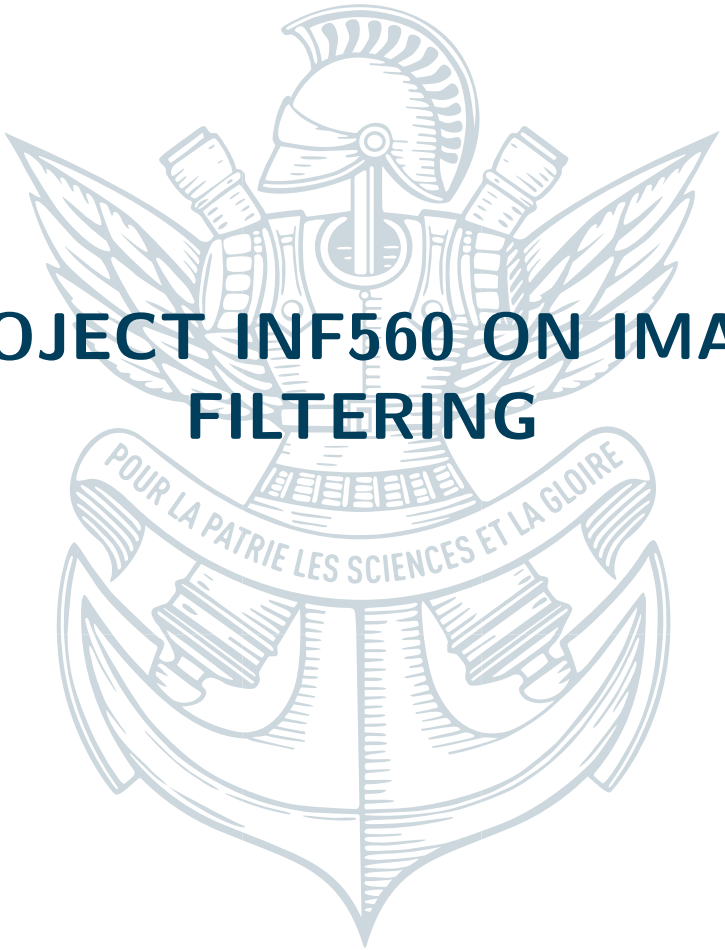



# PROJECT INF560 ON IMAGE FILTERING

12 March 2022

Manas UTESHEV
Vladimir OGORODNIKOV

# Contents

**1 Parallelism approaches and choices** **1**
1.1 Pure MPI version . . . 1
1.2 OpenMP implementation . . . 1
1.3 Striping . . . 1
1.4 I/O optimization . . . 2
1.5 CUDA implementation . . . 2

**2 Algorithms** **2**
2.1 Legacy MPI . . . 2
2.2 Striping MPI . . . 3

**3 Experimental evaluations** **4**
3.1 Before IO optimization . . . 4
3.2 After IO optimization . . . 5
3.3 Multithreading . . . 7
3.4 CUDA . . . 8

# 1 Parallelism approaches and choices

In this section we are going to present the following steps, which we have taken while developing this project on image filtering.

## 1.1 Pure MPI version

In our first step, we have decided to implement pure MPI version of our project. Here is the following idea: one of the MPI processes (nodes), which has rank 0 among all processes, plays a role of a scheduler and runs a scheduling loop, in which it distributes images among the slave processes, and after that collects the processed outputs from slave processes by using asynchronous MPI calls (*ISend, IRecv, etc*). In this setting, after having received an image from the master node the slave process will run sequentially filters on this image and return to the master process. Moreover, slave nodes will wait for the tasks from master process and in case they receive a signal to stop working from the master node, they will just shut down.

## 1.2 OpenMP implementation

In our second step, we have decided to exploit OpenMP parallelism in our project. In this version, our loop iterations in filter functions are parallelized by using *pragma omp for* and thus will be distributed among available OpenMP threads. In order to decrease number of barriers we have added *nowait* options for loop iterations, which do not interleave among themselves, and *reduction* option in our *for* loop in order to avoid considerable amount of critical sections for writing value in a boolean shared variable *end* of the blur filter function.

## 1.3 Striping

Until this moment we have worked with our first version of MPI+OpenMP algorithm. This algorithm has two following problems of efficiency: firstly, if we just have only one single image in our gif file, we do not exploit all available MPI processes and only one MPI node will process this image, and secondly, if we have just only two MPI processes, we use only 50% of our computational capabilities, because one of the nodes will work only as a scheduler. That is why we have decided to add another algorithm in our project, which is called *striping*. In this algorithm each image will be divided into roughly equal stripes and these stripes will be distributed among available MPI nodes. Furthermore, because of the convolutional nature of the filters applied to the images, we have to exchange pixels on the border of stripes between neighbours. Moreover, for the blur filter we have to exchange the value of *end* variable between slave nodes and master node.

Our MPI code in the project will function in two modes: *legacy* and *striping*. *Legacy* mode will function in the same way as explained in the section 1.1 and *Striping* mode has already been explained in this section. We choose one of these modes according to the number of images and MPI processes.

### 1.4 I/O optimization

While we were performing tests, we have noticed that the most significant fraction of the computational time was due to the network communication between MPI nodes. The problem is that the size of images is sometimes quite large and we have to transfer gigabytes of data in the network. That is why we have decided to use gif compression and decompression and temporary files in order to reduce the amount of the transferred data in the network. Henceforth, in this new setting each MPI node will read the input image by itself and after the processing is done, the slave nodes will store the result into temporary files and finally, the master process will merge these temporary files into one resulting file.

### 1.5 CUDA implementation

While we were performing profiling of our program, we have noticed that the majority of processing time dedicated to filter functions belongs to the blur filter function. That is why we have decided to implement the blur filter function using CUDA. Given the fact that in the OpenMP version of the blur filter function we have several *for* loops that can be run independently, we use CUDA streams in order to avoid unnecessary synchronization. Moreover, we have to compute the logical *end* among all the pixels of image and to do so we exploit multi-block reduction technique.

## 2 Algorithms

### 2.1 Legacy MPI

In legacy MPI mode master and slave processes will function in different ways. Here you can find two pseudo-codes for master and slave processes.

```
def master():
    for i in range(1, world_size - 1):
        Irecv(signal[i], i, req[i])

    while processed_images < N:
        i = Waitany(req)
        if signal[i] >= 0:
                processed[i] = True
                processed_images += 1
        if sent_images < N:
                Send(sent_images, i)
                Irecv(signal[i], i, req[i])
                sent_images += 1
        else:
                Send(-1, i)
                req[i] = MPI_REQUEST_NULL
```

```
    for i in range(1, world_size - 1):
        if req[i] != MPI_REQUEST_NULL:
                Wait(req[i])
                Send(-1, i)

def slave():
        Send(-1, 0)
        while True:
                i = Recv(0)
                if i < 0:
                        break
                process(i)
                Send(i, 0)
```

## 2.2 Striping MPI

In order to implement striping MPI mode in our project we have defined a structure called *striping_info*, which will contain information about striping of an image. It is implemented in a following way:

```
typedef struct {
    int min_row;
    int max_row;
    int single_mode;
    int top_neighbour_id;
    int bottom_neighbour_id;
    int stripe_count;
} striping_info;
```

The field *single_mode* above determines whether we have only one stripe.

As for blur filter we have to exchange pixels on the border of stripes between neighbours, we synchronize rows in our *while* loop and also synchronize the boolean variable *end* between stripes. Here is the pseudo-code for blur filter function which functions for two MPI modes:

```
def blur_filter():
        while True:
                end = 1
                if not s_info.single_mode:
                        synchronize_rows(s_info)

                # OpenMP stuff, setting up     end

                if not s_info.single_mode:
                        end = synchronize_bool_and(s_info, end)
```

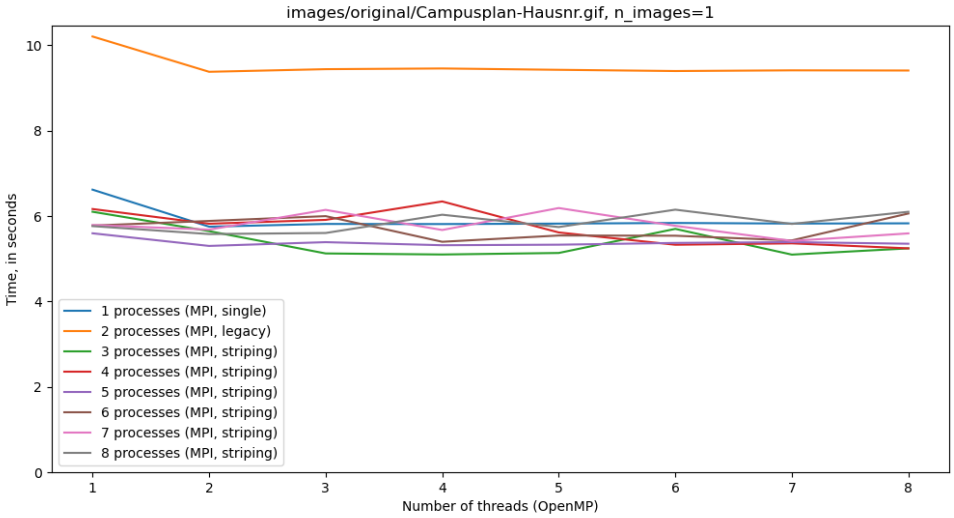
# 3 Experimental evaluations

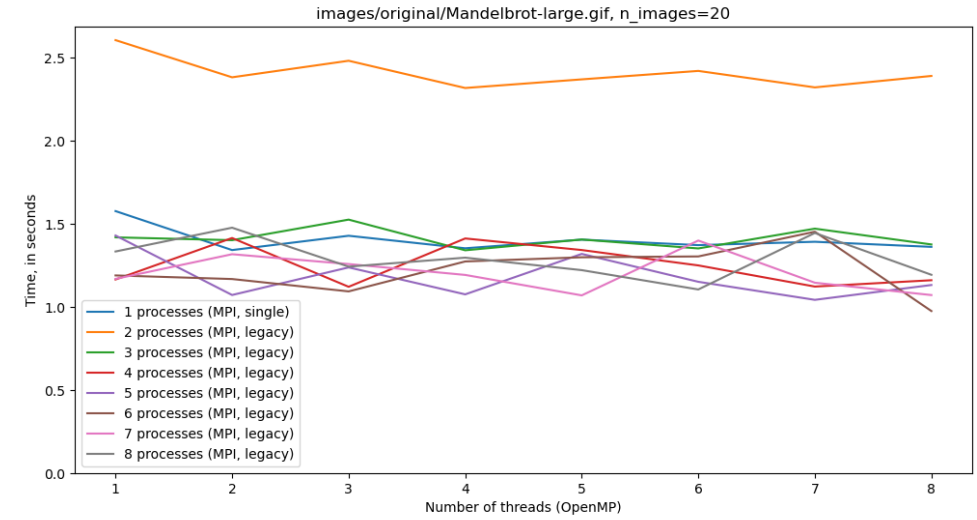
On the following graphs you can see the dependency of computational time on number of threads (along the X axis) and number of MPI processes (line color, see legend). Moreover, our decision policy, tha determines which MPI mode (Striping or Legacy) we will use, is based on the following selection criteria: if the number of MPI processes without the master node is smaller than the number of images and is at least 3, then the legacy mode is used, and the striping mode is used otherwise.

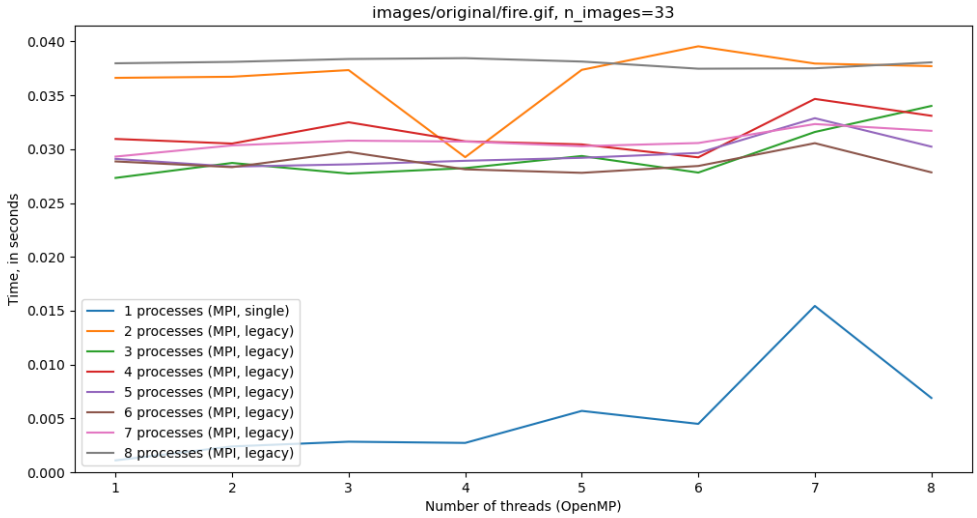
Furthermore, some of the images can be easily processed and the processing time becomes quite negligible even for the sequential version of our program making it quite difficult to see clearly the advantage of MPI+OpenMP implementation.

## 3.1 Before IO optimization

In this section the following graphs describe the algorithm, which does not use I/O optimization that has been explained in the section 1.4. In spite of using OpenMP and MPI in our program we do not see a considerable decrease in the processing time. That is due to the large number of MPI communications in the network. In some cases the pure sequential version of our program runs much faster than the MPI+OpenMP version.

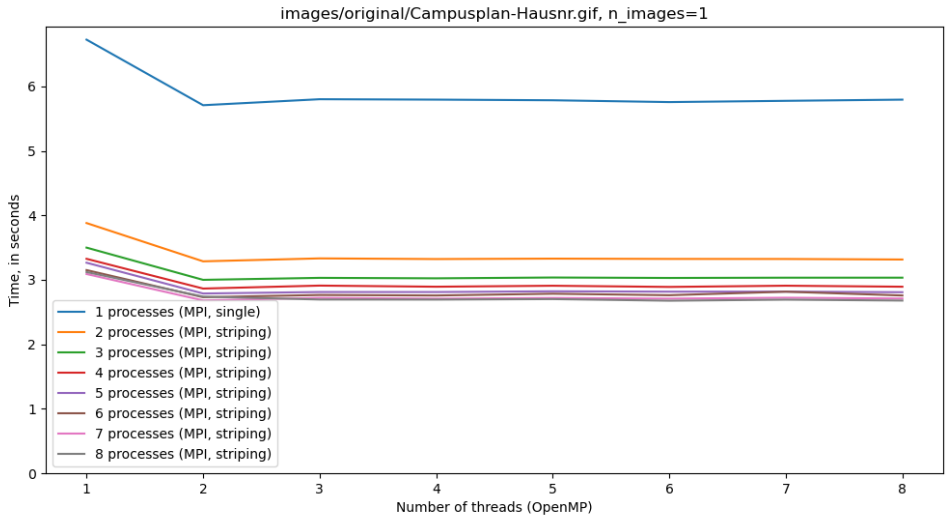


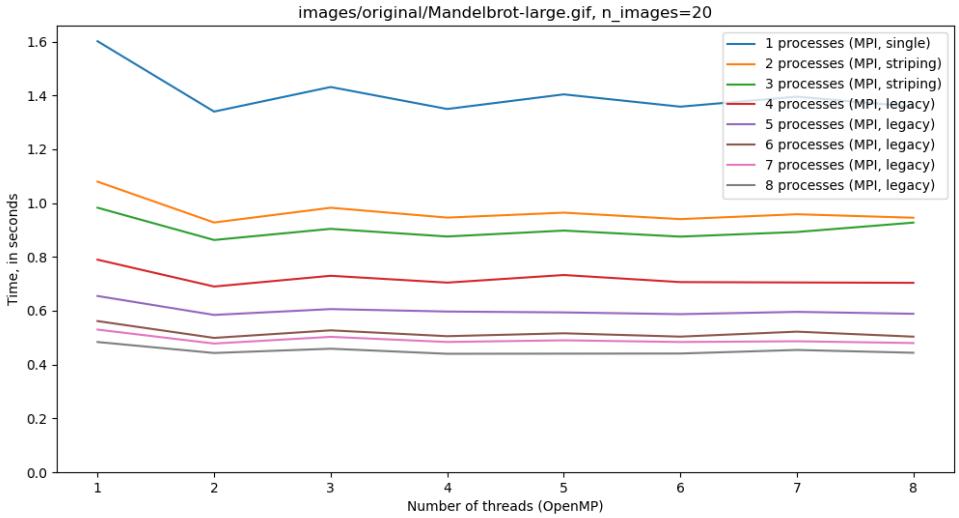


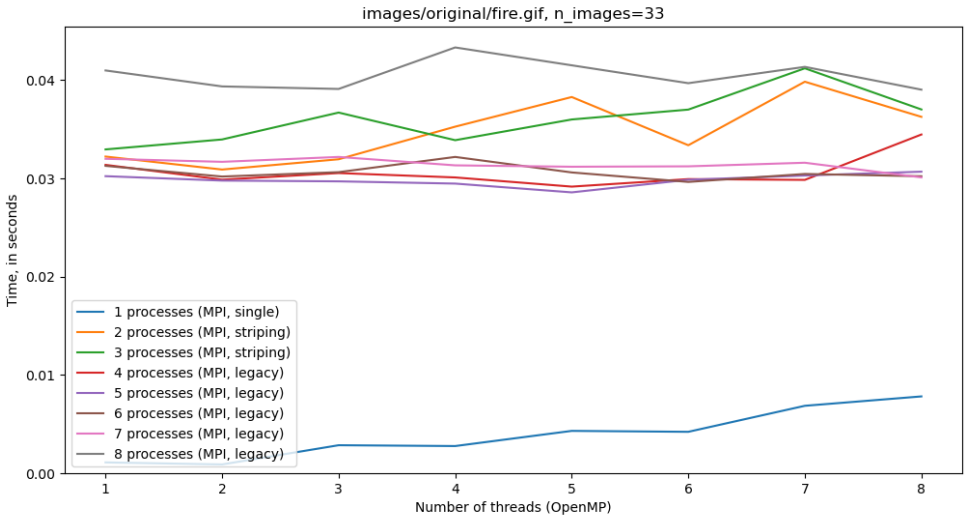




## 3.2 After IO optimization

After applying IO optimisations we see that the time of processing of huge files (`Campusplan-Hausnr.gif`) has been decreased dramatically. It seems to be roughly proportional to the inversed number of processes. However, overhead is still significant for small images. Also one can notice that striping mode can be worse than legacy mode because of additional synchronisation between neighbour stripes.

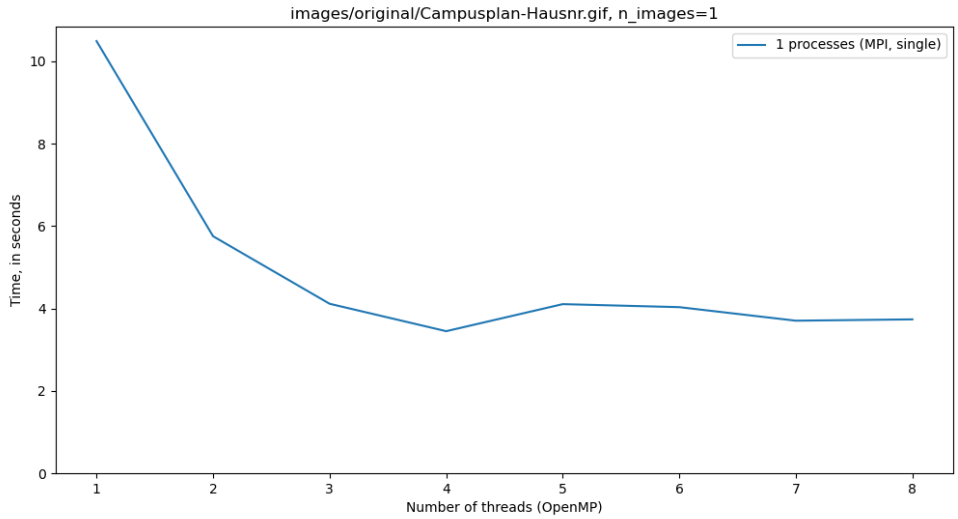

images/original/Campusplan-Hausnr.gif, n_images=1
Time, in seconds
Number of threads (OpenMP)
1 processes (MPI, single)
2 processes (MPI, striping)
3 processes (MPI, striping)
4 processes (MPI, striping)
5 processes (MPI, striping)
6 processes (MPI, striping)
7 processes (MPI, striping)
8 processes (MPI, striping)

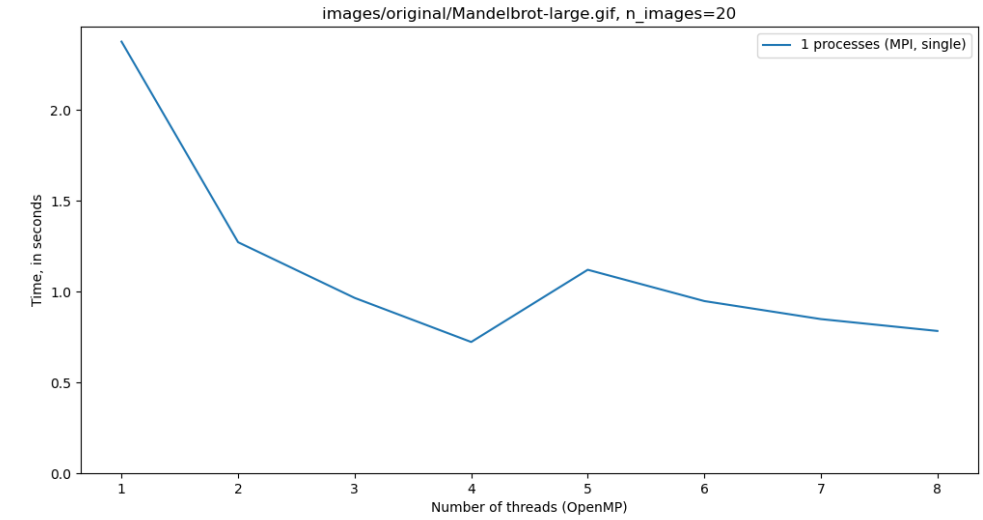

images/original/Mandelbrot-large.gif, n_images=20
Time, in seconds
Number of threads (OpenMP)
1 processes (MPI, single)
2 processes (MPI, striping)
3 processes (MPI, striping)
4 processes (MPI, legacy)
5 processes (MPI, legacy)
6 processes (MPI, legacy)
7 processes (MPI, legacy)
8 processes (MPI, legacy)

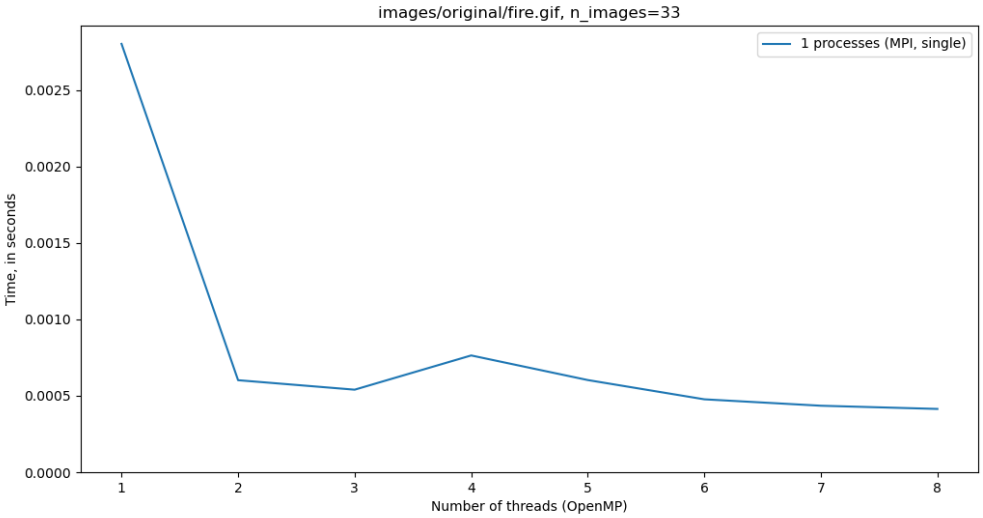



## 3.3 Multithreading

On the previous graphs you can notice that the dependency of computational time on number of OpenMP threads is very weak. We compiled our code on our laptop and ran it locally with one MPI process. The results appeared to be close to theoretical ideal (time $\propto$ inversed number of threads), with respect to the number of cores (8, but actually 4 + Intel Hyper-Threading). So either the problem is in the old compiler in the cluster, or it is in the high load of cluster nodes and moderate scheduling policy of SLURM.

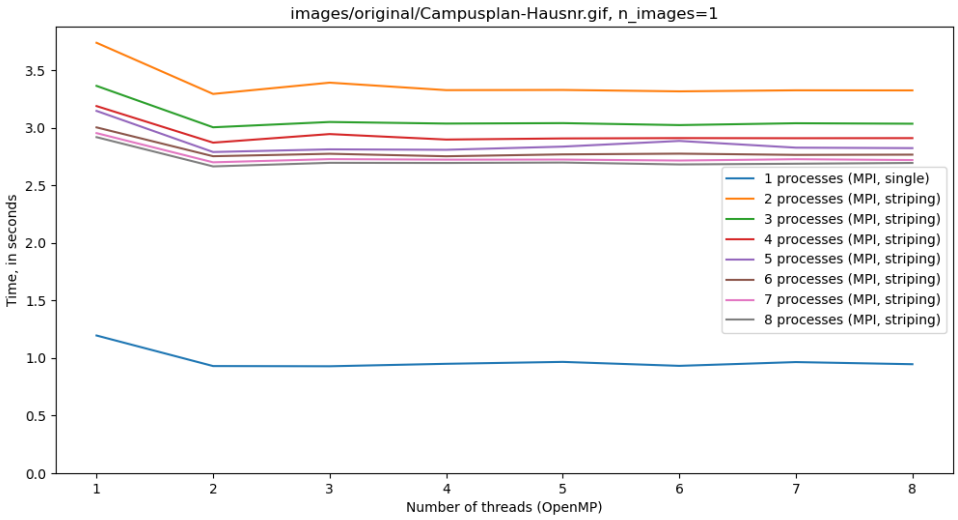


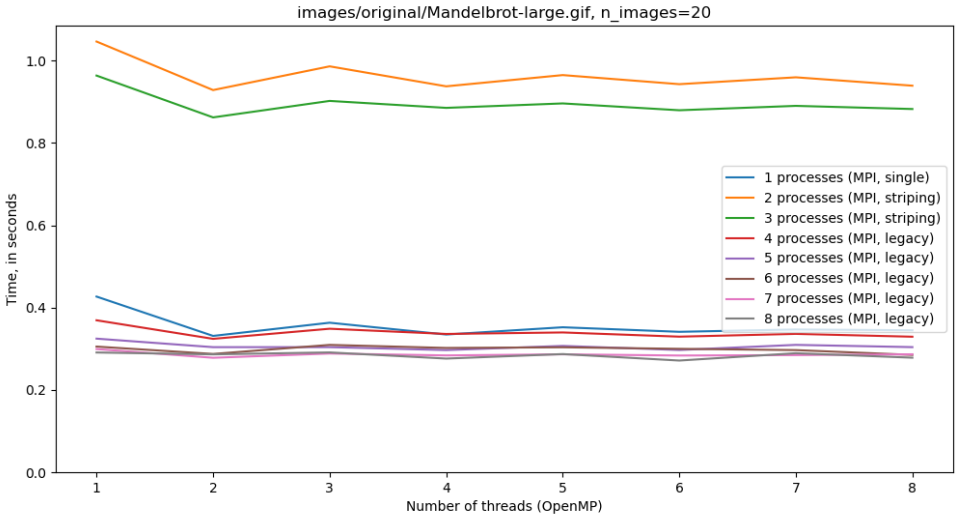


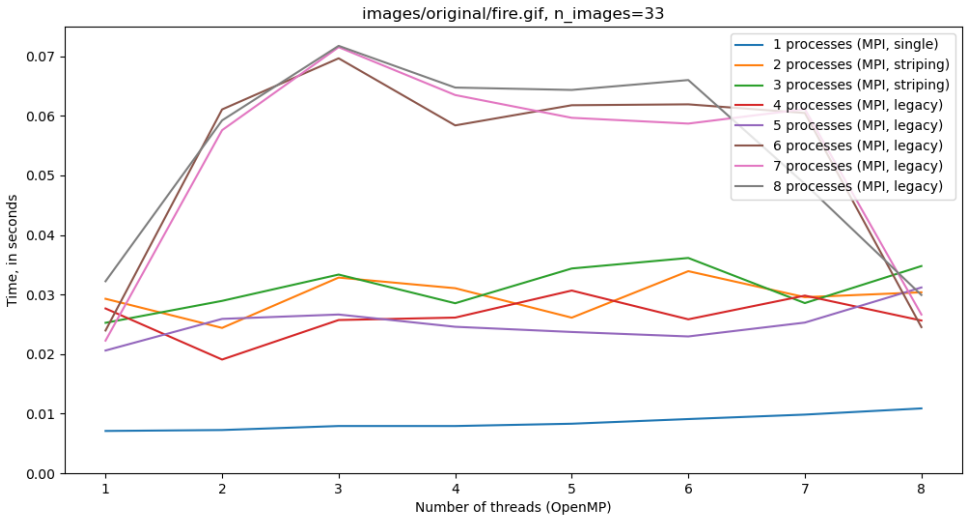




## 3.4 CUDA

We have implemented CUDA version of blur filter and combined it with our legacy MPI mode. It can be seen on the graph that the computational time is decreased in comparison with previous versions. It is also useful to note that in the setting when we have only one MPI process and one OpenMP thread the program still uses CUDA.


images/original/Campusplan-Hausnr.gif, n_images=1
Time, in seconds
Number of threads (OpenMP)
1 processes (MPI, single)
2 processes (MPI, striping)
3 processes (MPI, striping)
4 processes (MPI, striping)
5 processes (MPI, striping)
6 processes (MPI, striping)
7 processes (MPI, striping)
8 processes (MPI, striping)


images/original/Mandelbrot-large.gif, n_images=20
Time, in seconds
Number of threads (OpenMP)
1 processes (MPI, single)
2 processes (MPI, striping)
3 processes (MPI, striping)
4 processes (MPI, legacy)
5 processes (MPI, legacy)
6 processes (MPI, legacy)
7 processes (MPI, legacy)
8 processes (MPI, legacy)


images/original/fire.gif, n_images=33
1 processes (MPI, single)
2 processes (MPI, striping)
3 processes (MPI, striping)
4 processes (MPI, legacy)
5 processes (MPI, legacy)
6 processes (MPI, legacy)
7 processes (MPI, legacy)
8 processes (MPI, legacy)
Time, in seconds
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
1
2
3
4
5
6
7
8
Number of threads (OpenMP)