Concurrent Programming Engineering Project report

E7650 Valentin SERVIERES

1 Task analysis

1.1 Problem being solved

We have a sequence of images, it could be images from a movie in production. We want to have the average color of that image sequence.
It can be useful to know what color leads the image and then apply colorimetry correction on it.
In movie production, we might need with high resolution images, and a lot of images. So we need to get these calculations to be quick. However, computers have limited performences.
Thus, we can use cluster of computers to divide the work among several computers.

1.2 How concurrency is used

This is where our data concurrency is used. We give to each core of the processor of each computer on the network a list of images to process in order to get their average color.

1.3 Language/tools used

The language used is **C++** in order to get the best performences. The data concurrency is done using **MPI**.
The MPI tools used are **Scatter** to distribute the work to all computers and **Gather** to get the average values back.

2 Testing the program

2.1 Installation

You can download the full project with all the datasets here : `https://github.com/MagicTINTIN/MPIFarm`.

2.1.1 Dependencies

You will need to install the following dependencies:

```
bash $ sudo apt install g++ cmake openssh-server libopenmpi-dev openmpi-bin libopencv-dev
```

2.1.2 Compile

To compile the program, execute:

```
bash $ ./cmakecompile –release
```

2.1.3 Configure cluster (optionnal)

To configure the cluster you might need to configure a passwordless ssh. To do so, execute these commands between all worker-manager node:

```
bash $ ssh-keygen -t rsa -b 4096
bash $ ssh-copy-id username@remote-machine
```

Then, modify on each computer, modify the ssh deamon config:

```
bash $ sudo nvim /etc/ssh/sshd_config
```

You can also use nano, or whatever text editor:

```
PubkeyAuthentication yes
PasswordAuthentication no
ChallengeResponseAuthentication no
```

Finally, you can restart the daemon.

```
bash $ sudo systemctl restart ssh
```

2.2 Start the program

To execute the program, the simplest way is to use the following bash script:

```
bash $ ./start single|mega|custom your_image_set.json [number_of_processes_to_spawn]
```

2.3 Test the program

2.3.1 Generate a sequence file

In order to easily generate a json file which contains the image sequence, you can use the following program and just follow the instructions.

```
bash $ ./jsonGenerator.o
```

To compile this program you only need to execute:

```
bash $ ./jsonCompileGenerator.sh
```

2.3.2 Calculated value

To test the program, I created images with Krita for the specific images (in the folder otherImages/), and for the rainbow/ image sequence I used Blender (these two software are open source).
To test that the average color is the true one, I created otherImages/check1.jpg with only one solid color #ee7700.
Then, we just need to execute the program:

```
bash $ ./start.sh single imageSets/P170B328_ServieresV_L3_check.json 1
Global average color: #ee7700, R:238 G:119 B: 0
```

2.3.3 Start a benchmark

You can easily start a benchmark using the following script.

```
bash $ ./benchmark.sh
```

You can modify the first variable to choose the settings of your benchmark.
The results will be exported in benchmark.csv and benchmarkfull.csv.

3 Performance analysis

3.1 Datasets used

Here is the list of the datasets used in the benchmark. imageSets/P170B328_ServieresV_L3_smallest.json
imageSets/P170B328_ServieresV_L3_xxsmall.json
imageSets/P170B328_ServieresV_L3_0001-0003.json
imageSets/P170B328_ServieresV_L3_0001-0012.json
imageSets/P170B328_ServieresV_L3_0001-0024.json
imageSets/P170B328_ServieresV_L3_0001-0064.json
imageSets/P170B328_ServieresV_L3_0001-0128.json
imageSets/P170B328_ServieresV_L3_0001-0512.json
imageSets/P170B328_ServieresV_L3_0001-1024.json
imageSets/P170B328_ServieresV_L3_0001-2500.json

The two first datasets contains images of 9 pixels ($3px \times 3px$). The first one only contains 1 image, and the second contains 19 images.
It only took 0.0431879 seconds to a single thread to get the average value.

The other sets contain images in 4K (3840×2160). The smaller set contains 3 images, and the bigger one contains 2500 images.
It takes 151.636 seconds to a single thread to compute the average color of the latter sequence.

3.2 Benchmark results

This benchmark has been executed on a cluster of 3 computers:
- Manager : i7-7500U (4 cores) @ 3.5GHz with 7840 MiB of RAM
- Worker1 : Celeron (4 cores) @ 2.6GHz with 3259 MiB of RAM
- Worker2 : Celeron (4 cores) @ 2.2GHz with 3724 MiB of RAM

The operating system is Ubuntu 20.04.6 LTS x86_64 on each computer. Computers were connected between them by Wi-Fi network.

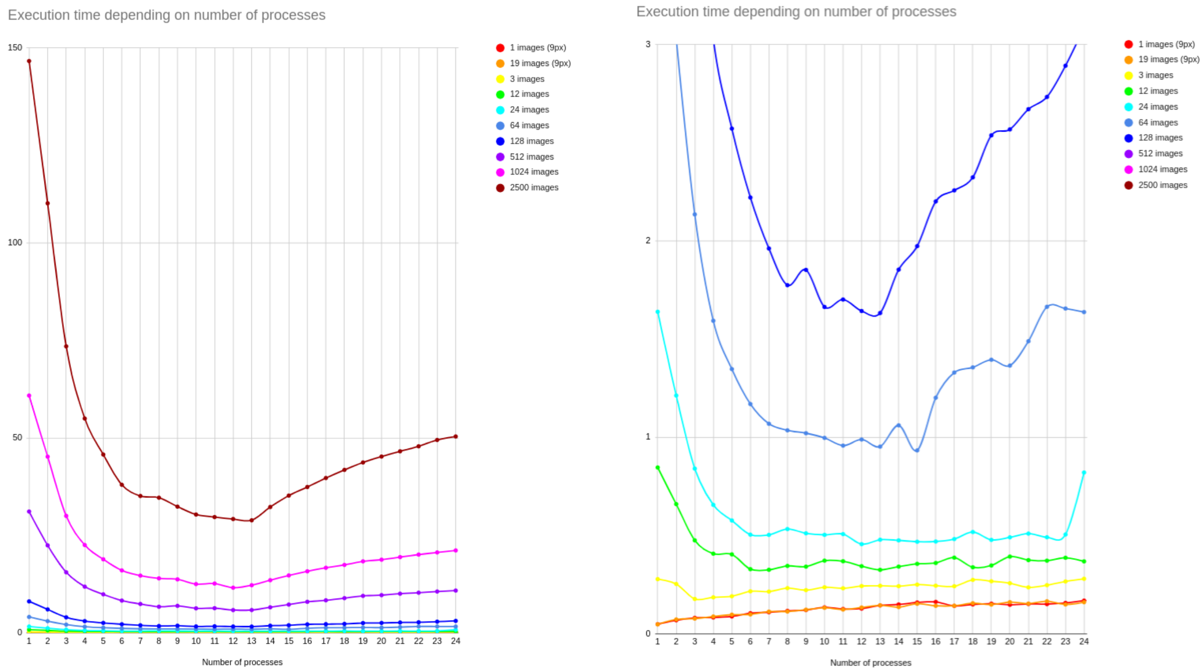

Figure 1: Results from benchmark

In these graphs, we plotted the calculations' duration (in seconds) depending on the number of processes available. We tested it for each dataset.
The values here are average values, each configuration has been tested 6 times.
You can see below the exact values for each test.

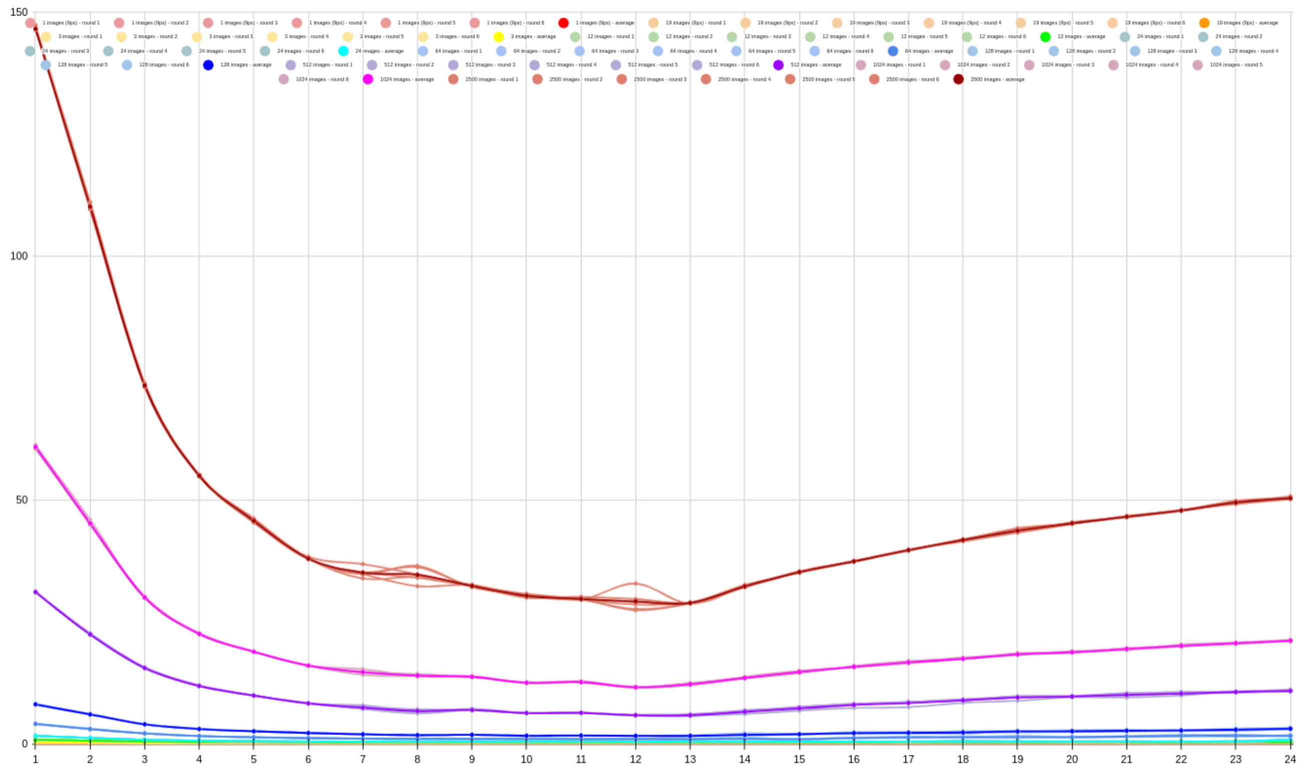

Figure 2: Results from benchmark

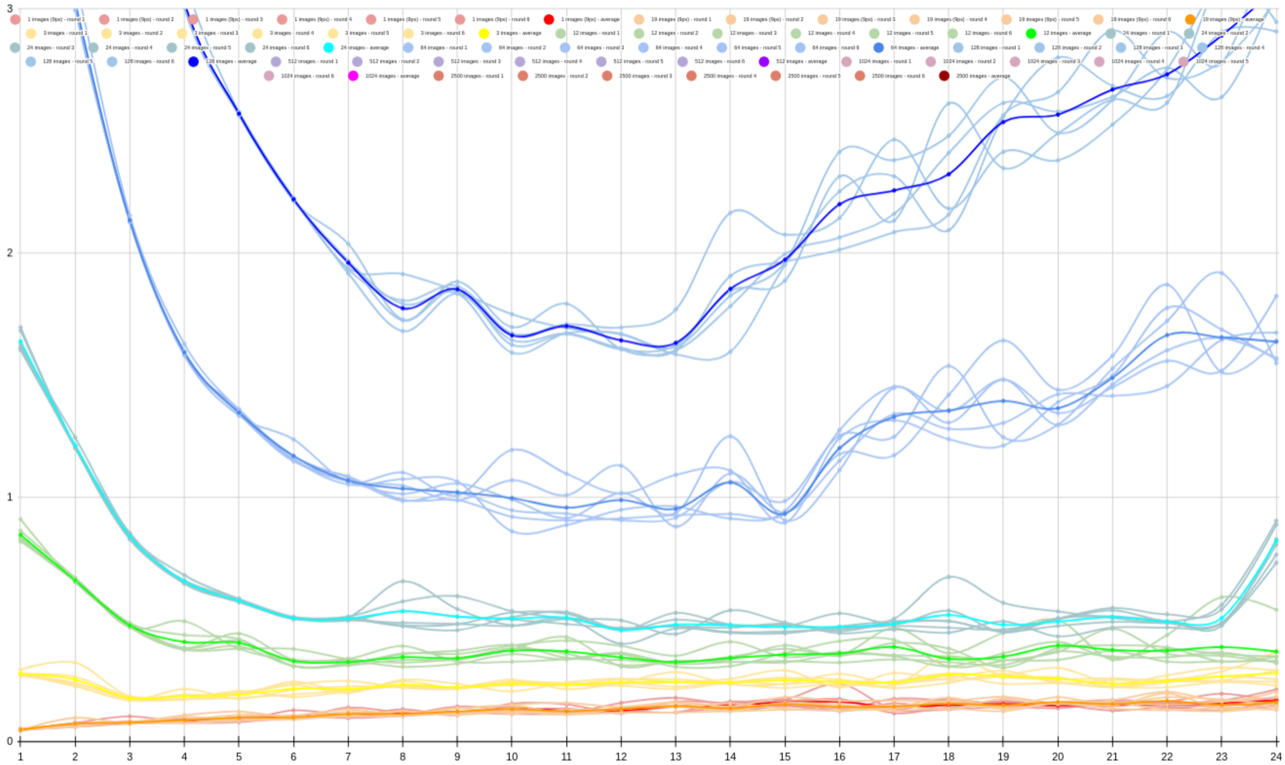

Figure 3: Results from benchmark

As we can see if the calculations are too simple, it is better to spawn only one process. For bigger sets, which need more calculations we can see that the best number of processes is around 12.

3.3 Explanations

MPI introduce a new latency, the network. Indeed, to communicate information with the other computers of the cluster, MPI needs to send requests which take more time than just executed the calculations on a single thread without communicating anything.
So if the dataset is too small (< number of cores of the cluster), and/or if the images of the set are too small (only few pixels), it is quicker to use a single thread.

For bigger datasets, we can see that the minimum of calculations' duration is around 12.
This is normal because it is the total number of cores of the cluster ($4 + 4 + 4 = 12$).
If we start fewer processes, then we do not use the processors at 100% of their capacity, so it will be less efficient.
Otherwise, if we start more processes, then we don't have "perfect" parallelism. Processors of each computer have to do context switching between all processes as there are more active processes than cores available to run them. This slows down calculations.

4 Conclusion

Time elapsed: 151.636s
Time elapsed: 74.8672s
Time elapsed: 23.531s