Video motion detection: Analysis Document

Paolo Fasano
p.fasano1@studenti.unipi.it

July 11, 2022

UNIVERSITÀ DI PISA

Master's degree in Computer science – University of Pisa
PARALLEL AND DISTRIBUTED SYSTEMS: PARADIGMS AND MODELS

1 Abstract

This document presents the Motion Detection project, which can be found at `https://github.com/PFasano99/SPR_Project_Unipi`, illustrating a sequential solution, a simple threads solution and a Fast Flow solution. The various paragraphs show how the sequential solution has been implemented and the thoughts behind the implementation process for the parallel system.
The final paragraph contains the instructions to build the code.

Contents

1 Abstract **1**

2 Introduction **3**
2.1 Aim of the system 3
2.2 Success criteria 3

3 Problem Analysis **4**
3.1 Data loading phase 4
3.2 Preprocessing phase 4
3.2.1 Gray-scaleing 4
3.2.2 Smoothing 5
3.3 Frames Comparison Phase 5

4 Sequential code Analysis **5**

5 Proposed Parallel System **6**

6 Parallel code Analysis **7**

7 Conclusion **9**

8 Deployment **10**

9 References **10**

2 Introduction

The following document purpose is to analyze the Video motion detect (AV) problem for the parallel and distributed systems: paradigms and models exam project.
This document aims to:

- analyze and understand the problem;
- define a sequential solution;
- design a parallel solution using native C++ threads;
- design a parallel solution using FastFlow;

2.1 Aim of the system

The aim of the system is to create a simple motion detection software based on the idea that if a certain percentage of pixels change from a frame to the next than some motion has been detected. The final output of the system should be the number of frames in which motion has been detected.
The different steps needed to build this system are:

- Load a video into memory (data loading phase)
- Divide the video into frames (data loading phase)
- Gray-scale and smooth all the frames (preprocessing phase)
- Pick the first frame as background
- Compare the background with the frame
- Keep track of the number of frames changed

All these steps will be analyzed in the "Problem analysis" section and the necessary operation to compute them, and a draft of the different possibilities to solve them, will be presented. Furthermore some of the steps will be divided in even smaller problems and an exploration for both parallel and sequential will be presented.
The sequential version of the solution will than be used to create a baseline for comparison against the naive threads and FastFlow solutions.

2.2 Success criteria

This application will be deemed successful if the implementation of all the above steps will be done: in an efficient way and if all the problems presented in the problem analysis will be solved properly both in parallel and in sequential programming.

3 Problem Analysis

In this section we are going to discuss the different steps designed to reach a solution.

3.1 Data loading phase

The first step to create a video motion detect software is to load a video and divide it into frames to analyze the different images. To do such operations the openCV library will be used. Many methods are provided in the openCV library that can help in the handling of image and thus videos. OpenCv could be used to further preprocessing the frames but due to the nature of this project and the aim to evaluate a sequential architecture against a parallel one, all the preprocessing methods will be implemented from ground up. The data loading phase will be managed only by the opencv class, using the VideoCapture method, this step will only be implemented in sequential.

3.2 Preprocessing phase

The second step is to preprocess all the frames, the preprocessing requires two different methods to be developed: the first method is the gray-scale which aim is to change the colours of the picture into only shades of gray; the second method needed is a smoothing method which scope is to clear noise off the pictures to have a "cleaner" picture to analyze. Both these methods must be applied on all the frames for the video, considering that standard video formats usually use between 24 and 60 frame per seconds, it means that for a 10 seconds video we should grayscale and smooth between 240 and 600 frames. The very high number of frames to preprocess would probably require a really long time in a sequential environment, therefore the need of a parallelized version.
For very long videos to have a better performance we could decide to analyze only part of the frames for each second trading off the precision of the detection in favour of the time needed.

3.2.1 Gray-scaleing

Different gray-scaling strategies are possible, in our project the one applied is a simple average for each pixel; this method consists in averaging the RGB values for each pixel. The result of this method is a image(frame) "coloured" only in black-white-gray shades.
Sequential: In the sequential implementation of the grayscale we simply iterate over each pixel, for each frame, and make a mean of the Red, Green and Blue value.
Parallel: The parallel implementation will implement the same sequence of steps as the sequential version of gray-scaling but utilizing two different techniques: the threads and the FastFlow; The parallel implementation will be discussed more in details in the section "Proposed parallel system".

3.2.2 Smoothing

By smoothing an image we are able to reduce the amount of high frequency content/detail, such as noise and edges; this process produces a "simpler" image that is simpler to analyze and compare. In this project the method implemented is a Gaussian blur also known as Gaussian smoothing. In this technique, an image should be convoluted with a Gaussian weighted kernel to produce the smoothed image.
Sequential: In the sequential implementation of the Gaussian smoothing we create a kernel matrix and apply the convolution over the matrix that represent the frame for each frame of a video.
Parallel: The parallel implementation will implement the same sequence of steps as the sequential version for smoothing but utilizing two different parallelization techniques: the threads and the FastFlow; The parallel implementation will be discussed more in details in the section "Proposed parallel system".

3.3 Frames Comparison Phase

The final step for the creation of a Motion Detection model is to compare a base frame, from now on defined as background, with the other frames. The starting frame used as background will be the first frame of the video. In general the background frame will be compared with a frame and this comparison can lead to two different results:

- if the background and the frame are equal than nothing happens;
- if the background has a number of different pixels, that is higher than a certain degree, from the frame f some steps are taken:
    1. A counter, which keep count of frames with motion detected, get updated;
    2. The background frame gets updated with the frame f

This steps get repeated for each frame of our video. When all the frames are analyzed the number of frames with motion detection is printed.

4 Sequential code Analysis

The sequential solution proposed in the previous paragraph has been implemented using four methods:

- loadVideo, which receives a path to a video and returns the video divided into frames;
- grayScale, which given a frame in Mat format returns the same frame but in black and white;

- gaussianFilter, which using a kernelMatrix and a frame, returns the Gaussian blur of that frame;
- compareFrame, which takes two frames and returns the percentage of different pixels between the frames;

(a) Starting frame (b) Grayscaling (c) Gaussian Blur

Figure 1: Example of change of a frame throw the preprocessing phase

To test the sequential approach video of different lenght have been used:

Video in seconds	8	59	90	120
Frames	175	1771	2290	3600
Loading video	6.1631	51.1540	65.9991	103.7257
Grayscale	1.1281	5.0914	6.5217	10.3246
Gaussian filters	14.2549	62.8575	80.9693	128.7111
Comparison	0.8251	4.1125	4.4386	6.9804
Total time	22.3716	123.2159	157.9298	249.7428

Analyzing the results on the different videos we can see that the most time is spent on loading the video and applying the Gaussian blur. While on the loading we cannot reduce the time due to the fact that is managed using opencv, the time used to perform the Gaussian filter can be reduced. The third longest time is the grayscaling which time can be reduced as well parallelizing the operation. The last thing worth noticing is that the comparison time even in case of a large number of frames is quite low.
Overall the focus to reduce the execution time will be on reducing grayscaling and Gaussian blur.

5 Proposed Parallel System

To find the ideal parallel solution we have to consider the sections of the code that we want to parallelize and how they are structured considering the kind of data that are used. As discussed in the problem analysis, and implemented in the sequential code, the frames are all loaded inside a vector of frames.
Mainly two different approaches to parallelize this problem could be used: the first one would be to divide a frame, which is a matrix of pixels, into n submatrices where n represents the number of processing units used; the second way

would be to divide the vector of frames in n chunks and compute the chunks in parallel. In both cases we have to consider that the overhead to divide and than recompose the input can be quite high and probably higher if we divide the single frames. On the other hand if we decide to divide the frames we could use some data parallel skeletons like the MAP.

6 Parallel code Analysis

The parallel implementation proposed in the previous paragraph has been implemented using simple threads and FastFlow; a third implementation in openMP is present as well.
The implementation in normal threads for both grayscaling and Gaussian filter has been done using a MAP skeleton and testing it with 4,8,16 and 32 threads. The results presented below will not reference all the times and statistic, for all the videos analyzed and the threads number tested, but will take in consideration the two extremes 175 frames and 3600 frames.

parallel units	Seq	4	8	16	32
Loading video	6.1631	6.0055	6.0156	6.0478	6.0165
Grayscale	1.1281	0.5140	0.2446	0.1652	0.1738
Gaussian filters	14.2549	4.0665	3.4335	2.0092	2.0152
Comparison	0.8251	0.8089	0.8412	0.8349	0.8528
Speedup	1	1.9632	2.1235	2.4699	2.46965
Scalability	1	1.9545	2.1141	2.4591	2.4587
Efficiency	1	0.4908	0.2654	0.1543	0.0771
Ideal Time	22.37166	5.5929	2.7964	1.3982	0.6991
Total time	22.37166	11.3953	10.5352	9.0574	9.0586

Table 1: results of threads for a video of 175 frames

parallel units	Seq	4	8	16	32
Loading video	103.7257	101.2322	99.9672	101.8461	99.8385
Grayscale	10.3246	4.6174	2.2776	1.9298	0.9508
Gaussian filters	128.7111	44.2457	25.0832	27.4385	9.5917
Comparison	6.9804	7.0102	7.0023	9.3527	9.3932
Speedup	1	1.8591	1.8591	1.6403	2.0850
Scalability	1	1.6419	1.9203	1.6943	2.1536
Efficiency	1	0.3974	0.2323	0.1025	0.0651
Ideal Time	249.7428	62.4357	31.2178	15.6089	7.8044
Total time	249.7428	157.1064	134.331	140.5671	119.7752

Table 2: results of threads for a video of 3600 frames

What we can see from the results, considering that most of the time left

when using multiple processing units is to load the video, we can achieve a good reduction of time. If for example we take the time to execute the gray-scaling and Gaussian blur, for the 3600 frames video, in sequential we have 139 seconds, while in parallel the same operations takes 11 seconds leading to a speedup of 12,5 and a efficiency of 0,4 .
A version where the comparison was paralleled has been implemented and tested but not included in the final project due to the timing being worst than the sequential version due to the overhead involved in decomposing and assembling the matrices in the right order.

The Fast Flow implementation follows the same idea used to implement the thread version, a map both for the grayscaling and for the smoothing, to implement it a parallel for was utilized.
The following tables report the results:

parallel units	Seq	4	8	16	32
Loading video	6.1631	5.9989	5.9979	6.0119	5.9951
Grayscale	1.1281	0.5100	0.2721	0.1458	0.0797
Gaussian filters	14.2549	7.2189	3.5998	1.8415	1.0292
Comparison	0.8251	0.8253	0.8241	0.8575	0.8080
Speedup	1	1.5369	2.0914	2.5250	2.8264
Scalability	1	1.5187	2.0666	2.4951	2.7930
Efficiency	1	0.3842	0.2614	0.1578	0.0883
Ideal Time	22.37166	5.5929	2.7964	1.3982	0.6991
Total time	22.37166	14.5559	10.6969	8.8597	7.9150

Table 3: results of FF for a video of 175 frames

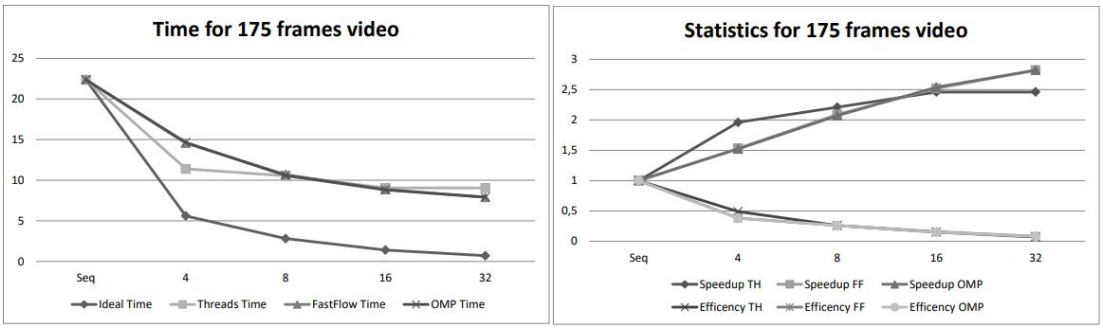
parallel units	Seq	4	8	16	32
Loading video	103.7257	99.5951	100.6004	101.7921	100.0782
Grayscale	10.3246	4.6727	4.1982	2.9045	0.7635
Gaussian filters	128.7111	88.4904	38.5417	29.5875	8.7905
Comparison	6.9804	6.9624	6.9101	11.3285	7.0209
Speedup	1	1.2504	1.6621	1.6047	2.1408
Scalability	1	1.6419	1.7154	1.6562	2.2095
Efficiency	1	0.3974	0.2077	0.1002	0.0669
Ideal Time	249.7428	62.4357	31.2178	15.6089	7.8044
Total time	249.7428	199.7241	150.2537	145.6241	116.6566

Table 4: results of FF for a video of 3600 frames

The results of the fast flow implementation show a even better time saving and just like before the most time is spent in the video loading phase. If we take the 3600 frame as example, we ca see that the grayscaling and smoothing in sequential takes around 139 seconds and the paralleled part is only 9 seconds

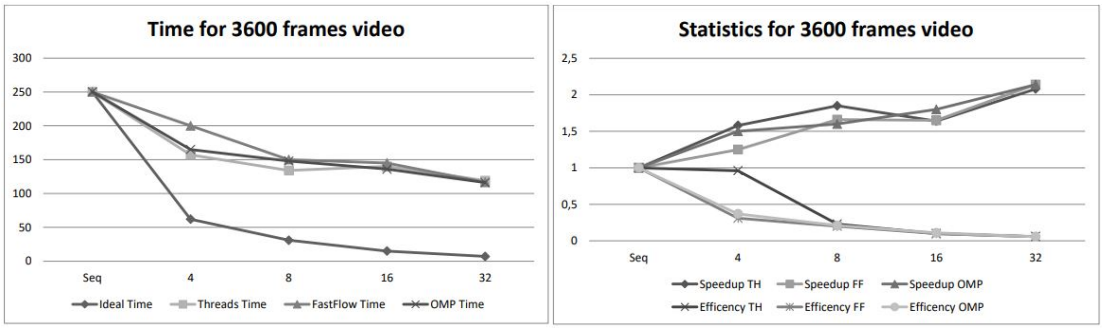
leading to a speedup of 15,5 and an efficiency of 0,48 .

The following graphs show the times and statistics to process the 175 and 3600 frames videos in Threads (th), FastFlow (FF) and OpenMp (OMP):



(a) (b)

Figure 2: Time and statistics for the 175 frames video



(a) (b)

Figure 3: Time and statistics for the 3600 frames video

The full data for all the video tested can be found on the Res.MD file either in the zip file or on github at: `https://github.com/PFasano99/SPR_Project_Unipi`

7 Conclusion

The project presented in this document shows that the use of multiple processing units can greatly improve the execution time of a solution.
Even tho the implementation choices regarding the paralleled code lead to a speedup and efficiency that are adequate to this problem, a way to reduce the

time even further would be to find an alternative to openCv that read/parallels the video division into singular frames.
In conclusion this project demonstrates how paralleling a problem using a parallel skeleton can vastly improve the solution quality and how we can use some metrics like speedup, efficiency, scalability to understand how good a solution actually is.

8 Deployment

To recompile and re-run the code there are two ways:
First method

- install FastFlow adding it to your c++ compiler include folder (following the instructions present on the FastFlow github: `https://github.com/fastflow/fastflow`);
- install opencv either using the guide given from opencv at `https://github.com/opencv/opencv` or using vcpkg (`https://github.com/microsoft/vcpkg`);
- run the CMakeLists inside the project folder, using cMake (`https://cmake.org/`), which automatically finds opencv and builds the project;
- after building the project an IDE, like Visual Studio or QtCreator, can be used to run it;

The second option

- download and build FastFlow and OpenCv;
- run the MotionDetection.cpp in the C++ compiler including the location of fast flow and configuring the opencv lib. The compiler line should be something like:
  ```
  g++-10 ./SPR_Project_Unipi/MotionDetection.cpp -o SPR_Project_Unipi/motionDetection
  ‘pkg-config --cflags opencv4‘ ‘pkg-config --libs opencv4‘ -pthread -std=c++17
  -O3 -I /usr/local/fastflow/ -fopenmp
  ```
- run the result of the compilation. The command should be:
  `./MotionSetection <Path to a video file>`

9 References

- Gaussian Filter/blur: https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm