Video Motion Detect

Marco Petix

July 12 2022

Abstract

The project subject of this report was centered on the implementation of a motion detection system for videos. The sequential version of said application has been used as baseline and comparison for the two other versions developed: the first one based on standard c++ threads and the second one based on common FastFlow's constructs.

1 Introduction

Frame-wise navigation of a video file, implemented thanks to the interaction with the OpenCV library, allowed us to simulate a stream of images. While each of these was involved in a series of manipulations described below, the first one was stored separately and defined as background. The remaining frames were then compared with said background to count the number of different pixels. The percentage of different pixels was finally compared to a predefined threshold to mark the frames they belonged to as containing motion. The output of each execution of the program in question was therefore the number of frames containing motion.

Excluding the **Frame extraction** phase, the sequence of steps involving each of these is made up by the following phases performed in an orderly manner:

Grayscale conversion It turns three-channel RGB images into single-channel ones. The average of the intensity levels for the three colors is used as the new value for the gray-channel.

Smoothing It tries to reduce the image noise by convolving the image with a 3×3[1] filter kernel that takes the average of all the pixels under its area and replaces the central element.

Background comparison Perform a pixel-wise comparison between the frame and the background in order to count the differing pixels.

Motion detection Increases the counter for frames containing motion if the percentage of pixels differing from the background ones is higher than the predefined threshold.

Figure 1 shows how the manipulations describe affect one of the frames of the video used in the remote machine experiments. Now, about the remaining sections of the report.
Section 2 contains a brief description of the files composing the project and the parameters needed to execute it.
Section 3 analyzes the three architectures proposed for the application described above: a first sequential version that will encourage a study of the computational bottlenecks and the more

[1] **Dimension of the kernel**: While a 3×3 kernel stands as the default kernel-dimension for the smoothing phase, larger kernels may be requested through the *kernelSize* parameter.

Figure 1: Example of manipulations performed on the frames.

parallelizable processes, a second parallel one implemented through the standard C ++ thread library and a third one, again parallel, implemented through the Farm class of the FastFlow library.
Section 4 analyzes the results of the experiments performed on the remote machine and discuss the impact of additional features like thread-pinning.
Section 5 is dedicated to a discussion of the project experience and overall results.

2 Implementation details

The main script of the project is *launcher.cpp*. It includes all the libraries and files necessary to run the program, this execution can be customized through the following parameters:

Parameter	Description	Valid inputs
filePath	Path to the video file	e.g. "demo_nature.mp4"
version	0 for Sequential, 1 for Thread-based, 2 for FastFlow	$\{0, 1, 2\}$
showTimers	0 shows the frames detected, 1 shows the computation time, 2 shows step-wise computation time (Sequential only)	$\{0, 1, 2\}$
motionThreshold	Percentage of differing pixels triggering the motion detection	Float $\in [0, 1]$
interFrameWorkers	Number of threads belonging to the thread pool	$\{1, 2, 3...\}$
intraFrameWorkers	Number of async intra-frame workers (See 3.2.2)	$\{1, 2, 3...\}$
threadPinning	0 avoids thread pinning, 1 performs thread pinning (See 3.2.1)	$\{0, 1\}$
kernelSize	Size of the convolution kernel filter	$\{3, 5, 7...\}$, 3 by default

The remaining files making up the overall project are:

- **FastFlow.cpp**. contains the farm of sequential nodes implementation of the program built with FastFlow.
- **Parallel.cpp**: contains the farm of sequential nodes implementation of the program built with a standard C++ thread pool.
- **Sequential.cpp**: contains the sequential version of the program.
- **utimer.cpp**: contains a class used to measure the computation time.

- **VideoUtils.cpp**: contains a class used to convert the frames to grayscale, smooth them and perform background comparison. It's used in all three versions of the program and the methods are implemented so to enable chunk-wise computation.

3 Architecture Design

3.1 Sequential Architecture

In the sequential version of the program, the frames are extracted and computed one after the other. The previously described manipulations: *Gray-scale conversion*, *Smoothing*, *Background comparison* and *Motion detection* are performed, by the same single core, in the order they have been introduced. To discourage unnecessary allocations of memory for the gray-scaled and smoothed versions of the frames, the variables containing the results of these operations, *cv::Mat* gray, smooth*, are reused across different frames. The same mechanism is applied to the *cv:: Mat* paddedGray* matrix, created in order to facilitate the convolution process.

3.1.1 Main computational bottlenecks

The methods of the *utimer* class, previously introduced during the lessons of the course, have been slightly modified and used in order to calculate the execution time for each of the frame manipulation phases. For the purposes of this analysis, the operations of *Background comparison* and *Motion Detection* were considered as a single step. **Figure 3.1.1** shows the results obtained by averaging multiple runs with increasing dimensions for the convolution kernel. This analysis confirms that, while the operations of *Gray-scale conversion* and *Motion Detection* maintain an almost fixed execution time, the *Smoothing* one tends to growth in workload together with the kernel.

(a) (b) (c)

Figure 2: Proportions between the execution time for the different phases of the sequential implementation: (a) kernelSize = 3, (b) kernelSize=9, (c) kernelSize=15.

Concerning the parallelization of the phases composing the program, it was soon decided to maintain a sequential version of the *Frame extraction* phase due to its dependence on the methods of the OpenCv library. The macro-phase formed by the *Background Comparison* and *Motion Detection* operations, on the other hand, was excluded from the scarce advantage and expected costly overhead that its parallelization would have entailed.

3.2 Thread-based Architecture

The entire motion detection process, from converting to grayscale to increasing the final counter, is characterized by transparency with respect to the order in which the frames, except the background,

are processed. This property, combined with the instead strongly constrained order of the individual manipulations that make up the frame-manipulation process, immediately favored the application of the stream parallel pattern of the Farm.

The idea of instantiating a new thread for each frame to be processed was quickly discarded due to the promise of costly overhead. This led to the consideration, and subsequent implementation, of a thread pool that could reuse instantiated threads by acquiring a new task from a shared queue.

The aforementioned queue was then implemented through the definition of packaged tasks enclosing the bind object made up by the worker function, sequentially carrying out the entire manipulation process over a single frame, and its target image. This frame-wise sequential approach was further encouraged by the saving in terms of inter-frame communication overhead that would have entailed a solution dividing the different stages of the process between different threads.

Figure 3 displays the architecture of the implemented program.

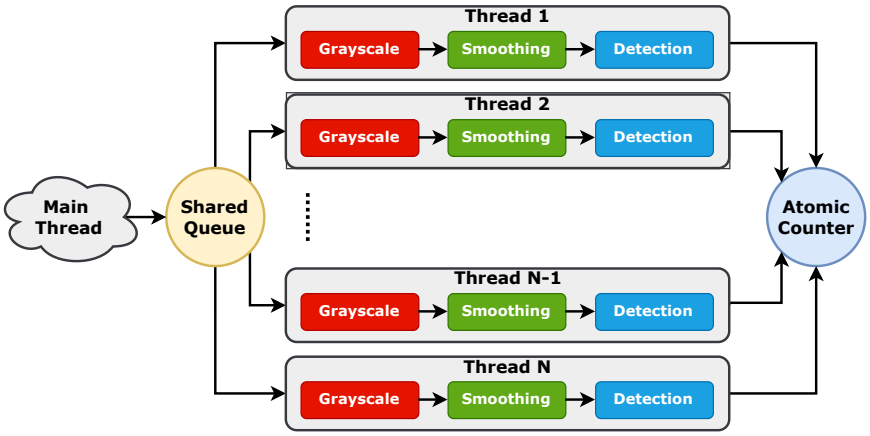


Figure 3: Standard thread-based farm architecture.

In order to avoid the potential inconsistency errors typical of the stencil / convolution operation, it was decided to avoid running this process "in-place". After adding a border on each side of the gray-scale image, every worker then used the latter in read-only and a new image, dedicated to the smoothed version of the frame, in write-only. Inconsistencies in the count of frames containing motion was avoided by defining the *motionCounter* atomic counter.

The preparation of the packaged tasks and their loading on the shared queue is performed by the main frame. The implementation of the queue in question, as well as of the main functions of the thread pool, follows what was seen during the lessons of the course.

3.2.1 Experiment A: Thread pinning

Migrating threads to a core other than the source one and the subsequent cache repopulation after a miss, especially if not sharing some of the higher cache levels, are known causes of potential overhead. The local machine memory topology, on which much of the project debugging took place, encouraged simple experiments with pinning the threads from the thread pool to single cores according to a cyclic scheme. While this feature does not fall within the default parallel implementation of the project, it was considered an interesting subject to investigate and, as such, the results of such endeavour are briefly discussed in 4.2.

3.2.2 Experiment B: Intra-frame parallelization by asyncs

Experiments have been carried out regarding the application of a data parallel pattern such as Map to the manipulation process of the individual frames. As analyzed in **Section 3.1.1**, operations such as converting to grayscale but, above all, the smoothing process require a great deal of computational effort from individual threads which could potentially be divided between additional computational entities. In order to pursue this hypothesis, and become more familiar with the C ++ constructs dedicated to threads, asyncs and all that is parallel, the parallel version of the application has been equipped with the possibility of requesting a parallelization of intra-frame processes through the *intraFrameWorkers* parameter required to run the program.

By assigning a value of one to this parameter, program execution will not change from what is described in the previous sections. A positive value greater than one will, however, encourage the definition, by the thread assigned to a particular frame, of intraFrameWorkers − 1 asyncs to which it will subsequently assign a chunk of rows of the frame matrix to work on.

Assigning chunk of rows is a theoretically sound decision when applied to these kind of operations due to how OpenCV tends to store the data from its Mat matrices in a row-by-row manner. Encouraging workers to work on neighboring rows is therefore a practice in line with the principle of locality. Once again additional matrices will be defined in order to avoid "in-place" computation and so to define separated memory areas to be considered read-only and write-only, respectively. Furthermore, the futures of each async instantiated by the frame owner thread will be recalled by the latter at the end of the grayscale conversion and smoothing phases so to guarantee the results of the work from said helpers to be computed before passing to the following steps of the frame manipulation process.

On a final note, while the idea of using threads, and not asyncs, for the role of intra-frame helpers was considered in development, this was discarded due to the frequent lower overhead entailed by the instantiation of asyncs with respect to threads, for a less complex integration with the thread pinning feature described in the previous subsection and in order to experiment with an additional construct of the C ++ libraries.

Due to unfortunate time issues, however, this component of the project has not reached the experimentation phase on the virtual machine and therefore will not be discussed in **Section 4** and is not to be considered as an official part of the parallel implementation that will be compared to the FastFlow-based in the later sections. The intra-frame parallellization remains anyway functioning, at least on the Docker environment provided as course material, and operable through the procedures described above.

3.3 FastFlow-based farm architecture

The parallel model of choice for the FastFlow-based implementation, for reasons similar to those described above, was again the Farm of sequential workers. After having become more familiar with the FastFlow nodes and its way to handle pointers, the implementation of the architecture in question continued mostly smoothly. The latter is based on the use of three nodes:

FarmSource a multi-output node, later promoted to emitter of the frame stream;

FarmWorker a standard FastFlow node and the frame-wise executor of the motion detection process;

FarmSink a multi-input node, then promoted to the results collector of the FarmWorker nodes.

Each Worker acts similarly to the thread-based implementation and sends an unsigned short to the collector with a value of 0 or 1 depending on the result of the motion detection process. These values are then added up by the collector in order to provide the final output of the program. **Figure 4** displays the architecture just described.

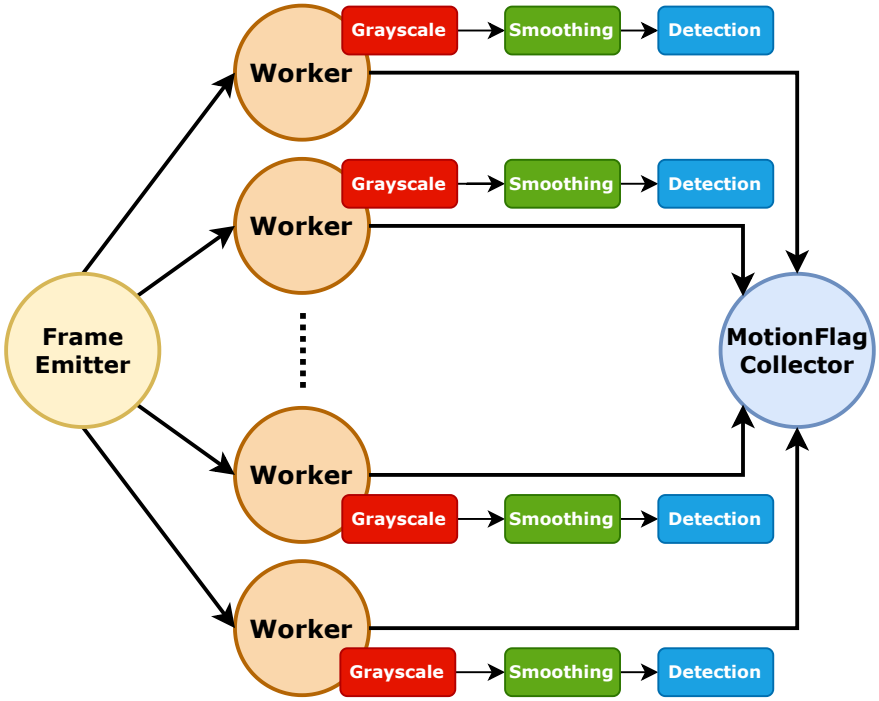


Figure 4: FastFlow Farm architecture.

4 Performance analysis

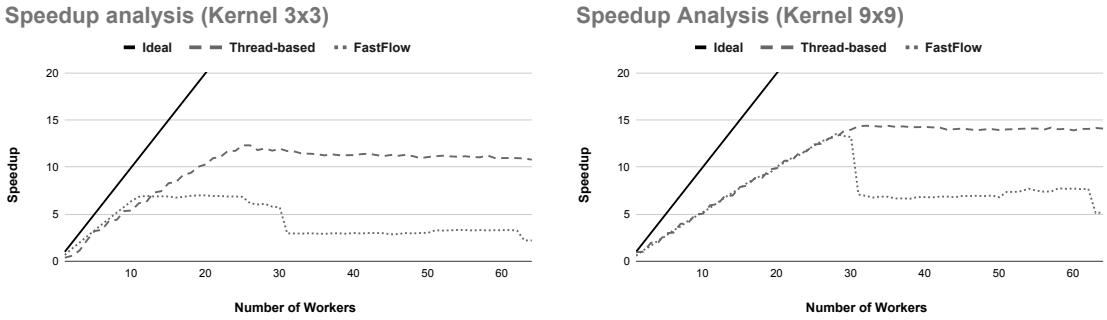
The analysis of the performances demonstrated by the two architectures was focused on the elaboration of a 1280x720 video of 544 frames. In addition to the use, as required, of a 3x3 size kernel for the implementation of the smoothing process, which has little influence on high resolution images such as the frames in question, the possibility of varying the size of the kernel has been implemented by adding the parameter *kernelSize*. Therefore, the tests carried out for the 3x3 kernel will be accompanied by the analysis carried out with a 9x9 kernel in order to analyze the performance of the program at a more onerous workload.

4.1 Thread-based and FastFlow implementations

The program demonstrated greater speedup potential when working with larger kernels. This is due to the poor performance of the sequential implementation when in contact with the large workload of smoothing processes with such large kernels. As we know, the workload required by the other operations, conversion to gray-scale and comparison with the background, are instead indifferent to the size of the kernel.

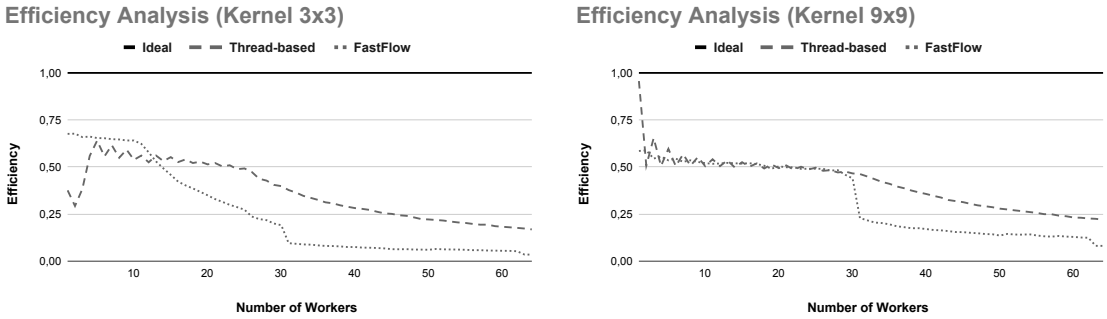
As shown by the **Figure 5**, both implementations present a stunting of the speedup around twenty to thirty workers. While in the case of the FastFlow implementation this is accompanied by a slight fall in performances, both architectures seems to maintain mostly stables speedups after

said threshold. This is attributable to the thirty-two cores present on the remote machine, the concurrency derived from a greater number of workers has certainly affected the computational resources of the workers themselves. The potential speedup benefit of an increased number of threads seems well balanced by the burden of managing them. Remarkably, in the case of the 9x9 kernel, the two architectures have virtually identical performance as long as both are below the processor core limit.



(a) (b)

Figure 5: Ideal speedup and comparison with Thread-based and FastFlow implementation: 3x3 kernel on the left and 9x9 kernel on the right.

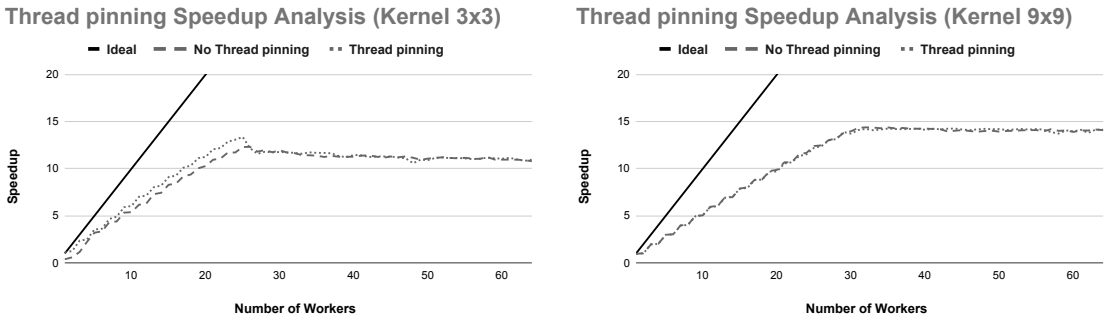


(a) (b)

Figure 6: Ideal efficiency and comparison with Thread-based and FastFlow implementation: 3x3 kernel on the left and 9x9 kernel on the right.

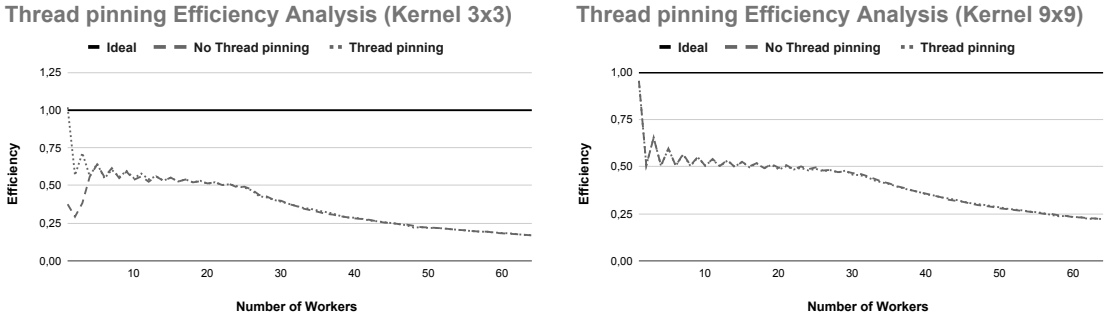
4.2 The impact of thread-pinning

The results of the brief experiment with the influence of thread pinning suggest little migration of threads from one core to another or, at the very least, a careful migration to threads sharing common cache levels. As shown in **Figure 7** the performances of the two versions of the parallel architecture show an almost identical trend if not for a subtle overtaking of the pro-pinning version in the less crowded regions of the computational experience using the 3x3 kernel.



(a) (b)

Figure 7: Ideal speedup and comparison among the thread-based implementations with and without thread pinning: 3x3 kernel on the left and 9x9 kernel on the right.



(a) (b)

Figure 8: Ideal efficiency and comparison among the thread-based implementations with and without thread pinning: 3x3 kernel on the left and 9x9 kernel on the right.

5 Discussion and Conclusions

The subject application of this project could have received a multitude of other architectures, each with their own advantages and disadvantages such as a higher memory burden or better completion times. Intra-frame parallelization remains an interesting initiative to consider and which encourages a study of the ideal trade-off regarding the relationship between intra- and inter-frame workers. I am grateful for the skills provided by the development of the project in question.