University of Pisa

Department of Computer Science

Parallel and Distributed Systems

Video Motion Detection

Luigi Quarantiello

July, 2022

Contents

1 Introduction

The project consists in the implementation of a simple motion detection algorithm. The process can be divided in four main operations:

- *Load*: Get a frame from the source video and store it in memory;
- *To Greyscale*: Transform the frame from a *RGB* image in a greyscale one, by substituting the value of each pixel with the average of the values from the three channels;
- *Smoothing*: Apply a filter to the image, with the effect that each pixel is averaged with the values of its neighbors;
- *Detection*: Compare the processed frame with the background image, *i.e.* the first frame of the video; the algorithm detects a motion if the number of different pixels between the current frame and the background is greater than a given threshold.

In the end, the algorithm returns the number of frames with motion.

2 Parallel Architecture Design

In this section, we will have an analysis of the problem, starting from the sequential implementation, from which we will make some considerations about how to parallelize the process, in order to achieve better performance.

2.1 Sequential implementation

The sequential code consists in a loop in which the four operations are performed one after the other. In this case, all the computation is done by a single thread at a time.

Before the loop, two `cv::Mat` objects are created, called `grey_frame` and `smoothed_frame`; these variables will hold the results produced by the `to_greyscale()` and the `smoothing()` functions respectively, and will be reused for each frame. In the `smoothing()` function, we can also choose the filter that we want to use, among the four matrices proposed in the project description.
Instead, a `cv::Mat frame` will be used to store each frame loaded from the source video.

In the end, the `motion_detection()` function, given the `processed_background`, the current `smoothed_frame` and a `threshold`, will increase the value of an integer variable `motion_frames`, if there are enough different pixels between the two frames.

The loop will break when the loaded frame is empty, meaning that the source video is finished.

In Figure 1 we have a representation of the workflow in the sequential implementation.

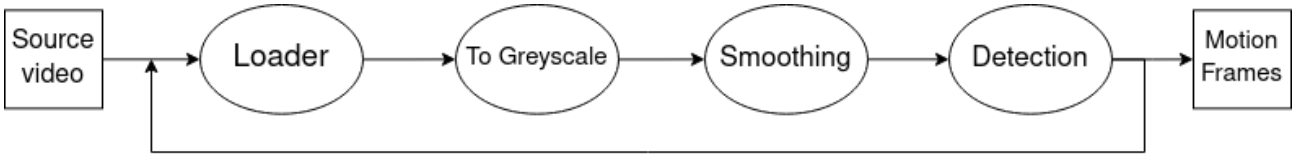


Figure 1: Scheme of the sequential implementation

2.2 Analysis of the performance

In order to study the problem from a theoretical point of view, we need to analyse the performance of the sequential code, taking into consideration the weight of each operation onto the overall computation time. To do so, a *verbose* version of the sequential code was implemented, in which a different timer is used for each operation.
Since the *Load* operation is provided by the ***OpenCV*** library, it cannot be parallelized; therefore, it was not included in this analysis, also because it turns out that it takes a negligible time with respect to the other operations.

In Figure 2 there is a representation of the time needed by the sequential implementation to process one frame, divided for each operation. The times are expressed in microseconds, and they are the result of an average of ten executions.
As we can see, the `smoothing()` operation is by far the most expensive one, since its execution time is one order of magnitude greater than the ones of the other two functions; in fact, it represents about the 85% of the entire execution time. That is because this operation involves several convolutions, which is a costly operation, especially if computed on large matrices. For this reason, the `smoothing()` operation is the bottleneck of the process, and it is the one that needs to be parallelized to get a speedup.

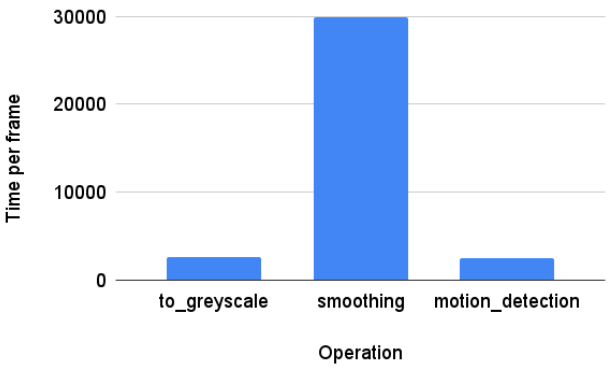


Figure 2: Execution time of the sequential code

On the other hand, the `to_greyscale()` and `motion_detection()` operations are much cheaper, since they basically require only a scan of the matrix plus simple operations on the pixels, which are, respectively, an addition and a division to compute the average and a subtraction to get the different pixels.

2.3 About the parallel implementation

The first idea about how to implement a parallel version of this application was to have all the *nw* workers processing a single frame per time step; that means to partition the frame into *nw* subframes, have each worker processing its own section, and in the end perform the reconstruction of the entire frame. This version was abandoned because it was found to be slower than expected. That may be caused by the fact that a single operation on a subframe takes an amount of time which is too small, and that makes the overhead produced by the coordination of the threads to generate a noticeable slowdown.

Instead, the final version of the parallel implementation consists in a *Farm* that process *nw* frames in parallel. In the following section, we will see more in details the architecture of this implementation.

3 Implementation details

The code is organized as follows:

- `main.cpp`: it handles the command line parameters, and then calls the requested implementation;
- `seq.cpp`: it contains the implementation of the three main operations, together with a method for the sequential execution;
- `par.cpp`: it handles the native C++ thread implementation, exploiting the methods defined in `seq.cpp`;
- `ff.cpp`: it handles the FastFlow implementation, exploiting the methods defined in `seq.cpp`;
- `SafeQueue.cpp`: it contains the implementation of a queue which includes a mechanism to handle the concurrency on read/write operations;
- `utimer.cpp`: it defines a timer used to measure the execution time.

To execute the `main` method, it needs some command line parameters:

```
./main video smoothing_matrix threshold verbose mode (nw)
```

`video`	The source video
`smoothing_matrix`	Select among the four matrices available. Integer between 1 and 4
`threshold`	The percentage threshold needed for the motion detection. Integer between 0 and 100
`verbose`	If 1, it measures the execution time divided for each operation. Only for sequential mode
`mode`	0 = Sequential mode, 1 = Standard C++ threads mode, 2 = FastFlow mode
`nw`	The number of threads to use. Not needed for sequential mode

Table 1: Command line parameters

3.1 Standard C++ Thread Implementation

In Figure 3, we can see a scheme of the workflow of the standard C++ thread implementation. All the computation is managed with `std::async`, using the `std::launch::async` policy to run the function on a new thread.

First of all, we have a *Loader* thread, which reads each frame from the source video and stores them in a queue. This thread, in addition to the reference to the source video, receives a reference to the shared queue, which is initialized with a proper size, in order to not saturate the memory and hence getting killed by the system. The queue size is computed taking into consideration the size in bytes of the backgroud image and the memory available on the machine. Further details about the SafeQueue implementation are in section 3.1.1.

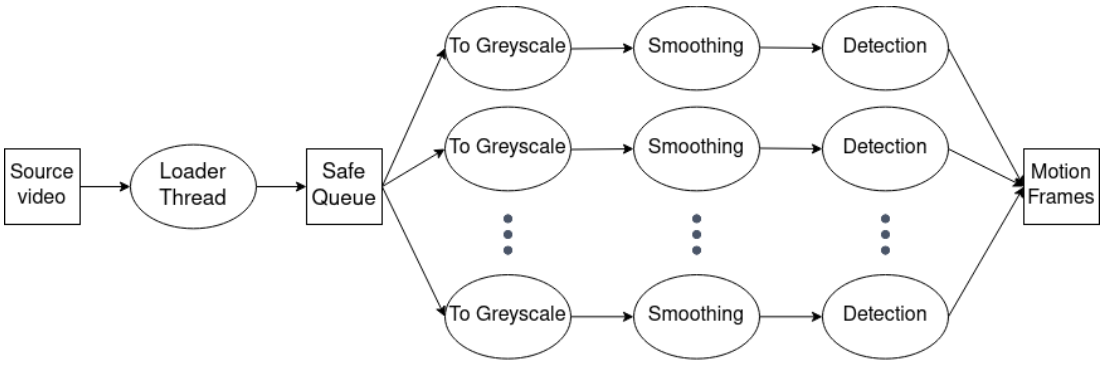


Figure 3: Scheme of the parallel implementation

Then, we have a Farm made by *nw* threads, in which each worker gets a frame from the shared queue and process it; each thread performs the work sequentially, using the methods defined in the sequential implementation.
Each thread creates two `cv::Mat` objects, a `grey_frame` and a `smoothed_frame`; both are defined having the same number of rows and columns as the original frame, but with a single channel. The `grey_frame` is needed to store the result of the `to_greyscale()` operation, which cannot be performed *in-place* because the original frame is defined to have three channels for the *RGB* values.
Instead, for the `smoothing()` operation, we use the additional `smoothed_frame` object to avoid the problems that arise from the *in-place* application of a stencil pattern. Plus, we pay particular attention to the computation at the borders of the matrix, which can be problematic and can lead to errors; for this reason, we always check that the position we want to access is between 0 and the number of rows/columns of the frame.

At the end, when a worker executes the `motion_detection()` method and if it returns `true`, it increases the value of a shared variable `motion_frames`, which is defined as an `atomic<int>` to avoid problems concerning the concurrency on the write operation.

Finally, the code returns the number of frames with motion, the completion time expressed in microseconds and an average of the latency of the process, expressed in microseconds and computed as the ratio between the completion time and the number of frames processed.

3.1.1 SafeQueue

SafeQueue is a class that defines a queue designed to be shared among threads, hiding the synchronization mechanism.
The constructor takes in input only the maximum size of the queue. Then, the two main methods are `push()` and `pop()`: the `push()` method takes in input an item, waits on a condition variable if the queue is full, inserts the item in the queue and notifies it to the other threads; the `pop()` method waits if the queue is empty, as long as the computation is not finished, then removes an item from the queue and returns it. These two operations are performed after acquiring a lock, to be sure to have only one thread modifying the queue at a time.

3.2 FastFlow Implementation

The FastFlow implementation is conceptually similar to the one provided with standard C++ threads. It exploits the `ff_Farm` high-level pattern offered by the library; a simple representation of the architecture can be seen in Figure 4.

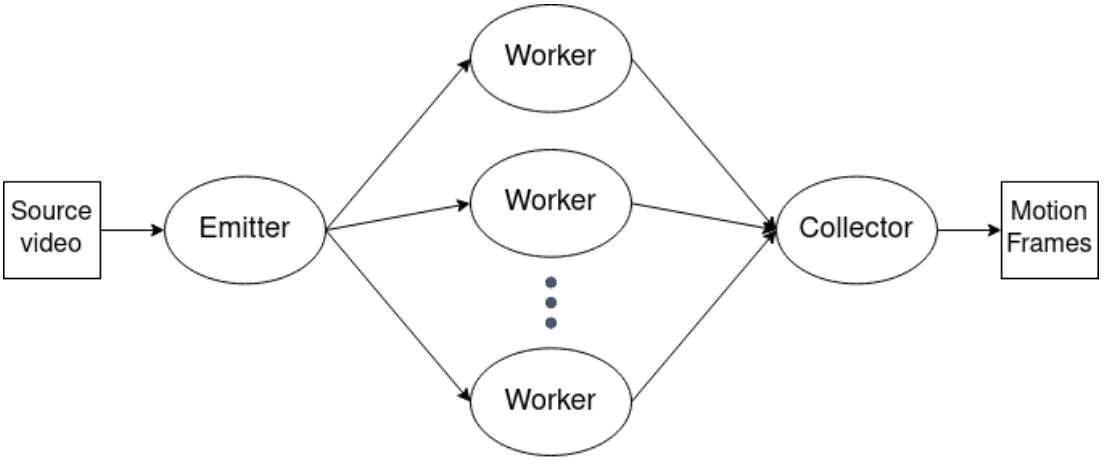


Figure 4: Scheme of the FastFlow implementation

Here we have an *Emitter* node that loads the frames from the source video, stores each of them in a `cv::Mat*` and sends them through the method `ff_send_out()`. Using this function, the library pushes the loaded frames

in the shared queues, so that the following workers can read them from there. When there are no more frames to read, the emitter returns the special value `EOS`, telling to the successive states that the computation has to be finished.

Then, there is a Farm of *nw* threads, each one performing the work sequentially, in the same way as the standard thread implementation. Also in this case, the threads use two auxiliary `cv::Mat` objects, and they exploits the function defined in the sequential implementation. Each thread return a `boolean` flag, indicating whether the current frame has motion or not.

This flag is passed to the *Collector* node, which increases the value of the integer variable `motion_frames`. In the end, the collector returns the special value `GO_ON`, meaning that the node is ready to receive the next input.

Also this implementation returns the number of frames with motion, together with the completion time and the latency, expressed in microseconds.

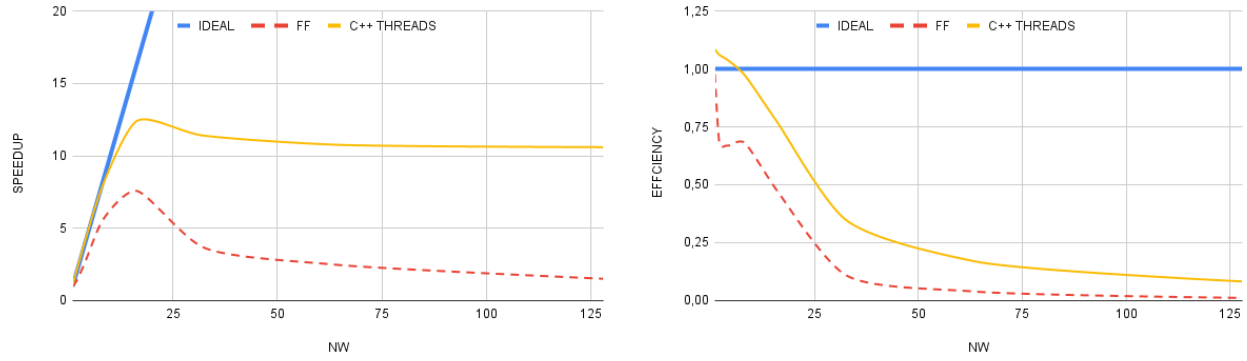
4 Results

The code was tested using a video with 601 frames, each made by 1280x720 pixels. The results are obtained through a bash script, that runs the two parallel implementations for several values of the *nw* parameter; for each configuration, the script performs ten runs, in order to have more reliable values.

In Figure 5a there are the curves for the speedup metrics. It is also included a curve for the ideal speedup, which is $f(n) = n$. As expected, we have an initial increase when the number of workers is low, and in particular smaller than 32, which is the number of cores of the machine; in this situation, the standard thread implementation quite follows the ideal curve, while the FastFlow implementation is a little bit lower. When the number of cores increases, becoming greater than 32, we have a drop in performances, and the speedup remains more or less stable.

In Figure 5b we have instead the curves for the efficiency metrics; also in this case, there is the curve for the ideal efficiency, which is $f(n) = 1$. These curves follows more or less the behavior of the speedup: at first, with small *nw*, the efficiency is close to 1; then it drops and it starts to assume very small values.

The drops in performance are expected because, when the number of threads approaches the number of real cores of the machine, the system spends more time to orchestrate the computation among the workers, therefore increasing the completion time. This situation gets even worse when *nw* is greater than the number of real cores, when we see a huge drop in efficiency, because of the overheads induced by threads management.



(a) Speedup

(b) Efficiency

Figure 5

5 Conclusion

We saw a comparison between the sequential code and two parallel implementations performing the video motion detection. The parallel implementations, especially with the right number of workers, offer a noticeable improvement with respect to the sequential implementation. I think that the speedup may be bigger if the computation was heavier, and in this context that may occur using a bigger matrix for the convolution in the smoothing process.
The development of this project gave me the opportunity to appreciate the difference between a standard C++ thread implementation and one using an high-level framework like FastFlow, which hides all the complexity and makes the implementation much easier and faster. The fact that the FastFlow implementation was a little bit less faster than the other one is maybe due to some error or missing optimization in the code.