Parallel and Distributed Systems: Paradigms and Models
Università di Pisa, Department of Computer Science

Project Report – Video Motion Detection

2021/2022

Andrea Laretto (619624)

1 Introduction

This report analyzes and presents some parallel implementations of the video motion detection problem, one implemented with the C++ threads of the Standard Template Library (STL) and one using the FastFlow parallel programming framework. A third implementation using the OpenMP framework is also presented for comparison.

2 Problem analysis

The full problem description is omitted here for brevity, and we just add some brief preliminary considerations. Although not present in the original description problem, an additional parameter T was assumed, representing the threshold for greyscale pixels to be considered as different. This was introduced to reduce the effect of camera noise in the video when comparing frames with the background. Furthermore, it is reasonable to assume that the videos must be read from a source using a streaming approach, for example by loading each frame from a file or by retrieving it from a live camera feed, and we consider the case where frames are not already loaded in memory and are not buffered at the beginning of the computation.
The objective is to compute and output the total amount of motion frames in the video. As a first consideration, we can notice how the output required by the problem immediately leaves room for optimization: no ordering on the frames is needed and the computations performed on each frame are independent of each other, except for the initial background bottleneck. Therefore, this case study is an example of an *embarassingly parallel problem.*
In our analysis, the processing of each frame can be subdivided into the composition of four different functions, and these conceptual stages will be referred to as follows:

- R: *read* a frame from the input stream;
- G: *greyscale* the frame by averaging of the 3 RGB channels of the frame, transforming it into a matrix of scalar values;
- B: *blur* the frame by applying a convolution with a blur kernel over the image;
- D: compute the pixel-wise *difference* between the processed frame and the first frame of the video (with difference threshold T), and check whether the total number of differing pixels over the total is greater than $P\%$.

Among the four functions considered, the work performed by R cannot be parallelized since it performs an I/O computation, and a detailed analysis will be provided in Subsection 3.2 to measure how this can effectively constitute the bottleneck of the entire computation.

2.1 Early termination assumption

We observe how the problem, as it is currently formulated, has the property that the time to process each frame is essentially uniform and does not dependent on the characteristics of the frame itself. Actually, one could take a different assumption for the problem setting, and consider the case where an *"early termination"* semantics is applied to frames: if the number of differing pixels already analyzed so far is above the required percentage, then the rest of the frame is not analyzed. Thus only a portion of the frame is processed and compared to the background, and this could be done in two ways: applying the blur B and greyscale G to the full frame and use early termination just in the difference stage D, or by applying the three steps progressively throughout the frame and terminate at any point.
For simplicity, we do not consider this early termination approach and instead focus on the case where the time to process each frame is uniform. It can be argued that, in real-world use cases, this assumption would not provide a significant imbalance between the computation of different frames: the idea is that motion frames are usually temporally close to each other, as detecting a movement usually means with a high probability that the next frames will also be motion frames.

2.2 Skeleton analysis

After analyzing the theoretical properties of the problem, several parallel pattern skeletons were considered and analyzed before actually tackling and coding the parallel implementation of the problem.

Proposal A

$$S_A = \mathsf{Farm}(\mathsf{Comp}(G, B, D)) \tag{1}$$

The first parallel skeleton and approach taken into account was the following: as different frames are loaded by the R function, they can be classified in parallel by different workers, which all process the frames independently and without any need for synchronization. The number of motion frames is then computed by a final collector stage that sums up the results. This effectively equates with the skeleton presented in Equation 1, which corresponds actually to the normal form for the composition of G, B, D.

Proposal B

$$S_B = \mathsf{Farm}(\mathsf{Pipe}(G, B, D)) \tag{2}$$

A second proposal is the skeleton presented in Equation 2, where a pipeline of the three stages required to process a frame is constructed instead of directly composing G, B, D. Supposing to take the same number of workers n, notice how the farm of S_B will have $1/3$ the branching factor of S_A.

Proposal C

$$S_D = \mathsf{Pipe}(\mathsf{Map}(\mathsf{Comp}(G, B)), \mathsf{Seq}(D)) \tag{3}$$

Finally, it can be noticed how, given a single frame, the G and B stages effectively perform independent computations on different parts of the frame, and the skeleton in Equation 3 could be considered to parallelize the work on a single frame.

2.3 Proposal discussion

It can be shown that, in absence of special situations, the normal form guarantees the best service time for a stream parallel skeleton composition. Indeed, directly composing the stages of the pipeline of S_B would reduce the overheads related to communications between different stages of the pipeline. Furthermore, S_A better exploits the principle of spatial locality since the data produced by each stage is ideally readily available in cache for the next stage to process. As we will see in Subsection 3.1, the three stages turn out to be very unbalanced in timing, thus limiting the service time actually provided by the pipeline.
The case of S_C could even possibly use a GPU to implement the Map pattern; in our setting, however, the Map pattern would incur in non-trivial overheads due to splitting and joining the frame results, and would not easily exploit locality of reference. This pattern could also be a suitable solution in the case where the problem necessarily enforced an ordering in the processing of frames. Since in this case the processing order of the frames is irrelevant, we can exploit the simpler and more direct farm pattern provided by the normal form which, in general, would not preserve ordering of outputs.
After considering some different approaches, it was concluded that the skeleton to most likely produce good results was the normal form S_A, due to its simplicity, good use of spatial locality, minimal presence of communication overheads, and best exploitation of the intrinsic properties of the problem. The FastFlow, Threads, and OMP implementations and their variants try to implement as closely as possible the parallel pattern described in S_A, and we will present our performance analysis for S_A.

2.4 Performance analysis

The performance model of the target skeleton S_A is provided in Equation 4,

$$T_s = \max\{t_e, \frac{t_G + t_B + t_D}{n}, t_c\}, \tag{4}$$

where T_e and T_c are, respectively, the emitter and collector times of the farm. The emitter effectively implements the R stage of the problem by reading the frames and supplying them to the farm workers, while the only use of the collector is to accumulate a binary information on whether the frame was a motion frame or not, and compute the total amount.

3 Methodology

In this section we briefly summarize the methodological choices and assumptions taken throughout the project. The OpenCV library is used to load the frames into memory from a `.mp4` file, albeit without employing any other functionality of the framework. The files considered in our analysis are described in Table 1.

Video name	**Resolution**	**Size**	**Length**
`test_big.mp4`	1920×1440	10.88MB	812 frames
`test_mid.mp4`	1280×720	3.62MB	431 frames
`test_small.mp4`	858×480	859kB	227 frames

Table 1: Test input videos considered and their characteristics.

Different kernels can be used to blur the images; since the kernel choice in general does not affect the computation time, in our analysis we consider for simplicity only the 3×3 average blur kernel $H1$, while still maintaining the code generic with respect to the kernel chosen

and its size, which can be selected at compile time. Similarly, the benchmarks performed in the project were taken with a standard set of blur and detection parameters fixed at the beginning of the analysis.
All experimental measurements contained in this report were repeated multiple times in order to smoothen out the statistical error and provide more accurate results, along with making sure that the experiments were run in optimal conditions with no external load whenever possible. The standard deviation for timings was also collected but is not reported here for ease of presentation; however, it turned out extremely valuable to discern experimental outliers and high variability results, e.g., the case of the machine becoming occupied by other colleagues in the middle of the experiments.

3.1 Measurements

In order to get a concrete perspective with the timings involved in the problem before starting the parallel version, the sequential version was implemented and some average times of the different stages R, G, B, D of the problem were obtained for the different video sizes. The average timings are reported in Table 2, averaged over 20 iterations. The timings t_R, t_G, t_B, t_D refer to the time spent in each stage for a single frame, while the last column T_{bg} reports the total time spent to open the file, read the background, and process it for the three kinds of video type. In other words, T_{bg} is the initialization time required to get the background ready and before we can start processing the actual frames of the videos. Since we are analyzing the sequential version and still abstracting from parallelism, this time does not include the initialization of threads and other parallel structures.

Video name	T_{seq}	t_R	t_G	t_B	t_D	T_{bg}
`test_big.mp4`	$64.95s$	$9326\mu s$	$16444\mu s$	$49770\mu s$	$2638\mu s$	$173849\mu s$
`test_mid.mp4`	$9.73s$	$2436\mu s$	$5816\mu s$	$18546\mu s$	$764\mu s$	$62852\mu s$
`test_small.mp4`	$2.34s$	$1104\mu s$	$2198\mu s$	$7109\mu s$	$218\mu s$	$35543\mu s$

Table 2: Measurements given by the sequential version, averaged over 20 runs.

From the results obtained, it can be noticed how the processing stages G and B are the points of the computation where most of the time is spent, with D being almost negligible. The background startup section T_{bg} turned out to be a considerable initial overhead, with opening the file only taking 20%/25% of the total startup time and the rest taken by the processing and the *first* read operation of the background frame.

3.2 Measuring read time

This read operation was analyzed more in detail, since it is the core bottleneck of the program and is also crucial in understanding the theoretical best speedup achievable in our setting. By analyzing the individual read operations, we noticed periodic spikes in latency, which suggested that frames are possibly buffered and read multiple at a time. Furthermore, it turns out that the read operation takes different times depending on the frequency with which it is performed, thus suggesting a form of prefetching and loading of frames ahead-of-time. This is crucial, since the reading time measured by the sequential time, which delays the read operations due to the grey and blur done immediately after, turned out to be *much greater* than the reading time measured by the parallel implementations, which issue read operations more frequently as the actual blur and greyscales is off-loaded to other workers by sending frames through a queue. This is probably done by the OpenCV framework in order to manage the frequent use case where the video is fully loaded in memory by applications and *then* processed, rather than processing frame-by-frame. We present the read times measured in the two scenarios in Figure 1 with a sample run, with their respective average shown as horizontal dashed lines. The performance improvement is shown in Table 3, and we notice how it is most noticeable in the `test_big.mp4`, which is the example that prompted us to do this analysis. The `test_small.mp4` case is not shown as the effect and timing difference is negligible, but is available in the accompanying Jupyter notebook of the project. This allowed to recompute our estimates of the time spent by the read operations as the frequency of the operations increases.

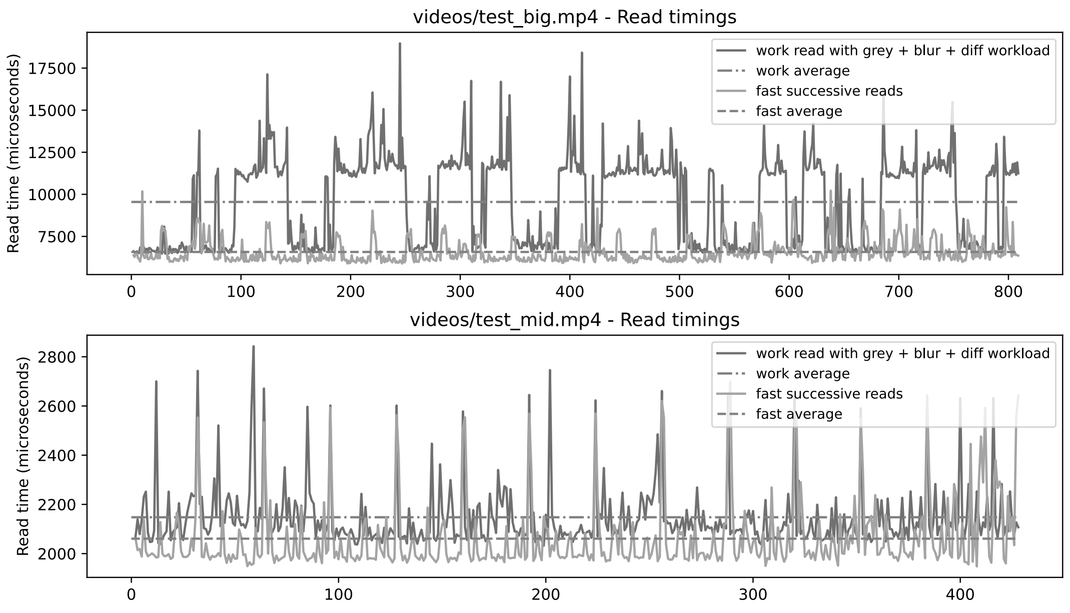

Figure 1: Read timings depending on the frequency of the reads: no wait, or time of $B + G + D$ workload.

Video name	Workload t_R	Successive t_R	Improvement%
`test_big.mp4`	$9326\mu s$	$6576\mu s$	29.48%
`test_mid.mp4`	$2436\mu s$	$2060\mu s$	15.43%
`test_small.mp4`	$1104\mu s$	$1040\mu s$	5.79%

Table 3: Read operations timings as read frequency increases.

3.3 Performance predictions

Considering a single frame execution, we can take again the performance normal form in Equation 4 and consider the case where, as the parallelism degree increases, the bottleneck of the computation shifts from the actual computation to the I/O bound read operation in Equation 5. Note how we equate t_e, the abstract time performed by the collector of the farm, with t_R. In this case, t_R corresponds with the *fast read*, since read operations are taken in quick succession since the frames are immediately provided to the workers. Since t_c only collects and accumulates the motion frames count, it can be assumed to be negligible in the following analysis. For simplicity, only the case of `test_big.mp4` was considered here, and a similar reasoning can be applied to the other video examples.

$$T_s(n) = \max\{t_R, \frac{t_G + t_B + t_D}{n}, t_c\} \approx \max\{6576, \frac{68852}{n}, t_c\} \tag{5}$$

A *theoretical best service time* $\widehat{T}_s \approx \max\{t_R, t_c\} = t_R \approx 6576\mu s$ is considered by taking the reading time as the only bottleneck of the computation: this can be achieved by increasing the parallelism degree to the point where the total processing time $t_P = t_G + t_B + t_D$ is comparable to t_R, which becomes the dominant time of the computation. By taking these two times as equal and rearranging Equation 5, we expect the *theoretical best speedup* $\widehat{s}$ to be reached around at least $n \geq 68852/6576 \approx 10.5$, with the other terms being negligible. Finally, a *theoretical best completion time* $\widehat{T}_{par}$ can be computed by considering the full computation in terms of $\widehat{T}_s$, as shown in Equation 6. This effectively coincides with the total serial time of the application, as per Amdahl's law.

$$\widehat{T}_{par} = T_{bg} + \widehat{T}_s \cdot (\text{number of frames}) \tag{6}$$

The best theoretical speedup is then computed to be $\widehat{s} = T_{seq}/\widehat{T}_{par}$ in the best case. The expected values are reported in Table 4, and we will refer again to these expected results and test them against the actual data obtained in the parallel version in Section 5. For completeness, we also show the predictions obtained by considering the (inaccurate) workload t_R provided by the sequential version.

	Successive t_R		**Workload t_R**	
Video name	$\widehat{s}$	$\widehat{T}_{par}$	$\widehat{s}$	$\widehat{T}_{par}$
`test_big.mp4`	12.0	$5.53s$	8.5	$7.78s$
`test_mid.mp4`	10.3	$0.95s$	8.1	$1.19s$
`test_small.mp4`	7.7	$271ms$	7.2	$286ms$

Table 4: Theoretical best predictions for the parallel implementation.

4 Implementation

This section briefly introduces and described the three implementations provided: one using C++ STL Threads, one with the FastFlow framework, and one using OpenMP.

Threads The workers are created according to a fork-join model, with a single emitter thread that provides the frames to be processed using a single thread-safe blocking unbounded queue. Each worker privately accumulates the count of motion frames for the ones that has received, with no external emission or synchronization. The queue can be signaled as *done*, in which case all n workers try to access a global atomic variable where each sums up their local motion frames count and the final result is computed. Both the standard version (`threads`) and a version experimenting with thread pinning (`threads_pinned`), using the identity mapping thread i to core i, are proposed and analyzed in the subsequent sections.

FastFlow A `ff_Farm` structure is used to coordinate emitter and workers: since the work performed by the collector is minimal, it would be wasteful to allocate an entire thread just to compute the results, which, similarly to the Threads version, only need to output their local counter once at the end of the entire computation. Therefore, we similarly decided to directly use an atomic counter that each worker writes their local counter to when they receive the `EOS` signal from the emitter, performing an atomic operation only $n-1$ times, with n being the parallelism degree. As anticipated with our problem analysis in Section 2, the computations for the frames are essentially balanced and therefore it was not necessary to use an `on_demand_scheduling` mechanism to improve the computations, and indeed it showed little to worse improvement. We provide the results for a standard version with no additional settings (`fastflow`), one using `blocking_mode` queues (`fastflow_blocking`), and one using both blocking queues and disabling the mapping on CPU cores (`fastflow_blocking_no_map`).

OpenMP This implementation was provided due to previous (positive) experience with the OpenMP framework, and its ease of implementation (`omp`). After setting up all the workers with the `parallel` pragma, a `single` block is used to read the tasks from input and takes a unique thread to act as emitter; the processing of each frame is done and distributed between the workers with a `task` block, and the global counter is updated atomically with an `atomic` block at the end of *each* frame processing.

4.1 Usage

We provide here a self-explanatory example to retrieve, compile, and execute the code of the project. It is required for the following dependencies to be installed: `FastFlow` (3.0.1), `OpenCV` (4.4.5), `OpenMP` (4.1.2). Upon successful compilation, the executables can be found in the folder `bin/`.

```
$ git clone https://github.com/iwilare/video-motion-detection
$ cd video-motion-detection/
$ cmake . && make
$ # Example with 0.02 greyscale threshold, >=30\% differing pixels,
$ # with default H1 average kernel, using 4 as total number of workers
$ bin/fastflow videos/test_mid.mp4 -d 0.02 -t 0.3 -n 4
> Motion frames over total video frames: (105 / 431)
> Total microseconds: 1695792
```

4.2 Project structure

A brief overview of the main files contained in the project is presented in the following tree.

- `videos/`: contains the three example videos.
- `plot/`: jupyter notebook for data visualization.
- `src/`: source files.
  - `lib/`: common files shared among all versions.
    - `args.cpp` — command line arguments management.
    - `cpu_affinity.cpp` — cpu affinity procedures.
    - `kernels.cpp` — example kernel definitions.
    - `shared_queue.cpp` — implementation of a thread-safe blocking queue.
    - `utimer.cpp` — helper structure for time measurements.
    - `video_detection.cpp` — main video detection logic and processing.
    - `main.cpp` — common main structure for all implementations.
  - `measurements.cpp` — standalone file to measure the individual stages timings.
  - `sequential.cpp` — sequential implementation.
  - `threads.cpp` — main implementation with `C++` STL threads.
  - `fastflow.cpp` — main implementation with `FastFlow`.
  - `omp.cpp` — main implementation with `OpenMP`.
  - `...` — other versions.
- `benchmark.sh` — main file to run every benchmark, writes .csv to `data/`.
- `measurements.sh` — script to measure sequential and read time on all videos.

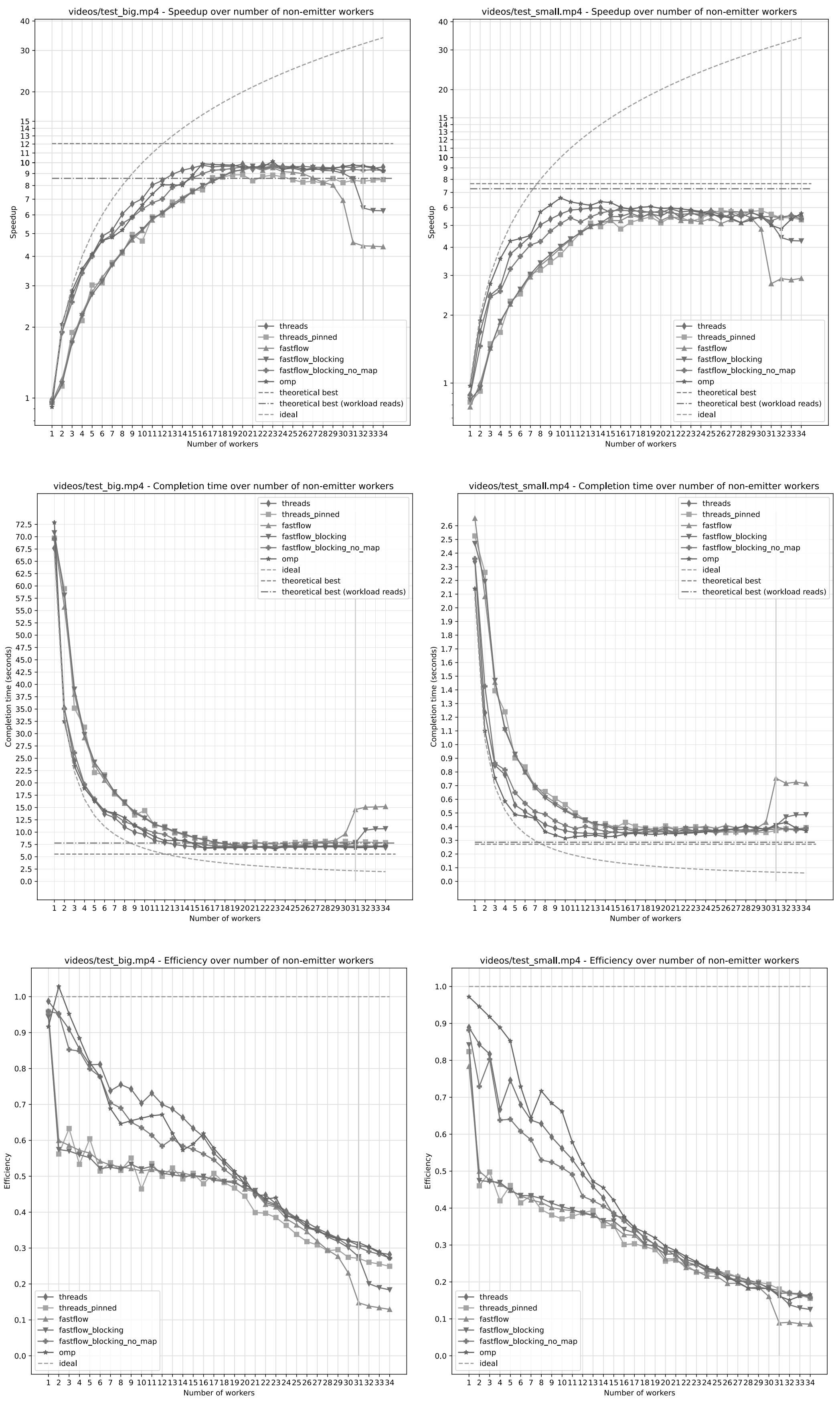

Figure 2: Speedup, completion times, and efficiency for the `test_big` and `test_small` cases, averaged over 5 runs.

5 Results

In this section we report the measurements obtained with the parallel versions and discuss them. These results were obtained by running them on the Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz machine with 32 cores, averaged over 5 runs, and are presented in Figure 2. For simplicity we present speedup, completion time and efficiency for the `test_big` and `test`, since he `test_mid` case shows similar behaviours to the other versions. The complete plots can be viewed in the Jupyter notebook. In order to provide a better interpretation of the results, graphs actually start with 2 as the number of *total* workers, since at least one emitter and one worker are required for all 6 versions to work. Thus, we refer to the number of *farm* workers, and, for the same reason, the ideal performance $T_{ideal} = T_{seq}/n$ reported in the graph takes n as the number of farm workers, even though strictly *total* workers would be more correct (e.g., an hypothetical parallel implementation that does not require an emitter should be favoured). The figures also indicate the theoretical best completion time and speedups previously computed in Table 4, using both the fast read time t_R with successive reads and, for completeness, the case of reads t_R separated in time with a workload. As can be seen in the case of `test_big`, the predictions given by the theoretical analysis would greatly overlap with the results if we had used the original time directly given by the sequential version, and this theory-driven observation was what prompted the more detailed analysis in Subsection 3.2.

General considerations. We notice how the best results were provided by the `threads`, `fastflow_blocking_no_map`, and `omp` versions. The `threads` version constitutes the most complex implementation, since it requires a custom-built implementation to define a shared queue mechanism. This notwithstanding, both the `fastflow` and `omp` versions provide sufficiently abstract parallel mechanisms that allowed us to focus on the main program logic. At the same time, both still provided quite satisfactory and comparable performance results to the `threads` versions, with the FastFlow framework being vastly more flexible and customizable in its behaviour and performance settings, as we have seen with the different versions tested.

Thread pinning. Among the 6 versions tested in our measurements, the behaviour of the versions can indeed be easily distinguished in two groups, namely, those that employ a form of thread pinning to CPU cores and those who do not. Thread pinning was done to minimize the effects of context switching between cores and maximize the locality of the caches: the CPU topology of the machine was viewed with `lstopo`, which presents no evident signs of cache sharing or hierarchical subdivision of caches. Experimentally, on this architecture the effect of thread pinning is quite noticeable and considerably degrades performances, especially with a low number of cores, while its effects seem to smoothen out as the number of threads increases. This could be due to the fact that setting affinity with a low parallelism degree here might lead to worse choices than those that can be performed by the kernel. However, thread pinning also seems to provide more stable and predictable performance measurements in all three video versions, probably because we remove the unpredictability factor of threads being moved from and to different CPU cores. Among the thread pinning versions, both the `threads_pinned`, `fastflow`, and `fastflow_blocking` show similar overall performance; approaching and going above the limit of 32 cores, the two FastFlow versions considerably drop in performance, with the no-mapping `fastflow_blocking_no_map` version not being affected, and this could be because of the overhead given by the CPU trying to pin more threads than the number of cores available. Similarly, all thread pinning versions have a sharp drop in efficiency after a second worker for the farm is introduced.

Blocking vs non-blocking. The version with blocking queues seems to have less of a performance drop when reaching the 32 cores limit, possibly because, despite the unsuccessful mapping, the unused cores do not perform active waits on the queues and degrade performance. Overall, the blocking version and the standard one seem to have comparable timings when a sufficient parallelism degree is reached.

Video size. The rationale for testing different video sizes is to experimentally verify how, on one side, small computations such as the `test_small.mp4` case should be harder to parallelize and achieve good speedups without incurring in the overheads of setting up threads and the parallel computation. Indeed, the small test case achieved overall good results, but is the only case where a slight progressive increase in completion time can be noticed as the number of threads to setup increases. On the other hand, computations with greater completion times allow us to amortize the setup and management of concurrent activities, and our expectation to see better speedups with tasks that workers take more time to complete was confirmed. The speedups and total times achieved in the case of `test_big` are quite satisfactory, and are even closer to the theoretical best than the case of smaller datasets.

Overheads. As argued in Subsection 3.1, the main bottleneck of the computation is constituted by the reading time t_R, which limits the maximum speedups achievable and constitutes the main contribution to the serial fraction of the program. However, our analysis does not take into account communication overheads, both in terms of concurrent access to the queues used in both `fastflow` and `threads` analysis, the overhead of writing the final results to the shared atomic counter, and, most importantly, the overhead related to moving and accessing frames from the different cores. Since no modification is made to the input frames, the effects of false sharing should be limited to none, but there is overhead in distributing the frames to the workers and accessing the common background frame. In order to slightly mitigate these overheads, each worker in our implementation keeps a local copy of the background frame copied at the beginning of the computation, and is thus not shared. Partially because of this, notice that the actual reading time t_R for the computations is slightly higher than the ideal one computed, since the read operations are delayed with the overhead of distributing frames: for example the `threads` version directly measures a $t_R \approx 6898\mu s$ instead of the minimum $6576\mu s$. Since we are doing predictions and considering measurements for the *ideal* case for the sequential version, we take $6576\mu s$ as the nominal (successive) read time. Furthermore, the time to initially setup concurrent resources was not taken into account, but this should constitute a relatively minor overhead, especially when comparing this one-off time with respect to the total computation time, e.g. in the case of `test_big.mp4`.

Finally, we report in Table 5 the minimum service times reached by the parallel versions along with the corresponding parallelism degree, which were computed as $T_s = (T_{par} - T_{bg})/(\text{number of frames})$.

	`test_big.mp4`		`test_mid.mp4`		`test_small.mp4`	
Timing	$\min\{T_s(n)\}$	n	$\min\{T_s(n)\}$	n	$\min\{T_s(n)\}$	n
`threads`	$8081\mu s$	19	$2723\mu s$	13	$1374\mu s$	13
`threads_pinned`	$8993\mu s$	19	$2800\mu s$	29	$1414\mu s$	25
`fastflow`	$8341\mu s$	20	$2823\mu s$	20	$1454\mu s$	18
`fastflow_blocking`	$8253\mu s$	20	$2780\mu s$	20	$1422\mu s$	20
`fastflow_blocking_no_map`	$8238\mu s$	22	$2771\mu s$	17	$1411\mu s$	15
`omp`	$8164\mu s$	22	$2731\mu s$	12	$1307\mu s$	9
Ideal service time $\widehat{T_s} = t_R$	$6576\mu s$		$2060\mu s$		$1040\mu s$	

Table 5: Minimum service time computed and the parallelism degree at which it is reached.