SPM - Project report v2

Leonardo Vona
545042

June 27, 2022

1 Reference sequential solution

The motion detection over a single frame can be decomposed into four steps:

1. Pick the frame from the input video.
2. Turn the frame to greyscale.
3. Smooth the frame.
4. Detect motion computing the percentage of different pixels between the background picture and the frame.

The sequential solution performs an infinite loop which retrieves the frame from the input video, checks if it has motion and stops when the frame retrieved from the video is empty.

1.1 MotionDetector

The utilities to handle frame capturing, elaboration and motion detection have been implemented into a dedicated object `MotionDetector`. The object is initialized with the filename of the input video and the parameter k, which is the threshold percentage to declare motion into a frame.

When the object is instantiated, it initializes the background picture. This is performed retrieving the first frame from the input video and applying to it a greyscale and a smooth filter in sequence.

1.1.1 Greyscaling

The greyscaling of a frame is implemented by three nested loops. The three indexes loop respectively through the rows, the columns and the channels of the frame.

Before the loops, a `Mat` object which will represent the greyscaled frame is filled by zeroes. The `Mat` is composed by `CV_8U` elements (`unsigned int` of 8 *bits*).

For each pixel of the frame, the average of the three channels (Red, Green, and Blue) is computed and assigned to the frame representing the greyscaled version.

1.1.2 Smoothing

The smoothing is performed by looping through each pixel of the frame, which must be composed by `CV_8U` elements.

For each pixels, the average of the neighbour pixels (whose distance in horizontal / vertical / oblique is 1) and the pixel itself is computed, and the result is assigned to a new frame representing the smoothed version.

The border pixels are managed by checking if the target neighbour pixels are out of range (the indexes are smaller than 0 or larger than the number of columns / rows). This solution requires additional time to perform each iteration of the loop, but it exploits the spatial and temporal locality of the data.

1.1.3 Motion Detection

The actual motion detection is implemented by comparing pixel by pixel the frame and the reference background picture. A double loop is used and a counter is incremented each time the pixel of the frame and the relative one of the background picture differ. At the end of the loops the number of different pixels is scaled to percentage and, if the result is larger or equal than the parameter k, the frame is declared to have motion.

1.2 Single step time

Some experiments have been performed on the remote machine (detailed into Table 1), in order to evaluate the time needed by each step to process a single frame. The results show that the most computational-intensive step is *smoothing* and there is quite divergence between the time required by each kind of step.

Resolution	Capture	Greyscale	Smooth	Detect motion
480x270	397 μs	524 μs	2534 μs	85 μs
640x360	682 μs	860 μs	4435 μs	126 μs
1280x720	2489 μs	3500 μs	18044 μs	494 μs
1920x1080	5539 μs	7664 μs	40767 μs	1180μs

Table 1: Each timing refers to the processing of a single frame. For the tests, the same video with different resolutions has been used. The capture time is the time needed to retrieve a frame from a video; it is computed as the time to retrieve all the frames from the video divided by the number of frames. Each timing is the result of the average of 10 tests performed on the remote machine.

The inner loops that compute the smoothing, greyscale and difference have been annotated with the directive to the compiler `pragma GCC unroll 8` in order to speedup the computation. The unrolling factor 8 has been chosen empirically, after performing measurements with different unrolling factors.

1.3 Balancing

Other experiments have been performed to establish if the time to perform greyscale, smooth and detect motion steps varies and depends on the frame (not the resolution, but the image itself) or it is stable. The results show that the computation is pretty similar for all the frames. Four different videos have been used for the tests.

2 Analysis

The computation of the number of frames with motion detected can be seen as a *stream parallel computation*.

Pipeline The motion detector can be implemented by means of a pipeline, composed of the following stages:

1. Pick frame from the input video.
2. Turn the frame to greyscale.
3. Smooth the frame.
4. Detect motion computing the percentage of different pixels between the background picture and the frame.

Using a *rplsh*-like notation, the structure of the application is (the service times of the stages are obtained averaging and normalizing the experimental results reported in Table 1):

```
pipe(seq(50), seq(65), seq(340), seq(10))
```

2.1 Refactoring

Farm out Stages 2, 3 and 4 of the pipeline are suitable for being farmed out, in order to reduce the overall service time. On the contrary, the first stage can't be farmed out because the access to the input video has to happen in mutual exclusion, and then is possible to retrieve only one frame at a time. As a matter of fact, the parallelization of the first stage can only introduce a slowdown with respect to the sequential execution, because no gain is achieved and overheads are introduced due to, as an example, synchronization between workers for the access in mutual exclusion.

The last stage, if farmed out, has to access in mutual exclusion a global variable which counts the number of frames where motion is detected, and then the parallel computation of the last stage is stateful (*accumulator pattern*). In order to reduce the number of accesses in mutual exclusion, it is possible to use a local variable which keeps the local count of frames with motion detected and then, when the worker has finished its computation (when it receives an EOS, in this case), it updates the shared state in mutual exclusion. This approach reduces the number of accesses in mutual exclusion from n (the number of frames) to nw_4 (the number of workers assigned to the farmed out stage 4). The reduction of accesses in mutual exclusion reduces the overhead due to synchronization between workers, which would not have been necessary if the computation was sequential.

In addition, each farmed out stage requires two extra workers for the emitter and the gatherer (which could be reduced from 6 to 4 if the emitter and gatherer of stages 2 - 3 and 3 - 4 are merged).

Normal form A theoretically better solution that uses a minimum number of workers, can be obtained applying the *normal form* and then refactor the pipeline of farm stages into a farm of composition of sequential steps.

```
farm(comp(seq(65), seq(340), seq(10))) with emitter_time = 50
```

The emitter of the farm picks a frame from the video and assigns it to a worker, whereas each worker takes a frame and computes stages 2, 3 and 4 as a single sequential computation.

The farm does not need a gatherer because when each worker receives the EOS they update the shared state represented by the overall number of frames with motion detected, which is the output of the program, and then does not need further elaboration (see Figure 1).

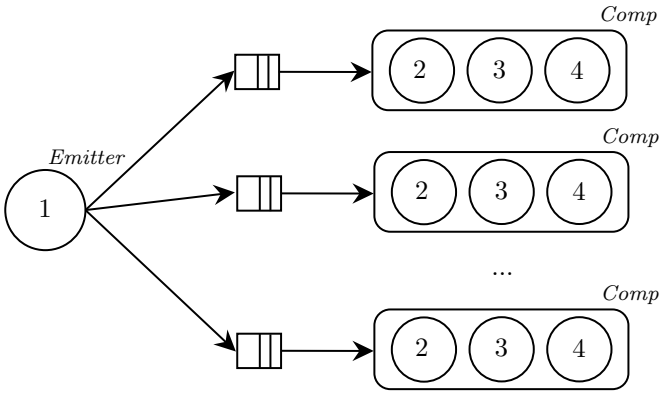


Figure 1: Structure of the parallel solution.

2.2 Structure of emitter and workers

The life-cycle of the worker is:

```
loop
    pop frame from queue
    if(empty frame)
        break
    else
        elaborate frame
update global state
```

The emitter assigns frames to workers in a round robin fashion. No dynamic scheduling is chosen because the amount of work needed for each frame is comparable, so the computation is quite balanced, and the risk that a worker takes much more time than the others (increasing the overall completion time) is limited.

2.3 Expected optimal number of workers

It is expected that the optimal speedup is achieved with 8 resources. This is because we reach the service time of the emitter and then we can't further improve.

```
[ ts: 50 ]:
farm(
    comp(seq() with [ ts: 65],
         seq() with [ ts: 340],
         seq() with [ ts: 10]))
with [ nw: 9 (8 without gatherer), emitter_time: 50 ]
```

The expected completion time is:

$$T_C = \max\left\{T_e,\ n \cdot \frac{T_2 + T_3 + T_4}{nw}\right\} = n \cdot \frac{T_2 + T_3 + T_4}{nw}$$

Where n is the total number of frames, nw is the number of workers, and the last equality holds true because $T_e < T_2 + T_3 + T_4$ and $n \gg nw$ in all significant cases.

3 Pthread solution

The chosen parallel solution to be implemented is the normal form one (farm of comp, see Figure 1).

The program takes as input the video file, the threshold used to declare if a frame contains motion and the parallel degree, which must be between two and the hardware concurrency of the machine where it is run. There must be at least two concurrent activities because there has to be at least the emitter and one worker.

Emitter The emitter is implemented as a thread running a function containing a loop which reads the frames from the video, assign them to workers in a round robin fashion and stops when it reads an empty frame from the input video. Before stopping, it pushes to each queue assigned to workers an EOS, which is represented by the empty frame.

Workers Each worker executes a loop where it retrieves a frame from its queue, checks motion into the frame and, if there is motion, increases a local counter. The loop stops when it retrieves an EOS from the queue. At the end of the computation sums its local counter to a global atomic variable, which stores the total number of frames with motion detected.

Communication channel The communication between the emitter and each worker is implemented by a dedicated Single Producer Single Consumer queue. No synchronization mechanisms are implemented in order to reduce overhead. The emitter simply pushes the frames into the queue, whereas the worker does a spinlock on the condition of empty queue. The spinlock would normally be implemented by an empty loop, which stops when there is something to pop from the queue. Anyway, the `-O3` compiler flag removes the loop during the optimization. Then, to overcome the issue, the assembly `NOP` instruction has been added into the branch of code executed when the queue is empty.

Pin threads To reduce the overhead of moving the working set from one core cache to another, each concurrent activity is pinned to a specific core for the whole computation.

Unbalancement During some tests of the pthread solution, it emerged that the time required by the workers is not uniform, even if they are all assigned the same number of frame to elaborate and the time to compute each frame is similar. In particular, only one thread, which always results to be the one pinned to the core which shares caches with the emitter, is faster up to a 25% factor with respect to the other workers. This happens because the faster thread shares the quickest levels of memory with the emitter, and then does not always need to access the slower main memory to retrieve the frames, since they are already present in cache. To balance the computation, the scheduling policy has been changed to a dynamic on-demand one. The policy has been implemented by modifying the emitter so that it pushes a new frame into the queue of a worker only when it is empty, and then a sort of a single position buffer is used.

It is necessary to point out that the unbalancement is less evident when the number of workers increases and the amount of work per worker decreases, due to overheads induced by the higher communication between threads.

4 FastFlow solution

For the FastFlow version it has been chosen to use the `ff_Farm` Core pattern. The emitter of the farm is a `ff_node_t<Mat>` object that simply retrieves the frames from the video and send them to the workers. Each worker is also a `ff_node_t<Mat>` which receives frames and checks if there is motion. The worker does not send out anything because the collector of the farm is not present. The `svc_end()` method of the worker updates the global state summing it up the local counter of frames with motion.

As reported in the previous section, some unbalancement is present between workloads of different workers, then the scheduling of the farm emitter has been set to on demand.

5 Performance evaluation

5.1 Memory overhead

The main source of overhead in the parallel version (pthread or FastFlow) with respect to the sequential solution is the memory overhead. In fact, the parallel version has to perform much more accesses to the slow main memory, because the sequential version (partially) has frames already in cache, whereas parallel needs to retrieve them from the main memory. This causes a 2x slowdown in the processing of a frame.

5.2 Communication overhead

Emitter The time lost by the emitter to send a frame is a source of overhead. The exact time spent to schedule frames to the workers is not predictable because depends on the dynamic execution and is larger as the number of workers increases, due to the higher number of checks involved.

Worker For the pthread version, experimental results show that the time lost in retrieving frames from the queue is the 0.5% of the total time required by a worker to process all the frames assigned to it, which is a limited overhead.

For the FastFlow version, the percentage of lost ticks to pop is comparable to the time lost in the pthread version.

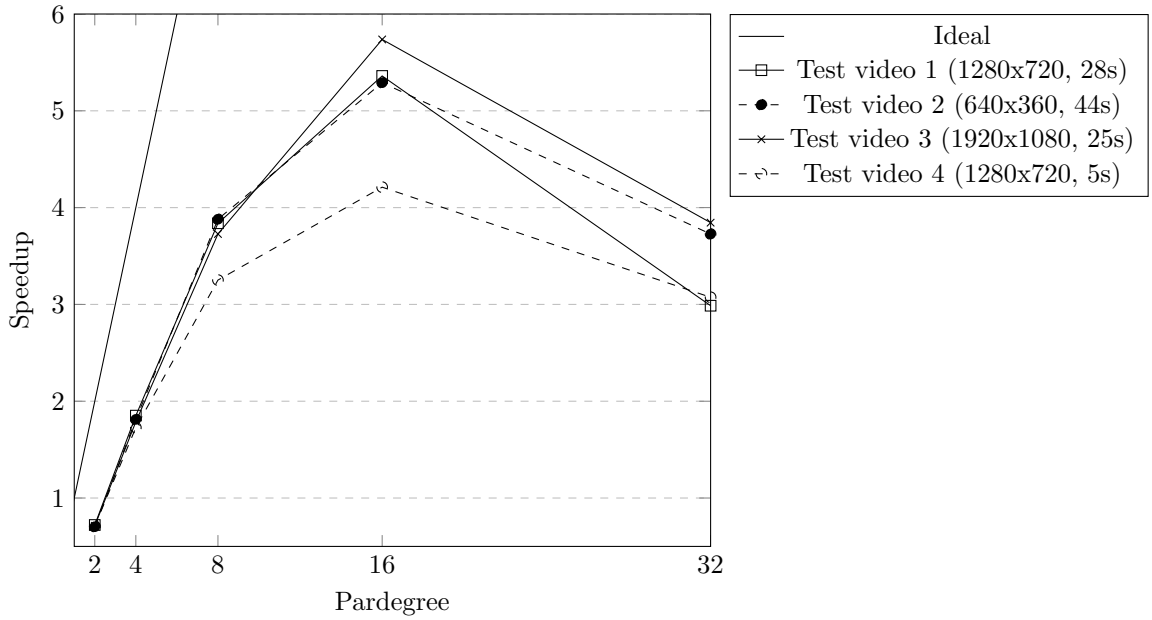
5.3 Optimal number of concurrent activities

The expected optimal number of concurrent activities was 8, but taking into consideration the overheads (memory, communication and others, such as the time to set up the threads), the time to process a frame is doubled, and then the number of concurrent activities that gives the best performances is 16. Above that value, performance degrades because the limit of the interarrival time is reached, and the overhead is larger because there are more concurrent activities.

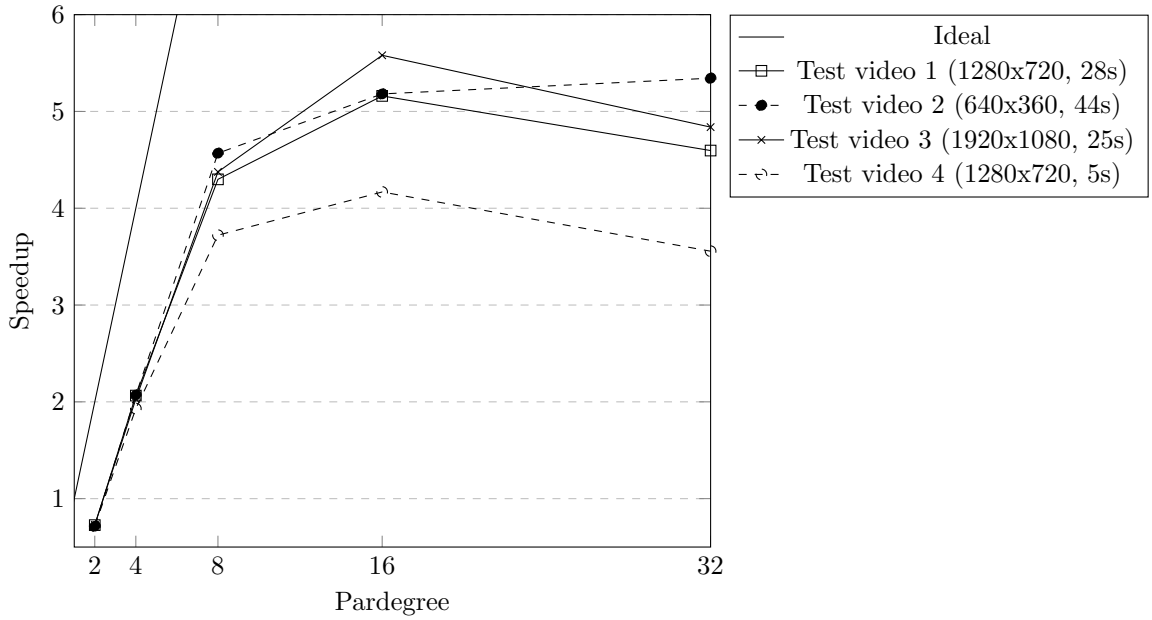
These considerations are supported by the results of the tests performed to measure the speedup of both the pthread and FastFlow version. The results are reported into Plots 1, 2, 3a, 3b, 3c, 3d, 4, 5.

6 Conclusions

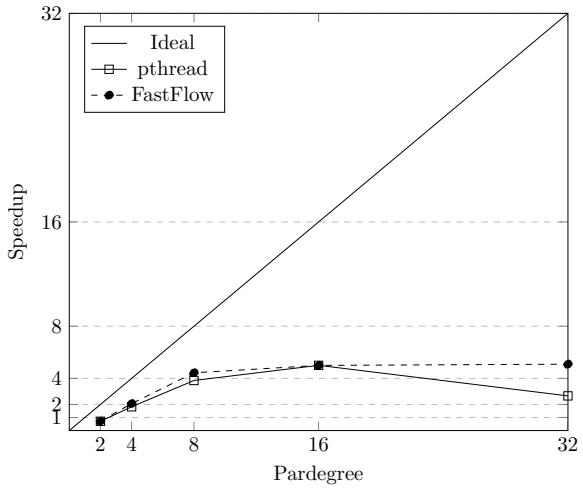
From the results it is possible to see that both parallel versions of the motion detector have similar performances and the nature of the problem induces very significant reduction with respect to the ideal speedup. The root cause is the limitation in the capturing of the frame, which necessarily has to be performed sequentially, and then limits the speedup achievable. The main overhead introduced in the parallel versions resides in the retrieval from the memory of the frames, which require a significant number of clock cycles compared to the actual time needed to process the frames.



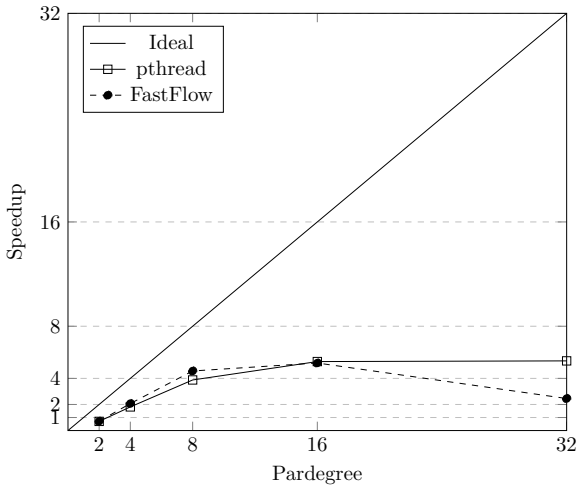
Plot 1: Tests performed on the remote machine to measure the speedup of the pthread version applied to videos of different resolution and duration.



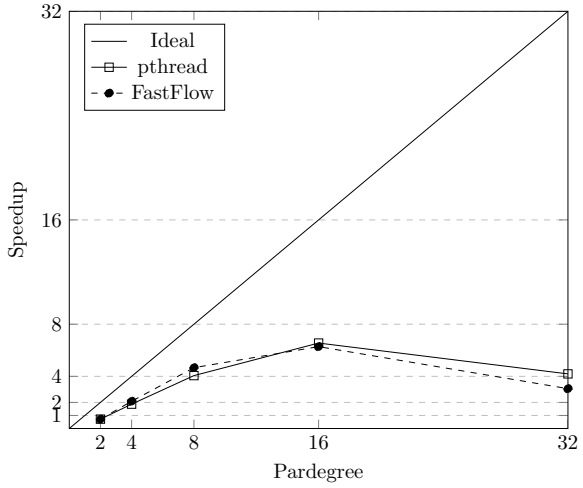
Plot 2: Tests performed on the remote machine to measure the speedup of the FastFlow version applied to videos of different resolution and duration.



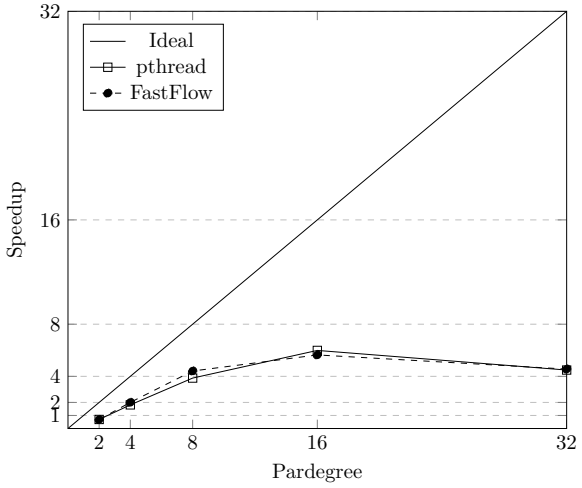
(a) Test video 2: resolution 480x270



(b) Test video 2: resolution 640x360

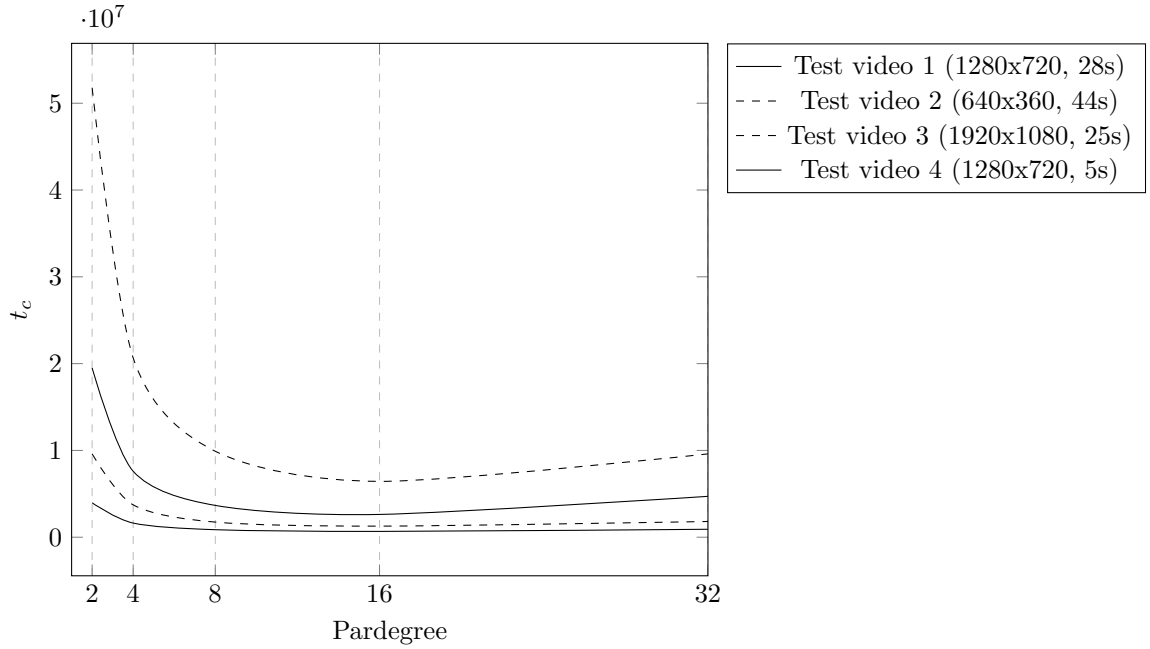


(c) Test video 2: resolution 1280x720

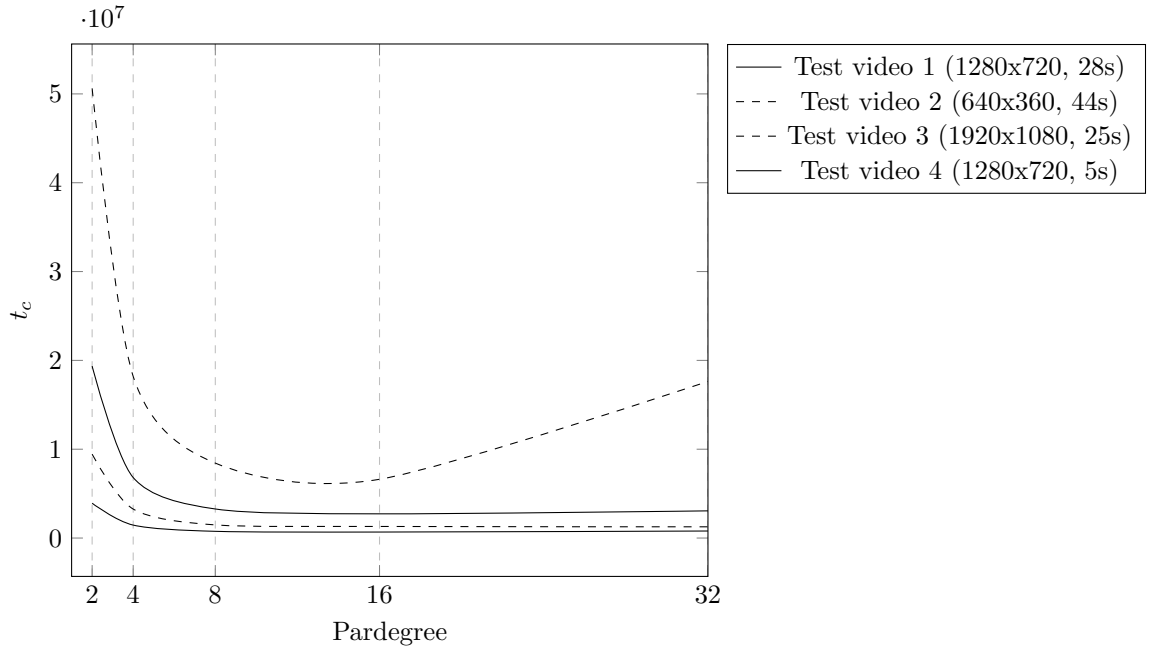


(d) Test video 2: resolution 1920x1080

Plot 3: Tests performed on the remote machine to measure the speedup of the pthread and FastFlow version applied to the same video with different resolutions.



Plot 4: Pthread parallel version applied to different videos. The plot compares the completion time and the pardegree.



Plot 5: FastFlow parallel version applied to different videos. The plot compares the completion time and the pardegree.

7 Instructions

In order to compile and run the application, it is needed to perform the following steps:

1. Move to project main folder
2. Execute the command `cmake .`
3. Execute the command `make`
4. At this point the application is compiled and ready to run. The following commands allow to execute the sequential, pthread or FastFlow version, respectively:
   - `sequential_motion_detection path-to-video k`
   - `pthread_motion_detection path-to-video k pardegree`
   - `ff_motion_detection path-to-video k pardegree`