# Video Motion Detection Project

Parallel and Distributed Systems Paradigms and Models

Jacopo Bandoni


Dipartimento di Informatica
Università di Pisa
24/08/2022

# Contents

# 1 Introduction

In this report will be discussed the parallelization of a video motion detection algorithm. To perform this task the program will perform an operation of gray scaling and blurring on every frames. The first frame of the video after passing through those transformations will be used as background and it will be compared with the other frames to check whether there is any movement.

## 1.1 Problem Structure

In more details, we can outline the structure of the algorithm in 4 different stages that will be repeated for every frame:

- ***Stage1***: reading a frame from the video.
- ***Stage2***: applying a gray scale conversion, by substituting each (R,G,B) pixel with a grey pixel which is the average of the R, G, B values.
- ***Stage3***: applying a smoothing transformation, by substituting each pixel with the average of the surrounding pixels. This operation will be controlled by a *kernel_size* parameter.
- ***Stage4***: detecting motion, by checking that at least $k$% of image pixels differ from the background image pixels. The threshold $k$ of the percentage will be controlled by a *percentage_threshold* parameter.

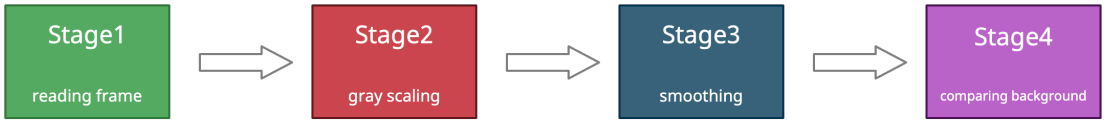


Figure 1: Problem structure

The output of the program will then be the number of frames where motion is detected.

## 1.2 Goal

For this problem was assumed that the algorithm receive the video in streaming; therefore, as a metric to optimize, was chosen the service time. In particular, having access to a 32 cores machine, the parallel programs will focus on gaining performance on all available cores. A program to gain better performance on fewer cores will be proposed, but it will not be the focus of the discussion in this report.

## 1.3 Context

The experiments will be tested with different input configurations.

- For the video it will be analyzed the ***owl.mp4*** video: a 251 frames fullHD resolution video.
- for the smoothing algorithm will be tested two different **kernel size**, of 3 and 7.
- the **percentage threshold** value will be left fixed at 95% since its variation does not have a significant impact on the performance of the program.

Therefore 2 configurations will be discussed in the report.

# 2 Performance Analysis

This section explores various approaches to solve the previously illustrated problem. Initially it is analyzed the sequential program: to see how does it change by changing the *kernel_size* and to look at the property of the single stages. Then several skeleton are proposed using performance models and single stage time data.

## 2.1 Sequential Analysis

To provide measurements of the stages in the sequential execution, experiments were repeated 10 times for each one of the configuration and the result was averaged to provide more significant results.

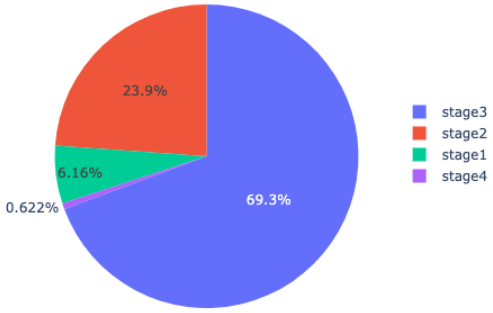


Figure 2: $kernel\_size = 3$

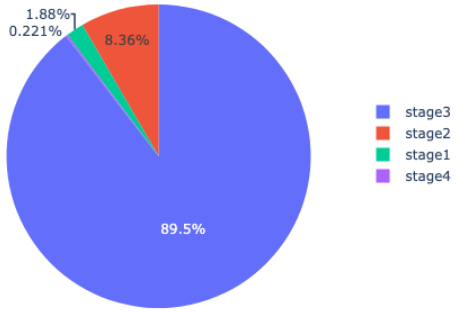


Figure 3: $kernel\_size = 7$

Note that to implement *Stage1* we will use an off the shelf solution taken from the *OpenCV* library. Therefore, in case a program will have a service time that match the time spent on *Stage1* bottleneck, we will assume that the program can no longer be parallelized.
By looking at the pie charts it is possible to observe how:

- As $kernel\_size$ grows, *Stage3* occupy more computation time.
- *Stage4* is negligible, therefore is not worth to parallelize. It will be merged together with *Stage3* and, from now on, with *Stage3* will be reported as both the blurring an the comparison stage.

## 2.2 Skeletons Proposal

From now on various skeleton will be proposed and their performance will be estimated:

**Normal Form**

$$Farm(Stage1, Comp(Stage2, Stage3)) \tag{1}$$

Where *Stage1* is the emitter providing the frame to be processed by the workers of the farm. The advantage of this proposal is that it has minimal communication overhead, just the one between the emitter and the workers and can exploit memory locality for the operation inside each worker. Furthermore, by looking at the performance model (with $n$ equals to the number of workers):

$$T_s = max\left\{T_{Stage_1}, \frac{(T_{Stage_2} + T_{Stage_3})}{n}\right\} \tag{2}$$

we can see how it is able to consistently lower service time as it is, by adding new workers to the farm, up to the emitter service time.
Note that another source of overhead is represented by the synchronization cost. Infact each workers of the farm need to claim the access to a counter, before increasing it, whenever a frame with motion is detected. An alternative was to use a collector responsible to sum the counters. However this choice is avoided, because it would lead to other sources of overhead and it would use an additional thread. This decision is made even for successive approaches and too simplify analysis it will not be considered in the performance models.

**Stage3 Farm** In this solution we merged *Stage1* and *Stage2* into a single sequential state that act as emitter of a farm, where each worker is a *Stage3*:

$$Farm(Comp(Stage1, Stage2), Stage3) \tag{3}$$

As shown by the following equation:

$$T_s = max\left\{T_{Stage_1} + T_{Stage_2}, \frac{T_{Stage_3}}{n}\right\} \tag{4}$$

this solution can let the service time be limited to the service time of $T_{Stage_1} + T_{Stage_2}$, however, in situation where this value isn't reached, this solution could provide additional performance.
To keep improving the service time, in case the bottleneck is reached, we can apply a rewriting rule, obtaining the following skeleton:

$$Pipe(Stage1, Farm(Stage2), Farm(Stage3)) \tag{5}$$

This solution could theoretically lead to good performance by breaking the bottleneck of the previous first stage but, in practice, those are hampered by the overhead. Infact data would need to pass across multiple stages, it would need to be accessed concurrently and memory locality would be little exploited.

## 2.3 Measurements

In this part it will be looked at the measurement time of the single stages for the various configurations and using previously presented performance models it will be given estimates and bounds on the service time.
In the following table are shown the exact execution times required to process each stage:

| *kernel_size* | *Stage1* | *Stage2* | *Stage3* |
|---|---|---|---|
| 3 | 5140 | 22626 | 75542 |
| 7 | 5216 | 22591 | 304223 |

Table 1: Stages execution time in $\mu s$ for the different *kernel_size*.

With both $kernel\_size = 3$ and $kernel\_size = 7$ it is applied the performance equation 3 of the normal form.

$$T_s = max\left\{5140, \frac{22626 + 75542}{n}\right\} \text{ for } kernel\_size = 3$$

$$T_s = max\left\{5216, \frac{22591 + 304223}{n}\right\} \text{ for } kernel\_size = 7$$

With those formulas it is possible to define a lower bound on the number of workers necessary to reach the service time bottleneck of *Stage1*. Infact would be needed at least 20 workers for $kernel\_size = 3$ and at least 63 workers for $kernel\_size = 7$. Given that the machine that will be used in the experiments is composed by 32 cores we can assume that for $kernel\_size = 7$ the bottleneck will not be reached.
The second solution (3) has a worst potential improvement. The service time is bounded by $T_{Stage_1} + T_{Stage_2}$ (4). Nevertheless until the service time of the workers reach this bottleneck this solution should provide better performance. Infact following the performance model for $kernel\_size = 7$ the following inequality is true:

$$T_s = max\left\{5216 + 22591, \frac{304223}{n}\right\} \leq max\left\{5216, \frac{22591 + 304223}{n}\right\} \text{ for } n\_workers \leq 11 \tag{6}$$

# 3 Implementation

In this section will be better described the implementation choices for the previous proposed skeletons and it will be indicated how to execute the code related to those implementations.

## 3.1 Details

To implement the previous described algorithm were used two different approaches: a standard parallel programming paradigm by using thread and FastFlow, an high level parallel framework.

### 3.1.1 Thread

For this section was chosen to implemented just the *Normal Form*, using the standard C++ library. A *SharedQueue* class is used to handle the communication and contention of the frames, implemented using lock and condition variable. While as already discussed, an atomic variable is used to handle synchronization between workers, for whenever they need to increase the value of the counter of frames with motion.

### 3.1.2 Fast Flow

With Fast Flow was implemented both the *Normal Form* solution and the solution of *Stage3 Farm*. Those skeleton are implemented both using on demand scheduling and Round-robin scheduling. Even here it is used an atomic variable to count occurrences of frames with motion.
To be more concise Normal Form implementation will be also refereed to as first Fast Flow implementation (es.: ff1), while the *Stage3 Farm* as second fast flow implementation (es.: ff2).

# 4 Documentation

This section explain how the project code is structured, how to compile execute, gather data about executions and visualize the results.

## 4.1 Project Structure

Here there are the main files we can found inside the project folders:

```
analyses/ :  data about execution and notebooks that analyze them ;
  data/ :  data about execution;
  parallelization_3.ipynb:  kernel 3 program analysis;
  parallelization_7.ipynb:  kernel 7 program analysis;
  stage.ipynb:  sequential program analysis;
  ...:  other files
build/
scripts/ :  for data gathering;
src/ :  source code;
  FrameComputation.cpp
  Seq.cpp
  ThreadPar.cpp
  ffPar.cpp
  ffPar2.cpp
  ...:  other files
video/ :  video sample;
main.cpp
...:  other files
```

## 4.2 Compilation

By executing the following command a *main* executable will be built inside the build folder:

```
$ cd build
$ cmake .. -DCMAKE_CXX_COMPILER=g++-10
$ make
```

## 4.3 Execution

Inside the build folder it is possible to run all the previous listed parallel programs with different configurations by passing different value to the arguments:

- 1° argument: the program version.
  - 0: for sequential program
  - 1: for thread program
  - 2: for fast flow 1 program
  - 3: for fast flow 2 program
- 2° argument: the number of workers.
- 3° argument: the kernel size.
- 4° argument: the percentage threshold.
- 5° argument: the name of the video file (of the one inside video directory).

```
$ cd build
$ ./main 1 2 7 95 owl.mp4
```

To gather execution data regarding sequential execution described in the Sequential Analysis section or to gather data about parallel execution we use two different scripts.
In this ways it is possible to conveniently analyze the correctness and characteristic of the programs.

```
$ cd scripts
$ ./stats_parallelization.sh # gather parallel program data
$ ./stats_stages.sh # gather sequential program data
```

The data produced by those script is saved in the *analysize/data* subfolder and it can be analyzed by the python notebooks present in the analysis folder.

# 5 Results

In this section it will be shown how the implemented algorithm perform in practice using the configurations shown in the previous chapters. Initially the various fast flow approaches are compared and in the end the analysis is extended to the standard thread approach. The discussion is performed with both kernel size.

## 5.1 Kernel Size 3

By looking at the following plot (5.1) we can observe how the second version of fast flow program perform a little bit faster up to 5 workers. But then, when the bottleneck of the emitter is reached, the service time stop decreasing and the speedup stay flat.
We can note how the scheduling strategy do not make too much difference in this case, but being the round robin version slightly better in absolute term (it has the maximum speedup), it is used as comparison with the thread version.

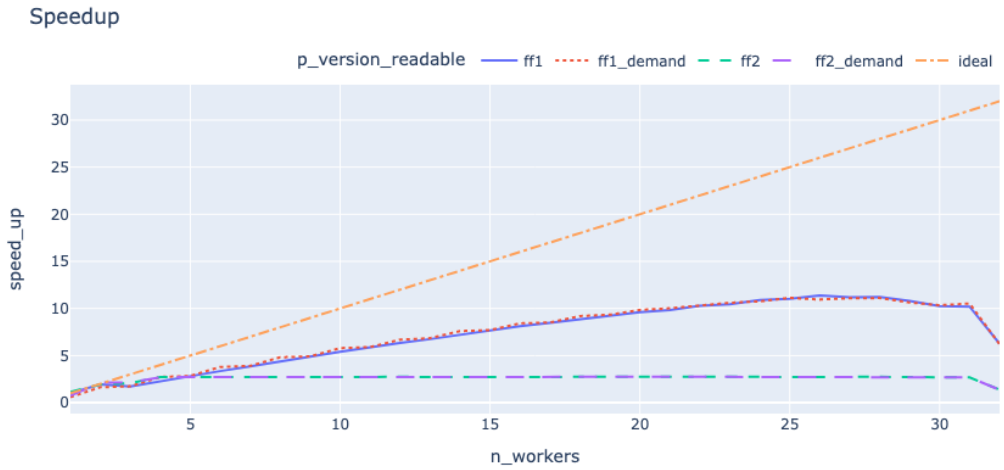


Figure 4: Speedup of fast flow algorithms with on demand scheduling and round robin.

From the comparison between the thread version we can check how initially it has a better speedup. But as *n_worker* increases, the gap get closer.

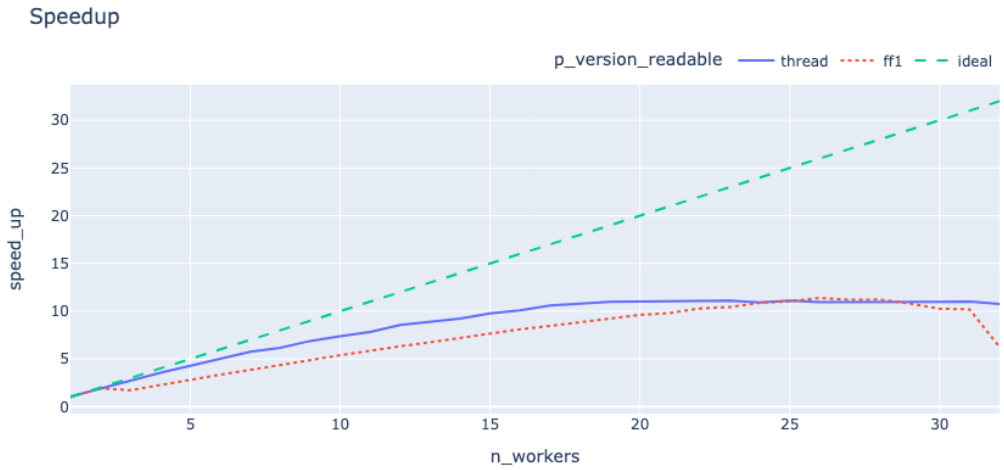


Figure 5: Speedup of the fastest fast flow algorithm compared to thread.

Infact by looking at the following table (3) we can check how fast flow provide a lower service time.

| | Threads | Fast Flow |
|---|---|---|
| Minimum Ts | 9358 | 9168 |
| Reached with #threads | 22 | 26 |

Table 2: Minimum Service time $T_s$ achieved by the models with the numbers of threads needed.

The fact that at increasing workers the performance do not increase it makes think that the emitter bottleneck is reached at about the 25° worker. However the observed service time is still a little bit far from the theoretical lower bound of 5140 $\mu s$. This discrepancy is caused by the overhead of the parallel computation.
They can be identified various components that play a role in the total overhead:

- the completion time of the single stages measured in the sequential version exploit memory locality. Unlike the parallel version.
- the communication between the emitter and the workers can be a major source of overhead.
- conversely the synchronization overhead between the workers on the atomic variable do not have a major impact on performance. This was tested by providing to a program a *percentage_treshold* so high that there were no frame detected as motion and it was observed that the completion time do not change significantly.
- instead, another source of overhead could be related to the *opencv read* function. Infact when this is executed, it calls some threads which, while not using the processes heavily, can lead to context switches that can degrade performance.

## 5.2 Kernel Size 7

For kernel equal to 7 the same steps are followed to analyze the various algorithms. As described before in the inequality 6, the second fast flow version provide slightly better performance up to 15 workers, until it matches the emitter bottleneck.
Here the scheduling makes a substantial difference, being the on demand one much better.

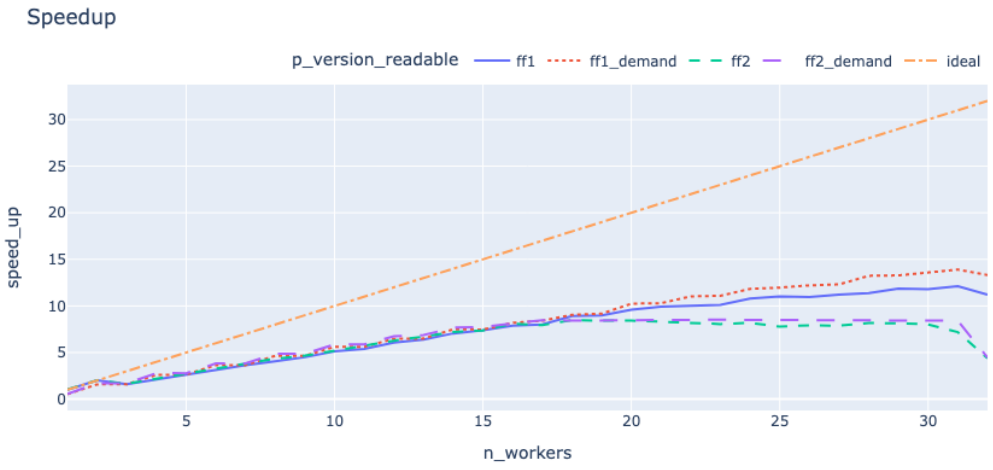


Figure 6: Speedup of fast flow algorithms with on demand scheduling and round robin.

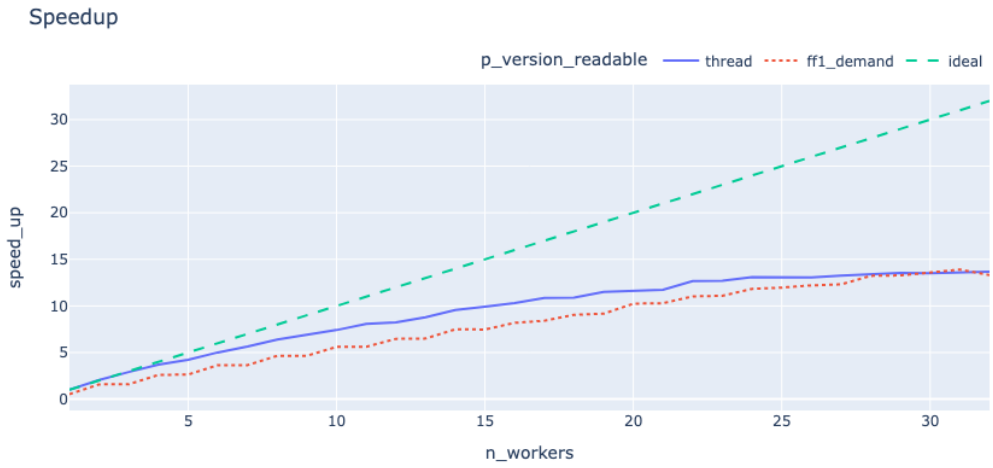


Figure 7: Speedup of the fastest fast flow algorithm compared to thread.

Concerning the comparison with the thread, the performances is similar to the $kernel\_size = 3$ version: there is an initial gap in performance between the two versions and as the number of workers increases the gap get closer. With fast flow that in the end provide the best absolute performance.
In this case the speed up keep increasing as it grows the number of workers, the bottleneck as already discussed is not reached and further speedup would be possible with additional core in the hardware.
In the following table is possible to look at the comparison of the service time between the two approaches:

| | Threads | Fast Flow |
|---|---|---|
| Minimum Ts | 25391 | 24964 |
| Reached with #threads | 32 | 31 |

Table 3: Minimum Service time $T_s$ achieved by the models with the numbers of threads needed.