PARALLEL AND DISTRIBUTED SYSTEMS: PARADIGMS AND MODELS

ACADEMIC YEAR 2021-2022

REPORT OF THE FINAL PROJECT VIDEO MOTION DETECT

DOMENICO ERRIQUEZ

Università di Pisa

Dipartimento di Informatica

1. Introduction

The goal of the following project is to parallelize the motion detect of a video using two approaches, one with Pthreads and one with Fastflow. The motion detection consists of the following steps: to take the first video's frame as a background and then to each frame apply first a grey filter, then a blur filter and then compare it with the background and check if there is a difference in terms of pixel, if this difference is more than a certain threshold then it means that there is a motion. At the end of the execution the program has to report the number of frames that are different from the background. It is a stream parallel problem, since the input is a stream of frames, so the goal is to minimize the service time.

2. Measures

The first step is to implement the sequential version in order to be able to measure the time each function takes to be executed. In the table below is reported in microseconds the time each function takes to be executed.

Video	Read frame	Grey filter	Blur filter	Check motion
720p	2600,8 µs	4071,4 µs	20127,6 µs	536,1 µs

The first observation is that the Read frame is the only function that cannot be parallelized, so based on this the service time cannot be below 2600,8 µs

3. Possible skeletons

The second step is to analyze different skeletons and to try to minimize the service time.

- ## Pipeline (read, farm(grey), farm(blur), motion)

  In the pipeline each stage computes a function which leads at the end to count the number of frames where there is a motion. The first stage reads the frame from the video and sends it to the second stage that will apply to it a *Grey* filter and will send it to the third stage. In the third stage there is a farm where each worker computes the function *Blur*, so each worker takes in input the grey frame from the stage 2 and computes a *Blur* filter to it and then it sends it to the

last stage. The last stage compares the frames with the first frame of the video and checks if there is a motion. The first problem of using this skeleton is to not take advantage of the locality; the second problem would be that by increasing the number of threads for the farms, would automatically lead to an increase of the time needed to send the frames from one stage to another, as more threads will try to access the same queue at the same time. In the farms there will not be any added emitter and collector, but it will be the previous stage to send the frame directly to the next stage that computes the function. So, there will be 1 thread for the *Read* function, and 1 thread for the *Motion* function, while the threads for the *Grey* function and the *Blur* function will be variable.

$$T_S\,(Pipeline) = max\left\{Tread,\ \max\left\{T_e, \frac{Tgrey}{nw'}, T_c\right\},\ max\left\{T_e, \frac{Tblur}{nw''}, T_c\right\},\ Tmotion\right\}$$

$$T_S\,(Pipeline) = max\left\{2600{,}8,\ max\left\{1, \frac{4071{,}4}{nw'}, 1\right\},\ max\left\{1, \frac{20127{,}6}{nw''}, 1\right\},\ 536{,}1\right\}$$

So, with this approach using 2 threads for the *Grey* function and 8 threads for the *Blur* function, the bottleneck is the *Read* function and the service time will be 2600,8μs using in total 12 threads.

- Farm (comp (grey, blur, motion))

In the farm, the functions are computed sequentially by the same thread, in this way there is not overhead to send frames from one stage to another. The most important thing is that is possible to take advantage of the locality, since the thread that applies the different functions, will probably already have the frame uploaded in its cache.

$$T_S(Farm) = max\left\{Tread,\ \frac{Tgrey + Tblur + Tmotion}{nw}\right\}$$

$$T_S(Farm) = max\left\{2600{,}8,\ \frac{4071{,}4 + 20127{,}6 + 536{,}1}{nw}\right\}$$

Therefore, using 1 thread for the *Read* function and using 10 threads for the Farm, the bottleneck will be again the *Read* function. The service time, therefore, will be 2600,8μs, using in total 11 threads.

After analyzed the two approaches, the implementation will be based on the second skeleton(Farm) because there is no overhead to send frames from one stage to another one, it is possible to take advantage of the locality and the service time will be 2600,8μs, using less resources.

4. Implementation (Pthreads, Fastflow)

In this chapter it is described the implementations with Pthreads and Fastflow.

4.1 Implementation with Pthreads

In the implementation with Pthreads the main thread, that from now on will be called also emitter, it is the one that has the task to read the video and send the frames to the threads.

The emitter uses the class VideoCapture from OpenCV to read the video and a Boolean variable called endVideo that will become true when the emitter will read an image that is empty.

At the beginning, the main thread reads the first frame and applies to it the *Grey* and Blur filters and saves this frame in a Mat variable called background; afterwards it will create the threads and push them in a vector of threads.

Then the emitter will start reading the video, so while the video is opened and not finished(endVideo=false), the emitter read the frames from the video and pushes them in a queue of Mat. The access at this queue is protected by a mutex, in this way more threads cannot access to the queue at the same time. The emitter uses a condition variable to notify the threads that a frame has been pushed in the queue.

From the other side, each thread tries to access the queue to pop the image and this operation is protected by the mutex. In case the queue is empty and the video is not finished they wait that the emitter sends them a notification that a frame has been pushed in the queue. At this point the thread that receives the notification obtains again the lock on the mutex and checks again if the queue is not empty and then it can proceed to pop the frames, unlock the mutex, apply the filters on the frame and check if there is a motion compared to the background. Each thread has a local variable where it increases its value if there is a motion in the frame it is computing. At the end, when the queue is empty and the Boolean variable endVideo is true, each thread will update the atomic variable with the value each thread has saved in the local variable.

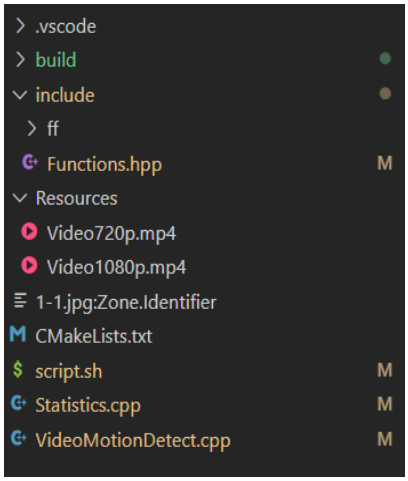
4.2 Implementation with Fastlow

About the Fastflow's implementation, it has been implemented two sequential nodes. The first one is the emitter and its type is ff_monode_t<Mat> since its job is to read the video and send the frames to multiple workers, this will stop when the emitter will read an empty image and in this case will send EOS to the nodes. The other sequential node is called worker and its type is ff_node_t<Mat> since it receives in input a frame. Each worker applies the filters to the frame received from the emitter and checks if there is a motion, in case there is a motion, the worker updates its local variable. When the worker receives EOS, it will terminate the computation and will compute the svc_end where it will update the atomic variable with the value each worker has saved in the local variable.

These two sequential nodes are used in a ff_Farm where has been removed the collector and the first sequential node is the emitter of the farm.

5. Documentation

5.1 Structure



On the image above is reported the structure of the project.

- **Include folder:** In this folder there are the Functions.hpp file and the ff folder. The Functions.hpp contains the functions used to calculate the motion of the video, like *Grey* and *Blur* filters, and the function to check if there is a motion in a certain frame. In the ff folder there are the headers for the Fastflow library.
- **Resources folder:** In this folder there are 2 videos, one is 720p and the other one is 1080p

- **Statistics.cpp:** It is a source file that can be used to compute the measures of the functions
- **VideoMotionDetect.cpp:** It is the file to detect the motion of the video and to visualize the completion time.

5.2 Compile

To compile the project it is enough to open the terminal and execute the following instructions:

```
$ cd VideoMotionDetect/
$ mkdir build && cd build
$ cmake ..
$ make
```

5.3 Execute

There are two files that can be executed, the first one is the VideoMotionDetect and the second one is Statistics.

5.3.1 VideoMotionDetect

This file cpp takes 4 inputs:

1. **Number of threads**(to execute the program in sequential mode use 0 as parameter)
2. **Fastflow:** It is possible to choose if to run the program using Fastflow or C++ threads, use 0 for C++ threads computation or 1 for Fastflow computation
3. **Scheduling ondemand:** For the Fastflow execution is possible to choose if the scheduling is ondemand or not. With 0 is not ondemand, with 1 is ondemand.
4. **Path video:** the path of the video to check the motion

   Example to run the program using 16 threads in Fastflow with a scheduling ondemand on the 720p video, assuming the position of the terminal is in the folder build.

```
$ ./VideoMotionDetect 16 1 1 ../Resources/Video720p.mp4
```

5.3.2 Statistics

This file cpp takes 2 inputs:

1. **Show video:** This value can be 0 or 1. 0 means to not show the video in output, with 1 instead it's possible to see the video in output and on terminal will be displayed simultaneously if there is a motion in that frame. This option works only with sequential code
2. **Path video:** the path of the video to check the motion

   Example to run the program showing on display the video 1080p assuming the position in the terminal is in the folder build

```
$ ./Statistics 1 ../Resources/Video1080p.mp4
```

6. Results

The following graphics are the results of the implementation described in the chapter 4. and it uses a 720p video. The machine used to execute the program is provided with Intel(R) Xeon(R) Gold 5120 CPU 2.20GHz with 32 cores. In the graphics for the performances of Fastflow is represented the implementation using a scheduling non on demand. About the number of threads showed in the graphics, they refer to the number of added workers used, without being taken into consideration the thread that is used to read the video, this is why with Fastflow using 32 threads there is a drop of performance, because in reality the threads used are 33(one being the emitter)

6.1 Service time

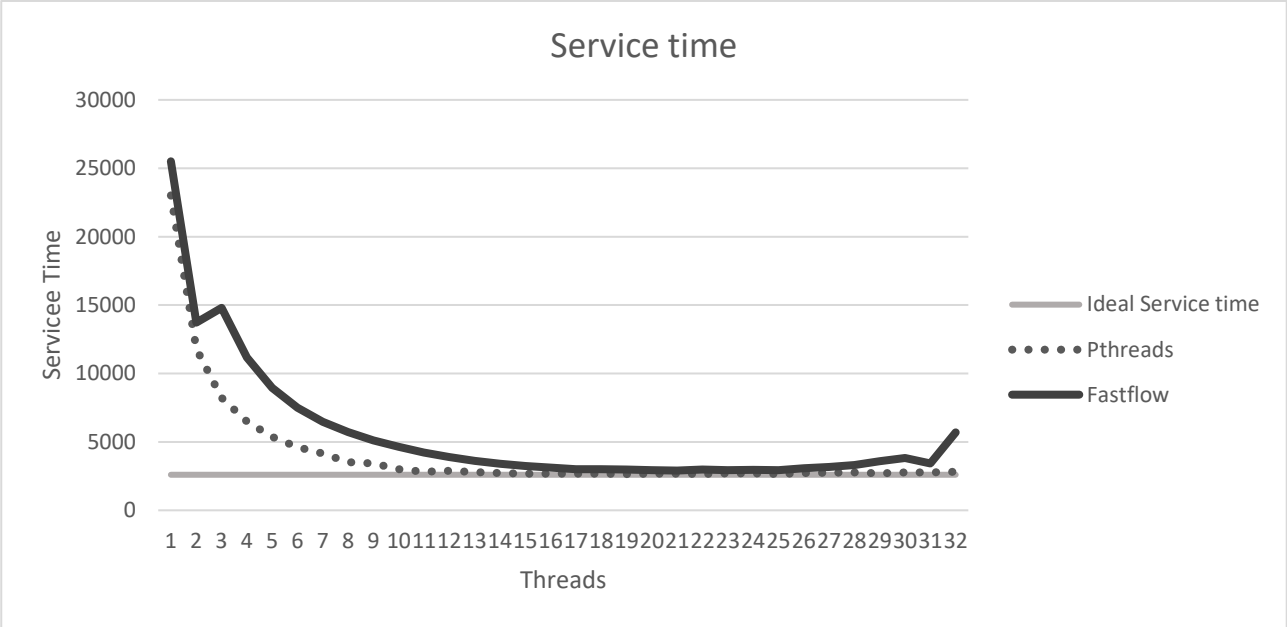


Figure 1: Service time

On the figure 1 is it possible to see that the service time gets close to the one that has been predicted in the chapter 3. It's not exactly the service time predicted due to the following overheads: at the beginning there is the setup of the program where the main thread creates the threads and takes the first frame as a background and compute the filters on it. The time of this overhead has an estimated value of 151982,7 μs. Since the 720p video is made of 805 frames, it can be amortized and say that each frame has an overhead of 188,80μs. Another overhead in case of the Pthreads implementation is that increasing the number of threads, there are more threads that try to lock the mutex to pop the frame, in case there is a frame in the queue, otherwise the thread will be set to sleep mode and it will wait to be notified by the emitter.

6.2 Speedup

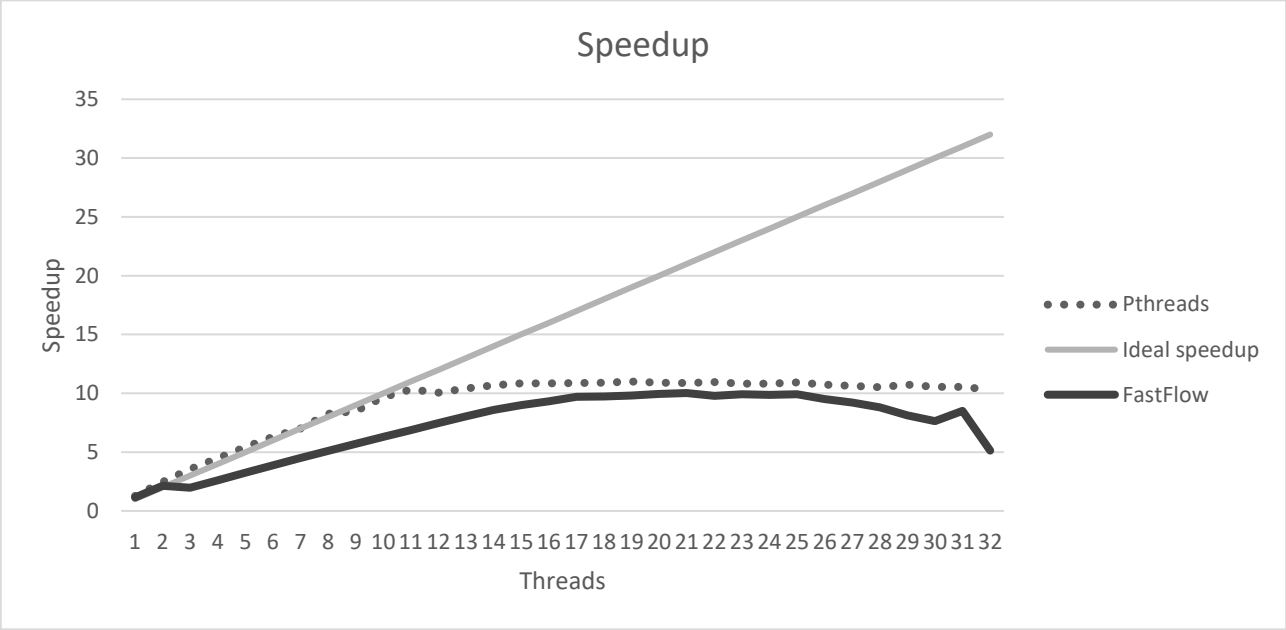


Figure 2: Speedup

6.3 Efficiency

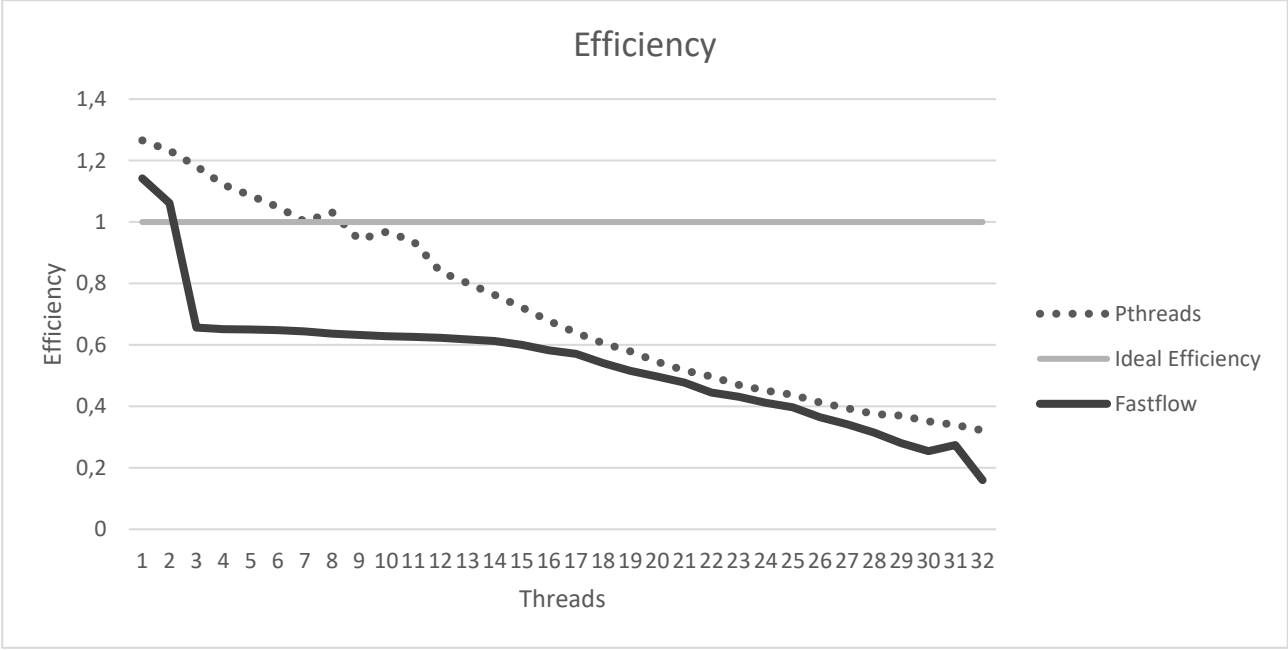


Figure 3: Efficiency

6.4 Scalability

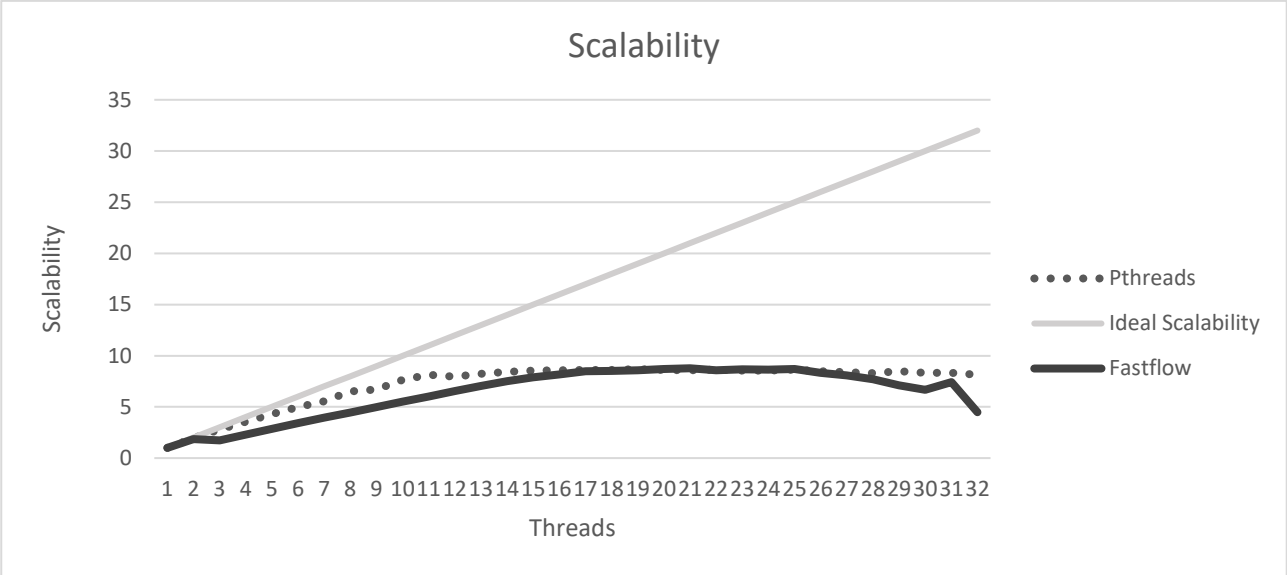


Figure 4: Scalability

6.5 Completion time

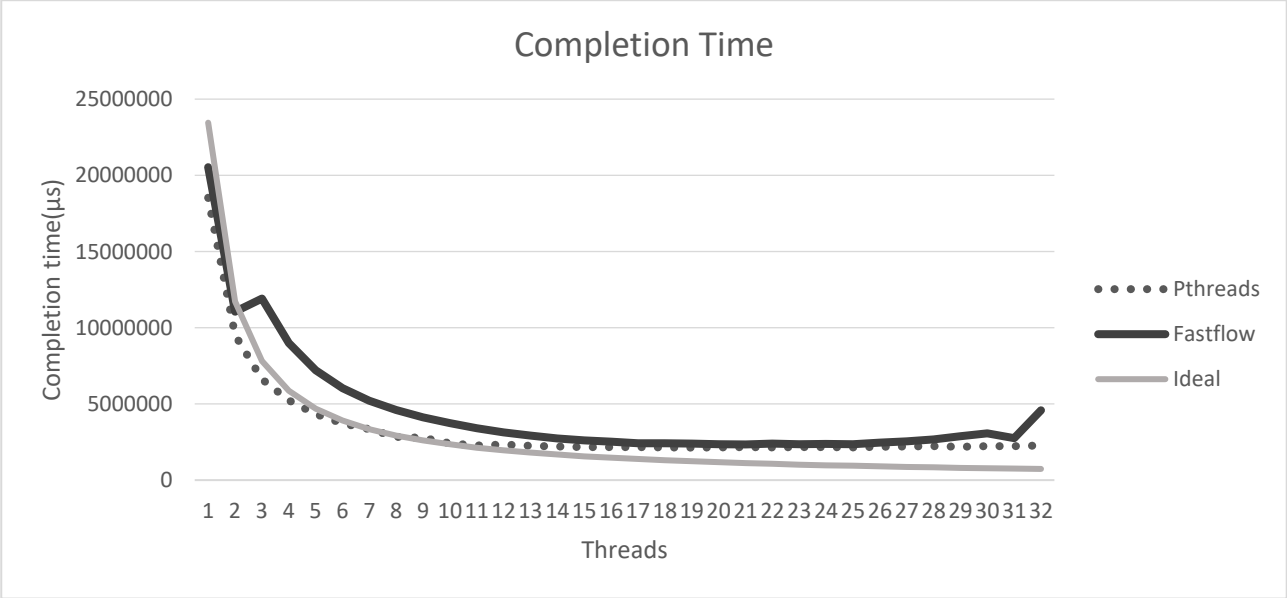


Figure 5: Completion time

6.6 Fastflow with scheduling ondemand and not ondemand

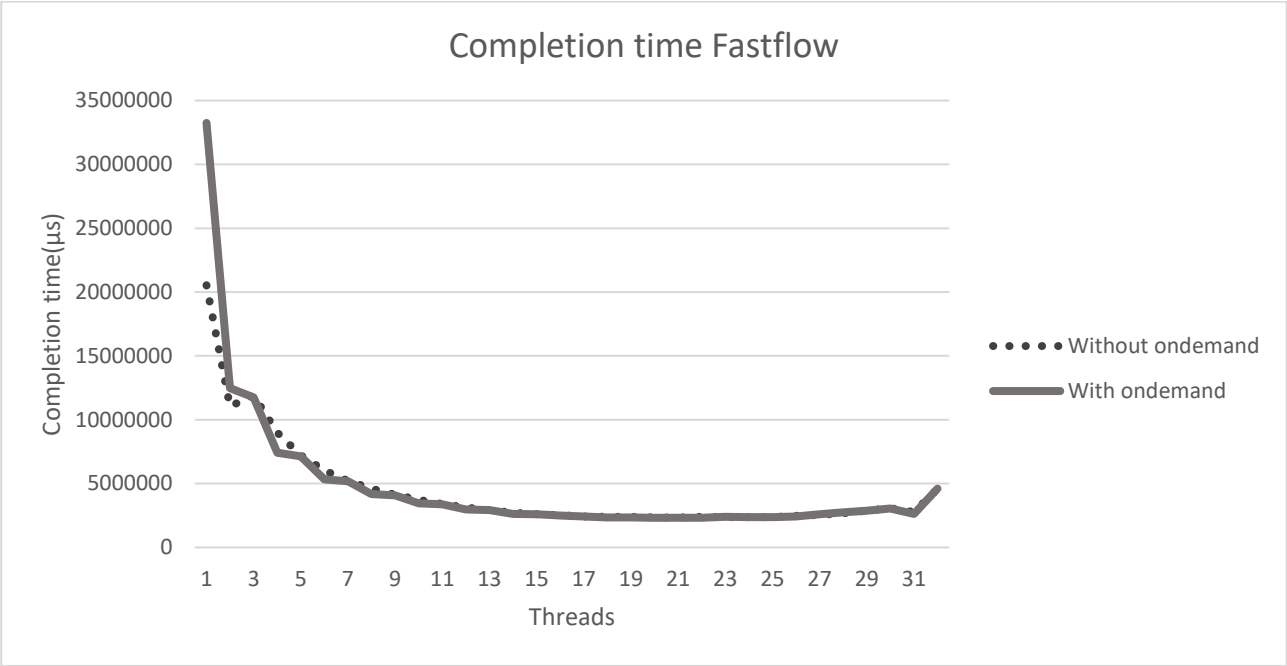


Figure 6: FastFlow ondemand and not ondemand

The execution with the scheduling ondemand with one or two workers is clearly slower because the emitter has to wait that the workers empty their queue in order to push the frame.

Otherwise, with few workers, the pseudo scheduling ondemand works better and assigns the frames to the workers that have been faster and have emptied their queue.

With more cores, the completion time converges and there is no difference between the 2 scheduling approaches.