Università di Pisa
Dipartimento di Informatica

Parallel and Distributed Systems: Paradigms and Models

Student: **Samuele Intaschi**
ID: **523864**

Parallel Video Motion Detect

Contents

1 Introduction **1**

2 Project Structure **2**

3 Design choices **3**
3.1 Sequential implementation 3
3.2 Native C++ threads implementation 3
3.3 FastFlow implementation 4
3.3.1 Master-Worker implementation 5
3.3.2 Farm implementation 5

4 Sources of Overhead **6**

5 Results **6**
5.1 Results obtained without thread pinning 7
5.2 Results obtained with thread pinning 8
5.3 Differences in times without vectorization 9

6 Execution **9**

1 Introduction

This report is organized as follows: the second section describes the content of the program folder, the third section presents the main design choices taken during the implementation of the application, the fourth section describes the principal sources of overhead, the fifth section presents the results obtained and finally the last section explains how to use this application.

2 Project Structure

There are four implementations of the application: one is the sequential application, one is the parallel application which uses native `C++` threads, and the others are two parallel applications which use the *FastFlow* programming library. The contents of the application folder are as follows:

- `sequential.cpp`: source code of the sequential implementation.
- `nthreads.cpp`: main source code of the `C++` threads implementation.
- `ffmw.cpp`: main source code of the *FastFlow* implementation that uses a *master-worker* pattern.
- `fffarm.cpp`: main source code of the *FastFlow* implementation that uses a *farm* pattern.
- `Makefile`: the commands to compile the applications.
- `nthreads` folder: contains files with code that implements classes used for the implementation with native `C++` threads, that are:
  - `greyscale_converter.hpp`: implements the functions to convert a RGB channels frame to black and white and to obtain the pixel average intensity of a black and white frame.
  - `smoother.hpp`: implements the functions to apply an average filter with a 3x3 kernel on a greyscale frame.
  - `comparer.hpp`: implements the functions to compare a frame with the background and to count the different pixels between the two frames.
  - `thread_pool.hpp`: implements the functions to synchronize all worker threads.
- `fastflow` folder: contains the files with the code to implement the classes used for the implementation with *FastFlow* and they are divided in subfolders, that are:
  - `mw`: contains program files for the *FastFlow* implementation that uses *master-worker* pattern:
    - `ff_emitter.hpp`: reads video frames and submits them to the master.
    - `ff_master.hpp`: receives video frames and dispatches them between the *workers*, then collect the results.
    - `ff_worker.hpp`: takes a video frame and applies a function on it, then returns the resulting frame.
  - `farm`: contains program files for the *FastFlow* implementation that uses *farm* pattern:
    - `ff_emitter.hpp`: reads video frames and submits them to the farm.
    - `ff_farm_worker.hpp`: takes a frame and applies a composition of functions on it.
    - `ff_collector.hpp`: receives the percentage of different pixels between the frames and the background, compares them with the value specified by the user, and finally updates the total number of frames with detected motion.
- `utils` folder: contains the files with the classes useful for all the implementations:
  - `seq_greyscale_converter.hpp`: implements the functions to convert a frame to greyscale, used to convert the background frame at the beginning of the execution.
  - `seq_smoother.hpp`: implements the functions to apply a smoothing filter to a frame, used to smooth the background frame at the beginning of the execution.
  - `file_writer.hpp`: implements a method to write the results to a file in the `results` folder.

- `results` folder: contains the results, organized in `txt` files with the name of the video to which they refer, and a *bash* script to parse these files:
  - `get_avg_times.sh`: gets, for each video, the average execution time per number of threads, filtered for specific implementation and `k` parameter.
- `results.cpp`: it is a script to automatize the execution of the application with different parameters, used only for tests.

3 Design choices

In this section the main implementative choices are described for each of the implementations. Some optimizations, like vectorization, are used for all implementations, for the benefit of all, as it is described in section 5.3.

3.1 Sequential implementation

The sequential implementation takes as parameter the fraction of pixels that must be different to detect a motion, then it starts taking the first frame as background, converting it to grayscale, performing smoothing on it and finally computing the average intensity of the pixels to establish a threshold for background subtraction, which is $\frac{1}{10}$ of this intensity. This threshold is used to ignore the small changes in pixels due to exposure.
After these initial operations, the program starts reading the frames of the input video and performing for each of them, as for the background, the conversion to greyscale and smoothing. The filter used for smoothing computes the average of the 3x3 neighborhood of the pixel and its application is implemented by summing the pixels and dividing the result by nine. When the matrix, which represents the pixels of the frame, is in greyscale and has the smoothing filter applied, it is subtracted from the background matrix and the pixels that differs more than the threshold are counted. Their fraction of the total is computed to be compared with the percentage value specified by the user to decide whether there is a movement in the frame or not. Specifying some flags, the program can print the average time spent for each phase, to better understand where bottlenecks are possible and where it is possible to improve using parallelism.

3.2 Native C++ threads implementation

This implementation aims to parallize the operations of the sequential implementation to achieve better performances, using native `C++` threads. Data parallelism is possible for each of the three phases and initially it was decided to use it in each phase, considering that operations on large matrix are performed, but it was immediately evident that the use of threads for data parallelism did not give the expected benefits, because the operations were very fast with only vectorization.
Next it was considered to create specific threads for each phase and put them in a *pipe* pattern, so threads for greyscale conversion, threads for smoothing and threads for background subtraction, with queues in between them. There was another problem: the three operations had very different latencies and the smoothing phase was much slower than the others, as it is visible in figure 8, so the greyscale conversion threads ended early their work and they were wasted threads from this point on, while the smoothing threads were full of work. Similar considerations could be made for the background subtraction threads, and using data parallelism to speed up the smoothing phase was not enough.
For these reasons, the structure that it is decided to implement is similar to a *Macro Data Flow* pattern, with a pool of threads, the number of which is specified by the user, which take executable tasks from a queue and execute them, whatever they are. In this way the threads works constantly if the *emitter*, which is the only non-parallelized part (it comes from the `OpenCV` library), can sustain the speed with which the threads takes tasks.
The user can also decide, when he starts the program, to have each thread mapped on a core

and the mapping is linear, but this is not the default behavior.
An executable task is represented by a function associated with a parameter and the number of the frame with which the function works. The possible functions are grayscale conversion, smoothing and background subtraction. In the first two cases the function is computed by the thread, the next task is put in the queue and an integer code is returned, while in the latter case there are no more tasks to execute and the returned code represents the fraction of pixels that differ between the actual frame and the background. The queue is a `priority_queue`, where the priority is given by the frame number because the goal is to produce a result for each frame analyzed as soon as possible. Instead, with a *FIFO* queue, it could be possible times when the queue is filled with tasks of only one type and this could produce delays in delivering final results, even though the completion time of the entire application would not change.
It is also introduced a limit for the main thread, that acts as an *emitter*, it reads a frame, prepare a greyscale conversion task and submit it to the thread pool, but it cannot queue its task if there are already 10 tasks. This is to prevent of running out of virtual memory with high resolution video which slow down all the operations on the frames. A scheme of the flow of execution of this implementation is shown in figure 1.

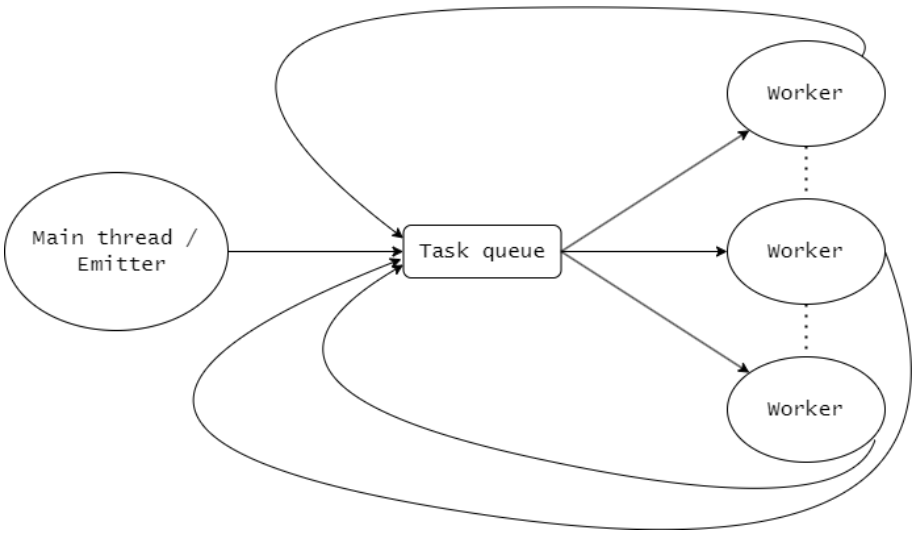


Figure 1: Schematic structure of the native `C++` threads implementation.

3.3 FastFlow implementation

The same considerations made for the implementation described in section 3.2 were made here, so it was decided to start by trying data parallelism, to see if the patterns of the *FastFlow* library, could do better. So `ff_Map` and `ParallelFor` were used for greyscale conversion and smoothing, while `ParallelForReduce` was used for background subtraction. These greatly reduced the time needed to compute the operations, in particular for the smoothing phase which was the more complex, but they still did not benefit as much as the stream parallelism. It was considered also to put data parallelism inside *workers* of a `ff_Farm` for the smoothing phase, and this gave good results, but the number of threads grew very fast, and using threads to analyze more frames, instead more threads for the same frame, gave better performances anyway.
For these reasons it was decided to realize two different implementations using *FastFlow* library, that are described in the following sections, one that is inspired by the implementation with native threads and it is realized with a *master-worker* pattern, and the other is implemented with a *farm* pattern.
For both the solutions the user can specify the number of *workers* to use and if he wants to have each thread mapped on a core with non-blocking mode, but the default behavior is to not map threads on physical cores, and, since so doing so to have multiple threads on the same

core, to use blocking mode. Moreover, as for the implementation described in 3.2, the capacity of the channels between the nodes is limited to 10 tasks, to be sure that a large number of high resolution frames do not run out of virtual memory.

3.3.1 Master-Worker implementation

This version is very similar to the implementation with `C++` native threads, it consists of an *emitter* thread, which reads video frames, prepares tasks and sends them to the *master* node, that forwards it to a *worker* node when it requests it, because the scheduling policy is *on-demand*, given that different operations require different times to be computed. The result of the computation is sent back to the master who decides whether to prepare the next task or store a final result.
When the video frames are finished, the *emitter* sends to the master the total number of them and then an `EOS` message, which master forwards to the *workers* only after receiving the results for all video frames, and this is the reason to know the total number of them.
A task here consists of a pointer to a frame and a code number that indicates the operation to be executed on the frame. If this number is between 0 and 1 represents the fraction of different pixels of the total of the frame compared to the background frame, then the *master* updates the count of frames with movement detected and does not forward the task, otherwise if this number is 2 the *worker* that accepts the task performs grayscale conversion, otherwise if it is 3 the *worker* performs smoothing, otherwise if it is 4 the *worker* performs background subtraction, if instead it is greater than 4 it represents the total number of frames sent by the *emitter* at the end of frames reading.
The *master-worker* pattern is implemented with a `ff_Farm` of *workers* , of which *master* is the *emitter* , the *collector* is removed and the reverse channel between *workers* and *master* is achieved with the function `wrap_around()` of *FastFlow* .

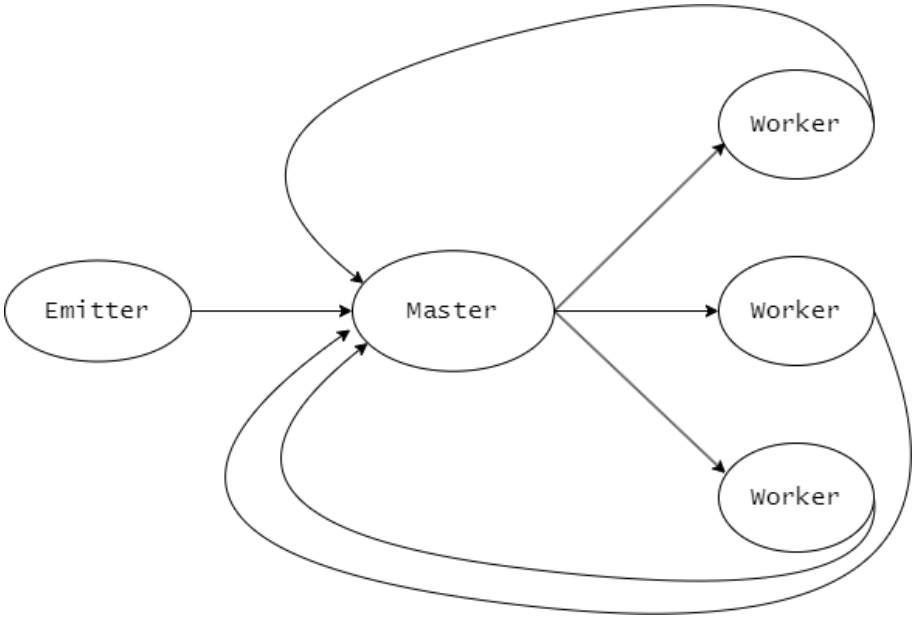


Figure 2: Schematic structure of *FastFlow* implementation with a *master-worker* pattern.

3.3.2 Farm implementation

This implementation is completely different and simpler than the others, here there is a *farm* with an *emitter* that reads and sends frames, some *workers* that apply a composition of functions on the frames and a *collector* that store the results.
The functions applied on the frame by the *workers* are in order grayscale conversion, smoothing

and background subtraction, and the final results given to the collector represents the fraction of pixels that differ between the frame and the background. The *collector* compares this value with the one specified by the user to determine if there is a movement in the frame or not, then updates the total count if necessary.
Even in this implementation the scheduling policy is *on-demand*, because each frame requires a different amount of time to be analyzed.

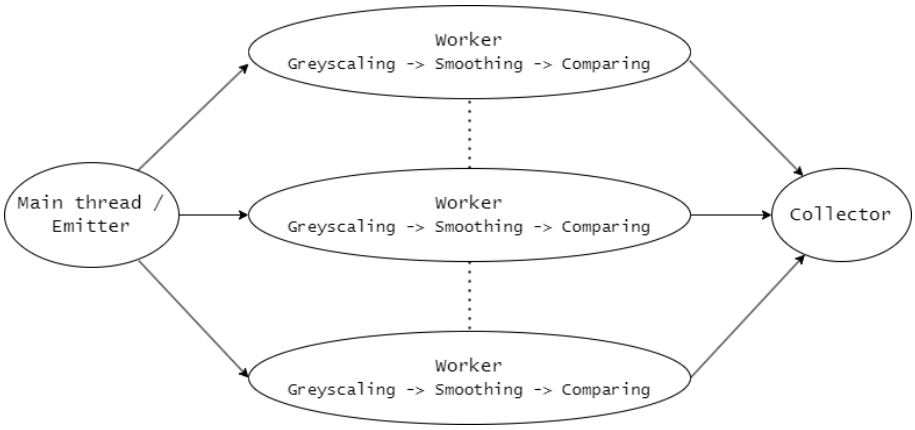


Figure 3: Schematic structure of *FastFlow* implementation as a `Farm(Comp(Seq))`.

4 Sources of Overhead

Moving from the sequential implementation to the parallel applications it is possible to notice that, as the number of threads increases, the performance improvements are not as expected, as it is visible in figure 4, observing the ideal time.
The first sources of overhead analyzed, in the implementation with native `C++` threads, are putting tasks in the queue and updating global variables, whose access is shared between the threads and therefore regulated by mutexes and condition variables, which they introduce waits.
There is a similar problem in the *FastFlow* implementations, the default behavior of which is not to map threads on physical core, because, as it is shown by the results in section 5, in this specific case it does not bring improvements, because probably the working set of the threads, which work with large data, does not fit entirely in the L1 cache of the physical cores. So it is possible to have multiple threads on the same core and it is preferable that a *worker* waiting for a task does not steal resources to another *worker* residing on the same core, so blocking mode is enabled and this introduces anyway an overhead.
Another potential source of delays in the native threads implementation is that there cannot be more than 10 tasks in the queue, and in this case the emitter has to wait a small amount of time before inserting a new task, but this is necessary to avoid the risk to overload the virtual memory. There is the same problem even in the *FastFlow* implementations, that are compiled with a flag that limits the channel buffers to ten elements, for the same reason as before.
The last source of overhead analyzed here is not really a source of overhead but it is more of a limit, and it is the *emitter*, in fact in all implementations it reads the frames from the video in sequence and to capture a frame it requires a time which is not negligible. For example, for the video used in most of the tests, this time is twice the time needed for the grayscale conversion.

5 Results

The results presented in this section are obtained using a remote virtual machine of University of Pisa equipped with an *Intel(R) Xeon(R) Gold 5120* CPU at 2.20GHz with 32 cores.
To measure the performance of the application, it was run on the on the remote machine at different times, to decrease the risk of the results being affected by some external factors, and

the average results were taken with an apposite *bash* script. All these executions have the same parameters, a video with 1138 frames of size 864x1152, where the percentage of different pixels to detect a movement specified is 10% of the total (tested seeing frames while running by launching the application with a specific flag) with the threshold used in the application (one tenth of the average of background matrix pixels value). This video is called `people1.mp4` and it can be found in the folder `Videos` of the project and represents a good target to do background subtraction because it starts with a static background upon which people then transit. The frame size is not so big but the results obtained with other videos, with larger frames, reflects the results obtained with this.
In all the charts shown in the following sections the number of threads does not take into account some additional threads needed to implement the patterns. In particular, there are two for the *FastFlow* implementation as *master-worker* pattern, the *emitter* of the frames and the *master* , and two in the implementation as *farm* pattern, the *emitter* and the *collector* of the *farm* . Then the number specified by the user only specifies the number of *workers* .
Instead, for the native threads implementation there is only one more thread, that is the main thread that generates the thread pool and then acts as the emitter of the frames.
The performance are measured both pinning threads on physical cores and without this feature, to analyze different behaviors. Moreover the times are shown only with a number of workers smaller or equal than 32 because there are 32 cores on the machine.

5.1 Results obtained without thread pinning

This represents the default behavior of the programs because, according to the measurements, it allows to obtain the best performance for most of the implementations.
In the figure 4 the execution times of the different implementations are compared with the ideal execution time for number of threads. It seems that the times obtained with two or three

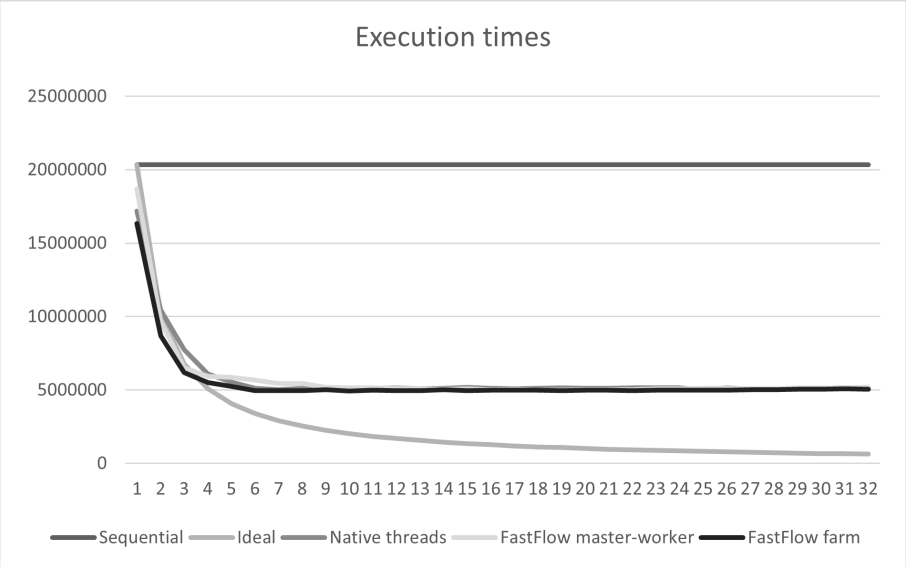
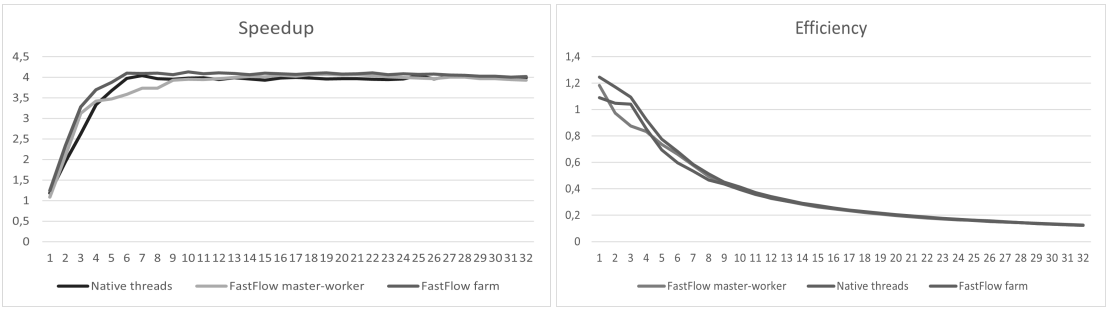


Figure 4: Average execution times obtained for the three parallel implementations compared to the ideal execution time and the sequential time, the y axis represents the time in microseconds, while the x axis represents the number of threads used.

threads go even below the ideal time for the *FastFlow* implementations but, as explained above, this is because the number on the x-axis represents the *workers* and does not take into account the additional threads needed to implement the patterns (two for *FastFlow* implementations, one for native threads implementation).
However, the average execution times for all applications follow more or less the trend of the ideal time up to about 5 threads used, then they begin to decrease very slowly and finally stop decreasing at about 10 threads used, because the improvement is also limited by the time the

emitter needs to capture the frames and by the initial operations.
In the figure 5 two derived measures are shown for number of thread, they are *speedup* and *efficiency* , and they reflect the execution times, which stop improving above a certain number of *workers* .



(a) Speedup obtained without thread pinning.

(b) Efficiency obtained without thread pinning.

Figure 5: Derived performance measures, the y axis represents the value of the measure, while the x axis represents the number of threads used.

5.2 Results obtained with thread pinning

Having threads linearly mapped on physical cores is not a great advantage in this specific case, because the performance obtained are slightly worse for the *FastFlow* implementations than in the previous case, where this feature is not present, while it is slightly better in the implementation with native threads. The execution times are shown in the figure 6. The main

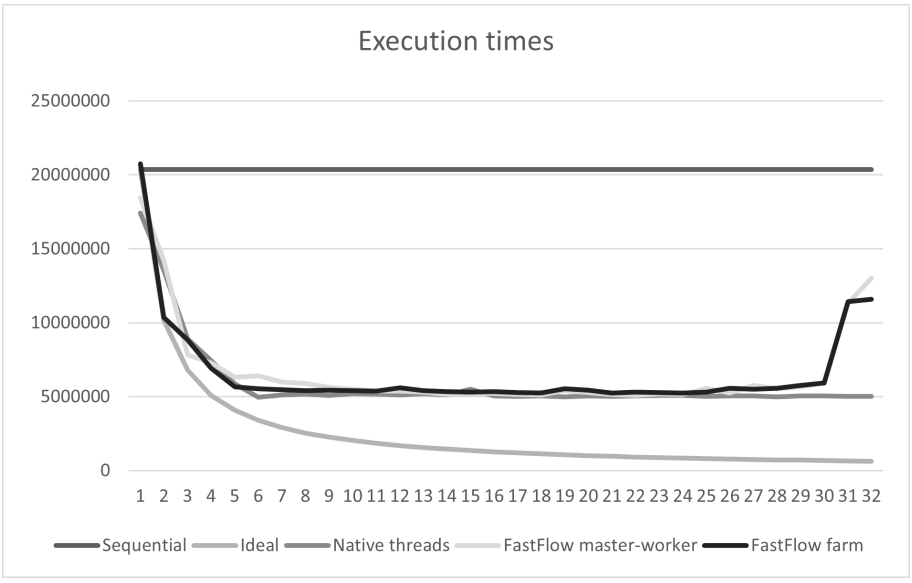
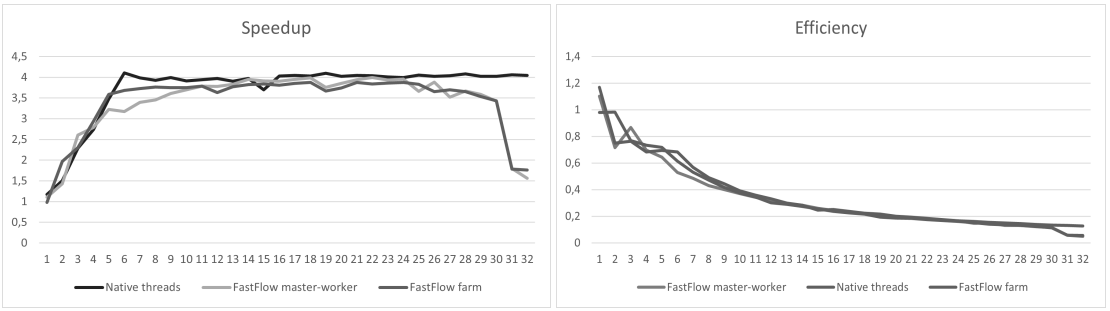


Figure 6: Average execution times obtained for the three implementations compared to the ideal execution time and the sequential time, the y axis represents the time in microseconds, while the x axis represents the number of threads used.

difference is that, due to the two additional threads not taken into account, when the number of *workers* is greater than 30 in the *FastFlow* implementations, the execution time grows rapidly and speedup decreases, because, by counting the additional threads necessary, the number of physical cores is exceeded. This does not happen for the implementation with native threads because it uses blocking mode, so when there are multiple threads on the same physical core,

when a thread is waiting it releases the resources, while the *FastFlow* implementations uses non-blocking mode (only when thread mapping is used) which keeps the physical cores busy. This difference in the behavior can be clearly observed if the `htop` command is used: when thread mapping is used and *FastFlow* implementations are running, the physical cores used are 100% busy, with native threads implementation do not. In the figure 7 there are the corresponding derived measures.



(a) Speedup obtained with thread pinning.

(b) Efficiency obtained with thread pinning.

Figure 7: Derived performance measures, the y axis represents the value of the measure, while the x axis represents the number of threads used.

5.3 Differences in times without vectorization

To test the benefits given by vectorization on the three main parts of the application, the sequential implementation was compiled also without the flags that enable it.
The figure 8 shows the average times obtained for each operation performed on the frames with (figure 8b) and without (figure 8a) vectorization. This is just an example of results obtained on the video used for the tests, but they are very similar in other tests as well.

```
Number of frames with movement detected: 839
Average time spent for greyscale conversion: 10274
Average time spent for smoothing: 61712
Average time spent for background subtraction: 8950
Total time passed: 97175289
```

(a) Times obtained for the sequential application without using vectorization.

```
Number of frames with movement detected: 839
Average time spent for greyscale conversion: 1837
Average time spent for smoothing: 10971
Average time spent for background subtraction: 747
Total time passed: 20146522
```

(b) Times obtained for the sequential application using vectorization.

Figure 8: Effects of the vectorization on the various parts of the program.

6 Execution

The correct way of using this application is to start from `Makefile`, where there are commands to compile the source code. The commands to be used for compilation are the following (when a recompilation is needed, if there is a change in an imported file, the old file must be deleted first):

- `make ffmw`: to compile the *FastFlow* implementation that used *master-worker* pattern in the `ffmw` file.
- `make ffmwtrace`: to compile the *FastFlow* implementation with a flag to see more accurate statistics over the *FastFlow* nodes.
- `make fffarm`: to compile the *FastFlow* implementation that used a *farm* pattern in the `fffarm` file.
- `make fffarmtrace`: to compile the *FastFlow* implementation with a flag to see more accurate statistics over the *FastFlow* nodes.

- `make nt`: to compile the native `C++` threads implementation in the `nt` file.
- `make seq`: to compile the sequential implementation in the `seq` file.
- `make seqnovect`: to compile the sequential implementation in the `seqnovect` file without using vectorization.
- `make res`: to compile the application to automatize the tests in the `res` file.
- `make clean`: delete all the executables.

When we have the executables we can use the following command:

```
./seq video_file k
```

for the sequential application, and

```
./nt video_file k -nw nw
./ffmw video_file k -nw nw
./fffarm video_file k -nw nw
```

respectively for implementations with native threads and with *FastFlow*, where `k` is the percentage of pixels on the total that must be different from background to decide if there is a motion in a frame and `nw` is the number of workers to use.
For each of the implementations there are also some flags that can be set for extra features:

- `-show`: shows the resulting frames for each of the three phases, it needs to use the GUI of the device used and it increases the completion times a lot.
- `-info`: shows the times for each of the three phases and, in case of *FastFlow* , shows also more statistics on the *FastFlow* nodes at the end, if the program has been compiled with `make ffmwtrace` or `make fffarmtrace`.
- `-help` prints the correct usage of the program.
- `-mapping`: maps threads on physical cores linearly.

The executable `res` was used only in development on the remote machine to produce results without manually launching the program each time, but sending the `./res` command without parameters, it shows the usage info, as all the other executables do. The default behavior is not to run the sequential application, but can be executed specifying `-seq` in the command.
The results obtained from the program executions are written to the file in format

```
Mon Aug 29 15:33:20 2022 - Videos/people1.mp4,ffmw,10,18,0,4902922,839
```

where we have the date followed in order by the name of the video, the type of implementation used, the parameter `k` specified by the user, the number of workers used, a flag indicating if the threads are mapped on physical cores (value 1) or not (value 0), the completion time in microseconds and the number of frames with motion found. The default output file for the results is a text file with the name of the video it refers to.
Finally, if there are many results in the `results` folder and the user wants to aggregate them, by sending the command `./get_avg_times.sh type nw k` they are shown the average execution times obtained with the implementation type specified by the `type` parameter and with a number of threads between -1 (case sequential execution) and the number of threads specified by the parameter `nw`, for each file in the `results` folder (assuming that each file contains the results related to a video) filtered with the `k` parameter specified.