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

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

2 Project Structure

There are three implementations of the application: one is the sequential application, one is the parallel application which uses native `C++` threads, and the latter is the parallel application which uses the *FastFlow* programming library. These implementations are accurately described

in the section 3.
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.
- `ff.cpp`: main source code of the *FastFlow* implementation.
- `Makefile`: the instructions to compile the applications.
- `nthreads` folder: contains files with code that implements classes used for the implementation with native `C++` threads, that is:
  - `greyscale_converter.hpp`: implements the functions to convert a RGB channels frame in black and white and to obtain the pixels average intensity of a black and white 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 all functions to synchronize all worker threads.
- `fastflow` folder: contains the files with the code to implement the classes used for the implementation with *FastFlow* , that is:
  - `ff_greyscale_converter.hpp`: implements the functions to convert a RGB channels frame in black and white using a `ParallelFor` and to obtain the pixels average intensity of a black and white frame using a `ParallelForReduce` , these functions are used only for the background, while those implemted in the next file are used in the *pipe*.
  - `ff_greyscale_converter_emitter.hpp`: reads video frames, implements the conversion from RGB channels to greyscale using a `ff_Map` node and sends the black and white frame to the smoothing *farm*.
  - `ff_comparer_collector.hpp`: gets matrices with a smoothing filter applied and compare them with the background, it is a multi input node that uses a `ParallelForReduce` to count the number of different pixels over the total of the matrix.
  - `ff_smoother_worker.hpp`: implements a worker `ff_node` of the smoothing *farm*, it receives black and white frames and applies a smoothing filter matrix to them, then sends them to the collector.
- `utils` folder: contains the files with the classes useful for all the implementations:
  - `smoother.hpp`: implements the functions to apply a smoothing filter to a frame, used by the sequential workers of the *farm* of the *FastFlow* implementation and by the workers of the thread pool of the native `C++` threads implementation.
  - `file_writer.hpp`: implements a way 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 two *bash* scripts to parse these files:
  - `compute_avg.sh`: computes the averages of the execution times within the files of the `results` folder.
  - `get_times.sh`: gets, for each video, the average execution time per number of thread, filtered for specific implementation and `k` parameter.
- `results.cpp`: it is a script to automatize the execution of the application with different parameters.

3 Implementative choices

In this section the main implementative choices are described for each of the three implementations.

3.1 Sequential implementation

The sequential implementation takes as parameter the fraction of pixels that must be different to detect a motion and two flags that indicate if the application whether or not will show the resulting matrix and the completion times for every operation, 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 one third of this intensity. The filter matrix used for smoothing is a square matrix of order 3 which computes the average of the 3x3 neighborhood of the pixel and the smoothing is implemented as a moving average on the pixels of the matrix, starting from the top-left corner to the bottom-right corner.
After these initial operations, the program starts to read the frames of the input video and performing for each of them, as for the background, the conversion to grayscale and smoothing. When the matrix, which represents the pixels of the frame, is in grayscale and has the smoothing filter applied, it is subtracted from the background matrix and the pixels that differs more than the threshold are counted and their fraction of the total is computed to be compared with the percentage value specified by the user to decide if there is motion in the frame or not.

3.2 Native C++ threads implementation

This implementation aims to parallelize, where possible, the operations of the previous implementation to achieve better performances using native `C++` threads. The first two parts that can be parallelized are the grayscale conversion and the background subtraction, since each operation involves a pixel and is independent of the value of the other pixels, so it is possible to use a *Data Parallel Stream* pattern. For this reason these operations are implemented with a *map* on the rows of the matrix, each thread takes $\frac{n}{nw}$ rows, where n is the number of total rows and nw is the number of workers used in the *map* , and performs the operation over the pixels of these rows.
On the other hand, for the smoothing operation it is not possible for the threads to operate on different rows of the matrix because it is implemented as a moving average, so when a pixel is processed, the previous pixels in its 3x3 neighborhood need to be already processed. Then, a pool of threads is used for the smoothing, each worker thread takes a black and white frame from a queue, performs the smoothing operation across the entire matrix and inserts the matrix with the filter applied into a queue to be taken and compared with the background.
The smoothing operation consists of a multiplication between two square matrices of order 3 for every pixel (except the pixels of the first and last row and column), then a computation of the value of the central pixel, so it is obviously the one with the highest service time of the stream, while the grayscale conversion and the background subtraction are very fast even with the sequential implementation (with the frame size of the videos considered). For this reason $\frac{2}{3}$ of the workers specified by the user are used for smoothing by the thread pool, while only $\frac{1}{3}$ for the data parallel computation of the first and last stage of the stream.
In the first implementation, having the phase with the highest service time in the middle, initially the background subtraction workers had no tasks to calculate, while, for the last period, it was the grayscale converters who were left without tasks. To optimize this, it was decided to have a single thread that takes both the conversion tasks and the background subtraction tasks from the same queue and executes them one by one with the help of the *map* workers.
In summary, the stream starts with the main thread, which prepares the background matrix and starts an *emitter* that reads the video frames, prepares the grayscale conversion task and puts it in a queue, then the thread described above takes it, converts it to black and white, prepares the smoothing tasks and puts it in another queue. Now a smoothing worker takes the greyscale frame and apply a smoothing filter on it, before putting the last task (background subtraction)

in the first queue from which it must be taken by the same thread as before which this time will perform background subtraction. When the frames are finished and all the threads have exited, the main thread gets the final value of the frames with motion detected. A scheme of the flow of execution of this implementation is shown in figure 1.

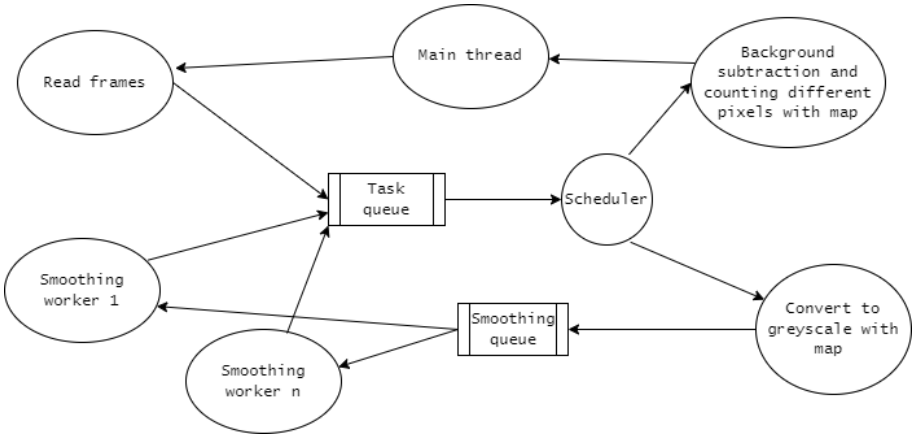


Figure 1: Schematic structure of the native `C++` threads implementation, the execution starts in the main thread, then follows the flow described in section 3.2 and finally the final result returns to the main thread.

3.3 FastFlow implementation

In the *FastFlow* implementation there is a directed graph that starts with the reading of the video frame and ends with a motion detection in the frame or not. Stream data are matrices representing the frame of the video. The first node is implemented through a `ff_Map` node, which for each pixels computes the average between the channels of the pixel, and map these operation on the rows of the matrix with a *parallel_for* pattern.
The resulting frame is sent to a farm (in particular to the default *emitter* of a farm, that sends the frame to one of the workers) implemented as a `ff_Farm` node, where the worker nodes perform the smoothing operation, that, as we said in section 3.2, is implemented as a moving average, so it is not possible to use a *Data Parallel Stream* pattern here.
The default *collector* of the farm is removed and a `ff_minode` (multi-input node) is used at its place. This node takes frames with the smoothing filter applied and compares them to the background, using a `ParallelForReduce` object to reduce the number of different pixels in a variable, then it updates the total number of frames with a detected motion.
All these nodes (the *source*, the *farm* and the *collector*) are inserted in a `ff_Pipe` which is started and, once terminated, the final result is available for the main thread. The execution flow is summarized in the scheme in figure 2.
The smoothing phase is the one with the highest service time, therefore $\frac{1}{2}$ of the workers specified by the user are used for it, $\frac{1}{4}$ for greyscale conversion and $\frac{1}{4}$ for background subtraction.
Also, when the number of threads exceeds the number of real cores, *FastFlow* has been configured to set the blocking mode, because we can have the *collector* , actively waiting for a frame to compare with the background, sharing a core with a smoothing worker, increasing its service time. Compiling with a flag that disables *thread* mapping on real cores instead, the results obtained are slightly worse also for a large number of threads, probably because these optimizations outweigh these disadvantages.

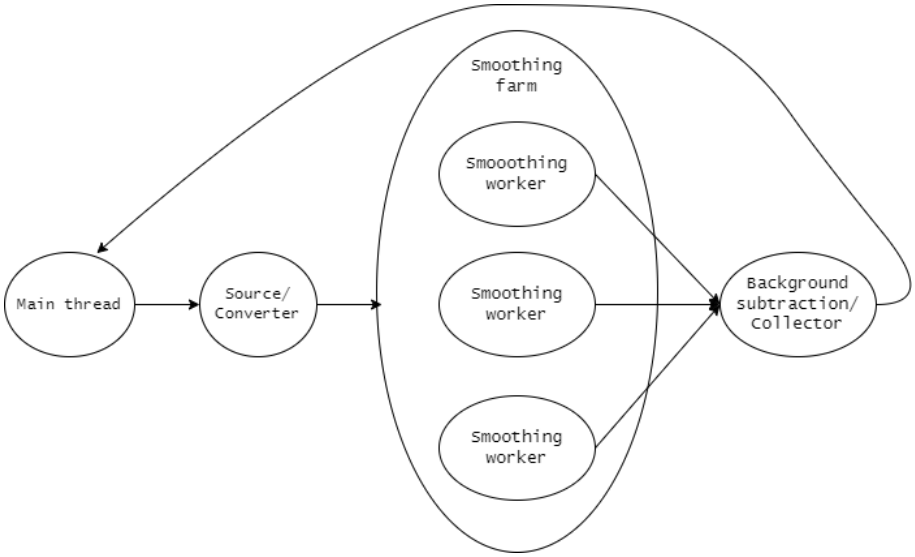


Figure 2: Schematic structure of *FastFlow* implementation as a `Pipe(Map,Farm(Seq),Map)`, the execution starts in the main thread, then follows the flow described in section 3.3 and finally the final result returns to the main thread.

3.4 Different behaviors between parallel implementations

Due to the different design of the application, there are also some small differences in the final behavior. The most important one is that, having in the native threads implementation the same queue for grayscale conversion tasks and background subtraction tasks and having a single thread taking those tasks from that queue, we have that initially there will be many of the first tasks in the queue , because the source node is very fast, and very few of the last tasks, because the intermediate phase that produces them is the one with the highest service time, so the application takes longer to produce the first results, while it is very fast at the end, when there are only the last tasks in the queue. This causes the latency to increase at the beginning and decrease towards the end.
Instead in the *FastFlow* implementation we have a *pipe* that produces a result as soon as a matrix with a smoothing filter applied is available, so the latency remains more or less the same during execution and this is probably better with a long video or with a real time application.

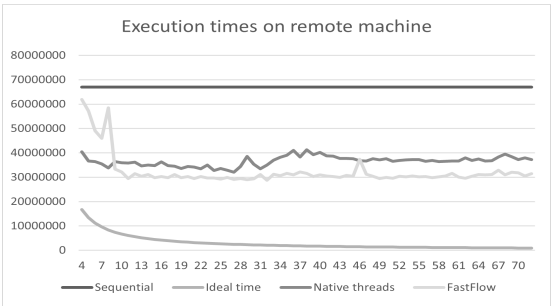
4 Sources of Overhead

Moving from the sequential implementation to the parallel applications, we can see that the performance improvements are not those expected, as it is visible in figure 3, for example for the operations where it was decided to use a *Data Parallel Stream* pattern. In particular the greyscale conversion is faster in the sequential implementation (compiled with `-O3` flag) than in the implementation with native `C++` threads when dealing with low order matrices, for which probably the overhead due to starting and joining threads is greater than the time the whole sequential operation takes. In the implementation with an `ff_Map` node the time is more or less the same as in the sequential implementation, and it is very much faster than the implementation with a simple `ParallelFor`, probably due to the internal optimizations of the `ff_Map` node.
Another source of overhead in the implementation with native `C++` threads is putting tasks in queues and updating some global variable such as the one for different pixel counter in background subtraction, where all the *map* workers have to update it. The queues and the global variables are resources shared by some threads, so it was necessary to use *atomic* variables, which are thread safe, or *mutexes* and *condition variables* to synchronize reads and updates by the threads, so we introduced active or passive waits which increased the completion time.

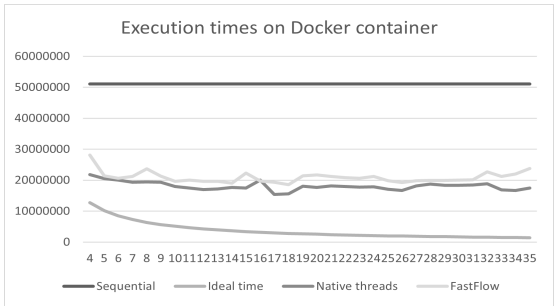
Instead in the *FastFlow* implementation we have a *pipe* where the *emitter* and the *collector* are very much faster than the farm workers, so there is a final period in which the *source* has finished the frames but the workers are busy, and some periods during the execution in which the *collector* waits for a frame from the farm. In fact, leaving the default *collector* of the farm and implementing the last node as `ff_Map` node instead of as a `ff_node` with `ParallelForReduce`, we had that the background subtraction phase was very fast, but the completion time was not less, but it increased for the introduction of a *collector* in the communication between the farm workers and the last node.

5 Results

The results presented in this section are obtained using a *Docker* container with 8 cores detected with *Linux* inside that runs on top of a *Windows 11* operative system on a personal computer equipped with an *AMD Ryzen 7 4700U* CPU and on a remote virtual machine of University of Pisa equipped with an *Intel(R) Xeon(R) Gold 5120* CPU at 2.20GHz with 32 cores detected.
To measure the performance of the application, it was run on the *Docker* container and 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. All these executions have the same parameters, a video with 1283 frames of size 300x402, where the percentage of different pixels to detect a movement specified is 2% of the total (tested seeing frames while running by launching the application with a specific flag) with the threshold used in the application (one third 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 video.
The first measure analyzed is the execution time of the implementations for different number of threads, which starts from 4 because the *FastFlow* implementation needs at least one thread per stage of the pipe. From the plots in figure 3 it can be seen that the *FastFlow* implementation does not work well with a small number of threads, especially on the remote machine, but as the number of threads increases it generally performs better than the native `C++` threads. On the container the results are very similar, but the native threads implementation always keeps a slightly better time, probably due to the low number of real cores detected and for the use of more threads for the smoothing phase, in fact when *FastFlow* sets the blocking mode (when the number of threads exceeds the number of real cores) we get slightly better results, for the reason explained at the end of section 3.3. The next measures that have been examined are *speedup*



(a) Execution times obtained on the remote machine for the three implementations compared to the ideal execution time

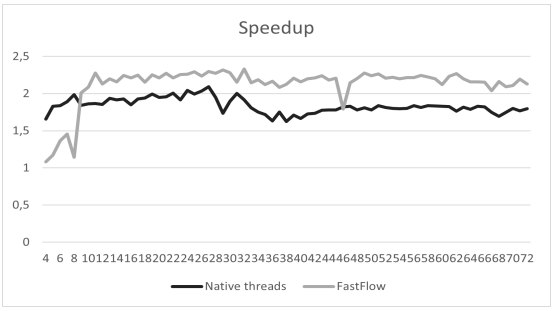


(b) Execution times obtained on the *Docker* container for the three implementations compared to the ideal execution time

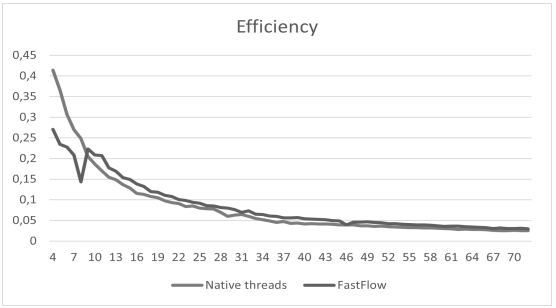
Figure 3: Average execution times obtained during the tests, the y axis represents the time in microseconds, while the x axis represents the number of threads, the maximum of which is different between the machines due to the different number of cores equipped.

(figure 4a for the remote machine and 5a for the local machine) and *efficiency* (figure 4b for the

remote machine and 5b for the local machine) for the parallel implementations. They reflects the results shown in figure 3, with a bad performance for the *FastFlow* implementation with a small number of threads and a better performance when the number of threads increases.

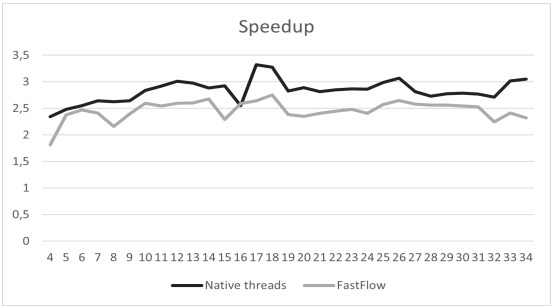


(a) Speedup obtained on the remote machine with the parallel implementations as compared to the sequential implementation.

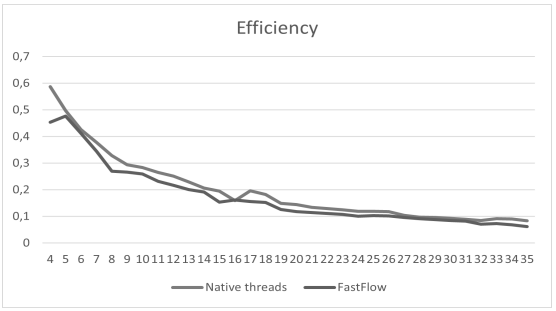


(b) Efficiency obtained on the remote machine with the parallel implementations.

Figure 4: Measures observed on the remote machine, the x axis represents the number of threads, the maximum of which is different between the machines due to the different number of cores equipped, while the y axis represents the value of the measure.



(a) Speedup obtained on the local machine with the parallel implementations as compared to sequential implementation.



(b) Efficiency obtained on the local machine for the parallel implementations.

Figure 5: Measures observed on the local machine, the x axis represents the number of threads, the maximum of which is different between the machines due to the different number of cores equipped, while the y axis represents the value of the measure.

From all these analysis we can deduce that on the *Docker* container we get the best efficiency with the native `C++` threads implementation by specifying a number of threads between 16 and 20, if we are willing to have a higher latency at the beginning, as explained in section 3.4, while on the remote machine we have the best efficiency using the *FastFlow* implementation and specifying a number of threads between 10 and 15.

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, a `make clean` command is needed first):

- `make ff`: to compile the *FastFlow* implementation in the `ff` file.
- `make fftrace`: 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 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
./ff video_file k 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 the `ff_Pipe` statistics at the end.
- `-output_file` followed by the name of a file: specify which file the results will be written to.

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 are written to the file in format

```
Wed Jun 22 20:03:42 2022 - Videos/people1.mp4,nt,2,16,0,34947964,421
```

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 whether the frames are shown in the GUI (value 1) or less (value 0), the completion time in microseconds and the number of frames with motion found. If no output file is specified by the user, the default output file for the results is a text file with the name of the video it refers to.
Finally, when there are many results in the `results` folder, by sending the command `./compute_avg.sh nw` it is possible to see for each video and for each implementation the average completion time obtained using `nw` worker, or for the sequential application if `nw` is -1. The command `./get_times.sh type nw k` instead shows 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) also filtered with the `k` parameter specified.