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.
  - `smoother.hpp`: implements the functions to apply an average filter with a 3x3 kernel on a grayscale 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_emitter`: reads video frames and send them to the farm that converts them in grayscale and apply a smoothing filter on them. It is the emitter of the first farm.
  - `ff_converter_smoother_worker`: implements the functions to convert a RGB channels frame in black and white and to apply a smoothing filter with a 3x3 kernel on it. It is an `ff_Map` node of a farm.
  - `ff_comparer_worker.hpp`: implements the functions to compare a frame with the background and to count the different pixels between the two frames. It is a node of a farm.
  - `ff_collector.hpp`: takes the percentage of different pixels between the frames and the background, compare them with the user-specified value and finally update the total number of frames with movement detected. It is the collector of the second farm.
- `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 Design 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 $\frac{1}{10}$ 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 grayscale and smoothing. The filter used for smoothing computes the average of the 3x3 neighborhood of the pixel and the smoothing is implemented by summing the pixels value divided by nine. 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.
At the end the program prints the average time spent for each of the three phases, to better understand where bottlenecks are possible and where we can improve it using parallelism.

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. Data parallelism is possible for each of the three phases (grayscale conversion, smoothing and background subtraction) because the operation over the pixels are independent from the other pixels of the original matrix, but observing the average times spent for each of them it is possible to notice that it is not very useful for grayscale conversion and background subtraction, that are very fast even in the sequential implementation. Instead it is decided to use it for smoothing, which computes more complex operations and takes much longer on average, so each worker takes chunks of $\frac{n}{nw}$ rows, plus the previous and next row to have access to the 3x3 neighborhood of the pixels of the first and last row. This is possible because the result of the operation is written to another matrix.
Even stream parallelism is possible for each phase, especially if we are not interested in the frame order but only the final number and it was decided to use it too to speed up the smoothing phase, and consequently, even if with less workers, also for the other two phases. In fact, when we have many smoothing workers, having only one grayscale converter can slow start up because, even if it is very fast, it can only supply one frame at a time, and having only one worker for background subtraction may not be sufficient to support the workload.
This is implemented with a thread pool, where there are two type of workers, one that performs smoothing using also data parallelism, and one that performs grayscale conversion or background subtraction depending on the task that they receives. In this way no threads are wasted, because the workers of latter type prepares the frames for the smoothing workers faster, and finally they are all used to do background subtraction. All these threads use two queues to communicate, one for the grayscale conversion and background subtraction tasks and one for the smoothing tasks.
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 a thread of the pool 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 another thread, which could be the one that

does the conversion to grayscale first, which this time will do 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.

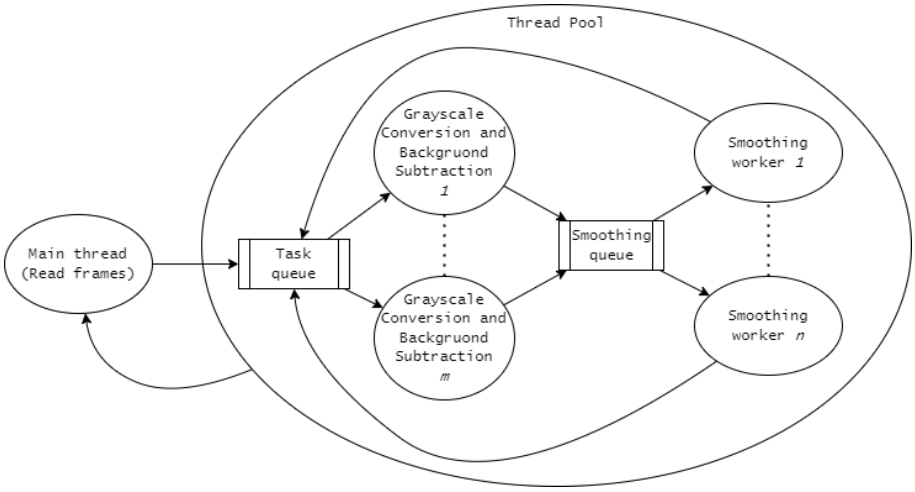

Figure 1: Schematic structure of the native `C++` threads implementation, the execution starts in the main thread, then follows the flow described in section 3.2 and finally the final result returns to the main thread.

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 of the `ff_Pipe` is an *emitter* which simply reads the frames of the video input and send them to a `ff_Farm` node. The worker of the farm are implemented through a `ff_Map` node, which first performs grayscale conversion, computing for each pixels the average between the channels of the pixel, and maps these operation on the rows of the matrix with a *parallel_for* pattern, then apply a smoothing filter always using a *parallel_for* pattern on the rows of the matrix. This farm has no *collector* and sends the result frames directly to the default *emitter* of another farm, in which the workers are `ff_node` that do background subtraction. The workers of this farm, unless otherwise specified, are approximately $\frac{1}{8}$ of the total number of threads, because they are much faster than the worker of the first farm. The default *collector* of this second farm is removed and a `ff_minode` (multi-input node) is used at its place as *collector* . This node simply receives the percentage of different pixels between the frame and the background out of the total and compares them with the user-specified `k` parameter, to decide if there is a motion in the frame.
It was possible to use also a `ff_a2a` node instead of two `ff_Farm` , but the results was not better, probably because `ff_a2a` needs `ff_monode` and `ff_minode` nodes, we had lost the optimization of the `ff_Map` node for the data parallelism.

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 the same workers 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

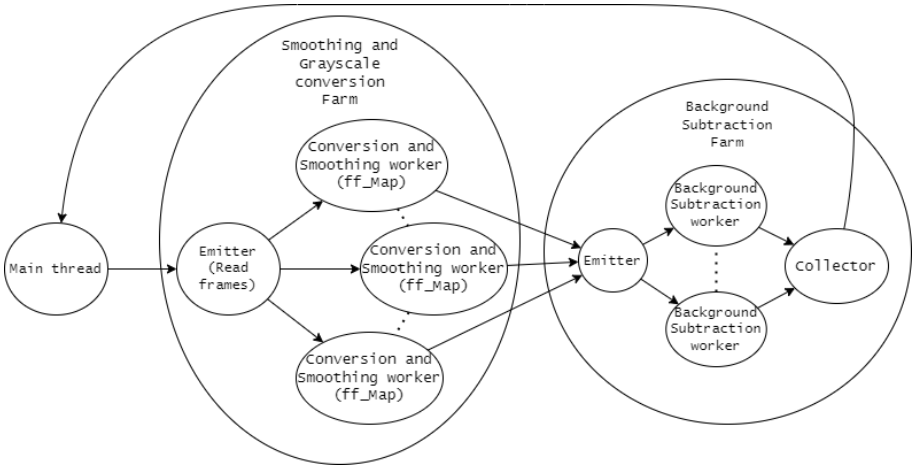

Figure 2: Schematic structure of *FastFlow* implementation as a `Pipe(Seq,Farm(Map),Farm(Seq),Seq)`, the execution starts in the main thread, then follows the flow described in section 3.3 and finally the final result returns to the main thread.

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. There is even a difference between the parallel implementations and the sequential one: the latter maintains the frame order, while the parallel implementations, using stream parallelism without ordering mechanisms to achieve better performances, are interested only in the final count of frames with motion in the video.

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* pattern. In particular for grayscale conversion and background subtraction we obtain no significant improvements with data parallelism 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.
Another source of overhead in the implementation with native `C++` threads is putting tasks in queues and updating some global variable, where all the *map* workers have to update it. The queues and the global variables are resources shared by the 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 waits which increase the completion time.
Instead in the *FastFlow* implementation we have a *pipe* where the *emitter* and the *collector* are very much faster than the internal nodes of the two farms, so there is a 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 value from the second farm. Using more worker in the phase that takes more time partly solves the problem but not completely.
Another source of overhead is memory usage, which also depends on the machine on which the application is running on, because the emitter reads frames very fast and puts them in a queue, so if we use a low number of threads the smoothing phase can be very slow and the raw frames can occupy a lot of memory before being processed and deleted. This, especially on the local machine, causes the system to kill the program due to insufficient memory when the frame size is very large and the video frames count is very high. Even if the program is not killed, execution can slow down a lot when the memory is almost full, then, as the frames are processed the performance returns the same.

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 detected. 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.
The first measure analyzed is the execution time of the implementations for different number of threads, which starts from 3 because the parallel implementations 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, but as the number of threads increases it generally performs better than the native `C++` threads. The next measures that have been

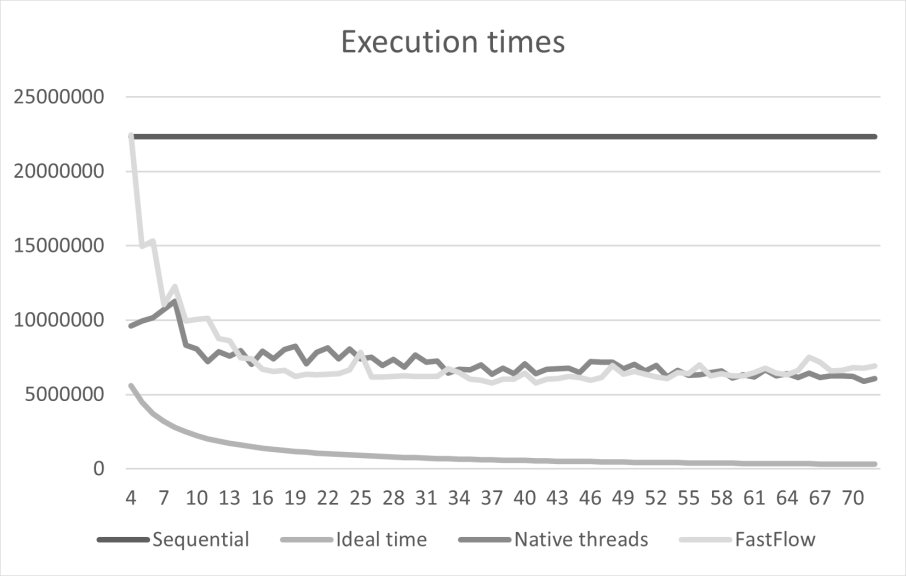

Figure 3: Average execution times obtained on the remote machine for the three implementations compared to the ideal execution time, the y axis represents the time in microseconds, while the x axis represents the number of threads. In this number, for the *FastFlow* implementation, the first node of the pipe that only reads video frames, the default *emitter* and the *collector* of the background subtraction farm are not considered. With bigger number of threads the *FastFlow* reaches its limits and *FastFlow* downsize it.

examined are *speedup* (figure 4) and *efficiency* (figure 5) 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.

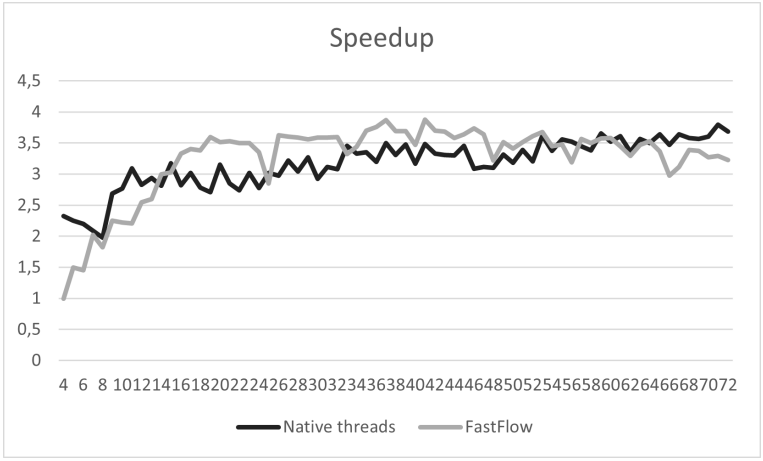

Figure 4: Speedup obtained on the remote machine with the parallel implementations as compared to the sequential implementation, the x axis represents the number of threads, while the y axis represents the speedup value.

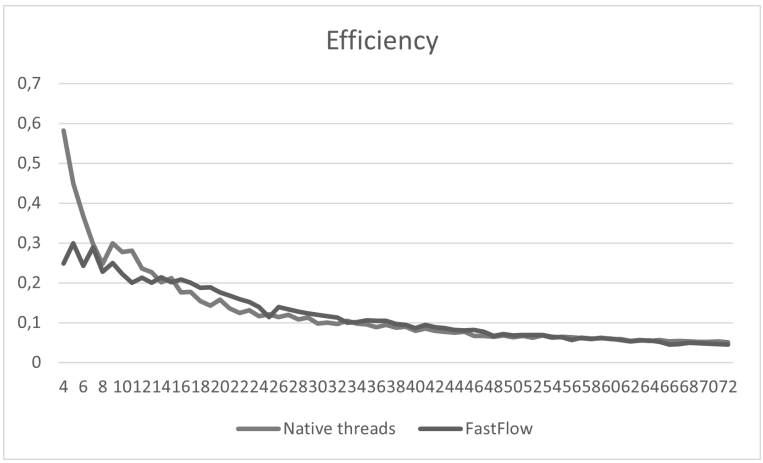

Figure 5: Efficiency obtained on the remote machine with the parallel implementations, the x axis represents the number of threads, while the y axis represents the efficiency value.

In all the previous figures the number of threads is the one specified by the user, which is automatically divided by the program in the best possible way according to the tests carried out. For both the implementations most of the threads are used in the smoothing phase, which is the more expensive in terms of computational time (in the *FastFlow* implementation these workers perform also grayscale conversion) and in both cases it has two level of parallelism, but, while for the native threads implementation it is decided to use a small fixed number of threads for data parallelism and the rest for stream parallelism, in the *FastFlow* implementation it is decided to use a many threads for the data parallelism and few for the stream parallelism.
It can also be seen from the images that the best results (low completion times) are obtained using the *FastFlow* implementation with a thread count between 35 and 45. Using the native `C++` threads implementation with approximately a thread count between 62 and 72 we achieved similar results, but using more resources as well, in fact this is less efficient. The ideal time is never reached due to the reasons explained in section 4.

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 nw
./ff 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. It is possible to also use

```
./nt video_file k -specific_stage_nw data_smoothing_workers
   stream_smoothing_workers stream_converter_comparer_workers
./ff video_file k -specific_stage_nw data_smoothing_converter_workers
   stream_smoothing_converter_workers stream_comparer_workers
```

if we want to specify a specific number of threads for each phase or level of parallelism, explicitly controlling the levels of parallelism.
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, if the program has been compiled with `make fftrace`.
- `-help` prints the correct usage of the program.

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

```
Tue Jul 12 17:46:20 2022 - Videos/people1.mp4,ff,10,18,0,6828734,839
```

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 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.