UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Parallel and Distributed Systems: Paradigms and Models

Project report: video motion detect

Studente:

Alex Pasquali

Contents

1 Introduction

The task at hand consists in implementing a simple motion detection system for videos. There is a stream of frames to be evaluated, and each one is compared against the background (i.e. the first frame): if they differ, motion has to be detected. The final result consists in the number of frames that were different from the background. From this, it is already possible to deduce that the data is not available in its entirety from the beginning, but there is instead a stream of pieces of that data (i.e. the frames) arriving to the motion detection system. For this reason, **stream parallel patterns**, such as pipelines or farms, look suitable. More in detail, for each frame, three operations have to be performed in the following order: RGB to grayscale conversion, smoothing and motion detection.

2 Sequential version

The sequential implementation simply process one frame at a time, applying to it all the three operations. The main code is in `src/sequential/main_sequential.cpp` and the operations are implemented in `src/sequential/sequential_funcs.cpp`.

2.1 Performance of sequential implementation

Let's consider the latency of each sequential operation (RGB to gray conversion, smoothing[1] and motion detection). This has been measured using an apposite script (`src/sequential/seq_funcs_perf_eval.cpp`), which runs the whole video through every stage, and for each of them, takes the latency averaged over every frame in the video. This process is repeated for a certain number of attempts and the results are further averaged, obtaining three mean latency values, one for each operation. Specifically, these are reported in Table 1.
The overall average latency of a complete processing of a frame is instead 63.6 ms. The average completion time is 15171.5 ms.

Stage	**Latency**
RGB to gray conversion	20869.6 μs / frame
Smoothing (cleaner code)[1]	89340 μs / frame
Smoothing (efficient code)[1]	39689.9 μs / frame
Motion detection	2820.6 μs / frame

Table 1: Latency of the sequential operations.

[1]Two implementations of the smoothing operation are provided: one has a cleaner code, generalized to work for pixels located both on the border and on the interior of the image; the second differenciates the two cases, and for internal pixels, which are the great majority, performs a simpler operation. Only the latter is used in the various implementations of the application (sequential, threads, FastFlow [1]).

2.2 Compile and execute

In the base directory of the project, run:
To compile: `make sequential` or `make all`
To execute: `./bin/main_sequential.out <path to video>`

3 Parallel version

As stated in Section 1, two main categories of patterns are considered: a pipeline and a farm, both with some variations.

3.1 Pipeline as main pattern

A suitable way of parallelizing the application could be using a pipeline with three stages, one for each operation. Moreover, each stage could be further parallelized:

- the first one shows no dependencies among different pixels, so it could be realized through a map operation;
- smoothing, instead, needs the surrounding neighborhood of each pixel, and could be realized with a stencil;
- finally, motion detection actually needs to could the number of differing pixels in the frame, producing a single number as output. For this reason, it is suitable to be implemented through a reduce operation.

3.1.1 Theoretical performance of pipeline as main pattern

The tool `rplsh` [2] has been used to calculate an estimate of the performance of the various parallel patterns that were considered, given the measured latencies of the sequential stages (Tab. 1).
A plain pipeline would have a service time equal to the maximum of those of its stages, which in this case is 39689.9 μs, coming from the smoothing operation (Tab. 1).
The ideal situation, though, would be to have a balanced pipeline, which means having every stage with a similar latency. A simple way to equalize these values would be to implement every stage through a farm, and adjust the number of workers accordingly. Considering a maximum number of resources of 32, the best balancing is obtained with the assignment of workers reported in Table 2, which gives an overall service time of 2608.7 μs. It has to be noted that the sum of workers in the table is not 32, but 26. This because every farm has also its own emitter and collector, and since the farms are 3 - one for every stage of the outer pipeline - there are 6 "wasted" resources.
Finally, implementing the three stages with a map, a stencil and a reduce respectively (as hypothesized previously) would not beat the performance obtained in this simpler way.

Stage	Number of workers	Corresponding latency
RGB to gray conversion	8	2608.7 μs
Smoothing	16	2480.62 μs
Motion detection	2	1410.3 μs

Table 2: Latency of the sequential stages if implemented with a farm using the specified number of workers.

3.2 Farm as main pattern

An alternative approach to the one of Sec. 3.1 could be to use a farm as main/outer pattern. The three stages could then be associated into a comp, realizing the so-called normal form.
Alternatively, it is also possible to create a farm of pipelines, where each stage could be further realized through a farm, in order to equalize their latencies, ideally reducing the overall service time, as explained in Sec. 3.1.1. This would have a cost in terms of resources, though, because every farm would need its emitter and collector.

3.2.1 Theoretical performance of farm as main pattern

Considering again a maximum number of resources equal to 32, a farm of 10 of these pipelines needs exactly 32 resource, and it would give a service time of 3968.9 μs. It is much higher than the one in Section 3.1.1, and this is due to the fact that the multiple instances of the pipeline are unbalanced. Trying to balance them, though, does not improve the performance due to the extra wasted resources.
The normal form, instead, wastes very few resources and provides a service time of 2112.67 μs, which is the lowest I was able to find on `rplsh` [2].

3.3 Native C++ threads implementation

This implementation realizes a normal form pattern, i.e. a farm of comps. The comp is simply realized with a function executing the sequential operations one after the other, while the farm is realized through a shared queue where the main thread pushes the frames. A certain number of threads take the role of the various workers of the farm: repeatedly, they pick a frame from the queue and execute the comp. The farm has no collector, but the workers update a shared variable keeping the count of the frames that differ from the background. Its access is synchronized because this variable is atomic.

3.3.1 Implementation of the shared queue

The shared queue is an important piece of the parallel implementation with pthreads. It is implemented in the file `include/parallel/shared_queue.hpp`. To make it more general, it is a template class, so it can work with any type of data. Among its attributes there is a standard C++ queue (`std::queue`), but the access to it is protected by default. In fact, its `push(T*)` and `pop()` methods work as follows:

- `push(T*)`: create a `std::unique_lock`, push the data into the actual queue, and, through a condition variable, notify one of the threads that are waiting to act on the shared queue.
- `pop()`: create a `std::unique_lock`. Then, using a condition variable, wait in case the queue is empty and the video is not finished. Next, if the queue is not empty, pop from the queue and return the data, otherwise return a `nullptr`.

This ensures that the accesses to the queue are synchronized. Furthermore, the use of condition variables ensure a passive wait in order not to waste resources.
When the main thread, which takes the role of the emitter, has pushed the last frame, it calls the method `no_more_pushes()`, which sets the attribute `finished` to `true`, and, through the condition variable, notifies all waiting threads, so they can terminate.

3.3.2 Shared counter

The number of frames differing from the background is kept in a variable that is shared among all the workers, this means that its accesses need be synchronized. This shared counter is implemented with an atomic variable, and this might introduce some overhead when multiple threads try to update its value. For this reason, I tried to give a "personal" counter to each thread: in the parent thread, a vector as long as the number of workers is created and then a single cell is assigned to each one of them. This way, threads do not need to wait in order to access the counter. After all the threads have terminated, the partial counts in the vector are summed together to get the final result. This method did not improve the performance. One reason of this might be false sharing, so I tried to create an oversized vector and assign a chunk of it to each worker. Not knowing the exact dimension of caches, different sizes of the vector have been tested, but neither of them improved substantially the performance.
The final implementation relies on the shared atomic variable due to its semplicity and the less memory space that it requires.

3.3.3 Compile and execute

In the base directory of the project, run:
To compile: `make threads` or `make all`
To execute: `./bin/main_threads.out <path to video> <number of workers>`

3.4 FastFlow implementation

In this case, the pattern is again a normal form, in order to compare the performance and the ease of implementation with respect to the use of native C++ threads (sec.

3.3). Thanks to FastFlow [1], it is not required to manage the shared communication channels among workers (e.g. queues), which is the most delicate part of the implementation of a parallel application such as the one considered in this report. In this implementation, every worker is a struct inheriting from `ff:ff_node_t`. In particular, I implemented the following structs:

- `FrameWithMotionFlag`: this is not a worker, but it represents a special type of data that was used in this implementation. It has two attributes: a pointer to a frame and a boolean flag indicating if motion was detected in the frame.
- `Emitter`: as the name suggests, it is the emitter of the farm. At every time step, it reads a frame from the video and returns it. This is sufficient because the library takes care of the communication between the emitter and the workers.
- `Comp`: this represents a farm's worker executing the composition of the three operations. It receives a `FrameWithMotionFlag`, executes the operations and if motion is detected, it puts the flag of the input data to `true`.
- `Collector`: it has an attribute to count the number of frames marked as different from the background. It keeps receiving `FrameWithMotionFlag`'s from the workers and, if their flag is set to `true`, it increments its counter.

Even though the pattern is the same (normal form), a noticeable difference w.r.t. the implementation of Sec. 3.3 resides in how the count of the frames with motion is kept. In the implementation with native threads there is no collector and the counter is a shared variable whose accesses are synchronizd. Here, instead, there is a collector which is responsible for updating the counter. This required the embedding of a motion flag in the frame (realized with a simple struct), but, since the collector is the only one to work with the counter, its access does not need to be synchronized.

3.4.1 Compile and execute

In the base directory of the project, run:
To compile: `make ff` or `make all`
To execute: `./bin/main_ff.out <path to video> <number of workers>`

3.5 OpenMP parallel for implementation

OpenMP's pragmas provide an extremely simple way of parallelizing for loops. It is the case that every one of the three main operations to run on each frame has to iterate over the whole image through two nested for loops. For this reason, they are suited to be parallelized using pragmas. Specifically, I added the line `#pragma omp parallel for num_threads(nw)` on top of each one to the outer cycles[2].

[2]In the motion detection function, it is technically `#pragma omp parallel for reduction(+:n_different_pixels) num_threads(nw)`. I briefly compared it with the standard parallel for (without reduction), but there wasn't much difference in performance.

Since all the implementations (sequential, pthreads, FastFlow[1]) use the same sequential stages, it was possible not only to use the pragmas as the sole parallelization method, but also to use them in combination with the already-parallel implementations of Sections 3.3 and 3.4.

3.5.1 Compile and execute

Pragmas as the sole parallelization method Run the sequential version, but pass as arguments the number of workers to use in the pragmas.
In the base directory of the project, run:
To compile: `make sequential` or `make all`
To execute: `./bin/main_sequential.out <path to video> [<nw rgb2gray>] [<nw smoothing>] [<nw motion detection>]`

Pragmas together with threads implementation In the base directory of the project, run:
To compile: `make threads` or `make all`
To execute: `./bin/main_threads.out <path to video> <number of main workers> [<nw rgb2gray>] [<nw smoothing>] [<nw motion detection>]`

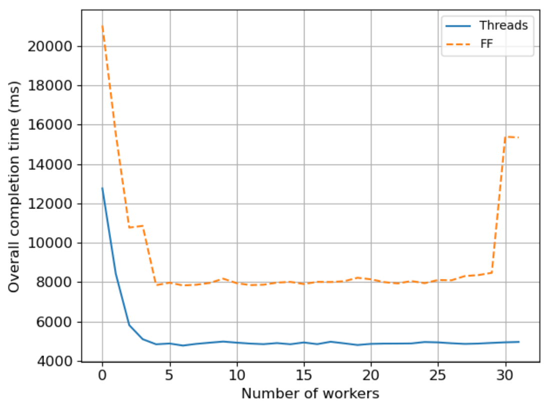
Pragmas together with FastFlow implementation In the base directory of the project, run:
To compile: `make ff` or `make all`
To execute: `./bin/main_ff.out <path to video> <number of main workers> [<nw rgb2gray>] [<nw smoothing>] [<nw motion detection>]`

3.6 Performance of the parallel implementations

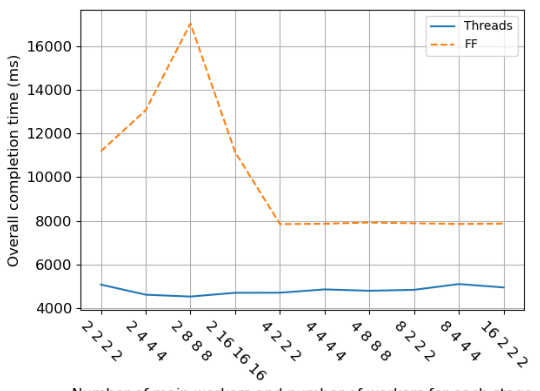
To get meaningful results, the same video has been used in all the following tests.

Completion times Three different tests have been made for the completion time: the first (Image 1(a)) compares the implementation using native C++ threads (Sec. 3.3) with the one using FastFlow[1] (Sec. 3.4). The test has been performed with a number of workers varying from 1 to 32. These numbers have to be intended excluding the parent thread pushing frames in the pthread version, and excluding the emitter and collector in the FlastFlow version. It is possible to notice that the first version is always a little faster than the latter, but it is also more "ad-hoc", not relying on a library implementation of the farm. Table 3 reports the minimum completion time for each implementation and the corresponding number of workers.

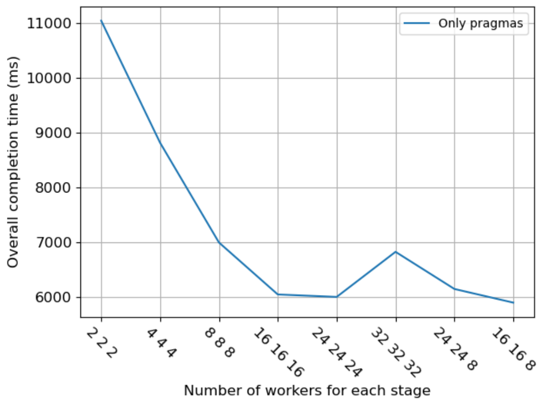
Speedups All the implementations eventually reach a flatspot in their speedup plot (Figure 2). In particular, only the pthread version (with pragmas too) surpasses slightly the three-fold speedup.



(a)



(b)



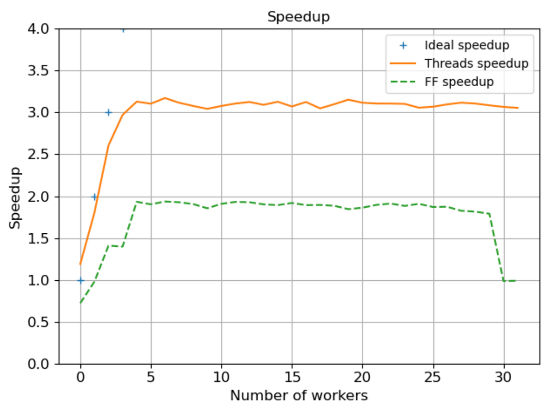
(c)

Figure 1: **(a)** Tc of the implementations with native C++ threads and with FastFlow[1]; **(b)** Tc of the same implementation of (a) "enhanced" with pragmas; **(c)** Tc of the pure pragmas parallel version.

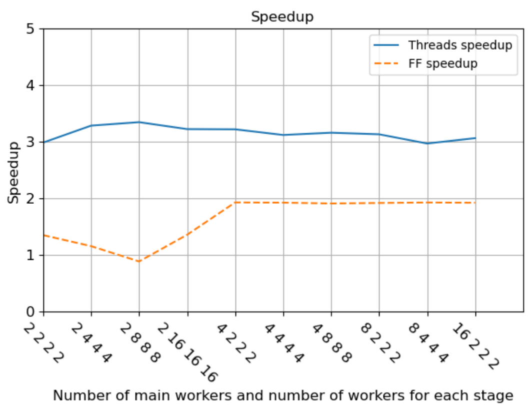
Implementation	Minimum completion time	Corresponding number of workers
Native C++ threads	4782.3 *ms*	7
FastFlow	7837.0 *ms*	7
Threads with pragmas	4528.2 *ms*	2 8 8 8
FastFlow with pragmas	7851.9 *ms*	4 2 2 2
Only pragmas	5906.4 *ms*	16 8 8 8

Table 3: Minimum completion time and corresponding number of workers for each implementation.

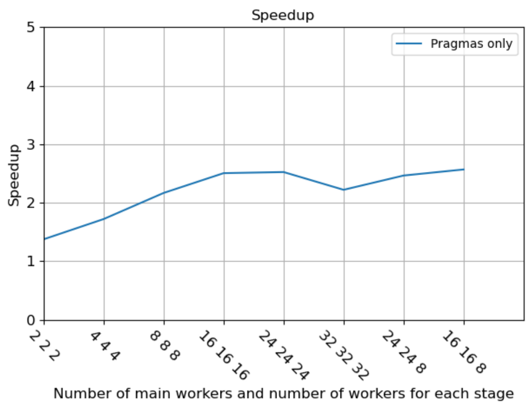
Efficiencies Concerning the efficiency, Figure 3 shows how the two implementations follow a similar trend, with the native C++ threads beigh slightly more efficient, especially for a smaller number of workers. As this grows, FastFlow starts to reach the same values.



(a)



(b)



(c)

Figure 2: Speedups of the various implementations: **(a)** native C++ threads and FastFlow; **(b)** same implementations of (a) "enhanced" with pragmas; **(c)** pure pragmas parallel version.

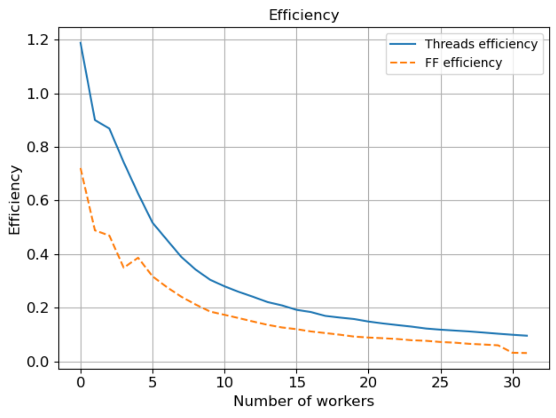


Figure 3: efficiency of the implementation with native C++ threads and with FastFlow.

References

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. John Wiley Sons, Ltd, 2017.

[2] Kevin P. Rauch and University of Maryland. Rplsh. https://www.astro.umd.edu/ rauch/rplsh/.