Università di Pisa
Scuola Superiore Sant'Anna

Master in Computer Science and Networking

SPM Project Report

Image Watermark

Elena Bucchianeri
463889
Academic year 2017-2018

1 Introduction

The goal of the project is to implement one sequential version and two different parallel version of the Image Watermark application and to give experimental results validating performance and design. The first parallel version has been implemented using the C++ standard library, the second one has been implemented using FastFlow framework.

2 Design

This section describes sequential and parallel versions without digging too much in the details of implementation.

2.1 Sequential version

Two sequential versions are realized.

The **first sequential version** is the real sequential one, it is implemented in file *sequenziale.cpp* and it can be described by the pseudocode in the Listing 1.

```
1 for (image_name : folder){
2     image.load(image_name)
3     applyMark(image, watermark)
4     image.save()
5 }
```

Listing 1: First sequential version pseudocode

Each image contained in the folder is loaded, the watermark image is applied and then image is saved. This application is composed by a great amount of readings (loading) and writings (saving) from disk, which are actions not parallelizable. The disk is a single shared and slow resource and all accesses to disk happen in a sequential way, therefore reading and writing actions are parts of the serial fraction and the watermark application is the non-serial fraction. According to Amdahl's law, the achievable speedup is limited by the serial fraction of the program. An evaluation of the law is shown in section 5.1.

A possibile parallel implementation based on this sequential version could be a farm with the emitter that reads images and streams them to the workers. Each worker applies the watermark on the image received and sends the result to the collector. The collector saves one by one all the received images, in a sequential way. This is not a good solution because the emitter sends images to workers slowly while the computation part of the workers is very quick. In this case emitter and collector are the bottleneck for the farm. The speeup is very low.

So, in order to achieve a better speedup, in the **second sequential version** of the program, **reading and writing parts are excluded from the computation to parallelize**. This second version is implemented in the file *main_seq.cpp* and the computation can be described by the pseudocode in Listing 2.

```
1 load(vectorimage, folder)
2 for (image : vectorimage) {
3     applyMark(image)
4 }
5 save(vectorimage)
```

Listing 2: Second sequential version pseudocode

The first function [line 1] loads all the images in a vector and it can be considered an initial reading phase. The save function [lines 5], at the end, saves all the image and it is considered a final writing phase. Only the `for` [lines 2-4], that is the watermark application, is the part of computation to parallelize, leaving out the reading and writing part.

2.2 Parallel version

The parallel pattern adopted to parallelize the computation [lines 5-7] in Listing 2 is a *farm*. After the initial phase of image loading, the emitter receives the vector of images and then it streams reference of images to the workers. Each worker applies the watermark on the received images and, receiving a reference, it modifies images in the vector. No need of a collector. The saving will be done after the ending of all workers.

Assuming an even distribution of the m images of size N among n workers, the performance model for the *completion time (Tc)* of the farm can be derived as:

$$T_{\mathrm{c}}(m, N, n) \approx T_{\text{init-threads}}(n) + m \times T_{\mathrm{s}}(N, n) \tag{1}$$

$$T_{\mathrm{s}}(N, n) \approx max\{T_{\mathrm{e}}, \frac{T_{\mathrm{w}}(N)}{n}\} \tag{2}$$

Where:

- $T_{\text{init-threads}}(n)$ is the time to create n thread workers;
- $T_{\mathrm{s}}(n)$ is the service time of the farm;
- T_{e} is the emitter service time;
- $T_{\mathrm{w}}(N)$ is the service time of a worker.

3 Implementation

The goal of this section is to explain how different versions are implemented, eventually showing the most important parts of the code.

3.1 General ideas

In all parallel versions the emitter sends to the worker a reference to an image using the shared memory. This solutions avoids to send the entire image from emitter to a thread worker. Every image reference, transmitted from emitter to threads, is not a reference to a CImg image directly, but instead it is a reference to an object of type `immagine`, which represents an image. The class `immagine` is used for combining a Cimg image with its final name, therefore it contains only public fields: a pointer to CImg image and a string used for storing the final name of the image. There is no need to protect accesses to a single image, because an image is created before threads and modified only by a single thread worker.

3.2 C++ Thread implementation

The C++ thread parallel version is implemented using a `farm` without a collector. Two parallel versions are realized: *main_thr.ccp* and *main_thrnorr.ccp*.

In the first C++ thread implementation (file *main_thr.ccp*) each thread has a queue shared only with the emitter. The queue is a single producer-single consumer queue and it is implemented in the file *queue_attiva.hpp*. It contains a *deque* of reference to the *immagine* objects and it has two methods: *pop()* and *push()*. The emitter pushes images in the threads' queues in a round-robin way. The thread executing the *pop()* is blocked in an active wait if queue is empty. The emitter interrupts the wait of the thread inserting an element in the queue with the *push()*.

In the the second C++ thread implementation (file *main_thrnorr.ccp*) all threads receive the images from emitter through a single shared queue. The queue, implemented in file *queue.hpp*, contains a *deque* of reference to the *immagine* objects and it has two methods: *pop()* and *push()*. Differently from the other queue a condition variable manages threads. The thread, executing the *pop()*, is suspended and descheduled if the queue is found empty. The emitter, executing *push()* method to add an element, wakes up one suspended thread with the *notify_one()*. Clearly this solution is better with a small parallelism degree, where less threads access the shared queue.

In both versions, initially, all threads are created and blocked on the empty queue, trying to pop from it something. Then the emitter puts in the queue one image reference at time, waking up or unblocking on thread at time. The Listing 3 shows the code of the threads. In an infinite loop, lines [4-10], a thread pops an image reference from the queue and executes the function `applicafirma`, which applies the watermark. This loop is broken only when the thread receives from the queue a NULL pointer. Therefore the emitter, after sending all images references, sends a NULL pointer exactly n times, that is one NULL pointer for each thread.

```
1  void func(CImg<IMGT>* firma, queue<immagine*>& input) {
2      immagine* current = NULL;
3      bool continua = true;
4      while(continua){
5          current=input.pop();
6          if (current!=NULL){
7              applicafirma(current, firma);
8          } else
9              continua = false;
10     }
11 }
```

Listing 3: Thread function code

3.3 FastFlow implementation

The FastFlow parallel version (file *main_ff.ccp*) is implemented using a `ff_Farm` without a collector. The emitter receives the vector of images and then sends images to workers in a round-robin way.

The code of the worker is shown in Listing 4. At the creation it receives the watermark to apply and then, for each image received from the emitter, it applies the watermark using the function `applicafirma()`.

```
1  struct worker: ff_node_t<immagine> {
2      CImg<IMGT>* firma;
```

```
3
4       worker(CImg<IMGT>* f){
5           firma = f;
6       }
7
8       immagine *svc(immagine *t){
9           applicafirma(t,firma);
10          return GO_ON;
11      }
12  };
```

Listing 4: Worker code

4 User manual

- "**src**" folder: contains all .hpp and .cpp files needed for the compilation.
- "**test**" folder: contains the images used in testing the application. The watermark image is named "2048_2_firma.jpg" and the collection of images is represented by the subfolder "2048x2048", containing only one image of size 2048x208.
- "**plot**" folder: contains plots (*.png) and two subfolder, one for the test with 500 images and the other for the test with 1000 images. Each subfolder contains all files needed to create plots and a README that explains how these files can be used to generate plots.
- "**makefile**" to compile different versions.
- "**script**" to check if two folder contains the same images.

4.1 Compilation

A Makefile is provided for the compilation of the application.

- `make seq.esteso.bin`: compilation of the first sequential version.
- `make main.seq.bin` : compilation for the second sequential version.
- `make main.thrnorr.bin` : compilation for the C++ thread version, with shared queue.
- `make main.thr.bin` : compilation for the C++ thread version, with active wait.
- `make main.ff.bin` : compilation for the FastFlow version.
- `make all` : this command cleans the environment and compiles all versions.

4.2 Usage

- To execute the first sequential version *seq.esteso.bin*:

```
./seq.esteso.bin 4 test/2048x2048 Risultati test/2048_2_firma.jpg
```

The **first argument** indicates the number of times each image in the folder is loaded, the **second argument** indicates the folder where to find the images, **third argument** is the subfolder in which the new images are saved and the **fourth argument** is the name of the watermark image.

- To execute the second sequential version *main.seq.bin*:

```
./main.seq.bin 4 test/2048x2048 Risultati test/2048_2_firma.jpg 0
```

The first four arguments are equal to the first sequential version. The **last argument** can be 0 or 1. If it is 0, the images in the folder are loaded from disk for a number of times specified by the first argument (as above, for the first sequential version). If it is 1, the images in the folder are loaded in memory from disk only one time, then the images in memory are copied for the number of times specified by the first argument. In this version the reading part is not considered so 1 as last parameter reduces the time need to read from disk.

- To execute the parallel versions *main.thr.bin*, *main.thrnorr.bin* and *main.ff.bin*:

```
./main.ff.bin 4 test/2048x2048 Risultati test/2048_2_firma.jpg 2 0
```

The first four arguments are equal to the first sequential version. The **fifth argument**, used only in parallel versions, is the parallelism degree. The **last argument** can be 0 or 1 and it indicates if images are all loaded or copied (as above, for the second sequential version). In this execution of the FastFlow version, for example, the only image in the "test/2048x2048"" folder is loaded 4 times and 2 workers are used in the farm. The four final images are saved in the subfolder "Risultati".

- The script "**script**" can be used for verifying that the output of different versions on the same input is the same. The script takes as arguments the name of two subfolder:

```
./script Risultati RisultatiSeq
```

4.3 Outputs

Each version of the application prints in output some information. For example, one output from a FastFlow version is:

```
Le immagini saranno salvate in "Risultati"
Parallelo in FF, con caricamento in memoria delle immagini:  ./main.ff.bin
Tempo di completamento:  9282ms
Numero immagini:  1000
Numero thread:  4
```

First line declares where the images are saved. Second line describes the executable. Next lines contain: completion time, number of images and number of thread.

The first sequential version has in output also information about the time to read from disk and write to disk.

5 Experimental validation

This section shows experimental results for completion time of the sequential versions and completion time, speedup, scalability and efficiency of the parallel implementations. These measured values are compared with the expected ones evaluated with the performance model explained in section 2.2.

5.1 Test first sequential version

In order to demonstate what is explained in section 2.1 about the **first sequential version**, here an output of the execution with 1000 images is shown:

```
Le immagini saranno salvate in "Risultati"
Sequenziale, senza caricare prima tutte le immagini in memoria:  ./seq.esteso.bin
Tempo di completamento:  834829ms
Numero immagini:  1000
Tot read disco:  263548ms
Tot write disco:  537566ms
Tot lavoro:  33715ms
```

The time to read from the disk, that is the total time to open 1000 images, is about the 32% of the completion time and the time to write to disk, that is the total time to execute 1000 times *save()* method, is about the 64% of the total time. The computation is only the **4%**.

Assuming that the method to open an image and the method *save()* have no parallelizable parts at all, so the serial fraction here is about the 96% of the total.

According to Amdahl, if $f \in [0, 1]$ is the serial fraction, and so $1 - f$ is the non serial fraction, the total speedup achieved is limited:

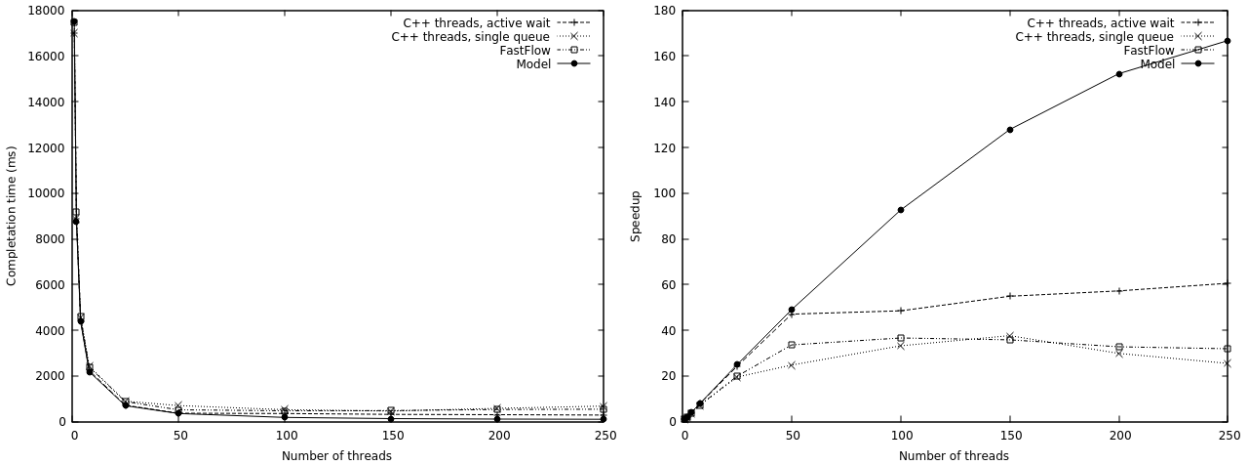
$$\lim_{n\to\infty} Speedup(n) = \lim_{n\to\infty} \frac{T_{\text{seq}}}{T_{\text{par}}(n)} = \frac{1}{f} \tag{3}$$

In this case:

$$\frac{1}{f} = \frac{1}{0.96} = 1.04 \tag{4}$$

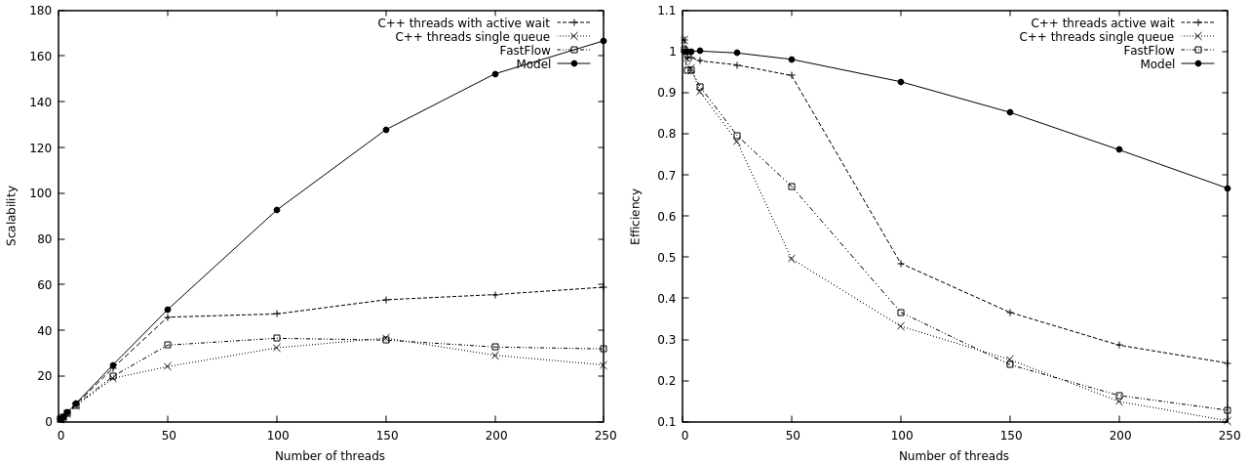
5.2 Performance

The tests for the parallel versions have been done with two different size of the stream. The first size is **500 images** and the second test is with **1000 images**. In both cases the size of the image is the same: 2048x2048.



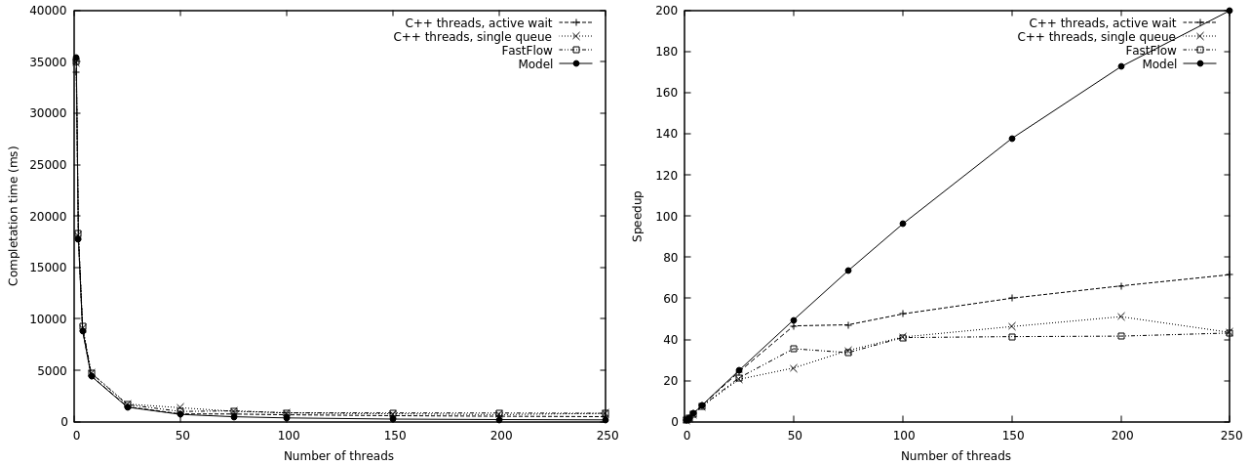
(a) Completion time (b) Speedup

Figure 1: Completion time and speedup with **500 images**.



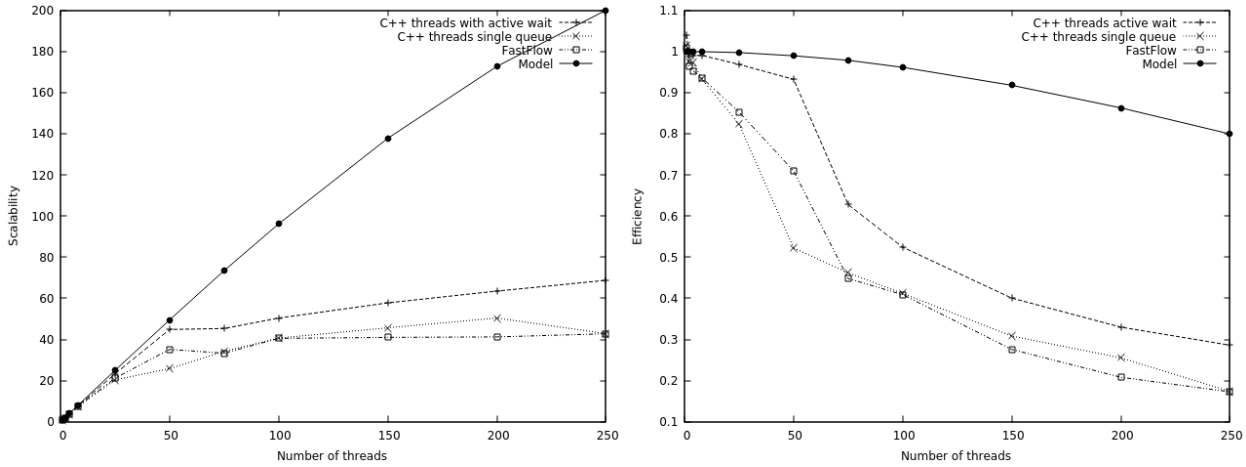
(a) Scalability (b) Efficiency

Figure 2: Scalability and efficiency with **500 images**.



(a) Completion time (b) Speedup

Figure 3: Completion time and speedup with **1000 images**.



(a) Scalability (b) Efficiency

Figure 4: Scalability and efficiency with **1000 images**.

	Tc sequenziale
500 images	17508ms
1000 images	35373ms

Table 1: Completion time of the sequential cases using *main.seq.bin*.

	Thread active wait *main.thr.bin*		**Thread single queue** *main.thrnorr.bin*		**FastFlow** *main.ff.bin*	
n	Tc(n)	Speedup(n)	Tc(n)	Speedup(n)	Tc(n)	Speedup(n)
50	372ms	47.0645	706ms	24.7989	521ms	33.6046
100	361ms	48.4986	527ms	33.222	478ms	36.6276
150	319ms	54.884	465ms	37.6516	489ms	35.8037
200	306ms	57.2157	586ms	29.8771	434ms	32.7865
250	289ms	60.5813	686ms	25.5219	548ms	31.9489

Table 2: Some values obtained to calculate graphs in figure 1 with **500 images**.

	Thread active wait *main.thr.bin*		**Thread single queue** *main.thrnorr.bin*		**FastFlow** *main.ff.bin*	
n	Tc(n)	Speedup(n)	Tc(n)	Speedup(n)	Tc(n)	Speedup(n)
50	759ms	46.6047	1354ms	26.1248	997ms	35.4794
100	675ms	52.4044	859ms	41.1793	865ms	40.8936
150	589ms	60.056	765ms	46.2392	855ms	41.3719
200	536ms	65.9944	693ms	51.0433	850ms	41.6153
250	495ms	71.4606	815ms	43.4025	820ms	43.1378

Table 3: Some values obtained to calculate graphs in figure 3 with **1000 images**.

Considering the stream of **500 images** (reference to tab. 2 and fig. 1):

- For the C++ thread version with active wait the max of the speedup is at $n = 250$, where $speedup(250) \simeq 61$.
- For the C++ thread version with single queue the max of the speedup is at $n = 150$, where $speedup(150) \simeq 38$.
- For the FastFlow version the max of the speedup is at $n = 100$, where $speedup(100) \simeq 37$.

Considering the stream of **1000 images** (reference to tab. 3 and fig. 3):

- For the C++ thread version with active wait the max of the speedup is at $n = 250$, where $speedup(250) \simeq 71$.
- For the C++ thread version with single queue the max of the speedup is at $n = 200$, where $speedup(200) \simeq 51$.
- For the FastFlow version the max of the speedup is at $n = 250$, where $speedup(250) \simeq 43$.

The two C++ thread parallel versions have different values because the active wait version can reach higher performance than the single queue version. This is due to the fact that the overhead related to the queue in the active wait version is smaller than the overhead in the single shared queue version. Actually, in the active wait version each thread has its own queue, so the synchronization due to the lock/unlock is only with the emitter and also, in case of an empty queue, the thread

performs an active wait. Instead, in the single shared queue version, all thread accesses the same queue. The time to pop an element may be greater with high number of thread, because the synchronization of lock/unlock is performed between all threads executing a pop and the emitter in the push. Moreover, at the beginning, all threads are suspended on the empty queue, so the emitter needs to notify (and wake up) all threads, one bye one.

Measured values are different from the expected ones evaluated with the model of section 2.2. This difference can be caused by several reasons:

1. According to the model, m (number of images) should be much greater than n (number of threads). "Much" may mean a factor of 100 or 10, but in these tests m is not so big, so the measures for the completion time of the model, for high values of n, are not accurate.

2. The model does not consider the overhead related to access of the queue.