[305AA] Parallel and distributed systems: paradigms and models (Project 2.2: Image Watermarking)

Department of Computer Science
University Of Pisa

June 2018

Contents

Chapter 1

Problem & Parallel Architecture Design

In this section we describe the design of the architecture (of several architectures, in fact), that may be used to solve the watermark problem. This problem consists in embedding a first image, the watermark, into a stream of images. Each image of the stream and the watermark typically mix-up by means of some element-wise operation on the pixels corresponding to the same positions. We call *watermarking*[1] the whole process of embedding the watermark into an image. If the dimensions of the stream images and the watermark do not match, we will simply apply a modulo operation on the watermark if it is smaller, or only embed part of it in case it is bigger.

We consider and analyze two cases:

1. The images are loaded from some directory on a storage device, watermarked, and saved back in a different directory;

2. We assume the images are readily available in memory, and after we watermark them we do not take into account the storing phase. Thus, we only focus on the watermarking process.

We will see that the bottlenecks in the two cases are different and it is interesting to analyze the performance in both of them.

The parallel architectures considered in the first case are schematically depicted in figure 1.1 and 1.2. In the second case the architecture is simplified to the one depicted in figure 1.3 Consider the three different type of operation represented in figures 1.1, 1.2 and 1.3. While in chapter 2 we will formalize this, here we illustrate their specification and discuss some of the ways in which they may be parallelized, intuitively pointing to the advantages that characterize the different methods.

- *Load*: This consists in reading an image, from some storage device, and loading it into memory. We may send multiple load requests in parallel, allowing the underlying system to optimize the actual reads from the storage device. The bandwidth of the storage device is also to be taken into account.

- *Mark*: This consists in applying some element-wise operation to the pixels of the images we wish to watermark. In this project the operation of choice is a simple weighted sum of the RGB values of the pixels. Once we have the image into memory, we may think of different ways of carrying out the watermarking process. The simplest method one could think about is to parallelize over the different images - this may work well when dealing with images of small size or when we have a large number of images readily available so that we can schedule them in a clever way, that achieves a good load balancing. Instead of taking as minimal unit the image, we could think to reduce it further and look at the single rows, columns or even individual pixels of the image. This makes it possible to increase the parallelism degree beyond the number of images, thus permitting to deal more efficiently with situations such as the handling of a stream of large images.

- *Save*: In our case this consists in storing a image that has been watermarked back on the storage device. Here as well we may send out several storage requests in parallel, allowing the underlying system to optimize the actual storage.

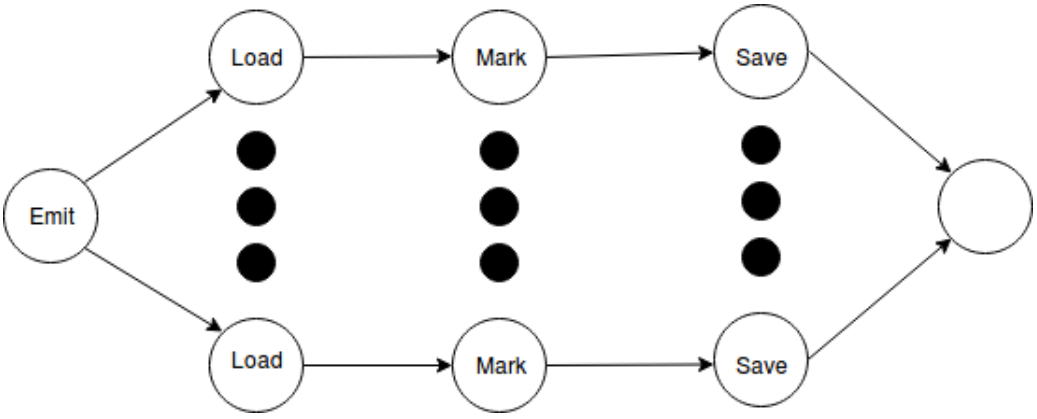

Figure 1.1: Farm of pipelines.

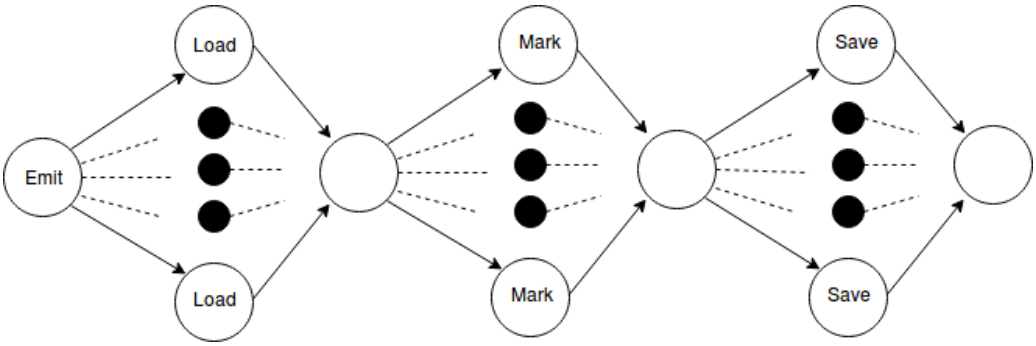

Figure 1.2: Pipeline of farms.

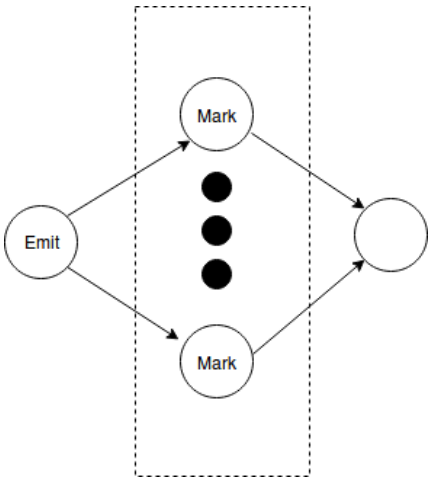

Figure 1.3: Here we focus only on the marking stage, assuming the time required to emit the images is negligible (e.g. images pre-loaded in memory and just passing the references).

Chapter 2

Performance Modeling

Let's now build the theoretical performance model and in chapter 4 we will see how this reflects on a running instance on a real machine.

2.1 Load, Mark and Store model

We start with a simple sequential model that loads, watermarks and then stores each image. Let us assume that the images are all of equal dimension and reason on that case, otherwise we could just expand the reasoning to the pixel level but it would needlessly complicate the analysis as well as excessively deviate from a real implementation where many other factors and properties such as locality can greatly influence the outcome.

Consider the following measures in the sequential case: let t_l be the loading time of a single image, t_m its watermarking time and t_s the storage time. Furthermore, let d be the number of images. The time required by the simple sequential model to process the d images is then $d*(t_l+t_m+t_s)$ - that is, for each image, we first load it, then apply the watermark and ultimately store it. The **throughput** in this case is $1/(t_l + t_m + t_s)$.

Consider now the more involved models in 1.1 and 1.2. As a general note, take into account that these models require synchronization and communication (for instance when assigning the task or gathering the results), so we have to keep in mind that there is an additional t_h that we will not explicitly model but whose presence will become apparent in the experiments. Let's start with the farm of pipelines (**fop**). Remember that a pipeline does not improve the latency of individual tasks and that the **throughput** of a pipeline is bounded by the throughout of its slowest stage. Here we assume to have p identical pipelines. The theoretical **throughput** is then given by $\frac{p}{max(t_l,t_m,t_s)}$, as we have p units operating in parallel, and each individual unit has a **throughput** bounded by its slowest stage. The expected processing time for d images, assuming the pipeline is in its steady-state, would then be $\frac{d*max(t_l,t_m,t_s)}{p}$. We can compute the **speedup** with respect to the sequential model as shown in equation 2.1: what this equation tells us, is that for a single pipeline we would obtain a maximum **speedup** of 3 if the stages of the pipeline were

perfectly balanced (i.e. $t_l = t_m = t_s$), and such gain would worsen as the time required by one stage prevails over the others. A p term greater than 1 amplifies this **speedup**, but we have to put some care into choosing it:

- If we decide to not split the individual images at all, then if the inter-arrival time of a set of d images is greater than the processing time of d images, choosing a $p > d$ is of no use.
- If we actually split the images into smaller chunks then we can improve the **throughput** by increasing p - notice the p term in the throughput of the **fop**. However in a real situation, besides having limited resources that we can allocate to construct the pipelines, splitting the images in smaller chunks will inevitably introduce some overhead as well.

$$speedup(fop) = \frac{d * (t_l + t_m + t_s)}{\frac{d * \max(t_l, t_m, t_s)}{p}} = \frac{p * (t_l + t_m + t_s)}{\max(t_l, t_m, t_s)} \tag{2.1}$$

In a similar fashion we can determine the **speedup** of the specular pipeline of farms (**pof**) model. Here we have one pipeline only, but each of its stages $s1, s2$ and $s3$ is a farm of respectively f_1, f_2 and f_3 nodes. Once again the **throughput** is bounded by the slowest stage, but since now the stages are actually farms and not single nodes, we can act on them and in this manner effectively impact it: we can express the **throughput** as in 2.2 and the **speedup** as in 2.3. This gives us a way to impact the **speedup** by allocating more resources on the slowest stage of the pipeline.

$$\begin{aligned} &throughput(pof) = \min(throughput(s1), throughput(s2), throughput(s3)) = \\ &\min(\frac{f1}{t_l}, \frac{f2}{t_m}, \frac{f3}{t_s}) \end{aligned} \tag{2.2}$$

$$speedup(pof) = \frac{d * (t_l + t_m + t_s)}{\frac{d}{\min(\frac{f1}{t_l}, \frac{f2}{t_m}, \frac{f3}{t_s})}} = (t_l + t_m + t_s) * \min(\frac{f1}{t_l}, \frac{f2}{t_m}, \frac{f3}{t_s}) \tag{2.3}$$

2.2 Restricted Watermark model

As we will see in the experiments, the loading and storing phases that need to interact with the underlying storage device soon become a bottleneck. It is of interest to analyze what would happen if we did not have such bottleneck - for this reason here we assume that the images are readily available (we will just pass pointers around after pre-loading them) and that we do not have to take care of storing them back on the storage device. In other words we assume t_l and t_s to be negligible with respect to t_m.

Once again we begin by looking at the sequential model and as it is easy to guess, we simply need to take out the t_l and t_s terms to obtain the new processing time $d * t_m$ and **throughput** $1/t_m$.

Since we only look at the watermarking phase, we now employ the model **fo** in 1.3. As previously noted, we can split the images into chunks as small as we wish - up to individual pixels - and assign each chunk to a separate node: the chunks do not have any overlap among them, and besides the overhead caused by the need of some form of communication such as the need to determine when an image has been completely marked, no additional processing work is created due to the chunking. This leads us to a total processing time, given n nodes in the **fo** model, of $\frac{d*t_m}{n} + t_h$. Differently from before, let us explicitly express all the overhead created via a single term t_h. The **speedup** can be determined as in 2.4: ideally, if we could parallelize without introducing any overhead ($t_h = 0$) we would achieve a **speedup** of n.

$$speedup(fo) = \frac{d * t_m}{\frac{d*t_m}{n} + t_h} = \frac{n * d * t_m}{d * t_m + n * t_h} \tag{2.4}$$

Now that we have these expected processing times, we'd like to see if we can find out where the bottleneck is - in other words when the overhead term t_h outweighs the benefit obtained by parallelizing over multiple workers. We will do this in the experiments in chapter 4.

Chapter 3

Structure, Implementation and Setup

We have different implementations, representing the various models presented so far. In standard `C++` a sequential implementation and a parallel one, with pre-loading, representing the model in figure 1.3 are provided. In **FastFlow**[2] both the restricted model in 1.3 and the complete models in figures 1.1 and 1.2 are implemented.

3.1 Structure

The structure of the project is as follows:

- `Makefile`: used to compile and link everything.
- `src/my_utils.[h|cpp]`: contains the definition of a number of structures and functions used across the various implementations of the models.
- `src/CImg.h`: library used to manage the images[3].
- `src/ffwatermarker.cpp`: **FastFlow** implementation of the watermarker models 1.1 and 1.2, the parallelism type is to be chosen with the `-p` flag. Default is 0, corresponding to a farm of pipelines, while 1 corresponds to a pipeline of farms.
- `src/middleffwatermarker.cpp`: **FastFlow** implementation of model 1.3.
- `src/seqwatermarker.cpp`: sequential implementation of the watermarker.
- `src/watermarker.cpp`: parallel implementation of the model in 1.3 using only standard `C++` mechanisms. Loading and Saving are also parallelized but, unlike in the `src/ffwatermarker.cpp` implementation, are not in a pipeline.
- `imgs/watermarks`: contains a set of watermarks that may be used as input via the `-w` flag, providing the appropriate path.
- `imgs/dataset5`: contains a dataset of 256 **256**×**256** images.

- `imgs/dataset6`: contains a dataset of 2 **23123**×**5093** images.

3.2 Implementation notes

To parallelize the watermarking computation, the `chunker` function in `my_utils.cpp` is used. This function splits a `CImg<float> *img` into n equally sized parts. For instance a 4×3 image would be split into 3 as shown in figure 3.1. In the implementation provided it is possible to specify the number of chunks to split each image into via the `-c` flag - the default value splits it into as many chunks as is the parallelism degree, given by the `-n` flag. If the desired behaviour is to operate at the image level, we just need to specify `-c 1` when running the program. The standard `C++` version uses a simple queue mechanism: after an image is split into a number of tasks, each defined by an `img_chunk` to be processed as well as a pointer to the image itself, these tasks are pushed into a queue that is then emptied by a number of workers as they process the chunks. No further synchronization is needed among the workers while processing the chunks as these have no overlap - that is each pixel of the image resides in exactly one chunk. The **FastFlow** implementation is straightforward - it has the definition of the `ff_nodes` corresponding to the nodes shown in figures 1.1 and 1.2, and then these nodes are arranged to form the desired parallel model.

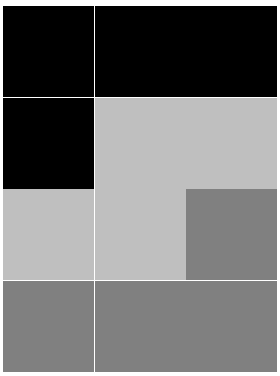

Figure 3.1: Splitting of a 4×3 image into 3 equally sized parts.

3.3 Setup and execution of the program

To setup the program and obtain the executables it is sufficient to run the `make` command in the project's directory. The executables will be created in the `out` directory. The four different executables have some shared and some unique flags: while in table 3.1 the set of available flags for each version is shown, a brief description of all the flags is the following, where * denotes that the corresponding flag is required to run the program:

- * `-s src_path`: Directory containing the images to be watermarked. The processed images will be put in a newly created `watermarked` directory in the given source directory path.
- * `-w watermark_file`: Path to the file to be used as watermark.
- `-c chunks`: Number of chunks to split each image into. It defaults to the parallelism degree if not specified.
- `-n parallelism degree`: Specifies the parallelism degree to run the program with. Defaults to 1 - note that this is not equivalent as the sequential version as setting up a parallel computation presents some overhead.
- `-p parallelism type`: Specifies the model to be employed, a value of 0 corresponds to a farm of pipelines (see figure 1.1) and a value of 1 corresponds to a pipeline of farms (see figure 1.2). Defaults to 0.
- `-i intensity`: Specifies the intensity of the watermark image. Ranges from 0 to 100, where 0 corresponds to a completely transparent watermark and 100 to a completely opaque one.

Table 3.1: Available flags for the different versions of the program

Flag / **Version**	s	w	c	n	p	i
seqwatermarker.cpp	✓	✓				✓
watermarker.cpp	✓	✓	✓	✓		✓
ffwatermarker.cpp	✓	✓	✓	✓	✓	✓
middleffwatermarker.cpp	✓	✓	✓	✓		✓

A sample of how to run the program is shown in figure 3.2.

```
→ project out/./ffwatermarker -s "imgs/" -w "imgs/watermarks/harambeblack.jpg"  -i 30 -n 2 -c 103 -p 3
Invalid parallelism type.
Usage -s <src_path> -w <watermark_file> -c <chunks> -n <parallelism degree> -p <parallelism type> -i <intensity>
-s src_path --- Directory containing the images to be watermarked
-w watermark_file --- Path of the watermark to be used
-c chunks --- Number of chunks to divide each image in, defaults to parallelism degree
-n parallelism degree --- Parallelism degree to be used
-p parallelism type --- 0 = farm of pipes, 1 = pipe of farms
-i intensity --- Intensity of the watermark image, from 0 (completely transparent) to 100 (completely opaque)

→ project out/./ffwatermarker -s "imgs/" -w "imgs/watermarks/harambeblack.jpg"  -i 30 -n 2 -c 103 -p 0
Watermark size: (289, 174)
Each image will be split in 103 chunks
FARM OF PIPES
Elapsed time is 1194 msecs
Processed a total of 8 images
```

Figure 3.2: Sample run of the **FastFlow** version of the program.

Chapter 4

Experiments

In this chapter we look at the performance of the various models on real data and on a real machine. The machine of use is a **PowerEdge C6320p** with 64 cores and 4 threads per core. All of the experiments are reported by taking the average over three trials.

First we consider `dataset5`. By looking at the performance of the sequential model compared to the parallel one with parallelism degree of 1 we hardly notice any difference as the overhead is relatively low - $10351\ msecs$ against $10288\ msecs$.

More interestingly, figure 4.1 shows the time in relation to the parallelism degree of the loading, watermarking and storing stages that are performed separately: as we can see the parallelism over the loading and storing stages peaks between 8 and 16, whereas the watermarking stage benefits from a higher parallelism degree. The time required by the central watermarking stage is also much lower in general which makes it glaring that the bottleneck resides in the other two stages.

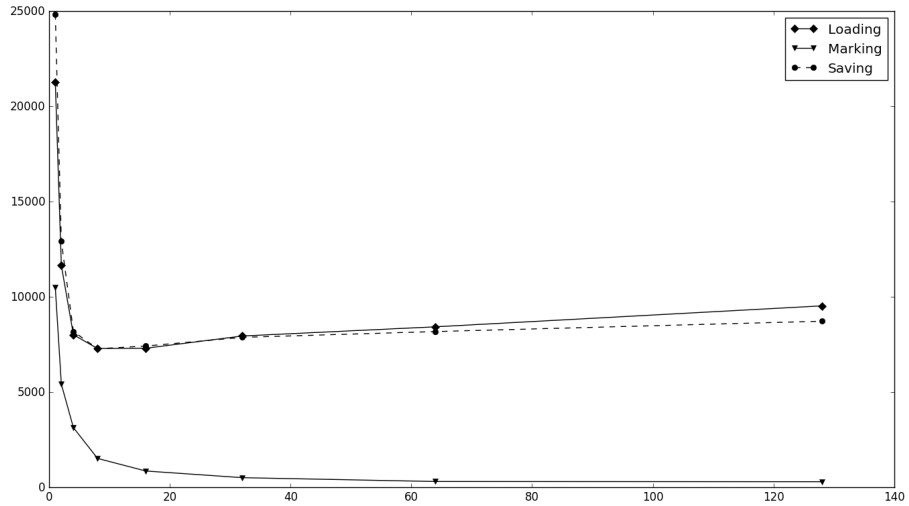

Figure 4.1: Running times of the three stages in relation to the parallelism degree

This is confirmed by the 1.1 and 1.2 models, whose performance is shown in figure 4.2, that largely sees the benefit of a higher parallelism degree disappear after a threshold ranging from 24 to 48 - that is the triple of what we just saw because only $1/3$ of the nodes are assigned to each of the stages.

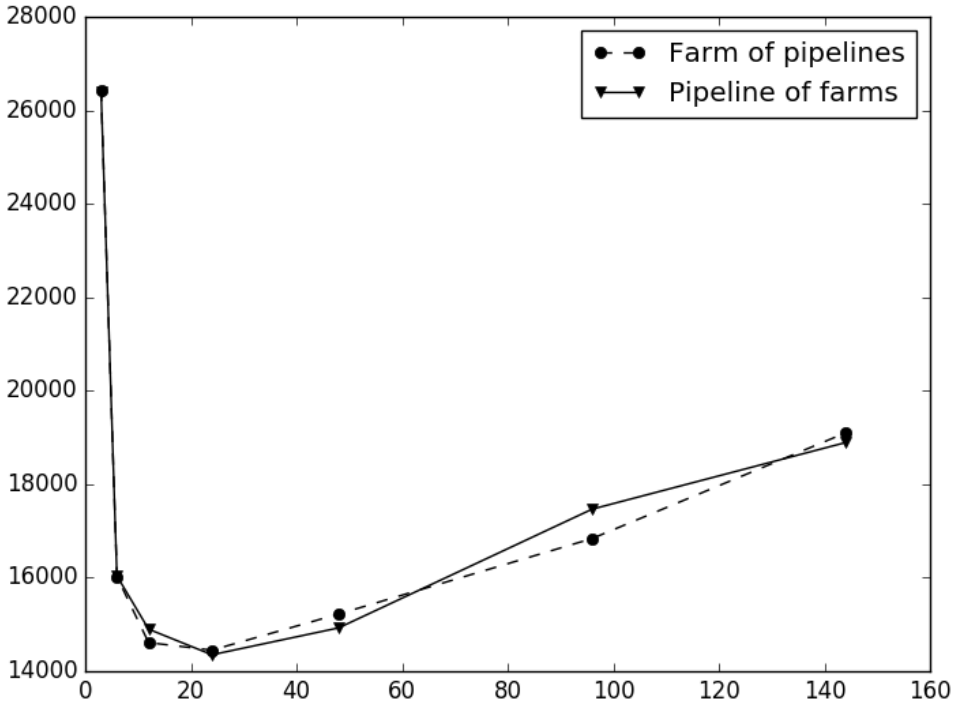

Figure 4.2: Running times of models 1.1 and 1.2 in relation to the parallelism degree

What we can say so far is that interaction with the storage devices is indeed the bottleneck - let us assume we do not need to manage this so that we can analyze what happens in the watermarking stage alone.

If we compare the **FastFlow** implementation and the standard `C++` ones we obtain the plot in figure 4.3. We can see how the overhead of **FastFlow** is generally heavier than the standard `C++` implementation - **FastFlow** performs worse with a parallelism degree of 64 than with one of 128, whereas the standard `C++` version still benefits from it, yielding a running time of 313 $msecs$ in the first case and of 300 $msecs$ in the second one. We can also investigate whether the theoretical **speedup**, presented in chapter 2, has been achieved. To do so let's look at table 4.1, presenting the actual **speedups** with respect to the sequential version - the overhead term in the **FastFlow** implementation appears to considerably more impactful than in the standard `C++`. Overall, up to a parallelism degree of 32 the parallel version is able to get reasonably close to the theoretical limit: also notice that this dataset contains 256 small images only, with a different dataset we might obtain different results.

Table 4.1: **Speedup** achieved on `dataset5` with respect to the sequential version.

Par. degree / Version	1	2	4	8	16	32	64	128
`C++`	0.994	1.967	3.926	7.726	15.196	27.003	41.534	41.992
`FastFlow`	1.040	1.806	3.005	5.600	10.466	17.830	26.722	21.010

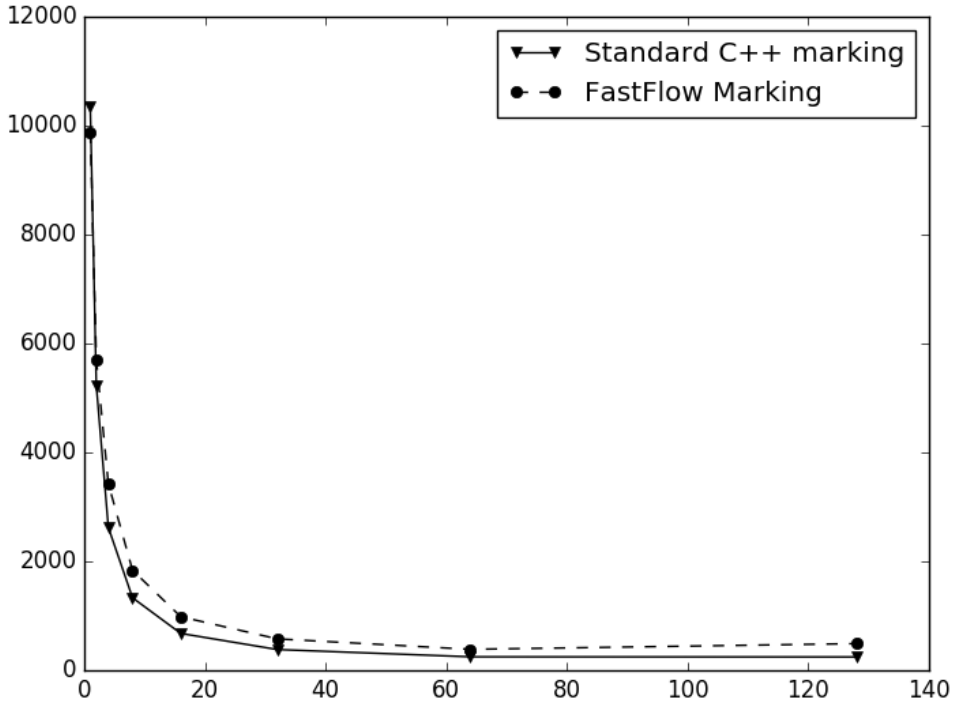

Figure 4.3: Comparison of the **FastFlow** implementation and the standard `C++` one.

It is interesting to see what happens when we change the way in which we chunk the images - to this purpose let's turn our attention to `dataset6`. This dataset contains only two large images and it will enable us to to see the importance of the way in which we split the images. In figure 4.4 a comparison on the number of chunks is shown: with only 2 images in the dataset if we use a chunk size of c then a parallism degree greater than $2c$ does not yield any benefit as we can notice when specifying 1 chunk per image. With a chunk size of 128 the situation is much better - it would require a parallism degree of 256 to reach the maximum parallelism. However if we overdo it and chunk the image excessively, for instance $4 * 10^6$ chunks per image, then the overhead introduced outweighs the benefits and we end up with a much worse processing time.

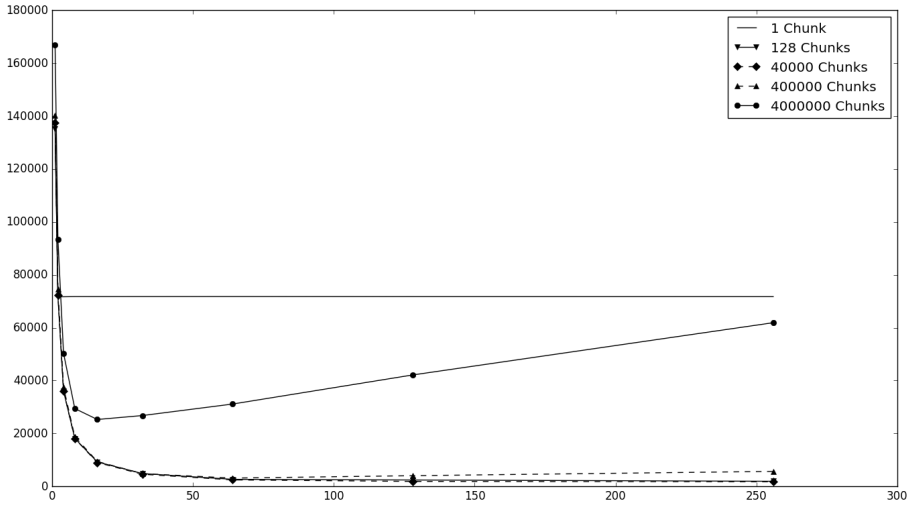

Figure 4.4: Comparison on `dataset6` over the number of chunks per image

Bibliography

[1] Chris Honsinger. Digital watermarking. *Journal of Electronic Imaging*, 11(3):414, 2002.

[2] Fastflow. `https://github.com/fastflow/fastflow`.

[3] Cimg. `http://cimg.eu/reference/`.