A parallel solution for image watermarking

Gemma Martini

Parallel and Distributed Systems
Paradigms and Principles

a.y. 2017-2018

Abstract

The task of image watermarking consists in operating on any single pixel of an image, in order to add a background effect to that image.

In the real World, such a technique is performed to "embed" the ownership of a logo or a template, a toy example of a watermark is displayed in this page.

In this paper, we studied how to parallelize the computation of watermarking a set of images, starting from a theoretical model and then digging into the technical details of the implementation.

3rd April 2019

Contents

1 Introduction

The watermarking problem was attacked preparing a vast set of black and white images as watermarks and a huge set of images to be watermarked.

Each pixel of the image which corresponds to a black pixel in the watermark needs to be updated.

We are going to show that this time is quite smaller than the time needed for loading and storing a picture, hence in this paper we also address a similar problem, which is a simplification of the original one: only one image gets watermarked and it is not stored on disk, after the operation. The strategy is to copy in main memory that specific image several times and the apply the watermark to each of them, accordingly to different algorithms.

2 A theoretical analysis

The problem of image watermarking can be explained decomposing it into 4 main operations. At first, the *watermark* needs to be *loaded* in main memory from disk; once this operation has been performed, we may proceed *loading* the *first image* to be watermarked, then it is *modified* and *stored*, until all the input images have been successfully watermarked.

Before digging into technicalities, we want to focus the attention of the reader on two main issues of this approach: first, we did not take into account any operation that allows us to keep track of which image has been processed and which are the ones that still need to be watermarked. On the other hand, this naıve approach, makes use of many unecessary conditional clauses in order to check whether the current pixel of the watermark is black (hence the image should be modified).

An attentive reader may notice at this point that two simple edits may be performed to overcome this issues. We add *two* intermediate *phases* between the operation of loading the watermark in memory and the load of the first image: we *process the watermark* in order to store only the positions of the black pixels and we *identify the input images* by means of a list.

A pictorial representation of such a solution is displayed in Figure 1.

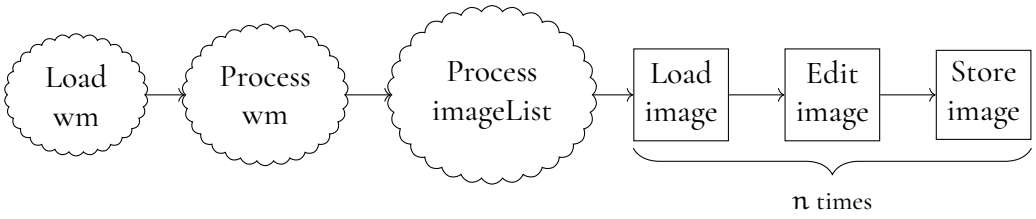

Figure 1: The curly tasks are of a sequential nature and concur to the value of the serial fraction, while the rectangular tasks can be parallelized.

where n is the number of input images to be watermarked.

Notice that in this problem the *serial fraction* is represented as the time spent to execute the three "cloudy" operation over the total cost of all the $3 \cdot n + 3$ operations. Formally,

$$f = \frac{t_{\text{load wm}} + t_{\text{process wm}} + t_{\text{process imgList}}}{t_{\text{load wm}} + t_{\text{process wm}} + t_{\text{process imgList}} + n \cdot (t_{\text{load}} + t_{\text{edit}} + t_{\text{store}})}$$

In this context the expected speedup using n_w workers is computed as follows

$$sp(n_w) = \frac{t_{seq}}{f \cdot t_{seq} + \frac{(1-f)\cdot t_{seq}}{n_w}}$$

Thanks to Gustaffson law we would expect the serial fraction to be amoritized when the data grain becomes finer and finer (the amount of data to be processed increases).

Let us analyze some parallel executions of such stages.

Since we are in presence of embarassing parallelism, we start by introducing the *map* pattern on this problem.

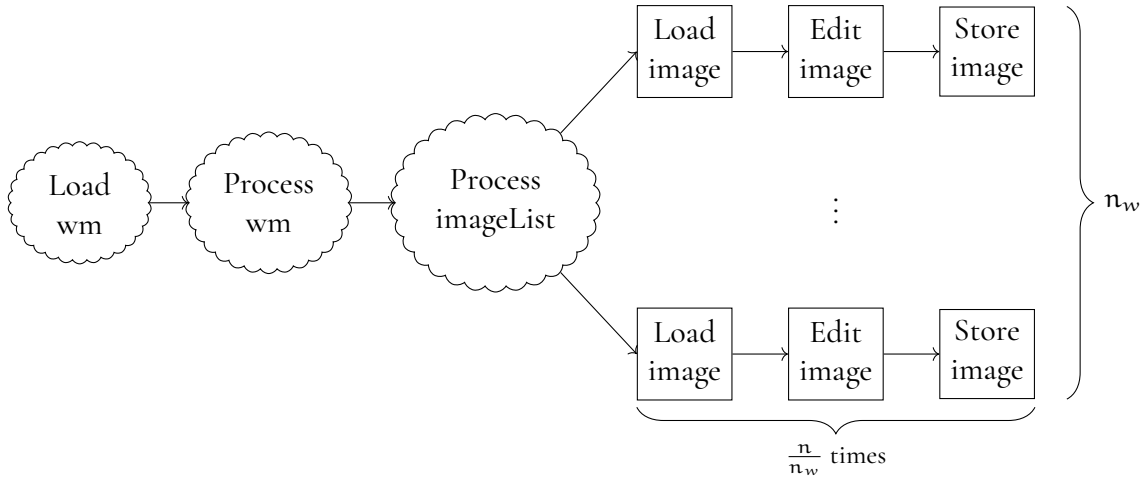

Figure 2: The tasks representd as curly circles cannot be parallelized, while the ones represented by squares can be (and are) parallelized.

In Figure 2 a pictorial representation of what can be formalized as
pipe(`load wm, process wm, process imglist,` **map**(n_w, `load img, edit img, store img`)).

On the other hand, this computation may be studied from another point of view using *stream parallelism*, as presented Figure 3.

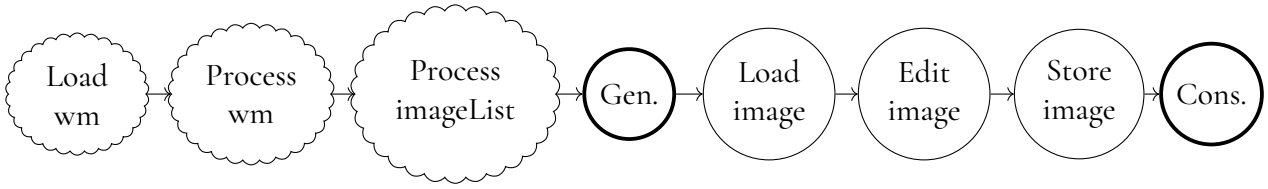

Figure 3: In this picture we represented as curly ellipses the tasks that need to be computed serially, while there are two rounded tasks (gen and cons) that generate and cosume the stream respectively, while the other three circles represent a task each.

Formally, **farm**(**pipe**(`load img, edit img, store img`)).

3 Practical scenario

In order to be able to have a glimpse of the time needed in the computations, the first thing to analyze is the time spent in executing the 6 algorithmic building blocks highlighted in the previous

section.

We decided to perform test on 8 different watermarks, with different black-density and on a set of 2,842 images of size 1920×1080px.

The sample of watermarks we picked is displayed in Figure 4.

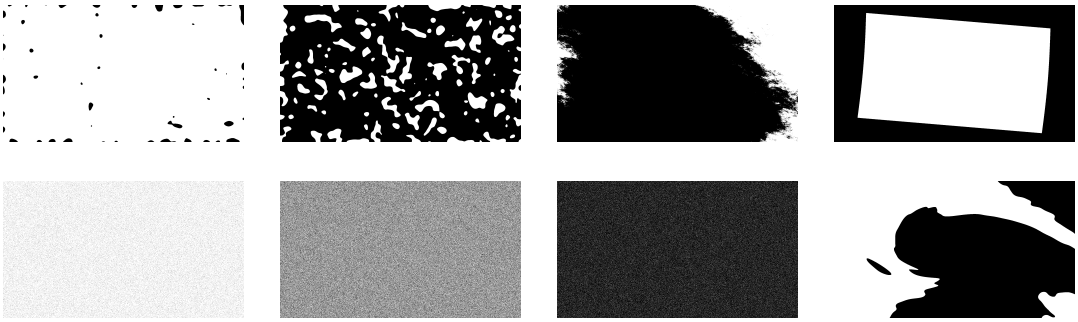

Figure 4: The 8 watermark images that have been used for experiments.

The experiments were run on two different machines:

- **64-core with hyperthreading Intel XEON PHI 7230**, where the cores work at a frequency of 1.30 GHz;
- **4-core with hyperthreading DELL XPS 2017**, where the cores are Intel i7-8550U, with a frequency of 1.80 GHz.

In the following lines the times needed to execute each of the six tasks presented in the previous section, with respect to the machine the experiments were run on.

The following times are the average value of thousands of experiments on different input images and multiple watermarks.

XPS:

- $t_{\text{load wm}} = 20144\mu s$
- $t_{\text{process wm}} = 8270\mu s$
- $t_{\text{process imgList}} = 1352\mu s$
- $t_{\text{load}} = 14588\mu s$
- $t_{\text{edit}} = 9194\mu s$
- $t_{\text{store}} = 14538\mu s$

Xeon Phi:

- $t_{\text{load wm}} = 78213\mu s$
- $t_{\text{process wm}} = 55046\mu s$
- $t_{\text{process imgList}} = 2534\mu s$
- $t_{\text{load}} = 80796\mu s$
- $t_{\text{edit}} = 137705\mu s$
- $t_{\text{store}} = 104757\mu s$

Thanks to this first analysis, we state the following:

XPS:

- $f(n) = \frac{20144+8270+1352}{20144+8270+1352+n\cdot(14588+9194+14538)} = \frac{29766}{29766+n\cdot 38320}$. We used 500 input images, hence the serial fraction becomes $f \approx 0.0016$.
- On the other hand, $sp(n_w) = \frac{19189766}{0.0016\cdot 19189766+\frac{0.9984\cdot 19189766}{n_w}}$

Xeon Phi:

- $f(n) = \frac{78213+55046+2534}{78213+55046+2534+n\cdot(80796+137705+104757)} = \frac{135793}{135793+n\cdot 323258}$. Following the same reasoning, $f \approx 0.0003$, where the amount of input images was 1499.
- $sp(n_w) = \frac{484699535}{0.0003\cdot 484699535+\frac{0.9997\cdot 484699535}{n_w}}$

This theoretical result is not reflected in the experiments and this is shown in Table 1 and in Table 2.

Par Deg	fastFlow	grPPi	parallelFor	C++ threads	theoretical
1	1.03568547448	0.67948418032	1.02902440195	1.02361104124	1.0000000000
2	1.77548235942	1.14509544453	1.50006322242	1.63585320138	1.996805111182
4	2.06846019453	1.4393966571	1.95938603075	2.04572360273	3.98089171974

Table 1: Speedup on the Dell machine, where the number of input images is 500.

Par Deg	fastFlow	grPPi	parallelFor	C++ threads	theoretical
1	0.99720483263	0.964023917365	1.00098333652	0.998464708246	1.00000000000
2	1.92192008524	1.90165312602	1.92433756106	1.92578129926	1.999400179946
4	3.77231008745	3.69871122975	3.77488863643	3.77400924137	3.996403237086
8	7.4103268408	7.0185402624	7.23174699878	7.16497525999	7.98323520606
16	14.5362887463	13.6881991578	13.7612858112	13.6829665846	15.92832254853
32	27.7252114247	26.670134673	27.2790065397	26.8769516286	31.70514217774
64	43.2734962888	39.4997673792	40.5659285372	39.1345148092	62.81283737363
128	49.6060852329	46.5048820127	47.0005252093	45.8571843541	123.30218668721
256	44.9867477006	43.2227264151	44.5667370661	43.5342040619	237.80771017185

Table 2: Speedup on the XeonPhi where the number of input images is 499

This behaviour is due to the fact that the disk operations (which require more than 50% of the computation time, see Figure 5) cannot be executed in parallel, since permanent memory does not allow such parallelism.

After noticing this drawback, we performed some more experiments on a modified problem, in which the images are loaded in memory and then processed and the results are displayed in Table 3 and in Table 4.

Par Deg	fastFlow	grPPi	parallelFor	C++ threads	theoretical
1	0.803712108462	0.710960510954	1.00278811572	1.09998362786	1.00000000000
2	1.37614163343	1.18029465036	1.94899337054	1.78847487338	1.99680511182
4	1.62784143079	1.44428470388	2.18310842504	2.13941302529	3.98089171974

Table 3: Speedup on the Dell machine in the case of execution "in memory" on 500 input images

Par Deg	fastFlow	grPPi	parallelFor	C++ threads	theoretical
1	0.903740796148	0.919217586071	0.945319267504	0.947095384737	1.00000000000
2	1.71164760399	1.81626666057	1.81620298564	1.81412371466	1.99940017994
4	3.32696674608	3.50313140901	3.61608047665	3.63192646437	3.99640323708
8	6.35215792483	6.96617218271	6.83464799687	7.24250090616	7.98323520606
16	11.8626995271	13.3991310687	13.4011918191	14.4211984205	15.92832254853
32	21.4833035807	26.6538763	26.7673867732	28.1892287433	31.70514217774
64	35.9642461546	48.6479495954	50.3796316136	53.4507198693	62.81283737363
128	41.2017248476	61.7874946026	60.5289785489	68.0129442588	123.30218668721
256	42.5016567555	74.12700936	72.6977380891	76.6040602177	237.80771017185

Table 4: Speedup on the Xeon Phi machine in the case of execution "in memory" on 500 input images

As much as scalability is concerned, we can observe that the "memory" version of the watermarking problem allows more efficient implementations, as shown in Figure 6.

Moreover, another kind of otimization was performed in order to better exploit the characteristics of the Xeon Phi machine. We compiled all the programs (except for grPPi) using the Intel compiler `icc`. The results are shown in Figure 7

4 Conclusions

We can reasonably assert that the watermarking problem as stated in the Introduction cannot scale because of its intrinsics characteristics. On the other hand, the modified version of the problem which overcomes the issue of sequential I/Os does allow good scalability and speedup.

The best algorithms are both the data parallel strategy implemented via C++ threads and the stream parallel strategy, obtained through grPPi.

Furthermore, we may observe that in terms of programmability C++ threads require fewer lines of code, hence they may be preferrable in a certain sense.

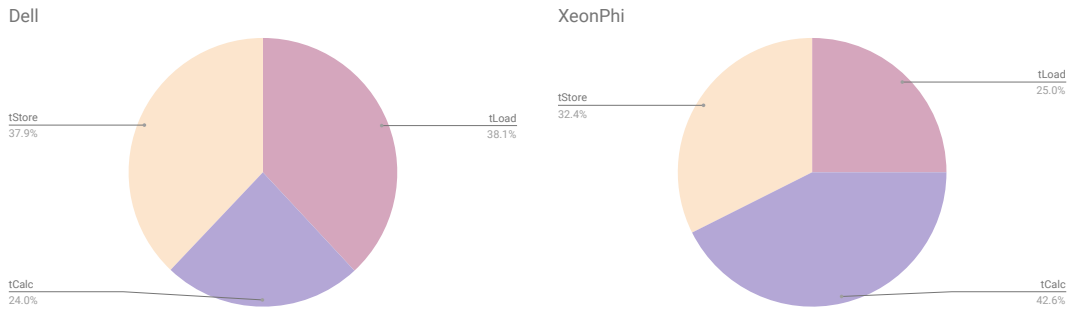

Figure 5: Percentage of time needed for loading an image, computing the watermark on it and storing the result

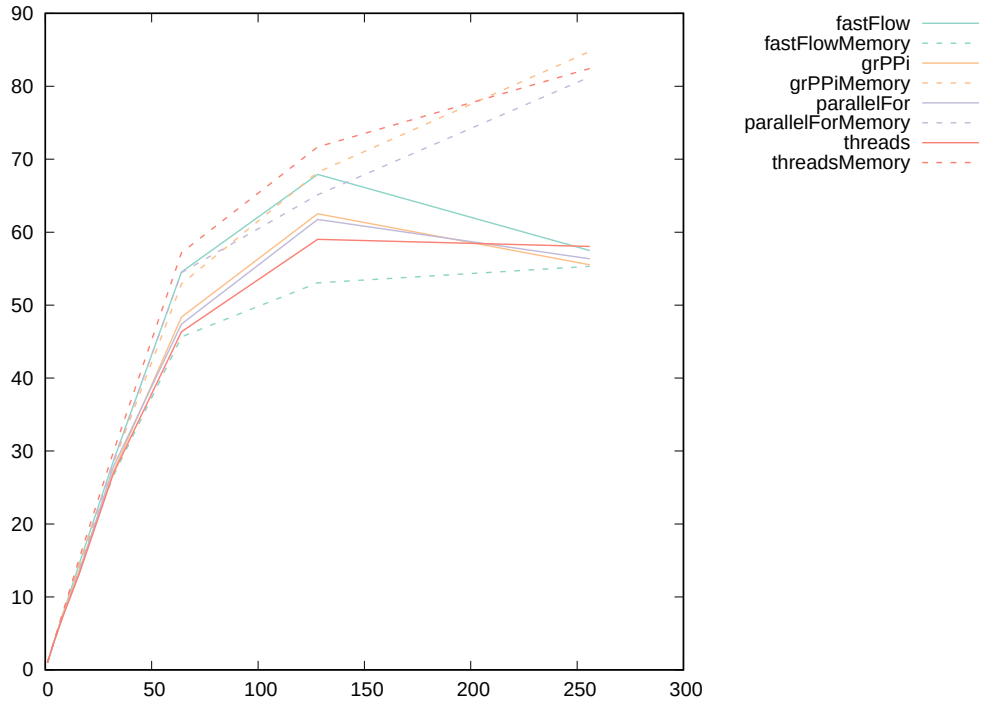

Figure 6: Scalability of all the implemented parallel strategies on the watermark random85 on the Xeon Phi machine.

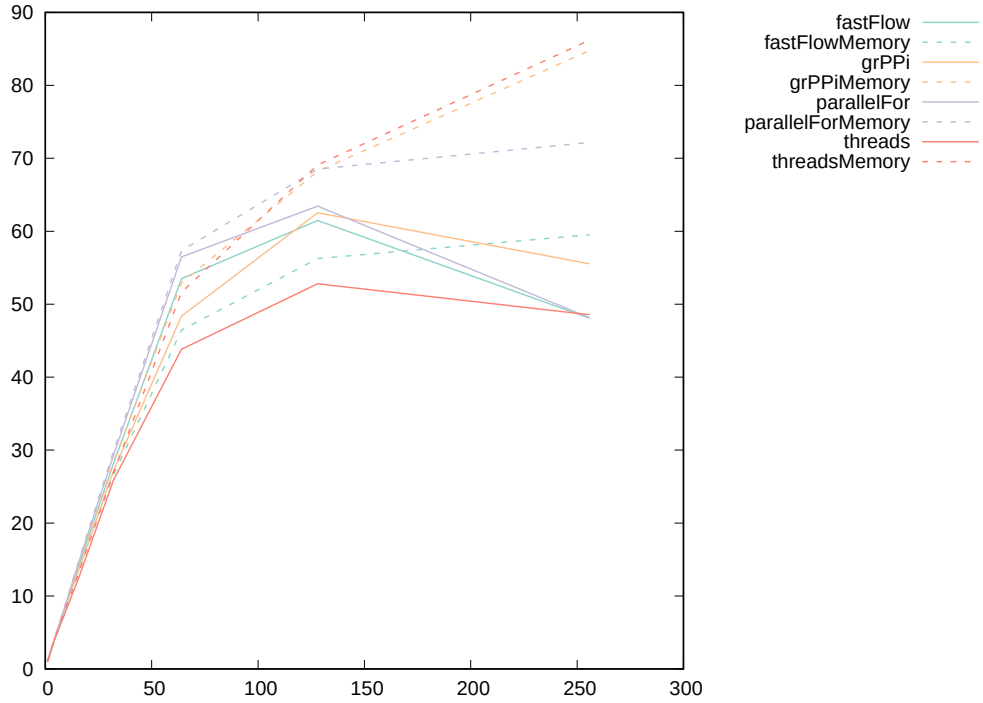

Figure 7: Scalability of all the implemented parallel strategies compiled using icc and run on the watermark random85 on the Xeon Phi machine.