Università di Pisa a.a. 2017-18

Project

Image watermarking

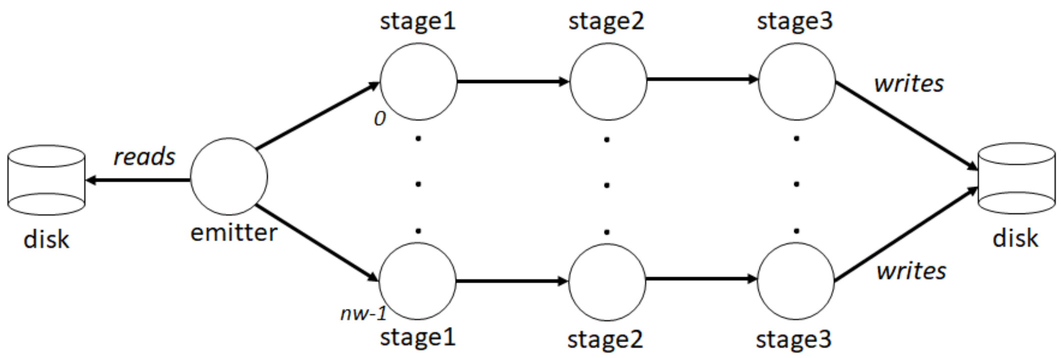
Giuliano Cornacchia - Mat: 502859

1 Introduction

Image watermarking consists in applying a mark to an image. It is typically used to identify the ownership of the copyright of the image.
In this project we consider only JPG images and the mark is represented by a JPG image that consists only of black and white pixels; the black-ones form the mark to apply.
The images used for testing the application are stored on the `Xeon PHI` in the folder
`smp18-cornacchia/Spm1718/SPM_Project/images/image_{small or big}`.
The folder `image_small` contains 300 images with a resolution of 1600×1200 pixels and `image_big` contains the same number of images with a resolution of 3200×2400 pixels.
The two "default" watermarks can be found in `../SPM_Project/wm_{small or big}.jpg`.
The folder `SPM_Project` contains also all the source code and a Makefile for compiling with a `make all` command.

2 Parallel architecture design

The main steps needed for applying the watermark to a single image consist in loading the image, applying the watermark and finally saving the new image in to the disk; so this analysis brings to a division of the main problem in three "sub problems" and the architecture that best fits this sub-division seemed to be a pipeline with three stages, respectively for loading, processing and saving the image.
From the moment that the application has to process many images the best choice is to add a *dispatcher* that generates the task for multiple pipelines, taking advantage of parallelism; in conclusion the chosen architecture is a *farm* of *pipeline*, moreover the *farm* doesn't have a *collector* because could represent a *bottleneck* for the application, in fact, the $stage_3$ writes directly on the disk.



More precisely the *emitter* extracts the paths of the images and pushes them as a *task* for the pipelines, in this way the loading of the image is performed in the first stage, reducing the serial fraction of the application.

3 Performance modeling

The architecture of the application as said is a *farm* of *pipeline*, hence the cost model $T_c watermark$ can be derived using the cost model of the two patterns, assuming that $T_c \approx t_s \cdot m$, where m is the number of items to process.
The completion time of a farm with nw workers can be computed as:

$$T_c farm(nw) \approx max\left\{(t_s emitter + t_s collector), \frac{t_s worker}{nw}\right\} \cdot m$$

The completion time of a pipeline with j stages can be computed as:

$$T_c pipeline \approx max\left\{ts_{stage_i}\right\} \cdot m \qquad with\ i \in [1, j]$$

And considering also the overhead due to the communication ($tcin$ and $tcout$) between stages we obtain

$$T_c pipeline \approx max\left\{(ts_{stage_i} + tcin_{stage_i} + tcout_{stage_i})\right\} \cdot m \qquad with\ i \in [1, j]$$

Combining them we can approximate the completion time of the application in the following way:

$$T_c watermark(nw) \approx max\left\{(t_s emitter + t_s collector), \frac{t_s pipeline}{nw}\right\} \cdot m$$

In the architecture there is no collector, so $t_s collector = 0$ and taking in account the time spent for the initialization (t_{init}) that takes in account the creation of threads and data structures:

$$T_c watermark(nw) \approx t_{init} + max\left\{t_s emitter, \frac{max\left\{(ts_{stage_i} + tcin_{stage_i} + tcout_{stage_i})\right\}}{nw}\right\} \cdot m$$

considering the architecture and the image watermarking problem we can state the following assumptions:

$t_{init} \approx (\#stages \cdot nw + 1)^{a} \cdot t_{creation}(thread) + ((\#stages - 1) \cdot nw + 1) \cdot t_{creation}(queue)$
$t_s emitter \approx t(extract\ path) + tcout_{emitter}$
$ts_{stage_1} \approx t(load\ image)$
$ts_{stage_2} \approx t(process\ image)$
$ts_{stage_3} \approx t(save\ image)$
$tcin_{stage_j} \approx t(pop\ queue_{j-1,j})$
$tcout_{stage_j} \approx t(push\ queue_{j,j+1})$

	`emitter`	`stage1`	`stage2`	`stage3`
t_{cin}	-	2140 ns	14.85 ms	6.76 ms
t_{cout}	378 ns	2291 ns	16.52 ms	-
ts_{small}	0.6 ms	160 ms	22 ms	321 ms
ts_{big}	0.6 ms	495 ms	144 ms	993 ms

Table 1: Time measured on the Xeon PHI.

[a] The application creates one thread for each stage of the nw pipelines, plus one thread for the emitter.

The overhead due to the creation of the threads and data structures deserved a deeper analysis. Is reasonable think that this overhead grows linearly with the parallel degree; this impression was confirmed by experimental measures (reported in the section experimental validation) on the `Xeon PHI`, at this point after performing a linear regression on each plot a simple, but reliable, mathematical model was obtained (the equation of a line): it can be used as an approximation of the overhead using the parallel degree as a parameter. The model is in the form $Toverhead(n) = n \cdot slope + q$, where the slope could be interpreted as the average time of the operation (e.g. creation of a single thread).

4 Implementation structure

4.1 Sequential C++

The sequential implementation of the image watermarking loads, process and saves one image at the time. The source code is in the file `../SPM_Project/IW_sequential.cpp`; the compilation can be performed through the Makefile with the command `make IW_sequential` and can be executed typing:
`./IW_parallel source_folder destination_folder watermark_image.jpg avg` [b]
`avg` can be 0 or 1 and indicates whether the watermark is computed with the average of the corresponding pixels (1) or simply overriding the pixel of the image with the black pixel of the mark (0).

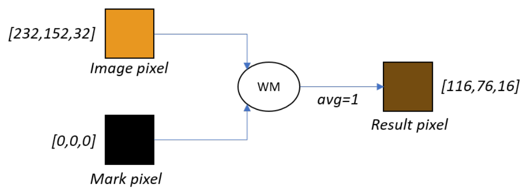


Figure 1: Watermark with avg=1

4.2 Parallel C++

The parallel implementation uses threads and shared data structures with the structure *farm* of *pipeline*. The shared data structures are basically queues, they are used to perform the communication and are summarized in the following table.

name	**elements**	**producer**	**consumer(s)**	**models**
`emitter_queue`	*string*	*emitter*	*all stages 1*	*image path*
`queue_stage12`	*(string, image)*	$stage_i 1$	$stage_i 2$	*image*
`queue_stage23`	*(string, image)*	$stage_i 2$	$stage_i 3$	*image watermarked*

This implementation can be found in the file `../SPM_Project/IW_parallel.cpp`; the compilation can be performed through the Makefile with the command `make IW_parallel` and can be executed typing:
`./IW_parallel source_folder destination_folder watermark_image.jpg avg parallel_degs`
`parallel_degs` is an integer parameter that specifies the parallel degree of the application (number of workers).

[b]In this and in all other cases the source/destination folder must be written **without** the final /, so for example is enough to write folder1/source or simply source if we are in the parent.

4.3 Fastflow

The implementation with the framework `fastflow` is in the file `../SPM_Project/IW_fastflow.cpp` that uses some functions implemented in the file `../SPM_Project/lib/IW_ff_elem.cpp`. The compilation can be performed in the same way as in the previous subsection using the command `make IW_fastflow` and can be executed typing:

```
./IW_fastflow source_folder destination_folder watermark_image.jpg avg parallel_degs
```

In `fastflow` *nodes* models parallel activities, consequently was defined a `ff_node` for the emitter and for each stage of the pipeline. Passing from the `c++` implementation to this one was very fast, in fact once created the structure of each `ff_node` is sufficient to use the same code of the `c++` application embedded in the `svc` method of each node with very small changes. Once created the nodes they can be combined through the *core patterns* (`ff_farm` and `ff_pipe`) for assembling the main architecture of the project.

4.4 Example of execution

Simple execution:

```
[user∼]$: ./IW_parallel image/image_small out wm_small.jpg 1 8
Parallel degree: 8
Processed 300 images in 19853 msec.
```

Test script:

In the main folder there is also a script `script_par.sh` made for executing in an automatized way several tests over the parallel implementation, the usage is the following:

```
[user∼]$: ./script_par.sh exe_name source dest watermark.jpg avg min max step n_exe
```

The semantic is: run `exe_name` with the parameters `source dest watermark.jpg avg` starting at the parallel degree `min` until `max` with a step of `step` and made `n_exe` executions for every parallel degree. The output will be saved in a text file named `log_test.txt`.

5 Experimental validation

The performance evaluation was performed on the `Xeon PHI`.

5.1 Basic measures and overhead

The first step consisted in evaluating the basic measures like the service time and the completion time of each activity. The assumption was that the service time of the stages was approximate by the loading time, processing time and saving time respectively for the first, second and third stage.
Furthermore also the overheads for the initialization of the application and for performing the communication were measured and all the values were taken after averaging several measures without taking in account outliers that could arise when the `Xeon PHI` is running several experiments at the same time. This tables recaps the measured service time:
The time spent for the initialization was measured by evaluating the creation time for a single a thread and for a queue; as expectable this time grows linearly with the parallel degree (fig. 2) and with a

	`emitter`	`stage1`	`stage2`	`stage3`
t_{cin}	-	$2140\ ns$	$14.85\ ms$	$6.76\ ms$
t_{cout}	$378\ ns$	$2291\ ns$	$16.52\ ms$	-
ts_{small}	$0.6\ ms$	$160\ ms$	$22\ ms$	$321\ ms$
ts_{big}	$0.6\ ms$	$495\ ms$	$144\ ms$	$993\ ms$

Table 2: Time measured on the Xeon PHI.

linear interpolation a mathematical model was obtained and used in the evaluation of the model given in the section performance modeling.
The creation time of a single thread is $\approx 0.26\ msec$ and of a single queue is $\approx 0.002\ msec$.

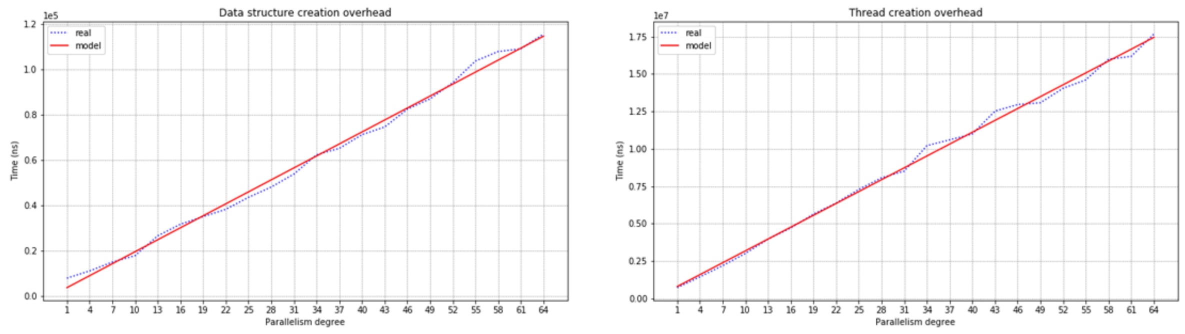
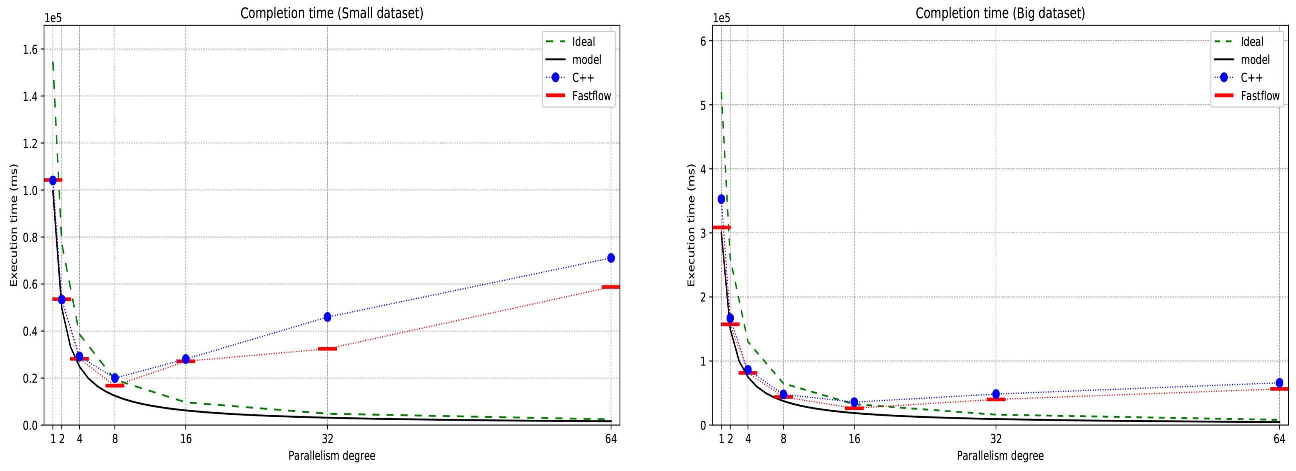


Figure 2: Overheads and fitting line.

The time spent due to the communication ($Tcin$, $Tcout$) between stages was taken simply measuring the time spent for performing a single *pop/push* from the respective queue, this time seemed not be affected by the parallel degree, so it's considered to be constant.

The completion time Tc was taken for all the implementations and for each dataset; each Tc is the result of an average of five measures, without outliers, for all the parallel degrees $p \in \{1, 2, 4, 8, 16, 32, 64\}$.



The ideal time was obtained with the formula $T_{ideal}(p) = \frac{T_{seq}}{p}$ and the model time refers to the model given before.
As we can see from the left-most plot in the figure, that is relative to the smaller dataset, the model approximates very well the completion time for the parallel degrees $p \leq 8$, for the $p > 8$ the completion

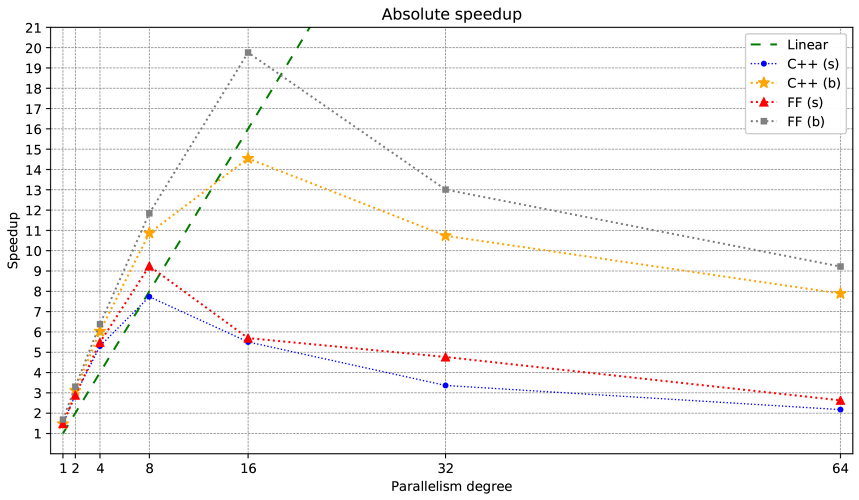
time starts to increase and consequently diverges from the model and from the ideal time too. This situation could be caused by the fact that the reads and the writes are executed sequentially on the disk and maybe increasing the number of writer/readers is critical for $p > 8$ mainly because some context switch can interleave the writes/reads causing several jumps in memory. Another cause can be the scheduling policy, like in the previous case can interleave the operations causing the degradation of the performance; certainly the time spent for the context switching is a factor for the total time growth, also the time spent for the synchronization and to guarantee the mutual exclusion contributes too. The same analysis is valid also for the larger dataset (right-most plot), but in this case the "critical" p is 16.

5.2 Derived measures

After collecting the basic measures, is possible to go deeper on the performance of the application evaluating: absolute speedup $sp_{abs}(p) = \frac{T_{seq}}{T_{par}(p)}$, relative speedup/scalability $sp_{rel}(p) = \frac{T_{par}(1)}{T_{par}(p)}$ and the efficiency $\epsilon(p) = \frac{T_{ideal}}{T_{par}(p)}$.

5.2.1 Absolute speedup

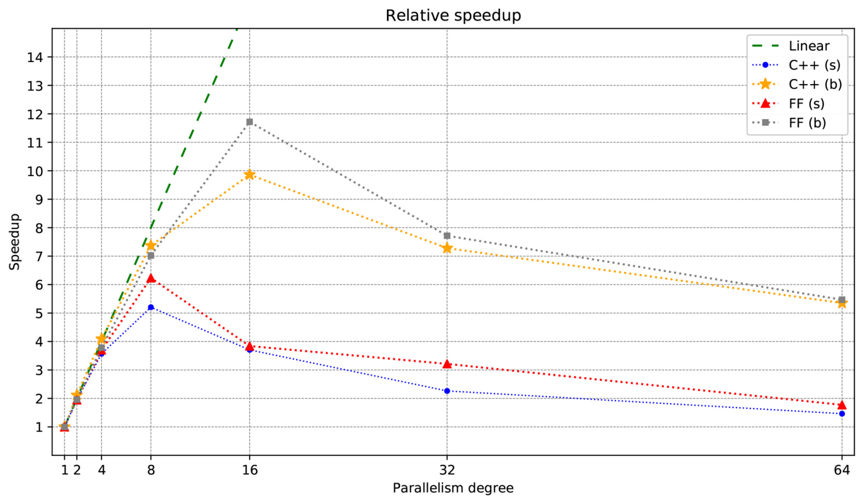
The absolute speedup gives an estimation of how "good" is the parallel implementation with respect to the sequential computation.
Looking at the plot we can notice how for $p < 8$ all the implementations have *super-linear* speedup, this can be caused by the fact that the sequential execution takes very long time; for $p = 16$ we have the maximum speedup (19.75) with the `fastflow` implementation applied over the larger dataset of images.



From the plot we can derive that the `fastflow` implementation is always better (in terms of speedup) with respect to the `c++` one applied over the same dataset. We can also notice that we can achieve a better speedup using bigger images, this can be explained with the *Gustafson-Barsis' law*, because seems that with the grow of the problem size (the size of the images) the serial portion grows slowly, for example, with $p = 32$ with the small images we can achieve a speedup of 4.76/3.34 for the `fastflow/c++` implementation while with bigger images we obtain a speedup of 13.02/10.73, about 4 times bigger. A further confirm came from the completion time plot, for $p = 32$ the two implementation have very similar Tc even if working on images four time bigger, we can make a program run in the same time with a larger workload.

5.2.2 Relative speedup

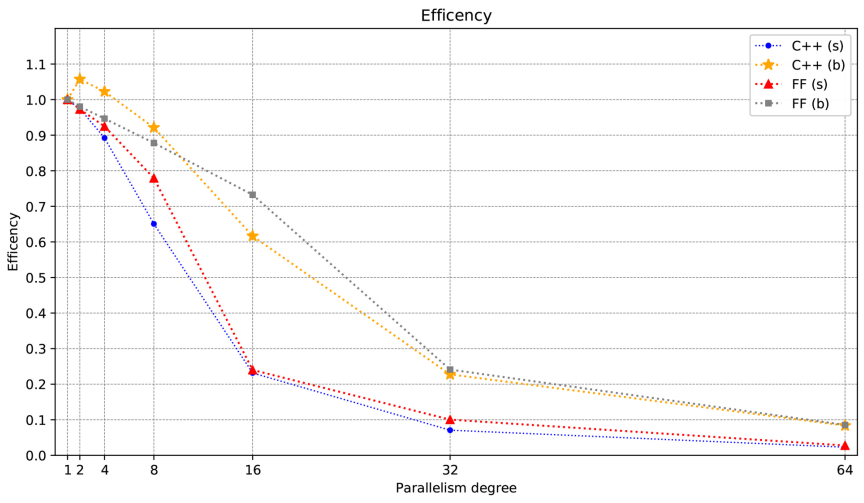
The relative speedup, also called scalability, eliminates the problem of the super-linear speedup caused by the sequential time over-estimation.



Only the `c++` implementation with the big dataset obtains super-linear speedup $(2.11,\ 4.09)$ for $p = 2, 4$. The previous consideration about the *Gustafson-Barsis' law* is still valid in this case.

5.2.3 Efficiency

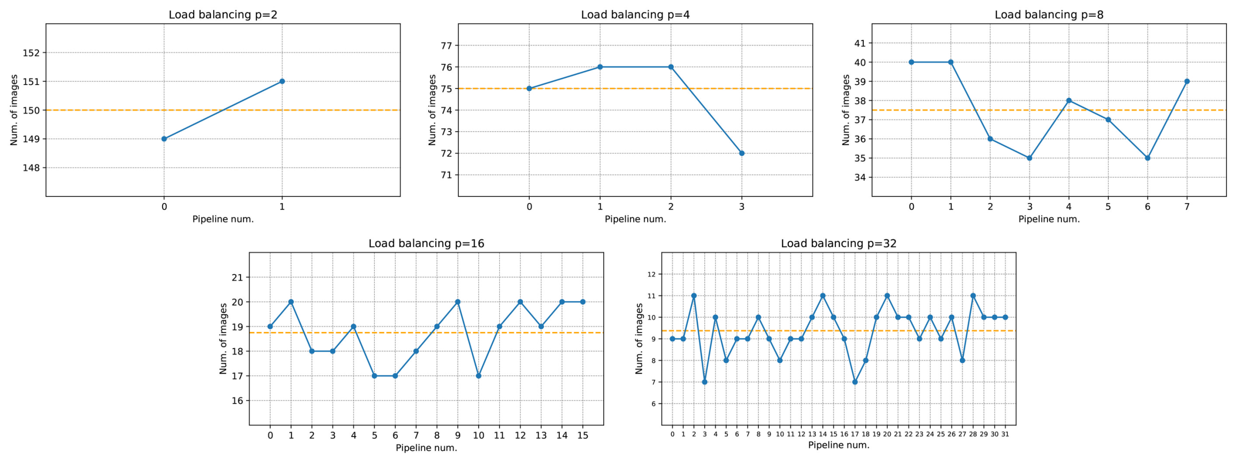
The efficiency says how "good" is the application's usage of the available hardware resources.



The `c++`'s efficiency applied over the big images exhibits an efficiency > 1, more precisely $1.05,\ 1.02$, for $p = 2, 4$; this can be explained with some cache optimization that occurred in the execution. Only for $p \leq 4$ all the implementations have efficiency ≥ 0.9, while for bigger parallel degree only the `c++` and `fastflow` with large images have an acceptable efficiency, while applied on the small images starting to became poor.

5.3 Load Balancing

The load balancing is fundamental to achieve good results, since we have to wait the termination of each worker, in the case of an un-balanced load some workers may be idle while some other working on many tasks. In the architecture the scheduling policy is very minimal (no overhead) but the optimal one considering the problem; in fact, the scheduling is *on demand* because every first stage pop from the emitter queue every time has performed the loading of the image. It is *optimal* because in our problem each worker compute the same "function" and consequently spend the same amount of time, furthermore the task have the same dimension. To be sure of avoiding this situations some empirical measures were taken.



The dotted line represents the ideal load of each worker, it is computed as $\frac{\#items}{\#workers}$.
The maximum affects the total computation, this means that this worker, representing the longest one, will compute max_{load} images instead of $ideal_{load}$, consequently the percentage of extra work is given by $\frac{max_{load}}{ideal_{load}} - 1$; for example, using the information for the image dataset (300 images) and referring at the figure for $p = 16$ we have $ideal_{load} = \frac{300}{16} \approx 19$ and $max_{load} = 20$ so using the formula the fraction of *extra work* is $\frac{20}{19} - 1 = 0.05$ then the slowest stage do only the 5% of *extra-work*. As we can notice from the figure the the number of images to process exceed at most of 2 images, we can state that the load is balanced.

6 Conclusions

The initial decomposition of the problem in three sub-problems led to the decision of a pipeline structure and for taking advantage of parallelism the initial structure evolved in a farm of pipeline. For the architecture was proposed a model for estimating the completion time, that takes in account overhead like the initialization of the data structures, creation of threads and communication; then were proposed two implementation in `C++` and `Fastflow` for the *farm of pipeline* structure. In the section experimental validation were performed the classical measure for the performance over the two implementations, executed on the `Xeon PHI`; other analysis were made, for example, for the load balancing.
In conclusion, which is the best execution? it depends of which parameter we are considering, the following tables (one for each dataset) shows the best implementation and parallel degree according to the different measures.

Small images	p	**implementation**	**value**
T_c	8	fastflow	16725 ms
sp_{abs}	8	fastflow	9.24
sp_{rel}	8	fastflow	6.23
ϵ	2	C++	0.974

Big images	p	**implementation**	**value**
T_c	16	fastflow	26322 ms
sp_{abs}	16	fastflow	19.75
sp_{rel}	16	fastflow	11.71
ϵ	2	C++	1.05