SPM project: parallel Image watermarking

Mario Leonardo Salinas, s.c. 502795

ml.salinas.23@gmail.com

a.y. 2018-19

I. Introduction

A collection of images (JPG) in a directory is processed to add to each image a BW "stamp" represented by a further image having only black and white pixels (no gray scale pixels). The input set of images and the stamp image have the same size, no need for alignment. The `Cimg` library should be used to read/write/process images. Students may consider a variant that substitutes a pixel correspond- ing to the black pixel of the "stamp" image with the average value of the original color, mapped to gray scale, and the black.

To compile all the source code a `Makefile` with a `make all` target is provided.

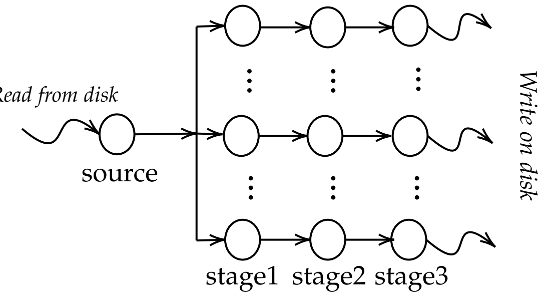
The images used to test the applications can be found onto the Xeon PHI in the folder `smp18-salinas/sync/project/imgs/source` (800x600) or `source_big` (1600x1200) and the result can be put in `result` or `result_big`.

The `imgs` folder also contains two watermark files.

II. Parallel Architecture Design

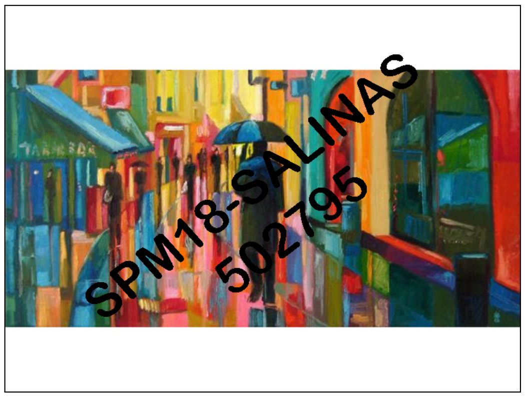
The real architecture design was preceded by an application analysis and decomposition into sub-problems. The main tasks the application has to execute are load, process and save an image. The parallel application should also provide an entity that generates the tasks.

Given these assumptions the parallel architecture that best fits the problem seemed a farm of pipelines; where the farm has no collector and one emitter, that generates the tasks, and each pipeline has three stages, one for each image sub-problem. The resulting design is shown below:

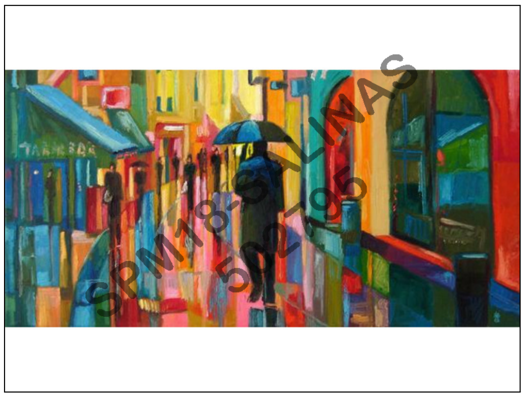


Notice that the different pipelines save *directly* on disk the watermarked images, hence there's no need of a collector. Finally this design also decouples[1] the reading of the path from the real image loading, shrinking the serial fraction of the application.

[1]Decouples in the sense that the emitter takes care of the paths extraction and pushes them as tasks, while the first stages pop a path from the queue and load the corresponding image in a `Cimg` object.



(a) Plain watermark



(b) Non-plain watermark

Figure 1: Examples of plain and non-plain watermark. If `average` parameter is set to *false* the application uses a plain watermark otherwise the non-plain one.

III. Sequential implementation

The sequential implementation of this application loads, processes and saves one image at the time until all the files in the source folder have been processed. The sequential watermark code is in the file `seq_watermark.cpp`, after the optimized compiling, to execute it type:

```
./seq_wm source_folder result_folder watermark.jpg average
```

where the `average` parameter is a boolean one and is true if the user want a non-plain watermark. The sequential application prints also the average time spent to load, process and save an image. These measures were used both to design the architecture and to understand where a bottleneck could rise in the pipeline.

IV. Parallel implementation

C++

The `C++` implementation uses `std::thread` and user-defined blocking queues.
The synchronization mechanism of the queues uses *mutexes* and *condition variables*.
The main data-structures are the task queues:

- the `task_queue` is single-producer-multiple-consumers `queue<string>` shared between the source and each stage1 of the pipelines.

  This queue contains the paths of the images to be watermarked.

- the `stage2_task_queue` is a vector of single-producer-single-consumer queues of `pair<string, Cimg>`. There's one queue for each pipeline and these queues are shared between the first two stages.

  In these queues we have pairs containing the loaded `Cimg` image and the corresponding path, that will be used by the third stage to save the resulting image.

- the `stage3_task_queue` has the same size and type of the `stage2_task_queue`. Each of these queues is shared between stage2 and stage3.

The size of the queues is equal to the chosen parallel degree, in this way each stage of each pipeline has its dedicated task queue.
The parallel entities involved in the computation are the stages of the pipeline, while the farm emitter sequentially reads a path from the source folder and pushes it in the `task_queue`.
The source it's also in charge of the termination. This is made by pushing one `EOS` for each pipeline. The propagation of the message is managed by the different stages.
All the business-logic code is in the file `pipeline.cpp`. The main method is in the file `cpp_watermark.cpp` and to execute it the user must type:

```
./cpp_wm source_folder result_folder watermark.jpg par_degree average
```

where all the parameters have the same meaning as before and `par_deg` is the number of workers (pipelines) the user wants to use.

Fastflow

The `Fastflow` code of the `ff_<node>` that encapsulate the business-logic of the parallel computations and the farm emitter is in the file `ff_farm.cpp`, while the main is in `ff_watermark.cpp`. The execution command is the same as the `C++` one.
The framework integration cost very little additional coding-time but, as described in the experimental validation section of this document, it resulted in not negligible performance improvements.
The code is simple: an `ff_Farm` is initialized with a vector of `ff_Pipe` as workers. The size of the vector is equal to the user-chosen parallelism degree.
After the farm instantiating, an emitter is assigned; this emitter is the one that sequentially reads the paths of the files and pushes them to the workers in a standard *Round-Robin* way[2].
At this point the computation is started.

V. Performance model

The application is built as a farm of pipelines with three stages each, hence the parallel *completion time* T_{Cp} can be estimated as follows:

$$T_{Cp} = \frac{t_{pipe} \times n}{p} + T_{sf}$$

where:

p is the parallelism degree.

t_{pipe} is the throughput of a generic pipeline. This metric is derived as the throughput of the slowest stage. After computing the average throughput of each stage t_{stage_i}, the third stage resulted to be the slowest one with a $t_{stage_3} \approx 110ms$ for the smaller images and $\approx 280ms$ for the bigger ones, from which it can be derived that, after an initial time interval, each pipeline will be able to deliver a watermarked image every $110ms$ or $280ms$ approximately.

n is the input size, namely the number of processed images.

[2]The decision of not applying any other load-balancing technique to the emitter follows from the assumption that all images are equal in size, thus tasks are granted to be balanced.

T_{sf} is the serial fraction of the total application computation.

This formula doesn't take explicitly into account the time that the generic pipeline needs to full all its stages, but this problem can be solved by using an $\geq$ instead of an equality.
Before discussing the experimental validation of the proposed model and architecture, in Table 1 are shown the *service time* of each pipeline's stage.

Table 1: Service time table, the main assumption is that the service time of each stage is well approximated by the time needed to load, process or save a generic image. Table 1 resulted from an average of five serial executions each over 600 800x600 (on the left) and 1600x1200 (right) JPG images. The last line of the table shows the increasing factor of the time.

$\mathbf{T_{load}}$ (ms)	$\mathbf{T_{proc}}$ (ms)	$\mathbf{T_{save}}$ (ms)
$77.26 - 158.25$	$5.76 - 23.45$	$109.81 - 280.28$
$x2.04$	**x4.06**	$x2.55$

This table justifies the choice of a pipeline-based architecture mainly because stage one and three have comparable times so it makes sense to decouple the two operations involved; moreover even if the second stage, whit smaller images, has a small service time, it has been maintained separated as well because of two reasons:

1. the images used to test the application didn't have such a size and definition to result in a computationally-hard watermark application, but just doubling the images dimension makes the service time of stage2 increasing twice faster than the other two (factor of 4 wtr 2); this could easily lead to a critical increase of T_{proc}, making the application scaling better on heavier images with the respect to an implementation where the first two stages are merged.
2. before deciding to use the design described in the first section, various architecture were tried; among them there was also a variant of the current one with stage one and two merged in one unique stage. In the end this version didn't result in any performance improvement.

VI. Experimental validation

The performance evaluation was executed on a Xeon PHI KNC with 64 physical cores and 254 logical contexts.
Multiple measuring on various metrics have been done:

1. average time needed for loading (T_{load}), processing (T_{proc}) and saving (T_{save}) an image. In this analysis we assume that these measures approximate well the service time respectively of the first, second and third stage of the pipeline. This was done in order to better understand where a bottleneck could rise in the pipeline's stages and to estimate the t_{pipe} defined in the previous section.
2. average T_C of the sequential and parallel implementation. These measures resulted from an average of 5 execution for each value of $p \in [1, 2, 4, 8, 16, 32]$.

   Furthermore the measured T_Cs allowed to derive other useful metrics:

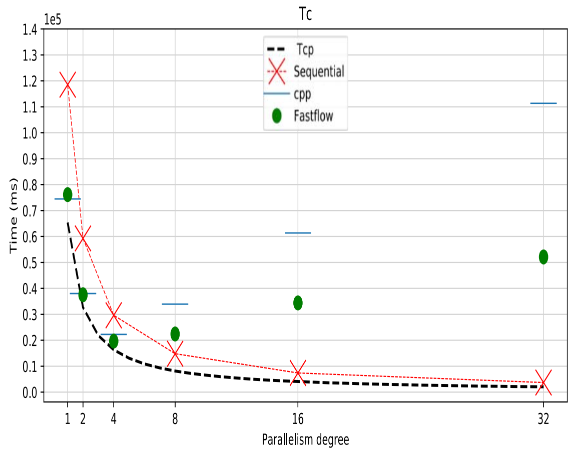
   (a) absolute and relative *speedup*, respectively $s_R(p)$ and $s_A(p)$, both of the `C++` implementation and the `Fastflow` one.

   (b) *efficiency* $\epsilon(p)$ again of both parallel implementation.

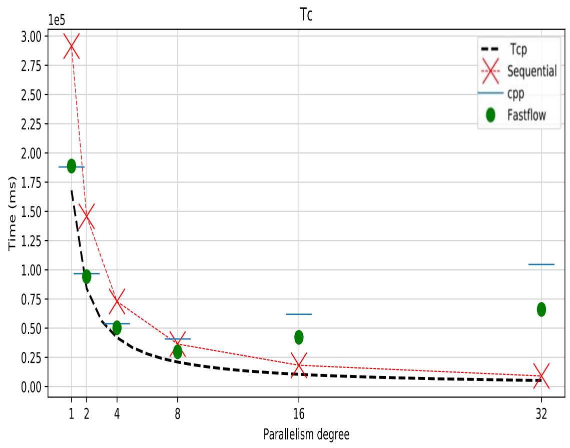
To have a better view over the ability to scale of the proposed applications, all of these measurements and computations have been performed on a workload of 600 images both 800x600 and 1600x1200 pixel large.

Performance comparison

The plots in Figure 2 show the comparison between measured T_Cs and the expected one T_{Cp} while increasing the parallelism degree.



(a) T_Cs on 800x600 JPG images



(b) T_Cs on 1600x1200 JPG images

Figure 2: Completion time plots, where: the sequential time T_{seq} related to the parallelism degree p was obtained dividing T_{seq} by p and T_{Cp} was computed setting $T_{sf} = 0$.

In the plots the dashed line represents the ideal *estimated* T_{Cp}.
Looking at figure 2a it can be said that both the parallel implementations behave well until $p \leq 4$, then, as p increases, performances starts to become poor and diverge from the ideal time.
This may happen because:

1. the overhead of the creation, scheduling and termination of threads is negligible until the number of pipelines involved in the computation is ≤ 4
2. if p threads try to write in parallel on the same disk, these writes will be serialized and probably executed in an interleaved way, causing the disk to jump from one sector to the other, wasting time.
3. the presence of too much queues, as blocking entities, can cause severe performance degradation. The figure shows in fact that the `Fastflow` version, that uses non-blocking synchronization techniques, reaches better performances than the `C++` one increasing the number of threads.

Figure 2b outlines a much better scenario. Mainly because the service time of the second stage is rapidly increased, the application scales much better and the critical parallelism degree becomes $p = 8$. In fact until that point both versions are close to the model.
It's also worth to mention that in this plot, even if with $p > 8$ performances start to diverge from the model, with bigger images the behaviour is more stable and produces smoother curves.

Both on 2a and 2b it's interesting to compare the sequential time, marked with crosses, with the parallel T_Cs: until p doesn't reach the critical degree of parallelism both parallel implementations outperform the serial threshold. This is because simply dividing the sequential time models more likely a master-worker architecture; while with the chosen architecture the proposed performance model works better and gives a stable and reliable lower bound.
From this initial analysis it can be concluded that when images are small the application behaves well until no more than 4 pipelines are used, while with bigger images is possible to mitigate the bottleneck in the third stage, leading to good performances until $p \leq 8$.
In both cases the application performances are well described by the proposed model until the threshold value for p is reached.

Speedup and efficiency analysis

A further analysis on $s(p)$ and $\epsilon(p)$ is needed in order to be sure that, even if T_C scales until $p \leq$ the measured thresholds, it is *convenient* to assign that amount of resources to the application. Speedup and efficiency were derived by the collected T_C with the classical formulas. Both relative and absolute speedup have been analyzed.

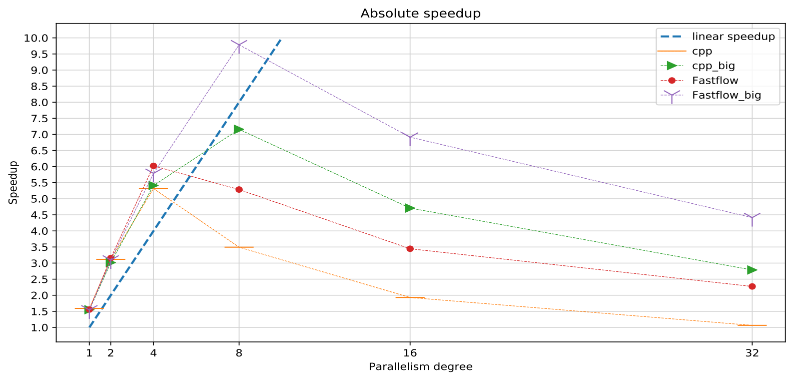


Figure 3: Absolute speedup plot, where $s_A(p)$ is computed as $\frac{T_{seq}}{T_p}$ and p is the parallelism degree.

Figure 3 shows that both parallel applications exhibit *super-linear* absolute speedup. This can be due to the fact that the absolute speedup doesn't model properly pipeline workers: using T_{seq} as numerator in the classical speedup formula, instead of T_1 - that is T_C of the application executed with one worker and core- will lead to an overestimate of the speedup. Hence the relative speedup should be considered more reliable in this scenario. Anyway, in this plot is already critical the effect of the Gustafson-Barsis' Law: since doubling the workload size the serial fraction of the application grows much slower than the parallelizable work, with $p = 8$ the application speedup increases; as a result the application runs up to ≈ 10 times faster than sequential version and twice faster than the execution on smaller images.
Figure 4 plots the relative speedup. Up to $p \leq 4$, all versions give *sub-linear* speedup, with `Fastflow` that stays closer to the linear speedup bound. Then, like for T_C, performances start to degrade, but the effect of the Gustafson-Barsis' Law is still evident, in particular in the `Fastflow` version executed on 1600x1200 images: it continues to exhibit a sub-linear and decent speedup, while all the other speedup curves are far below.
Finally, the efficiency plot in Figure 5 confirms that the `Fastflow` implementation outperforms the plain `C++` one.

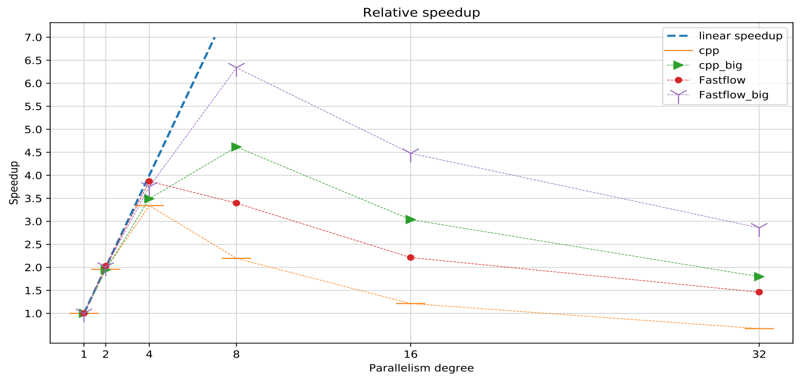


Figure 4: Relative speedup plot where $s_R(p)$ is computed as $\frac{T_1}{T_p}$ and p is the parallelism degree.

The critical parallelism degrees are still the same as before, but it can also be said that until that point the application exploits *almost-optimally* its resources and thus such a parallelism degree its not only the best possible for these implementations, but it's also used very well.
In particular, until $p \leq 4$:

> the `Fastflow` efficiency stays very close to 1, and when $p = 2$ the application exhibits an efficiency of 101.3%, meaning that probably some cache optimization happened or the disk was used in a more clever (and perhaps lucky) way.

> plain `C++` application also behaves well, but uses worse its resources than `Fastflow`.

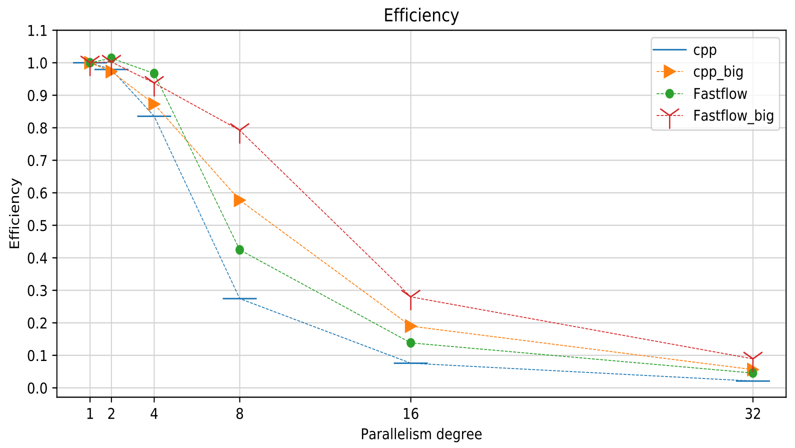


Figure 5: Efficiency plot where $\epsilon(p)$ is computed as $\frac{T_1}{pT_p}$ and p is the parallelism degree.

Instead, with $p = 8$ only the `Fastflow` application executed over bigger images exhibits good efficiency; and with $p > 8$ efficiency rapidly becomes poor.
In both speedup and efficiency plots there are also Amdahl's Law effects: speedups, if possible, are bounded, just as efficiency, by the inner serial fraction of the application and this, together with the explanations provided at the beginning of the previous section, justifies the ending shapes of the presented plots.

VII. Conclusions

The initial application analysis lead to the choice of a farm-pipeline architecture. This choice was confirmed also by Table 1: each pipeline on average will produce a watermarked image every $110 - 280ms$ instead of $\approx 200 - 461ms$ needed by a worker that loads, process and saves.
Two implementations of the same architecture have been proposed: one in `C++` that uses standard threads and blocking queues, and the other using the `Fastflow` framework.
Once the `C++` parallel implementation was ready, the `Fastflow` one resulted in minimum coding effort; in practice it was almost enough to plug the business-logic code of the parallel computations into `ff_nodes`.
The setup of the farm and pipelines required no more than five lines of code and both synchronization and termination are completely coder-transparent since handled by the framework itself. Finally it was discovered that both versions work with an almost-linear speedup and and efficiency of $\approx 98\%$ until the parallelism degree is $p \leq 4$, then both suffered minor and critical performance degradation, depending on the size of the image. In this scenario `Fastflow` always outperformed standard `C++` and increasing the size of the tasks resulted in a not negligible performance improvement.
Amdahl's and Gustafson-Barsis' Laws were also used to deepen the analysis of speedup and efficiency behavior and the `Fastflow_big` execution proved how the application is able to scale better increasing the size of the parallel workload.

	Big with $p = 8$	**Small with $p = 4$**
$\lim_{\mathbf{T_{sf}} \to 0} \mathbf{T_{Cp}}$	$35ms$	$27.25ms$
`Fastflow`	$49.64ms$	$32.8ms$
`C++`	$67ms$	$37.16ms$

Table 2: Latency table where is shown the time needed to complete a task of each version and execution compared with the ideal one obtained considering $T_{sf} = 0$. The Big and Small columns are related respectively to the execution that processes 800x600 and 1600x1200 images.

Furthermore, Table 2 shows how closely the application follows the proposed performance model until $p \leq 4$ for 800x600 JPG images and $p \leq 8$ for 1600x1200 ones, being able to produce one watermarked image every $\approx 32 - 49ms$, scaling very well with the respect to the $\approx 200 - 461ms$ needed by the sequential version.
In the end Figure 5 experimentally proves that this performance improvement is convenient, and with a little coding effort in order to integrate the proposed framework, even better results in terms of completion time, speedup and efficiency were reached.