SPM Project: Image "watermarking"

Davide Scano - Computer Science and Networking - 457926

a.y 2018-2019

1 Objective

A collection of images (JPG) in a directory is processed to add to each image a B&W "stamp"represented by a further image having only black and white pixels (no gray scale pixels) or/and substitutes a pixel corresponding to the black pixel of the "stamp" image with the average value of the original color, mapped to gray scale, and the black. The input set of images and the stamp image have the same size, no need for alignment. We can take advantage of parallel architecture of Xeon PHI KNC, with 64 physical cores and 254 logical context, for run the parallel program developed for perform image watermarking. The main problem that we have to deal with for realize this parallel application is the **memory wall**, because the off-chip memory rates have not grown as fast as on-chip computation rates and performances of many applications is fundamentally bounded by memory performance and not by computation. During the deployment of the project we can see how to mitigate this effect and how we can realize the parallel application.

2 Introduction

In the Xeon PHI there is a folder `smp18-scano/SPM_project_DS` where we can find all the files needed for running the program:

- *Makefile:* use to compile all the source code a with a make all target is provided.
- *Image folders:* one folder with image 800X600 an another with image 1600x1200.
- *Watermark files:* one watermark file for image 800X600 an another for image 1600x1200.
- *out folder:* where will be save the processed images.
- *lib folder:* contain all the functions utilized and a *queue* used in the *C++* and *FastFlow* version.

3 Parallel Architecture Design

Before to determine the parallel architecture we need to understand which are the problems to deal with.The sequential program that implement image watermarking can be schematize as a pipeline of three steps: *load*, *process* and *save*. Is easy understand that first and third step can be bottlenecks because they loads/saves images in/from the disk. *Table1* contain time detected running the sequential program on Xeon PHI, as we can see time for loading and saving images in memory is one order of magnitude bigger than processing time.

	Load (ms)	Process (ms)	Save (ms)	$T_{Cseq}(ms)$
`800x600`	85,259	5,299	124,226	43400,8
`1600x1200`	168,771	22,105	278,560	59527,8

Table 1: First detections of sequential version, over 200 images.

The time needed for reading all the file in the folder is sequential, so we can't improve is performance and saving time and loading time have the highest value, they are bottlenecks. We can parallelize those steps adding workers for achieve better performances. The parallel architecture propose is **Pipe**[*Farm(Load, nw_r)*, *Proces*, *Farm(Save, nw)*] show in *Figure 1.*

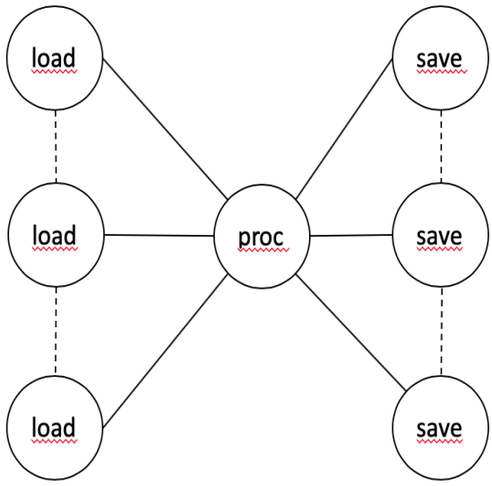

Figure 1: Parallel architecture proposal 1

In these architecture the number of workers used to parallelize loading and saving is not the same because as we can see in *Table 1* they have different performance, so we need to deal with this aspect. In the program implementation I use the ratio between the two time for understand which is the correct number of workers to assign at each steps when gradually increase parallelism degree. I have implement these parallel architecture in *C++* and *FastFlow*. We can suppose that in this parallelization the processing node at certain point become a bottleneck. A better architecture could be **Farm**[(Save(Proces(Load))), nw)], show in the figure:

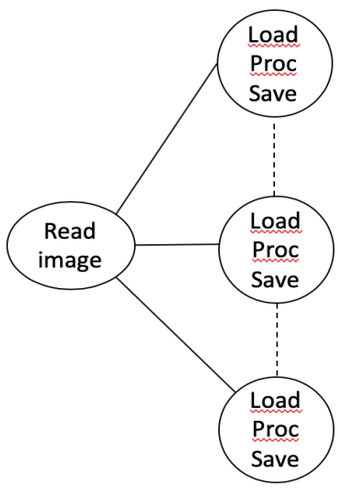

Figure 2: Parallel architecture proposal 2

I have implemented the parallelization in *Figure 2* only with *FastFlow* for understood if we can achieve better performance respect to my first proposal.

4 Programs Implementation

4.1 Sequential

The sequential implementation of this application load, process and save one image at a time until all the files in the source folder have been processed and stored in the folder named *out*, program executes all these actions on a single thread. The sequential code is in the file *sequential.cpp*, after the optimized compiling, to execute it type:

Figure 3: Outcome, in order: original, average and standard.

```
./run_watermarking image_800x600/ out watermark_800x600.jpg -seq -black 0
```

Writing *black/grey* we can activate watermarking with black or gray scale and the number at end of line code is the number of workers that user wants to use, in this case is forced to be 0. 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.

4.2 Parallel

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:

• `to_stage1` is multiple-producer-single-consumer queue <string,Cimg>shared between the stage 1 workers and stage 2 of the parallel architecture. In these queues we have pairs containing the loaded Cimg image and the corresponding path that as to be processed.

• `to_stage2` is single-producer-multiple-consumer queue <string,Cimg>shared between the stage2 and each stage3 workers of the parallel architecture. In these queues we have pairs containing the loaded Cimg image and the corresponding path already processed, that will be used by the workers to save the resulting image.

The parallel entities involved in the computation are the stages of the pipeline (saving and loading). Workers that perform the loading receive an interval that tell their which are the images that they must "load in Cimg". When the worker has loaded the image is push it in the task queue `to_stage1` and once he has "terminate" the images to be loaded he must push NULL in the task queue. The propagation of the messages is managed by the different stages. All the business-logic code is in the file *parallel.cpp* and to execute it the user must type:

```
./run_watermarking image_800x600/ out watermark_800x600.jpg -par -black 1
```

where all the parameters have the same meaning as before.

The *Fastflow* code of the `ff_<node>` that encapsulate the business-logic of the parallel computations and is in the file `ff_parallel.cpp`. The execution command is:

```
./run_watermarking image_800x600/ out watermark_800x600.jpg -ffpar -black 1
```

The framework integration cost very little additional coding-time but, as described in the experimental validation section of this document. We use a pipeline of two farm: the workers of the first `ff_Farm` take an interval containing the name of files that as to be "load in Cimg". These workers send a pair `std::pair<std::string*,cimg_library::CImg<int>*>` containing the loaded Cimg image and the corresponding path to source of the second `ff_Farm` that process the image applying the watermark and sending another pair `std::pair<std::string*,cimg_library::CImg<int>*>` to the workers for saving the images.

The parallel architecture proposed in *Figure 2* is in the file `ff_parallel_one.cpp`. The execution command is:

```
./run_watermarking image_800x600/ out watermark_800x600.jpg -ffparone -black 1
```

The code is simple, we use only one `ff_Farm` where emitter send to the workers a pair `std::pair<int,int>`, that telling at each workers which are the images that has to manages and each one make loading, processing ad saving.

5 Performance model

We start observing the sequential behaviour, that is essentially:

$$T_{Cseq} = T_{load} + T_{proc} + T_{save}$$

As we have seen in *Table 1*, saving and loading time are one order of magnitude bigger than processing time. In the case of the parallel implementation of architecture 1, completion time become:

$$T_{Cpar} \geq \frac{T_{load}}{nw_r} + T_{proc} + \frac{T_{save}}{nw} + T_{sf}$$

Where nw is the parallelism degree, $nw_r = nw \times ratio$ is the parallel degree properly adapt, as previously discussed. The T_{sf} is the serial fraction. This formula doesn't take explicitly into account the time that the pipeline needs to full all its stages, but this problem can be solved by using an $\geq$ instead of an equality.

nw_r	n_w	T_{load}	T_{proc}	T_{save}	T_{Cpar}	nw_r	n_w	T_{load}	T_{proc}	T_{save}	T_{Cpar}
1	1	26979	1456	34274	63264,2	1	1	44035	5172	68377	114814,2
1	2	27805	1731	17451	45763,8	1	2	44439	5920	34390,50	79056,2
2	4	14503,5	3732	9412	24309,2	2	4	24655,50	8410,00	17570,00	34220,2
5	8	7867,8	7024	4284,25	17679,4	5	8	11443,20	9302,00	9165,25	26586,4
10	16	7190,7	6506	2181,37	16716,2	10	16	10194,90	10131,00	5525,13	24951,2
19	32	8279,21	9326	1468,40	20611	19	32	10615,42	12798,00	3086,09	26846,8
39	64	6154,61	8574	535,53	16733,6	39	64	9734,74	11675,00	1184,30	27449,75

Table 2: *C++* Parallel version of Architecture 1 over 200 images 800x600 and 1600x1200.

As we can see in *Table 2* collecting the times: T_{load}, T_{proc}, T_{save} help us to understand what's going on in the various stages and how does they change increasing the size of the images and the parallelism degree. In the case of the parallel implementation of architecture 2, completion time become:

$$T_{Cpar} \approx max\{\frac{T_{worker}}{nw}, T_E\}$$

Where $T_E \approx T_{sf}$ in this case and $T_{worker} \approx T_{load} + T_{proc} + T_{save}$. Even i this case collecting T_{load}, T_{proc}, T_{save} help us to understand how they affect T_{worker} when gradually increasing the parallelism degree and size of images.

n_w	T_{load}	T_{proc}	T_{save}	T_{Cpar}	n_w	T_{load}	T_{proc}	T_{save}	T_{Cpar}
1	27596	1000	27929	57845	1	46332	4368	58729	112104
2	15325	548,5	14884	32169	2	25106	2308,5	30645	59558
4	9746,5	283,75	8392,5	19390	4	13926,25	1236,25	16111,75	32608
8	8502,37	633,37	6199,87	17156	8	9264,875	670	9195	20405
16	7459,18	1755,43	4199,75	17875	16	8608,43	885,31	6035,56	19241
32	7387,15	1679,03	3920,96	17533	32	10531,59	1287,81	5615,18	21551
64	4752,85	1245,14	9068,35	20660	64	10252,32	3535,04	6527,87	25265

Table 3: *FasFlow* Parallel version of Architecture 2 over 200 images 800x600 and 1600x1200.

6 Experimental validation

All the experiments have been done on the Xeon PHI announced before. All the measures resulted from an average of at least 5 execution for each $n \in [1, 2, 4, 8, 16, 32, 64]$. Multiple measures on various metrics have been done:

- average T_C of parallel implementation.
- time needed for T_{load}, T_{proc}, T_{save}.
- speedup $sp(n) = \frac{T_{seq}}{T_{par(n)}}$
- efficiency $\varepsilon(n) = \frac{sp(n)}{n}$
- scalability $scalab(n) = \frac{T_{par(1)}}{T_{par(n)}}$.

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 200 images both 800x600 and 1600x1200.

6.1 Performance comparison

The plots in *Figure 4* and *Figure 5* show the comparisons between the two parallel structure while increasing the parallelism degree and changing the images size.

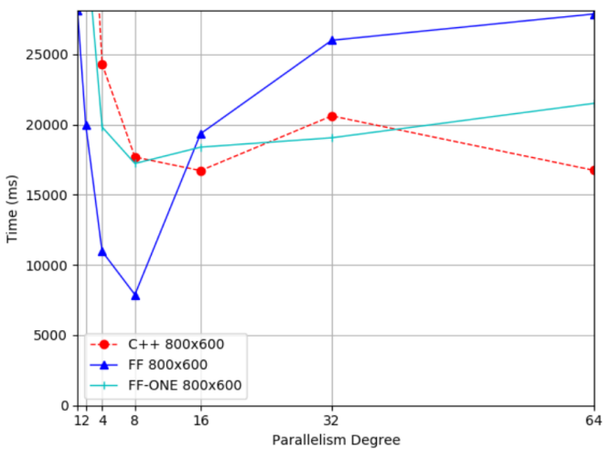

Figure 4: T_C images 800x600

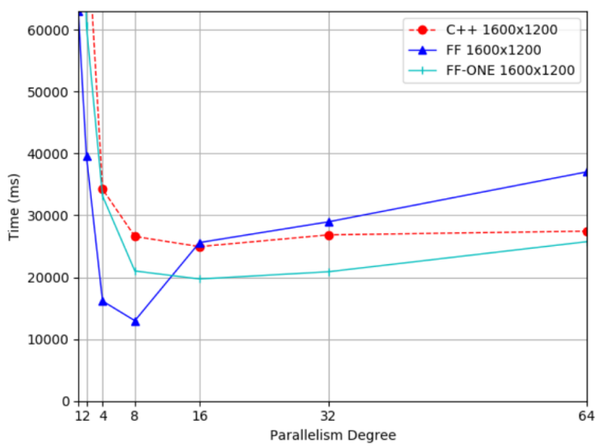

Figure 5: T_C images 1600x1200

Looking at *Figure 4* it can be said that both the *FasFlow* implementation of parallel structure implementations behave well until n$\leq$ 8, while the *C++* implementation of architecture 1 reach n$\leq$ 16. In *Figure 5* the *FasFlow* of Architecture 1 behave well until n$\leq$8 and in the other hand *C++* and *FasFlow* of Architecture 2 reach n$\leq$16. The architecture 1 ,as n increases, performances starts to become poor because the second stage become a bottleneck, in both *FasFlow* graph the steep slope is due to the fact that the second stage became bottleneck and strongly decreasing is performance. In *C++* the same thing happens but the time for processing increase more slowly. Performance degradation, in both architecture, is due to the overhead of creation, scheduling and termination of threads and also to the communication mechanisms (especially for architecture 1) and even more important some architecture components, in the machine, are access sequentially for example: if n threads try to write in parallel on the same disk, these writes will be serialized and probably executed in an interleaved way.

6.2 Speedup, Efficiency and Scalability Analysis

All the metrics are derived in the same way as described in Section 6, the analyzed speedup is the absolute one (relative to T_{Cseq}).

The speedup of *FasFlow*, referred to architecture 1, is slightly bigger to the ideal speedup until n = 4, program exhibit *superlinear speedup*, and still decrease until n=8. After that value the effect of huge grow of the time in the second stage (it became a bottleneck) makes fast deterioration of speedup performance. In this architecture is critical the effect of the Gustafson-Barsis' Law: since

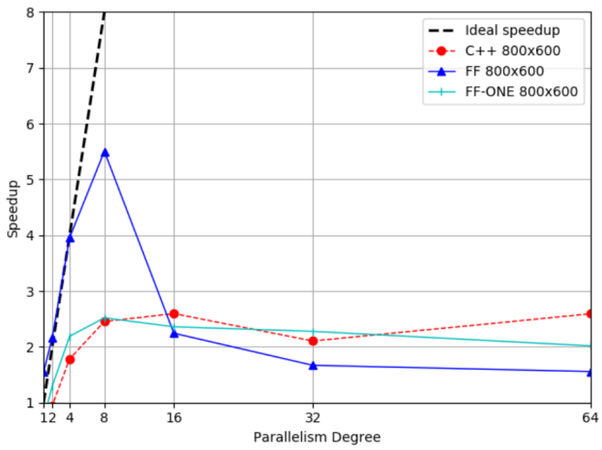

Figure 6: $sp(n)$ images 800x600

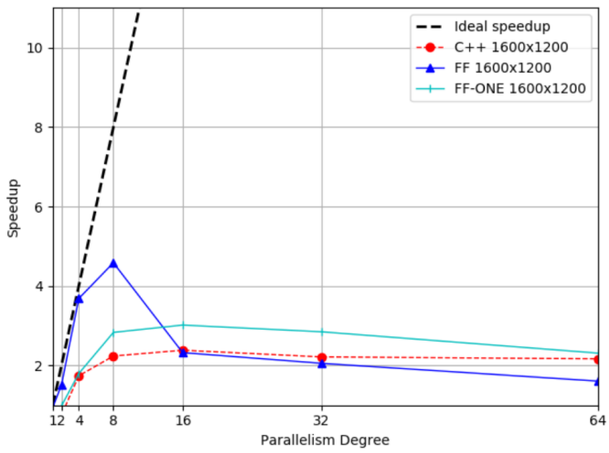

Figure 7: $sp(n)$ images 1600x1200

doubling the workload size the serial fraction of the application grows faster than the parallelizable work, so we have a reduction of the speedup among the *FasFlow* graph of 800x600 and 1600x1200. Architecture 2 is affect by the fact that some components are access sequentially but is more smooth more slowly.

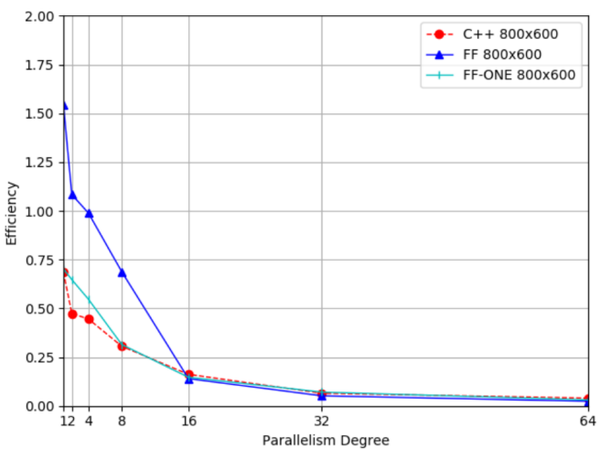

Figure 8: $\varepsilon(n)$ images 800x600

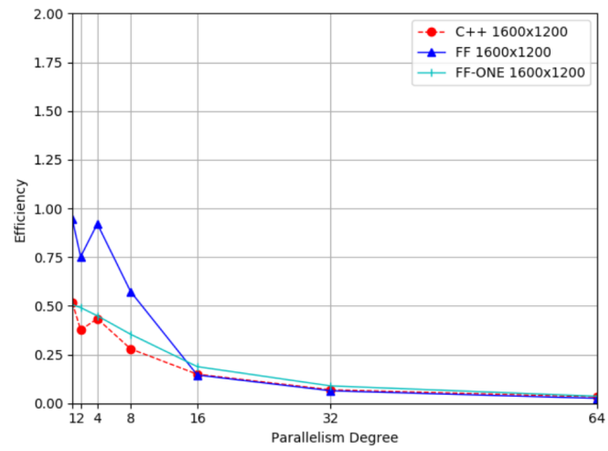

Figure 9: $\varepsilon(n)$ images 1600x1200

In *Figure 8* the *FasFlow* of architecture 1 achive ε>1, we know from the theory that when a program reach this value, because exhibit *superlinear speedup*, is due to the fact that the program's performance is strongly dependent on having a sufficient amount of cache memory, and no single worker has access to that amount. Increasing the size of images we lost the *superlinear speedup*, this is due to the fact that all the images fill into the cache. Gradually increasing the parallel degree the efficiency start deteriorating, this is due to overheads present in the parallel implementation (threads and communication mechanisms).

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, that achieve a shorter bound for the pipeline respect to the farm for speedup and efficiency.

7 Conclusion

The initial application analysis lead to the **Pipe**[*Farm(Load, nw$_r$)*, *Proces*, *Farm(Save, nw)*] and **Farm**[*(Save(Proces(Load))), nw)*]. This two architecture proposal are choice for parallelize the sequential program for image watermarking, that show important difference in terms of time spent between the load/save and the process action, as we can see in *Table 1*. Two parallel implementation of the architecture 1 have been proposed: one in *C++* that uses standard threads and queues and the other one using the*FasFlow* framework, while for architecture 2 have been proposed only the *FasFlow*. Both architecture shows through different metrics and representation that the program behaves well in case of small images, while the performances start deteriorating soon for the biggest. This is due to the fact that in architecture 1, gradually increasing the number of workers and the overheads related to communication between stages increase rapidly. In the

other hand architecture 2 suffer to the fact that we have cache shared among the threads, and increasing the parallelism degree we have problem because at certain point we access sequentially at the memory. If we change the machine where run the programs probably the architecture 1 and 2 may be achieve better or even worst performance. In conclusion we can saw that architecture 2 show a better completion time increasing the size of images, and smooth curves respect to a architecture 1 that achieve a huge degradation of performance due to the fact of stage 2 in the pipeline become a bottleneck, so it's preferable use architecture 2 for parallelize this kind of problem.