SPM Report: Histogram Thresholding

Student: Andrea Lisi

1 Program's Logic

The program computing the B/W image is structured, in big steps, in this order:

1. Load the image from the disk, `Load`;
2. Build the histogram, function `h`;
3. Compute the filter, function `f`;
4. Save the image into the disk, `Save`.

2 Designing the Parallel Application

Apart from stages `Load` and `Store`, which rely on the disk, this application can be nicely supported by parallel computation. Both stages `h` and `f` access to an image and this can be done in parallel, once a good decomposition strategy is defined. Moreover, since all the images received are independent, other techniques are available (maintaining the order of the tasks is not required). This application suits well to exploit both **Data** and **Stream** parallel patterns. We can implement them with a combination of a Master/Worker together with a Loop Parallelism pattern.
These patterns give flexibility to the application, since we can vary the number of executors involved in each pattern in order to find a combination which suits well for our needs.

2.1 Skeleton Composition Analysis

We now exploit the mentioned application design patterns to build an abstract parallel skeleton composition for this application. We have different possibilities to parallelize the application. Let's explore them.

A stage describing the `Load` portion of the program can be used to generate the stream, as well a stage describing the `Store` portion can represent the final stream collector of the results. `Load` and `Store` cannot be parallelized since they rely on the disk, and therefore the core of the parallelization will be represented by `h` and `f`. So now we evaluate how to compose these two functions.

We start by having a sequential composition `Comp(h, f)`. We can now introduce either a farm or a pipeline. Introducing a farm we get `Farm(Comp(h, f))` and this is a first possible skeleton.
Introducing a pipeline, instead, we would have ended up with `Pipe(Seq(h), Seq(f))`. At this point we can introduce a farm as well and obtain finally `Farm(Pipe(Seq(h), Seq(f)))`: this is a second possibility. These two compositions only support stream parallelism.

The same reasoning can be done starting with a `Comp(Map(h), Map(f))` in order to exploit data parallelism. We may use this one as skeleton to try to boost as much as we can the speed to compute our functions on a single stream element. Anyway, it is possible to combine stream and data parallelism patterns, for example by introducing either a farm or a pipeline, or even both.

3 Performance Model

The overheads of each one of the program's portions are computed calling the high_resolution_clock now() function provided by the C++ chrono library, with millisecond precision.

3.1 Sequential Measurements

Three sample inputs are used to test the overhead of the application and its portions. The time spent by each portion of the program is reported in table 1 in *ms* precision. The target architecture is the **Ninja** machine.

$ImageSize$	$Load$	H	F	$Store$
512×512	8,40	4,39	13,59	52,74
1428×968	42,65	37,59	73,48	273,81
4252×2853	336,4	335,2	672	2384,4

Table 1: Sequential measurements, in ms

We can define the (expected) **Completion Time** of the application, on a stream of n elements, as follows:

$$((Load + Store) + (H + F)) \times n$$

where the first inner term corresponds to the **Serial Fraction** and the second to the **Parallel Fraction**.

Note: *since the Serial Fraction relies on the disk, we should pay this term for each element of the stream. This would be the real bottleneck of the application. For the purposes of the exam the application will load the image once at the very beginning and store it at the very end. Therefore the time expected for a stream of n elements changes to:*

$$(Load + Store) + (H + F) \times n$$

Table 2 shows the time spent, in average, by the sequential application computed on a stream of 300 elements of different sizes:

$StreamLength$	512×512	1428×968	4252×2853
300	5183,5	33601.5	$318147 \sim 5min$

Table 2: Average completion time, in ms

3.2 Resources Evaluation

We now perform a more concrete analysis by using the sequential measurements shown in section 3.1. In table 3 are listed the possible skeleton compositions[1] introduced in section 2.1 and the **optimal** number of workers needed by the compositions to reach service time roughly equal to the **Inter-Arrival Time** T_A, i.e. the service time of the stream generator, in order to avoid both waste of resources (excessive parallelism) and bottle-necks.

[1]SC stands for Skeleton Composition

The image used to compute T_A is the biggest one, in order to have finer granularity. Since the application is implemented by using both C++ threads and FastFlow framework, we got two slightly different values of T_A: *29 ms* with FastFlow and *23 ms* with C++ threads. The results shown in table 3 make use of the FastFlow value. Once we notice the best (theoretical) models, we can just compute the optimal number of workers using the C++ thread's T_A only on these models.
Remark: refer to table 1 lists for the sequential times.
We have the following groups of parallel executors:

- *nwFarm* represents the farm's workers;
- *nwH* represents eventual data parallel on `h`;
- *nwF* represents eventual data parallel on `f`;
- *nwTot* represents the total parallel executors involved, adding also the stream generator and collector;

	Composition	$nwFarm$	nwH	nwF	$nwTot$
SC 1	Farm(Comp(h, f))	35	-	-	37
SC 2	Farm(Pipe(h, f))	24	-	-	50
SC 3	Comp(Map(h), Map(f))	-	24	48	50
SC 4	Pipe(Map(h), Map(f)))	-	12	24	38
SC 5	Farm(Comp(Map(h), Map(f))))	24	1	2	50
SC 6	Farm(Pipe(Map(h), Map(f))))	12	1	2	38

Table 3: Summary of the compositions analyzed

The number of workers *nwFarm* of **SC 1** is computed as follows:

$$\begin{gathered} T_S(Comp(h,f)) = T_S(h) + T_S(f) = 1007 \rightarrow \\ 29nwFarm = 1007 \rightarrow nwFarm = \lceil 1007/29 \rceil \rightarrow \\ nwFarm = 35 \rightarrow \\ nwTot = 35 + 2 = 37 \end{gathered}$$

The computation of the number of workers regarding the other skeletons follows more or less the same procedure.

Let's look at **SC 4** and **SC 3** which are the only ones which don't use a farm. The aim of these compositions is to minimize the latency, and therefore try to reduce the service time with a "waterfall effect". Since a pipeline works better if the service times of the stages are equal, and since we have that $T_s(f) \sim 2T_s(h)$, we assign to the stage computing `f` twice the resources of the stage computing `h`. In order to simplify computations, we use this as a rule of thumbs for the other cases. Therefore, the number of workers are computed as follows:

$$\begin{gathered} T_S(P(Map(h,x), Map(f,2x))) = max(T_S(h)/x, T_S(f)/2x) = 336/x \rightarrow \\ 29 = 336/x \rightarrow x = \lceil 336/29 \rceil \rightarrow \\ nwH = x = 12 \rightarrow nwF = 24 \rightarrow \\ nwTot = 2 + 12 + 24 = 38 \end{gathered}$$

and

$$\begin{gathered} T_S(C(Map(h,x), Map(f,2x))) = T_S(h)/x + T_S(f)/2x = 671/x \rightarrow \\ 29 = 671/x \rightarrow x = \lceil 671/29 \rceil \rightarrow \\ nwH = x = 24 \rightarrow nwF = 48 \rightarrow \\ nwTot = 2 + max(24, 48) = 50 \end{gathered}$$

Since in the pipeline the two stages go in parallel, we take the max of the service time of the two, in order to compute the service time of the pipeline, but their parallel executors exist all at the same time, and hence the number of workers is additive. For the sequential composition, instead, holds the opposite. From the computation we see that the latter case looks to be a worst model than the former.
Moreover, since SC 4 adjust both pipeline stages at the same speed (thus deleting the bottle-neck at stage `f`), the farm needs fewer number of workers than **SC 2**: thus we omit the computation of $nwTot$ of SC 2 and we can consider it discarded.

Finally we have **SC 5** and **6**. Both try to combine stream and data parallelism. Since we have many combination of values, we start by using $nwH = 1$ and $nwF = 2$. In this case the farm workers are computed as follows:

$$T_S(F(C(Map(h,1), Map(f,2)))) = T_S(h) + T_S(f)/2 = 671 \rightarrow$$
$$29nwFarm = 671 \rightarrow nwFarm = \lceil 671/29 \rceil \rightarrow$$
$$nwFarm = 24 \rightarrow$$
$$nwTot = 2 + 24 \times max(1,2) = 50$$

Increasing the value of nwH, and as consequence the value of nwF, we end up with the same $nwTot$ (ex with $nwH = 2$ and $nwF = 4$ we get $nwFarm = 12$ and therefore $nwTot = 50$).
SC 6 adds a pipeline, modifying the computation of $nwFarm$ in this way:

$$T_S(P(C(Map(h,1), Map(f,2)))) = max(T_S(h), T_S(f)/2) = 336 \rightarrow$$
$$29nwFarm = 336 \rightarrow nwFarm = \lceil 336/29 \rceil \rightarrow$$
$$nwFarm = 12 \rightarrow$$
$$nwTot = 2 + 12 \times (1 + 2) = 38$$

At the end of the analysis, we have that compositions **1, 4, 6** use, more or less, the minimum amount of resources in order to reduce both thread idling and bottleneck.

4 Implementation Snapshot

Both implementations, FastFlow and C++ Threads, try to follow exactly the same steps.
Two concurrent stages implement `Load` and `Store` as stream Generator/Collector. Other concurrent stages implement the `h` and `f` functions. If the program exploits data parallelism, the histogram is computed partially by each parallel executor and then a final merge is performed: doing so avoids expensive locks on the histogram structure. Differently, each parallel executor used to filter the image has assigned its own portion of image and therefore there are no write conflicts.
Regardless the composition used, the main program builds the application with a master/slave structure of Generator - Composition - Collector where Generator and Collector substitute the standard emitter and collector. Composition is built according to certain input parameters (see section 6).

5 Performance Indicators

Some performance indicators are gathered following the following setup: *a stream of 300 elements of the the biggest image*. Each skeleton composition is evaluated more times, each time increasing the number of workers assigned from a minimum of 2 to a max of 240. The optimal numbers listed in table 3 are used as well.
The performance indicators evaluated are: **Completion Time**, **Efficiency** and **Speedup**. The speedup plot also includes the **Ideal Speedup**.
Figures 1 and 2 shows how our performance indicators vary as the parallelism degree increase.

Table 4 lists the (theoretical) optimal number of workers needed by each composition, of both implementations, in order to reach a service time almost equivalent to the Inter-Arrival Time.
Note: the number of workers in the plots don't include the Generator and Collector.

Impl	*Composition*	*Acronym*	T_A	$nwFarm$	nwH	nwF	$nwTot$
Th	F(C(h, f))	sf	23 ms	44	-	-	46
Th	P(Map(h), Map(f)))	dp	23 ms	-	15	30	47
Th	F(P(Map(h), Map(f))))	pf	23 ms	15	1	2	47
FF	F(C(h, f))	sf	29 ms	35	-	-	37
FF	P(Map(h), Map(f)))	dp	29 ms	-	12	24	38
FF	F(P(Map(h), Map(f))))	pf	29 ms	12	1	2	38

Table 4: Summary of the compositions involved

5.1 Results

As shown in the legend, for brevity, the compositions will follow this naming convention: **sf** stands for farm of sequentials; **pf** stands for pipeline-farm and, finally, **dp** stands for data parallel, that is, pipeline of map.[2]

FastFlow

Each composition performs almost equivalently if we look at the Completion Time, with *pf* be the one speeding more up until it uses to 32 workers. *pf* reaches the highest speedup with 33 workers; the other two compositions reach the highest speedup with 128 workers, but not much higher than using 64 or 32: indeed, the efficiency at 128 is quite low. Composition *pf* with 32 workers is still competing with the ideal speedup, while *sf* and *dp* compete only up to 8-16 workers. A final consideration, the compositions exploiting data parallelism reach a super-linear speedup when few workers are involved, but they don't reach the minimal completion time with those workers.

C++ Threads

At the first glimpse we notice that the *dp* composition starts losing speed as soon it uses around 16 workers: this is caused by a constant spawn of a large number of threads. Instead, the other two compositions compete almost equivalently, even if *sf* looks to perform a bit better when both use around 32 workers. Composition *sf* competes with the ideal speedup up to 16 workers meanwhile *pf* starts losing with those workers; both of them reach the max speedup with 32-33 workers. Unlike FastFlow, no composition reach super-linear speedup, even though *sf* has efficiency almost 1 with 2 and 4 workers.

Conclusions

All the 3 compositions analyzed behave almost identically when few workers are used. **FastFlow** looks to have a way better performance when data parallel is involved, while the **C++ Threads** look to have a better farm implementation. Both implementations stabilize their maximum speedup at roughly 20.
The speedup peak is reached by FastFlow's *pf* composition when it uses 33 workers and thus making it the fastest but not the more efficient (with those workers).

[2]The acronym dp, which should represents full data parallel, is not properly correct since it's combined with a pipeline, which is a stream pattern.

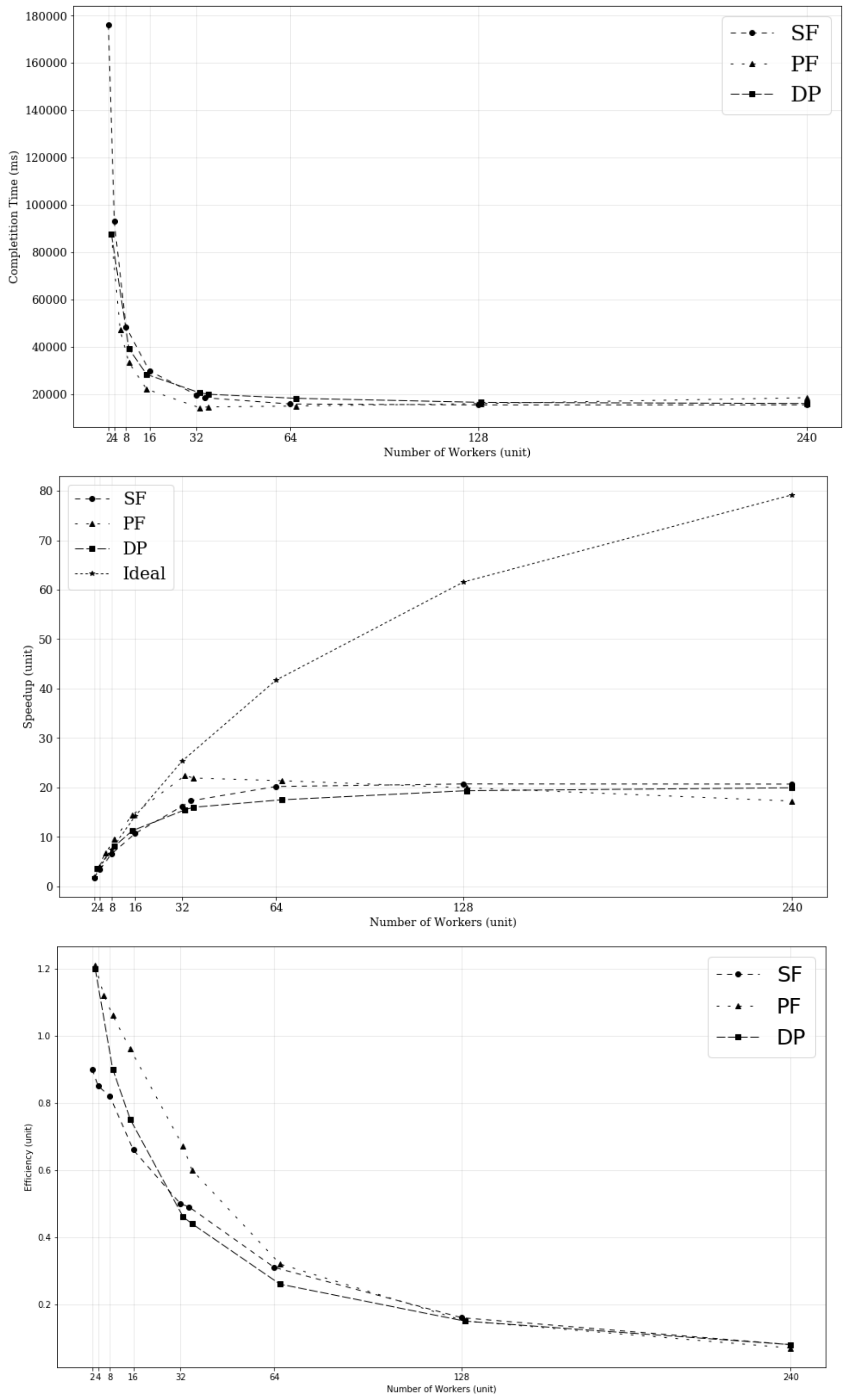

Figure 1: Completion time, speedup and efficiency with **FastFlow**

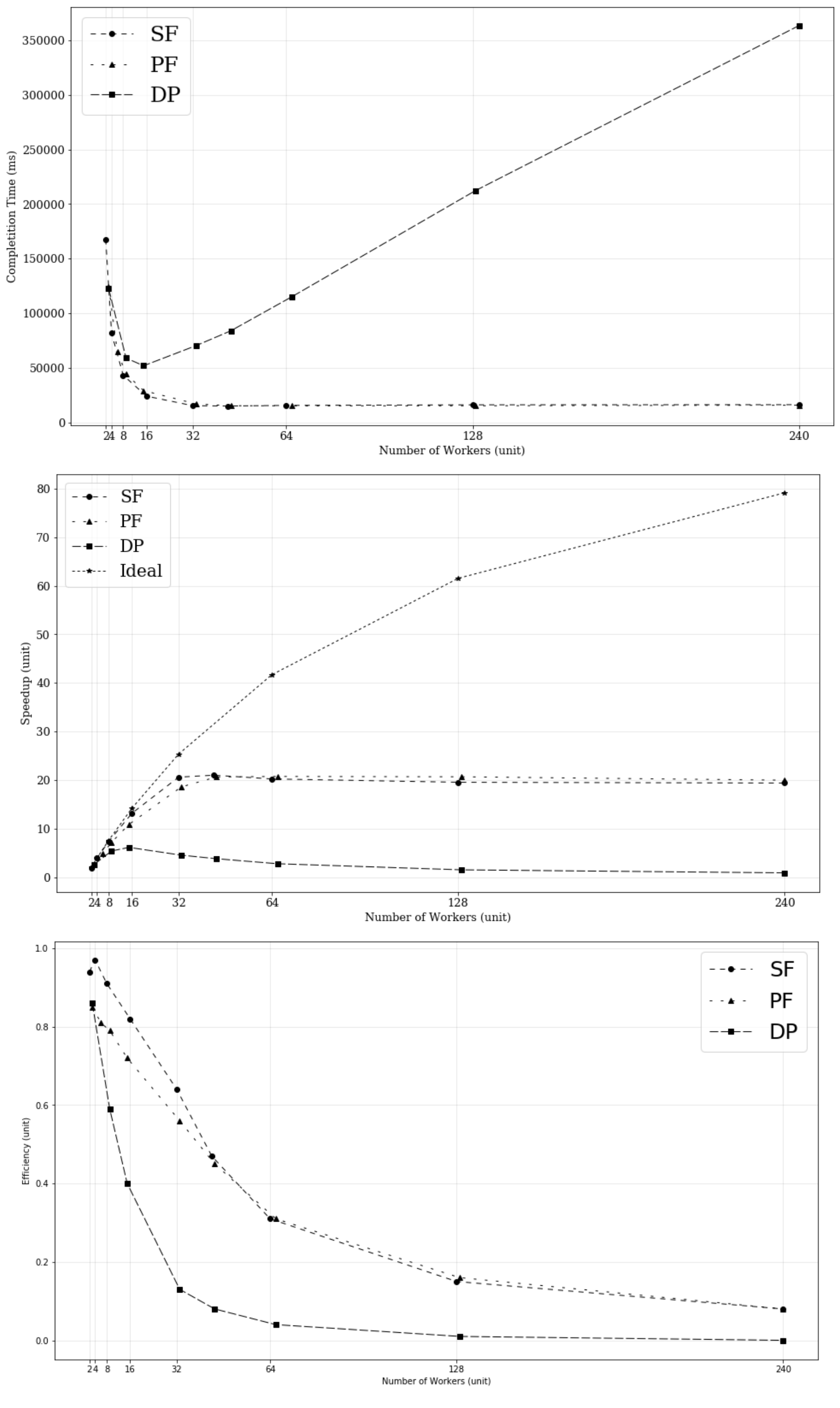

Figure 2: Completion time, speedup and efficiency with **C++ Threads**

6 User Manual

The folder *Data* contains the images used to test the application. The folders *Sequential*, *Thread* and *FastFlow* contain the code of the so-called implementations.
The file *input.config* is a naive configuration file providing functional parameters to the parallel applications:

- **-skeleton**: the possible values are **pf** (pipe-farm), which represent the compositions **4, 6**, and **sf** (sequential farm), which represents composition **1**;
- **-definition**: the possible values are **hd**, **md** and **ld**[3], which identify the images from the biggest to the smalllest;
- **-sigma**: the sigma value. It should between [0, 1].

The files *structs.hpp* and *functions.hpp* define and implement structs and functions shared by all implementations; *CImg.h* is the library used in to read/write pictures with C++.
Both parallel implementations make use of files named *stages.hpp* to implement the application's stages and *Filter.cpp* which includes the main. The Thread implementation also defines a file *MyQueue.hpp* which implement a minimalist concurrent queue exploiting std vector.

The parameters of the sequential application are all inputs of the main; the parameters of both parallel versions are separated: the non-functional parameters of the application, together with the stream length to simplify different runs, are inputs of the main; the functional parameters are in *input.config*.
The output is a bitmap file called *"bw_test'**def**'.bmp"*, for example *bw_testhd.bmp* if the *hd* image is used.

Instructions to compile and run the application:

Sequential

- Enter into *Sequential* folder;
- Compile[4] with **make Filter** to use g++ compiler; **make IFilter** to use icc[5];
- run with arguments: **./Filter sl=1 def=hd sigma=0.1**, where the parameters represent stream length, definition of the image and sigma value. They assume the shown default values if omitted. The order must be respected.

C++ Thread and FastFlow

- Enter into *Thread* or *FastFlow* folders;
- Compile with **make Filter** to use g++ compiler; **make IFilter** to use icc;
- run with arguments: **./Filter sl=1 nwFarm=1 nwH=1 nwF=1**, where the parameters represent stream length, number of workers of the farm, of `h` and `f` (if parallel). They assume the shown default values if omitted. The order must be respected.

[3]they stand for HighDefinition, MediumDefinion and LowDefinion, just mnemonic
[4]It is going to take a while
[5]the executable's name follows the rule used