# SPM@Unipi 17/18 Project
# [Histogram Thresholding]

*by : Lanpei Li ( Matricola: 533772)*

# 1. Usage Manual

- Code repository:

  https://github.com/lilanpei/SPM_HistogramThresholding.git

- How to compile the code :

  1. Download fastflow :
     - ➢ cd ~
     - ➢ svn co https://svn.code.sf.net/p/mc-fastflow/code/fastflow
  2. Download source code from github repository and unzip it to the work folder.
  3. Enter the work folder, Compile the code :
     - ➢ source /opt/intel/compilers_and_libraries/linux/bin/compilervars.sh intel64
     - ➢ make

- How to run the code :

  1. Sequential code :
     - ➢ ./Sequential nImages threshold
  2. C++Threads code :
     - ➢ ./Threads nImages nWorkers threshold
  3. Pipeline code :
     - ➢ ./Pipeline nImages threshold
  4. Farm code :
     - ➢ ./Farm nImages nWorkers threshold
  5. Farm with Pipeline code :
     - ➢ ./Farm_Pipeline nImages nWorkers threshold
  6. Sequential_parellelFor code :
     - ➢ ./Sequential_parellelFor nImages threshold
  7. Pipeline_parellelFor code :
     - ➢ ./Pipeline_parellelFor nImages threshold
  8. Farm_Pipeline_ParellelFor code :
     - ➢ ./Farm_Pipeline_ParellelFor nImages nWorkers threshold
  9. For test different number of works in the farm :
     - ➢ bash ./test.sh

  ❑ Note:

- nImages : Number of images in the Input stream.
- nWorkers : Number of works in the farm.
- threshold : target percentage for thresholding

## 2. Major Design and Implementation Choices

1) Sequential :

    a) A while-loop looping the images one by one (as a stream) to do Histogram-Thresholding.
    b) Histogram-Thresholding process has 3 steps :
        i) Convert image from RGB to GrayScale.
        ii) Calculating Histogram.
        iii) Perform Thresholding.

2) C++Threads :

    a) Inside the while-loop, fork a vector of threads, each thread process a single image.
    b) Because the image read and write can't be paralleled, so just perform once and using only a single input image, copy it to do image processing n times.

3) Fastflow -> Pipeline :

    a) Consider the time usage of each step of Histogram-Thresholding for the input image from 1), RGB to GrayScale is the bottleneck, in order to get the minimum service-time, so put RGB to GrayScale step as a single stage and combine Histogram Calculating and Thresholding together into the second stage.

4) Fastflow -> Farm :

    a) Using the idea from 2), built a farm that each worker process a single image.
    b) A Emitter for scheduling the given number of image as a stream input.
    c) removing the unused default collector.

5) Fastflow -> Pipeline embedded in Farm :

    a) Combine the idea from 3) and 4) together, embed a two stage Pipeline from 3) into each worker of the Farm from 4).
    b) Need to check the overhead introduced compared to 4).

6) Sequential with parallelFor :

    From the result of running the 1) Sequential implementation , the time usage of each step for processing a single image is shown below:

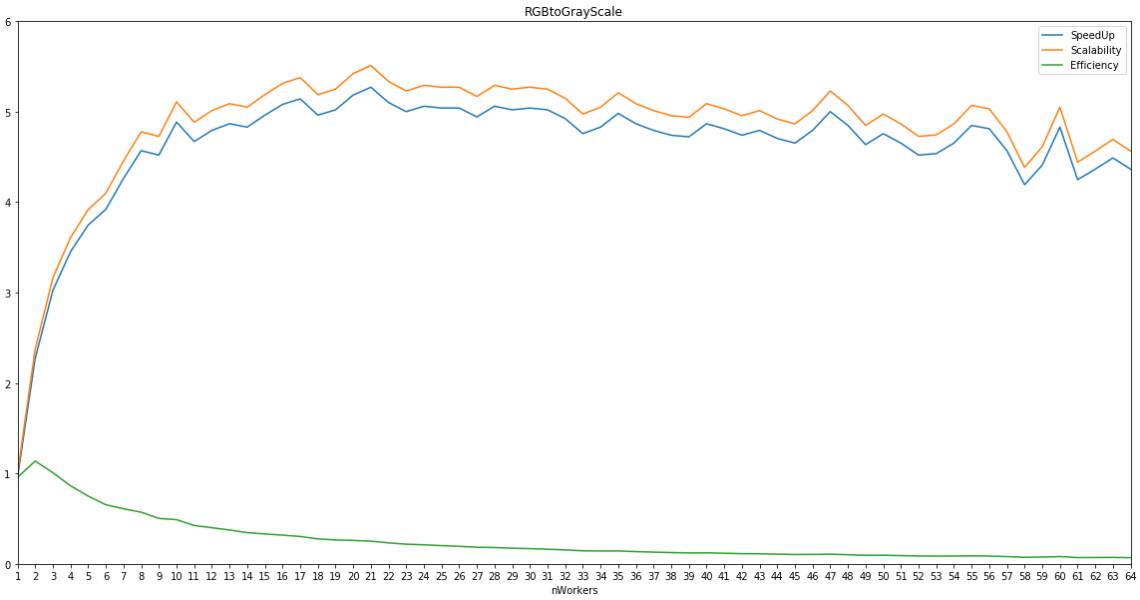
    a) Open image (761 ms),
    b) RGBtoGrayScale (1270 ms),

c) Histogram (235 ms),
d) Thresholding (520 ms),
e) SaveImage (728 ms).

The Sequential part “open image” a) and “save image ” b) cannot be paralleled, so the adopted strategy is just open image once then send the copies to the next step and save processed image only once(drop others processed image).

The rest of the steps can be paralleled by “parallelFor”, because the for loop are adopted in each steps.

This “Sequential with parallelFor” implementation is used for testing how many workers used in “parallelFor” can give the minimum service time, the performance plotting is as below:




From the performance plotting above, “RGBtoGrayScale ” step with 21 nWorkers and “Thresholding” step with 3 nWorks can give the minimum service time for processing a single image :

a) RGBtoGrayScale : nWorkers (21) -> 241 (ms)
b) Thresholding : nWorkers (3) -> 211 (ms)

(Note that the sequential “Histogram” step take (235 ms) < “RGBtoGrayScale” step with 21 nWorkers - 241 (ms), so don’t need to parallel this step to get the minimum service time.)

7) Pipeline with parellelFor :

Based on the results above, in order to speedup the whole processing, I built a "pipeline (feedback) with parallelFor" implementation, the architecture is shown below:

a) First stage : An emitter to schedule each image as a stream; if received a feedback of processed image then save it (only once) and return EOS.
b) Second stage : RGBtoGrayScale with 21 nWorkers
c) Third stage : Histogram
d) Fourth stage : Thresholding with 3 nWorkers

After testing this “Pipeline with parellelFor” implementation, I found that, the time usage of the Second stage is about 420 (ms), and the time usage of the third stage is about 525 (ms), they are unstable for processing each image. This probably because after 21 nWorkers used to parallel the second step, it’s hard to guarantee the follow steps are ran in the same cpu socket, so copy data from different caches will introduce the overhead.

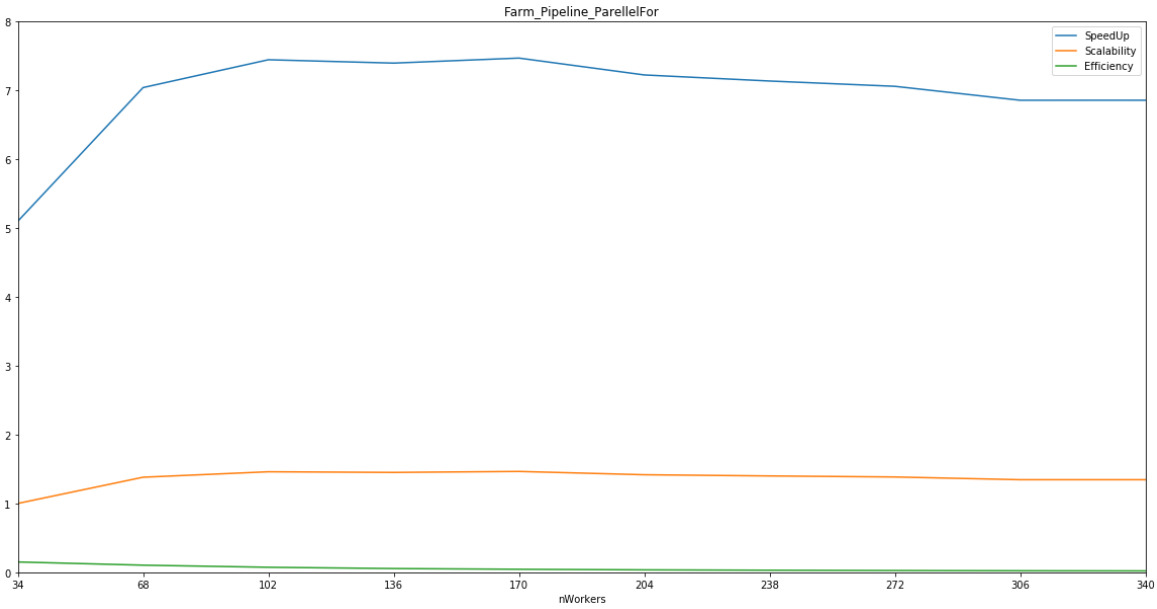
After tuning the nWokers for each steps (Second stage 28 nWokers, third stage 3 nWorkers, fourth stage 3 nWorkers), the final time cost for 300 nImage using “Pipeline with parellelFor” implementation is 144492 (ms), it’s much larger than the best 4) “Farm” implementation (28 nWorkers -> 39943.8 ms) and the best 5) “Pipeline embedded in Farm” implementation (28 nWorkers -> 37717.2 ms).

8) Pipeline with parellelFor embedded in Farm :

based on the result of 7), in order to speedup the processing, I try to put the “Pipeline with parellelFor” embedded in each worker of a Farm, then adopt a emitter to schedule the images to workers and remove the collector but adopt feedback from each worker to emitter for saving the image.

After testing this “Pipeline with parellelFor embedded in Farm” implementation, the best result for 300 image is 83424.8 (ms) with 3 nWorkers in the farm,it’s still much larger than the best 4) “Farm” implementation (28 nWorkers -> 39943.8 ms) and the best 5) “Pipeline embedded in Farm” implementation (28 nWorkers -> 37717.2 ms).

The performance plotting of 8) is shown below:




9) Conclusion :

The implementations 7) and 8) have a very good service time for processing a single image, but when processing 300 images, the overhead introduced is much larger than the performance gained (probably due to different cpu socket need to copy data from different caches).

So consider the time usage for 300 images and the performance, the best implementation is the 4) “Farm” implementation (consider its Efficiency performance is better than 5) “Pipeline embedded in Farm” implementation).

## 3. Performance Plots

★ Note : In my all implementations, the input parameter nWorks denotes the number of works used in the farm, so for easy plotting, I use this input parameter to present the parallelism degree n, but for the case of Pipeline embedded in Farm, actually the parallelism degree should also consider the two-stage-pipeline ’s contribution(nWorks*2).

1) Speed Up :

   a) s(n) = Tseq / Tpar(n)
   b) Speedup gives a measure of how good is our parallelization with respect to the “best” sequential computation.
   c) When 28 threads used for “Farm” and “Pipeline embedded in Farm” implementations, we can achieve the best SpeedUp, but for “C++Threads” case seems when use 198

threads can achieved the best SpeedUp.

d) After about the number of threads exceeds 61, both “Farm” and “Pipeline embedded in Farm” implementations starts the first drop due to the overheads introduced.

❏ Note: For “Pipeline embedded in Farm” when at the number of works equal to n, but consider the pipeline’s contribution to the parallelism degree, actually the number of threads used already exceeds 2n.


2) Scalability :

a) scalab(n) = Tpar(1) / Tpar(n)

b) Scalability should looks slightly different than speedup. It measures how efficient is the parallel implementation in achieving better performances on larger parallelism degrees.

c) But in my implementations, due to the image read and write just perform once for each parallel implementation no matter how many number of threads used, so Tpar(1) is more or less similar to the Tseq in all parallel implementations.

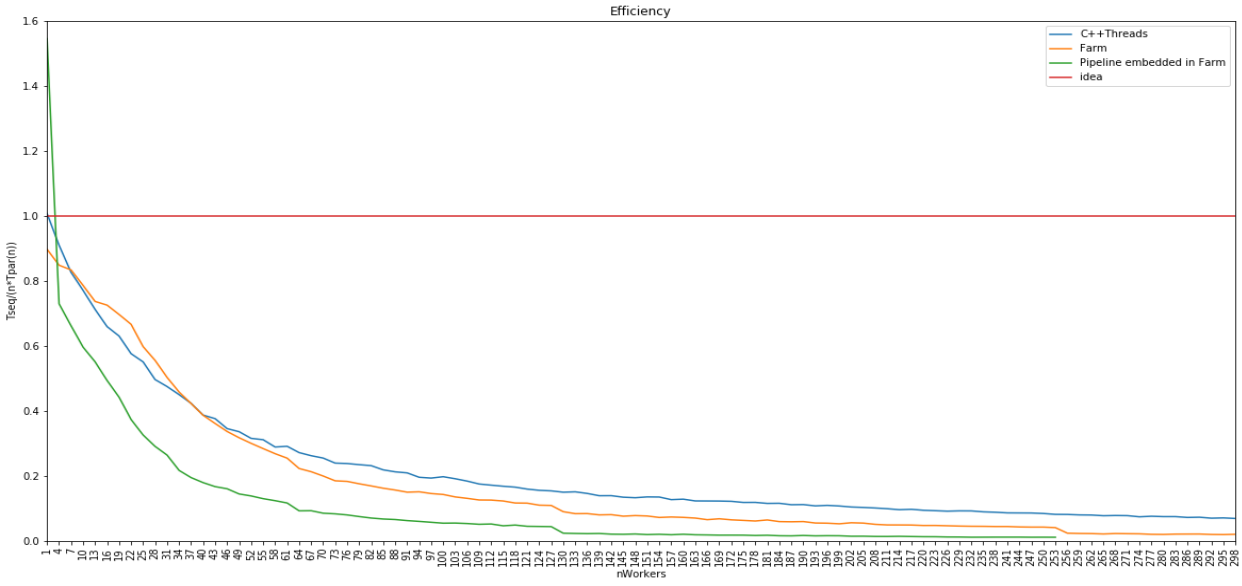
d) So consider the plots of Scalability is quite similar to the plots of SpeedUp, we can get the same conclusion as the Speed Up plots analysis.


3) Efficiency :

a) $\epsilon(n) = Tseq / ( n * Tpar(n) )$

b) Efficiency measures the ability of the parallel application in making a good usage of the available resources.

c) In the plots, we can observe that between 10 to 30 number of threads used, “Farm” implementation has a better Efficiency, after 30 threads, the “C++Threads” implementation has a better Efficiency.



## 4. Conclusions

Consider my implementation choices and their plots shown above :

1) The “Farm” implementation with 28 number of works has the best performance.
2) After using 61 number of works, “Farm” and “Pipeline embedded in Farm” implementations tend to have a worse performance due to the overhead introduced become larger than the performance gained.
3) “Farm” and “Pipeline embedded in Farm” implementations have quite similar performance, so the pipeline embedded seems does not showing much benefit.

## 5. References


1. SPM lecture and notes : http://didawiki.cli.di.unipi.it/doku.php/magistraleinformaticanetworking/spm/start
2. FastFlow tutorials and examples : http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:tutorial
3. CImg tutorials and examples : http://cimg.eu/reference/group__cimg__tutorial.html
4. C++ Spec and examples : http://www.cplusplus.com/reference/
5. RGB to Grayscale using CImg : http://obsessive-coffee-disorder.com/rgb-to-grayscale-using-cimg/
6. Histogram Calculating program : https://stackoverflow.com/questions/7428657/histogram-calculating-program
7. Ahmad-Alleboudy’s submission for the SPM video filter project : https://github.com/alleboudy/spm