CSC2002S – ASSIGNMENT 1

PCP Median Filter Image Smoothing

MYZMBA002 | August 2022

Introduction

Image smoothing is a process by which the noise on an image is removed to give a smooth or blurred effect. This is achieved by assigning the middle most pixel, within a given range of pixels, an average value determined by the values of its surrounding pixels.

The process to achieve this effect on an image can be done sequentially, by running through an image from length to length and calculating and assigning the middle pixel with the calculated mean value. Although, in theory a Parallel algorithm can be used in place of a sequential algorithm, to achieve the same result at in a more efficient time. Unlike a sequential algorithm, the main aim of Parallel program is to achieve or produce faster results by maximizing the use of the complete computational resources, by dividing the workload between the computer architecture, mainly being the cores within a multicore computer system. In this way, each core handles a fraction of the task, as opposed to having one CPU doing all the work.

Method and Design

Within this particular experiment our aim was to demonstrate the Speedup of the image smoothing when utilizing a Parallel Program. This was achieved by contrasting with Serial Programs that execute the same algorithm sequentially.

With regard to the method of pixel manipulation to achieve a smoothed image affect a Median Filter and a Mean Filter was used. The Mean Filter assigned the middle most pixel with the average value of surrounding pixels, whilst the Median Filter assigned the middle most pixel with the median value of the surrounding pixels. Both were tested in Serial and Parallel programs, giving us four classes in total. Within each program a *Window* integer variable was declared, allowing manipulation for how big or small the window for the specific iteration could be. The *Window* any on whole number which would determine how many pixels the program dealt with at a time.

1. Mean Serial Filter Class

This classes read in an image and stored it within a BufferedImage variable *img*. Using a nested for-loop with limits corresponding to the height and width of the image, each pixel rgb value of the image was extracted. In another nested for-loop, in the existing, the separate rgb values were added to their corresponding colors and stored in integer colour variables. The average of each colour variable was then calculated before being assigned to an output image variable.

2. **Mean Parallel Filter Classes**

   The constructor of this class considered four arguments, mainly the image start-width and end-width, and the start height and end height. Using a divide and conquer algorithm inherit to the Fork-Join Frame work, the program divided the image along its width using a minimum threshold value. The class had an inner method *meanCompute()* which utilized the same algorithm as the Mean Serial Class. Within a *compute()* method, given that the minimum width threshold value is met then the *meanCompute()* would be called and computing for a smaller given image width. This was achieved using the Fork-Join Pool which enable the initialization of multiple threads needed.
3. **Median Serial Filter Classes**

   Reads in input image into a BufferedImage variable such as the last and the pixel values are computed from the image using the image coordinates corresponding to the image width and height. Is the specific rgb values are read in from each pixel they are stored in corresponding integer Array lists. After the completion of each window size each array list is at capacity, after which each is sorted using the *Collections.sort()* method. The median or middle index value of the Array list is then stored in a new pixel and assigned to an output image.
4. **Median Parallel Filter Classes**

   The constructor of this class also took in four arguments such as the Mean Filter Parallel class and also divided the image along its width. This class although had a *medianCompute()* method which programmed a similar algorithm as the Median Serial Class. This class also utilized a Fork-Join Framework of divide and concur, and invoke the Fork-Join pool to create the necessary amount of threads needed for each execution.

Testing and Benchmarking

The variable of concern in the experimentation was mainly the time, which was computed for the duration of the process of creating a smooth image. The *System.currentTimeMillis()* was strategically placed within the program at the beginning and the end of the execution and the execution time was calculated form the difference. All the Programs were run and tested on a four core machine (Lenovo Ideapad).

Within the Serial Programs *System.currentTimeMillis()* was placed before and after the four nested For-loops.

Within the Parallel Programs the *System.currentTimeMillis()* method was called before and after the invoke method of the Fork-Join Pool. This was done with careful consideration to the fact that the method waits for all running threads to execute to completion before computing the results.

In this experiment the window size was also manipulated three times, with 3, 15 and 31 being the sizes tested for.

The image size was also manipulated to test whether the parallel program was more or less efficient for a given window size. The images where used.

- Image 1: 450x338
- Image 2: 900x599
- Image 3: 1000x1497

For the Parallel Executions Specifically the Sequential cut-off value was manipulated to test which was most efficient and was manipulated three times for each of the given images and window sizes.

- SQ 1: 100
- SQ 2: 350
- SQ 3: 100

Results

Each test was repeated five times with the best out of the five results being used to create graph. Time stamps were taken in milliseconds and converted to seconds.

Original Import Image

Mean Filter Image Output

Mean Filter Serial

Mean Filter Parallel

Median Filter Image Output

Median Filter Serial

Median Filter Parallel

Mean Filter

		Serial	Parallel		
			Seq Cut-off		
			100	350	1000
Image 1	3	0.134	0.39	0.17	0.13
	15	1.25	5.08	1.58	1.30
	31	4.61	17.44	5.47	4.67
Image 2	3	0.37	2.04	0.61	0.37
	15	4.38	33.17	8.64	4.24
	31	16.06	133.80	31.84	16.18
Image 3	3	0.60	5.17	1.29	0.92
	15	10.96	90.97	23.33	13.29
	31	44.56	364.40	94.48	55.10

Median Filter

		Serial	Parallel		
			Seq Cut-off		
			100	350	1000
Image 1	3	0.39	1.34	0.48	0.37
	15	11.65	37.63	13.24	10.90
	31	51.92	209.23	60.10	49.97
Image 2	3	0.96	6.44	1.69	0.96
	15	41.31	282.31	71.31	41.15
	31	210.08	N/A	347.84	210.10
Image 3	3	2.12	14.96	4.20	2.73
	15	118.70	778.71	194.35	139.63
	31	206.19	N/A	959.58	675.39

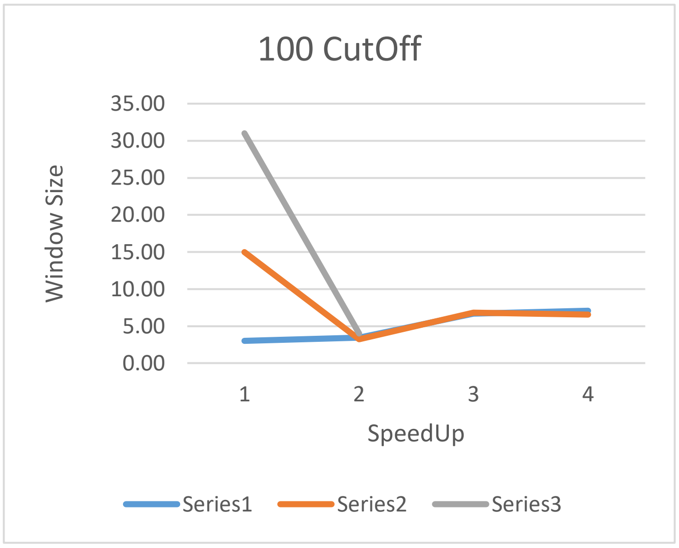
N/A – ran for too long and had to manually stop.

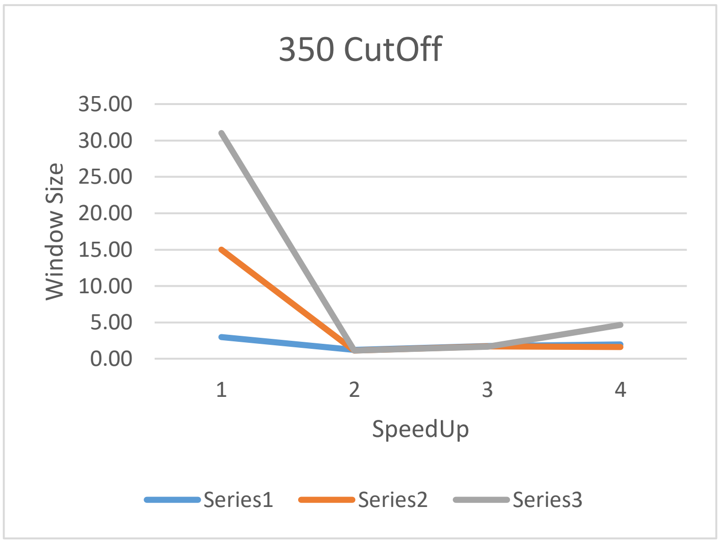
Speedup is the comparison of the Work (time taken on a single processor) versus the Span (time take on an infinite amount of processors). This relation between the Work and the Span is expected (in theory) to produce an exponential gradient, with the serial execution time reducing to a fraction of the number of CPU's used in the Parallel execution.

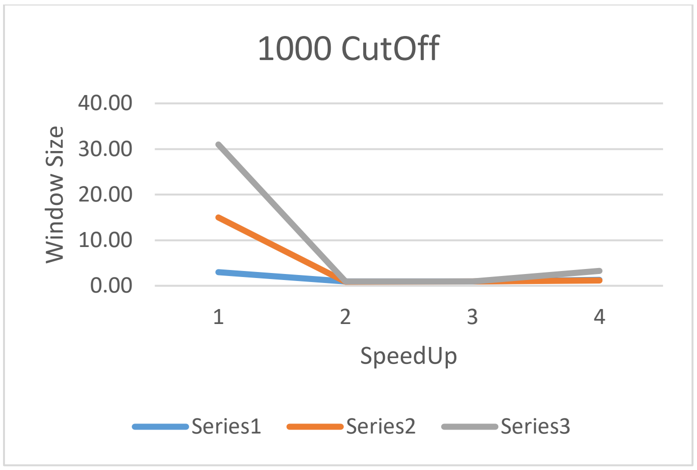
To display this relation, we use the equation:

Speed up for P processors = T1/ Tp = time on 1 / time on P processors

As mentioned before, the parallel execution was run on a computer with 4 cores, and so we would expect the speed up to be 4 times the serial execution.







Above is the graphs showing the

Analysis

After much testing and benchmarking the experiment did not produce presumed results. Instead it revealed that in some cases (when the image is smaller and the window size is larger) Parallel program is less efficient than the Serial algorithm. In other cases, (Where the image is larger and the window size is smaller) the Parallel execution does bring a level of efficiency but the speed up is still not significant enough. This may be due to the computer used or a poor algorithm.

As the pictorial outputs show, by comparison the Median Filter Class achieve a more distinct Blurred effect on the images, and can be seen as working better than the Mean Filter Classes. This could be due to the fact that extracting and assigning a median value to the center pixel is best suited for creating such an affect.

The values highlighted in blue present the fastest Parallel execution times which can be compared with the corresponding serial execution times.

Overall, this experiment could have been improved if less variables where being compared all at once, which would align more with the fundamental principles of Scientific Experimentation which underline having a single dependent variable and a single independent variable. In this case to many things were manipulated with and so making too many independent variables for a concise comparison.