EXPERIMENT OF A SEQUNTIAL PROGRAM AND A PARALLEL PROGRAM USING THE PICTURE

Introduction

This report presents the speedup and the worthiness of the parallel programs compared to the sequential one. There will be four programs, two sequential ones and two parallel ones namely: MeanFilterSerial, MedianFilterSerial, MeanFilterParallel and MedianFilterParallel referred to here as M1FS, M2FS, M1FP and M2FP. This practical experiment will be a valuable experience because similar filters are often being used by almost every teenager mostly the designers. This practical experiment presented on this report uses the programs to filter a particular picture of different sizes(dimensions) and make it blur. The expected possible speed of 1.5 – 2, running this on 4 core machines.

Hypothesis:

1) The Sequential programs will be much slower than that of parallel programs on the picture of a bigger size.
2) The parallel programs will be much faster on the machine with more cores (CPUs).

Variables:

1) Dependent Variable:
    - Speedup
2) Independent variables:
    - Image Size (area of the image = number of pixels = width x height)
    - Machine (number of cores)
    - Window width
3) Controlled variable:
    - Temperature – comparable experiment/programs are executed on the same day

Method description

MeanFilterSerial.java

Method (mean ()) reads the image and stores it as a Buffered image variable then iterate through it using the loops, where the first two loops keep track of the target pixel coordinates. I then create the window (box) inside the first two loops then calculate the mean/average of the colours of the neighbours of the target pixel then write back to the copy of my Buffered image at the location of the coordinates from the outer loops.

MedianFilterSerial.java

Method (median ()) also reads the image and iterate through it but this time it does not calculate the mean instead it stores the colours independently on their own separate array lists, and sort them ascendingly then take the middle value to replace the target value at the copied image.

MeanFilterParallel.java & MedianFilterParallel.java

These classes do not have default constructors, the constructor **MeanFilterSerial (int width, int start, int end)** and **MedianFilterParallel (int width, int start, int end)** takes three integers the width of the original picture, the starting value which initially is zero, and the height of the image. The method main () does the reading of the image, calling the method compute (), and the writing then done filtering. The method compute () filters the image sequentially only if the area – number of pixels on the task is below the sequential cut-off (seqT) using the average of the colours from the neighbours of the target pixel and the middle value between the neighbours of the target pixel. If the area is still big then the task is still big it will be broken down on the invokeAll () at the splitting position calculated with the changing height. The tasks are divided in halves by the height, while the width remains the same, and all tasks run or work independently of each other parallelly.

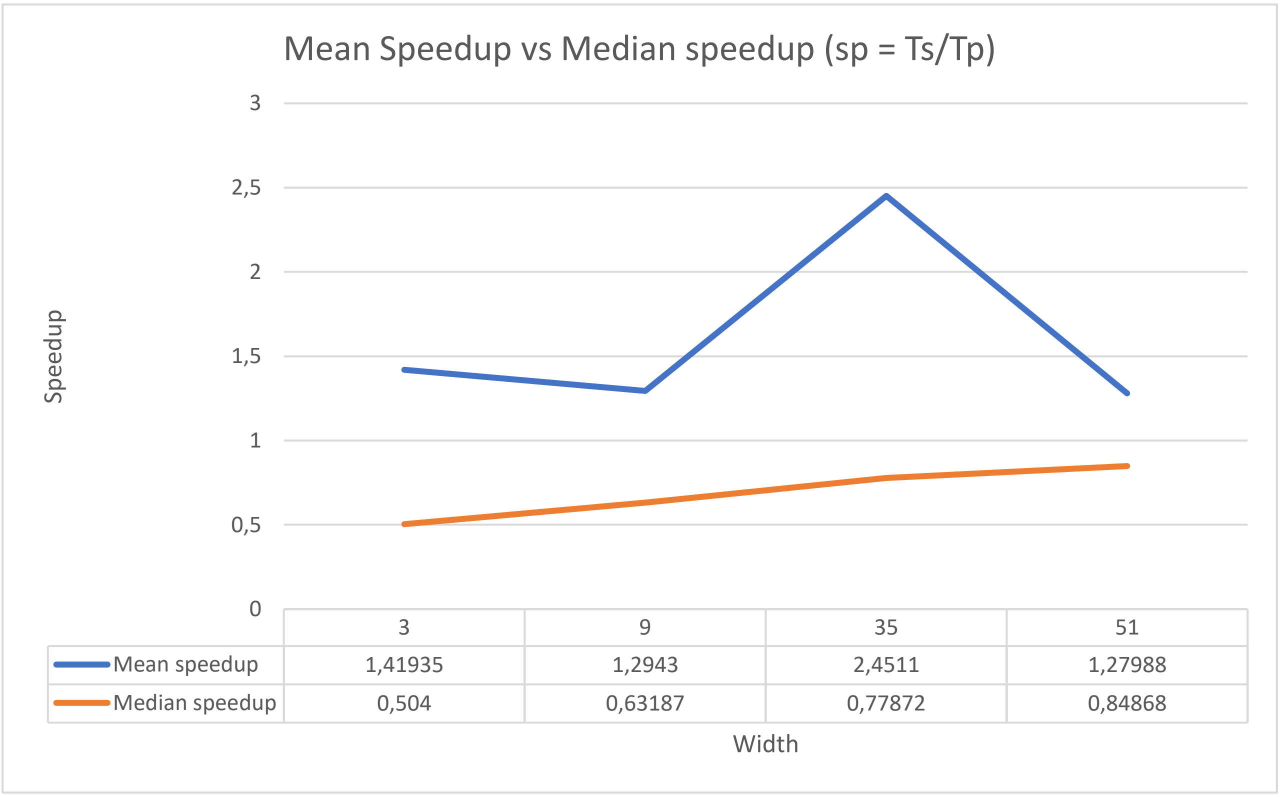
How my parallelisation algorithms work

- Given the inputs (pathname of the input picture) as a string, (pathname of the output picture), and the window width (odd number above 2) as a string then converted to integer
- Reads the image and sores it as a Buffered image.
- Make the copy of the image
- Let window width be 'w', looping through the image starting at the index position w/2 on both loops to exclude the borders
- Initialize the array in mean filter or the array list in median filter
- Do the filtering sequentially if the area (number of pixels are of the picture) is below the sequential cut-off
- Looping from the index position of the outer loop less w to the index position of the outer loop more w to create a box window with the neighbour pixels around the target pixel
- Getting the pixel from the window and extract the colours individually.
- Calculate the mean and sort the array list ascendingly, get the middle value (colour) from the array list
- Change the colours back to the pixel
- Replace the pixel on the copy image at the position of the target from the outer loops with the mean and with the middle-coloured pixel on the median filter
- Split the image in halves by the height if the area is still big, one task starts from the index 0 to the index of the splitting less 1
- The other task starts from splitting position of the height the position of the height of the image, and height keeps updating as its broken down
- These tasks execute independently, and parallelly
- The main method writes the image to the path given by the user
- In validating the correctness of the results, for the same input. The validating algorithm takes two inputs, which are output copies and make copies to avoid changing the data and get false results

- Then iterate through these inputs comparing from pixel to pixel and colour to colour on the pixels
- Then return a Boolean ‘true’ when they are the same
- For the timing, on each class, there’s a print statement after calculating the difference from the starting time (calling the compute on parallels and calling the mean and median on serials)
- Speedup is measured by tacking the best minimum results from 10 of runs,
- I ran this on my PC with 8 cores and senior and ishango labs with 4 cores.

Results and discussion:

The Graph showing the speedup achieve in Mean parallel executions vs speedup achieve in median Parallel executions with the window with for PC with 8 cores.



	3	9	35	51
Mean speedup	1,41935	1,2943	2,4511	1,27988
Median speedup	0,504	0,63187	0,77872	0,84868

The graph shows the Parallel algorithm is not efficient for the window width between 3 and 9, but its shows to be more efficient for the window width 9 up to 35. For the median algorithm it is efficient as the window size increases. This speedup experiment is done on a small image.

Conclusion:

After completing this project, I have observed that parallel algorithms are not always ideal for any problem with different input sizes and different number of threads a parallel algorithm uses.

Issues:

My parallel took almost forever to run, especially on the 4-cores machines, hence this experiment is conducted on a small picture size. I think it is due to my sequential cut-off. I think the problem can be fixed by adjusting the sequential value.

Because of this issue, I do not have a conclusive argument.