MBLSHA003

CSC2011S

PCP1_Assignment

Report

The aim of this assignment is to find the benchmark of when is worth the extra effort to use parallel programming instead of sequential programming when applying mean or median filter on a 2D image. I will use two parallel programs and two sequential programs and test them on **different machines** (one with 4 cores and the other with 8 cores) the time they take to smooth RGB colour images with mean and median filters. I will use the physical computers from the Senior labs (they have 4 cores) and my computer that has 8 cores. On carrying out the experiment I will benchmark the parallel programs to the serial ones after getting their execution times they take processing different image sizes with different filter sizes (e.g 3x3,5x5,11x11,15x15). Benchmarking the parallel graphs to the serial ones I will be able to generate speedup graphs that show when I get the best parallel speedup.

I will also determine the point at which one should begin using fork/join algorithm instead of sequential processing. I will then graph all my findings to draw concrete conclusion. Therefore, with these points above in mind, let us begin.

Method section:

The first thing I did was to come out with an efficient algorithm to move the sliding-window on the images with different filter sizes. I came up with the algorithm you will see below using this simple fictitious visualization of a 4x4 size image presented by its pixels:

Fictitious image

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

I then took the whole image as a matrix where each pixel is represented by x and y co-ordinates. I used the top left corner as the starting point vector of (0,0) then moving across the image I incremented the x co-ordinate and similar when moving down the image I increment the y co-ordinate. Here is the matrix representation of the above fictitious pic:

Matrix visualisation of the image

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

With the matrix representation of the image I used a 3x3 sliding-window to calculate the mean filter first and then the median one. Here is the snapshot of the code:

```
public class Check {
public static void main(String[] args) {

int[][] num = new int[4][4];

int count = 1; // use to initialize the pixels in the matrix

// populate the matrix fromm left to right and then down with values
//each inner loop iteration from 1 to 16

for(int i = 0; i< num.length;i++){
for(int j=0;j<num[0].length;j++){
num[i][j] = count;
count++;
}
}

int windowWidth = 3; // sliding window width

int sum1 = 0, r = 0, r2 = windowWidth, c = 0, c22 = windowWidth;

while(r2<5){
for(int i = r;i<r2;i++){
for(int j=c;j<c22;j++){
// print statement to check co-ordinate print correct values
System.out.println("i = "+i+" j = "+j + " from "+num[i][j]);

// sum values in a window
sum1 += num[i][j];
}
}
System.out.println("Sum "+sum1);
System.out.println("Average (mean) "+sum1/9);

System.out.println("Pixel position to be used to filter "+r2-
windowWidth /2+" "+c22- windowWidth /2);

// reset sum to zero for next window values
sum1 = 0;

// verify what were the middle values
System.out.println("c22 "+c22);
System.out.println("r2 "+r2);
```

```
// check the right edge of the image reached to increment the height
variable of the window
if(c22>3){
System.out.println("True");
c = 0; c22 = 3; r++; r2++;
}else{
c++; c22++;
}
}
}
}
```

So I basically used hard-coded values of an imaginary image of 4x4 size by populating a 2D array **num[][]** with 1 to 16 to verify calculation when I calculate the mean of the pixels within a sliding-window are correct and that every all the pixels were transverses except the pixels on the edges that are half the width of the sliding window used.

I used a while loop to slide a window on the image until the edge of the bottom of the image height. Within the while loop I had two for-loops to loop every pixel within a window. I initialized the start and the end values of the two for-loops outside the while loop equal to the window width in subject and incremented these variables guarded by the actual width of the image. Consequently, when the edge is reached, I increment the values guarding the height of the window and the process repeats until the I reach the bottom of the image. I then printed out the values for the **sum**, **average** and the **co-ordinates** to use on the other image to apply the filter.

Here is the output I got from the code:

Image 1: Results from the code to confirm my algorithm works

```
shaun@shaun-Inspiron-3580:~/AssignmentPCP1$  cd /home/shaun/AssignmentPCP1 ; /usr/bin/env /usr/lib/jvm/java-17-o
penjdk-amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/shaun/.config/Code/User/workspaceStorage
/a865abaf09e69438c95e317dcd2167bd/redhat.java/jdt_ws/AssignmentPCP1_889ff86c/bin Check
i = 0 j = 0 from 1
i = 0 j = 1 from 2
i = 0 j = 2 from 3
i = 1 j = 0 from 5
i = 1 j = 1 from 6
i = 1 j = 2 from 7
i = 2 j = 0 from 9
i = 2 j = 1 from 10
i = 2 j = 2 from 11
Sum 54
Avg 6
Pixel position to be used to filter 1 1
c22 3
r2 3
i = 0 j = 1 from 2
i = 0 j = 2 from 3
i = 0 j = 3 from 4
i = 1 j = 1 from 6
i = 1 j = 2 from 7
i = 1 j = 3 from 8
i = 2 j = 1 from 10
i = 2 j = 2 from 11
i = 2 j = 3 from 12
Sum 63
Avg 7
Pixel position to be used to filter 1 2
c22 4
r2 3
True
i = 1 j = 0 from 5
i = 1 j = 1 from 6
i = 1 j = 2 from 7
i = 2 j = 0 from 9
i = 2 j = 1 from 10
i = 2 j = 2 from 11
```

```
i = 3 j = 0 from 13
i = 3 j = 1 from 14
i = 3 j = 2 from 15
Sum 90
Avg 10
Pixel position to be used to filter 2 1
c22 3
r2 4
i = 1 j = 1 from 6
i = 1 j = 2 from 7
i = 1 j = 3 from 8
i = 2 j = 1 from 10
i = 2 j = 2 from 11
i = 2 j = 3 from 12
i = 3 j = 1 from 14
i = 3 j = 2 from 15
i = 3 j = 3 from 16
Sum 99
Avg 11
Pixel position to be used to filter 2 2
c22 4
r2 4
True
```

Thus, I was able to verify my code works by trying different sizes of the images and different sliding-window sizes. I then used this algorithm on both the serial and parallel programs, however, substituting the 2D array with the actual pixels of an image which have the structure of a 2D array.

From the above output and algorithm I was able to see how many times each pixel is visited by a counter variable leaving out the edges of the image half the window size. For instance, on the folder called image inside the AssignmentPCP1 folder with my submission work,

there is a 460x461 image. The edges of this image and all other images' edges cannot be filtered/ smoothed when you run the programs because of the sliding-window, and they become clearer as the sliding-window size used is increased. The mean and median filter on the these images almost look the same but the median image is more blurry and has sharp edges than the mean filter. Here is the example seen from a 11x11 window using a mean and median filter of a serial program:

Image 2: Original image

Image 3: Mean filtered image

Image 4: Median Filtered

For the parallelisation algorithm I used the fork/join framework and applied the divide and conqueror approach. I chose my **cut-off (optimum threshold)** value to be the ratio of the image height to the number of processors as system has less one. Therefore, my cut-off value was not static, but dynamic based on the picture and the number of processors the system has. Hence, my algorithm/ approach is scalable to maximise the available processors when using the common pool method of the ForkJoinPool class. This I concluded through changing the cut-off value of the compute method during the experiment using a python file to automate the whole processes and found that it produced the optimum speedup. For example, an image with 399 pixels height and using a system with 8 processors, my cut-off value would be 399/(8-1) which is 57 pixels. That is where my computation would kick in.

For the parallel programs I used a **public inner class** that extends the RecursiveAction class using the java.util.concurrent package. Within this class I then overridden its compute() method. That is where I had the cut-off implementation to signal when computation can be done by a thread or else divide the problem into smaller pieces and until the threshold of the cut-value is reached. Whenever the program divides the problem into smaller pieces I create an object of the RecursiveAction and used it to create a new thread for each piece of the divided problem. For the main thread not to be idle, I used the compute method on another object of the RecursiveAction to run the compute() method as normal method call. I used two objects of the RecursiveAction and split half of the problem to each object. One to call the normal compute() method as a normal call and another assigning a new thread for it

and lastly calling the join() method on the object I assigned a thread, that it may not terminate and as well as other threads joined to them until the main thread terminates and consequently other threads.

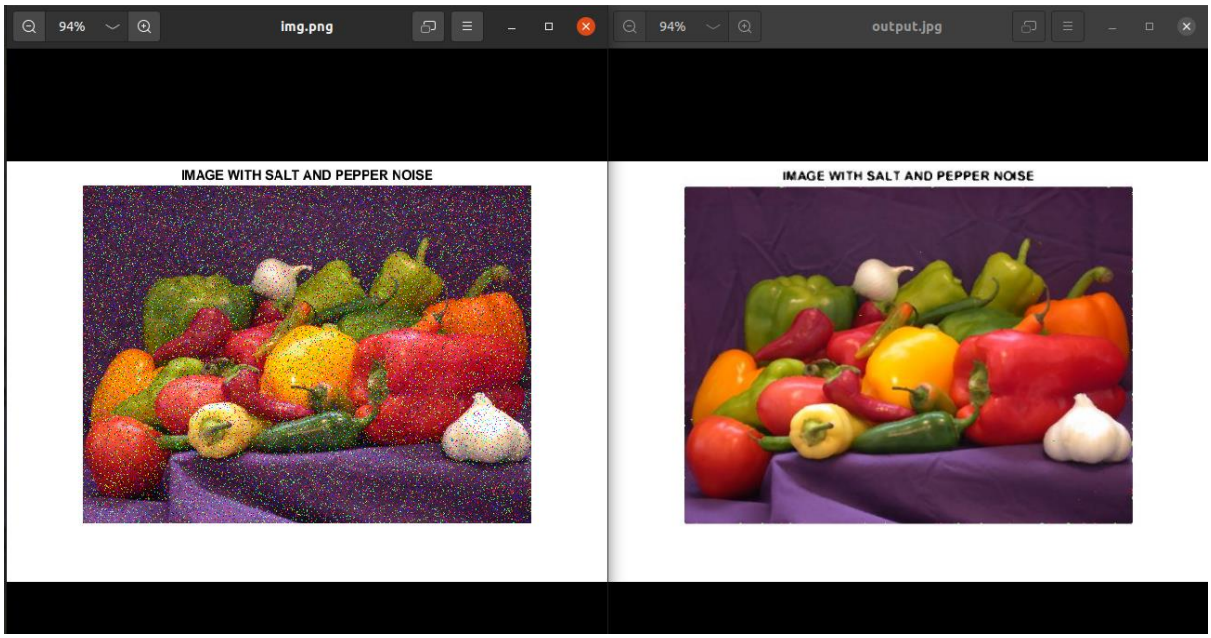
Furthermore, on **the outer class**, I created the main method and inside of the main method I created the ForkJoinPool object to calls its method invoke() to run a pool of threads concurrently using the available processors of the system to run the tasks/threads of the RecursiveAction class I created.

To validate my parallel programming was correct and efficient, I benchmarked it to the serial programs to see if they output the same filter on images and if the parallel programs were faster than the sequential programs when computing larger images in size. Indeed, the difference is evident and will elaborate more on it, please continue reading.

With curiosity, I also used an online source where someone else used a median filter on his image to further verify my program. However, as expected it produced the same results. Here is the soft link: https://www.imageeprocessing.com/2017/10/part-3-median-filter-rgb-image.html.

This is the output I got from the parallel programs' median filter below. The original image is available on the image folder if you would like to verify it too.

Image 5: Image that shows median filter



To **time** the algorithms I used a static long variable called startTime and initialized it to zero. I then created a private static void method called tick() and a private float method called toc(). Inside the tick method I set the startTime variable equal to System.currentTimeMillis() to get the current time the system start to run in milliseconds. Inside the toc method I return (System.currentTimeMillis()-startTime)/1000.0f, subtracting the System current time from the initial startTime. I then used these methods inside all the main methods of my driver classes. I first called the tick() method inside the main method before any execution of the program and then after the main method last statement I call the toc method. Finally, when the programs terminate I print the results of the startTime variable on the console in seconds, the image size dimensions, sliding window size and the number of processors the system has.

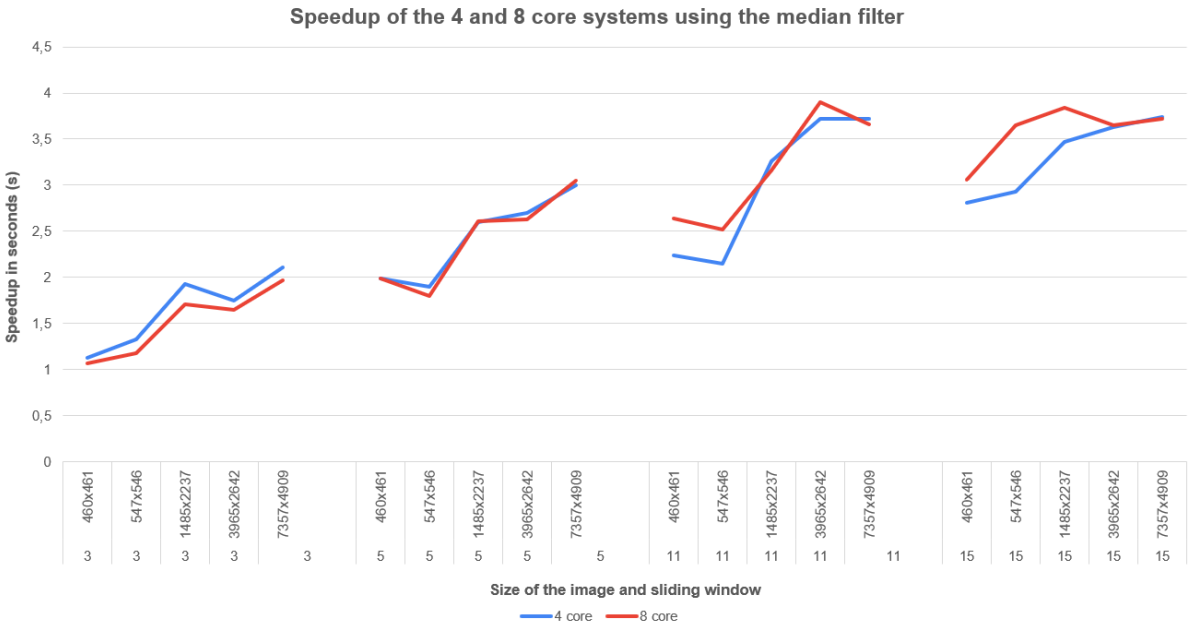
Furthermore, to get the **speedup graphs** in order to determine when one should use parallelism instead of linear programming I divided the linear programs with the parallel programs. I divided the mean filter programs separately and the median filter programs separately too. I used this equation, $\mathbf{T_1/T_p}$. $\mathbf{T_1}$ is the time taken to complete the sequential program and $\mathbf{T_p}$ is the time taken by p processors to complete a task and in our case is the time taken by the parallel programs. Therefore, having generated the time taken by the each of the programs through the Experiment.py file I took the readings for each program and recorded them on Microsoft Excel and divided the respective time output of the programs to generate the speedup graphs. To ensure I have reliable data I repeated the generation mean/ median filter on different number of window sizes and images sizes each 5 times and chose the lowest values I got.

I repeated my experiment 5 times cause I noticed the is a cache effect that I had to account for which was determined by the architecture and algorithm used by the operating system on how to handle the threads.

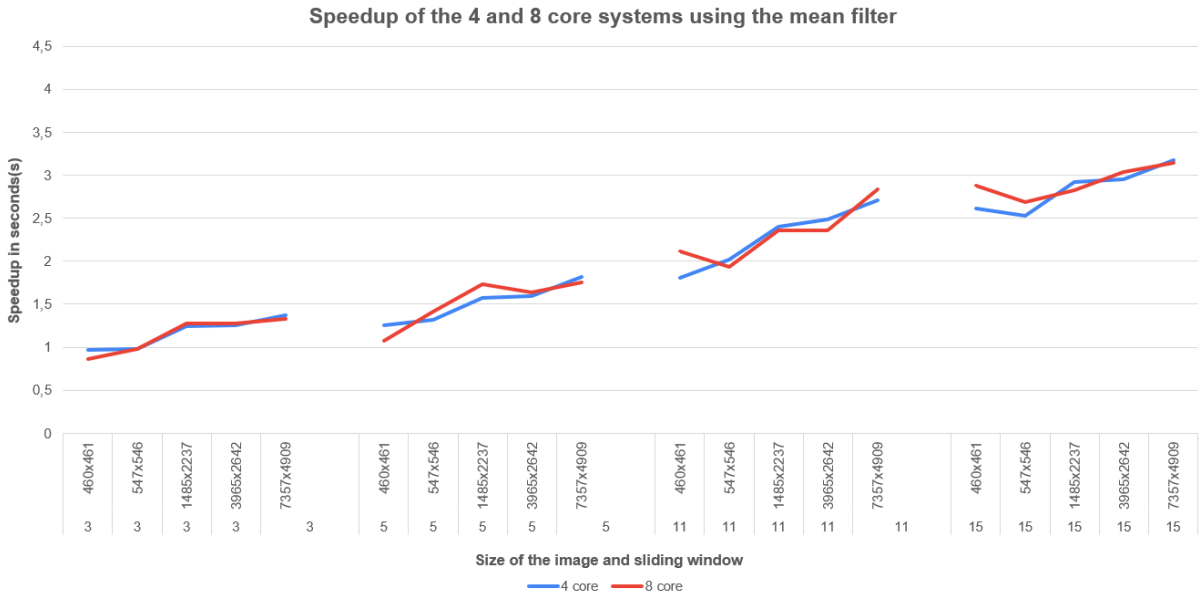
Result section:

Speedup Graphs

Speedup of the 4 and 8 core using the median filter on different image sizes and sliding window



Speedup of the 4 and 8 core using the mean filter on different image sizes and sliding window



From the two graphs above, we can clearly see that the speedup when using parallelism increases with the increase in size of the sliding-window you use to filter the image and with the size of the image that is used. When a 3 x 3 window is used for the mean filter, the 4 core system speedup is more than the 8 core speed up for images less than 550 x 550 in dimension but for dimensions greater than 550 x 550 they are approximately equal in speed. However, when you are using a median filter with a 3 x 3 window the 4 core has more speedup that the 8 core. This happens because the parallel programs require synchronization or waiting for other threads before the main thread can terminate, whereas the serial program does not. This is called thread overhead.

We see almost the same trend when using a window size of 5 x 5 that the speed for the 4 core system is more than of the 8 core system for images less than 500 x 500 in size. For images of 500 x 500 and above in size, the 8 core system using parallelism speedup is more than of the 4 core system. However, surprisingly the speedup becomes almost the same when the size of the image reaches 4000 x 2600 and afterwards the speedup of the 4 core system becomes bit more faster. For a 11 x 11 window, parallelism speedup does not follow any pattern or is not deterministic. However, both systems still show the trend of increase in speed with the increase in the sliding window size used and image size. On the other hand, the 8 core system is faster than the 4 core system almost for every image size used.

One striking point we observe on the speedup graph using mean filter is that the maximum speed up is less than 3,3 seconds whereas the one for the median filter is almost 4 seconds. Another point we can observe is that the lines of the speed up for the mean filter graph looks more linear than the ones of the median filter graphs. This is because the median filter has more computation and the than the mean filter.

For 3 x 3 window, the speed of the 4 core processor is more than the speed of the 8 core system processor. The reason for this is that the waiting becomes more because of having to sort an array first to find the median and each thread having to wait for other threads to terminate before their execution results can be executed by the main thread. This overhead of having to wait in serial programs is not present, the program executes linearly. Furthermore, we observe when using a window size of 5 x 5 performing a median filter that the two systems parallelization speedup look almost the same or the same. However, when a 11 x 11 window is used we see a dramatically huge difference between the speedup of the two system. The one of 8 core is faster than of 4 core at this point and onwards.

When using the median filter parallelism, we see that the image size does not matter anymore when using a sliding window of size of 11 x 11 or more. The 8 core system is

always, if not at most greater than the 4 core system with a greater margin of difference. However, the increase do not seem like a perfect linear increase but show a sign of diminishing return as the sliding-window and image size is increased.

The ideal speed speedup was suppose to be 4 times for the 4 core and 8 for the 8 core system and the 8 core was supposed to be 2 times faster than the 4 core. The constraints to this expectation has to do with the software hardware architecture I was using. The two systems could potentially have different architecture and clock-speed. Moreover, the operating system operations when using threads are not deterministic. It uses its own algorithm. However, the speedup was more prominent when using median filter than the mean filter.

Therefore, the point where someone can use parallelization is when one is processing an image using a window 11 x 11 and or processing a larger image than 1480 x 2200. Above this mark is when you see the benefits of parallelization more. However, there is a point where the speedup benefit from parallelization start to decay and never passes a threshold equal to the number of physical processors you have. Therefore, from this experiment we can see the beauty of parallelization when computing problems that have high computation demand. Moreover, the algorithm you use adds a great deal of benefit in your parallelization in terms of efficiency. Hence, good algorithm and having more processors on your system are ideal for problem with more computational demand as seen in the experiment.

Moreover, for the parallel algorithms I discovered that through experimenting various values of cut-off that using a cut-off of the image size divided by the number of processors your system has and less 1 is the optimum cut-off. This made sense because the system is already running a processor before even running your program. To ensure I got more reliable measurements I ran the experiment 5 times for each window and image size I used and took the lower speeds values. However, because of the architecture factor and the operating system being indeterministic I had anomalies still. This is due to built in algorithm of how the operating system works and the architecture built for the system.

Therefore, in conclusion sequential programming is more viable and efficient in computing smaller problems with less complexity and parallel programming is more efficient and faster for computing larger problems. I would recommend to a person who wants to do mean filter or median filter of sizes 550 x 550 or less and using smaller window of 5 x 5 to use a sequential program because they are bit faster than the parallel programs and use less resources. On the other hand, if you want to process larger images 1480 x 2200 and window sizes 11 x 11 or more they should use parallel programming because they are more faster.