2D Median Filter for Image Smoothing

Jordy Kafwe

University of Cape Town

CSC2002S – Parallel and Concurrent Programming

14 August 2022

2D Median Filter for Image Smoothing

Method

Parallelisation Algorithms

The parallel implementations for both the mean and median filter divide an image into vertical strips. Each vertical strip has the same width and height (height is unchanged from the original image).

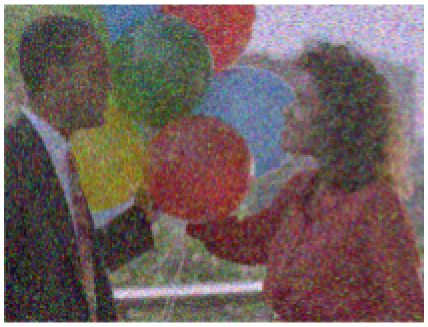
The constructors in both the **MeanFilterParallel** and **MedianFilterParallel** classes have the following parameters: a BufferedImage source, an int start, an int length and a BufferedImage destination. The BufferedImage **source** is the image to apply the filter to and the BufferedImage **destination** is the destination image to write the results to. The integer **start** is the index (in the width of the original image) to start processing. The **length** integer is the width of the region to process. **Start** is capped to a minimum of $(window\ width\ -\ 1)\ \div 2$ and **length** is capped to a maximum of $width\ -\ neighbouringPixels\ -\ start$. This is done to ensure that a full window can always fit.

The **compute()** method in both classes either performs the smoothing directly or splits it into two smaller tasks and then executes them in parallel. It uses the length (width) of its instance to determine whether a task is small enough – the sequential cutoff. In other words, if the width of this region is small enough, 200 or less pixels, directly apply the filter to all pixels in this instance's region.

Both parallel algorithms make use of the fork/join framework. They inherit from **RecursiveAction** as their tasks return no result; instead, writing the values of the filtered pixels to the destination buffered image. The tasks are performed by the default **ForkJoinPool** which has parallelism equal to the number of logical processors on the machine where it is executed. This should likely ensure good performance regardless of the underlying architecture.

Algorithm Validation

Both the mean and median filters are often used to remove noise from an image. This is also known as smoothing an image. With the median filter having the advantage of preserving edges better after smoothing an image. It can be visually seen whether a filter works because it should remove the noise from an image. Therefore, the image with RGB noise was used to validate the correctness of the serial implementation of the filters. Both filters work because, by inspection, they appear to remove noise from the image – the median better than the mean.


Original Image with RGB noise

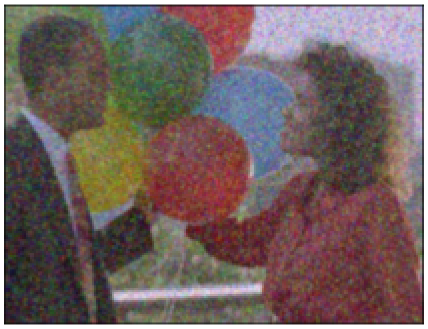


Image with mean filter

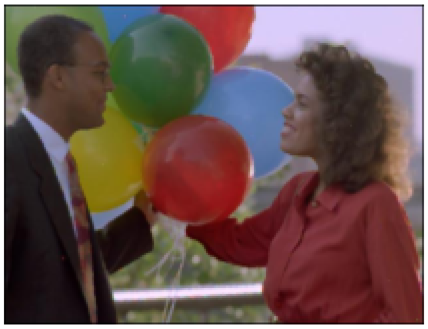


Image with median filter

To validate the correctness of the parallel algorithms, parameterised JUnit tests were written (in the **FilterTests** class) which compared the serial and parallel output of filtered images which used the same input image, window width and filter. The tests were compiled with the make rule **compiletest** and run with the make rule **runtest**. Arrays of integer pixels in the default RGB colour model for both images were compared using **AssertArrayEquals()**. If the tests passed, then every pixel in the serial and parallel images corresponded with each other. All the tests that were performed passed.

Timing

Only the core work of applying the filter was timed for both the serial and parallel programs. This was done using **System.currentTimeMillis()** which returns a long representing the current system time in milliseconds. This was done by calling **System.currentTimeMillis()** before and after the method that needed to be benchmarked and calculating the difference. In the serial programs, the execution of the **apply()** method was timed. In the parallel programs, the execution of the **smooth()** method was timed.

For example:

```java
long startTime = System.currentTimeMillis();
BufferedImage filteredImage = medianFilter.apply(inputImage);
long endTime = System.currentTimeMillis();
```

Optimal Sequential Threshold

A Python script was written to run both parallel programs with 3 images of various sizes (one small, one medium and one large) and a window width of 9. 5 different sequential cutoffs were tested per image as well – starting at 200 and incrementing by 200 each time until 1000. Each program was also run 5 times per image for every cutoff in order to give the framework a chance to "warm up" so as to get the fastest possible execution time. This is because the fork/join framework is written in Java and the compiler needs time to decide this code is performance critical and optimise it. The smallest value of the 5 was used. The algorithms appeared to perform well with lower sequential cutoffs as the image was split into many smaller tasks that kept the CPU saturated. The results can be found in the result folder and in the **sequential_cutoff** directory.

Machine Architectures

The programs were benchmarked on 2 computers:

1. A Macbook Pro with an Intel Core i5 CPU that has 2 physical cores, 4 logical cores, 2.50 GHz clock speed and 16 GB of RAM
2. Nightmare, the departmental server, which has an Intel Xeon CPU with 4 physical cores, 8 logical cores and 24 GB of RAM

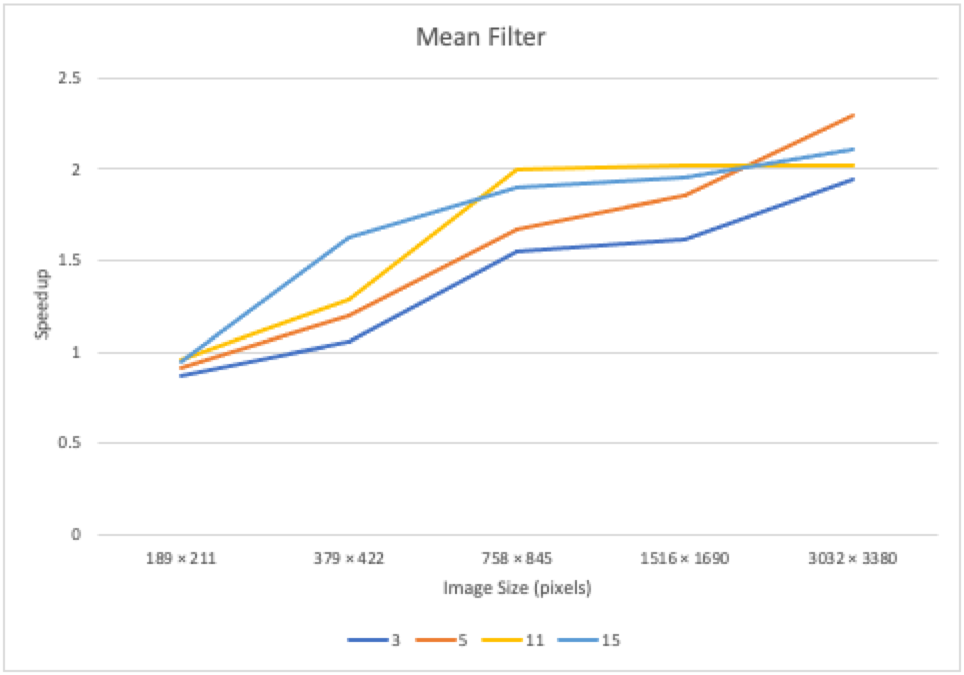
Problems/Difficulties

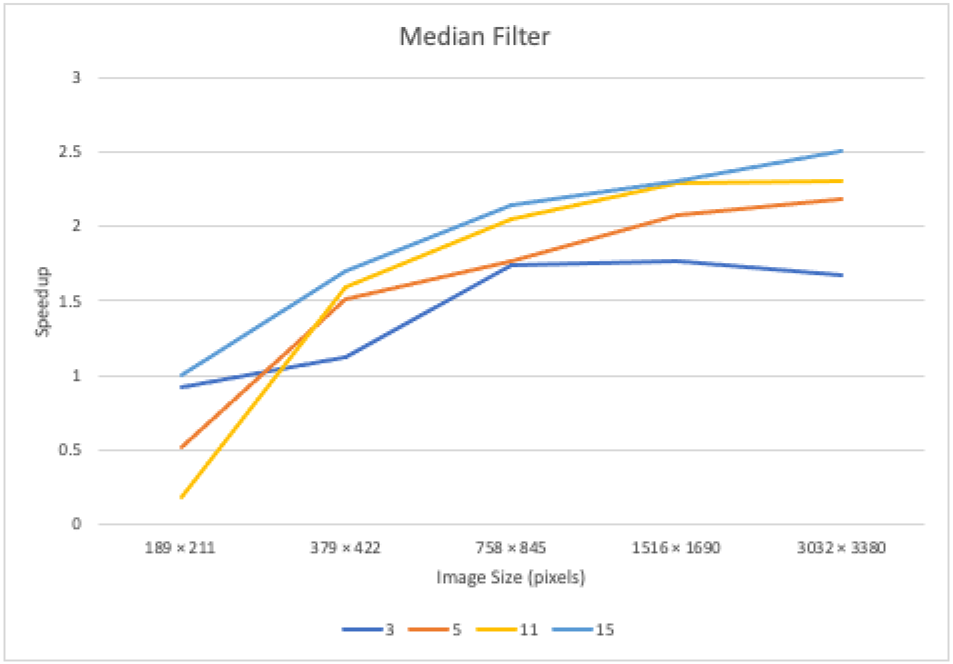
A problem that was encountered with testing on Nightmare was that the SSH connection kept breaking in the middle of the benchmarking which would stop all the processes that were started to conduct the benchmarking. **tmux** was used to fix this. This was done by starting a process inside a tmux window, which allows one to log off from the remote machine and still keep their processes running inside tmux. In addition, one can come back and attach their tmux window and monitor their process' progress.

Results

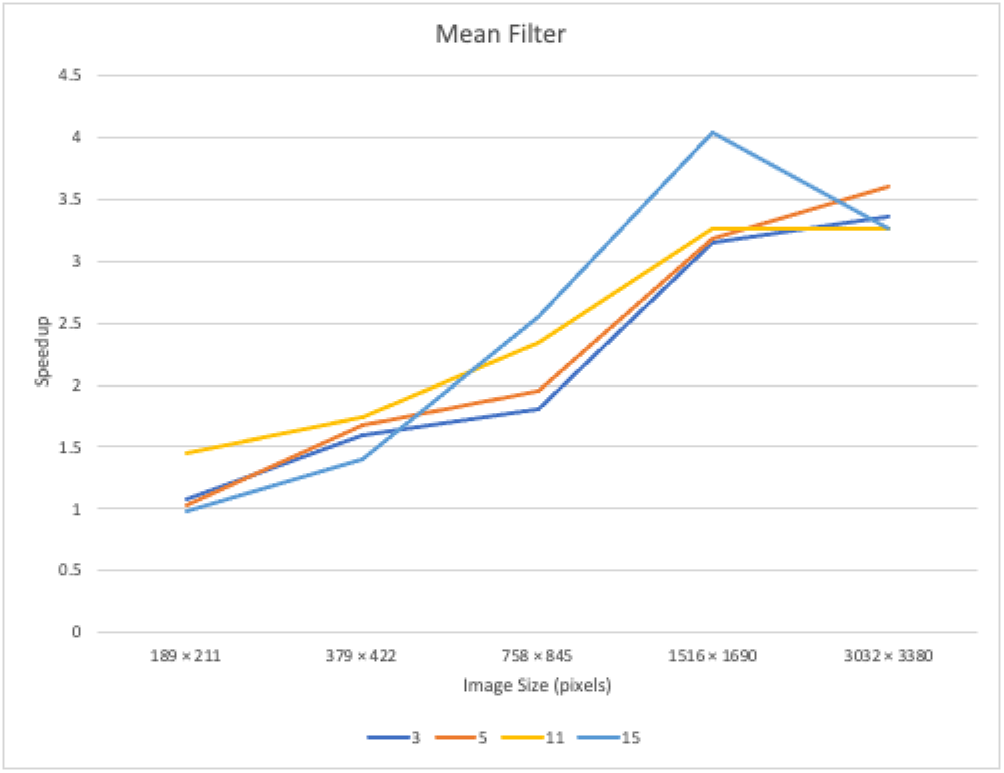
(key in all of graphs is of the window width)

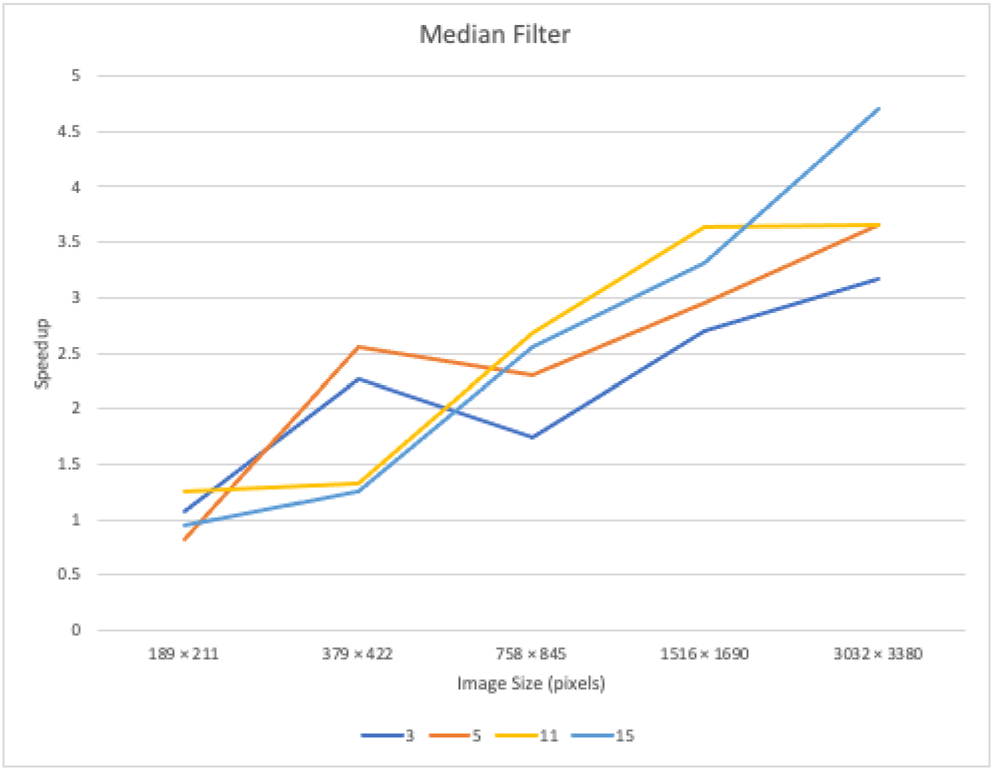
Macbook





Nightmare


Mean Filter
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
Speed up
189 × 211
379 × 422
758 × 845
1516 × 1690
3032 × 3380
Image Size (pixels)
3
5
11
15


Median Filter
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
Speed up
189 × 211
379 × 422
758 × 845
1516 × 1690
3032 × 3380
Image Size (pixels)
3
5
11
15

Discussion

Both algorithms performed better with a lower sequential cutoff across all image sizes. This is because the image was divided into many smaller tasks that could happen in parallel and keep the CPU saturated. When the sequential cutoff was too high (very close to or bigger than the width of the image), it would result in tasks of bigger sizes occurring sequentially as the image would not be divided enough. This would also result in not all the cores being utilised – or them being left idle for too long. The optimal sequential cutoff that was decided was ideal 200.

The parallel programs performed well for medium and large images (images with width of 800 and above) this is because the image could be split into enough tasks to be processed in parallel and keep the CPU saturated. More importantly, the benefit was seen more for bigger images because those same images run with the sequential program took anywhere from double the time to more (depending on the architecture) than their parallel counterparts. Double the time, in this instance, were times like 7 minutes. Whereas a parallel algorithm could process that image in 2.5 minutes.

The parallel algorithms did not perform well when the image was too small because either the image just ended up getting entirely processed in serial or did not get split into enough pieces that the additional work of synchronising and creating parallel tasks ended up making it slower than the serial program.

The maximum speedup obtainable, in theory, is n with n being the number of processors on the computer. The results showed that for big enough problems this speedup was achievable for both parallel programs. For example, the Macbook, with 2 physical and 4 logical cores, achieved a speed up of between 1.8 to a little over 2. Nightmare, with 4 physical cores and 8 logical cores, achieved a similar speedup of between 3.5 to a little over 4 for large enough datasets. The reason the speedup is close to the ideal is because only the core work of applying the filter was timed – no blocking operations such as reading and writing images to slow down the speedup. It could also be because both programs were run in controlled environments which ensured that they did not have to share the CPU with other programs' processes.

The measurements were reliable because they were taken at times with low traffic for Nightmare and all other programs were switched off on the Macbook so that the programs had the entire CPU to use – taken in as a controlled environment as possible. Activity on Nightmare was checked using the **who** command to see how many users were connected to the remote machine and **top** to see if they were running any CPU intensive processes.

Conclusion

Parallel programming (and the extra effort involved) is worth it when the dataset is large and the problem is computationally intensive enough that the serial version of the program takes long. This is because with an easily parallelisable algorithm a speedup close to the number of processors (linear speedup) on the machine can be achieved. This is the case with the filters as when the image size and window sizes increase a greater number of calculations need to be performed to smooth the image.