SMBTHA002: Thabang Sambo

CSC2002S Assignment PCP1 2022

Parallel Programming with the Java Fork/Join framework: 2D Median Filter for Image Smoothing

METHODS DESCRIPTION:

The program takes in command-line parameters. The user is required to enter the image name, including the file extension; then the output image name, including the file extension; and finally take in the window width which is a positive odd integer that greater or equal to 3.

READ IMAGE

```
BufferedImage newPic = new BufferedImage
```

- Used the method to get the image details and to read and write an image

```
public static BufferedImage data( BufferedImage picture)
```

- Takes in the image and evaluates its parameters, we take its height and width assign it to a variable and store the data pixels into an array so that we can access them and manipulate them

READ AND CHECK WINDOW WIDTH

```
protected static void window(int windowWidth)
(windowWidth <= 21 && windowWidth >= 3 && windowWidth % 2!=0);
```

- Takes in the window width and checks if it is valid or not

TIMING

```
long speedCount = System.currentTimeMillis();
long stopCount = System.currentTimeMillis();
     System.out.println(Math.abs(speedCount-stopCount)+"
seconds(m)");
```

- used `System.currentTimeMillis()` to time my program to be able to record the start of the code and the end and record the time that elapsed to be able to get how long the code takes to process the information

FORK JOIN

```
MedianFilterParallel fb = new
MedianFilterParallel(OriginalPic, 0, OriginalPic.length,
newPic ,windowWidth);
ForkJoinPool pool = new ForkJoinPool();
```

- for both mean and median methods of filtering I used the same implementation for using fork join to accept the image inserted and return the picture that we have filtered

CODE VALIDATION AND ARCHITECTURE

Validation:

I would not say my code is correct because in parallel coding and experimentation there is a margin of error and application that can be wrong but the way I went about seeing if my code does at least function was I compared it to the serial code, the differences are in the way we implement and use a different method which is fork join so in terms of using different approaches that are very complex in parallel the serial code is almost simple and dynamic by deriving the parallel code from the serial it allowed me to come the parallel code that I have and that is how I came to the conclusion that it works because of the functional comparisons of the two.

Machine architecture used:

I ran the code on at least 3 different types of machines, I ran it on a –

Intel Core i3 1005G1, Processor 8GB RAM, with 2 cores

Intel Core i5 7010 Desktop i5 8 GB 240 GB with 4 cores

Linux lab machine with at least 8 cores

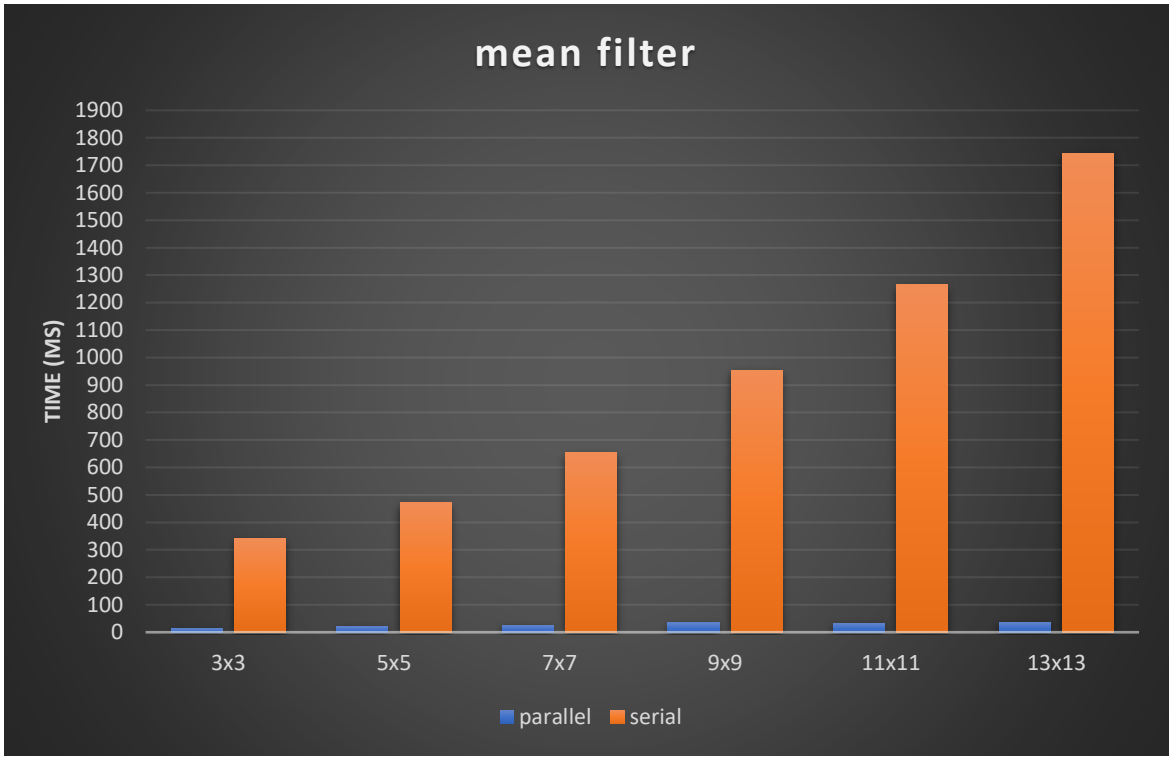
Problems faced:

- the machine I am running the code is slow and takes too much time to load the code
- the larger the pixels the more it takes to finish
- running the parallel program affects the locality
- with multiple core machines one core sometimes ignores a particular memory location, the version of the memory location cached by another core gets ignored as well

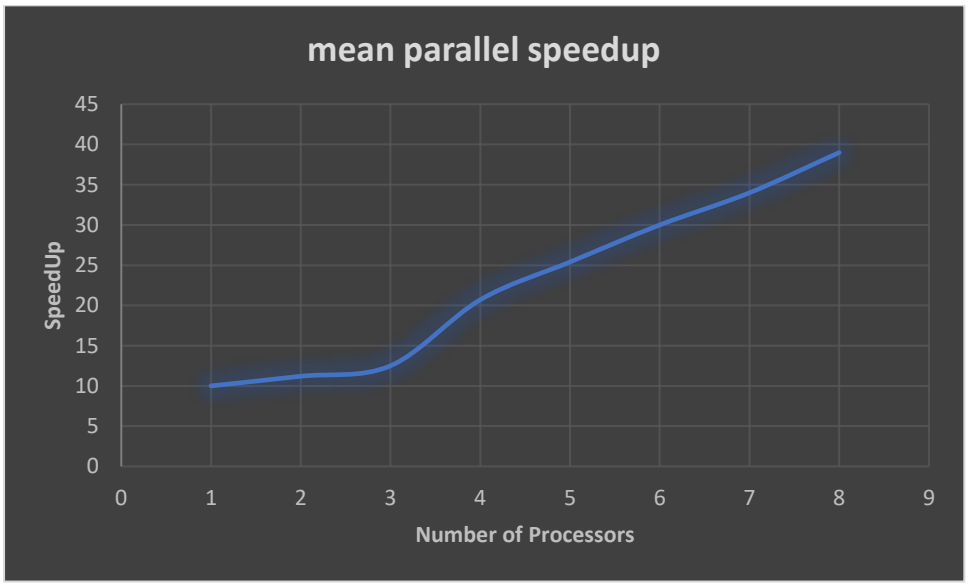
EXPERIMENTAL RESULTS:

Mean Filter

For the experiment, I used an Intel Core i3 1005G1, Processor with 8GB RAM, with 2 cores, an Intel Core i5 7010 Desktop i5 8 GB 240 GB with 4 cores and a Linux machine that has 8 cores. to get the processing time of a 275x183 image and using a window size of 3x3, 5x5, 7x7, 9x9, 11x11 and 13x13 with a serial mean filter and parallel mean filter. Gathered the run from each machine and combined the data to get the average time it executes on the different machines. The results below are the combined average of the execution times I got from running it on different architectures.

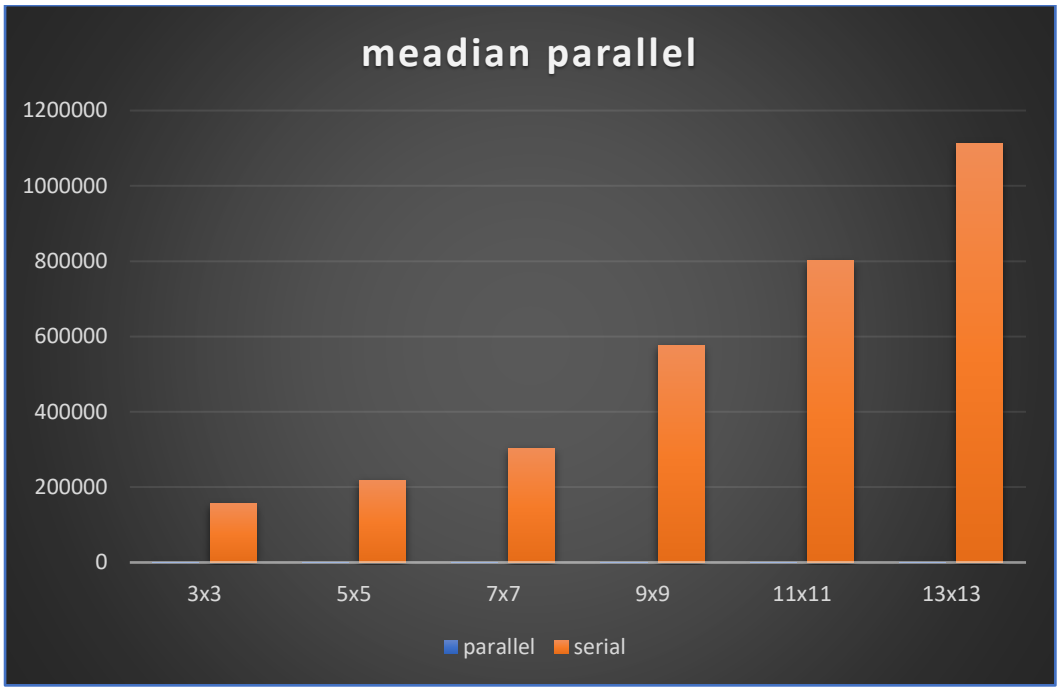


Speed up for a parallel mean filter

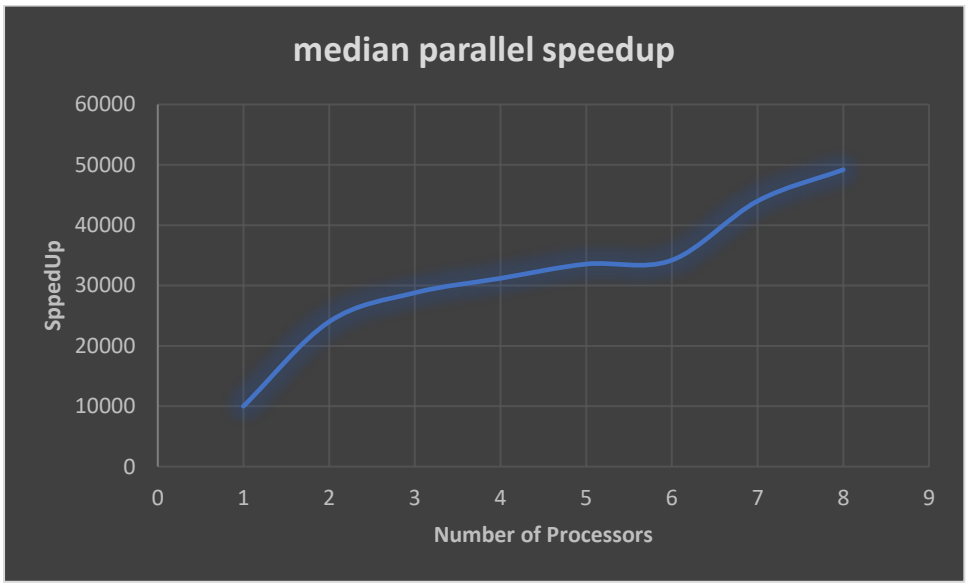


Median Filter

For the experiment, I used an Intel Core i3 1005G1, Processor with 8GB RAM, with 2 cores, an Intel Core i5 7010 Desktop i5 8 GB 240 GB with 4 cores and a Linux machine that has 8 cores. to get the processing time of a 275x183 image and using a window size of 3x3, 5x5, 7x7, 9x9, 11x11 and 13x13 with a serial median filter and parallel median filter. Gathered the run from each machine and combined the data to get the average time it executes on the different machines. The results below are the combined average of the execution times I got from running it on different architectures.



Speed up for a parallel median filter



DISCUSSION

For both instances I used a sequential cut-off of 50000 through multiple runs and different data sizes used, it was the most suitable value because it allowed me to run a variation of data while maintaining optimal performance through different inputs. The programs perform well with a small sample of data, the graphs above show that - as our window width increases the amount of time taken to complete the program increases, with the results obtained the conclusion is that the parallel programs perform at their best when the window size or width is small, they seem to produce the fastest time between a window size of - 3x3 to 7x7. Using these different ranges of results I was able to calculate the speed up for different cores sizes, the maximum speed up obtained from an 8 core machine for a parallel mean filter was approximately 40 and for a parallel median filter was approximately 50000, there's not much of a difference between the two set of results with a slight deviation that could be caused by the machines used, the code structure, the amount of data background data and the amount of time spent between runs, on some recordings there were some anomalies that affected the results produced and these slight deviations could be the cause for the data to skew from the expected results.

CONCLUSION

I compared a serial and a parallel code and discovered that a parallel code has more positives in its favour, because it uses threading because all the threads in a process would share a resource such as memory or data. One application can have different threads within the same address space because of the open sharing of data. On a machine that has more than two processors, each thread would run on a separate processor in parallel utilizing multithreading giving an increase in the concurrency of the system showing the difference between single processor system and a multiprocessor system, where only one process or thread can run on a processor at a time resulting in a decrease in execution time, so I can conclude and say that it is more efficient to use parallel programs to tackle systems in which you need to process a large sum of data and you want your code to be effective while doing it.