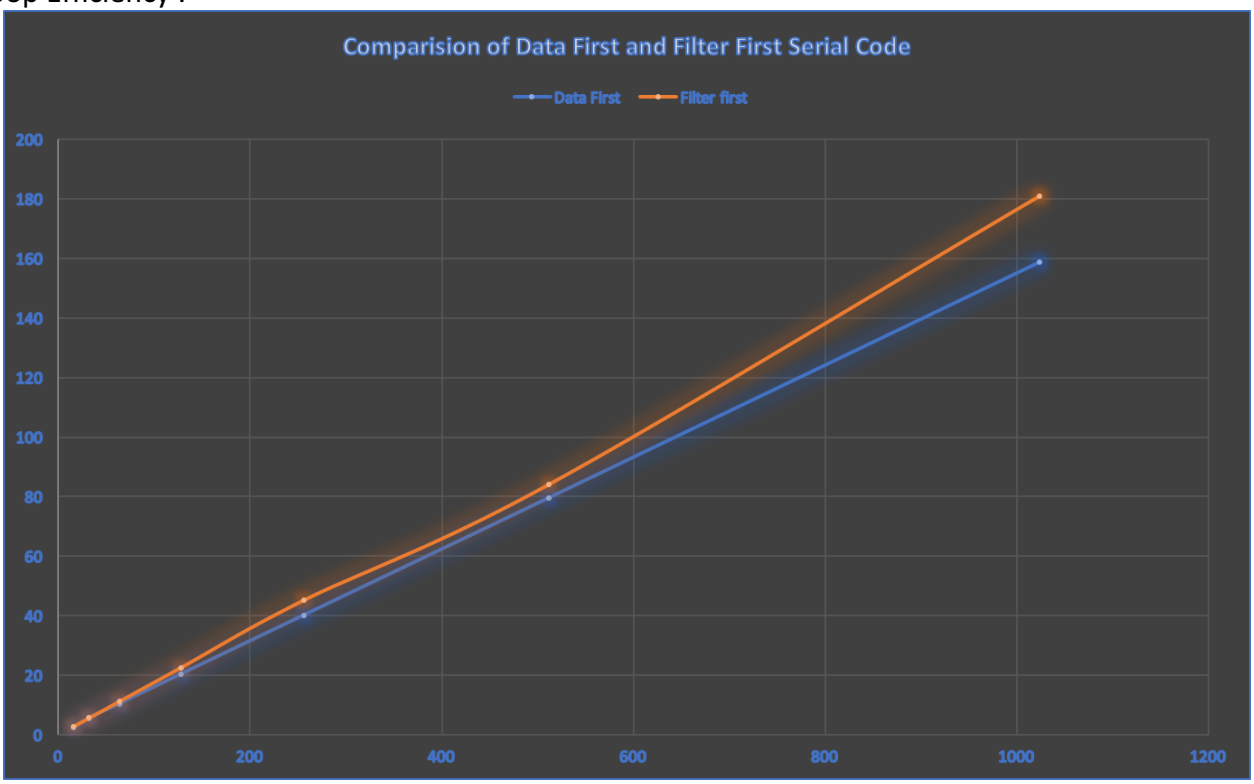
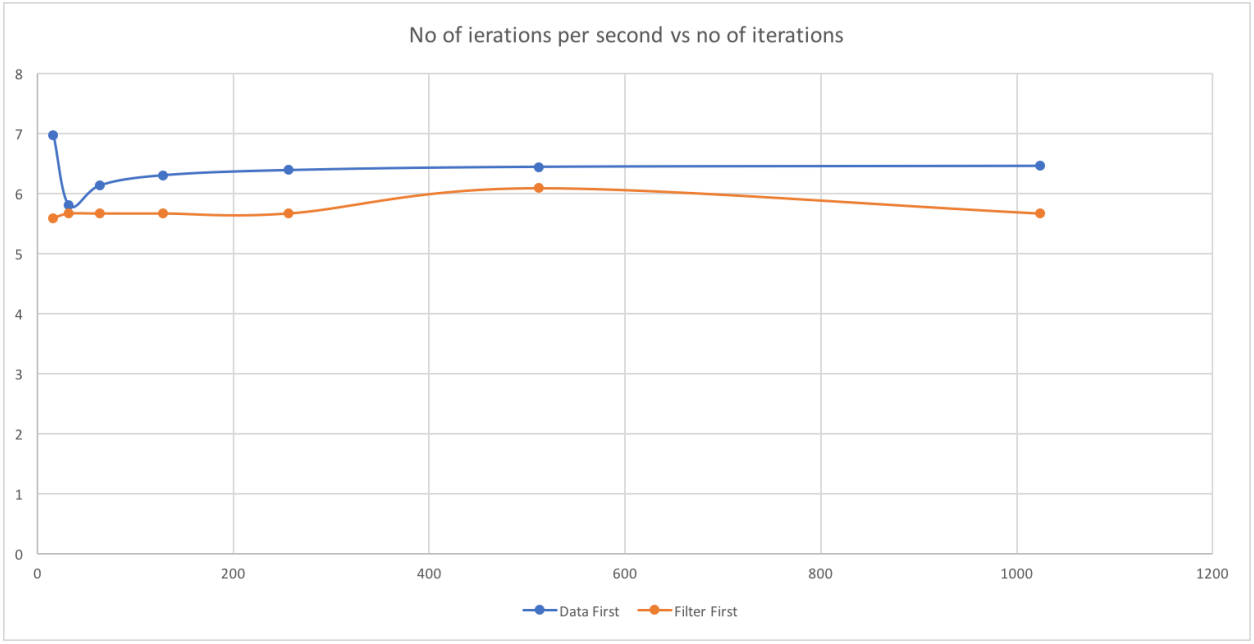
Q 1 : Loop Efficiency :

a)



x axis : no of iterations y axis : time taken (sec)

b)



X axis : no of iterations Y axis : No of iterations per second (no of iterations/time)

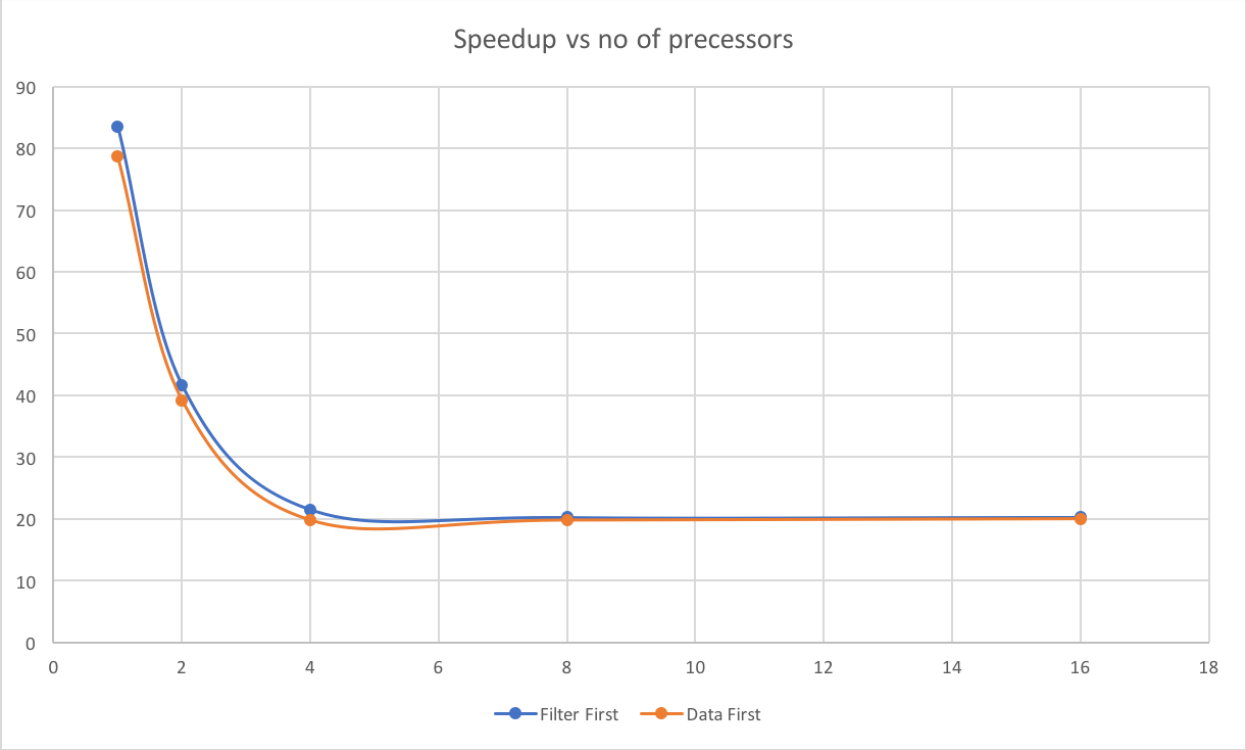
c) Filter first is more efficient.

This is because of cache memory that is picked up each time the loop picks up any memory location. When filter first is in the outer loop, the data array element is stored in the cache and easily accessed.

d) Relative performance almost remains constant with filter size. This is because it is a serial implementation of the loops and no of iterations don't do much.

Q 2 : Loop Parallelism :

1. .Functions created
2. a)



X axis : no of processors Y axis : time taken to run 512 Filters(secs)

b) We got a speed up which maxed at 8 processors. Increasing form 1->8 processors and decreasing thereby. Speedup tailed off after 8 cores. We used a computer with 8 processors and thereby when we divided our loop into more than 8 parts for distribution, it did not have any effect. Instead it increased overhead to some extent.

3.

a) Data First application of the code was faster in parallel version. This is because it had less overhead as compared to a data first loop. There is more restructuring going on in this form.

b) For processors = 2, p=1.000;
Procc=4, p=0.989149;
Procc=8, p=0.865364;
Procc=16, p=0.807453;

c) I would attribute the non ideal behavior to startup costs. To start with the compilers need to distribute the loop variables to different processors and this will take up time. This is specialy a problem since we are dealing with large loops with big chunks of memory shared and private.

Q 3 : Optimisation :

a) Loop Unrolling : Loop unrolling improved the runtime in both filterfirst and Datafirst version of the parallel function. This is because it reduced the overhead of parallel loop distribution.
Time reduced to 16.523648 from 20.2711 for 8 processors.

Once unrolled the loop, we make the no of for iterations to half and increase the number of operations in each loop iterations, this means there are only half the number of iterations to be divided into different processors. Each time the task is divided into processors, there is some overhead attached to it that is counted in the runtime. Reducing the number of iterations reduces this overhead.

b) Custom Scheduling :
Dynamic Scheduling : This doesn't change the time much. This is because each task inside the for loop is almost equal time taking process. Dynamic Scheduling is useful in case when there is uneven time associated to each task which is to be divided among different processors. It dynamically allocated the task to each processor as and when it is finished with previous tasks. But since the tasks are almost of equal length, there is no loss of processor time in static case as well. So the runtime should not be effected.
Dynamic Scheduling : 20.192
Normal 8 core : 20.2711
Static Scheduling with specific block : This increases the runtime for small chunks and doesn't matter for larger block sizes. Static means that the processors are assigned jobs in chunks once. That means that if there had been any skew, the runtime would suffer drastically, but since the jobs inside for loop are almost equal size, this method only adds up to the overhead costs.
Dynamic Scheduling with chunks : This is similar to the normal dynamic Scheduling situation with 1 difference. As the chunk size increase, this goes back to normal default static situation, which is the optimal for this algorithm.