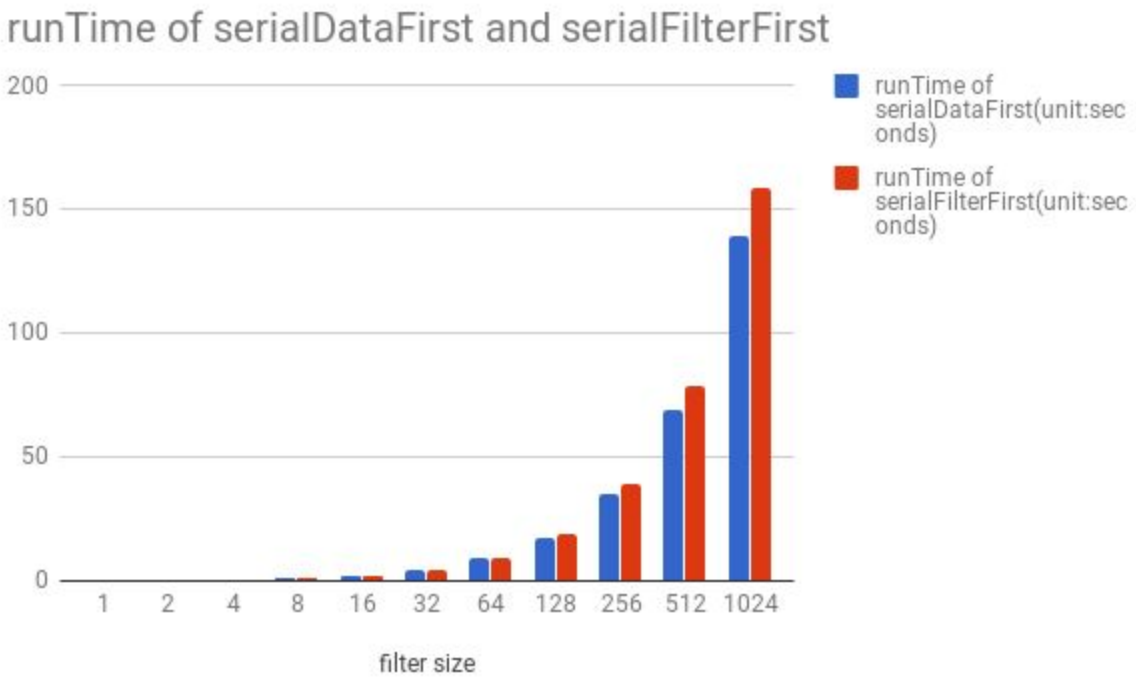
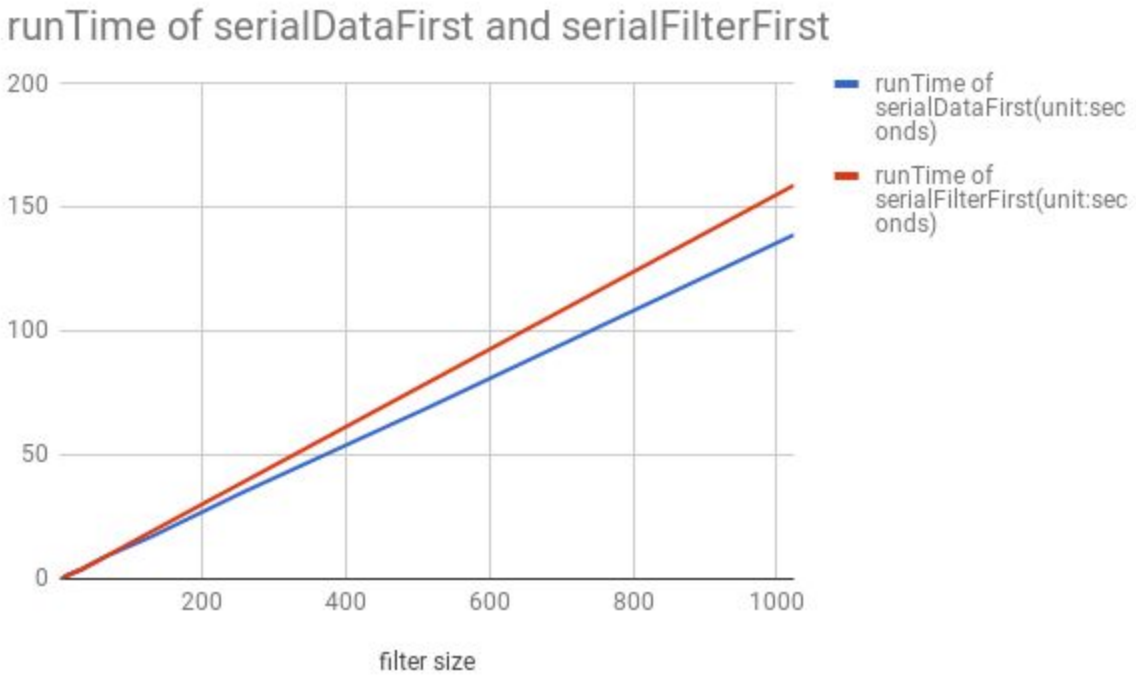
Write Up of Project 2

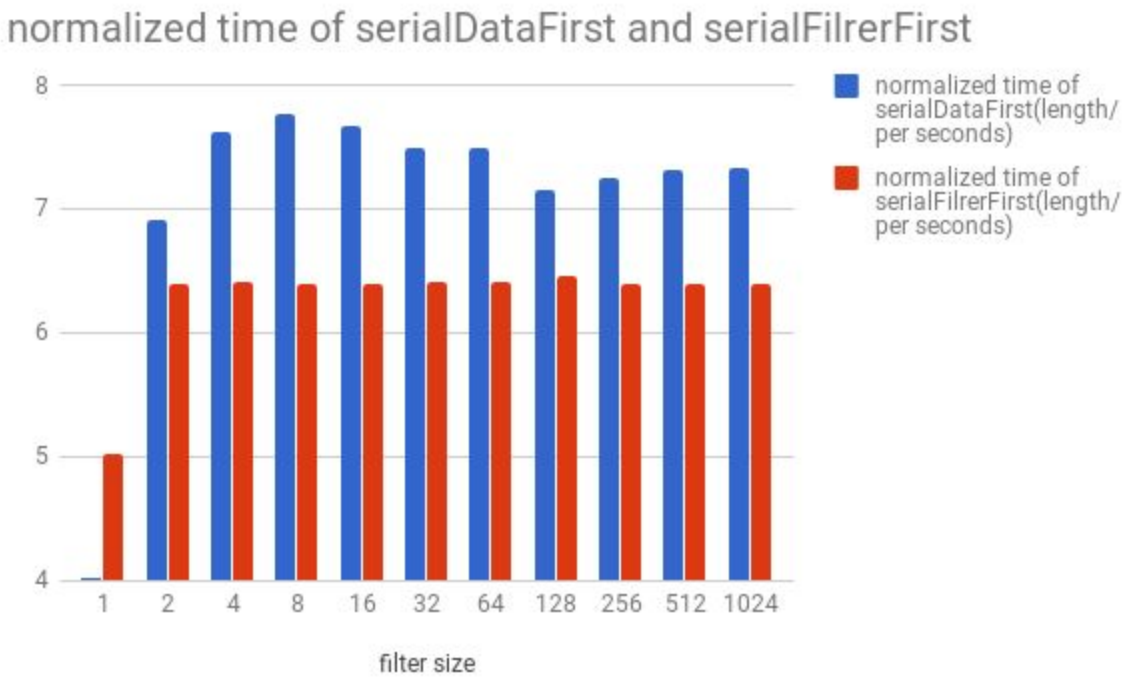
Part I: Loop Efficiency

(1) I plotted two kinds of charts for you:





(2) As for normalized running time charts, I compute the how many filter size can be handled per second:



(3) It is more efficient to have data in the outer loop than to have filter in the outer loop. The reason is that temporary locality. In the system, normally the system will store recently used data in the cache, and pop out least recently used data out of cache. Thus, when we need to get array element, if we can directly cache hit targets, it will be super fast compared to get data from memory. Thus, as for serial data first model, cache hit happens more frequently than serial filter first model (due to the length of data_len is much larger than the size of filter_size). Thus, that's why serial data first method is more efficient.

(4) Based on the graph in the question 2, I think the performance firstly becomes better as filter_size grows, and then the performance looks like keep fluctuate around certain value. I think there are two major reasons, first is that the system needs to take some time start the program. Then, at the beginning, the cache didn't have any desired target element in it, so the program is not able to directly find elements in the cache, so it needs to take some time to get elements from memory and stores them into the cache temporary so that cache hits can happen later on. Later on, as for program runs for a while, some elements have been stored in the cache, and then cache hits will happen frequently, so the performance becomes better. After the system thoroughly used the power of cache, the performance becomes steady.

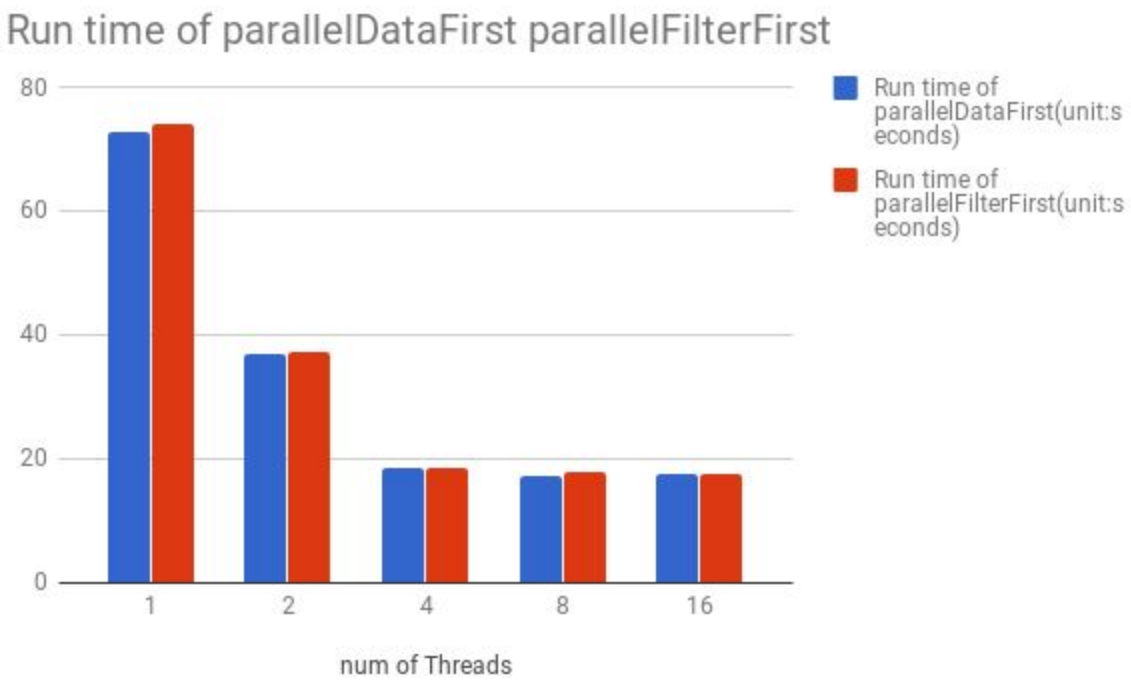
Part II:Loop Parallelism:

(1) Here is running result after implementing parallel functions:

Serial filter first took 17 seconds and 984127 microseconds. Filter length = 51

Data check failed offset 1
Serial data first took 17 seconds and 157792 microseconds. Filter length = 512
Data check failed offset 1

(2)



As we can see as num of threads grows, the running time decreases in both
Of two cases. Also, when num of threads is smaller than 4, as the number of threads grows twice, the running time shrinks into the one half of its original running time. However, when num of threads continues to grow, we found that the running time didn't decrease very much, and It looks it is stuck at some certain value. It is because of the number of cores and other kinds of resource in this system is limited, the system has used all of available resource. The program runs on Amazon High-CPU 8-core instances (c5.2xlarge), and it can help to increase the speed of program compared to other kinds one-core system when handling multithreading programming and parallel programming.

(3) Question a: In generally, the fastest version is the case where number of threads is 16. It is because the system tries to use all of cores in the system, and try to used all of 16 threads to parallelize data processing and parallelizable coding part.

Question b: In this case, from numThreads = 1 to numThreads = 2, we can calculate P = 0.9977 based on Amdahl's law. Then, when we changes from numThreads = 2 to the numThreads = 4, based on the Amdahl's formula, we can get $37.33/18.6 = 1/(1-p+p/2)$, then p = 1.006, then when we change from numThreads = 4 to numthreads = 8, then P = 0.0668.

Question c: First of all, startup costs, when we need to start our program, the initial starting time couldn't contribute parallelism speed up, so it is not an ideal speed up in this case. Also, when multiple threads need to get access to shared resource in this program, it still couldn't

contribute to parallelism speedup since the system sometimes need to use mutex or critical section technique to protect shared resource so that different threads are not able to affect with each other. This kind of inference will also not contribute to speedup. So, when we have too many threads, this kind of inference will decrease performance of our parallel program. Also, we have skew issue, when threads will generate sub-threads, then this process still needs to take some time, so it also will not improve parallelism speedup. Thus, skew is not an ideal speedup.

An optimized version:
We can just focus on filter_size = 512, and try to optimize the whole codes to see their performance.
Case 1: Original Version
In this case, without using parallelism technique, then we can find that the running time of serial data first model is 69 seconds and 986976 microseconds, and the running time of serial filter first model takes 79 seconds and 986531 microseconds.

Case 2: Add #paragma omp parallel for around the outer for loop
In this case, we find that the running time of serial data first model takes 17 seconds and 157792 microseconds, and the serial filter first model takes 17 seconds and 984127 microseconds. The reason tries to parallelize for loop and multi-threads to parallelize programming based on available resources of the system such as the number of cores.

Case 3: Use loop unrolling try to speed up
In this case, I modified original parallel codes into the following codes:
```
 #pragma omp parallel for
 for (int y=0; y<filter_len; y++) {
        /* for all elements in the data */
        for (int x=0; x<data_len; x+=4) {
        /* it the data element matches the filter */
        if (input_array[x] == filter_list[y]) {
        /* include it in the output */
        output_array[x] = input_array[x];
        }

        if(input_array[x+1] == filter_list[y]){

        output_array[x+1] = input_array[x+1];
        }

        if(input_array[x+2] == filter_list[y]){

        output_array[x+2] = input_array[x+2];
        }
```

```
        if(input_array[x+3] == filter_list[y]){
        output_array[x+3] = input_array[x+3];
        }

        }
 }
```

and we also change into:

```
 #pragma omp parallel for
 for (int x=0; x<data_len; x++) {
        /* for all elements in the filter */
        for (int y=0; y<filter_len; y+=4) {
        /* it the data element matches the filter */
        if (input_array[x] == filter_list[y]) {
        /* include it in the output */
        output_array[x] = input_array[x];
        }
        if (input_array[x] == filter_list[y+1]) {
        /* include it in the output */
        output_array[x] = input_array[x];
        }
        if (input_array[x] == filter_list[y+2]) {
        /* include it in the output */
        output_array[x] = input_array[x];
        }
        if (input_array[x] == filter_list[y+3]) {
        /* include it in the output */
        output_array[x] = input_array[x];
        }
        }

 }



```

Then, we have following result:

Serial filter first took 14 seconds and 129160 microseconds. Filter length = 512

Data check failed offset 1

Serial data first took 13 seconds and 990352 microseconds. Filter length = 512
Data check failed offset 1

We can see after using loop unrolling, serial filter first has significantly improved its performance (from 17s t 14s running time). Also, serial data first model's running time changed from 17s to 13.99s, so using loop unrolling can help us increase performance of our program. The reason is that loop unrolling has reduced or eliminated instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration; reducing branch penalties; as well as hiding latencies including the delay in reading data from memory (referenced some explanation from wikipedia).

Case 4: add scheduling or dynamic in our parallel codes
Here are how I have changed the codes, I just change '#pragma omp parallel for' into '#pragma omp parallel for schedule(dynamic)', and it did improve my running time, and the running time is 12-13s in both serial data first model and serial filter first model. In theory, the static schedule can improve running time compared to static schedule, since the system will assign a iteration to the threads that are free. Thus, compared to static schedule, it can avoid the situation whre some threads need to take much longer time to finish their tasks even though other threads have finished their tasks at earlier time. However, sometimes using static schedule will have some overhead if we use dynamic model. It is because the threads must stop and receive a new value of the loop variable to use for its next iteration at every iteration if we use dynamic schedule very frequently. Luckily, we didn't find this kind of situation in our cases.