K-mer Counting

Aryan Gupta (2020101091)
Anmoldeep Kaur Dhillon (2020101085)

Problem Statement

- In bioinformatics, k-mers are substrings of length k contained within a biological sequence.
- So, a sequence of length L will have L-k+1 k-mers and n^k total possible k-mers, where n is number of possible monomer.
- For a given K we count the number of all occurrences of K-mer in the sequence.
- K-mer counting is very a simple task if we do not care about efficiency , however, that efficient execution of this task, with reasonable memory use, is far from trivial.

Applications

Before looking at how to solve the problem let’s look at why this problem is so important to solve:

- In sequence assembly, k-mers are used during the construction of De Bruijn graphs.
- Beyond being used directly for sequence assembly, k-mers can also be used to detect genome mis-assembly by identifying k-mers that are overrepresented which suggest the presence of repeated DNA sequences that have been combined.
- k-mer frequency and spectrum variation is heavily used in metagenomics for both analysis and binning.

Problem Statement

- In this project, we just deal with DNA sequences which are represented as string with 4 alphabets (monomers) : A, T, C, G . So there will be 4^k total possible K-mers.
- Since, the problem highly depends on the length of sequence given and the value of k, we divided our problem into 3 cases:
  1. Small sequence size (less than 256 MB) and k<13.
  2. Small sequence size (less than 256 MB) and k<13.
  3. k>=13..

Compute Configuration

MODEL	Intel® Core™ i7-9750H CPU @ 2.60GHz × 12
Generation	9
No of Cores	6
Hyper - Threading availability	YES
Thread per core	2
Total CPU	12
Min Frequency	2600 Mhz
Max Frequency	4500 Mhz
SIMD ISA	Vector ISA
CACHE SIZE	L1d Cache : 192 KB L1i Cache : 192 KB L2 Cache : 1.5 KB L3 Cache : 12 KB
MAIN MEMORY BANDWIDTH	41.8 GB/s
Theoretical Peak FLOPS	216 GFlops/sec

Correctness Evaluation Approach

We wrote a brute force tester program so that we are certain the output is correct although it might not be very efficient . To check if our output is correct , we used the linux diff command utility to check if the output generated by the checker program and the program we wrote matches .

Performance Evaluation Framework

We used Execution_Time as a metric to compare various implementations . We used the simple yet efficient approach of the tick and tock function as was used in the BLAS implementation .

It allowed us to easily compute the execution time for whatever section of code we wanted without any hassle .

Datasets Used

For our initial testing , we generated random strings of 256 MB and 2GB consisting of characters A,G,T and C .

Later , we tried to test on the datasets given to us and also some sequences (in FASTA format) which we used in the Biology course we had this semester .

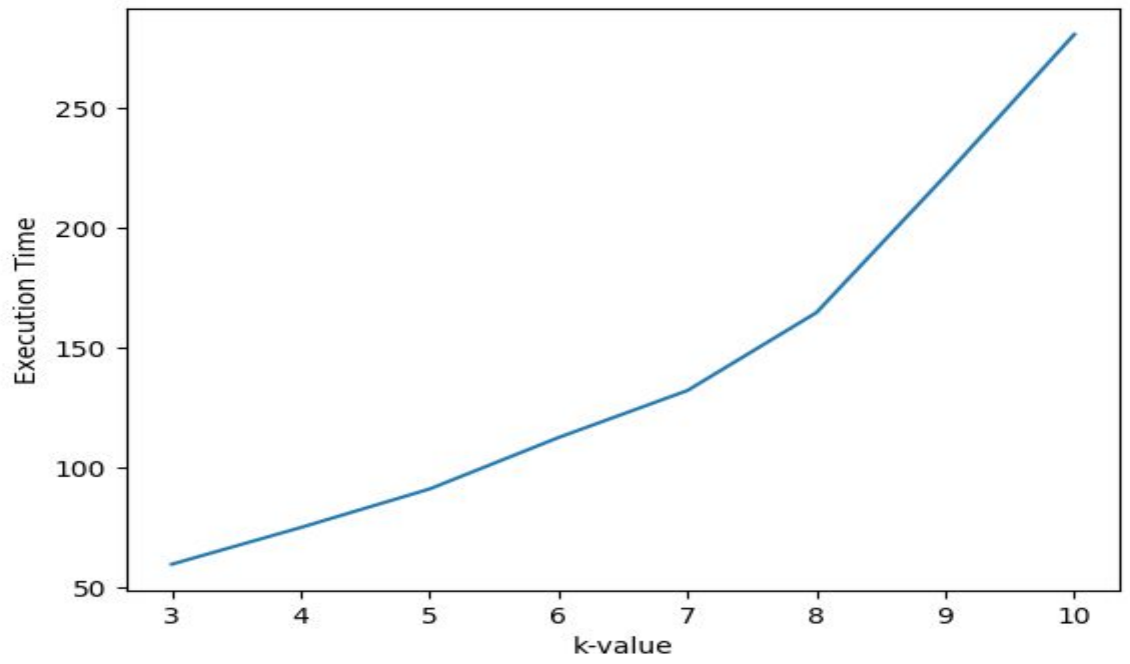
Baseline Performance

As explained in the problem statement , we divided our program into three cases . These are the execution times we got for the individual cases :

1) **Small sequence size (less than 256 MB) and k<13 -> Around 60 seconds**

2) **Large sequence size(around 2GB) and k <13 -> Around 500 seconds**

3) **K>12 -> Taking more than 10-15 minutes for executing fully**


Execution Time
50
100
150
200
250
3
4
5
6
7
8
9
10
k-value

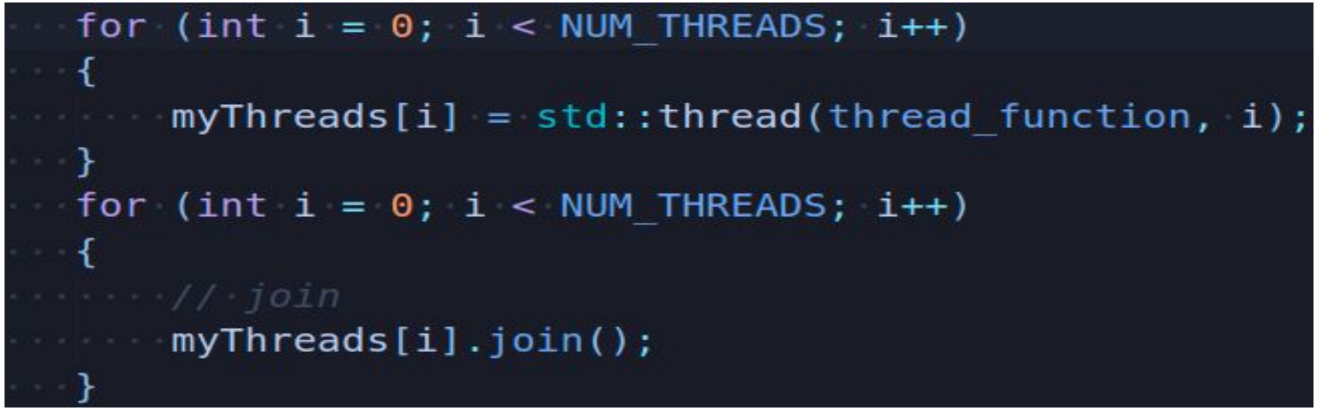
Optimization Techniques

We used multiple techniques for each case for optimization .

The important observation for us was about for k around 12~14 , we don’t need to make map for storing the k-mers with their corresponding frequency . This is because we can easily construct an array of size 4^(12) and consider all possible combinations , while this would not have been possible for larger value of k since the number of possible combinations would need several GigaBytes of data to be completely stored .

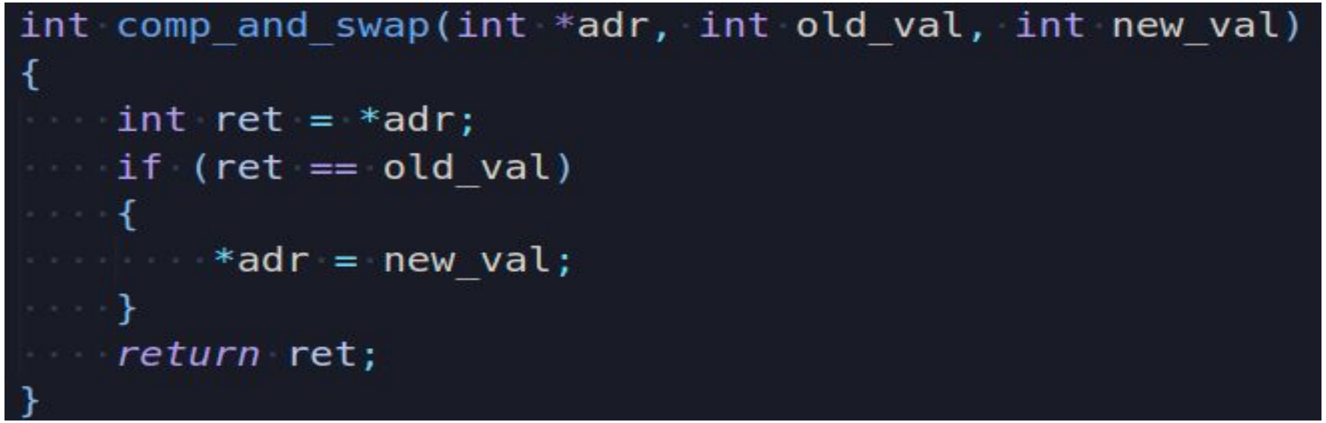
So for smaller value of k , we used threading to invoke parallelization along with the use of CAS (compare and swap) to implement a lock free data structure . This improved the execution time of the program by multiple folds .

THREADING : Instead of using openmp library for parallelization , we instead implemented the threads manually using the thread library of c++ . This gave us more freedom with what we wanted to do with each thread .



```
    for (int i = 0; i < NUM_THREADS; i++)
    {
        myThreads[i] = std::thread(thread_function, i);
    }
    for (int i = 0; i < NUM_THREADS; i++)
    {
        // join
        myThreads[i].join();
    }
```

CAS : Since we are using a single array to update the count of each k-mer , it is important to take care that two threads don't simultaneously try to change the value at the same array index . One way to solve is by using lock but it unnecessarily takes CPU time and eventually performs than a single thread . Instead what we did is we used a lock free data structure CAS which guarantees at at least one thread is non starving .



```
int comp_and_swap(int *adr, int old_val, int new_val)
{
    int ret = *adr;
    if (ret == old_val)
    {
        *adr = new_val;
    }
    return ret;
}
```

Large DNA sequence :

For the large DNA sequences (of size around 2 GB) , it was not possible for us to execute the program in a single go , so rather what we did was we divided the original sequence file into number of smaller sequence files and gave them to each of the thread to handle which later combined the result .

Doing this also provided a major boost in speed up .

For larger values of k , we have to apply a different strategy since now we cannot store all the substrings . Also , we are now interested in the most frequent k-mers rather than all since it is not computationally feasible to store the count for each k-mer .

Doing this involves a combination of Bloom-Filters with min sketch , Sorting and Compression :

Let us understand them one by one :

Bloom Filter with min align: They are a space-efficient probabilistic data structure that is used to test whether an element is a member of a set and if they are approximately how many times with have been same .We begin with a empty array and use 4 different hash function .

If all the hash values for a k-mer is above a threshold , we add it to an array else we just increase the count at each of the four hash values .

Sorting and Compaction: After the array A is filled , we sort the newly added elements , then do a linear traversal through them combine those which are the same .

After this , we employ a two pointer method on the already sorted string and the newly sorted string so that each k-mer appear only once and in the new space generated , the remaining elements are added until array is full and process is repeated .

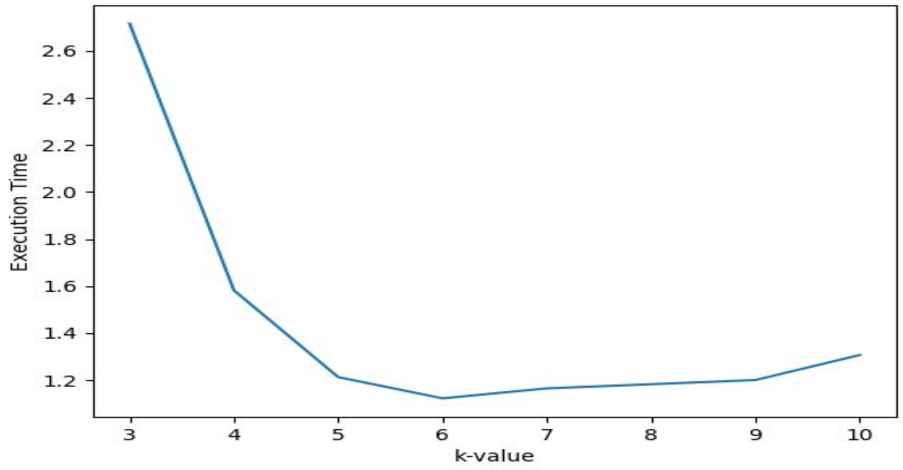
For best performance , we use radix sort because of the small value of b .

Speedup-> Around 100 times

Using a combination of these optimization techniques , we were able to achieve the following speed up ->

1) For k -value<13

   and Small Data Set :

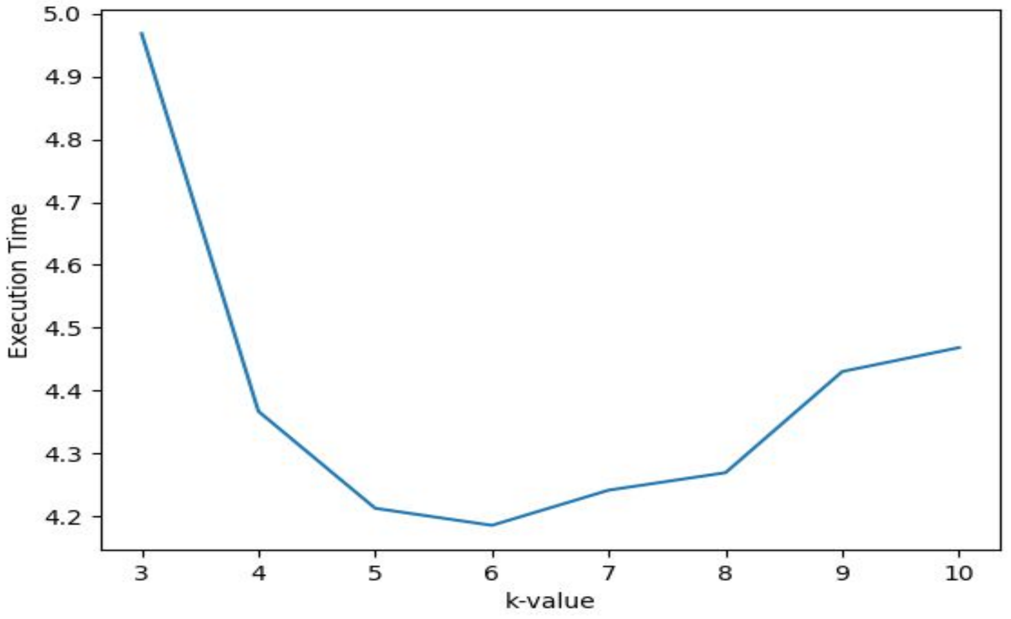


By comparing the two graphs (baseline and optimized version) , we can see that that on average , **the speed-up is about a 100 times** .

The k-mer counting is now incredibly fast since we have improved its time complexity , we well as involved parallelization which together are producing incredible results .

For k value < 13 and Large Dataset :

In the optimized version , where we are dividing the original file into several smaller files and running them on separate cores , the time taken for k = 3 is about 5 seconds .



The baseline version for k = 3 took 500 seconds , so we reckon sinec the optimized version is taking less than 5 seconds for every k , **the speedup has to greater than 100 times** .

K>12:

Finally for the most complex case , the baseline version was taking a lot of time for execution since the number of k-mer are in GigaBytes . So our aborted the process after it did not produce output for more than 10 minutes .

However , the optimized code we wrote using a combination of bloom filter with min align , sorting and compression produced the output in 10 seconds which is very fast considering that since k i so huge , there would be so many different combinations which we would have to consider to generate final output .

So **Speed Up >= 100 times**

Summary

In this project we tried to solve the k-mer counting problem efficiently. For this, depending on the size of the input string and the value of K we used various optimization techniques. While we could use small DNA sequences as it is, by taking as input, the large sequences had to be divided into smaller parts for fitting into the memory. We tried to obtain parallelization while counting through threads. Other than that we used various optimization techniques like compare and swap, bloom filters, sorting, compaction,etc. along with compiler optimizations to get the best results. We checked the correctness of our codes using the brute force programs which lack efficiency but are accurate. On comparing performance of our most optimized programs with baseline programs, we got a speedup of more than 100 in each case.

Insights And Future Work

We believe there can be lot of improvement in our work specially in the case when k> 12 . We can do a whole lot of things to further improve the space and memory complexity and produce more and more approximate results .

For example we can implement something called pattern-blocked Bloom Filter which localizes the generated hashes to a certain block which would allow for lesser cache miss and more memory locality .