Bioinformatics
Date: 09 Dec 2018

OXFORD

KMer Abundance Estimation

KmerEstimate: A Streaming Algorithm for Estimating k-mer Counts with Optimal Space Usage-Parallel Implementation

Aravind Reddy Ravula [112026804], Jahnavi Chowdary Tharigopula [112078393], Samanvitha Reddy Panyam [112025771] and Sriram Gattu [112007687]

Computer Science, Stony Brook University, Stony Brook, 11790, United States

Abstract

We implemented a parallel version of the algorithm provided in *'KmerEstimate: A Streaming Algorithm for Estimating k-mer Counts with Optimal Space Usage'* [2]

Motivation

Both space and time efficient algorithms are important for any problem that deals with the humongous size of data that we deal with in Computational Biology problems. While the above algorithm is efficient in terms of space, it can further be optimized in terms of time as well by implementing it in a parallel fashion. Hence, this formed the motivation towards working out a parallel implementation of the existing solution.

Results

Our parallel implementation of the serial algorithm achieved a 25% decrease in the execution time using 2 threads.

1 Introduction

Counting distinct number of k-mers (substrings of length k in a DNA/RNA sequence), and computing the frequency distribution of k-mers (k-mer abundance histogram), in a genome data is central component in many bioinformatics applications. Some example applications are sequence assembly, read error correction, genome size prediction and estimation of its characteristics, changes in copy number of highly repetitive sequences, digital normalization, and parameter tuning of k-mer analysis based tools. People have tried several approaches for computing counts of all distinct k-mers present in sequence data like sorting, suffix-array, filter and sketch data structure, and parallel disk-based partitioning. These exact count algorithms did not scale well for large datasets because of time and space complexity. Instead approximate count algorithms give better results. Streaming algorithms have been proved to be efficient for approximating the k-mer abundance histogram and related counts because of their memory and space usage. The tools which use streaming algorithms approach for k-mer counting problem are KmerGenie, KmerStream, ntCard, Kmerlight. KmerEstimate is yet another streaming algorithm that approximates the number of k-mers with a given frequency in a genomic data set.

In our project we are implementing the parallel version of KmerEstimate. The source codes and relevant documents including results are available at `https://github.com/aravindreddyravula/KMerEstimateParallel`

2 Existing Approach

Our goal is to implement a parallelised version of the algorithm provided in Paper [2] and their code available at [3], which is a space efficient solution for KMer Abundance Estimation. The paper uses the idea of the nt-card algorithm[4] and builds a variant of it to devise a space optimal solution.

- It samples a set of k-mers from S, the set of distinct k-mers appearing in the stream by hashing each k-mer uniformly at random to 64 bits. The algorithm uses ntHash[5] to compute the canonical hash values of k-mers.
- In order to build a collision free multiplicity table, this approach maintains 64 space efficient hash tables where the i^th hashmap encompasses all those k-mer samples whose corresponding hash values end with exactly i zero's.
- Then in order to achieve space optimality, they retain only those k-mers which have rightmost 's' bits as all zeros where 's' corresponds to a sampling rate $\frac{1}{2^s}$. Let B_s be the set of sampled k-mers.
- Here, the value of 's' is incremented dynamically by a value of 1 when the no.of k-mers inserted into the tables the reaches a certain threshold value L. This essentially doubles the sampling rate at every updation.

- Post every such updation, only the tables that have at least 's' trailing zeros in their 64-bit hash value are preserved while the rest of them are dropped.
- Once all the sequences are processed, the value of 's' at this point is the final sampling rate and the the frequency k_i, which is the number of sampled k-mers which have frequency of exactly i in the stream is estimated as $f_i = k_i \cdot 2^s$.
- This way of updating the 's' value dynamically is an improvement in comparison to the ntCard algorithm where the sampling rate was fixed to 7 or 11.

The above serial implementation of the algorithm has been described in detail in 'Algorithm 1' described below.

Now, while this algorithm is a good space optimal, efficient solution for the problem of KMer Abundance Estimation, it bears scope for improvement of its time complexity by parallelizing the approach and this is our focus of work.

Algorithm 1 Serial Implementation

0: **procedure** KMerEstimate
1: th : number of trailing zeros
2: s_i : sequence of reads upto s_n
3: k_j : k-mers j of a sequence read upto k_m
4: T : Table used by the update and output process. Partitioned into 64 hashmaps based on the number of trailing zeros.
5: L : Maximum records that T can hold
6: $count$: number of entries in the table
7: x_i : number of hashes in Table T which have frequency = i
8: th_f : Final sampling rate
9: **for** s_i = s_1,....,s_n **do**
10: **for** k_j = k_1,...,k_m **do**
11: h(k_j) = hash(h_j)
12: **if** h(k_j) ends with $\geq$ 0's **then**
13: **if** h(k_j) is in table T **then**
14: Increment frequency of h(k_j) and go to next k-mer
15: **else**
16: Insert h(j) into table with frequency 1
17: **if** table size = L **then**
18: Increment s
19: Retain all the entries with hash values having s lsb bits 0's
20: **else**
21: go to next k-mer
22: **end if**
23: **end if**
24: **else**
25: go to next k-mer
26: **end if**
27: **end for**
28: **end for**
29: **Estimation**
30: **for** f_i = f_1,....,f_{64} **do**
31: $\hat{f}_i = x_i 2^{s_f}$
32: **end for**
32: **end procedure**=0

3 Our Approach

In order to implement the above described algorithm parallelly, there are two problems that need to be addressed.

1. How do the threads get their input, i.e how should the huge set of streams in the entire dataset be accessed by the different threads in an efficient manner.
2. Once each thread gets its set of sequences, how does it process these sequences in a parallel manner in order to build the KMer abundance histogram.

We brainstormed certain approaches (which are described below) to solve the above problems, and chose the approach that seemed to be a good parallelized solution. First, we worked towards obtaining the sequence reads for the multiple threads in an efficient manner.

3.1 Obtaining the Sequences:

Initial Approach:

- A naive initial thought was to read the sequences serially one by one as before, then divide the sequence into 'n' chunks, where 'n' is the number of threads running, and process these chunks on each of the threads parallelly.
- The idea here is that a new sequence is read only after all the threads are done processing the previous chunks of the sequence.
- The problem with this approach is that not all threads may take the same amount of time to process the chunk of the sequence that they get as input, thus leading to a significant amount of Waiting Time.
- Hence, this approach does not really solve the problem of parallelizing the algorithm, but instead may introduce an extra overhead of processing time for dividing the sequences into chunks in addition to the waiting time.

Optimized Approach:

- With input from Professor Rob Patro, we then moved ahead to use 'FQFeeder', which is a simple module that is a self-contained, multi-threaded, Fasta/q parser [6].
- This parser maintains a queue, has producers that read information from a file and push it onto the queue and has consumers that read from the data in the queue and process them.
- This is an efficient and also liable method for the threads to obtain the sequences as the writing and reading from the queue can happen parallely.
- And, unlike the above approach, we do not need to divide the sequences into chunks which we believe is a boost in terms of feasibility and complexity.

Now, having the 'FQFeeder' method to obtain the sequence reads, we moved onto the next problem of processing these reads in a parallel, multi-threaded fashion.

3.2 Processing the Sequences:

Initial Approach:

- An initial naive approach was to maintain a vector of Hash tables of size 64 for each thread (that would result in 64n hash tables in total) and run the same single threaded algorithm on each thread to update their respective 64 hash tables.
- And then to obtain the final KMer abundance histogram, sum up all the counts from the above hash tables.
- There are two major issues with this approach.

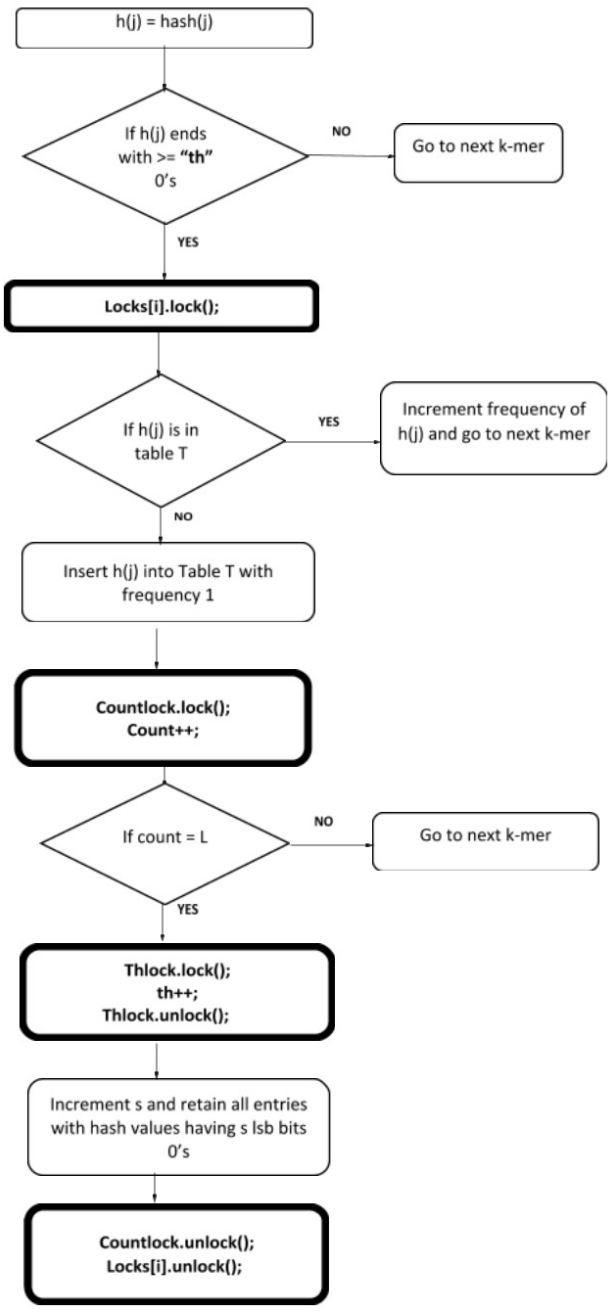

Fig. 1. Flow of our algorithm where the boxes in bold indicate the locking and unlocking system that we introduced to the existing serial approach. The rest of the flow is same as that provided in the paper.

1. The proposed solution in the paper is a space efficient solution, but this approach completely compromises on the space efficiency by maintaining high number of hash tables.
2. Also, the s-value in the algorithm is updated dynamically based on the sequence reads, and hence different threads will have obtained different s-values as the set of sequences being processed by each of these threads is different. Thus, even if we take the sum of counts of a k-mer in the different hash tables, this would be inaccurate due to the non-uniformity of the s-value among the threads.

Optimized Approach:

To overcome the above mentioned issues, we decided to maintain a single vector of Hash tables of size 64 (as in serial implementation) and identified the possible concurrencies that need to be addressed.

1. Multiple threads may want to access/update a particular hash table at the same time which could lead to inaccurate reads and updations.
   - To address this, we introduce a 'Lock' for each of this table, i.e we introduce an array of Mutexes of size 64 for each of the table.
   - Thus, after the thread decides on which hash table to access (based on the no.of trailing zeros in its hash value), before accessing this hash table directly, it first acquires the lock on the particular hash table and then continues its processing.
   - It releases the lock after it completely processes its current request (until Step 30 in the Algorithm 2).
   - When a thread wants to access a table whose lock is acquired by another thread, it must wait until the processing thread releases the lock.
2. The s-value that is being updated dynamically needs to be updated uniformly among all the threads, i.e all the threads update a single shared variable 's' (which is the sampling rate).
   - In order to take care of this scenario where multiple threads do not inaccurately update the s-value, we introduce a 'Lock' exclusively for the incrementing of the s-value.
   - Everytime when a thread wants to update the s-value, it first acquires the lock, updates the s-value and then releases the lock. This way, the updations to the s-value happen accurately.
3. In the algorithm, there is a limit on the total no.of k-mers inserted into the 64 hash tables, and when the limit is reached, the s-value is incremented. This cumulative count on the no.of k-mers being inserted into hash tables is also shared among the threads as different threads parallely process different no.of k-mers and are required to keep track of the accurate count of k-mers in the hash tables.
   - Hence, another 'Lock' exclusive to this count is maintained.
   - The lock is acquired by a thread after it decides that it needs to insert the current k-mer into a hash table as it is not present in the tables. It then increments the count value, and releases the lock after it completely processes the current k-mer and before the lock on the hash table is released.

Hence, using the above approach we implement a parallel solution of the KMerEstimate algorithm that takes into consideration the possible concurrencies that could occur with it.

4 Experiments and Results

- We tested our implementation on Human Chromosome 14 fragment 1 data set for different no.of threads (with nt = 1,2,...,10), k-mers (with k-mer size = 31, 15), Coverage (with coverage = 64, 12) and computed the Time in seconds that the algorithms takes.
- The results, i.e the KMer Abundance Histogram obtained by using different threads is same as that obtained using the single threaded implementation.
- The run corresponding to No.of threads = 1 is nothing but the result of the serial implementation provided in the paper.We evaluated the time taken for higher no.of threads against this value. The results are tabulated below in Table 1.
- We plotted the No.of threads Vs the Execution Time taken for the different k-mer size and Coverage in Figure 2.

Algorithm 2 Parallel Implementation

```
0: procedure KMerEstimateParallel
1: Initialize variables 1 to 8 from Algorithm 1
2: np : number of producers for FQFeeder
3: nt : number of threads
4: mutex locks[64] : locks for each hash table
5: mutex thlock : lock for updating the sampling rate
6: mutex countlock : lock for updating the number of entries in the table
7: FQFeeder parser start
8: for i = 1.....nt do
9:    Thread i reads sequences from FQFeeder
10:   for s_i = s_1.....s_n do
11:     for k_j = k_1.....k_m do
12:       h(k_j) = hash(h_j)
13:       if h(k_j) ends with ≥ th 0's then
14:         // Thread i locks the hash table which has k-mers with th no
            // of trailing zeros
15:         locks[i].lock()
16:         if h(k_j) is in table T then
16:             Increment frequency of h(k_j) and go to next k-mer
17:         else
18:           Insert h(j) into table with frequency 1
19:           countlock.lock()
20:           Increment count
21:           if count = L then
22:             thlock.lock()
23:             Increment th
24:             thlock.unlock()
25:             Retain all the entries with hash values having s lsb bits 0's
26:           else
27:             go to next k-mer
28:           end if
29:           countlock.unlock()
30:         end if
31:         locks[i].unlock()
32:       else
33:         go to next k-mer
34:       end if
35:     end for
36:   end for
37: end for
38: FQFeeder parser stop
39: Estimation
40: for f_i = f_1,....,f_64 do
41:    f̂_i = x_i 2^{s_f}
42: end for
42:   =0
```

Table 1: Results

S.No	KMer Size	Coverage	No of threads	Time(seconds)
1	31	64	1	116.612
2	**31**	**64**	**2**	**89.3561**
3	31	64	4	377.905
4	31	64	8	377.037
5	31	64	10	379.173
6	15	12	1	133.771
7	**15**	**12**	**2**	**105.95**
8	15	12	4	386.624
9	15	12	8	378.339
10	15	12	10	374.201

Fig. 2. Comparision of Execution time for KmerEstimate based on k-mer size and coverage

5 Discussion

We performed a Time and Space analysis of our Parallel implementation Vs the Serial implementation proposed in the paper.

5.1 Time Analysis

- We observe that in both the scenarios for different values of k-mer size and coverage, we get the best solution using number of threads equal to 2 which is a significant decrease from the single threaded implementation, thus satisfying the requirement of implementing a parallel solution for the provided paper. But, the execution time begins to increase as the number of threads increase more than 2.
- This behaviour may be due to how the sequences are read by the different threads. The FQFeeder takes as input the number of producers and the number of consumers. In our algorithm, we considered a single thread that reads from file and writes onto the queue.
- As the number of consumer threads increase, there are a lot more threads trying to read from the queue whereas there is a single queue working on writing onto the queue. We believe that this could cause quite an amount of Waiting Time for the consumer threads wherein the queue may be empty at multiple access times.
- Based on our understanding and knowledge, the reason for considering a single producer was because when we are trying to read from a single file from the disc, file reading would happens serially as there will be only a single disk pointer for both reading and writing from/to the disk. Hence, even if we span multiple threads to read from the file on the disc, when one thread is reading from the disk the other threads would have to wait until the current thread finishes it's task. Thus, serializing the process and making it equivalent to having a single thread do the job.

5.2 Space Analysis

Now, we analysed the Memory footprint of our algorithm, as it is important to retain the space efficient nature of the original algorithm.

- The additional resources that we are using in our algorithm are '66 Mutexes' (64 for the Hash table, 1 for the s-value, and 1 for the number of k-mers inserted into the tables) and 'nt' (number of consumer threads) + 'np' (number of produce threads) threads.

- As mentioned by Professor Rob Patro, and also by researching online, we learned that the size of a std::mutex is 40 bytes and the size of a std::thread is 8 bytes.
- Thus, the additional mutexes introduce an extra memory of 2640 bytes. And, for our implementation, we considered a single producer (np = 1), and we obtained a time efficient solution for 2 threads (nt = 2). Thus, these introduce an additional memory of 24 bytes.
- Thus, in total our parallelised implementation introduces an extra memory of 2664 Bytes which we believe is a reasonable tradeoff with respect to the time efficiency that is being achieved.

6 Future work

- A possible limitation in our approach could be that there could be collisions wherein multiple threads are trying to update the same hash table. In this scenario, there can be some wait time causing multiple threads to wait on the lock of each hash table in order to get their turn to do the processing.
- As a solution to this, we came up with an approach where we change the structure of the hashmap as follows, the 'key' is the kmer and the 'value' is an object of a class which has variables 'count' and 'mutex'.
- So, we use the FQFeeder to parse the input file and when a thread has to insert a kmer into the hashtable it checks whether that particular kmer exists in the hashtable with corresponding number of trailing zeros or not, If it isn't there it will insert it into the table. Otherwise the thread will check for the mutex of the corresponding kmer and update it as and when the lock is free.
- Essentially, by doing this we are introducing a mutex for each entry in the hash tables thus catering to lower no.of collisions as it is less likely that multiple threads come to access the same kmer is a hash table simultaneously.
- Though this approach seems to be a solution to decrease the number of collisions, we see that this introduces a huge number of mutexes which could introduce quite a big memory overhead based on the fact that each mutex takes upto 40 Bytes (as discussed above), thus violating the space efficiency nature of the existing algorithm.
- Hence, we understood that there exists a tradeoff between the space and time complexity of the problem, and hence moved ahead with the previous described optimized solution.
- Although, it would an interesting future work to see if some variation of the currently described approach has a feasibility of improving the time complexity.

7 Conclusion

Computational Biology involves working with huge datasets and this demands implementation of Time and Space efficient solutions to various problems among which KMer abundance histogram estimation is one. We worked towards implementing a parallel version of a space efficient, approximate, streaming algorithm for this problem and were successful in improving the execution time by using 2 threads. This project has helped us think in the direction of coming up with optimal, feasible solutions where in we brainstormed through multiple approaches and evaluated their correctness, as well as their time and space complexities and built upon the different methods to implement a time optimal solution.

8 Acknowledgements

- We would like to thank Prof. Rob Patro who provided us with important insights into the problem and helped us when needed to come up with a feasible solution.
- Also, the introduction of our paper has been used from Paper [2] which draws on the importance of KMer Abundance Histogram Estimation problem and also details the various existing streaming algorithms to solve this problem.
- The Figure 1 is an updated version that combines our parallelized algorithm to that that provided in the paper
- The Algorithm 1 is that described in the paper and the Algorithm 2 is built upon this which incorporates our parallel implementation.

9 References

[1] https://docs.google.com/document/d/1ZahvQe67imDQkm4YA2EbavS5NmgB7jpoGnAqMXvZuqM/edit

[2] KmerEstimate: A Streaming Algorithm for Estimating k-mer Counts with Optimal Space Usage

[3] Gregory P. 2016. sparsepp. https://github.com/greg7mdp/sparsepp. (2016)

[4] ntCard: a streaming algorithm for cardinality estimation in genomics data

[5] Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, and Inanc Birol. 2016.ntHash: recursive nucleotide hashing. Bioinformatics 32, 22 (2016), 3492–3494.

[6] FQFeeder - https://github.com/rob-p/FQFeeder/tree/master/src