Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Submission Date: 9 December 2018

CSE549 Computational Biology

ParallelKmerEstimate: A Streaming Algorithm for Estimating k-mer Counts with Optimal Space Usage in Parallel

Urmil Kadakia[1], Vishnu[2], Nitin[3], Pavan Kumar Peela[4]

Stony Brook University, New York

Abstract

Motivation: The Frequency distribution of k-mers (sub-strings of length k in a DNA/RNA sequence) is very useful for many bioinformatics applications that use next-generation sequencing (NGS) data. Some examples of these include de Bruijn graph based assembly, read error correction, genome size prediction, and digital normalization. In developing tools for such applications, counting (or estimating) k-mers with low frequency is a pre-processing phase. However, computing k-mer frequency histogram becomes computationally challenging for large-scale genomic data. There are several algorithms which tackle this problem namely KmerStream, the state-of-the-art $\mathbf{ntcard}$, and the less preeminent KmerEstimate. Our algorithm $\mathbf{ParallelKmerEstimete}$ is based on $\mathbf{KmerEstimate}$ an approximate Kmer Estimate based streaming algorithm proposed in Sairam-Behera et al., for approximating the the number of k-mers within a given frequency.

Results: We implemented and tested our algorithm on 'Human Chromosome' 14 data set from Genome Assembly Gold-standard Evaluation (GAGE). The run-time of our algorithm is better than that of our baseline Kmer Estimate algorithm by atleast 70%. In addition, our algorithm has provable approximation and time usage guarantees.

Availability: The original paper and the source code can be found at https://dl.acm.org/citation.cfm?doid=3233547 and https://github.com/srbehera11/KmerEstimate. The modified source code of the algorithm is available at https://github.com/vishd29/KmerEstimate-Parallelized.

Contact: urmil.kadakia@stonybrook.edu[1], vishnudutt.paladugu@stonybrook.edu[2], nitin.mangalam@stonybrook.edu[3], pavankumar.peela@stonybrook.edu[4]

1 Introduction

The problem of counting the distinct number of k-mers, and more predominantly computation the frequency distribution of k-mers, like a histogram, in the data of a genome is a mesial component of many bioinformatics applications.

For computing counts of all distinct k-mers present in sequencing data indefinite number of tools like Tallymer [Kurtz et al. 2008], Jellyfish [Marçais and Kingsford 2011], BFCounter [Melsted and Pritchard 2011], DSK [Rizk et al. 2013], Khmer [Zhang et al. 2014], Turtle[Roy et al. 2014], Squeakr [Pandey et al. 2017] etc. are available which are based on the essential approaches such as sorting, suffix-array, filter and sketch data structure, and parallel disk-based partitioning. But however, in existence of huge data-sets these tools do not scale due to their inherent inefficiency in time and space complexity.

When dealing with huge data-sets a more pragmatic approach is to compute approximate counts instead of computing the exact counts. The streaming algorithms lately have substantially proven to be very effective for approximating the k-mer abundance histogram and related counts because of their efficient memory and time usage. To date, tools like KmerGenie[Chikhi and Medvedev 2013], ntCard[Mohamadi et al. 2017], KMerEstimate[Behera et al. 2018] etc. have evolved and incorporated streaming algorithms approach for k-mer counting problems.

Our algorithm is based on $KmerEstimate$, a streaming algorithm that approximates the number of k-mers with a given frequency in a genomic data set. We implemented and tested our algorithm on 'Human Chromosome' 14 data set from Genome Assembly Gold-standard Evaluation (GAGE). The run-time of our algorithm is better than that of our baseline Kmer Estimate algorithm by atleast 70%. In addition, our algorithm has provable approximation and time usage guarantees.

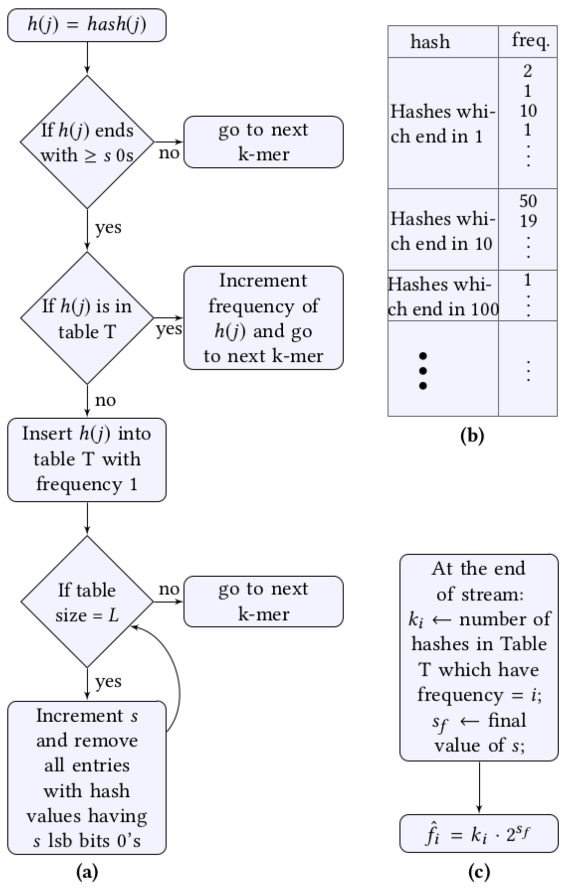
© The Author . Published by Oxford University Press. All rights reserved. For permissions, please e-mail: journals.permissions@oup.com

2 Problem Statement

The KmerEstimate is a streaming algorithm for estimating k-mer Counts Behera et al. [2018]. The inputs in a data streaming model are of form $s_1, s_2, ..., s_m$, where each s_i is a data item coming from a known universe of n items. Here our goal is to approximate a function of interest in a single pass over the stream only storing limited information. In our application, each data item is a $k - mer$ from a genome data set. The frequency of a k-mer is the number of occurrences of that k-mer in the data set. Let f_i be the number of k-mers with frequency i. Thus $f_1, f_2, ...f_m$ denotes the number of k-mers with frequency $1, 2, ..., m$ respectively. We are interested in approximating f_i for all i ($Kmer$ abundance histogram). When we have a huge dataset, scaling through the whole data set takes up significant amount of time. There is a scope to reduce the run-time of the streaming algorithm by incorporating multi-threading. The main challenge is to maintain the accuracy in estimation while using separate threads.

3 Methods

3.1 kmerEstimate



$h(j) = hash(j)$
If $h(j)$ ends with $\geq s$ 0s
no
go to next k-mer
yes
If $h(j)$ is in table T
yes
Increment frequency of $h(j)$ and go to next k-mer
no
Insert $h(j)$ into table T with frequency 1
If table size = L
no
go to next k-mer
yes
Increment s and remove all entries with hash values having s lsb bits 0's
(a)
hash
freq.
Hashes which end in 1
2
1
10
1
Hashes which end in 10
50
19
Hashes which end in 100
1
(b)
At the end of stream: $k_i \leftarrow$ number of hashes in Table T which have frequency = i; $s_f \leftarrow$ final value of s;
$\hat{f}_i = k_i \cdot 2^{s_f}$
(c)

Fig. 1. (a) The update process for the current k-mer j. The variable s is initialized to 0. (b) Table T used by the update and output process. T is partitioned into 65 hashmaps based on the number of trailing zeros. A maximum of $(L1)$ entries are allowed in T, for a parameter L. (c) The estimate for f_i for any $i > 1$, at the end of stream. The details are in method subsection-2.

The pseudo code for KmerEstimate is given in the Algorithm1. The basic idea is to sample a set of distinct k-mers from stream S. This is done by hashing the hashing the k-mer which has rightmost s bits all zeros, for a specified s. Then for each sampled k-mer its frequency is counted in the stream. After the whole stream is processed we calculate the k_is which are number of sampled k-mers which has frequency i in the stream. As the sampling rate is $1/2^s$ we can get the estimate of f_i as $\hat{f}_i = k_i \cdot 2^s$.

The algorithm is started with the sampling rate of $s = 0$ and as soon as the table size reaches the value L, the sampling rate is double by making $s = 1$. All the samples which have at least 1 trailing zeros in the hash values are retained and other are discarded. This step is performed recursively and increment in s is performed until the whole stream is processed. The final value of the s is the sampling rate of our algorithm.

In the given code they have used 65 space efficient hashmaps instead of a single array. In the implementation they have used sparsepp[Gregory [2016]] hashmaps. Each hashmap stores sampled k-mers with a certain number of trailing zeros. The program starts with $s = 0$ and once the total number of sampled k-mers reaches L, it deletes s^{th} hashmap and increments s value by 1. This ensures that after this step the total number of sampled k-mers are always less than or equal to the table size. This process is recursively applied until it finishes reading of all sequences in the given file. In the end the total number of sampled k-mers are always less than or equal to the table size.

Algorithm 1: Algorithm of estimating f_i

```
$h \leftarrow$ a uniformly random hash function mapping[n]$\rightarrow$ [n];
$L \leftarrow$ a parameter fixed in the analysis;
$B_0 \leftarrow \phi$;
$s \leftarrow 0$;
for arrival of data item $j \in 1, 2, , n$ in the stream do
    if $zeros(h(j)) \leq s$ then
        if $key \in B_s$ then
            $key.count = key.count + 1$;
        else
            $Insert(key, B_s)$;
            $key.count = 1$;
    while $|B_s| \leq L$ do
        $B_{s+1} \leftarrow$ Remove all keys $(\alpha, \beta)$ with $\beta = s$ from $B_s$;
        $s \leftarrow s + 1$;
$s_f \leftarrow s$;
$k_i \leftarrow$ the number of samples in $B_{s_f}$ with count value exactly $i$;
$Return\hat{f}_i = k_i \cdot 2^{s_f}$;
```

3.2 ParallelkmerEstimate

In parallel programming, one can divide a large problem into smaller sub-problems and then execute on different threads concurrently. To parallelize the k-mer estimation problem we have a producer-consumer approach. Here the producer is producing the data and consumers will be using it for their required calculations. After the parallel thread execution completes, consolidation is to be done to get the final desired result, in other words the input and the output will be the same for both serial and parallel but the underlying approach for parallel will split the data and execute it on multiple thread to speed up the process. The flow of the algorithm is shown in Figure 2.

In our case, a producer thread will read b number of sequences from the given file and then put them in the private queue of each consumer. here the b is parameter can be tuned to get the best performance for the given system. If a consumer finds it has an queue empty it will notify the producer to fill up his queue. In the serial case the sampling size was L, but here the sample size L is divided between all the threads, thus each thread will have $L^{'} = L/m$ as their sample size, where m being the number of threads. The reason for dividing the L is that when we consolidate the hashmaps to get a global hashmap the frequency of each k-mer will be less than or equal to the global sample size L. Along with sample size $L^{'}$ each thread will have its own copy of $s^{'}$ and the hashmap.

After the producer has filled up the queues of each thread the flow of the algorithm is analogous to that of the serial code. Each thread is started with the sampling rate of $s^{'} = 0$ and as soon as the table size reaches the value $L^{'}$, the sampling rate is double by making $s^{'} = 1$. All the samples which have at least 1 trailing zeros in the hash values are retained and others are expunged. This step is performed recursively and the increment is performed on $s^{'}$ until the producer completes the reading of the whole stream and consumer thread has emptied it's private queue. Next step is to consolidate all the local hashmaps into the global hashmap. We iterate over all the local hashmaps that are generated by the threads and adding the k-mer frequency value of this hashmap to the global consolidate map. The pseudo code for the parallel implementation is given in the algorithm 2.

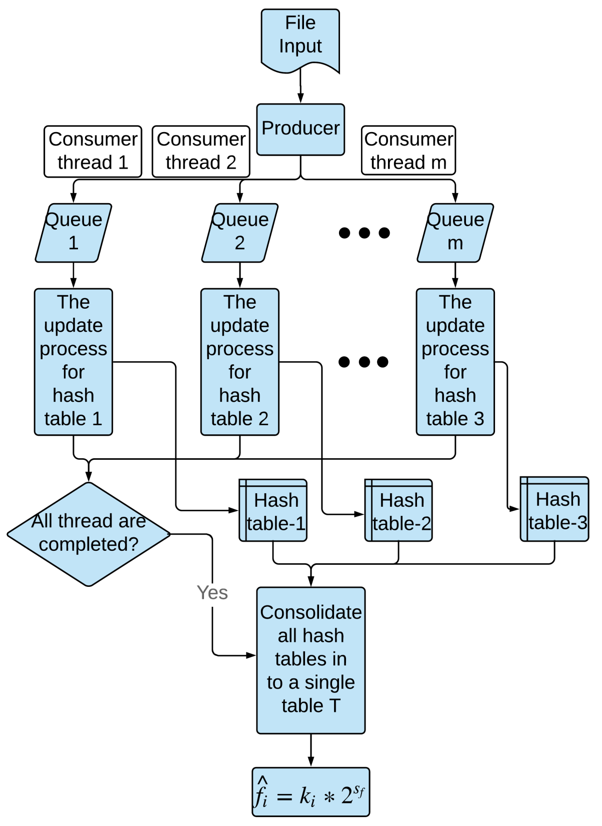

File Input
Producer
Consumer thread 1
Consumer thread 2
Consumer thread m
Queue 1
Queue 2
Queue m
The update process for hash table 1
The update process for hash table 2
The update process for hash table 3
Hash table-1
Hash table-2
Hash table-3
All thread are completed?
Yes
Consolidate all hash tables in to a single table T
$\hat{f_i} = k_i * 2^{s_f}$

Fig. 2. The update process for the hash tables is the same as described in part(a) for the serial process. Here each thread will have its own Table T, which is used by the update and output process. T is partitioned into 65 hashmaps based on the number of trailing zeros. A maximum of $(L/m - 1)$ entries are allowed in T, for a parameter L and number of threads m. Finally, the estimate for f_i for any $i > 1$, at the end of the stream. The details are in method subsection-2.

Algorithm 2: Algorithm of estimating f_i in parallel

```
$h[m] \leftarrow$ a uniformly random hash function mapping[n]$\rightarrow$ [n];
$L \leftarrow$ a parameter fixed in the analysis;
$L^{'} \leftarrow L/m$;
$b \leftarrow$ a parameter to change batch size;
$B_0[m] \leftarrow \phi$;
$s^{'} \leftarrow 0$;
$consolidated\_map \leftarrow \phi$;
$producer \leftarrow$ master thread;
$p \leftarrow$ queue size;
$queue[m] \leftarrow empty$;
$notify \leftarrow zeros[m]$;
for arrival of data items in the stream do
    if $notify[i] == True$ then
        $queue[i] \leftarrow$ producer will put $p$ items in the queue
    The following loop is running in parallel
    for all consumer thread t in $(1..m)$ do
        if $queue[i]$ is empty then
            raise $notify[i]$;
        else
            if $zeros(h[t](j)) \leq s$ then
                if $key \in B_{s^{'}}[t]$ then
                    $key.count = key.count + 1$;
                else
                    $Insert(key, B_{s^{'}}[t])$;
                    $key.count = 1$;
        while $|B_{s^{'}}[t]| \leq L^{'}$ do
            $B_{s^{'}+1}[t] \leftarrow$ Remove all keys $(\alpha, \beta)$ with $\beta = s^{'}$ from $B^{'}_{s}[t]$;
            $s^{'} \leftarrow s^{'} + 1$;
for $map$ in $B_{s^{'}}$ do
    for $kmer$ in $map$ do
        if $kmer$ not in $consolidated_map$ then
            $consolidated\_map[kmer] \leftarrow map[kmer]$
        else
            $consolidated\_map[kmer] += map[kmer]$
$s_f \leftarrow max(s^{'})$;
$k_i \leftarrow$ the number of samples in $consolidated\_map_{s_f}$ with count value exactly $i$;
$Return \hat{f}_i = k_i \cdot 2^{s_f}$;
```

K	f1/F0	DSK	KmerEstimate	Error(%)	ParallelKmerEstimate	Error(%)
31	$f1$	372,088,750	372,163,712	0.0066	372,057,920	0.0083
	$F0$	472,030,322	472,005,440	0.0073	471,942,016	0.0187
47	$f1$	385,778,023	385,892,800	0.33	385,770,624	0.0019
	$F0$	484,389,437	484,519,296	0.0285	484,426,624	0.0077
63	$f1$	336,752,336	336,800,384	0.0124	336,764,096	0.0035
	$F0$	432,569,742	432,633,440	0.0009	432,531,456	0.0089
79	$f1$	240,303,417	240,009,168	0.5341	239,963,040	0.1416
	$F0$	329,415,228	329,235,664	0.411	329,172,480	0.0737

Table 1. Accuracy of algorithms in estimating F0 and f1 on data set of 'Human Chromosome' 14 from Genome Assembly Gold-standard Evaluation (GAGE)

4 Results

For evaluating the performance and accuracy of **ParallelkmerEstimate**, we have used Human Chromosome 14 from Genome Assembly Gold-standard Evaluation (GAGE) dataset.Salzberg et al. [2012]. We evaluated the performance of **ParallelkmerEstimate** by comparing with **kmerEstimate**. The accuracy of the results is compared against that of **DSK**, the exact k-mer counting tool. The results are obtained on a pc with 2.2Ghz Intel core i7 and 16GB 2400 Mhz DDR4 RAM.

The results for $f1$(count of singleton k-mers), F0(count of distinct k-mers) and the error percentages of **kmerEstimate** and **parallelkmerEstimate** are shown in table-1. The error percentage is calculated based on benchmark **DSK** The parallel implementation has almost similar error bound to that of serial implementation, see table-1.

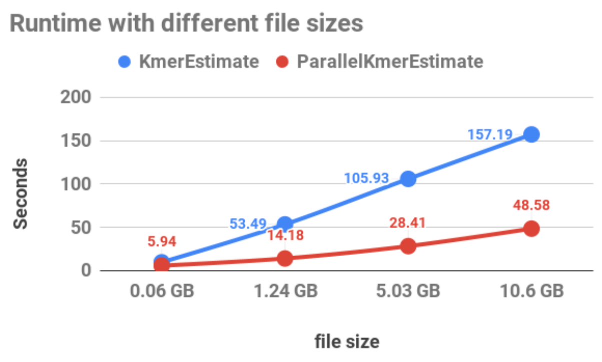


Fig. 3. Run-time analysis tested against on 'Human Chromosome' 14 data set from Genome Assembly Gold-standard Evaluation (GAGE) with parameters $k = 31$ for both algorithms and number of threads $m = 12$ in the parallel implementation

As the file size increase the **ParallelkmerEstimate** takes lesser time to run compared to the **kmerEstimate**. As shown in figure-3, our algorithm has a speedup of at least 3 as the file sizes are increasing.

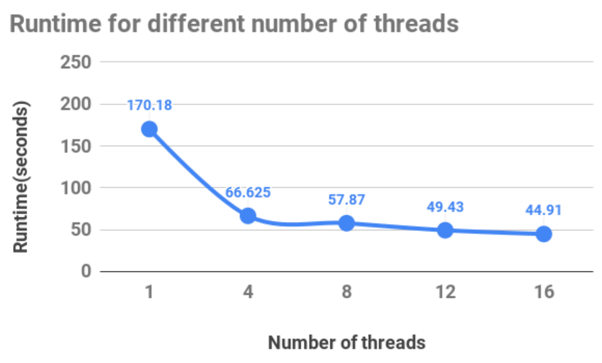


Fig. 4. Run-time analysis tested against on 'Human Chromosome' 14 data set from Genome Assembly Gold-standard Evaluation (GAGE) with parameters $k = 31$ and queue size $p = 10,000$

From the figure-4, we can see that as we increase the number of threads from 1 to 4 there is a sharp decrease in the run-time, but after that, we did not get much improvement. That is because as we increase the number of threads, the producer has to fill the same number of queues. After the consumer thread are done updating the hash table we have to consolidate them in a single table. As the number of thread increase, this process takes more time, as it has to consolidate a number of tables.

The queue size also has an impact on the run-time of the algorithm. One can observe in the figure-5, we increase the queue size from 10 to 1,00,000 there is a reduction in the run-time. As we put the queue size as 10,00,000 there is an increase in the run-time, that can be because of 2

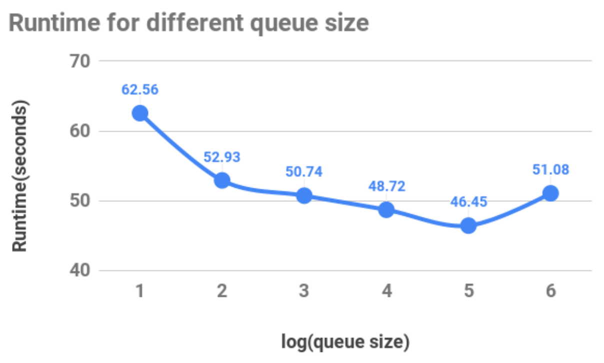


Fig. 5. Run-time analysis tested against on 'Human Chromosome' 14 data set from Genome Assembly Gold-standard Evaluation (GAGE) with parameters $k = 31$ and number of threads $m = 12$

reasons. First, the file size is smaller, thus the producer has not assigned work to every consumer thread(queues of some threads is empty). Second, the queues are generated by the producer is so large that processor cannot fit all the queues on the RAM, so it has to page them. As the paging is performed from the hard-drive it is significantly slower than the RAM.

5 Conclusion

In dealing with a large amounts of genomic data we need efficient computational approaches and data streaming models are widely-accepted computational model for their proven efficiency in space and time complexity compared to other models. In our project we try to develop a $ParallelKmerEstimate$, that approximates Kmer abundance histogram, which accelerates the streaming algorithm by incorporating multi-threading and reduces the run-time by up to 70% when compared to the previous streaming algorithms. Our algorithm is based on KmerEstimate $Beheraet\ al.$ 2018 which is a serial streaming algorithm. We implemented and tested our algorithm on 'Human Chromosome' 14 data set from Genome Assembly Gold-standard Evaluation (GAGE). The run-time of our algorithm is better than that of our baseline Kmer Estimate algorithm by atleast 70%. In addition, our algorithm has provable approximation and time usage guarantees.

References

Sairam Behera, Sutanu Gayen, Jitender S Deogun, and NV Vinodchandran. Kmerestimate: A streaming algorithm for estimating k-mer counts with optimal space usage. In *Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 438–447. ACM, 2018.

Rayan Chikhi and Paul Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31–37, 2013.

Gregory. Sparsepp: A fast, memory efficient hash map for c++. `https://github.com/greg7mdp/sparsepp`, 2016.

Stefan Kurtz, Apurva Narechania, Joshua C Stein, and Doreen Ware. A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC genomics*, 9(1):517, 2008.

Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

Pall Melsted and Jonathan K Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC bioinformatics*, 12(1):333, 2011.

Hamid Mohamadi, Hamza Khan, and Inanc Birol. ntcard: a streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, 33(9):1324–1330, 2017.

Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*, 34(4):568–575, 2017.

Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. Dsk: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.

Rajat Shuvro Roy, Debashish Bhattacharya, and Alexander Schliep. Turtle: Identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, 30 (14):1950–1957, 2014.

Steven L Salzberg, Adam M Phillippy, Aleksey Zimin, Daniela Puiu, Tanja Magoc, Sergey Koren, Todd J Treangen, Michael C Schatz, Arthur L Delcher, Michael Roberts, et al. Gage: A critical evaluation of genome assemblies and assembly algorithms. *Genome research*, 22(3):557–567, 2012.

Qingpeng Zhang, Jason Pell, Rosangela Canino-Koning, Adina Chuang Howe, and C Titus Brown. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *PloS one*, 9(7):e101271, 2014.