A probabilistic approach to k-mer counting

QP, Jason, Rose, Adina, CTB

November 4, 2012

1 Abstract

Introduction: K-mer counting is widely used for many purposes in sequence analysis, including data preprocessing for de novo assembly, repeat detection, and sequencing coverage estimation. However, currently available tools have high memory requirements when used on short-read sequencing data.

Results: We present the khmer software package for fast and memory efficient counting of k-mers in sequences. Unlike previous methods based on data structures such as hash tables, suffix arrays, and trie structures, khmer relies entirely on a simple probabilistic data structure, a Count-Min Sketch. On sparse data sets, this data structure is considerably more memory efficient than any possible exact data structure. The tradeoff is that the use of a Count-Min Sketch introduces a systematic positive overcount for k-mers. Here we analyze the analysis speed, memory usage, and miscount rate of our Count-Min Sketch implementation for simulated and real sequencing data sets. We also compare the performance of khmer to two other commonly used k-mer counting packages, Tallymer and Jellyfish. Finally, we present two applications of khmer to analyzing short-read sequencing data: first, for doing reference-free error analysis of short-read data with k-mer abundances, and second, for trimming errors from short reads to enable more memory-efficient assembly.

Conclusion: The Count-Min Sketch, as implemented in khmer, is an effective and efficient tool for k-mer counting in biological sequences. In particular, the miscount behavior of the Count-Min Sketch performs well on error-prone short-read sequencing data. We show that the khmer package can decrease the memory usage of several k-mer-based applications by 5-20 fold.

2 Introduction

A k-mer is a substring of length k in a DNA sequence. The goal of k-mer counting is to determine the number of occurances for each k-mer in a dataset composed of sequence reads. Efficient k-mer counting plays an important role in solving many bioinformatics problems.

One important problem in biology is de novo assembly of a large number of short reads. With the development of next generation sequencing technology, many research groups can afford to sequence samples of specific species or even metagenomic samples containing numerous different species[**?**]. Large numbers of next generation sequencing short reads can be generated by modern sequencers and often de novo assembly is required for these sequence data sets[10]. De Bruijn graph methods are popular for de novo assembly of short reads because they avoids the need for an all-by-all comparison of reads that scales poorly with the number of reads[10]. Several popular assemblers have been developed based on de Bruijn graphs, including Velvet[14], ALLPATHS[3], ABySS[13] and SOAPdenovo[7].

Each k-mer in a sequence dataset is represented as a vertex in the de Bruijn graph. If two k-mers share (k-1)-mer overlap, the two k-mers are connected. Since the k-mer is such a basic cornerstone in de Bruijn graph-based assembly, it is necessary to determine the occurrence of k-mers. One example of an application is for the detection and removal of sequencing errors. Sequencing errors can generate many erroneous k-mers, we can filter out the reads with too many unique k-mers prior to assembly. Similarly, we can filter out reads with k-mers that occur too frequently for smoothing MDA-abundance reads datasets. Pre-assembly filtering of reads to reduce dataset sizes is a crucial component of time and memory reduction. Additionally, we can use k-mer counting to evaluate genome size and the coverage of sequencing reads. K-mer counting can also give important information for predicting regions with repetitive elements such as transposons.[6]

Current methods for k-mer counting involve data structures such as hash tables, suffix arrays, or binary trees and tries (cite). Jellyfish uses a hashing data structure to efficiently count k-mers [8]. Tallymer uses a suffix array data structure[6].

The central challenge for k-mer counting in short-read data sets is that these data sets are both relatively sparse and contain many unique k-mers. The data sets are sparse because for typical values of k such as k=32, only a small fraction of the total possible number of k-mers (4^{32}) are actually present in the data set. However, because k-mers are generated from error-prone short-read sequencing reads, there are still many distinct k-mers.

Our motivation for exploring efficient k-mer counting comes from metagenomic data, where we have encountered data sets that contain 300e12 bases of DNA and over 50 billion distinct k-mers.

Below, we employ a simple probabilistic data structure for k-mer counting. This data structure is an implementation of a Count-Min Sketch, a generalized probabilistic data structure for storing the frequency distributions of distinct elements [4]. Our implementation is based on an extension of the Bloom filter, which has been previously used for k-mer counting and de Bruijn graph storage and traversal [1, 2, 9]. We provide a software implementation in a freely available package called 'khmer' and show that we can efficiently and effectively employ khmer for several common k-mer counting applications.

3 Methods

3.1 Sequence Data

Two human gut metagenome reads datasets (MH0001 and MH0002) were used from the MetaHIT (Metagenomics of the Human Intestinal Tract) project[11]. The MH0001 dataset contains approximately 59 million reads, each 44bp long. The MH0002 dataset consists of about 61 million 75bp long reads. We trimmed each FASTA file to remove low quality sequences.

Five soil metagenomics reads data sets with different size were taken from GPGC project for benchmark purpose. (Iowa Prairie Table 1 to cite here.)

We also generated four short read data sets to assess the false positive rate and miscount distribution. One is a subset of a real metagenomics data set from the MH0001 dataset previously mentioned. The second consists of randomly generated reads. The third and fourth contain reads simulated from a random, 1 Mbp long genome. One has a substitution error rate of 3%, and the other one contains no errors. The four data sets were chosen to contain identical numbers of unique 12-mers

3.2 Comparing with other k-mer counting programs

We counted the number of unique k-mers in five soil metagenomic reads datasets with different sizes using our khmer counting approach and two other k-mer counting methods - Tallymer and Jellyfish - to compare the performance. To investigate the k-independance of k-mer counting, we used both k=22 and k=31.

Tallymer is from the genometools package version 1.3.4. It was run with the following options. For the suffixerator subroutine we used: `-dna -pl -tis -suf -lcp`. We varied the `-parts n` option to create the index with only 1/n of the total data in main memory, which reduces the memory usage. We separately used `-parts 4` and `-parts 1` to test performance.

For the mkindex subroutine, we used: `-mersize 31` and `-mersize 22`.

Jellyfish is version 1.1.2 and the multithreading option is off.

Jellyfish uses a hash table to store the k-mers and the size of the hash table can be modified by the user. When the specified hash table size is not large enough and fills up, jellyfish writes it to the hard disk and initializes a new hash table to more k-mers. Here we use a similar stratage as in [?] and chose the minimum size of the hash tables for Jellyfish so that all k-mers were stored in memory.

We ran Jellyfish with the options as below:

`jellyfish count -m 22 -c 2 -C` for k=22.

`jellyfish count -m 31 -c 2 -C` for k=31.

3.3 Count-Min Sketch implementation

We implemented the Count-Min Sketch data structure, a simple probabilistic data structure for counting distinct elements. Our implementation uses N independent hash tables, each containing a prime number of counters M. The hashing function for each hash table is fixed, and bijectively converts each DNA k-mer up to k=32) into a 64-bit number to which the modulus of the hash table size is applied. This provides N distinct hash functions (see also [?]).

To increment the count associated with a k-mer, the counter associated with the hashed k-mer in each of the N hash tables is incremented. To retrieve the count associated with a k-mer, the minimum count across all N hash tables is chosen.

In this scheme, collisions are explicitly not handled, so the count associated with a k-mer may not be accurate. Because collisions only falsely *increment* counts, however, the retrieved count for any given k-mer is guaranteed to be no less than the correct count. Thus the counting error is one-sided.

3.4 Source code and scripts

We implemented this approach to k-mer counting in a software package named khmer, written in C++ with a Python wrapper. khmer is freely available under the BSD license at http://github.com/ged-lab/khmer/.

The version of khmer used to generate the results below is available at @@@. Scripts specific to this paper are available in the paper repository at @@@.

3.5 Benchmarking

4 Results

4.1 kher memory usage is fixed and low

Unlike other approaches to count k-mers, memory usage of our bloom count approach is independent of the size of the dataset. It only depends on the size(S) and number(N) of hash tables to use (i.e. O(SN)). As discussed above, for a given dataset, the size and number of hash tables will determine the accuracy of k-mer counting. Thus, the user can control the memory usage based on the desired level of accuracy. The time usage for the first step of k-mer counting , to consume the k-mers into a counting data structure, depends on the total number of k-mers in the dataset since we must walk through every k-mer in each read. The second step, k-mer retrieval, depends on the number of distinct k-mers in the dataset.

While the memory usage can be fixed at an arbitrary point, this leads to different false positive rates.

4.2 Collision rates are predictable and measurable

Firstly, the Bloom counting hash consists of one or more hash tables rather than a single hash table used in a typical Bloom filter. Secondly, each entry in the hash tables is a counter representing the number of k-mers that hash to that location, instead of a single bit to represent the existence or absense of an item. Third, in our implementation, the hash function is to take the modulus of a prime number number close to the size of the hash tables. For each hash table there is a different prime number. In other word, each hash table has a different hash function. Given a new k-mer, it is hashed to a specific location in each hash table using specific hash function. The counter in that location in each hash table can be updated. Like a Bloom filter, different k-mers can be hashed to the same location in a hash table, which results in collisions. The effect of the collisions is that the counter in a specific bin may not represent the exact count of a specific k-mer, but the maximum possible count of every k-mer that can be hashed to that location. If a k-mer has a count of 5 in one hash table, that k-mer has an occurrence no more than 5, including 0. On the other hand, a k-mer with exactly 5 occurrences may have a counter no less than 5 in a hash table. Like the Count-Min sketch, in our implementation, we pick the minimum count from all hash table bins for a k-mer to represent its occurrence to reduce the effect of collisions. By default, we make each bin 8 bytes in size for a maximum count of 255, but we also implement an extension that can expand the bin size to enable high abundance k-mer counting.

Like Count-Min sketch, this is a probabilistic data structure and the counting is not exactly correct. There is a one-sided error, which can result in an overestimate, but cannot be smaller than actual count. The chosen parameters of the data structure will influence the accuracy of the count, which can be estimated well like Count-Min sketch[4]. Such probabilistic properties suit the next generation sequencing short reads data sets well; that is, the counts are not so wrong for next generation sequencing reads data sets because of the generally skewed abundance distribution of k-mers in those data sets. We will talk more about this in section 4.1.2. Secondly, in many situations the accurate count of k-mers is not necessary. For example, when we want to eliminate reads with low abundance k-mers, we can tolerate the fact that a certain number of reads with low frequency will remain in the resulting data set falsely because of the frequency inflation caused by collision. If necessary we can do the filtering iteratively so each in step reads with low abundance k-mers can be discarded after every iteration. Furthermore, the rate of inaccurate counting can be predicted pretty well. We can adjust the parameters of the data structure to make sure the count accuracy satisfies our downstream analysis. Secondly, this k-mer counting structure is highly scalable. For certain sequence data set, counting error rate is related to memory usage. Generally, the more memory we can use, the more accurate the counting will be. However, no matter how large the data set is, we can predict and control the memory usage well with choosing specific parameters. Because of the similar characteristics as Bloom filter, given certain parameters like the size and number of hash tables to use, the memory usage is constant and independent of the length of k-mer and the size of the sequence data set to count. Our method will never break an imposed memory bound, unlike some other methods, while there is a tradeoff that the miscount rate will be worse.

4.3 False positive rates and miscount rates are predictable

We want to use the Count-Min sketch like data structure to count the occurrence of k-mer in a data set. So we define the counting error rate to be the possibility that the count is incorrect (off by 1 or more)

Suppose N unique k-mers have been counted with Z hash tables with size as H(Here we assume the hash tables have similar size), the probability that no collisions happened in a specific entry in one hash table is $(1 - 1/H)^N$, which can be estimated by $e^{-N/H}$. The individual collision rate in one hash table is $1 - e^{-N/H}$. The total collision rate, which is the probability that a collision occurred in each entry where a k-mer maps to in all Z hash tables, is $(1 - e^{-N/H})^Z$. In this situation, the counts in all Z hash table bins cannot give the true count of a k-mer.

Above we discussed the counting error rate of our counting approach, which tells the possibility that a count is incorrect. In some applications like abundance filtering, we also want to determine how poor the counts are by evaluating the difference between the overestimate and the actual count. In the analysis of the Count-Min sketch[4], the offset between the incorrect count and actual count is related to the total number of k-mers in a dataset and the size of each hash table. Specifically, suppose d hash tables with size as w are used in counting a dataset with T total k-mers, with probability at least $1 - e^{-d}$. The offset between an inaccurate count and the actual count is smaller than $e * T/w$. This analysis is distribution-independent and, being a worst case upper bound, is too conservative. In practice, for some data sets, the error estimation is reasonable, but for other data sets, the data structure we are using can outperform the theoretical worst-case bounds by many orders of magnitude. Further study shows that the behavior of Count-Min sketch depends on the characteristics of data set such as skewness[12]. Generally, for low skew data in which the k-mers have uniform distribution of frequency, the k-mers can be distributed into hash tables evenly with high probability. Thus, the average miscount is related to the average abundance of k-mers in the hash tables (T/w). In this case, the error estimation is closer to the actual error. Nevertheless, for high skew data in which a few k-mers consume a large fraction of the total count, generally these high abundance k-mers will be distributed to fewer entries in hash tables than low abundance k-mers, which results in fewer collision in the counting hash table in general[5]. This implies that the minimum count in the hash tables will have smaller miscount and the average miscount will be smaller than that for low skewed data. For high skewed dataset, the error estimation should be tighter than the confidence bound described above. If the skew of data can be determined by other methods, a higher bound can be acquired by using the Zipf coefficient.

According to the discussion above, number of unique k-mers in a data set , number of hash tables and size of hash tables will determine the possibility that a count is exactly correct, which is defined as counting error rate in this paper. Furthermore, the total number of k-mers in a data set, which is related to the size of the data set, and hash table size will influence how bad the counts are if they are not correct. The hash table number will determine the predictability of the error estimation. More used hash tables will increase the possibility that the count error is located in a interval, which can be predicted from number of total k-mers and hash table size. The k-mer abundance distribution also has influence to the degree of miscount. Given a fixed number of k-mers and hash table sizes, a skew abundance distribution generally has a smaller miscount rate than a uniform abundance distribution.

4.4 khmer can count k-mers efficiently

The time usage and memory usage for counting k-mers in five soil metagenomic reads data sets using khmer in comparison to Tallymer and Jellyfish are shown in Figure 1 and Figure 2. Figure 1 shows that the time usage of our khmer approach is comparable to Tallymer, but is slower than Jellyfish. From Figure 2, we see that the memory usage of both Jellyfish and Tallymer increases linearly with dataset size, although Jellyfish is more efficient than Tallymer in memory usage for smaller k size. Using option -parts 4 for Tallymer subroutine suffix can reduce the memory usage. But the second step of Tallymer counting method

- subroutine mkindex will always use more memory as dataset to count increases. For a 5 GB dataset with 2.7 billion total k-mers, Jellyfish uses 5 GB memory while Tallymer cannot handle a 4 GB dataset on a machine with 24 GB memory. Thus, both tools cannot satisfy the k-mer counting requirement for metagenomic data sets on computers with a modest amount of memory. In addition, the memory usage of our khmer approach also increases linearly with data set size. This is because we want to keep the counting error rate unchanged, like 1% or 5%. In fact the memory usage of our khmer approach can be adjusted with the tradeoff of counting error rate. We can decrease the memory usage by increasing counting error rate as shown in this figure. We can also see from the figure that with a decent counting error rate like 1%, the memory usage is still considerably lower compared to other programs. As shown in Figure 2, our khmer approach and Tallymer are both k-independence. Length of k-mer will not influence the memory usage, while the memory usage of Jellyfish depends on length of k-mer heavily. In fact Jellyfish only allows counting k-mers shorter than 32. Another drawback to consider is the disk usage when both Jellyfish and Tallymer generate large index files on hard disk. Figure 3 shows that the disk usage also increases linearly with the dataset size. For a dataset of 5 gigabytes, the disk usage of both Jellyfish and Tallymer is around 30 gigabytes. For larger metagenomic datasets, disk usage must be economized . The curve for khmer in Figure 3 is the size of hash tables used in our khmer approach with fixed counting error rate(1%). While the hash tables for k-mer counting can be stored on disk in khmer, this is not a requirement if the data structure can be held in memory for downstream analysis.

4.5 The measured miscount rate is low on short-read data

We have discussed assessment of counting accuracy in section 4.1.2. In practice, we are more interested in the performance of the approach when it is applied to high diversity or high error datasets. Like metagenomics dataset, a large number of the k-mers in such high diversity datasets is unique. Here we use real metagenomics datasets and three simulated high diversity datasets to evaluate the counting performance in practice.

From Figure 4 it is apparent that with larger hash table, the average offset decreases. The average offset is closely related to the number of distinct k-mers. For example, simulated reads without error has the most k-mers and highest average miscount out of the four data sets. This observation proves the conclusion discussed in section 4.1.2, which is that the number of total k-mers in the dataset and hash table sizes will influence how poor the counts are.

Figure 5 shows the relationship between average miscount and counting error rate for different test data sets. For a fixed counting error rate, simulated reads without error has the highest average miscount and simulated k-mers has the lowest average miscount. This is because they have the highest and lowest number of total k-mers, respectively. We can have more correct counting for real error-prone reads from a genome than for randomly generated reads from a genome without error and with a normal distribution of k-mer abundance. Thus, we can conclude that our counting approach is more suitable for high diverse data sets, like simulated k-mers and real metagenomics data, in which larger proportion of k-mers are low abundance or unique due to sequencing errors.

4.6 Sequencing error profiles can be measured with k-mer abundance profiles in reads

When dealing with large amount of sequencing data, one pressing issue is sequencing errors. Generally, the sequencing error occurs randomly. If k is large enough, most of the erroneous k-mers should be unique in the reads dataset. Detecting those sequencing errors to correct or remove them from reads can be computationally challenging with large datasets. Before doing any data analysis, we should have some idea about the sequencing error such as the estimation of the sequencing error rate, or error distribution. Generally, the quality score of sequencing reads can be a good reference to detect errors. According to the discussion above, k-mer abundance can be another approach to estimate the pattern of sequencing error. This quality-score free method can also be a useful approach to evaluate the validity of quality score generated by sequencing procedure.

Here we use this method to investigate the sequencing error pattern of an E. coli Illumina reads dataset as an example.

As shown in Figure 6 there are more unique k-mers close to the 3' end of reads. As we have discussed above, sequencing errors can generate unique k-mers. Also Figure 7 shows that there are more k-mers with high abundance (frequency = 255) close to the 5' end of reads.

Figure 8 shows the average frequency of k-mers in different position in reads. Because of the higher sequencing error rate, the frequency of k-mers close to the two terminals of reads is slightly lower due to more unique k-mers. All these results are consistent with the knowledge that Illumina reads are error-prone – especially on the 3' side.

Using our k-mer counting approach to do k-mer abundance analysis is an effective way to investigate the pattern of sequencing error of Illumina reads data. Knowing such pattern of sequencing error is important for choosing the proper filtering strategy or filtering threshold, which is a crucial step in preprocessing data for sequence data analysis and manipulation like assembly.

4.7 khmer can trim errors from reads

The most common approach to assembling short reads from next-generation sequencers is based on the de Bruijn (i.e. k-mer) graph. This approach has been applied with some success to human microbiome data. Because of technical limitations, there are sequencing errors in the reads, resulting in erroneous k-mers. As discussed in section 5.3, we can detect such erroneous k-mers by detecting low frequency k-mers. Removing or trimming reads containing unique or low-abundance k-mers will remove many errors. On the other hand, low-abundance k-mers, even though some of them are correct, do not contribute much to the assembly. In light of this fact, we filter out reads with low frequency k-mers to decrease time and memory usage.

Our k-mer counting approach is efficient for filtering reads based on k-mer abundance, and can be used for arbitrary k. One approach to k-mer abundance filtering involves removing any read that contains even a single low-abundance k-mer. This filtering can be implemented in two passes: the first pass for loading k-mers from reads and the second pass for filtering the reads. The counting error here manifests as an overestimated count in hash entries with one or more collisions, in which case k-mers hashing to that entry may not be correctly flagged as low-abundance. High counting error rate therefore manifest as "lenient" filtering, in which reads may not be properly removed. However, any read that is trimmed will be correctly trimmed. To reduce the effect of such counting error rate, we can do the filtering iteratively. After each run of filtering, some more reads with low-abundance k-mers will be discarded. This graceful degradation in the face of large amounts of data is a key property of our k-mer counting approach.

As an example of this method, we filtered out reads with low abundance k-mers from a human gut microbiome metagenomic dataset(MH0001) with more than 42 million reads. In fact we want any read with any unique k-mer to be discarded. We used hash tables with different size to show the influence of hash table size. We also showed the effect of iterative filtering to reduce false positive rate. To assess the counting error rate, we used Tallymer to obtain the actual accurate count of the k-mers in the dataset.

From Figure 9, we see that after each run, more low-abundance reads were discarded. With larger hash table, the low-abundance reads were discarded faster. On the other hand, from Figure 10, we can see that after each iteration of filtering, the percentage of "bad" reads - reads with unique k-mers - decreased. After four iterations, the percentage of "bad" reads was less than 4%. The result showed that with our method nearly 40% of the original reads were discarded by removing the low-abundance reads with an acceptable false positive rate (less than 4% after four iterations of filtering). It can reduce the memory and time requirement effectively in the following effort of assembly.

5 Discussion

5.1 khmer scales k-mer counting significantly and usefully

From this comparison, we can conclude that for large metagenomic datasets, other k-mer counting programs like Jellyfish and Tallymer fail to do the counting successfully because of the high memory and disk usage while our khmer counting approach can do the counting, although with the tradeoff of inaccuracy. However from the discussion in previous section, the inaccuracy can be estimated well, and we can evaluate if the counting accuracy suffices the requirement of a specific Bioinformatic analysis.

5.2 Error rates in k-mer counting are low and predictable

5.3 khmer applications

5.4 Future

5.5 Conclusions

5.6 Scaling the Bloom counting by partitioning k-mer space

Scaling the Bloom counting hash to extremely large data sets with many unique k-mers requires a large amount of memory: approximately 446 GB of memory is required to achieve a false positive rate of 1% for $N?50x10^9$. It is possible to reduce the required memory by dividing k-mer space into multiple partitions and counting k-mers separately for each partition. Partitioning k-mer space into N partitions results in a linear decrease in the number of k-mers under consideration, thus reducing the occupancy by a constant factor n and correspondingly reducing the collision rate.

Partitioning k-mer space can be a generalization of the systematic prefix filtering approach, where one might first count all k-mers starting with AA, then AC, then AG, AT, CA, etc., which is equivalent to partitioning k-mer space into 16 equal-sized partitions. These partitions can be calculated independently, either across multiple machines or iteratively on a single machine, and the results stored for later comparison or analysis.

6 Conclusions

K-mer counting has been widely used in many bioinformatics problems, including data preprocessing for de novo assembly, repeat detection, sequencing coverage estimation. Here we present the khmer software package for fast and memory efficient counting of k-mers. Unlike previous methods bases on data structures like hash tables, suffix arrays, or trie structures, Khmer uses a simple probabilistic data structure, which is similar in concept to Count-Min sketch. It is highly scalable, effective and efficient in analyzing large next generation sequencing dataset involving k-mer counting, despite with certain counting error rate as tradeoff. We compared the memory usage, disk usage and time usage between our khmer program and other programs like Tallymer and Jellyfish to show the advantage of our method. The counting accuracy was also assessed theoretically and was validated using simulated data sets. Our k-mer counting approach can be used efficiently and effectively for any high diverse dataset with lots of low-abundance k-mers, like next generation sequencing data sets, which are biased towards low-abundance k-mers due to errors. We further showed applications of khmer software package in tackling problems like detecting sequencing errors in metagenomic reads and removing those erroneous reads to reduce data set size through efficient k-mer counting. Our approach can also be implemented parallelly or distributably to speed up or handle larger data sets with reasonable counting error rate.

References

[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[2] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

[3] Jonathan Butler, Iain MacCallum, Michael Kleber, Ilya A Shlyakhter, Matthew K Belmonte, Eric S Lander, Chad Nusbaum, and David B Jaffe. Allpaths: de novo assembly of whole-genome shotgun microreads. *Genome Res*, 18(5):810–20, May 2008.

[4] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, April 2005.

[5] Graham Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *SDM*, 2005.

[6] 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.

[7] Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, Songgang Li, Huanming Yang, Jian Wang, and Jun Wang. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res*, 20(2):265–72, Feb 2010.

[8] 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.

[9] Páll Melsted and Jonathan K Pritchard. Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC bioinformatics*, 12:333, January 2011.

[10] Jason R Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–27, Jun 2010.

[11] Junjie Qin, Ruiqiang Li, Jeroen Raes, Manimozhiyan Arumugam, Kristoffer Solvsten Burgdorf, Chaysavanh Manichanh, Trine Nielsen, Nicolas Pons, Florence Levenez, Takuji Yamada, Daniel R Mende, Junhua Li, Junming Xu, Songgang Shaochuan Shengting Li, Dongfang Li, Jianjun Cao, Bo Wang, Huiqing Liang, Huisong Zheng, Yinlong Xie, Julien Tap, Patricia Lepage, Marcelo Bertalan, Jean-Michel Batto, Torben Hansen, Denis Le Paslier, Allan Linneberg, H Bjø rn Nielsen, Eric Pelletier, Pierre Renault, Thomas Sicheritz-Ponten, Keith Turner, Hongmei Zhu, Chang Yu, Min Jian, Yan Zhou, Yingrui Li, Xiuqing Zhang, Nan Qin, Huanming Yang, Jun Jian Wang, Sø ren Brunak, Joel Doré, Francisco Guarner, Karsten Kristiansen, Oluf Pedersen, Julian Parkhill, Jean Weissenbach, Peer Bork, and S Dusko Ehrlich. A human gut microbial gene catalogue established by metagenomic sequencing. *Nature*, 464(7285):59–65, 2010.

[12] Florin Rusu and Alin Dobra. Sketches for size of join estimation. *ACM Transactions on Database Systems*, 33(3):1–46, August 2008.

[13] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven J M Jones, and Inanç Birol. Abyss: a parallel assembler for short read sequence data. *Genome Res*, 19(6):1117–23, Jun 2009.

[14] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Res*, 18(5):821–9, May 2008.

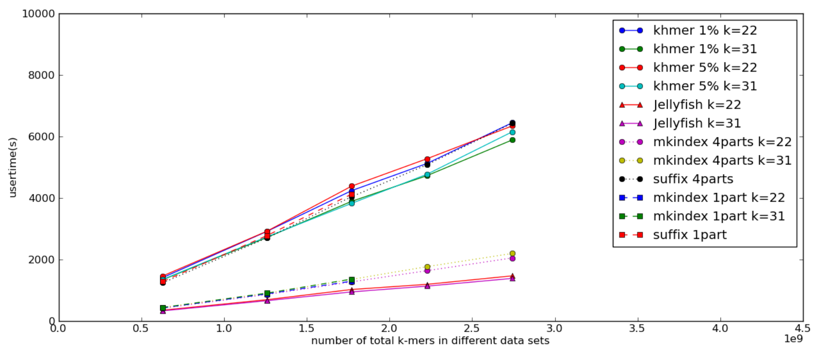

Figure 1: Time usage of different khmer counting tools

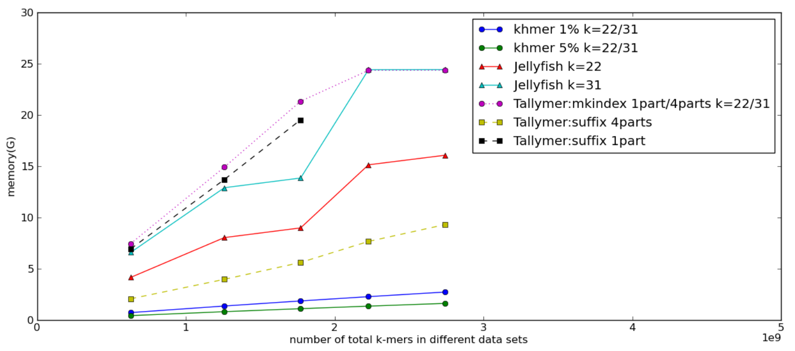

Figure 2: Memory usage of different k-mer counting tools

	size of file(GB)	number of reads	number of unique k-mers	total number of k-mers
dataset1	1.90	9,744,399	561,178,082	630,207,985
dataset2	2.17	19,488,798	1,060,354,144	1,259,079,821
dataset3	3.14	29,233,197	1,445,923,389	1,771,614,378
dataset4	4.05	38,977,596	1,770,589,216	2,227,756,662
dataset5	5.00	48,721,995	2,121,474,237	2,743,130,683

	Real metagenomics reads	Totally random reads with randomly generated k-mers	Simulated reads from simulated genome with error	Simulated reads from simulated genome without error
Size of data set file	7.01M	3.53M	5.92M	9.07M
Number of total k-mers	2917200	2250006	3757479	5714973
Number of unique k-mers	1944996	1973430	1982403	1991148

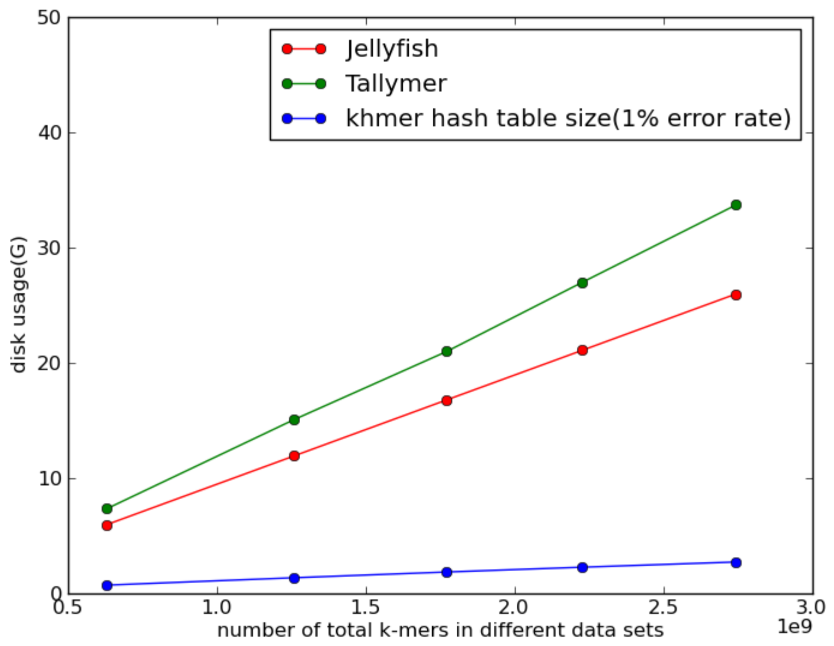

Figure 3: Disk storage usage of different k-mer counting tools

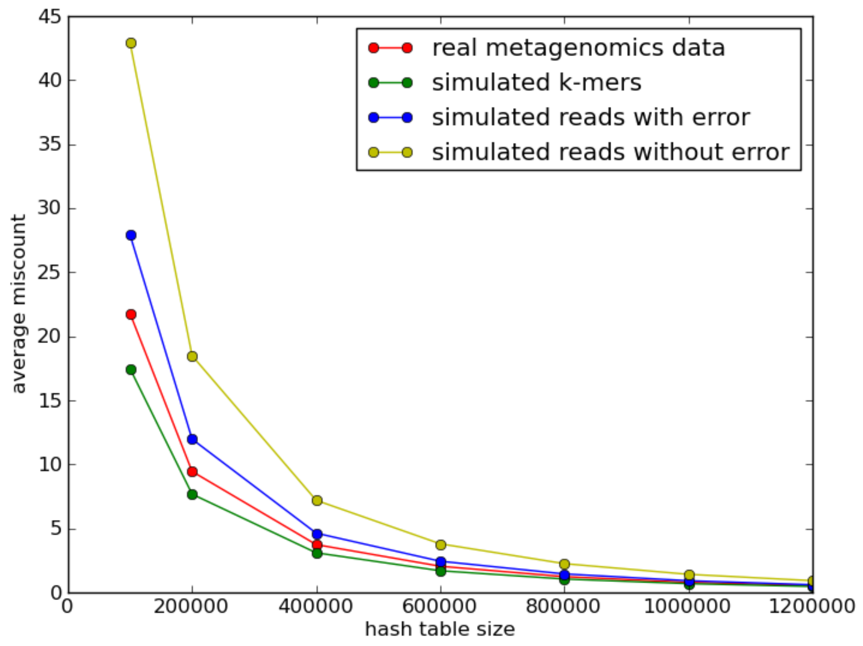

Figure 4: average miscount with different hash table size

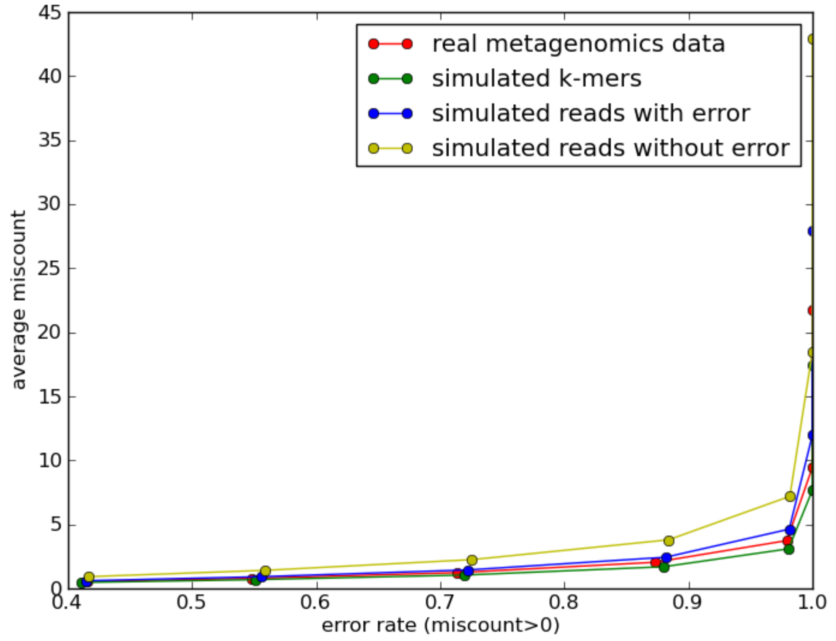

Figure 5: relation between average miscount and counting error rate

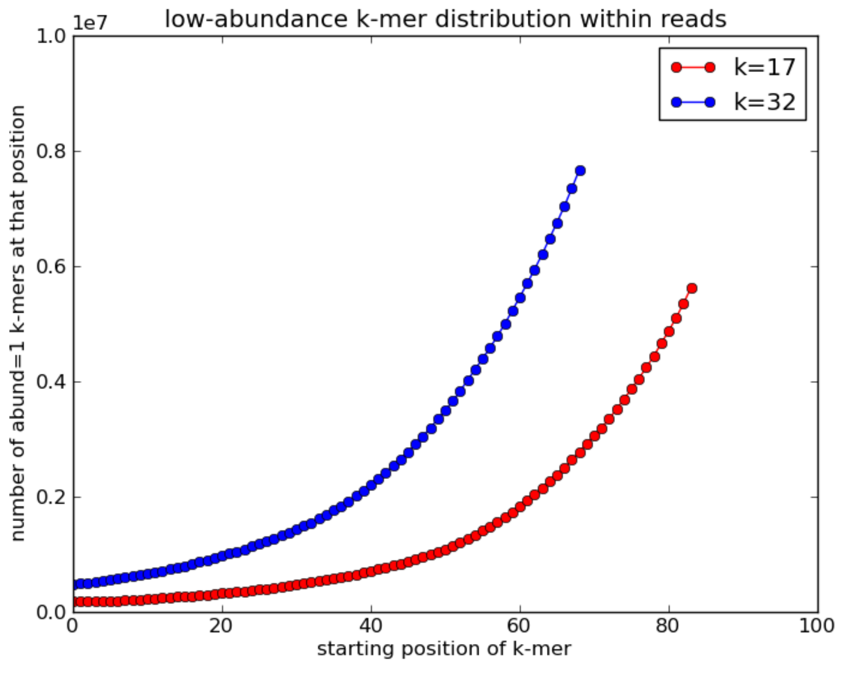

Figure 6: Percentage of the unique k-mers starting in different position in reads

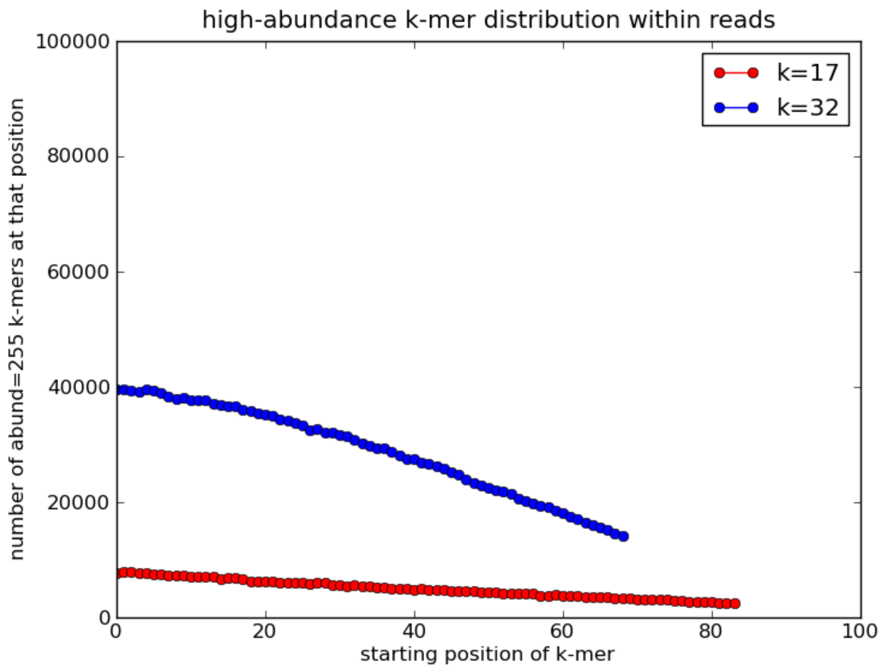

Figure 7: Percentage of the high abundance k-mers starting in different position in reads

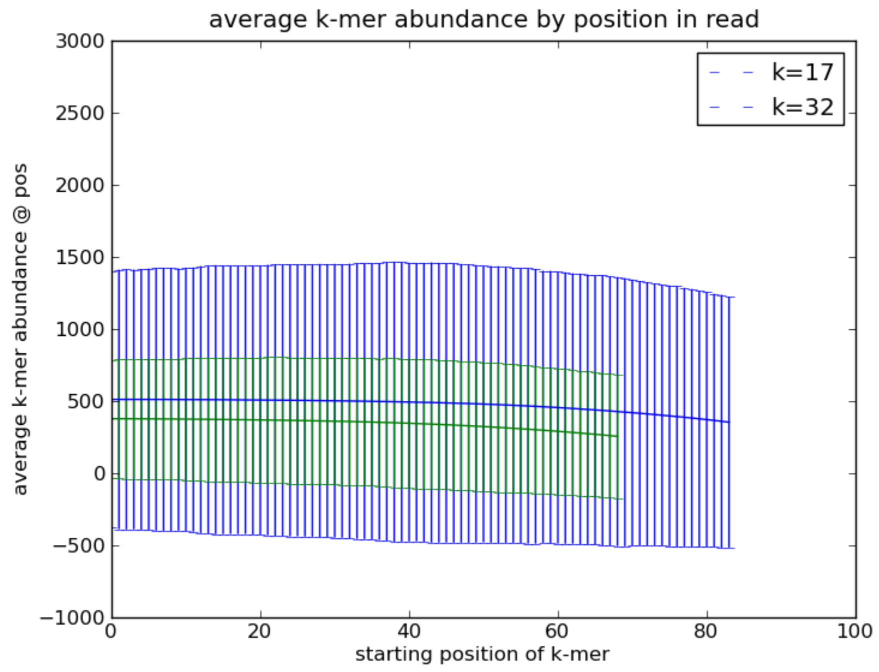

Figure 8: Average frequency of k-mers starting in different position in reads

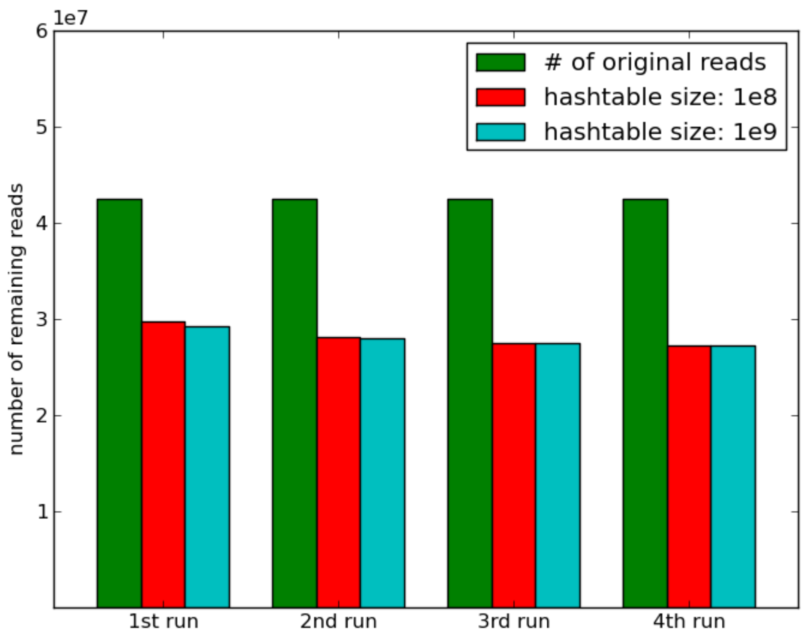

Figure 9: Number of remaining reads after iterating filtering out low-abundance reads that contain even a single unique k-mer with hash tables with different sizes(1e8 and 1e9) for a human gut microbiome metagenomic dataset(MH0001, 42,458,402 reads)

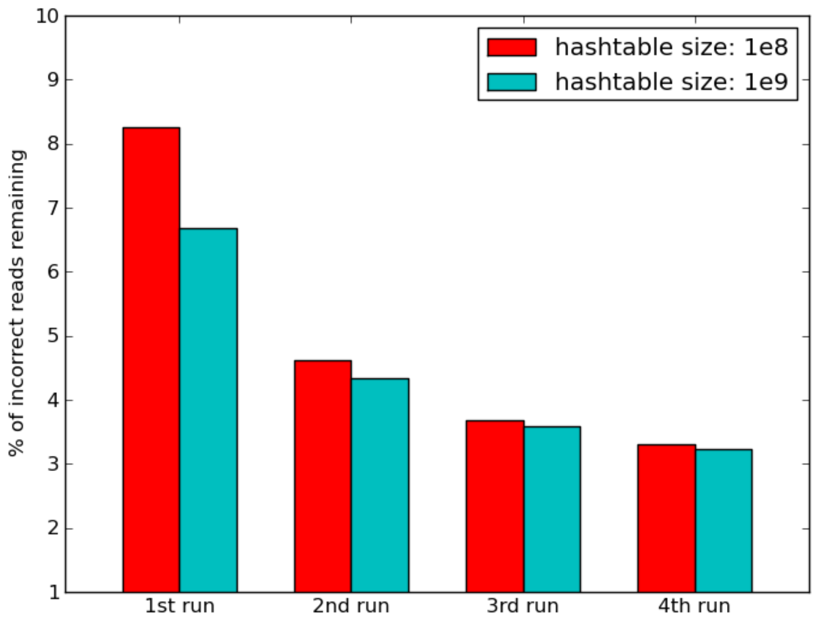

Figure 10: Percentage of incorrect reads in the remaining reads after iterating filtering with hash tables with different sizes(1e8 and 1e9) for a human gut microbiome metagenomic dataset(MH0001, 42,458,402 reads)