A probabilistic approach to k-mer counting

QP, Jason, Rose, Adina, CTB

September 25, 2012

1 Abstract

1.1 Background:

K-mer counting has been widely used in many bioinformatics problems, including data preprocessing for de novo assembly, repeat detection, sequencing coverage estimation. However current available tools cannot handle the high throughout data generated by next generation sequencing technology efficiently due to high memory requirements or impractically long running time.

1.2 Results:

Here we present khmer software package for fast and memory efficient counting of k-mers. Unlike previous methods bases on data structures including hash tables, suffix arrays, or trie structures, khmer uses a simple probabilistic data structure, Bloom counting hash, which is similar in concept to the Bloom filter and Count-Min sketch. It is highly scalable and has the ability to handle huge amount of data, which is difficult to deal with using other methods. We compared the memory usage, disk usage and time usage between our khmer software and other methods like Tallymer and jellyfish to show the advantage of our method. The counting accuracy was evaluated besides. Finally we presented two applications of khmer software in evaluation of sequencing error by k-mer abundance distribution in reads and in filtering reads for efficient de Bruijn graph-based assembly.

1.3 Conclusion:

The khmer software package is an effective and efficient tool in applications involving k-mer counting to analyze large next generation sequencing data set, despite with certain miscount rate as tradeoff. Some

analysis to data set that is too large to be handled by other available software becomes practical using our khmer software.

2 Background

A k-mer is a substring with length k in a sequence. K-mer counting is the problem to determine the occurrences of such k-mers in a dataset composing of sequences. Efficient k-mer counting plays an important role in solving many bioinformatics problems.

One important problem is de novo assembly of very large number of short reads. With the development of next generation sequencing technology, many research groups can afford the sequencing of the sample of specific species or even metagenomic samples with numerous different species[10]. Large amount of next generation sequencing short reads are generated and de novo assembly is required for these sequence data sets[11]. Currently, de Bruijn graph method is popular in the attempts to do de novo assembly because of its advantage in assembling next generation sequencing short reads[12]. Several popular assemblers have been developed based on de Bruijn graph, such as Velvet[16], ALLPATHS[3], ABySS[15] and SOAPdenovo[7]. All the k-mers in a sequence data set are represented as large number of nodes in the de Bruijn graph. If two k-mers have an overlap of (k-1)-mer, the two k-mers can be connected. Since k-mer is such a basic cornerstone in de Bruijn graph de novo assembly, it is of great importance to determine the occurrence of k-mers. One example is that sequencing errors can generate many erroneous unique k-mers and we can filter out the reads with too many unique k-mers before doing the assembly. Similarly, we can also filter out the reads with k-mers that occur too many times for smoothing MDA-abundance reads dataset. Pre-filtering reads to reduce the size of reads data set to assemble is important to reduce the time and memory usage. Besides, we can also use k-mer counting to evaluate the size of genome and the coverage of the sequencing reads.Besides,k-mer counting can also give important information for predicting regions with repetitive elements as transposons with important biological function.[6]

Current methods to do k-mer counting involve the data structures like hash tables, suffix arrays, binary trees or tries structures. If the size of sequence dataset to count is modest, a simple hash table can do the counting, where the k-mers are the keys and the the corresponding counts are the values. However the disadvantage is evident. If the size of sequence dataset is larger, the efficiency of the counting using a simple hash table plummets dramatically. Another k-mer counting tool - Tallymer, uses suffix array data structure.[6] It is more efficient than simple hash table in general. However the memory requirement is linear to the number

of unique k-mers. So this method is not so scalable. If the size of dataset is too large, which is normal for today's project with the help of next generation sequencing technology, the cost in time and memory is intolerable. For example, in a soil metagenomics project, the size of reads data set generated from one soil sample have already exceeded 400G bytes, with only a limited coverage. (to cite GPGC) Reads dataset with size like this is difficult for Tallymer to handle. Jellyfish, [8] another counting tool released recently, uses updated hash table data structure, which can reduce the memory requirement to store k-mers and the "lock-free" feature of the hash table also realize the parallelism of Jellyfish to make it more efficient. But it still has the same problem as Tallymer of linear increase of memory usage with respect to the number of unique k-mers in data set to count.

Here we present Bloom counting hash, a simple probabilistic data structure, which is similar in concept to Bloom filter[1] [2] [9]and Count-Min sketch[4], to show a new approach to count k-mer effectively and efficiently. It is highly scalable and can be used to count k-mers in really large sequence data set with a tradeoff of one-side counting inaccuracy.

3 Method

3.1 Sequence data sets

Two human gut metagenomics reads datasets(MH0001 and MH0002) were taken from MetaHIT (Metagenomics of the Human Intestinal Tract) project.[13] Total DNA was extracted from the fecal specimens from 2 healthy human adults and sequenced by Illumina GA technology. There are about 59 million 44bp-long reads in MH0001 as a 2.61Gb fq file. There are about 61million 75bp-long reads in MH0002 as a 4.58 Gb fq file. Next we trimmed the reads to get rid of low quality sequences and got 3.2Gb and 4.2Gb fasta files separately for MH0001 and MH0002.

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

Four short reads data sets are generated to test the assessment of false positive rate and miscount distribution. One is a subset of a real metagenomics data set from MetHit project (MH0001). The second one is a reads data set with all k-mers in it generated randomly. The other two are two reads data sets simulated from a genome with 1 million base pairs, which is also simulated randomly. One of them is with error generated

randomly with error rate as 3%. The other one is without any error. The four data sets are generated with different size to include similar number of unique k-mers. (k=12)

3.2 Comparing with other k-mer counting programs

Tallymer is from the genometools package version 1.3.4. It was run with options as below: For suffixerator subroutine: -dna -pl -tis -suf -lcp db

For subroutine tallymer mkindex: -mersize 32 -esa 3

Jellyfish is version 1.1.2 and multithread option is off.

jellyfish count -m 22 -c 2 -C -s 100000000
jellyfish merge -o jelly.merged mer_counts_*
jellyfish stats -o jelly.stats jelly.merged

4 Algorithm

4.1 A constant-memory probabilistic data structure for k-mer counting

We present a simple probabilistic data structure, a Bloom counting hash for fast and memory efficient counting of k-mers. The concept of the Bloom counting hash is similar to Bloom filter, with several difference. Firstly, the Bloom counting hash consists of one or more hash tables of different size instead of one 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"0" or "1" in a typical Bloom filter to represent the existence of an item. In our implementation, The hash function is to take the modulus of a number representing the k-mer with the table size. Given a new k-mer, it is hashed to a specific location in each hash table using the 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 so collisions happen. The effect of the collisions is that the counter in a location in a hash table may not represents 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 counter of 5 in one hash table, which means it 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 concept of Count-Min sketch, in our implementation, we pick the minimum counter over all the hash tables for a k-mer

to represent its occurrence to reduce the effect of collisions. If accurate counting of k-mers is really required, we also implement an extension to counting the high abundance k-mers accurately.

Firstly, like Count-Min sketch, this is a probabilistic data structure and the counting is not exactly correct. There is one-side error, which is, any count may be equal to or bigger than the actual count, but cannot be smaller than actual count because of the collision in the Bloom filter like data structure. The chosen parameters of the data structure will influence the accuracy of the count, which can be estimated well like Count-Min sketch[4]. In fact such probabilistic properties suit the next generation sequencing short reads data sets well. Firstly, the counts are not so wrong for next generation sequencing reads data sets because of the generally skew 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 so required. For example, when we want to get rid of reads with low abundance k-mers, in practice, 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 more and more reads with low abundance k-mers can be thrown after every iteration. Furthermore the rate of inaccurate count and false filtering can be predicted pretty well. We can twist 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 more memory we can use, 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, 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 get worse and worse.

4.2 Assessment of counting error

We want to use the Bloom counting hash 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 is $e^{-N/H}$. The individual collision rate in one hash table is $1-e^{-N/H}$. The total collision rate, which is the probability that collision happened in all the entries where a k-mer is hashed to in all Z hash tables, will be $(1-e^{-N/H})^Z$. Only in this situation, all the counters in all Z hash tables cannot give the true count of a k-mer because of the collisions.

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 figure out how bad a count is if it is not correct, in other word, what the offset between an inaccurate count and the actual count is. According to the analysis of Count-Min sketch paper,[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 hash table. Specifically, suppose d hash tables with size as w are used in the counting to 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 is a distribution independent confidence bound and this error estimation is a worst-case upper bound and too conservative. In practice, for some data sets, the error estimation is reasonable, for some other data sets, the data structure we are using can outperform the theoretical worst-case bounds in many orders of magnitude. Further study shows that the behavior of Count-Min sketch depends on the characteristics of data set, like if it is skewed or not. [14] Generally, for low skew data set in which the k-mers have uniform distribution of frequency, the k-mers can be distributed into hash tables evenly with high probability. So the average miscount is related to the average abundance of k-mers in hash tables (T/w). In this case, the error estimation is closer to the actual error. Nevertheless for high skew dataset 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 and this results in fewer collision in the counting hash table in general[5]. So the minimum count in the hash tables will have smaller miscount and the average miscount will be smaller than that for low skewed data. This means for high skewed dataset, the error estimation should be tighter than the confidence bound described above. If the skew of data set can be determined by other methods, a higher bound can be acquired according to 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 number of total 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. With certain number of total k-mers in a data set and certain hash table size, skew abundance distribution will have smaller miscount generally than even abundance distribution.

4.3 Analysis of memory usage and running time

Not like many other approaches to count k-mers, memory usage of our bloom count approach is independent of the size of dataset. It only depends on the size(S) and number(N) of hash tables to use. O(S)*O(N) As discussed above, with certain dataset, the size and number of hash tables will determine the accuracy of k-mer counting. So the user can control the memory usage totally according to their requirement of accuracy. As to the time usage, firstly in the first step of the counting, which is to consume all k-mers in a data set and store the occurrence in hash tables, the time usage depends on the number of total k-mers(T), which is related to the size of data set. O(T). The second step of the counting is the retrieving of count. So in this step, the time usage depends on the number of unique k-mers whose count we want to know.

4.4 Implementation

This approach has been implemented in a software named khmer, written in C++ with a Python wrapper and freely available under the BSD license. http://github.com/ged-lab/khmer. The scripts for the benchmark are also included in the khmer repository as well as many handy scripts that can be used for many applications.

5 Result and discussion

5.1 Speed and memory usage on soil metagenomic reads data sets

To show the time usage and memory usage of the khmer counting approach, we counted the number of unique k-mers in 5 soil metagenomic reads data sets with different size.

The time usage and memory usage for counting k-mers in 5 soil metagenomic reads data sets using khmer and other two programs 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 the other two programs. From Figure 2 we can see that the memory usage of both Jellyfish and Tallymer increases linearly with data set size, although Jellyfish is more efficient than Tallymer in memory usage. This brings a big problem. Nowadays the size of next generation sequencing reads data set grows bigger and bigger, the memory usage is becoming a heavy burden. For a 5G dataset with 2.7 billion total k-mers, Jellyfish uses 5G memory and Tallymer cannot handle a 4G dataset with a machine with 24G memory. So both tools cannot satisfy the k-mer counting requirement for metagenomic data sets. It looks like 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. 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 data set size. Here for a dataset of 5 gigabytes, the disk usage of both Jellyfish and Tallymer is around 30 gigabytes. For a metagenomic data set as large as hundreds of gigabytes, the huge disk usage is annoying and cannot be neglected. The curve is the size of hash tables used in our khmer approach with fixed counting error rate. There is an option in our implementation to store the hash tables filled with count on hard disk for future use. But the khmer can do everything without doing that

From this comparison, we can conclude that for large metagenomic dataset, other k-mer counting programs like Jellyfish and Tallymer fail to do the counting successfully because of the high memory usage 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 specific analysis with specific limited memory to use.

5.2 Assessment of false positive rate and miscount distribution

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 dataset. Like metagenomics dataset, a large amount of the k-mers in such high diversity dataset 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. And the average offset is closely related to the number of total k-mers. For example, simulated reads without error has the most k-mers out of the four data sets. And the average miscount is the highest. This observation proves the conclusion discussed in section 4.1.2, which is, number of total k-mers in data set and hash table size will influence how bad the counts are if they are not correct.

Figure 5 shows the relationship between average miscount and counting error rate for different test data sets. For fixed counting error rate, simulated reads without error has the highest average miscount and simulated k-mers has the lowest average miscount. It is because they have the highest and lowest number of total k-mers separately. 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. So here 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

even unique due to sequencing errors.

5.3 Investigation of sequencing error pattern by k-mer abundance distribution

When dealing with large amount of sequencing data, one annoying concern is the sequencing error. Generally the sequencing error occurs randomly. So if k is large enough, most of the erroneous k-mers should be unique in the reads dataset. How to detect those sequencing errors and correct or remove them from reads is always a challenge. Before doing any data analysis, we should have some idea about the sequencing error, like the estimation of sequencing error rate, or the distribution of sequencing error rate. Generally 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 sequencing error pattern of an e.coli Illumina reads data set 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 relatively 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.

Here we show that using our k-mer counting approach to do the 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 proper filtering strategy or choosing appropriate filtering threshold, which is otherwise a crucial step in preprocessing data for other data analysis like assembly.

5.4 Removing reads containing low-abundance k-mers to reduce size of data set for efficient assembly

An important approach to assemble short reads from next generation sequencer is the de Bruijn graph, which relies on k-mer graph. This approach has been applied with some success to human microbiome data. Because of technical limit, there are sequencing errors in the reads, which bring about erroneous k-mers. As discussed in section 5.3, we can detect such erroneous k-mers by detecting low frequency k-mers. So removing or trimming reads containing unique or low-abundance k-mers will remove many errors. On the other hand, low-abundance k-mers, even some of them are genius, do not contribute much to the de Bruijn graph assembly. So before we do the assembly, we want to filter out the reads with low frequency k-mers to decrease memory usage and time usage.

The Bloom counting hash is an efficient and constant-memory data structure 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 of the Bloom counting hash manifests as a too high 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 removed will be correctly removed. 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 the Bloom counting hash.

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 get the actual accurate count of the k-mers in the dataset.

From Figure 9 , we can 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 novel 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.

5.5 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 quite a bit of memory: approximately 446 Gb of memory are 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 A partitions results in a linear decrease in the number of k-mers under consideration, thus reducing the occupancy by a constant factor A 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. However current available tools cannot handle the high throughout data generated by next generation sequencing technology efficiently due to high memory requirements or impractically long running time. Here we present the khmer software package for fast and memory efficient counting of k-mers. Unlike previous methods bases on data structures including hash tables, suffix arrays, or trie structures, Khmer uses a simple probabilistic data structure, which is similar in concept to Bloom filter and Count-Min sketch data structure. It is highly scalable, effective and efficient in applications involving k-mer counting to analyze large next generation sequencing dataset, 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 counting approach can be used efficiently and effectively for any high diverse data set 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] Michael L. Metzker. Sequencing technologiesi: the next generation. *Nat Rev Genet*, pages 31–46, 2010.

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

[12] P A Pevzner, H Tang, and M S Waterman. An eulerian path approach to dna fragment assembly. *Proc Natl Acad Sci U S A*, 98(17):9748–53, Aug 2001.

[13] 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.

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

[15] 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.

[16] 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.

	size of file(GB)	number of reads	number of unique k-mers
dataset1	1.90	9,744,399	561,178,082
dataset2	2.17	19,488,798	1,060,354,144
dataset3	3.14	29,233,197	1,445,923,389
dataset4	4.05	38,977,596	1,770,589,216
dataset5	5	48,721,995	2,121,474,237

	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

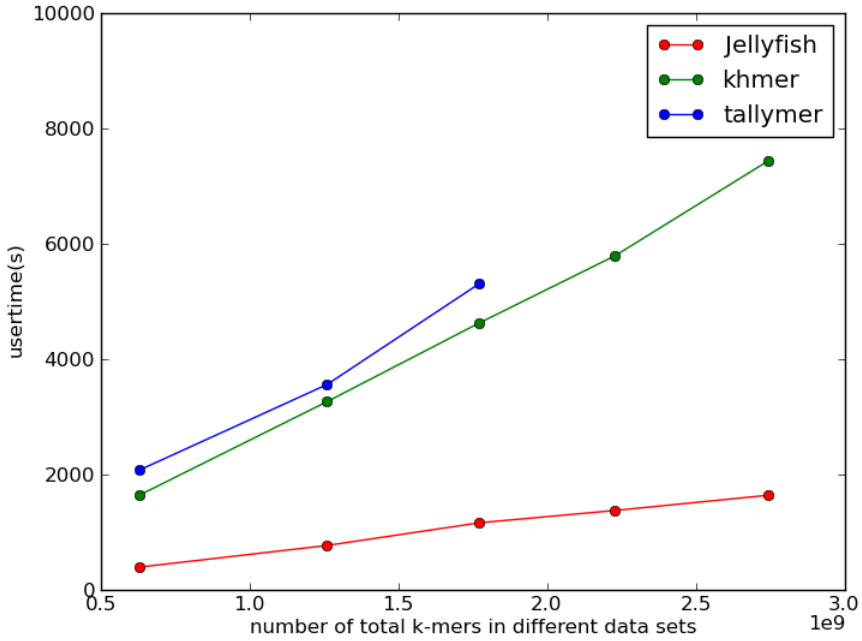

Figure 1: Time usage of different khmer counting tools

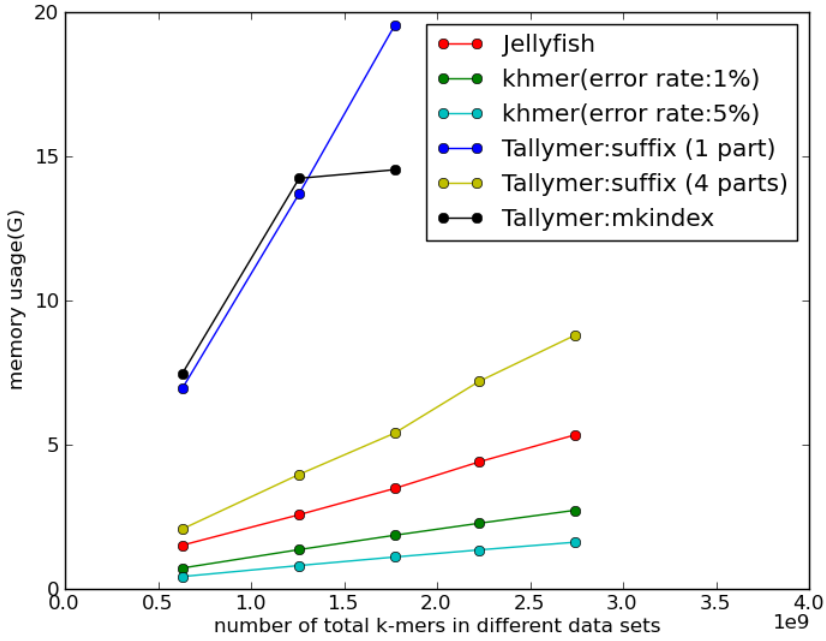

Figure 2: Memory usage of different k-mer counting tools

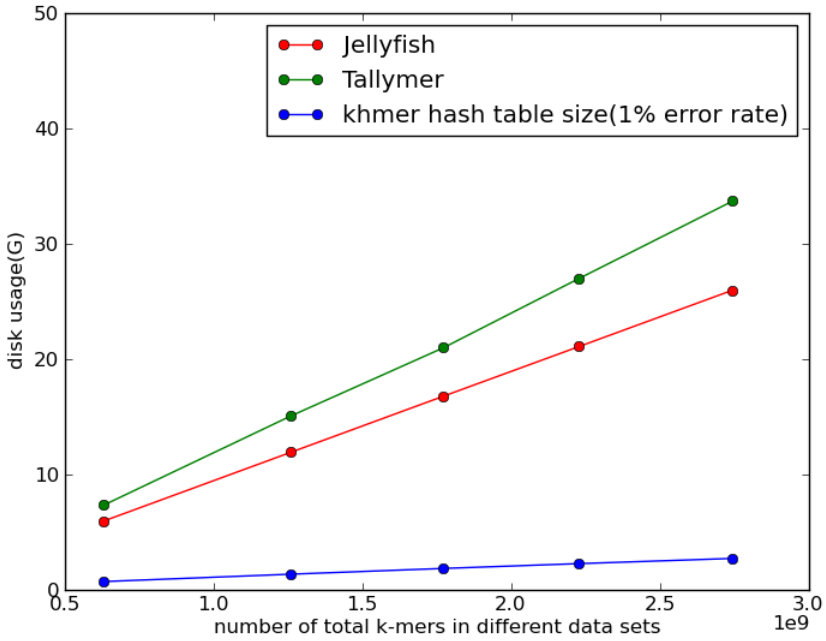

Figure 3: Disk storage usage of different k-mer counting tools

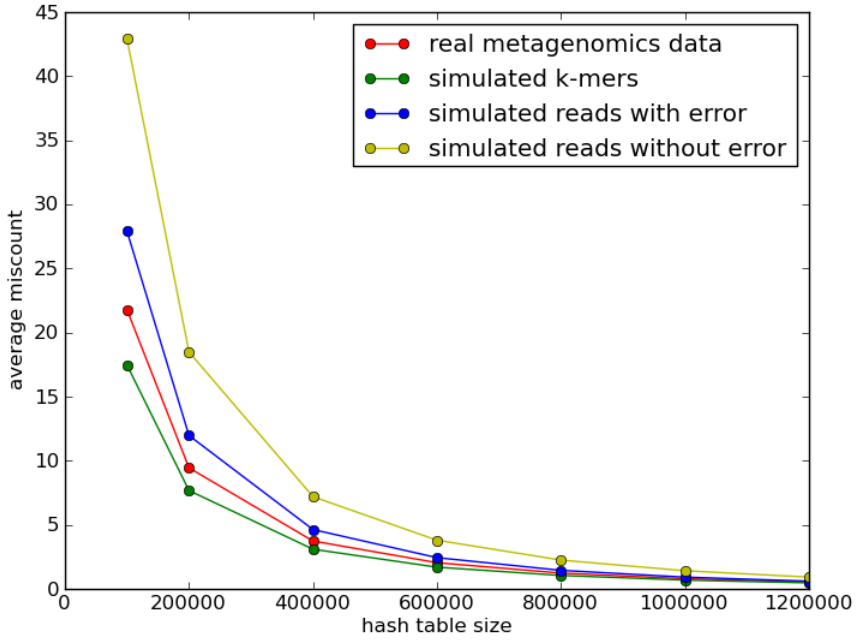

Figure 4: average miscount with different hash table size

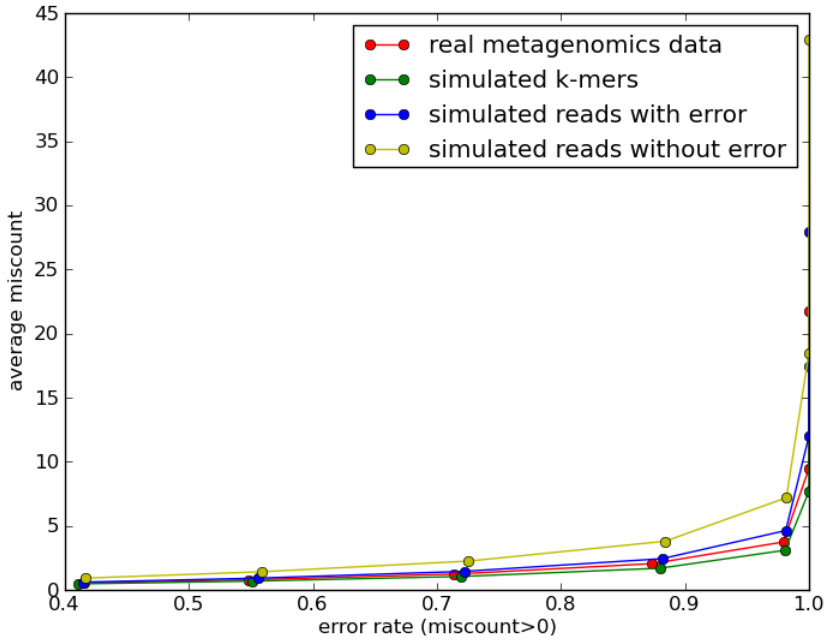

Figure 5: relation between average miscount and counting error rate

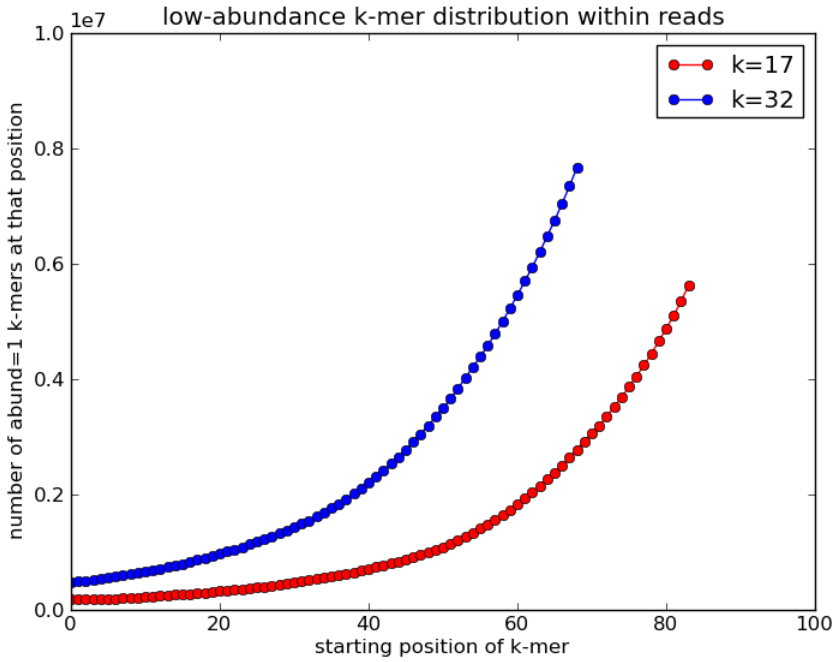

Figure 6: Percentage of the unique k-mers starting in different position in reads

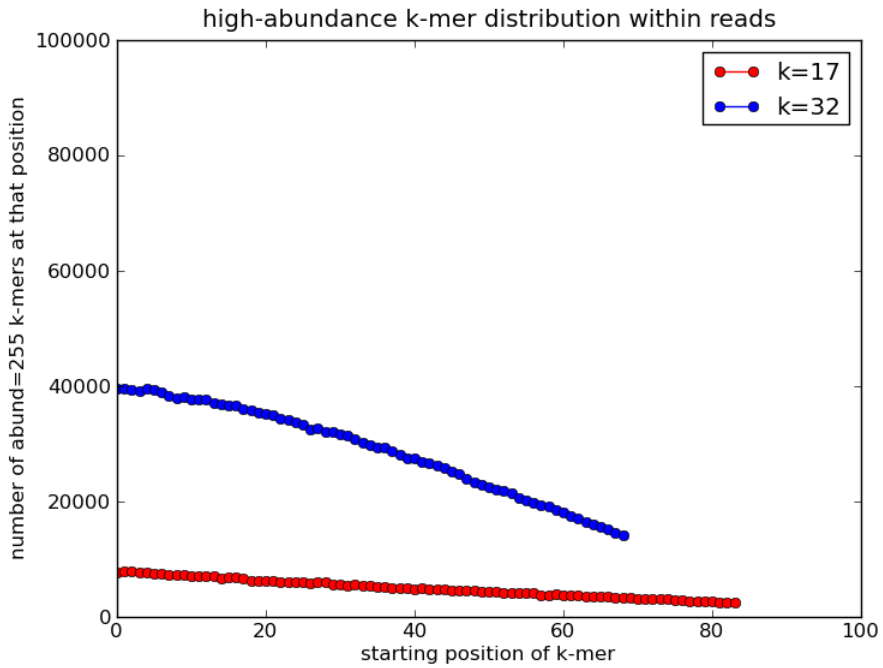

Figure 7: Percentage of the high abundance k-mers starting in different position in reads

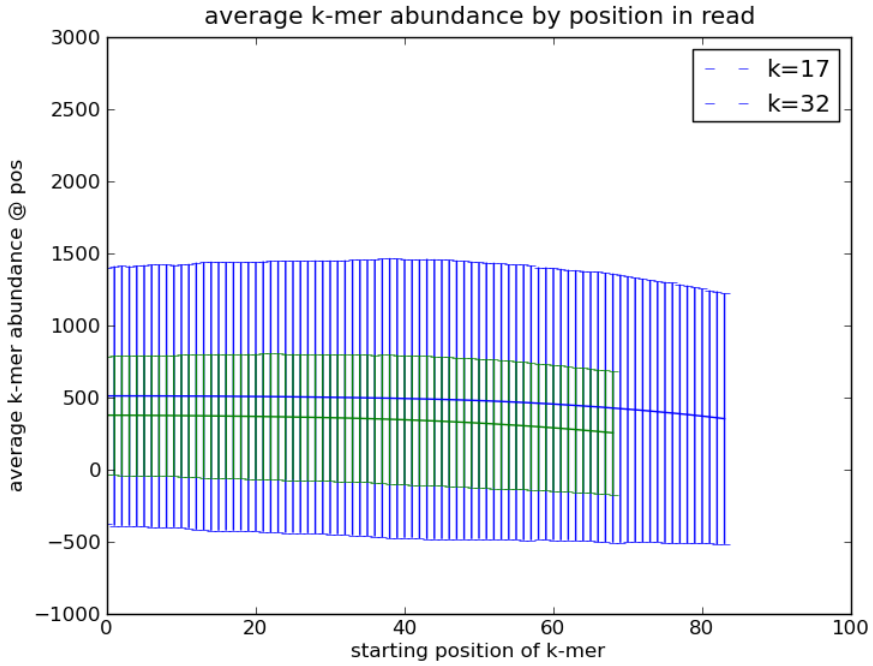

Figure 8: Average frequency of k-mers starting in different position in reads

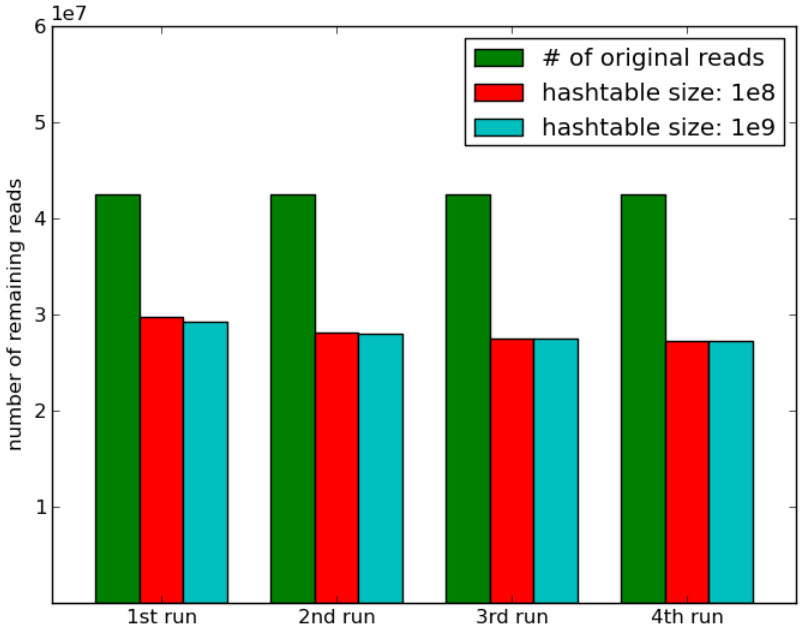

Figure 9: Number of remaining reads after iterating filtering out low-abundance reads that contain even a single unique k-mer with hash tables with different sizes(1e8 and 1e9) for a human gut microbiome metagenomic dataset(MH0001, 42,458,402 reads)

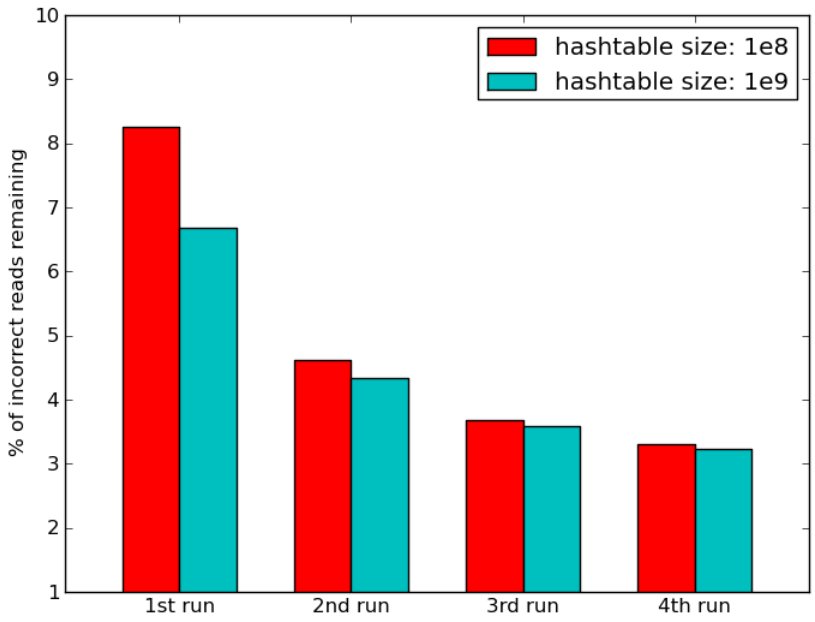

Figure 10: Percentage of incorrect reads in the remaining reads after iterating filtering with hash tables with different sizes(1e8 and 1e9) for a human gut microbiome metagenomic dataset(MH0001, 42,458,402 reads)