Send Orders for Reprints to reprints@benthamscience.net

REVIEW ARTICLE

Estimating the *k*-mer Coverage Frequencies in Genomic Datasets: A Comparative Assessment of the State-of-the-art

Swati C. Manekar* and Shailesh R. Sathe

Department of Computer Science and Engineering, Visvesvaraya National Institute of Technology, Nagpur, India

ARTICLE HISTORY

Received: July 23, 2018
Revised: October 05, 2018
Accepted: October 24, 2018

DOI:
10.2174/1389202919666181026101326

Abstract: ***Background*:** In bioinformatics, estimation of *k*-mer abundance histograms or just enumerating the number of unique *k*-mers and the number of singletons are desirable in many genome sequence analysis applications. The applications include predicting genome sizes, data pre-processing for de Bruijn graph assembly methods (tune runtime parameters for analysis tools), repeat detection, sequencing coverage estimation, measuring sequencing error rates, *etc.* Different methods for cardinality estimation in sequencing data have been developed in recent years.

***Objective*:** In this article, we present a comparative assessment of the different *k*-mer frequency estimation programs (ntCard, KmerGenie, KmerStream and Khmer (abundance-dist-single.py and unique-kmers.py) to assess their relative merits and demerits.

***Methods*:** Principally, the miscounts/error-rates of these tools are analyzed by rigorous experimental analysis for a varied range of *k*. We also present experimental results on runtime, scalability for larger datasets, memory, CPU utilization as well as parallelism of *k*-mer frequency estimation methods.

***Results*:** The results indicate that ntCard is more accurate in estimating F_0, f_1 and full *k*-mer abundance histograms compared with other methods. ntCard is the fastest but it has more memory requirements compared to KmerGenie.

***Conclusion*:** The results of this evaluation may serve as a roadmap to potential users and practitioners of streaming algorithms for estimating *k*-mer coverage frequencies, to assist them in identifying an appropriate method. Such results analysis also help researchers to discover remaining open research questions, effective combinations of existing techniques and possible avenues for future research.

Keywords: *K*-mer abundance histogram, High-throughput sequencing, Hashing, Streaming algorithms, Singleton *k*-mers, Distinct *k*-mers.

1. INTRODUCTION

In *k*-mer counting, the occurrences of fixed length substrings of length *k* (*k*-mers) in DNA/RNA sequence or set of sequences are counted [1]. *k*-mer counting is an essential preliminary step in many bioinformatics applications. These applications include genome assembly [2-7], error correction to improve genome assembly [8-12], repeat detection and annotation [13-18] and many others. Hence, in recent years, a large number of *k*-mer counting programs have been designed, such as Jellyfish [1], KMC3 [19], Gerbil [20], DSK [21], *etc.* Most of the early *k*-mer counting tools were based on the in-memory approach. The new tools make use of disk [19-21] or succinct and compact data structures [22] in their implementations. Due to such approach and data-structures, exact *k*-mer counting tools work efficiently for high-throughput sequencing data with billions of reads, on commodity hardware. The time and memory required have been reduced considerably using these improved methods. Such exact *k*-mer counting methods generate output as distinct *k*-mers along with frequencies. From such output, a *k*-mer abundance histogram can be easily constructed. Generating such histograms require storing the large number of *k*-mers in memory. For the exact *k*-mer counting, it is mandatory to store all the possible *k*-mer of all input reads in memory or disk, where all the *k*-mer are processed base by base. The great demand for rapid and reasonable DNA sequencing resulted in the improved sequencing technologies that produce enormous amounts of data at ever decreasing costs. For sequencing applications, where the sequence data are used to infer abundances or other molecular activity, the streaming approaches offer significant opportunities to make on-the-fly comparisons allowing data to be discarded rapidly [23]. Digital normalization (a lossy compression algorithm) is implemented as a streaming algorithm for the elimination of redundant reads in sequence datasets containing millions to billions of short reads based on saturating coverage of a de Bruijn graph [24]. With the increase in dataset sizes, stream-

*Address correspondence to this author at the Department of Computer Science and Engineering, Visvesvaraya National Institute of Technology, Nagpur - 440010, India; Tel/Fax: (+91-712-) 2222828, 2801294, 2231636, 2226750; E-mail: swati.manekar@gmail.com

ing approaches are potentially more promising to the analysis of sequencing data. For tremendous amounts of Next-Generation Sequencing (NGS) data, exact *k*-mer counting approach can be both time and memory consuming. Though the efficient disk and in-memory approaches are available for *k*-mer counting, counting all *k*-mers in this data may require tens or even hundreds of gigabytes of memory and even some days. Whereas, the approximation of the *k*-mer counts histogram by streaming analysis of the data is both memory and time efficient [25-30]. Over the last few years, many streaming algorithms using probabilistic data structures [25-30] have been developed to scale *k*-mer frequencies analysis methods to the rapidly growing amount of NGS data.

A framework for streaming algorithms was first proposed by Alon *et al.* [31]. The basic fundamental behind giving the approximate count is to sample the data in intervals and make estimation rather than processing the entire data (process each *k*-mer). Streaming algorithms are useful for solving data analysis problems normally in a single pass, where the data are so huge to easily be stored (or processed) in available memory [24]. The *k*-mer abundance histogram is a table of the f_i values, where f_i is the number of distinct *k*-mers that appear *i* times in a given sequencing dataset [29]. With the frequencies f_i of the histogram given, the *k*-th frequency moment, F_k, is defined as follows [29]:

$$F_k = \sum_{i=1}^{\infty} i^k . f_i$$

The zeroth-frequency moment of the reads refers to the number of distinct *k*-mers that occur in the set of reads and is denoted F_0 [32]. Flajolet and Martin [33] designed the first algorithm for approximating F_0 in the data stream model. The problem of estimating F_0 is also addressed (improvement in time/space tradeoffs) by Bar-Yossef *et al.* [32]. The first-frequency moment F_1 is simply the total number of *k*-mers in the reads and can be simply counted by incrementing a counter one time for each *k*-mer [29]. The estimation of second-frequency moment F_2 was first solved by Alon, Matias and Szegedy [31]. F_2 is the repeat rate and is of particular interest because the repeat rate provides useful statistics on the reads. Cormode and Muthukrishnan [34] examined F_∞ which gives the highly frequent *k*-mers in the input reads [28]. The streaming algorithms for each of the frequency moments can perform their estimations within a factor of $(1 \pm \boldsymbol{\varepsilon})$ with high probability utilizing only $O(\boldsymbol{\varepsilon}^{-2}\log(N))$ memory, where *N* indicates the number of distinct *k*-mers [29].

Approximation of the *k*-mer counts histogram can be an order of magnitude faster and uses only a fraction of the memory compared with exact disk-based or in-memory *k*-mer counting algorithms. Hence, in a recent years, the programs estimating the *k*-mer frequencies of sequencing data (*i.e.,* the number of distinct *k*-mers (F_0), the number of *k*-mers that appear exactly once in the set of reads (f_1) and/or the *k*-mer coverage frequency histograms), such as, ntCard [28], KmerStream [29], KmerGenie [30] and Khmer (unique-kmers.py and abundance-dist-single.py) [26] have been designed.

The estimated *k*-mer frequencies are useful in many applications in nucleotide sequence analysis, such as, predicting genome sizes [1], measuring sequencing error rates sampled in the sequencing experiment [29], in genome assembly tools based on the de Bruijn graph framework where the best possible value of *k* is estimated using the generated abundance histograms [30]. *k*-mer abundance analysis is also used in sequencing coverage estimation [27], repeat detection [27] and in estimating heterozygosity of genomes [35]. Estimation of *k*-mer cardinality is also valuable in choosing the initial memory allocation for Bloom filters and Count-Min Sketches [27] like data structures. It has application in the estimation of memory requirements for de Bruijn graph assemblers [6]. For example, a fast sequence categorization tool (BioBloom tools) which utilizes Bloom filters, makes use of estimated *k*-mer frequencies to set the Bloom filter size [36].

With the different streaming programs available for *k*-mer abundance analysis, it becomes difficult for users to know which ones to use and when. There is no study that solely compares the performances of such tools. Zhang *et al.* [27] and Perez *et al.* [37] have compared both the exact and approximate *k*-mer counting tools in their study. To have reasonable comparisons, we present our study which considers only those tools estimating the *k*-mer frequency statistics using a streaming algorithm. We perform the analysis of the most relevant work in this area to date using several datasets of varying size and provide guidelines for their practical use, as well as point out areas of improvement for further research. The accuracy of estimated output from each tool is checked by comparing with the exact results of DSK (an exact *k*-mer counter). Besides, we also evaluate tools on various other parameters, such as their main approach, parallelism, Runtime, Memory (RAM), Central Processing Unit (CPU) utilization, scalability to large values of *k* and large-sized datasets.

This paper is organized as follows: Section 2 reviews different approaches for estimating *k*-mer coverage frequencies and F_0. Section 3 presents the evaluation methodology and datasets used. Section 4 is devoted to the benchmarking experiment and analysis of results. Conclusions and future research directions are given in the last section.

2. APPROXIMATE *K*-mer COUNTING APPROACHES

2.1. KmerStream

KmerStream [29], a hashing based approach implements a streaming algorithm to estimate F_0, F_1 and f_1 in the set of sequencing reads. F_1 is simply the number of *k*-mers in the reads and is counted easily by maintaining a single counter that is incremented once for each *k*-mer. To estimate f_1 and F_0, KmerStream uses an approach based on the K-Minimum Value algorithm [32], where it samples the data streams at different rates and then selects the optimal sampling rate to obtain the best results. The approach can be explained briefly as follows:

A list of arrays (tables) is used where every element in the array acts as a *k*-mer counter (two bits counters to store values from 0 to 3). Firstly, reads are broken into *k*-mers and sampled. The hash value for each *k*-mers is generated. De-

pending upon the number of trailing zeros in the binary equivalent of the hash value, the table level (*i.e.,* table number) is decided for that *k*-mer whereas hashing decides its position in array (an index into the array) and the counter value is incremented. Depending on the following factors, the output is then estimated: total levels (*i.e.,* the number of arrays in the array list), the size of the array and the probability of the counter being zero and one. An accurate estimate of f_1 is obtained by extending the results of F_0.

2.2. ntCard

ntCard [28] uses the hashing approach to estimate the *k*-mer frequencies. The workflow of ntCard algorithm for estimating the *k*-mer coverage frequencies and F_0 in DNA sequence streams can be explained as follows:

The algorithm starts by hashing the *k*-mers in the read streams. 64-bit hash value is generated for each *k*-mer in input read streams using ntHash algorithm [38]. The first *s* bits of this 64-bit hash value are used as sampling rate parameter to obtain the sampled dataset with a cardinality of $1/2^s$. Only those *k*-mers are selected in the sampling step for which starting *s* bits of their hash values are zero. Last *r* bits of this 64-bit hash value are used to construct the array for frequencies of count *i*. This array of size 2^r holds the counts of all the sampled *k*-mers. Last *r* bits of the 64-bit hash value of sampled *k*-mers acts as the index into this array, where *r* is a resolution parameter. The value of *r* is selected in such a way that a trade-off between accuracy and computational resources be maintained. The values in the array represent approximate counts of sampled *k*-mers. Relative frequencies f_i (relative frequency count for all counter values) are then estimated statistically using the extension (array folding). Now, from this relative frequencies f_i (for $i \geq 1$) of sampled *k*-mers, estimated multiplicity frequencies of the entire population is computed. The histogram and the number of distinct *k*-mers (F_0) can be inferred through the statistical model. ntCard works efficiently to give accurate count with least error rates.

2.3. Khmer

Khmer [26, 27] uses a count-min sketch [34] which is a probabilistic data structure and also uses the Bloom filter [39] to identify the *k*-mers that repeat more than once. Khmer [27] generates an approximate multiset representation of *k*-mer counts. The brief outline of an algorithm can be given as follows:

The algorithm starts by creating the number of hash tables depending on the maximum false positives required. Then using bloom filters, the repeating *k*-mers are ensured. In the last, the *k*-mers are inserted and their counts are incremented. Khmer [25] is a toolkit for *k*-mer-based dataset analysis. Khmer [26] is the latest implementation, build on the top of the existing infrastructure of the Khmer library [40]. Khmer [26] implements the HyperLogLog algorithm for *k*-mers. The HyperLogLog sketch (HLL) [41] estimates the cardinality of a set, *i.e.* a number of distinct elements of a set. HyperLogLog algorithm (a probabilistic cardinality estimator) approximates the number of distinct elements in a multiset and the algorithm works as follows: A multiset of uniformly distributed random numbers is obtained by applying a hash function to each element in the original multiset. By calculating the maximum number of leading zeros in the binary equivalent of each number in the set, the cardinality of this randomly distributed set is then be estimated. For example, if the maximum number of leading zeros observed is *x*, the number of distinct elements in the set is estimated as 2^x. In Khmer [26], the *k*-mers are hashed using MurmurHash3 that gives uniform hash value distribution. A Hyperloglog sketch consists of an array of size 2^p, where *p* is the number representing the first *p* bits of the hash value of *k*-mer. By varying the size of the array (*p*), a more accurate estimation can be made. The number of the distinct elements is estimated by considering the number of leading zeros of the binary equivalents of these hash numbers in the multiset. The variance factor is attained by dividing multiset into many subsets. Finally, for each subset, the value of a maximum number of leading zeros is calculated and then their normalized harmonic mean is taken to approximate the cardinality of the entire set.

2.4. KmerGenie

KmerGenie [30] generates histograms for multiple values of *k* in a single run. KmerGenie gives full *k*-mer abundance histograms and is the hashing based approach. The proportion of distinct *k*-mers is sampled based on some parameter ε. A state-less 64 bits hash function [42] is used for hashing *k*-mers. In the first step of the algorithm, a hash function is used that uniformly distributes all *k*-mers into a number of sets. Afterwards, only those *k*-mers are counted which hashes to zero. Finally, by scaling the number of *k*-mers with a given abundance by ε, abundance histogram is computed from the *k*-mer counts.

The tools discussed above are sketched in (Table **1**), together with a brief description of their estimated output and length of *k* they support. Detailed description of these methods, such as, their underlying paradigms, programming language/license, *etc.* is given in Supplementary Table **S1**.

3. EVALUATION METHODOLOGY AND DATA SETS

The benchmarking experiment aims at contrasting the competitive performance of the state of the art algorithms for estimating *k*-mer coverage frequencies and F_0. To that end, KmerStream, ntCard, Khmer and KmerGenie are selected. These are the best-known algorithms and have performed best in recent studies. We choose the latest working implementations of these tools. The accuracy and performance of these tools are estimated on publicly available four real datasets of varying size. More specifically, the human genome with large coverage, *i.e., H. sapiens* 1, *H. sapiens* 2 and human genome NA19238 are chosen for performance estimation. The detailed information regarding the datasets is shown in Table **2**. All of these datasets have multiple compressed FASTQ files. Usually, the output data from high-throughput sequencing technologies are divided into multiple files. Some tools can efficiently perform parallelization in their first phase by reading from individual input files using separate threads (the tools can utilize Data Parallelism in input level). KmerGenie, KmerStream and ntCard have been directly tested on these multiple compressed FASTQ files as they support input in multiple compressed files. Whereas for

Table 1. Streaming algorithms employed in the comparative experiment along with their estimated output.

Streaming Algorithms	Estimated Output	Supported Maximum Length of *k*
KmerGenie [30]	F_0 and full *k*-mer frequency histogram	Arbitrary large length
KmerStream [29]	F_0 and f_1	Arbitrary large length
Khmer 2.1.1 (unique-kmers.py) [26]	F_0	Arbitrary large length
Khmer 2.1.1 (abundance-dist single.py) [26]	Full *k*-mer frequency histogram	≤ 32
ntCard [28]	F_0 and full *k*-mer frequency histogram	Arbitrary large length

F_0: distinct number of *k*-mers in input read set; f_1: the number on singletons in input read set.

Table 2. Sequence datasets.

S. No.	Organism	Genome Length (mega-bases)	Average Read Length (bases)	Total No. of Bases (giga-bases)	Input FASTQ File Size (Gigabytes)	Number of Reads
1	*F. vesca*	214	353	4.5	10.9	12,803,137
2	*Homo sapiens*1	2,991	151	123.7	292.1	819,148,264
3	*Homo sapiens* 2	2,991	100	135.3	339.5	1,339,740,542
4	Human genome for the individual NA19238	5,712.43	250	228.5	507.6	913,959,800

Khmer 2.1.1 (unique-kmers.py) and Khmer 2.1.1 (abundance-dist single.py) these files have been first decompressed and concatenated into one file.

Not all the tools estimate full *k*-mer abundance histogram. The comparison based on metrics, *e.g.*, wall clock time, memory usage, *etc.* might not always be comparable given the different estimates by the different tools. For those techniques estimating full *k*-mer abundance histogram cannot necessarily be compared with others tools estimating only values of F_0 and f_1 or tools estimating only F_0, because they have different use cases. For example, KmerGenie uses the generated abundance histograms to find the best value of *k* for assembly; KmerStream estimates the genome size and the error rate of the sequencing experiment based on the *k*-mer frequency statistics. However, to obtain better insights into the strengths and weaknesses of known approaches for estimating *k*-mer coverage frequencies, this study focuses mainly comparing them on the accuracy basis. Additionally, we also use a common set of well-defined metrics for evaluation, *i.e.,* scalability for larger *k* and larger sized dataset, runtime, parallelism, memory and CPU utilization. We compare F_0, f_1 and the full histograms, estimated by the different algorithms with the exact output of DSK. Currently, only three tools, namely, ntCard, KmerGenie and Khmer (abundance-dist-single.py) generate the full *k*-mer abundance histogram. Note that, Khmer has two different scripts: i) unique-kmers.py estimates the total number of distinct *k*-mers (F_0) for large *k* lengths and ii) abundance-dist-single.py gives full *k*-mer abundance histogram and works for $k \leq 32$.

We ran all programs on the machine with Intel (R) Xeon (R) CPU E5-2698 v3 @ 2.30 GHz processor with 64 GB RAM, 1 and 4 TB HDDs and 16 CPUs. The machine is running 64-bit Ubuntu 14.04 OS. All programs have been run with the parameters based on guidance provided by the developers and has been tested in parallel mode. While running each tool, the parameters related to the resource usages have been set in a way to utilize the maximum capacity on the underlying machine. The command lines for all considered programs are given in the Supplementary Material. Khmer (abundance-dist-single.py) for full abundance histogram has been run with a maximum memory of 32 GB for *F. vesca*, 48 GB for *H. sapiens* 1 and 54 GB for *H. sapiens* 2 and human genome NA19238, according to the guidelines. All these programs are free for download and can be downloaded from the links given in (Supplementary Table **S2**). We have collected statistics for DSK, ntCard, KmerGenie, KmerStream and Khmer on 1TB HDD. DSK was run on 4TB HDD owing to its more disk usage for human genome NA19238.

4. RESULT ANALYSIS

4.1. Accuracy

The estimated values for F_0 and f_1, for k = 25, 50, 75, 100 and 125, of ntCard, KmerStream, KmerGenie and Khmer (unique-kmers.py) along with error rates on four datasets, namely, *F. vesca*, *H. sapiens* 1, *H. sapiens* 2 and human genome NA19238 are summarized in Tables **3**-**6**, respectively. *H. sapiens* 2 has an average read length of 100 bases hence the values of F_0 and f_1 are

Table 3. Estimated values of F_0 and f_1 by ntCard, KmerGenie 1.7040 and Khmer 2.1.1 for *F. vesca* for k = 25, 50, 75, 100 and 125.

k	-	DSK 2.2.0	ntCard	Error%	KmerStream 1.1	Error%	KmerGenie 1.7040	Error%	Khmer 2.1.1 (unique-kmers.py)	Error%
25	F_0	583,137,847	583,676,933	**0.09**	530,329,548	9.06	601,623,000	3.17	591,556,352	1.42
	f_1	323,527,880	323,786,587	**0.08**	271,301,619	16.14	340,844,130	5.35	-	-
50	F_0	914,031,454	914,604,363	**0.06**	912,466,570	0.17	950,320,256	3.97	920,025,399	0.65
	f_1	602,056,795	602,439,035	**0.06**	601,435,694	0.10	636,040,064	5.64	-	-
75	F_0	1,098,780,218	1,099,778,393	**0.09**	1,096,628,468	0.20	1,145,461,053	4.25	1,128,906,482	2.67
	f_1	776,776,680	777,519,706	**0.10**	773,700,917	0.40	821,351,861	5.74	-	-
100	F_0	1,191,576,112	1,192,817,786	**0.10**	1,190,435,159	0.10	1,244,998,932	4.48	1,239,588,773	3.87
	f_1	876,971,790	878,084,632	**0.13**	876,638,409	0.04	928,811,850	5.91	-	-
125	F_0	1,232,899,836	1,233,719,858	**0.07**	1,232,868,822	0.00	1,291,769,640	4.77	1,295,874,877	4.86
	f_1	933,435,198	934,166,463	**0.08**	935,890,616	0.26	991,233,936	6.19	-	-

Column 'Error%' shows errors in percent. Column 'DSK 2.2.0' shows the exact values of F_0 and f_1.

Table 4. Estimated values of F_0 and f_1 by ntCard, KmerGenie 1.7040 and Khmer 2.1.1 for *H. sapiens* 1 for k = 25, 50, 75, 100 and 125.

k	-	DSK 2.2.0	ntCard	Error%	KmerStream 1.1	Error%	KmerGenie 1.7040	Error%	Khmer 2.1.1 (unique-kmers.py)	Error%
25	F_0	11,217,637,486	11,216,386,861	**0.01**	6,751,865,989	39.81	16,165,719,040	30.61	15,836,062,038	557.13
	f_1	8,490,459,593	8,485,840,663	**0.05**	3,714,396,067	56.25	13,017,354,240	34.78	-	-
50	F_0	13,699,865,268	13,695,660,817	**0.03**	13,564,472,565	0.99	21,248,181,675	35.52	21,159,640,890	294.99
	f_1	10,675,454,671	10,673,259,381	**0.02**	10,575,153,639	0.94	17,847,311,013	40.18	-	-
75	F_0	12,875,754,286	12,857,905,621	**0.14**	12,817,488,354	0.45	20,643,246,978	37.63	20,592,981,331	206.38
	f_1	9,779,384,551	9,757,713,845	**0.22**	9,702,616,635	0.78	17,212,356,585	43.18	-	-
100	F_0	10,630,623,336	10,611,400,488	**0.18**	10,606,879,146	0.22	17,214,539,980	38.25	16,959,187,669	151.24
	f_1	7,575,718,688	7,554,340,962	**0.28**	7,577,620,078	0.03	13,906,537,380	45.52	-	-
125	F_0	7,080,173,077	7,071,172,312	**0.13**	7,066,649,232	0.19	11,641,131,786	39.18	11,712,142,863	88.90
	f_1	4,469,409,703	4,460,297,170	**0.20**	4,454,226,447	0.34	8,873,732,302	49.63	-	-

Column 'Error%' shows errors in percent. Column 'DSK 2.2.0' shows the exact values of F_0 and f_1.

reported for k up to 100 only (not for k = 125). k-mer counting results of DSK are used as empirical evidence in the evaluation, same as used in ntCard [28], to have fair comparisons. The 'DSK 2.2.0' column of Tables **3-6** reports the exact values of F_0 and f_1 over all datasets, that constitutes ground truth in evaluating accuracy. In Tables **3-6**, more accurate estimates of F_0 and f_1 are highlighted in bold. The results for F_0 and f_1 may be explained as follows:

It is clear that the ntCard estimation of F_0 and f_1 are highly accurate, with the constant error rates over all four datasets compared to the other state-of-the-art competitors. The error rates of ntCard are less than 0.4% for all four datasets. KmerStream has error rates below 2% in the case of all four datasets for all considered values of k except for k = 25. The numbers in Tables **3-6** reveal an interesting finding: Despite overall lower error rates, KmerStream has higher error rates (higher for f_1 than for F_0) for small value of k (k = 25). Khmer 2.1.1 (unique-kmers.py) has higher error rates for *H. sapiens* 1 which is up to 557%, whereas for *F. vesca*, *H. sapiens* 2 and human genome NA19238 error rates are below 5%, 3% and 1% respectively. KmerGenie also has higher error rates for *H. sapiens* 1 which is up to 50%, whereas for *F. vesca*, *H. sapiens* 2 and human genome NA19238 error rates are low which are below 6.5%, 5% and 1% respectively. For *H. sapiens* 1, Khmer (unique-kmers.py) and KmerGenie, have higher error rates (> 30%) than ntCard and KmerStream. Khmer 2.1.1 (unique-kmers.py) has

Table 5. Estimated values of F_0 and f_1 by ntCard, KmerGenie 1.7040 and Khmer 2.1.1 for *H. sapiens* 2 for k = 25, 50, 75, 100 and 125.

k	-	DSK 2.2.0	ntCard	Error%	KmerStream 1.1	Error%	KmerGenie 1.7040	Error%	Khmer 2.1.1 (unique-kmers.py)	Error%
25	F_0	6,317,577,945	6,321,370,851	**0.06**	4,440,606,371	29.71	6,493,972,870	2.79	6,447,772,640	2.02
	f_1	3,726,921,849	3,728,402,513	**0.04**	2,124,581,580	42.99	3,900,164,180	4.65	-	-
50	F_0	7,576,436,303	7,575,568,030	**0.01**	7,507,180,258	0.91	7,768,465,452	2.53	7,808,249,016	2.97
	f_1	4,610,596,550	4,612,437,322	**0.04**	4,554,225,163	1.22	4,803,170,055	4.18	-	-
75	F_0	6,645,775,719	6,634,466,410	**0.17**	6,620,493,247	0.38	6,770,079,405	1.87	6,820,243,427	2.56
	f_1	3,601,391,827	3,590,235,204	**0.31**	3,574,106,736	0.76	3,726,960,545	3.49	-	-
100	F_0	2,055,560,283	2,054,217,987	**0.07**	2,055,238,955	0.02	2,067,553,684	0.58	2,055,943,971	0.02
	f_1	1,668,703,535	1,667,591,992	**0.07**	1,668,568,360	0.01	1,679,933,472	0.67	-	-

Column 'Error%' shows errors in percent. Column 'DSK 2.2.0' shows the exact values of F_0 and f_1.

Table 6. Estimated values of F_0 and f_1 by ntCard, KmerGenie 1.7040 and Khmer 2.1.1 for human genome NA19238 for k = 25, 50, 75, 100 and 125.

k	-	DSK 2.2.0	ntCard	Error%	KmerStream 1.1	Error%	KmerGenie 1.7040	Error%	Khmer 2.1.1 (unique-kmers.py)	Error%
25	F_0	15,695,189,022	15,698,708,120	**0.02**	7,874,114,589	49.83	15,774,752,125	0.51	15,821,852,733	0.80
	f_1	12,590,059,674	12,589,842,091	**0.00**	4,097,744,520	67.45	12,660,147,875	0.56	-	-
50	F_0	21,386,123,607	21,388,812,027	**0.01**	21,129,592,460	1.20	21,527,611,504	0.66	21,394,927,325	0.04
	f_1	18,003,764,501	18,008,739,129	**0.03**	17,708,817,546	1.64	18,125,064,461	0.67	-	-
75	F_0	22,940,994,545	22,903,722,006	**0.16**	22,768,767,319	0.75	23,081,411,904	0.61	23,158,655,606	0.94
	f_1	19,455,121,869	19,413,069,020	**0.22**	19,200,406,066	1.31	19,580,787,984	0.65	-	-
100	F_0	22,825,882,964	22,795,280,303	**0.13**	22,837,840,964	0.05	22,981,764,792	0.68	22,657,973,838	0.74
	f_1	19,311,399,602	19,271,642,379	**0.21**	19,350,438,432	0.20	19,462,845,864	0.78	-	
125	F_0	21,623,019,167	21,584,850,645	**0.18**	21,572,310,929	0.23	21,771,418,913	0.69	21,674,560,549	0.24
	f_1	18,103,091,932	18,054,837,842	**0.27**	17,990,743,190	0.62	18,227,641,426	0.69	-	-

Column 'Error%' shows errors in percent. Column 'DSK 2.2.0' shows the exact values of F_0 and f_1.

highest error rate for f_1 amongst other tools for most of the datasets however, for human genome NA19238 error rates are less than 1% for all considered values of k.

The full k-mer abundance histograms for k = 25, 50, 75, 100 and 125 from DSK, ntCard, KmerGenie and Khmer (abundance-dist-single.py) on *F. vesca*, *H. sapiens* 1, *H. sapiens* 2 (k upto 100) and human genome NA19238 along with the error rates (%) are presented in the Supplementary Tables **S3-S6**. k-mer frequency histograms up to frequency 50 are reported in Supplementary Tables **S3-S6** owing to the space limitation. Currently, for larger lengths of k, only ntCard and KmerGenie give full k-mer abundance histogram. Khmer (abundance-dist-single.py) gives full k-mer abundance histogram for $k \leq 32$ and hence the entries for k = 50, 75, 100 and 125 of Khmer (abundance-dist-single.py) are missing in the Supplementary Tables **S3-S6**. Estimation of full k-mer abundance histogram of ntCard is more accurate than the estimation of KmerGenie and Khmer (abundance-dist-single.py).

In summary, for estimation of F_0, f_1 and full k-mer abundance histogram, ntCard is more accurate compared with other methods irrespective of the length of k.

4.2. Runtime, Memory and CPU Utilization

The runtime, memory and CPU utilization of all considered tools, namely, ntCard, Khmer (abundance-dist-single.py), Khmer (unique-kmers.py), KmerStream and KmerGenie are presented in Appendix (Table **A1** for *F. vesca*), (Table **A2** for *H. sapiens* 1), (Table **A3** for *H. sapiens* 2 for k up to 100) and (Table **A4** for human genome NA19238) for k = 25, 50, 75, 100 and 125. The runtime, memory and CPU utilization are reported for: (i) ntCard, Khmer (abundance-dist-single.py) and KmerGenie for obtaining the full k-mer frequency histograms, (ii) for KmerStream 1.1 are reported for obtaining F_0 and f_1 and (iii) for Khmer (unique-kmers.py) are reported for obtaining F_0. For each dataset, the best results of different methods are highlighted in bold whereas average results are underlined. The runtime and memory usages for smaller (*F. vesca*) to larger (human genome NA19238) datasets for k = 25, 50, 75, 100 and 125 are shown in graphical form in Fig. (**1**). We will not discuss the performance of DSK, as DSK is an exact k-mer counter.

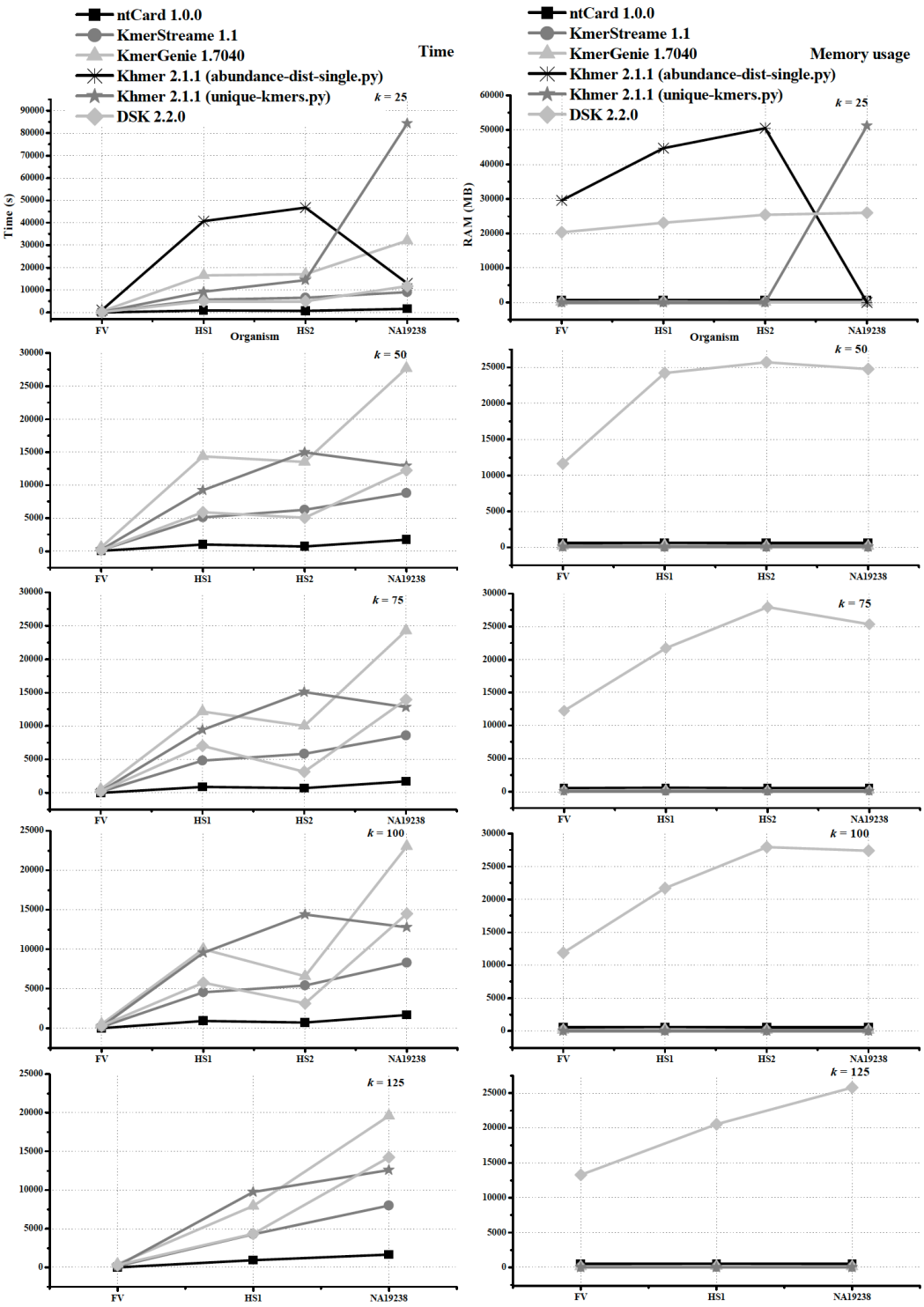

Fig. (1). Computation time *versus* datasets of varying size on left-hand side and memory usage *versus* datasets of varying size on right-hand side. Three of these datasets are human dataset with large coverage. Runtime is reported in seconds and memory usage in megabytes (MB). Note that *H. sapiens* 2 has average read lengths of 100 bases hence in plot for *k* =125 the data is missing for *H. sapiens* 2. Abbreviations: FV = *F. vesca;* HS1 = *H. sapiens* 1; HS2 = *H. sapiens* 2; and NA19238 = human genome NA19238.

Table **7** summarises the best and average performing programs with respect to runtime, memory and CPU utilization using results from Appendix (Table **A1-A4**). The results from Appendix (Table **A1-A4**)**,** Table **7** and Figs. (**1-2**) may be explained as follows:

ntCard is the fastest among all the tools, for all *k* and in all four datasets, ntCard took a maximum of 28 minutes 33 seconds, 11 minutes 53 seconds, 16 minutes 21 seconds and 22 seconds for human genome NA19238, *H. sapiens* 2, *H. sapiens* 1 and *F. vesca* respectively. Whereas Khmer

Table 7. Summary of results from Appendix (Tables A1-A4).

Dataset	*k*	Time		Memory (RAM)		CPU Utilization (%)	
		Lowest	Highest	Lowest	Highest	Lowest	Highest
F. vesca	25	ntCard 1.0.0	Khmer 2.1.1 (abundance-dist-single.py)	Khmer 2.1.1 (unique-kmers.py)	Khmer 2.1.1 (abundance-dist-single.py)	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	50	ntCard 1.0.0	KmerGenie 1.7048	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	75	ntCard 1.0.0	KmerGenie 1.7048	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	100	ntCard 1.0.0	KmerGenie 1.7048	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	125	ntCard 1.0.0	KmerGenie 1.7048	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
***H. sapiens* 1**	25	ntCard 1.0.0	Khmer 2.1.1 (abundance-dist-single.py)	Khmer 2.1.1 (unique-kmers.py)	Khmer 2.1.1 (abundance-dist-single.py)	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	50	ntCard 1.0.0	KmerGenie 1.7048	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	75	ntCard 1.0.0	KmerGenie 1.7048	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	100	ntCard 1.0.0	KmerGenie 1.7048	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	125	ntCard 1.0.0	Khmer 2.1.1 (unique-kmers.py)	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	ntCard 1.0.0	Khmer 2.1.1 (unique-kmers.py)
***H. sapiens* 2**	25	ntCard 1.0.0	Khmer 2.1.1 (abundance-dist-single.py)	Khmer 2.1.1 (unique-kmers.py)	Khmer 2.1.1 (abundance-dist-single.py)	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	50	ntCard 1.0.0	Khmer 2.1.1 (unique-kmers.py)	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	75	ntCard 1.0.0	Khmer 2.1.1 (unique-kmers.py)	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	100	ntCard 1.0.0	Khmer 2.1.1 (unique-kmers.py)	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
human genome NA19238	25	ntCard 1.0.0	Khmer 2.1.1 (abundance-dist-single.py)	Khmer 2.1.1 (unique-kmers.py)	Khmer 2.1.1 (abundance-dist-single.py)	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	50	ntCard 1.0.0	KmerGenie 1.7048	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	75	ntCard 1.0.0	KmerGenie 1.7048	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	100	ntCard 1.0.0	KmerGenie 1.7048	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)
	125	ntCard 1.0.0	KmerGenie 1.7048	Khmer 2.1.1 (unique-kmers.py)	ntCard 1.0.0	KmerStream 1.1	Khmer 2.1.1 (unique-kmers.py)

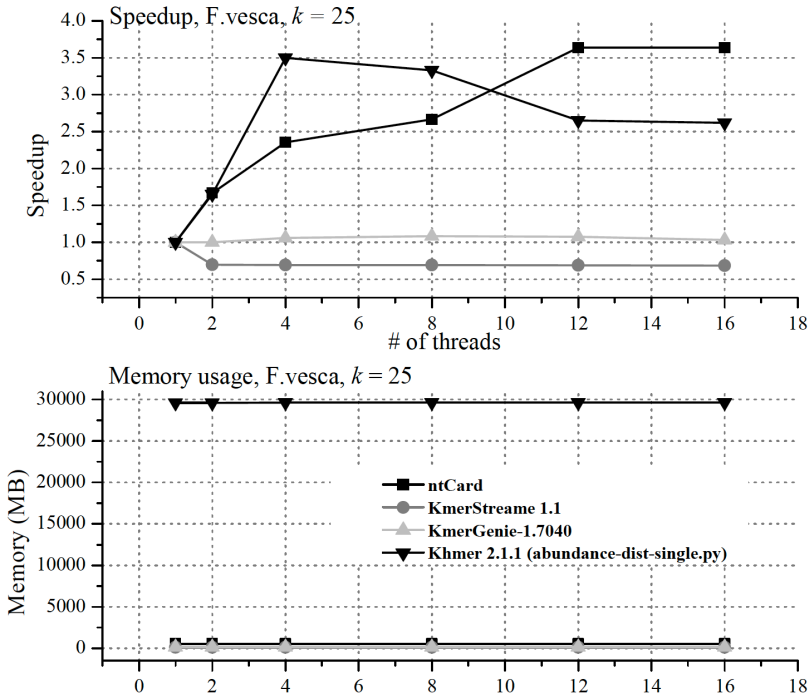

Fig. (2). Speedup and memory usage for various numbers of threads.

(abundance-dist-single.py) has the most extended runtime in estimating the full *k*-mer multiplicity histogram for $k = 25$ when compared to ntCard and KmerGenie. For most of the considered tools, the runtime grows with the input size. To complete estimation of full *k*-mer abundance histogram Khmer (abundance-dist-single.py) took around 19 minutes for *F. vesca*, around 11 hours 20 minutes for *H. sapiens* 1, around 13 hours for *H. sapiens* 2 and around 23 hours 25 minutes for human genome NA19238. KmerGenie for estimating the full *k*-mer multiplicity histogram has longer runtime than ntCard in the case of all four datasets for all values of *k*. KmerGenie took less than 11 minutes for *F. vesca*, at most 4 Hours 37 minutes on *H. sapiens* 1, maximum 4 Hours 45 minutes on *H. sapiens* 2 and for human genome NA19238 took at most 8 Hours 54 minutes. It can be seen that in the case of all four datasets with growing *k* there is the drop in the runtime of KmerGenie.

Memory requirement of ntCard that compute full *k*-mer multiplicity histogram is atmost 558 MB over all four datasets. Khmer (abundance-dist-single.py) has the highest memory requirement whereas ntCard has the second highest memory requirement. The memory requirement for ntCard is mainly for the count table to compute the whole *k*-mer abundance histogram. ntCard package has a hyperloglog version called 'nthll' to compute a distinct number of *k*-mers, F_0, with the memory requirement of at most 27 MB among all four datasets (Supplementary Table **S7**). Khmer (unique-kmers.py) requires lower (but not than ntCard) amount of memory but only estimates F_0 where it requires at most 30MB of memory over all four datasets.

Out of the tools which estimate the full *k*-mer multiplicity histogram, KmerGenie has the lowest memory requirement, whereas Khmer (unique-kmers.py) has the highest CPU utilization. ntCard often has the highest accuracy, good CPU utilization and the lowest time requirement.

4.3. Multithreaded Analysis

We also examine the performances of parallelism of various tools by recording the computation time for varying number of threads on *F. vesca* for $k = 25$. Fig. (**2**) shows the speedup and memory usages for a parallel execution of all considered tools with 1, 2, 4, 8, 12 and 16 threads. The *k*-mer frequency estimation algorithms have relatively uniform memory usages regardless of the number of threads because the sampling and estimation process are marginally affected by the memory usages. The results from Fig. (**2**) may be explained as follows:

The overall speedup for every tool except for Khmer 2.1.1 (abundance-dist-single.py) and ntCard is in the range of 0.5-1.25. KmerStream 1.1 has no gain in a speedup for varying number of threads; contrarily with the increasing number of threads its performance degrades (its parallel execution is slower than the serial). The Khmer 2.1.1 (abundance-dist-single.py) shows good speedup and speedup is up to ~3.5. The parallelization paradigm in ntCard is based on Data Parallelism and it uses OpenMP for parallelization. The maximum speedup for ntCard is up to 3.64. ntcard parallelizes on the number of files and can, therefore, show good speedup on the input having more number of files whereas kmerGenie parallelize on the number of *k* values.

CONCLUSION

In this paper, we report on a large-scale empirical comparison of streaming algorithms that estimates F_0 (the number of distinct *k*-mers), f_1 (the number of *k*-mers that occur exactly once) and/or full *k*-mer abundance histogram. We use varying size of high-throughput sequencing genomics datasets. The streaming approach is aimed at processing data in minimum space and uses a significantly lower amount of memory than needed to store the total distinct elements. Out of the tools considered in the study, ntCard is

more accurate in estimating F_0, f_1 and full k-mer abundance histograms, than other methods irrespective of the size of k. ntCard is several folds faster than the state-of-the-art approaches but has more memory requirement compared to KmerGenie. ntCard and KmerGenie generate full k-mer coverage frequency histogram for larger values of k. Khmer (abundance-dist-single.py) also generate full k-mer coverage frequency histogram but supports the smaller values of k ($k \leq 32$). Khmer (abundance-dist-single.py) often has the highest error rates and also uses comparatively large amount of memory for all runs. Out of these three programs, ntCard is the best concerning accuracy and runtime. KmerGenie has often the lowest memory requirements, but the longer runtime. Concerning the ranking of the streaming algorithms for estimation of k-mer frequencies, it is clear that there is no single winner. With the advancement of next-generation sequencing technologies generating billions of reads per experiment, there will always remain scope of improvement of algorithms and tools for improving accuracy, runtime and memory footprint.

CONSENT FOR PUBLICATION

Not applicable.

CONFLICT OF INTEREST

The authors declare no conflict of interest, financial or otherwise.

ACKNOWLEDGEMENTS

Declared none.

SUPPLEMENTARY MATERIAL

Supplementary material is available on the publisher's website along with the published article.

APPENDIX

Table A1. *F.vesca* results of considered tools for k = 25, 50, 75, 100 and 125.

S. No.	Tools (Version)	k = 25		k = 50		k = 75		k = 100		k = 125	
		Time (sec)	Memory (MB) / Disk (GB)	Time (sec)	Memory (MB) / Disk (GB)	Time (sec)	Memory (MB) / Disk (GB)	Time (sec)	Memory (MB) / Disk (GB)	Time (sec)	Memory (MB) / Disk (GB)
1	DSK 2.2.0	117	20,337 / 3.7	140	11,627 / 3	217	12,260 / 2	315	11,865 / 34	326	13,269 / 79
	CPU utilization (%) (Comments)	770.492 (Initial 45% of time consistent to 95, remaining 55% inconsistent in the range of 1600 – 10)		690.89 (Inconsistent)		607.77 (Initial 20% of time consistent to 1500, remaining 80% inconsistent in the range of 1600 – 100)		567.65 (Inconsistent in the range of 1600 – 100)		538.23 (Inconsistent in range of 1600–100)	
2	ntCard 1.0.0	**22**	534 / –	**22**	534 / –	**21**	534 / –	**22**	534 / –	**22**	534 / –
	CPU utilization (%) (Comments)	388.46 (In initial 50% of time declined from 1100 to 100, rest 50% of time consistent with 100)		399.87 (In initial 50% of time declined from 1000 to 100, rest 50% of time consistent with 100)		408.2 (In initial 60% of time declined from 900 to 130, rest 40% of time consistent with 100)		383.91 (In initial 50% of time declined from 900 to 140, rest 50% of time consistent with 100)		342.56 (In initial 50% of time declined from 900 to 100, rest 50% of time consistent with 100)	
3	KmerStreame 1.1	197	107 / –	187	106 / –	184	106 / –	182	106 / –	176	106 / –
	CPU utilization (%) (Comments)	99.41 (Inconsistent in the range of 94 – 100)		99.53 (Inconsistent in the range of 94 – 106)		99.77 (Inconsistent in the range of 94 – 106)		99.57 (Inconsistent in the range of 93 – 100)		99.55 (Inconsistent in the range of 94 – 106)	
4	KmerGenie 1.7048	630	144 / –	610	144 / –	514	144 / –	538	152 / –	447	159 / –
	CPU utilization (%) (Comments)	115.10 (Consistent in the range of 94 – 200)		115.55 (Consistent in the range of 94 – 200)		118.03 (Consistent in the range of 94 – 200)		115.11 (Consistent in the range of 94 – 200)		120.01 (Consistent in the range of 94 – 200)	
5	Khmer 2.1.1 (unique-kmers.py)	229	**25** / –	231	**26** / –	235	**25** / –	227	**25** / –	226	**26** / –
	CPU utilization (%) (Comments)	**3,124.7** (Consistent)		**3,132.6** (Consistent)		**3,089.68** (Consistent)		**3,132.06** (Consistent)		**3,132.91** (Consistent)	
6	Khmer 2.1.1 (abundance-dist-single.py)	1,112	29,631 / –	ERROR: Khmer only supports k-mer sizes ≤ 32.							
	CPU utilization (%) (Comments)	1,191.52 (Consistent)									

Best results are indicated in bold font and average results are underlined. Abbreviations: sec = Seconds, GB = Gigabytes, MB = Megabytes.

Table A2. *H.sapiens 1* results of considered tools for k = 25, 50, 75, 100 and 125.

S. No.	Tools (Version)	k = 25		k = 50		k = 75		k = 100		k = 125	
		Time (sec)	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**
1	DSK 2.2.0	5,046	23,106 / 102.8	5,881	24,240 / 67.7	7,033	21,731 / 5.6	5,767	21,678 / 41.9	4,348	20,511 / 32.7
	CPU utilization (%) (Comments)	470.34 (Initial 45% of time consistent to 800, next 55% inconsistent in the range of 1600 – 300)		499.35 (Initial 40% of time consistent to 800, next 60% in-consistent in the range of 1600–550)		419.15 (Initial 35% of time consistent to 800, next 65% in-consistent in the range of 1200–700)		427.96 (Initial 40% of time consistent to 780, next 65% inconsistent in the range of 1300–650)		348.97 (Initial 50% of time consistent in the range 550–10, next 50% in the range of 1600–550)	
2	ntCard 1.0.0	**981**	530 / –	**974**	528 / –	**920**	558 / –	**950**	527 / –	**941**	528 / –
	CPU utilization (%) (Comments)	144.31 (Inconsistent in the range of 70 – 200)		129.13 (Inconsistent in the range of 80 – 180)		121.40 (Inconsistent in the range of 80 – 170)		108.72 (Inconsistent in the range of 50 – 160)		97.94 (Inconsistent in the range of 50–140)	
3	KmerStreame 1.1	5,637	57 / –	5,107	56 / –	4,854	57 / –	4,539	56 / –	4,315	58 / –
	CPU utilization (%) (Comments)	98.85 (Consistent)		98.90 (Consistent in the range of 80 – 106)		98.98 (Consistent in the range of 80 – 106)		98.98 (Consistent in the range of 80 – 106)		98.90 (Consistent in the range of 80 – 106)	
4	KmerGenie 1.7048	16,619	223 / –	14,368	277 / –	12,154	196 / –	10,013	248 / –	7,956	173 / –
	CPU utilization (%) (Comments)	115.84 (Consistent in the range of 94 – 200)		118.30 (Consistent in the range of 94 – 200)		121.90 (Consistent in the range of 94 – 200)		126.66 (Consistent in the range of 94 – 200)		134.02 (Consistent in the range of 100 – 200)	
5	Khmer 2.1.1 (unique-kmers.py)	9,281	**26** / –	9,248	**25** / –	9,424	**25** / –	9,550	**25** / –	9,744	**26** / –
	CPU utilization (%) (Comments)	**3,120.4** (Consistent)		**3,119.17** (Consistent)		**3,115.77** (Consistent)		**3,117.37** (Consistent)		**3,113.13** (Consistent)	
6	Khmer 2.1.1 (abundance-dist-single.py)	40,794	44,711 / –	ERROR: Khmer only supports k-mer sizes ≤ 32.							
	CPU utilization (%) (Comments)	1,590.34 (Consistent)									

Best results are indicated in bold font and average results are underlined. Abbreviations: sec = Seconds, GB = Gigabytes, MB = Megabytes.

Table A3. *H.sapiens 2* results of considered tools for k = 25, 50, 75 and 100.

S. No.	Tools (Version)	k = 25		k = 50		k = 75		k = 100	
		Time (sec)	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**
1	DSK 2.2.0	5,065	25,400 / 113.7	5,049	25,739 / 69.7	4,778	25,322 / 49.8	3,168	27,930 / 31.7
	CPU utilization (%) (Comments)	418.559 (Inconsistent)		484.26 (Inconsistent)		391.295 (Consistent)		247.251 (Initial 60% of time consistent, last 40% inconsistent (1250 – 10))	

(Table A3) contd….

S. No.	Tools (Version)	*k* = 25		*k* = 50		*k* = 75		*k* = 100	
		Time (sec)	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**
2	ntCard 1.0.0	**710**	538 / –	**713**	558 / –	**710**	539 / –	**708**	538 / –
	CPU utilization (%) (Comments)	387.25 (Highly inconsistent in the range of 100 – 800)		341.52 (Highly inconsistent in the range of 90–700)		283.68 (Highly inconsistent in the range of 50–650)		202.91 (Highly inconsistent in the range of 70–400)	
3	KmerStreame 1.1	6,677	48 / –	6,282	49 / –	5,877	47 / –	5,422	48 / –
	CPU utilization (%) (Comments)	98.73 (Inconsistent in the range of 60 – 106)		98.73 (Inconsistent in the range of 80 – 106)		98.68 (Inconsistent in the range of 60 – 106)		98.62 (Inconsistent in the range of 60 – 106)	
4	KmerGenie 1.7048	17,109	145 / –	13,526	145 / –	9,997	145 / –	6,562	144 / –
	CPU utilization (%) (Comments)	119.03 (Consistent in the range of 94 – 200)		124.12 (Consistent in the range of 94 – 200)		132.73 (Consistent in the range of 100 – 200)		150.38 (Consistent in the range of 100 – 200)	
5	Khmer 2.1.1 (unique-kmers.py)	14,378	**26** / –	14,932	**25** / –	15,066	**29** / –	14,412	**26** / –
	CPU utilization (%) (Comments)	**3,110.69** (Consistent)		**3,110.97** (Consistent)		**3,112.2** (Consistent)		**3,114.46** (Consistent)	
6	Khmer 2.1.1 (abundance-dist-single.py)	46,861	50,470 / –	ERROR: Khmer only supports *k*-mer sizes ≤ 32.					
	CPU utilization (%) (Comments)	1,592.78 (Consistent)							

Best results are indicated in bold font and average results are underlined. Abbreviations: sec = Seconds, GB = Gigabytes, MB = Megabytes.

Table A4. Human genome NA19238 results of considered tools for *k* = 25, 50, 75, 100 and 125.

S. No.	Tools (Version)	*k* = 25		*k* = 50		*k* = 75		*k* = 100		*k* = 125	
		Time (sec)	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**	**Time (sec)**	**Memory (MB) / Disk (GB)**
1	DSK 2.2.0	11,829	25,991 / 220	12,254	24,795 / 155.8	13,933	25,325 / 136.9	14,516	27,400 / 122.9	14,215	25,800 / 277
	CPU utilization (%) (Comments)	437.71 (Initial 50% of time consistent to 600, rest 50% consistent to 300)		478.81 (Initial 50% of time consistent to 600, rest 50% inconsistent in the range of 1200 – 400)		489.37 (Initial 40% of time consistent to 800, rest 60% inconsistent in the range of 1200 – 400)		515.74 (Initial 40% of time consistent to 850, rest 60% inconsistent in the range of 1200 – 300)		479.66 (Initial 40% of time consistent to 850, rest 60% inconsistent in the range of 1200 – 300)	
2	ntCard 1.0.0	**1,711**	527 / –	**1,707**	528 / –	**1,713**	528 / –	**1,698**	530 / –	**1,680**	528 / –
	CPU utilization (%) (Comments)	146.36 (Initially consistent in the range of 110–200, for last 10% of time suddenly dropped to 100 then to 50)		139.33 (Initially consistent in the range of 110 – 200, for last 10% of time suddenly dropped to 100 then to 50)		133.68 (Initially consistent in the range of 110 – 180, for last 10% of time suddenly dropped to 100 then to 40)		127.19 (Initially consistent in the range of 90–170, for last 10% of time suddenly dropped to 100 then to 50)		123.27 (Initially consistent in the range of 90–170, for last 10% of time suddenly dropped to 100 then to 50)	

(Table A4) contd….

S. No.	Tools (Version)	*k* = 25		*k* = 50		*k* = 75		*k* = 100		*k* = 125	
		Time (sec)	Memory (MB) / Disk (GB)	Time (sec)	Memory (MB) / Disk (GB)	Time (sec)	Memory (MB) / Disk (GB)	Time (sec)	Memory (MB) / Disk (GB)	Time (sec)	Memory (MB) / Disk (GB)
3	KmerStreame 1.1	9,138	75 / –	8,822	76 / –	8,599	76 / –	8,278	76 / –	8,013	76 / –
	CPU utilization (%) (Comments)	98.72 (Consistent in the range of 90 – 106)		98.71 (Consistent in the range of 90 – 106)		98.67 (Consistent in the range of 90 – 106)		98.68 (Consistent in the range of 90 – 106)		98.62 (Consistent in the range of 90 – 106)	
4	KmerGenie 1.7048	32,071	144 / –	27,682	144 / –	24,276	144 / –	23,045	144 / –	19,614	144 / –
	CPU utilization (%) (Comments)	113.09 (Consistent in the range of 94 – 200)		115.26 (Consistent in the range of 94 – 200)		117.38 (Consistent in the range of 94 – 200)		118.57 (Consistent in the range of 94 – 200)		121.68 (Consistent in the range of 94 – 200)	
5	Khmer 2.1.1 (unique-kmers.py)	12,952	**26** / –	12,896	**30** / –	12,807	**25** / –	12,802	**27** / –	12,582	**25** / –
	CPU utilization (%) (Comments)	**3,114.51** (Consistent)		**3,107.84** (Consistent)		**3,110.24** (Consistent)		**3,111.61** (Consistent)		**3,111.19** (Consistent)	
6	Khmer 2.1.1 (abundance-dist-single.py)	84,283	51,221 /–	ERROR: Khmer only supports *k*-mer sizes ≤ 32.							
	CPU utilization (%) (Comments)	1,588.08 (Consistent)									

Best results are indicated in bold font and average results are underlined. Abbreviations: sec = Seconds, GB = Gigabytes, MB = Megabytes.

REFERENCES

[1] Marçais, G.; Kingsford, C. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, **2011**, *27*(6), 764-770.

[2] Miller, J.R.; Delcher, A.L.; Koren, S.; Venter, E.; Walenz, B.P.; Brownley, A.; Johnson, J.; Li, K.; Mobarry, C.; Sutton, G. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, **2003**, *24*(24), 2818-2824.

[3] Jaffe, D.B.; Butler, J.; Gnerre, S.; Mauceli, E.; Lindblad-Toh, K.; Mesirov, J.P.; Zody, M.C.; Lander, E.S. Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome. Res.*, **2003**, *13*(1), 91-96.

[4] Miller, J.R.; Koren, S.; Sutton, G. Assembly algorithms for next-generation sequencing data. *Genomics*, **2010**, *95*(6), 315-327.

[5] Pevzner, P.A.; Tang, H.; Waterman, M.S. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. USA,* **2001**, *98*(17), 9748-9753.

[6] Zerbino, D; Birney, E. Velvet: Algorithms for *de novo* short read assembly using de Bruijn graphs. *Genome. Res.*, **2008**, *18*(5), 821-829.

[7] Simpson, J.T.; Wong, K.; Jackman, S.D.; Schein, J.E.; Jones, S.J.; Birol, I. ABySS: A parallel assembler for short read sequence data. *Genome. Res.*, **2009**, *19*(6), 1117-1123.

[8] Kelley, D.R.; Schatz, M.C.; Salzberg, S.L. Quake: Quality-aware detection and correction of sequencing errors. *Genome. Biol.*, **2010**, *11*(11), R116.

[9] Shi, H.; Schmidt, B.; Liu, W.; Müller-Wittig, W. A parallel algorithm for error correction in high-throughput short-read data on CUDA-enabled graphics hardware. *J. Comput. Biol.*, **2010**, *17*(4), 603-615.

[10] Liu, Y.; Schröder, J.; Schmidt, B. Musket: A multistage k-mer spectrum-based error corrector for Illumina sequence data. *Bioinformatics*, **2012**, *29*(3), 308-315.

[11] Medvedev, P.; Scott, E.; Kakaradov, B.; Pevzner, P. Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*, **2011**, *27*(13), 137-141.

[12] Salmela, L.; Schröder, J. Correcting errors in short reads by multiple alignments. *Bioinformatics*, **2011**, *27*(11), 1455-1461.

[13] Li, R.; Ye, J.; Li, S.; Wang, J.; Han, Y.; Ye, C.; Wang, J.; Yang, H.; Yu, J.; Wong, G.K.S.; Wang, J. ReAS: Recovery of ancestral sequences for transposable elements from the unassembled reads of a whole genome shotgun. *PLoS Computat. Biol.*, **2005**, *1*(4), e43.

[14] Price, A.L.; Jones, N.C.; Pevzner, P.A. *De novo* identification of repeat families in large genomes. *Bioinformatics*, **2005**, *21*(Suppl 1), 351-358.

[15] Campagna, D.; Romualdi, C.; Vitulo, N.; Del Favero, M.; Lexa, M.; Cannata, N.; Valle, G. RAP: A new computer program for *de novo* identification of repeated sequences in whole genomes. *Bioinformatics*, **2004**, *21*(5), 582-588.

[16] Lefebvre, A.; Lecroq, T.; Dauchel, H.; Alexandre, J. FORRepeats: Detects repeats on entire chromosomes and between genomes. *Bioinformatics.*, **2003**, *19*(3), S319-S326.

[17] Healy, J.; Thomas, E.E.; Schwartz, J.T.; Wigler, M. Annotating large genomes with exact word matches. *Genome. Res.*, **2003**, *13*(10), 2306-2315.

[18] Kurtz, S.; Narechania, A.; Stein, J.C.; Ware, D. A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes. *BMC. Genom.,* **2008**, *9*(1), 517.

[19] Kokot, M.; Długosz, M.; Deorowicz, S. KMC 3: Counting and manipulating k-mer statistics. *Bioinformatics*, **2017**, *33*(17), 2759-2761.

[20] Erbert, M.; Rechner, S.; Müller-Hannemann, M. Gerbil: A fast and memory-efficient k-mer counter with GPU-support. *Algor. Mol. Biol.,* **2017**, *12*(1), 9.

[21] Rizk, G.; Lavenier, D; Chikhi, R. DSK: k-mer counting with very low memory usage. *Bioinformatics*, **2013**, *29*(5), 652-653.

[22] Conway, T.C.; Bromage, A.J. Succinct data structures for assembling large genomes. *Bioinformatics*, **2011**, *27*(4), 479-486.

[23] Stephens, Z.D.; Lee, S.Y.; Faghri, F.; Campbell, R.H.; Zhai, C.; Efron, M.J.; Iyer, R.; Schatz, M.C.; Sinha, S.; Robinson, G.E. Big data: Astronomical or genomical? *PLoS Biol.,* **2015**, *13*(7), e1002195.

[24] Brown, T.C.; Howe, A.; Zhang, Q.; Pyrkosz, A.B.; Brom, T.M. A reference-free algorithm for computational normalization of shotgun sequencing data. arXiv:1203.4802, **2012**.

[25] Pell, J.; Hintze, A.; Canino-Koning, R.; Howe, A.; Tiedje, J.M.; Brown, C.T. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proc. Natl. Acad. Sci. USA*, **2012**, *9*(33), 13272-13277.

[26] Junior, L.C.I.; Brown, C.T. Efficient cardinality estimation for k-mers in large DNA sequencing data sets. *F1000. Res.,* **2016**, 1-5.

[27] Zhang, Q.; Pell, J.; Canino-Koning, R.; Howe, A.C.; Brown, C.T. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *PloS. One*, **2014**, *9*(7), e101271.

[28] Mohamadi, H.; Khan, H.; Birol, I. ntCard: A streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, **2017**, *33*(9), 1324-1330.

[29] Melsted P.; Halldorsson B.V. KmerStream: Streaming algorithms for k-mer abundance estimation. *Bioinformatics*, **2014**, *30*(24), 3541-3547.

[30] Chikhi R.; Medvedev P. Sequence analysis informed and automated k-mer size selection for genome assembly. *Bioinformatics,* **2014**, *30*(1), 31-37.

[31] Alon N.; Matias Y.; Szegedy M. *The space complexity of approximating the frequency moments.* Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC), Philadelphia, Pennsylvania, U.S, **1996**, pp. 20-29.

[32] Bar-Yossef, Z.; Jayram, T.S.; Kumar, R.; Sivakumar, D.; Trevisan, L. Counting distinct elements in a data stream. In: *International Workshop on Randomization and Approximation Techniques in Computer Science,* Springer: Berlin, Heidelberg, Germany, **2002**, pp. 1-10.

[33] Flajolet, P.; Martin, G.N. Probabilistic counting algorithms for data base applications. *J. Computer Syst Sci.*, **1985**, *31*(2), 182-209.

[34] Cormode G.; Muthukrishnan S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms.*, **2005**, *55*(1), 58-75.

[35] Simpson, J.T. Exploring genome characteristics and sequence quality without a reference. *Bioinformatics*, **2014**, *30*(9), 1228-1235.

[36] Chu, J.; Sadeghi, S.; Raymond, A.; Jackman, S.D.; Nip, K.M.; Mar, R., Mohamadi, H.; Butterfield, Y.S.; Robertson, A.G.; Birol, I. BioBloom tools: Fast, accurate and memory-efficient host species sequence screening using bloom filters. *Bioinformatics*, **2014**, *30*(23), 3402-3404.

[37] Pérez, N.; Gutierrez, M.; Vera, N. Computational performance assessment of k-mer counting algorithms. *J. Comput. Biol.*, **2016**, *23*(4), 248-255.

[38] Mohamadi, H.; Chu, J.; Vandervalk, B.P.; Birol, I. ntHash: Recursive nucleotide hashing. *Bioinformatics*, **2016**, *32*(22), 3492-3494.

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

[40] Crusoe, M.R.; Alameldin, S.; Awad, E.; Boucher, A.; Caldwell, R.; Cartwright, A.; Charbonneau, B.; Constantinides, G.; Edvenson, S.; Fay, J.; Fenton, T.; Fenzl, J.; Fish, L.; Garcia-Gutierrez, P.; Garland, J.; Gluck, I.; González, S.; Guermond, J.; Guo, A.; Gupta, J.R.; Herr, A.; Howe, A.; Hyer, A.; Härpfer, L.; Irber, R.; Kidd, D.; Lin, J.; Lippi, T.; Mansour, P.; McA'Nulty, E.; McDonald, J.; Mizzi, K.D.; Murray, J.R.; Nahum, K.; Nanlohy, A.J.; Nederbragt, H.; Ortiz-Zuazaga, J.; Ory, J.; Pell, C.; Pepe-Ranney, Z.N.; Russ, E.; Schwarz, C.; Scott, J.; Seaman, S.; Sievert, J.; Simpson, C.T.; Skennerton, J.; Spencer, R.; Srinivasan, D.; Standage, J.A.; Stapleton, S.R.; Steinman, J.; Stein, B.; Taylor, W.; Trimble, H.L.; Wiencko, M.; Wright, B.; Wyss, Q.; Zhang, E.; Zyme; C.T. Brown. The khmer software package: Enabling efficient nucleotide sequence analysis. *F1000 Res.,* **2015**, *4*(115), 900.

[41] Flajolet, P.; Fusy, É.; Gandouet, O.; Meunier, F. Hyperloglog*:* The analysis of a near-optimal cardinality estimation algorithm In: *Discret. Math. Theor. Comput. Sci*, **2007**, pp. 137-156.

[42] Press, W.H.; Teukolsky, S.A.; Vetterling, W.T.; Flannery, B.P. *Numerical recipes: The Art of Scientific Computing,* 3rd ed.; Cambridge University Press: New York, NY, USA, **2007**.