Bioinformatics, 38(4), 2022, 933–940
https://doi.org/10.1093/bioinformatics/btab797
Advance Access Publication Date: 26 November 2021
Original Paper

Sequence analysis

KCOSS: an ultra-fast k-mer counter for assembled genome analysis

Deyou Tang [1,2,*], Yucheng Li[1], Daqiang Tan [1], Juan Fu[3], Yelei Tang[1], Jiabin Lin[1], Rong Zhao[1], Hongli Du[4] and Zhongming Zhao [2,5,6,*]

[1]School of Software Engineering, South China University of Technology, Guangzhou, Guangdong 510006, China, [2]Center for Precision Health, School of Biomedical Informatics, The University of Texas Health Science Center at Houston, Houston, TX 77030, USA, [3]School of Medicine, South China University of Technology, Guangzhou, Guangdong 510006, China, [4]School of Biology and Biological Engineering, South China University of Technology, Guangzhou, Guangdong 510006, China, [5]Human Genetics Center, School of Public Health, The University of Texas Health Science Center at Houston, Houston, TX 77030, USA and [6]MD Anderson Cancer Center UTHealth Graduate School of Biomedical Sciences, Houston, TX 77030, USA

*To whom correspondence should be addressed.
Associate Editor: Tobias Marschall

Received on February 13, 2021; revised on October 13, 2021; editorial decision on November 11, 2021; accepted on November 19, 2021

Abstract

Motivation: The k-mer frequency in whole genome sequences provides researchers with an insightful perspective on genomic complexity, comparative genomics, metagenomics and phylogeny. The current k-mer counting tools are typically slow, and they require large memory and hard disk for assembled genome analysis.

Results: We propose a novel and ultra-fast k-mer counting algorithm, KCOSS, to fulfill k-mer counting mainly for assembled genomes with segmented Bloom filter, lock-free queue, lock-free thread pool and cuckoo hash table. We optimize running time and memory consumption by recycling memory blocks, merging multiple consecutive first-occurrence k-mers into C-read, and writing a set of C-reads to disk asynchronously. KCOSS was comparatively tested with Jellyfish2, CHTKC and KMC3 on seven assembled genomes and three sequencing datasets in running time, memory consumption, and hard disk occupation. The experimental results show that KCOSS counts k-mer with less memory and disk while having a shorter running time on assembled genomes. KCOSS can be used to calculate the k-mer frequency not only for assembled genomes but also for sequencing data.

Availabilityand implementation: The KCOSS software is implemented in C++. It is freely available on GitHub: https://github.com/kcoss-2021/KCOSS.

Contact: dytang@scut.edu.cn or zhongming.zhao@uth.tmc.edu

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

K-mers are substrings of length k of a given string. K-mer frequency refers to the number of occurrences of k-mers in a specified dataset. Counting k-mer frequency is an essential step in many bioinformatics applications. These applications include genome sequence assembly (Jaffe *et al.*, 2003), multiple sequence alignment (Edgar, 2004), sequence compression (Deorowicz, 2020), duplicate detection (Manekar and Sathe, 2018), species identification (Nasko *et al.*, 2018), mutation detection (Audoux *et al.*, 2017) and sequence error correction (Kelley *et al.*, 2010) in sequencing data.

The k-mer frequency is also significant for assembled genomes analysis. In genomic features analysis, k-mers of length 7–9 bp is primary for motif analysis (Cserhati *et al.*, 2019), and specific k-mer patterns help identify DNA features (Sievers *et al.*, 2017). In genome-wide association studies, k-mers provide a versatile descriptor to capture genetic variants (Jaillard *et al.*, 2018) and overcome the limitations of single nucleotide polymorphism-based association (Lees *et al.*, 2016). In pan-genomics researches, the conservation classes of the k-mers highlighted their abundance or sparsity along the different studied lines and sub-genomes (Gordon *et al.*, 2020), and k-mer multiplicity is a way of measuring genome similarity (Bonnici *et al.*, 2018). In metagenomics research, common k-mers between the reads and the genomes help improve the accuracy of metagenomic classification (Břinda *et al.*, 2015), and the k-mer database is a fundamental utility for filtering continuous stretch (Tu *et al.*, 2014). K-mers have been studied in virus integration sites in the host genomes (Xu *et al.*, 2021). Furthermore, k-mer is necessary for phylogenetic distances estimation (Röhling *et al.*, 2020), genome evolution

(Bernard *et al.*, 2018) and genome-wide phylogeny and complexity evaluation (Wang *et al.*, 2015) in phylogenetic analysis.

Hence, many k-mer counting tools have been developed in the past decades. We comparatively evaluated several of the latest and competitive k-mer counters for both sequencing data and assembled genomes. However, when we input a whole genome instead of a sequencing dataset, only KMC3 (Kokot *et al.*, 2017), CHTKC (Wang *et al.*, 2020), GenomeTester4 (Kaplinski *et al.*, 2015) ($k \leq 32$ only) and Jellyfish (Marcais and Kingsford, 2011) could work well. Furthermore, the memory-based k-mer counter consumes memory linearly, resulting in running errors because of the limited memory. The disk-based k-mer counter consumes disk linearly, resulting in running slowly because of I/O cost. There is still a strong need for improvement regarding computational time and space efficiency for the k-mer counting algorithm.

After conducted multiple k-mer scans on the human genome, we found that more than 90% of k-mers are singletons for $k > 16$. Here, singleton refers to the k-mer with an occurrence frequency of 1. Singleton k-mers distribute consecutively in a genome, providing a possibility of representing multiple k-mers with a sequence to compress the storage of k-mers. Suppose that we have a DNA sequence of length 100 containing 82 consecutive singleton 19-mers, and we can save up to 94% memory consumption by storing the sequence directly instead of storing 82 19-mers. Less memory consumption helps the k-mer counter increase cache hits and reduce running time.

These observations inspire us to design a high-performance k-mer counter by reducing memory consumption for genomic k-mer counting. Based on these characteristics, we propose a novel k-mer frequency counting algorithm for genome-level data, k-mer Counter based on Overlapping Sequence Set (KCOSS), to count k-mer with less memory and disk while having a shorter runtime. Potential singleton k-mers are organized with a data structure called overlapping sequence sets. The overlapping sequence set makes multiple k-mers share a finite length memory instead of allocating a kbp memory for each k-mer, thus substantially decreasing the memory consumption. KCOSS uses Bloom filters for picking out potential singleton k-mers instead of rejecting k-mers with an occurrence of 1. It also utilizes multiple threads, lock-free thread pools, memory block recycling, lock-free cuckoo hash tables (Fan *et al.*, 2013; Li *et al.*, 2014) and other methods to accelerate the counting of k-mers. Our experimental results show that KCOSS is the most time-efficient and space-saving k-mer counting algorithm for assembled genome when k is $\geq$32.

2 Related works

Reducing the time and space overheads of a k-mer algorithm is a widely concerned problem in analyzing massive genome data. As early as 2008, the Tallymer algorithm (Kurtz *et al.*, 2008) was proposed to count k-mer and applied to the genomes of corn and other plant species. The Tallymer method stores k-mers in enhanced suffix arrays and its memory usage is related to the gene size being counted.

Hash table is the most commonly used data structure in k-mer algorithm design. The Jellyfish algorithm (Marcais and Kingsford, 2011) uses thread-safe and lock-free hash tables, as well as prefix arrays in multithreading to accelerate counting. It is one of the well-known k-mer counting algorithms with the best comprehensive performance. ntCard (Mohamadi *et al.*, 2017) uses the ntHash algorithm to efficiently compute hash values for streamed sequences. CHTKC (Wang *et al.*, 2020) solves the k-mer counting problem with a lock-free hash table that uses linked lists to resolve collisions. Experimental results show that the hash table is a feasible method to solve the k-mer counting problem.

Bloom filter is also widely used as a shortcut of optimization for many k-mer counting algorithms. The BFCounter algorithm (Melsted and Pritchard, 2011) is based on the Bloom filter to filter out single-occurring k-mers and reduce memory consumption subsequently. The algorithm can save up to 50% of memory usage. The DSK algorithm (Rizk *et al.*, 2013) implements hash mapping and partition caching to achieve memory, time and disk trade-offs. The Turtle algorithm (Roy *et al.*, 2014) first applies pattern-blocked Bloom filters to remove infrequent k-mers and then uses a sort-and-compress scheme for actual counting.

Disk-based k-mer counting algorithm deploys disk as the main medium for processing while consuming less memory. Usually, disk-based k-mer counters (Audano and Vannberg, 2014; Deorowicz *et al.*, 2013; Li and Yan, 2015) deploy two-phase counting. The first phase processes the partitions in memory and writes the results to disk files. The second phase merges disk files. The KMC3 algorithm uses a dual-disk architecture to reduce the slowness of the IO bottleneck, which is by far the most outstanding k-mer algorithm that mainly using external memory. This algorithm has been updated many times (Deorowicz *et al.*, 2015), and the latest version is KMC3 (Kokot *et al.*, 2017).

In addition, other techniques of counting are also extensively developed to address the k-mer counting problem, including probability distribution estimation, burst trie, counting quotient filter and GPU parallel programming. The khmer algorithm (Crusoe *et al.*, 2015) transforms the counting problem of k-mer into a probability distribution estimation problem. The KCMBT (Mamun *et al.*, 2016) is an algorithm based on in-memory burst trie. As an alternative to the hash table, tries are naturally ordered and there will be no hash collisions. The Squeakr (Pandey *et al.*, 2018) applies a counting quotient filter to serve as a natural and efficient structure for representing and operating on multisets of k-mers. The Gerbil algorithm (Erbert *et al.*, 2017) is implemented in a GPU method to accelerate the counting of k-mers, which has good performance in k-mers when a k value is greater than 32.

3 Materials and methods

3.1 The k-mer counting workflow

KCOSS leverages the continuity of singleton k-mers, as well as lock-free Bloom filter, lock-free thread pool and cuckoo hash table to accelerate calculation (Fig. 1, Supplementary Fig. S1). KCOSS uses a single producer-multiple consumers model combined with a lock-free queue to build a lock-free thread pool. The producer extracts the reads from the FASTA-formatted sequence files, divides the reads into blocks and puts the blocks into a lock-free queue. The consumers fetch blocks from the lock-free queue and convert the A, C, G and T encoded sequences to 00, 01, 10 and 11 encoded binary streams, respectively. When $0 < k \leq 14$, KCOSS uses a shared hash table to count k-mers due to the small scale of k-mers. When $k > 14$, KCOSS utilizes a lock-free segmented Bloom filter to determine whether the k-mer is the first occurrence. KCOSS inserts a subsequence containing multiple consecutive first-occurrence k-mers into an overlapping sequence set. Then it writes a memory block filled with overlapping sequence sets to the disk asynchronously. If a k-mer is not the first occurrence, KCOSS searches the fixed-size hash table to find a hash entry to insert or update. If a hash entry matches with the k-mer, KCOSS updates the corresponding frequency with the atomic operation provided by C++. If KCOSS does not locate a hash entry in the fixed-size hash table, it further checks the elastic cuckoo hash table to find a hash entry to insert or update. Finally, KCOSS merges the results saved in overlapping sequence sets and hash tables to generate the ultimate counting results.

3.2 Filtering and encoding of k-mers

We first encoded A, C, G and T as 00, 01, 10 and 11, respectively, to decrease memory consumption. KCOSS uses a data structure named overlapping sequence set to manage continuous singleton k-mers so that such k-mers do not need to be stored independently anymore. The elements of the overlapping sequence set are pairs of overlapping sequences and their length. The overlapping sequence, we call it C-read, is a variable-length subsequence extracted from the scanned read. KCOSS treats each k-mer encountered for the first time as a singleton k-mer (Fig. 2). The length of the overlapping sequence depends on the number of consecutive singletons extracted from the scanned subsequences. In the extreme case, for a sequence with length l, we only need l bp to save all k-mers if all k-mers extracted from the sequence are singletons. The overlapping sequence set is asynchronously written to disk when a memory block is full of C-reads.

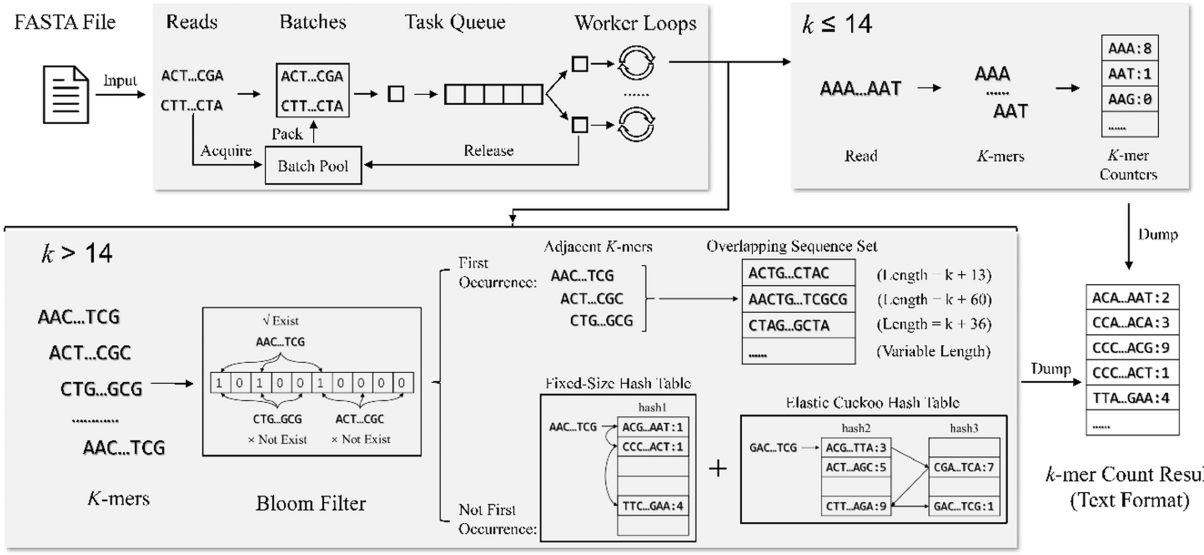

Fig. 1. The workflow of KCOSS algorithm

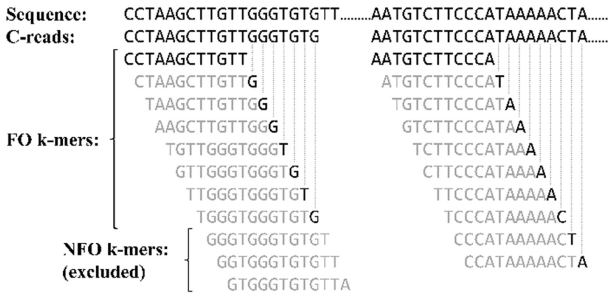

Fig. 2. The schematic diagram of overlapping sequence. Suppose that k-mer 'TGTTGGGTGTGT' has been detected before, then C-read 'CCTAAGCTTGTTGGGTGTG' is an overlapping sequence composed of k-mers ($k = 12$) 'CCTAAGCTTGTT', ..., 'TTGTTGGGTGTG'. FO and NFO are the abbreviations of first-occurrence and non-first-occurrence

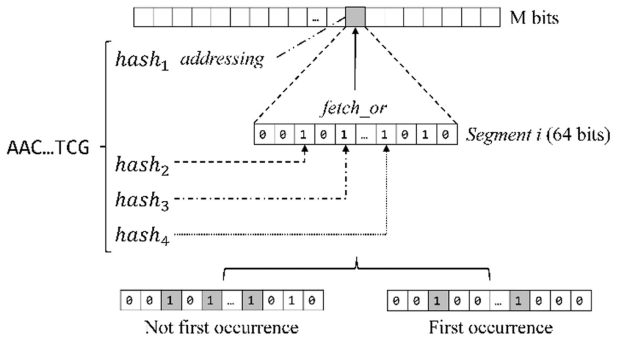

Fig. 3. The segmented Bloom filter

As mentioned before, we treat the first-occurrence k-mer as a singleton k-mer and create a new C-read or append the last base of the k-mer to the current C-read. This process includes four cases. The first case is that the previous k-mer and the current k-mer are singletons. We only need to append the last base of the new k-mer to the current C-read in this case. The second case is that the k-mer is a singleton and the previous k-mer is not a singleton, and we need to initialize a new C-read for the singleton. The third case is that a k-mer is not a singleton and the preceding k-mer is a singleton, and we insert the current C-read and its length into the overlapping sequence set. The last case is that both the current k-mer and the preceding k-mer appear before, and we search the fix-sized hash table or elastic hash table to find a hash entry to insert or update.

Bloom filter is a probabilistic data structure. Comparing with typical data structures such as bitmaps and trees, the Bloom filter has the characteristics of high space efficiency and has nothing to do with the length of the sequence to be inserted. We implemented a segmented Bloom filter to accelerate the detection of the first-occurrence k-mers (Fig. 3). KCOSS divides the hash space of length M bits into segments of length S bits. First, KCOSS uses hash function *hash1* to locate which segment that the current k-mer should check. Then KCOSS computes three hash functions *hash2*∼*hash4* and checks the bits identified by *hash2*∼*hash4*. If all three bits are 1, it means that the k-mer may appear before. Otherwise, it means the k-mer is the first occurrence, and we set the three bits to 1 and pass the k-mer to the overlapping sequence set. We invoke the *fetch_or* atomic operation to perform a lock-free insertion of the Bloom filter.

Suppose the Bloom filter has M *bits*, the number of distinct k-mers is N, and the number of hash functions is H. We use the following formula to compute the false-positive rate of the conventional Bloom filter.

$$f_p = \left(1 - \left(1 - \frac{1}{M}\right)^{HN}\right)^H \approx \left(1 - e^{-\frac{NH}{M}}\right)^H. \quad (1)$$

In the segmented Bloom filter, suppose the number of segments is n, the segment size is S and the number of hash functions is $H+1$, the probability that a bit in a segment is not set to 1 is $\left(1 - \frac{1}{s}\right)^H$. Assume mapping a k-mer to a segment using *hash1* is random, each segment needs to check $p = \frac{N}{n}$ k-mers. After traversing all k-mers and setting the corresponding bits to 1, the probability that a bit in a segment is set to 1 is $1 - \left(1 - \frac{1}{s}\right)^{Hp}$. Then, the false-positive rate of the segmented Bloom filter is as:

$$f_{seg} = \left(1 - \left(1 - \frac{1}{s}\right)^{Hp}\right)^H \approx \left(1 - e^{-\frac{Np}{S}}\right)^H. \quad (2)$$

If we replace n in $p = \frac{N}{n}$ with $n = \frac{M}{S}$, we get $p = \frac{NS}{M}$. And f_{seg} will be:

$$f_{seg} \approx \left(1 - e^{-H\left(\frac{N}{M}\right)}\right)^H. \quad (3)$$

Compared f_p with f_{seg}, we found that the false-positive rate of the segmented Bloom filter for k-mer is the same as that of the

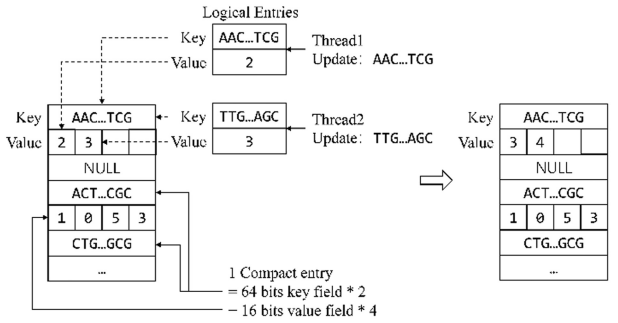

Fig. 4. The compact hash table

unsegmented Bloom filter, which is independent of the number of segments and the number of bits in segments.

3.3 Counting the non-first-occurrence k-mers

We used two hash tables to save the non-first-occurrence k-mers. The first is a large but fixed-length hash table, and the second is a small but elastic hash table. KCOSS calls atomic operations CAS and *fetch_add* to implement lock-free updates of a hash table. KCOSS tries to find the target hash entry in the fixed-length hash table to complete the insertion or update, and it searches the elastic hash table only after many failed attempts of the fixed-length hash table. The fixed-length hash table handles the majority of insert operations and resolves collision in the open addressing method. We used the quadratic probing method to make the elements in the array evenly distributed. We empirically set the maximum probing times to 16 as a compromise of time and space performance. The elastic hash table handling all the unsuccessful insertions in the fixed-length hash table is a scalable parallel cuckoo hash table implemented by *libcuckoo*.

KCOSS provides conventional key-value table entry and compact hash table entry for the hash tables. The condensed entry structure stores multiple k-mer frequencies in a 32-bit integer or a 64-bit integer to meet the word alignment requirement without any byte padding. KCOSS also utilizes the complementary property of DNA sequence and stores a k-mer and its forward complementary sequence in an entry to improve space utilization. For example, if k is not greater than 32, a compact entry contains two 64-bit keys and four 16-bit values to store four k-mers and their frequency (Fig. 4). The condensed scheme only consumes 24 bytes instead of 48 bytes to meet the word alignment of memory access. We defined a runtime parameter for the user to decide whether to use the compact entry structure.

3.4 Optimizations of memory access

KCOSS deploys the single producer-multiple consumers model, block recycling mechanism and lock-free queue to distribute and process reads in sequencing files. Two memory address queues respectively save the addresses of the unused memory blocks and the used memory blocks. Initially, the producer allocates a specified quantity of memory blocks and saves theirs addresses to the free memory block queue. In reading the sequence file, the producer fetches a memory block from the free memory block queue and packs the reads into a C-read block, which the size is just right for a memory block. If no memory block is available in the free memory block queue, the producer creates a new memory block and links the block to the used block queue. To avoid memory overflow, we defined a maximum number of memory blocks used to read the sequence. A consumer takes a block from the used block queue by polling and puts the memory block address into the free memory block queue after counting the k-mers. KCOSS accesses the free memory block queue and the used memory block queue with the CAS operation to ensure thread-safe and concurrent access. In this method, KCOSS reduces the memory access overhead of reading sequences.

KCOSS represents each nucleotide by two bits and applies word alignment in allocating memory for k-mers to reduce memory consumption and improve access efficiency. It stores 17-mer to 32-mer individuals with 8-byte unsigned integers and stores 33-mer to 64-mer individuals with 16-byte unsigned integers. It uses C-read to store consecutive singletons to enhance the spatial locality, increase the cache hit, and improve the memory access performance. Furthermore, a C-read block is written to disk asynchronously when a memory block about the same length as a disk block is full of C-reads. Disk mapping technique *mmap* is applied to avoid redundant memory copy operations and reduce the risk of IO bottleneck.

4. Results and discussion

We verified and evaluated the KCOSS algorithm using seven reference genomes: *Homo sapiens* (GCF_000001405.38), *Triticum urartu* (GCA_003073215.1), *Tupaia chinensis* (GCF_000334495.1), *Capsicum annuum* (GCF_000710875.1), *Periophthalmus magnuspinnatus* (GCA_000787105.1), *Erythrura gouldiae* (GCA_003676055.1) and *Sphenodon punctatus* (GCA_003113815.1) (Table 1). Note that in the reference genome of *Triticum urartu*, its ratio of the total number of distinct k-mers to the length of the genome and the ratio of the total number of singletons to the length of the genome are extremely low when $k = 18$. This is a challenge for KCOSS and we used it to test the robustness of KCOSS. We downloaded and compiled Jellyfish-2.3.0, KMC-3.1.1, KCMBT-1.0.0, DSK-2.3.3, Tallymer-1.2.2, Turtle-0.3.1, BFCounter-1.0.0, CHTKC, GenomeTester4 and khmer-2.1.2 from GitHub. We ran all algorithms separately on a 64 core Xeon Phi 7210 KNL server with 112 GB DDR4 RAM, and the operating system is Red Hat 4.8.5-39.

After comparative tests, we ultimately compared KCOSS with Jellyfish2, KMC3 and CHTKC. For KMC3, we used the RAM-only mode and denoted it as KMC3-R. For CHTKC, we used the optimized version and denoted it as CHTKC-O. We executed all the tests three times and took the average values as the results. The ram and disk usage are measured using the runtime peak value of the program.

Table 1. Genome datasets used in the experiments

Dataset	Genome length (bp)	GC content	$k = 18$		$k = 32$		$k = 55$		$k = 64$	
			Γ	β	γ	β	γ	β	γ	β
H.sapiens	3 099 706 404	41.48%	0.656	0.562	0.833	0.788	0.897	0.858	0.908	0.870
T.urartu	4 851 895 022	45.94%	0.236	0.166	0.398	0.308	0.586	0.482	0.639	0.533
T.chinensis	2 846 580 235	42.00%	0.654	0.567	0.847	0.806	0.898	0.865	0.904	0.873
C.annuum	2 935 884 163	35.41%	0.502	0.394	0.809	0.755	0.915	0.891	0.926	0.907
P.magnuspinnatus	701 696 780	40.00%	0.774	0.712	0.908	0.881	0.942	0.924	0.946	0.929
E.gouldiae	1 070 785 140	41.00%	0.873	0.810	0.942	0.921	0.951	0.933	0.953	0.936
S.punctatus	4 272 214 985	42.64%	0.544	0.457	0.751	0.706	0.829	0.800	0.840	0.814

Note: γ is the ratio of the total number of distinct k-mers to the genome length. β is the ratio of the total number of singletons to the genome length.

The comparison results using the seven genome datasets are shown in Tables 2–8, including runtime memory (in GB), runtime disk space (in GB) and running time (in seconds). When $k = 18$, the overall performance of KMC3, CHTKC and KCOSS is relatively close except for Jellyfish2, which has a longer running time on all test datasets. With the increase of k, the performance of KCOSS becomes more and more prominent. The running time of KCOSS is 9.4–21.4% of that of Jellyfish2, 17.3–45.6% of that of CHTKC and 19.3–58.5% of that of KMC3 when $k = 64$. The experimental results showed that KCOSS is the fastest k-mer counting algorithm for the assembled genomes. From the results, we found that even for challenging datasets like the *T.urartu* genome, there was no significant degradation in KCOSS performance. Overall, KCOSS has the characteristics of lower memory usage, less disk usage and higher time efficiency than Jellyfish2, KMC3 and CHTKC. The scalability of KCOSS in processing different lengths of k-mers is given in Supplementary Figure S2.

We further applied these algorithms to three sequencing datasets: *D.melanogaster*, *F.vesca f.semperflorens* and *D.rerio* (Supplementary Table S3). The experimental results showed that KCOSS also had excellent performance compared with Jellyfish2, KMC3 and CHTKC (Supplementary Tables S4–S6). The overall performance of KCOSS is slightly inferior to that of KMC3 and CHTKC but better than that of Jellyfish2. The current version of KCOSS is not optimized for sequence data. For datasets with deeply sequenced and huge volumes, we split reads into blocks and merged the outputs of subtasks to obtain the results. Due to sequencing errors, the proportion of k-mers with frequency <4 was relatively high in some datasets, which might result in the low efficiency of the segmented Bloom filter and cause a large memory footprint by keeping k-mers. Therefore, filtering low-frequency k-mers is critical in counting k-mers for large sequencing datasets, which will be one of our main foci to continue optimizing the KCOSS algorithm in the future. However, given that the KCOSS algorithm mainly orients to assembled genomes, its performance on sequencing data has been achieved as expected. We will optimize KCOSS for sequencing data while keeping the algorithm framework unchanged in the future.

4.1 Running time

KCOSS has a significant time advantage over Jellyfish2, KMC3 and CHTKC for assembled genomes. The running time of KCOSS does not increase significantly with the increase of k, while the running time of Jellyfish2, KMC3 and CHTKC positively correlates with k.

Table 2. Counting k-mers for *H.sapiens* with 62 threads

Method	$k = 18$			$k = 32$			$k = 55$			$k = 64$		
	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time
Jellyfish2	14.74	17.03	377.75	36.33	28.84	482.24	72.02	46.63	665.43	85.58	52.42	786.46
CHTKC-O	—	—	—	—	—	—	—	—	—	—	—	—
KMC3-R	12.12	8.54	96.07	13.36	18.78	151.95	12.69	35.68	262.2	13.24	41.33	293.46
KCOSS	14.32	5.45	93.20	12.68	3.22	56.71	16.70	4.76	84.62	16.69	4.89	86.80

Note: CHTKC-O reports 'parse file error' on the human genome.

Table 3. Counting k-mers for *T.urartu* with 62 threads

Method	$k = 18$			$k = 32$			$k = 55$			$k = 64$		
	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time
Jellyfish2	7.81	9.59	218.28	36.33	21.58	469.49	72.02	47.63	750.55	85.58	57.71	866.95
CHTKC-O	24.24	5.28	91.34	35.17	14.06	143.74	37.27	36.28	350.03	37.27	44.56	410.23
KMC3-R	13.63	4.40	91.42	14.62	12.50	129.97	13.60	33.31	242.07	14.41	41.89	318.22
KCOSS	25.27	4.84	122.54	25.27	6.19	124.5	34.39	11.47	192.49	34.39	11.87	193.97

Table 4. Counting k-mers for *T.chinensis* with 62 threads

Method	$k = 18$			$k = 32$			$k = 55$			$k = 64$		
	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time
Jellyfish2	7.81	15.61	213.23	36.33	26.95	461.58	72.02	42.85	631.00	85.58	47.92	712.51
CHTKC-O	35.58	9.53	98.23	37.27	20.00	167.22	37.27	35.79	291.46	37.27	40.84	331.09
KMC3-R	11.74	7.94	91.73	13.00	17.70	143.16	12.60	33.08	234.18	12.86	38.10	271.52
KCOSS	17.24	4.55	76.12	15.41	2.47	48.88	18.54	3.21	61.60	18.53	3.24	61.55

Table 5. Counting k-mers for *C.annuum* with 62 threads

Method	$k = 18$			$k = 32$			$k = 55$			$k = 64$		
	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time
Jellyfish2	7.81	12.35	196.77	36.33	26.54	462.05	72.02	45.02	687.73	85.58	50.65	781.37
CHTKC-O	28.96	7.05	83.95	37.27	19.21	160.42	37.27	37.38	334.4	37.27	42.62	387.26
KMC3-R	11.52	5.88	77.28	13.06	16.96	135.87	12.74	34.53	249.59	13.10	40.12	297.01
KCOSS	21.22	5.34	106.58	17.17	3.09	59.62	21.41	2.88	66.49	21.40	2.74	65.89

Table 6. Counting k-mers for *P.magnuspinnatus* with 62 threads

Method	$k=18$			$k=32$			$k=55$			$k=64$		
	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time
Jellyfish2	4.09	4.55	59.19	9.51	7.12	76.9	18.35	11.08	105.16	21.78	12.36	114.74
CHTKC-O	17.82	2.87	29.72	19.52	5.26	42.27	22.35	9.15	60.93	22.42	10.43	67.63
KMC3-R	5.15	2.39	29.56	9.87	4.68	40.68	9.54	8.54	63.01	11.46	9.82	70.52
KCOSS	2.94	0.92	17.04	2.64	0.65	11.77	2.76	0.85	13.89	2.59	0.87	13.63

Table 7. Counting k-mers for *E.gouldiae* with 62 threads

Method	$k=18$			$k=32$			$k=55$			$k=64$		
	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time
Jellyfish2	4.09	7.83	95.95	18.62	11.28	173.17	36.20	17.08	224.6	43.15	19.01	243.05
CHTKC-O	23.34	4.94	46.33	25.01	8.35	63.91	30.19	14.05	92.19	30.23	15.96	102.45
KMC3-R	8.10	4.12	45.31	11.43	7.42	63.58	11.24	13.12	95.81	11.67	15.02	106.50
KCOSS	4.79	1.55	27.88	4.08	0.92	18.64	5.19	1.31	24.74	4.30	1.35	22.79

Table 8. Counting k-mers for *S.punctatus* with 62 threads

Method	$k=18$			$k=32$			$k=55$			$k=64$		
	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time
Jellyfish2	14.74	19.46	425.88	36.33	35.86	583.02	72.02	59.38	813.55	85.58	66.82	957.89
CHTKC-O	37.27	11.78	132.05	37.27	26.5	250.77	37.27	49.05	442.77	37.27	56.45	497.54
KMC3-R	13.00	9.65	119.36	13.82	23.16	189.59	13.31	45.54	363.79	13.82	52.88	432.23
KCOSS	18.92	6.63	121.17	15.48	4.32	73.91	19.72	5.52	90.82	19.72	5.53	90.58

The excellent time performance of KCOSS owes to the optimization of memory access and the application of advanced data structure. The techniques of filtering the first-occurrence k-mers, storing the overlapping set to disk, and keeping only non-singletons in memory decrease the memory requirement and the number of memory access. The less memory consumption and the more centralized memory access improve the locality of accessing memory, increase the cache hit and reduce the time consumption.

Modern processors typically have a cache line length of 512 bits. For a sequence consists of consecutive singletons of 64-mer, a cache line can store at most 256 bases or a C-read containing 193 consecutive singletons. The KCOSS occurs a cache miss only when it creates a C-read, and the key-value pair approach results in a cache miss per singleton in the worst case. It means KCOSS can reduce the cache miss from 193 to 1 in the best case. Generally, there is a nearly 100-fold speed difference in accessing L1-cache and main memory. More cache hit reduces the times of memory access and decreases the running time of counting consecutive singletons. When we increase k, the ratio of singleton increases, and more consecutive singletons appear. Therefore, KCOSS has a short running time when k is large.

Meanwhile, the utilization of the disk mapping technique *mmap* noticeably avoids redundant memory copy operations and reduces the risk of I/O bottleneck. Also, the application of memory block recycling for memory management reduces the performance overhead caused by frequent system creation and reclaiming memory space.

The improvement of the delivered performance of Jellyfish2 lies in the utilization of thread-safe and lock-free hash tables as well as prefix arrays. However, the shared data structure restricts the increase. As the number of threads increases, the memory consumption increases, the competition of shared cache becomes more intense. The speedup ratio slowly increases and gradually becomes stable. Data structures such as Bloom filter and hash table need to be shared among multiple threads, which also affect the further improvement of performance. Therefore, to improve the parallelism, we designed a segmented Bloom filter to reduce the race access and used the atomic operation to access the hash table.

The KMC3 is a disk-based algorithm and uses a two-stage strategy to count frequency, and it needs two rounds of disk read-write operations. The first round reads sequences from the disk, divides the k-mers into groups and writes groups to the disk. The second round loads partitions from the disk and writing the merged results to the disk. IO bandwidth of the computer system affects the performance of disk-based counting algorithms. Our test shows that the disk access time of KMC3 at $k=12$ is around 160 seconds for the human genome, which is the total running time of the algorithm exactly since KMC3 uses asynchronous writes. In contrast, KCOSS only needs one round of disk read-write operation, utilizes less disk space and uses asynchronous write. Therefore, IO bandwidth has less influence on KCOSS than KMC3, and multithreading works better.

CHTKC uses a chaining hash table and lock-free technology to achieve high performance and utilizes as much memory as possible to handle more entries in memory and speed up the entire process. CHTKC prefers to use all pre-allocated memory, but the tremendous hash table with a load factor of 0.75 restricts the memory efficiency. The random access to memory makes it is hard to increase the cache hit. Therefore, we found that CHTKC did not have the advantage over KMC3 for $k=64$ when the memory consumption increased.

The drawback of the algorithm is that, although the speedup ratio continues to improve, the parallel efficiency decreases obviously with the increase of the thread number. When the thread number exceeds 48, the consequence of resource competition tends to counteract the benefit of multiple threads, and keeping increasing thread number does not significantly reduce the elapsed time (Supplementary Fig. S2). Reducing the competing access to resources is still the best option to improve the time performance of the k-mer counting algorithm in a shared memory computing environment.

4.2 Memory consumption

KMC3, Jellyfish2 and KCOSS adopt an active write disk strategy, while CHTKC adopts an aggressive memory strategy. It is not surprising that the memory overhead of the first three algorithms is relatively low when $k = 18$. The memory consumption of Jellyfish2 linearly grows as k increases. When $k \leq 14$, KCOSS uses a single hash table to count k-mers, and the total memory consumption of KCOSS is $22*k + 2$MB if we use 32 bits unsigned integers to save the counting results. When $k > 14$, the memory consumption of KCOSS mainly consists of the Bloom filter and hash tables. The overlapping sequence set is asynchronously written to disk while counting, so the memory consumption of the overlapping sequence set is limited. As the proportion of singleton k-mer increases with k, the overlapping sets intercept more k-mers, and the hash tables process fewer k-mers are processed. The total memory consumption increases little for both 17-mers to 32-mers and 33-mers to 64-mers overall. However, for large genomes such as *T.urartu*, whose singleton ratio and total k-mer ratio are small even in the case $k = 64$, due to fewer consecutive singleton cases, more k-mers need to be processed in the fixed-size hash table and elastic hash table. Therefore, the memory consumption of KCOSS is high, which is a puzzle that we need to address in the future.

By contrast, KMC3 is a counting algorithm based on external memory, which needs a large amount of disk space instead of main memory to complete counting with limited memory. The memory management of KMC3 is very efficient, and the main factor that affects the algorithm performance is the high I/O cost. Jellyfish2 uses thread-safe and lock-free hash tables and prefix arrays, but it does not optimize the k-mer compression except for binarization. When k increases, the memory usage of Jellyfish 2 increases linearly. CHTKC's strategy is to use memory as much as possible to complete the count, resulting in more memory consumption than KCOSS and KMC3.

4.3 Disk space utilization

The disk consumption consists of the disk used in counting k-mer and the disk used for dumping k-mer and its frequency. KCOSS saves the overlapping sequence set to disk when a memory block is full of C-reads in the process of counting k-mer. All k-mers are treated as singleton when they first enter the counting pipeline, so the maximal size of the overlapping set is the number of distinct k-mer. KCOSS merges continuous singletons as a C-read. The ratios of the total number of singletons to the genome length and the total number of k-mers to the genome length determine the length distribution of C-read. Therefore, genome size, the number of distinct k-mers and the number of singletons decide the size of the overlapping set and subsequently affect the disk size used in counting. Let us denote the ratios of the total number of singletons to the genome length and the total number of k-mers to the genome length as β and γ, respectively. Given a reference genome and k, we find the disk usage is inversely proportioned to the product of β and γ when the product is large than 0.1. However, when $\beta\gamma < 0.1$, both the proportion of singleton number to genome length and total k-mers are small, and the number of C-read is small too, leading to that the rule does not hold.

In contrast, Jellyfish2 converts character bases into a binary stream without further optimizations. The KMC3 uses Minimizers to compress the k-mer, and its total disk consumption is slightly less than Jellyfish2. The actual hard disk usage change trend is similar to Jellyfish2, which increases linearly as the k value increases. CHTKC compresses the k-mers that do not exist in memory and temporarily stores them on disk. Consecutive k-mers are compressed into longer super k-mers to save disk space and I/O costs. For continuous k-mer processing, CHTKC and KCOSS have similarities, and the difference is that KCOSS merges consecutive first-occurrence k-mer to a C-read. The average C-read length in KCOSS is longer than the average super k-mer length in CHTKC. Therefore, KCOSS reduces more disk space occupation. The experimental results demonstrated that the disk overhead of KCOSS was lower than that of CHTKC in all test datasets.

5 Conclusion

K-mer frequency counting is an important step to dissect the complexity of genomes and assembled genome analysis. So far, researchers have proposed memory-based and disk-based solutions. In this article, we proposed an approach using the segmented Bloom filter and splitting the flow through the Bloom filter, which is different from simply kicking out the k-mer with an occurrence of 1. Through the use of lock-free Bloom filters, lock-free thread pools, lock-free hash tables, autonomous memory management, and disk mapping techniques, the overall performance of KCOSS is superior to a series of existing k-mer frequency counting algorithms. KCOSS has the advantages of low runtime memory consumption, short runtime and high output file compression rate and allows a k-mer size of up to 64. Due to the utilization of shared Bloom filters, too many threads will cause resource expropriation when they try to access the same memory area, and parallelism cannot play well when the number of cores exceeds 48. In the future, we will continue to improve the access performance of the shared Bloom filter to make the program better adapted to multi-core servers. We will also optimize the algorithm to reduce its dependence on hardware and improve the performance in processing sequencing data. Finally, sequence complexity (e.g. repeats) varies within different regions of a genome and among genomes. We will explore such features and improve the k-mer counting algorithm.

Acknowledgements

The authors thank Dr Pingjian Zhang and Dr Hu Chen for valuable suggestion, Mr Ibrahima Diallo for English editing, Mr Weihao Xiao for interface implementation, and Mr Jingyu Chen and Mr Xiaodong Lian for the help of data collection.

Funding

This work was partially supported by the National Key R&D Program of China [2018YFC0910200 to D.T.] and Cancer Prevention and Research Institute of Texas [CPRIT RP180734 to Z.Z.].

Conflict of Interest: none declared.

References

Audano,P. and Vannberg,F. (2014) KAnalyze: a fast versatile pipelined K-mer toolkit. *Bioinformatics*, **30**, 2070–2072.

Audoux,J. *et al.* (2017) DE-kupl: exhaustive capture of biological variation in RNA-seq data through k-mer decomposition. *Genome Biol.*, **18**, 243.

Bernard,G. *et al.* (2018) k-mer similarity, networks of microbial genomes, and taxonomic rank. *mSystems*, **3**, e00257-18.

Bonnici,V. *et al.* (2018) PanDelos: a dictionary-based method for pan-genome content discovery. *BMC Bioinformatics*, **19**, 437.

Břinda,K. *et al.* (2015) Spaced seeds improve k-mer-based metagenomic classification. *Bioinformatics*, **31**, 3584–3592.

Crusoe,M. *et al.* (2015) The khmer software package: enabling efficient nucleotide sequence analysis. *F1000 Research*, **4**, 900.

Cserhati,M. *et al.* (2019) K-mer-based motif analysis in insect species across Anopheles, Drosophila, and Glossina Genera and its application to species classification. *Comput. Math. Methods Med.*, **2019**, 4259479.

Deorowicz,S. (2020) FQSqueezer: k-mer-based compression of sequencing data. *Sci. Rep.*, **10**, 578.

Deorowicz,S. *et al.* (2013) Disk-based k-mer counting on a PC. *BMC Bioinformatics*, **14**, 160.

Deorowicz,S. *et al.* (2015) KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, **31**, 1569–1576.

Edgar,R.C. (2004) MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Res.*, **32**, 1792–1797.

Erbert,M. *et al.* (2017) Gerbil: a fast and memory-efficient k-mer counter with GPU-support. *Algorithms Mol. Biol.*, **12**, 9.

Fan,B. *et al.* (2013) Memc3: compact and concurrent memcache with dumber caching and smarter hashing. In: *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pp. 371–384.

Gordon,S.P. *et al.* (2020) Gradual polyploid genome evolution revealed by pan-genomic analysis of *Brachypodium hybridum* and its diploid progenitors. *Nat. Commun.*, **11**, 3670.

Jaffe,D.B. *et al.* (2003) Whole-genome sequence assembly for mammalian genomes: arachne 2. *Genome Res.*, **13**, 91–96.

Jaillard,M. *et al.* (2018) A fast and agnostic method for bacterial genome-wide association studies: bridging the gap between k-mers and genetic events. *PLoS Genet.*, **14**, e1007758.

Kaplinski,L. *et al.* (2015) GenomeTester4: a toolkit for performing basic set operations-union, intersection and complement on k-mer lists. *Gigascience*, **4**, s13742–13015.

Kelley,D.R. *et al.* (2010) Quake: quality-aware detection and correction of sequencing errors. *Genome Biol.*, **11**, R116.

Kokot,M. *et al.* (2017) KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, **33**, 2759–2761.

Kurtz,S. *et al.* (2008) A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*, **9**, 517.

Lees,J.A. *et al.* (2016) Sequence element enrichment analysis to determine the genetic basis of bacterial phenotypes. *Nat. Commun.*, 7, 12797.

Li,X. *et al.* (2014) Algorithmic improvements for fast concurrent Cuckoo hashing. *Proc. Ninth Eur. Conf. Comput. Syst. EuroSys*, **2014**, 1–14.

Li,Y. and Yan,X. (2015) MSPKmerCounter: a fast and memory efficient approach for K-mer counting. *arXiv:1505.06550*.

Mamun,A.-A. *et al.* (2016) KCMBT: ak-mer counter based on multiple burst trees. *Bioinformatics*, **32**, 2783–2790.

Manekar,S.C. and Sathe,S.R. (2018) A benchmark study of k-mer counting methods for high-throughput sequencing. *Gigascience*, **7**, 7–13.

Marcais,G. and Kingsford,C. (2011) A fast, lock-free approach for efficient parallel counting of occurrences of -mers. *Bioinformatics*, **27**, 764–770.

Melsted,P. and Pritchard,J.K. (2011) Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, **12**, 333.

Mohamadi,H. *et al.* (2017) ntCard: a streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, **33**, 1324–1330.

Nasko,D. *et al.* (2018) RefSeq database growth influences the accuracy of k-mer-based lowest common ancestor species identification. *Genome Biol.*, **19**, 165.

Pandey,P. *et al.* (2018) Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*, **34**, 568–575.

Rizk,G. *et al.* (2013) DSK: k -mer counting with very low memory usage. *Bioinformatics*, **29**, 652–653.

Röhling,S. *et al.* (2020) The number of k-mer matches between two DNA sequences as a function of k and applications to estimate phylogenetic distances. *PLoS One*, **15**, e0228070.

Roy,R. *et al.* (2014) Turtle: identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, **30**, 1950–1957.

Sievers,A. *et al.* (2017) K-mer content, correlation, and position analysis of genome DNA sequences for the identification of function and evolutionary features. *Genes (Basel)*, **8**, 122.

Tu,Q. *et al.* (2014) Strain/species identification in metagenomes using genome-specific markers. *Nucleic Acids Res.*, **42**, e67.

Wang,D. *et al.* (2015) KGCAK: a K-mer based database for genome-wide phylogeny and complexity evaluation. *Biol. Direct*, **10**, 53.

Wang,J. *et al.* (2020) CHTKC: a robust and efficient k-mer counting algorithm based on a lock-free chaining hash table. *Brief. Bioinf.*, **22**, bbaa063.

Xu,H. *et al.* (2021) DeepVISP: deep learning for virus site integration prediction and motif discovery. *Adv. Sci.*, **8**, 2004958.