BIOINFORMATICS ORIGINAL PAPER

Vol. 30 no. 14 2014, pages 1950–1957
doi:10.1093/bioinformatics/btu132

Sequence analysis

Advance Access publication March 10, 2014

Turtle: Identifying frequent *k*-mers with cache-efficient algorithms

Rajat Shuvro Roy[1,2,3,*], Debashish Bhattacharya[2,3] and Alexander Schliep[1,4,*]

[1]Department of Computer Science, [2]Department of Ecology, Evolution and Natural Resources, [3]Institute of Marine and Coastal Sciences and [4]BioMaPS Institute for Quantitative Biology, Rutgers University, New Brunswick, NJ 08901, USA

Associate Editor: Michael Brudno

ABSTRACT

Motivation: Counting the frequencies of *k*-mers in read libraries is often a first step in the analysis of high-throughput sequencing data. Infrequent *k*-mers are assumed to be a result of sequencing errors. The frequent *k*-mers constitute a reduced but error-free representation of the experiment, which can inform read error correction or serve as the input to *de novo* assembly methods. Ideally, the memory requirement for counting should be linear in the number of frequent *k*-mers and not in the, typically much larger, total number of *k*-mers in the read library.

Results: We present a novel method that balances time, space and accuracy requirements to efficiently extract frequent *k*-mers even for high-coverage libraries and large genomes such as human. Our method is designed to minimize cache misses in a cache-efficient manner by using a pattern-blocked Bloom filter to remove infrequent *k*-mers from consideration in combination with a novel sort-and-compact scheme, instead of a hash, for the actual counting. Although this increases theoretical complexity, the savings in cache misses reduce the empirical running times. A variant of method can resort to a counting Bloom filter for even larger savings in memory at the expense of false-negative rates in addition to the false-positive rates common to all Bloom filter-based approaches. A comparison with the state-of-the-art shows reduced memory requirements and running times.

Availability and implementation: The tools are freely available for download at http://bioinformatics.rutgers.edu/Software/Turtle and http://figshare.com/articles/Turtle/791582.

Contact: rajatroy@cs.rutgers.edu or schliep@cs.rutgers.edu

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

Received on April 29, 2013; revised on February 25, 2014; accepted on March 4, 2014

1 INTRODUCTION

K-mers play an important role in many methods in bioinformatics because they are at the core of the *de Bruijn* graph structure (Pevzner *et al.*, 2001) that underlies many of today's popular *de novo* assemblers (Simpson *et al.*, 2009; Zerbino and Birney 2008). They are also used in assemblers based on the overlap-layout-consensus paradigm like Celera (Miller *et al.*, 2008) and Arachne (Jaffe *et al.*, 2003) as seeds to find overlap between reads. Several read correction tools (Kelley *et al.*, 2010; Liu *et al.*, 2012; Medvedev *et al.*, 2011) use *k*-mer frequencies for error correction. Their main motivation for counting *k*-mers is to filter out or correct sequencing errors by relying on *k*-mers that appear multiple times and can thus be assumed to reflect the true sequence of the donor genome. In contrast, *k*-mers that appear only once are assumed to contain sequencing errors. Melsted and Pritchard (2011) and Marcais and Kingsford (2011) make a more detailed compelling argument about the importance of *k*-mer counting.

In a genome of size g, we expect up to g unique *k*-mers. This number can be smaller because of repeated regions (which produce the same *k*-mers) and small k, as smaller *k*-mers are less likely to be unique, but is usually close to g for reasonable values of k. However, depending on the amount of sequencing errors, the total number of *k*-mers in the read library can be substantially larger than g. For example, in the DM dataset (Table 2), the total number of 31-mers is ~289.20 M, whereas the number of 31-mers occurring at least twice is ~131.82 M. The size of the genome is 122 Mb (megabase pairs). This is not surprising because one base call error in a read can introduce up to k false *k*-mers. Consequently, counting the frequency of all *k*-mers, as done by Jellyfish (Marcais and Kingsford, 2011), which is limited to $k \leq 31$, requires $O(N)$ space where N is the number of *k*-mers in the read library. This makes the problem of *k*-mer frequency counting time and memory intensive for large read libraries like human. We encounter similar problems for large libraries while using Khmer (Pell *et al.*, 2012), which uses a Bloom filter-based (Bloom, 1970) approach for counting frequencies of all *k*-mers. Ideally, the frequent *k*-mer identifier should use $O(n)$ space where n is the number of frequent *k*-mers ($n \ll N$). The approach taken by BFCounter (Melsted and Pritchard, 2011) achieves something close to this optimum by ignoring the infrequent *k*-mers with a Bloom filter and explicitly storing only frequent *k*-mers. This makes BFCounter more memory-efficient compared with Jellyfish. However, the running time of BFCounter is large for two reasons. First, it is not multi-threaded. Second, both the Bloom filter and the hash table used for counting incur frequent cache misses. The latter has recently been identified as a major obstacle to achieving high performance on modern architectures, motivating the development of cache-oblivious algorithms and data structures (Bender *et al.*, 2005), which optimize the cache behavior without relying on information of cache layout and sizes. Additionally, BFCounter is also limited to a count range of 0–255, which will often be exceeded in single-cell experiments because of the large local coverage produced by whole genome amplification artifacts. A different approach is taken by DSK (Rizk *et al.*, 2013) to improve memory efficiency. DSK makes many passes over the read file and uses temporary disk space to trade off the memory requirement. Although Rizk *et al.* (2013) claimed DSK to be faster than BFCounter, on our machine

*To whom correspondence should be addressed.

using an 18 TB Raid-6 storage system; DSK required more wall-clock time compared with BFCounter. Therefore, we consider DSK without dedicated high-performance disks, e.g. solid state, and BFCounter to be too slow for practical use on large datasets. A disk-based sorting and compaction approach is taken by KMC (Deorowicz *et al.*, 2013), which was published very recently, and it is capable of counting k-mers of large read libraries with a limited amount of memory. However, in our test environment, we found it to be slower than the method described here.

We present a novel approach that reduces the memory footprint to accommodate large genomes and high-coverage libraries. One of our tools (scTurtle) can report frequent 31-mers with counts (with a very low false-positive rate) from a human read set with 135.3 Gb using 109 GB of memory in <2 h using 19 worker threads. Like BFCounter, our approach also uses a Bloom filter to screen out k-mers with frequency one (with a small false-positive rate), but in contrast to BFCounter, we use a pattern-blocked Bloom filter (Putze *et al.*, 2010). The expected number of cache misses for each inquiry/update in such a Bloom filter is one. The frequency of the remaining k-mers is counted with a novel sorting and compaction-based algorithm. Our compaction step is similar to run-length encoding (Salomon, 1997). Note that this is similar to the strategy of KMC, which was developed as a concurrent and independent work. Though the complexity of sorting in our compression step is $O(n \log n)$, it has sequential and localized memory access that helps in avoiding cache misses and will run faster than an $O(n)$ algorithm that has $O(n)$ cache misses as long as $\log n$ is much smaller than the penalty issued by a cache miss.

For larger datasets, where $O(n)$ space is not available, the aforementioned method will fail. We show that it is possible to get a reasonable approximate solution to this problem by accepting small false-positive and false-negative rates. The method is based on a counting Bloom filter implementation. The error rates can be made arbitrarily small by making the Bloom filter larger. Because the count is not maintained in this method, it reports only the k-mers seen more than once (with a small false-positive and false-negative rate), but not their frequency.

We call the first tool *scTurtle* and the second one *cTurtle*.

2 METHODS

2.1 scTurtle

2.1.1 Outline By a k-mer, we always refer to a k-mer and/or its reverse complement. Our objective is to separate the frequent k-mers from the infrequent ones and count the frequencies of the frequent k-mers. To achieve this, first, we use a Bloom filter to identify the k-mers that were seen at least twice (with a small false-positive rate). To count the frequency of these k-mers, we use an array of items containing a k-mer and its count. These are the two main components of our tool. Once the counts are computed, we can output the k-mers having a frequency greater than the chosen cutoff. For the sake of cache efficiency, the Bloom filter is implemented as a pattern-blocked Bloom filter (Putze et al., 2010). It localizes the bits set for an item to a few consecutive bytes (block) and thus reduces cache misses. The basic idea is as follows: when a k-mer is seen, the Bloom filter is checked to decide whether it has been seen before. If that is the case, we store the k-mer in the array with a count of 1. When the number of items in the array crosses a threshold, it is sorted in place, and a linear pass is made, compressing items with the same k-mer (which lie in consecutive positions of the sorted array) to one item. The counts add up to reflect the total number of times a k-mer was seen. Note that this strategy is similar to run-length encoding (Salomon, 1997) of the items. Our benchmarking (Table 1) shows that this simple approach of storing items and their frequencies is faster than a hash table-based implementation. An outline of the algorithm is given in **Algorithm 1**. More details are provided in the following subsections.

Table 1. Comparison of sort and compress and hash table-based implementations for counting items and their frequencies

Method	Number of insertions/updates			
	458 M		2.2 B	
	Time (s)	Space (GB)	Time (s)	Space (GB)
Sort and compress	153.37	2.70	523.41	7.10
Jellyfish	296.49	2.40	1131.70	7.20
Google dense hash	626.77	20.47	6187.95	40.38
Google sparse hash	1808.48	7.44	28069.18	10.60

Note: Jellyfish is a highly optimized hash table-based implementation for the k-mer counting problem. We also compare against general purpose tools that uses Google sparse/dense hash maps for storing k-mers and their counts.

Note that the improved efficiency of sort and compaction also suggests that it can speed up the k-mer counting step for all *de Bruijn* graph-based assemblers where k-mer counting is required for building the graph. We found that ABySS (Simpson *et al.*, 2009) and SPAdes (Bankevich *et al.*, 2012) require 3660 and 2144 s, respectively, for k-mer counting on the DM dataset (see Table 2). But the sort and compaction method takes only 523.41 s. We provide a single-threaded preliminary tool called *aTurtle* that implements this method for counting all k-mers and their frequencies.

Algorithm 1 scTurtle outline

1. Let S be the stream of k-mers coming from the read library, BF be the Bloom filter, A be the array to store k-mers with counts and t be the threshold when we apply sorting and compaction.
2. **for all** k-mer $\in S$ **do**
3. **if** k-mer present in BF then
4. Add k-mer to A
5. **end** if
6. **if** $|A| \geq t$ **then**
7. Apply sorting and compaction on A
8. **end** if
9. **end for**
10. Apply sorting and compaction on A.
11. Report all k-mers in A with their counts as frequent k-mers and their counts.

2.1.2 k-mer extraction and bit encoding For space efficiency, k-mers are stored in a bit-encoded form where 2-bits represent a nucleotide. This is possible because k-mers are extracted out of reads by splitting them on 'N's (ambiguous base calls) and hence contain only A, C, G and T. Because we consider a k-mer and its reverse complement to be two representations of the same object, whenever we see a k-mer, we also compute the bit representation of the reverse complement and take the numerically smaller value as the unique representative of the k-mer/reverse complement pair.

Table 2. Descriptive statistics about the datasets used for benchmarking

Set ID	Organism	Genome size (Mb)	Read library	Bases (Gb)
DM	*Drosophila melanogaster*	122	SRX040485	3.7
GG	*Gallus gallus*	1×10^3	SRX043656	34.7
ZM	*Zea mays*	2.9×10^3	SRX118541	95.8
HS	*Homo sapiens*	3.3×10^3	ERX009609	135.3

Note: The library sizes range from 3.7 to 135.3 Gb, and the genome size ranges from 122 Mb to 3.3 Gb.

2.1.3 Identification of frequent k-mers with pattern-blocked Bloom filter A Bloom filter is a space-efficient probabilistic data structure, which, given an item, can identify whether this item was seen before with some prescribed, small false-positive rate. We use this property of the Bloom filter to identify k-mers that were seen at least twice. An ordinary Bloom filter works as follows: a large bit-array (B) of size L is initialized to 0. Given an item x, k hash values $(h_1, h_2, \ldots, h_k)$ using k-independent hash functions {within the range $[0, (L-1)]$} are computed. We now check all the bits $B[h_1], \ldots, B[h_k]$. If they are all set to 1, with high probability, this item has been seen at least once before. If not, it is certainly the first appearance of this item, and we set all of $B[h_1], \ldots, B[h_k]$ to one. For all subsequent appearance(s) of this item, the Bloom filter will report that it has been seen at least once before. In this way, the Bloom filter helps us to identify frequent k-mers. Note that if the bit locations are randomly distributed, because of the large size of the Bloom filter, each bit inspection and update is likely to incur one cache miss. Thus, the total number of cache misses per item would be k. On the contrary, if the bit locations are localized to a few consecutive bytes (a block), each item lookup/update will have a small number of cache misses. This can be done by restricting $h_1, \ldots, h_k$ to the range $[h_1(x), h_1(x) + b]$ where b is a small integer. The bit pattern for each item can also be precomputed. This is called the pattern-blocked Bloom filter. Putze *et al*., (2010) observe that the increase in false-positive rate because of this localization and precomputed patterns can be countered by increasing L by a few percent. To summarize, we first select a block for an item (using a hash function), select $h_1 < h_2 < \ldots < h_k$ from a set of precomputed random numbers such that all of them lie within the block and update/inquire them sequentially. Note that Bloom filters are widely used in many applications like assembly (Chikhi and Rizk, 2012; Pell et al., 2012), and we believe using a more optimized version of this data structure (like the pattern-blocked Bloom filter) will benefit such applications.

2.1.4 Counting frequencies with sorting and compaction Our next objective is to count the frequencies of the frequent k-mers. The basic idea is to store the frequent k-mers in an array A of size $> n$, where n is the number of frequent items. When this array fills up, we sort the items by the k-mer values. This places the items with the same k-mer next to each other in the array. Now, by making a linear traversal of the array, we can replace multiple items with the same k-mer with one item where a count field represents how many items were merged, which is equal to how many times this k-mer was seen; see Figure 1. Note that this is similar to run-length encoding. Here is a toy example: say $A = [\ldots, (i, 1), \ldots, \ldots, (i, 1), \ldots\ldots, (i, 1)]$. After sorting $A = [\ldots, \ldots, (i, 1), (i, 1), (i, 1), \ldots\ldots]$ and compressing results in $A = [\ldots, (i, 3), \ldots]$. We have to repeat these steps until we have seen all items. To reduce the number of times, we sort the complete array, and we apply the following strategies. We select a threshold $n < t < |A|$. We start with an unsorted k-mer array. It is sorted and compacted (Phase-0 Sorted and Compacted array or Phase-0 SAC). We progress in phases as follows. At phase i, a certain number of items in the beginning of the array are already sorted and compressed [Phase-(i–1) SAC]. The new incoming k-mers are stored as unsorted items in the empty part of the array. Let m be the total number of items in the array. When $m > t$, we sort the unsorted items. Many of these k-mers are expected to exist in Phase-(i–1) SAC. We make a linear traversal of the array replacing k-mers present in both Phase-(i–1) SAC and the newly sorted part with one item in Phase-(i–1) SAC. The k-mers not present in Phase-(i – 1) SAC are represented with one item in the newly sorted part. The counts are added up to reflect the total number of times a k-mer was seen. This takes $O(m)$ time. Note that this compaction has sequential and localized memory access, which makes it cache-efficient. After a few such compaction steps, $m > t$, and we sort and compress all the items in the array to produce Phase-i SAC.

By repeatedly applying this mechanism on the frequent items, we ultimately get the list of frequent k-mers with their counts decremented by 1. This is due to the fact that when inserted into the array for the first time, an item was seen at least twice unless it is a false-positive finding. To offset this, we simply add 1 to all counts before writing them out to a file.

2.1.5 Parallelization We implemented a one-producer, multiple-consumer model with a pool of buffers. The producer extracts k-mers from the reads and distributes them among the consumers. Each consumer has its own Bloom filter. Because a k-mer should always pass through the same Bloom filter, we distribute the k-mers to the consumers using the modulo operation, which is one of the cheapest hash functions available. Because modulo a prime number shows better hash properties compared with non-primes, it is recommended that one uses a prime (or at least an odd) number of threads because this spreads out the k-mers more evenly among the consumers, which is helpful for speeding up the parallelization. The k-mers are stored in buffers, and only when the buffers fill up are they transferred to the consumer. Because consumers consume the k-mers at an uneven rate, having the same fixed buffer size for all consumers may cause the producer to block if the buffer for a busy consumer fills up. To reduce such blocking, we have a pool of buffers, and the number of buffers is more than the number of consumers. If a consumer is taking longer to consume its items, the producer has extra buffers to store its k-mers in. This improves the speedup.

With many consumers (usually >13), the producer becomes the bottleneck. Therefore, it is important to make the producer more efficient. The two most expensive parts of the producer are converting reads to k-mers and the modulo operation required to determine which consumer handles a particular k-mer. Modern computers support SSE (Patterson and Hennessey, 1998) instructions that operate on 128-bit registers and can perform arithmetic/logic operations in parallel on multiple variables. We used Streaming SIMD Extensions (SSE) instructions for speeding up bit encoding of k-mers. It is also possible to design approximate modulo functions that execute much faster than regular modulo instruction for some numbers (e.g. 5, 7, 9, 10, 63) (Warren, 2012). But each of these functions has to be custom designed. If we restrict the number of consumers to the numbers that have efficient modulo function, it is possible to improve the producer's running time even further.

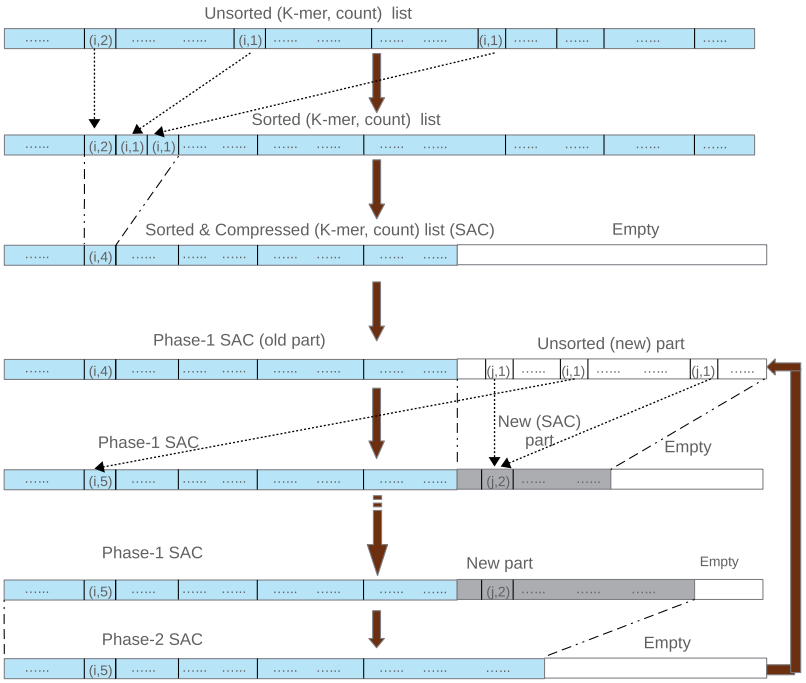

Fig. 1. The Sorting and compaction mechanism. We start with an unsorted k-mer array. It is sorted and compacted (Phase-0 SAC). The empty part is filled with unsorted k-mers, sorted and compacted. After repeating this step several times, the compacted new part almost fills up the whole array. Then, all items are sorted and compacted to produce Phase-1 SAC. This cycle repeats until all k-mers have been seen

2.1.6 Running time analysis We first analyze the sort and compress algorithm. Let the total number of frequent k-mers (those with frequency ≥ 2) be N, and let n be the number of distinct frequent k-mers. We use an $xn, x > 1$, sized array A for storing the frequent k-mers and their counts. First, consider the following simplified version of our algorithm: $(x-1)n$ new items are loaded into the array, and they are sorted and compacted. Because there are n distinct k-mers, at least $xn - n = (x-1)n$ locations will be empty after sorting and compaction. We again load $(x-1)n$ items and perform sorting and compaction. We iterate until all N items have been seen. Each iteration takes $O(xn \log xn + xn)$ time, and we have at most $N/(x-1)n$ such iterations. Thus, the total time required is:

$$O\left(\frac{N}{(x-1)n}(xn \log xn + xn)\right) = O\left(\frac{x}{(x-1)}(N \log xn + N)\right) \leq O\left(\frac{x}{(x-1)}(N \log N + N)\right)$$

As discussed earlier, to reduce the number of times sorting is performed, which is more expensive than compaction, we implemented a modified version of the aforementioned method, which delays sorting at the expanse of more compactions. Our benchmarking shows this to be faster than the naive method. The algorithm we implemented progresses in phases as follows. At the beginning of phase i, the array is filled up with unsorted elements. They are sorted and compacted [$O(xn \log xn + xn)$]. This is called the Phase-(i–1) SAC. Let e be the number of empty locations after each complete sorting and compaction step. Then, $(x-1)n \leq e \leq xn$. The new incoming k-mers are stored as unsorted items in the empty locations. When the empty part is full, we sort the new items [$O(xn \log xn)$]. Many of these k-mers are expected to exist in Phase-(i–1) SAC. We make a linear traversal of the array replacing k-mers present in both Phase-(i–1) SAC and the newly sorted part with one item in Phase-(i–1) SAC. The k-mers not present in Phase-(i–1) SAC are represented with one item in the newly sorted part. The counts are added up to reflect the total number of times a k-mer was seen. The total cost of a lazy compaction is therefore upper bounded by $O(xn \log xn + xn)$. This again creates empty locations at the end of the array, which allows us to perform another round of lazy compression. We assume that the incoming items are uniformly distributed, and every lazy compaction stage reduces the size of the empty part by an approximately constant fraction $1/c$. Therefore, on average, we expect to have c lazy compaction stages. This completes Phase-i, the expected cost of which is upper bounded by:

$$O(xn \log xn + xn + c(xn \log xn + xn)) = O((c+1)(xn \log xn + xn))$$

To compute how many phases are expected to consume all N items, we observe that, at every phase, the lazy compaction steps consume a total of at least $(x-1)n\{1 + (1 - \frac{1}{c}) + (1 - \frac{2}{c}) + \ldots + (1 - \frac{c-1}{c})\} = (x-1)n(c+1)/2$ items. So, on average, each phase consumes at least $(c+1)n(x-1)/2$ items, and hence the expected number of phases is at most $2N/n(c+1)(x-1)$. Therefore, the total expected cost would be:

$$\leq \frac{2N}{(c+1)n(x-1)} O(xn(c+1) \log xn + xn(c+1)) = \frac{2x}{(x-1)} O(N \log xn + N) \leq O\left(\frac{x}{(x-1)}(N \log N + N)\right)$$

Note that we obtained the same expression for the naive version of sorting and compaction. It is surprising that this expression is independent of c. As an intuitive explanation, observe that more lazy compactions within a phase result in more items being consumed by a phase, which in turn decreases the number of phases. This inverse relationship between c and the number of phases makes the running time independent of c. We found the naive version to be slower than the implemented version in empirical tests and therefore believe our bound to be an acceptable approximation.

Table 3. Comparative results of wall-clock time and memory between Khmer, KMC, scTurtle and cTurtle on a machine with 48 cores (AMD OpteronTM 6174 Processors, clocked at 2.2 GHz) and 256 GB memory for 5–19 worker threads

Set ID	Tool	Multi-worker-threaded wall-clock time (min:sec)								Memory (GB)
		5	7	9	11	13	15	17	19	
DM	Khmer	17:03	13:53	11:28	9:27	8:30	8:37	7:39	8:11	19.1
	KMC	7:30	5:41	3:55	3:44	3:14	3:00	2:47	2:38	5.6
	scTurtle	4:43	3:40	3:08	2:52	2:49	2:57	2:56	2:58	5.3
	cTurtle	3:41	2:43	2:04	1:55	1:55	1:55	1:55	1:56	4.2
GG	Khmer	163:17	127:50	111:26	101:18	90:57	102:13	86:12	63:15	59.7
	KMC	70:52	67:01	43:54	52:06	44:17	32:58	32:01	33:40	46.8
	scTurtle	57:52	45:09	42:48	37:07	38:59	33:15	31:28	31:29	44.9
	cTurtle	45:16	33:04	23:40	21:37	21:06	21:16	21:34	21:21	29.9
ZM	Khmer	482:39	353:12	287:12	240:19	215:33	220:35	179:58	275:05	234.4
	KMC	194:57	184:22	115:58	107:24	106:15	86:30	87:48	82:55	82.0
	scTurtle	159:49	122:42	104:22	84:48	79:25	80:39	84:26	86:31	82.1
	cTurtle	131:00	98:48	72:48	65:12	65:36	65:12	63:24	63:48	51.6
HS	Khmer	≥600:00	≥600:00	≥600:00	584:50	336:03	248:55	232:42	223:04	234.4
	KMC	253:18	303:8	170:42	149:22	147:40	129:47	121:14	111:27	108.8
	scTurtle	219:21	157:26	127:21	110:05	111:04	112:05	112:16	116:30	109.5
	cTurtle	171:00	123:12	98:00	87:12	91:12	89:24	88:00	90:24	68.5

Note: The input of a run is a single read file (FASTA or FASTQ format), and the output is a text file containing k-mers and their frequencies (FASTA or tab delimited format). Khmer does not provide a tool for dumping. Therefore, its run times are for counting only. Each reported number is an average of 5 runs. Runs requiring >10 h were not reported. The k-mer size is 31. Recall that KMC, scTurtle and Khmer report k-mers and their counts, whereas cTurtle only reports the k-mers with count >1.

We now analyze the performance of sorting and compaction-based strategy against a hash table-based strategy for counting frequency of items. Let p be the cache miss penalty, h be the hashing cost, s be the comparison and swapping cost for sort and compress and b be the number of items that fits in the cache. The cost of frequency counting in the hash-based method will be $(p+h)N$ because each hash update incurs one cache miss. For sorting and compress, we will have one cache miss for every b operation, and thus the cost for sorting and compaction will be $(p/b+s)a(N\log N+N)$, where $a=\frac{x}{(x-1)}$. To compute the value of N for which sorting and compaction will be faster than a hash-based method, we write:

$$(p+h)N \geq (p/b+s)a(N\log N+N)$$

$$\log N \leq \frac{(p+h)}{(p/b+s)a}-1$$

Let a comparison and swap be one unit of work. A conservative set of values like $s=1, p=160$ (Levinthal, 2008), $h=8$, $b=256$ (assuming 8 byte items and 2 KB cache), $a=2$ results in $N\leq 2^{50}$. Therefore, for a large range of values of N, with a fast and moderate-sized cache, the sorting and compaction-based method would run faster than a hash-based method.

Because every observed k-mer has to go through the Bloom filter, the time required in the Bloom filter is $O(M)$ where M is the total number of k-mers in the read library. Thus, the total running time that includes the Bloom filter checks and the sorting and compression of the frequent items is $O(M)+O(N\log N+N)$. Our measurements on the datasets used show that the total time is dominated by the Bloom filter updates [i.e. $O(M)>O(N\log N+N)$].

2.2 cTurtle

When there are so many frequent k-mers that keeping track of the k-mers and their counts explicitly is infeasible, we can obtain an approximate set of frequent k-mers by using a counting Bloom filter. Note that the number of bits required for a Bloom filter for n items is $O(n)$, but the constants are small. For example, it may be shown that for a 1% false-positive rate, the Bloom filter size is recommended to be ~9.6n bits (Fan *et al.*, 2000). On the other hand, with a k-mer size of 32 and counter size of 1 byte, the memory required by a naive method that explicitly keeps track of the k-mers and their count is at least 9n bytes or 72n bits. With data compression techniques like prefix of k-mers being implied from the context of the data structure as in Jellyfish and KMC, this is <9n bytes but, from the memory comparison between Jellyfish and cTurtle presented in Table 3, we believe it still remains considerably higher than the 9.6n bits required by the Bloom filter.

The basic idea of our counting Bloom filter is to set k bits in the Bloom filter when we see an item for the first time. When seen for the second time, the item is identified as a frequent k-mer and written to the disk. To record this writing, k' more bits are set in the Bloom filter. For all subsequent sightings of this item, we find the $(k+k')$ bits set and know that this is a frequent k-mer that has already been recorded. For cache efficiency, we implement the counting Bloom filter as a pattern-blocked counting Bloom filter as follows. We take a larger Bloom filter (B) of size L. When an item x is seen, k values $(h_1, h_2, \ldots, h_k)$ within the range $[h(x), h(x)+b]$, where $h(x)$ is a hash function and b is the block size, are computed using precomputed patterns. If this is the first appearance of x, with high probability, not all of the bits $B[h_1], \ldots, B[h_k]$ are set to 1, and so we set all of them. When we see the same item again, we will find all of $B[h_1], \ldots, B[h_k]$ set to 1. We then compute another set of locations $(h_{k+1}, h_{k+2}, \ldots, h_{k+k'})$ within the range $[h(x)+b, h(x)+2b]$ using precomputed patterns. Again, with high probability, not all of $B[h_{k+1}], \ldots, B[h_{k+k'}]$ are set to 1, and so we set all of them. At the same time, we write this k-mer to the disk as a frequent k-mer. For all subsequent observations of this k-mer, we will find all of $B[h_1], \ldots, B[h_{k+k'}]$ set to 1 and will avoid writing it to the disk. Note that a false-positive rate in the second stage means that we do not write the k-mer out to file and thus have a false-negative rate.

Currently, cTurtle reports k-mers with frequency >1. But this strategy can be easily adopted to report k-mers of frequency greater than $c>1$. We argue that for most libraries with reasonable uniform coverage, $c=1$ is sufficient. Let C be the average nucleotide coverage of a read library with read length R. Then, the average k-mer coverage is $C_k = \frac{C(R-k+1)}{R}$ (Zerbino and Birney, 2008). Suppose we have an erroneous k-mer with one error. The probability that the same error will be reproduced is $\frac{1}{3k}$ where $1/k$ is the probability of choosing the same position, and 1/3 is the probability of making the same base call error. Therefore, the expected frequency of that erroneous k-mer is $1+\frac{C_k-1}{3k}$. For $R=100$ and $k=31$, this expression is $1+0.0075C$. Therefore, we need $C>132.85$ at a location for an erroneous 31-mer to have a frequency >2. Because most large libraries are sequenced at a much lower depth ($<60x$), such high coverage is unlikely except for exactly repeated regions, and therefore our choice of frequency cutoff will provide a reasonable set of reliable k-mers. However, this does not hold for single-cell libraries, which exhibit uneven coverage (Chitsaz *et al.*, 2011). Note that frequent k-mers are considered reliable only for uniform coverage libraries, and thus single-cell libraries are excluded from our consideration.

The parallelization strategy is the same as that for scTurtle.

3 COMPARISONS WITH *K*-MER COUNTERS

The datasets we use to benchmark our methods are presented in Table 2. The library sizes range from 3.7 to 135.3 Gb for genomes ranging from 122 Mb to 3.3 Gb. Experiments were performed on a 48-core computer with AMD OpteronTM 6174 processors, clocked at 2.2 GHz, 256 GB of memory and 18 TB Raid-6 storage system, and an 80-core Intel machine with Intel® Xeon® CPU E7-4870, clocked at 2.40 GHz and 1 TB of memory. According to our experiments, with limited memory, KMC (Deorowicz *et al.*, 2013) is the fastest open-source k-mer counter. DSK (Rizk *et al.*, 2013) is also memory-efficient but is slow. Khmer (Pell *et al.*, 2012) and BFCounter (Melsted and Pritchard, 2011) use Bloom filter-based methods for reducing memory requirements. We have a similar strategy for memory reduction but achieve a much better computational efficiency.

We decided not to report times for any tool that required >10 h of wall-clock time, and therefore some data are missing in Table 3. KMC was able to perform k-mer counting for all the datasets but was slower than Turtle for the larger datasets. Note that for the sake of comparison, we allowed KMC to use the same amount of memory that Turtle used, but it is capable of performing the computation with smaller amount of memory. Unexpectedly, on large datasets (ZM and HS), BFCounter required more memory than scTurtle (for 128 versus 109 GB). We suspect this is due to the memory overhead required to reduce collusions in the hash table used for storing frequent k-mers, which we avoid using in our sort and compaction algorithm. Rizk et al. (2013) claimed DSK to be faster than BFCounter, but on our machine, which had an 18 TB Raid-6 storage system of 2 TB SATA disks, it proved to be slower (1591 versus 1012 min for the GG dataset). Rizk et al. (2013) reported performance using more efficient storage systems (solid-state disks). This might explain DSK's poor performance in our experiments. The detailed results are presented in Table 3 for multi-threaded Khmer, KMC, scTurtle and cTurtle. Because BFCounter (single threaded) and DSK (4 threads) do not allow a variable number of threads, we present their results separately in Table 4.

Table 4. Performance of BFCounter and DSK (4 threads) for 31-mers

Set ID	Tool	Wall-clock time (min:s)	CPU utilization (%)	Space (GB)
DM	BFCounter	78:35	99	3.24
	DSK	170:37	318	4.86
GG	BFCounter	1011:51	99	29.26
	DSK	1590:54	290	48.59
ZM	BFCounter	>2289:00	NA	>166.00
	DSK	>2923:00	NA	NA
HS	BFCounter	>3840:00	NA	>128.00
	DSK	>1367:00	NA	NA

Note: Some of the results are not available because those computations could not be completed within a reasonable time.

Table 5. Comparative results of wall-clock time between Jellyfish, KMC, scTurtle and cTurtle on a SMP server with 80 cores (Intel® Xeon® CPU E7-4870 clocked at 2.40 GHz) and 1 TB of memory

Set ID	Tool	Wall-clock time (min:s)	Memory (GB)
DM	Jellyfish	3:01	7.4
	KMC	1:48	5.6
	scTurtle	2:12	5.3
	cTurtle	1:59	4.2
GG	Jellyfish	32:03	81.9
	KMC	17:58	46.8
	scTurtle	22:39	44.9
	cTurtle	21:31	23.9
ZM	Jellyfish	42:44	158.2
	KMC	90:36	82.0
	scTurtle	60:15	82.1
	cTurtle	58:43	51.6
HS	Jellyfish	197:42	238.0
	KMC	103:27	108.8
	scTurtle	88:31	109.5
	cTurtle	85:39	68.5

Note: The input of a run is a single read file (FASTA or FASTQ format), and the output is a text file containing k-mers and their frequencies (FASTA or tab delimited format). The k-mer size is 31. Each tool was run six times with 19 worker threads, and the average was reported.

Jellyfish's (Marcais and Kingsford, 2011) performance was inconsistent on the AMD machine (details not shown). For example, while running with 17 worker threads, it started with a near-perfect central processing unit (CPU) utilization of ~1700% but steadily declined to ~100%, resulting in an average CPU utilization of only 290%. The computation required 16 h and 56 min of wall-clock time and 238 GB of memory. This is inconsistent with Jellyfish's performance on Intel machines reported by Marcais and Kingsford (2011) and Rizk et al. (2013). Therefore, to compare our tools with Jellyfish, we ran additional experiments on the Intel machine. Table 5 presents the wall-clock times for Jellyfish, KMC, scTurtle and cTurtle run with 19 worker threads on the Intel machine for all the datasets. On

Table 6. Performance of scTurtle (counting only) and cTurtle for 64-mers

k-mer size	Set ID	Tool	Wall-clock time (min:sec)	CPU utilization (%)	Space (GB)
31	DM	scTurtle	02:18	2335.8	5.50
		cTurtle	1:28	580.0	4.20
	GG	scTurtle	24:48	2388.0	47.10
		cTurtle	28:51	413.2	29.90
	ZM	scTurtle	55:57	1838.0	82.15
		cTurtle	74:46	756.0	51.60
	HS	scTurtle	73:24	1563.0	109.53
		cTurtle	98:24	512.0	68.55
48	DM	scTurtle	2:33	1790.0	8.30
		cTurtle	2:30	871.4	4.76
	GG	scTurtle	25:11	1373.8	70.69
		cTurtle	25:29	693.8	29.34
	ZM	scTurtle	90:16	1125.0	129.09
		cTurtle	81:28	782.0	52.35
	HS	scTurtle	112:11	953.4	172.11
		cTurtle	105:10	657.6	69.29
64	DM	scTurtle	1:40	948.0	8.30
		cTurtle	1:28	580.0	4.76
	GG	scTurtle	31:60	825.8	70.69
		cTurtle	28:51	413.2	29.35
	ZM	scTurtle	79:90	1037.0	129.09
		cTurtle	74:46	756.0	52.35
	HS	scTurtle	79:56	952.0	172.11
		cTurtle	98:24	512.0	69.29

Note: The tools ran with fast mod and 31 worker threads. Each reported number is an average of five runs.

Table 7. False-positive and false-negative rates of scTurtle and cTurtle

Set ID	scTurtle (FP only) (%)	BFCounter (FP only) (%)	cTurtle	
			FP (%)	FN (%)
DM	0.178	0.300	1.9×10^{-4}	2.3×10^{-4}
GG	0.848	0.027	0.31	0.08

Note: For the large datasets, because of memory constraints, the exact counts for all *k*-mers could not be obtained, and therefore, these rates could not be computed.

this machine, Jellyfish's count step had a .5% for 19 worker threads. We found KMC to be the fastest tool for the small datasets (DM and GG), but for the two large datasets ZM and HS, Jellyfish and cTurtles, respectively, were the fastest. Jellyfish had the highest memory requirements for all datasets.

To support our claim that the wall-clock time (and therefore parallelization) may be improved by speeding up the producer, we made special versions of scTurtle and cTurtle, which use 31 worker threads and a fast approximate modulus-31 function. For the largest library tested (HS), on average, the special version of scTurtle (counting only) produces frequent 31-mers in ~73 min compared with ~87 min by the regular version (a 19% speedup). As we use 64-bit integers for storing *k*-mers of length 32 and less and 128-bit integers for storing *k*-mers of length in the range 33–64, the memory requirement for larger *k*-mers was also investigated. Again, for the largest dataset tested (HS), we found that scTurtle's memory requirement increased from 109 GB for $0 < k \leq 31$ to 172 for $32 \leq k \leq 64$ (a 58% increase). Note that the Turtles require less memory for up to 64-mers than Jellyfish for 31-mers. Detailed results of all the datasets for the Turtles are presented in Table 6.

We also examined the error rates for our tools and BFCounter. Note that, just like BFCounter, scTurtle has false-positive rates only, and cTurtle has both false-positive and false-negative rates. We investigated these rates for the two small datasets (see Table 7) and found error rates for all tools to be <1%. For the large datasets, because of memory requirements, we could not get the exact counts for all *k*-mers and therefore could not compute these rates.

The error rates also increase if the expected number of frequent *k*-mers is underestimated. As discussed in the introduction, for a genome of size *g*, we expect to observe approximately *g* frequent *k*-mers in the read library. To get an understanding of how the underestimation of *g* drives up the error rates, we tested the DM dataset with expected number of frequent *k*-mers to be *g*, 0.9*g* and 0.85*g* and found the false-positive rates to be 1.87, 1.99 and 2%, respectively. However, the true number of frequent *k*-mers in the DM library is $\approx 1.07g$. Therefore, we recommend setting this parameter to $\approx 1.1g$.

Software versions, commands and parameters used for producing the results presented in this article are provided in Supplementary Materials.

4 CONCLUSION

Identifying correct *k*-mers out of the *k*-mer spectrum of a read library is an important step in many methods in bioinformatics. Usually, this distinction is made by the frequency of the *k*-mers. Fast tools for counting *k*-mer frequencies exist, but for large read libraries, they may demand a significant amount of memory, which can make the problem computationally unsolvable on machines with moderate amounts of memory resource (≤128 GB or even with 256 GB for large datasets). Simple memory-efficient methods, on the other hand, can be time-consuming. Unfortunately, there is no single tool that achieves a reasonable compromise between memory and time. Here we present a set of tools that make some compromises and simultaneously achieve memory and time requirements that are matching the current state-of-the-art in both aspects.

With our first tool (scTurtle), we achieve memory efficiency by filtering *k*-mers of frequency one with a Bloom filter. Our pattern-blocked Bloom filter implementation is more time-efficient compared with a regular Bloom filter. We present a novel strategy based on sorting and compaction for storing frequent *k*-mers and their counts. Because of its sequential memory access pattern, our algorithm is cache-efficient and achieves good running time. However, because of the Bloom filters, we incur a small false-positive rate.

The second tool (cTurtle) is designed to be more memory-efficient at the cost of giving up the frequency values and allowing both false-positive and false-negative rates. The implementation

is based on a counting Bloom filter that keeps track of whether a *k*-mer was observed and whether it has been stored in external media. This tool does not report the frequency count of the *k*-mers.

Both tools allow a *k*-mer size of up to 64. They also allow the user to decide how much memory should be consumed. Of course, there is a minimum memory requirement for each dataset, and the amount of memory directly influences the running time and error rate. However, we believe, with the proper compromises, the approximate frequent *k*-mer extraction problem is now computationally feasible for large read libraries within reasonable wall-clock time using a moderate amount of memory.

ACKNOWLEDGEMENT

The authors thank Mohammad Pavel Mahmud for helpful discussions regarding the run-time analysis and implementation of our method.

Funding: RSR was partially supported by a grant from the National Science Foundation (DEB-1004213) to DB.

Conflict of Interest: none declared.

REFERENCES

Bankevich,A. *et al.* (2012) Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *J. Comput. Biol.*, **19**, 455–477.

Bender,M.A. *et al.* (2005) Cache-oblivious b-trees. *SIAM J. Comput.*, **35**, 341–358.

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

Chikhi,R. and Rizk,G. (2012) Space-efficient and exact de Bruijn graph representation based on a Bloom filter. In: Raphael,B. and Tang,T. (eds) *Algorithms in Bioinformatics, volume 7534 of Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, pp. 236–248.

Chitsaz,H. *et al.* (2011) Efficient *de novo* assembly of single-cell bacterial genomes from short-read data sets. *Nat. Biotechnol.*, **29**, 915–921.

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

Fan,L. *et al.* (2000) Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, **8**, 281–293.

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

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

Levinthal,D. (2008) Performance analysis guide for Intel Core i7 processor and intel xeon 5500 processors.

Liu,Y. *et al.* (2012) Musket: a multistage k-mer spectrum based error corrector for illumina sequence data. *Bioinformatics*, **29**, 308–315.

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

Medvedev,P. *et al.* (2011) Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*, **27**, i137–i141.

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

Miller,J.R. *et al.* (2008) Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, **24**, 2818–2824.

Patterson,D.A. and Hennessey,J.L. (1998) *Computer Organization and Design: the Hardware/Software Interface*. 2nd edn. Morgan Kaufmann Publishers, Inc., San Francisco, CA.

Pell,J. *et al.* (2012) Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proc. Natl Acad. Sci. USA*, [Epub ahead of print, doi:10.1073/pnas.1121464109, June 26, 2012].

Pevzner,P.A. *et al.* (2001) An Eulerian path approach to DNA fragment assembly. *Proc. Natl Acad. Sci. USA*, **98**, 9748–9753.

Putze,F. *et al.* (2010) Cache-, hash-, and space-efficient Bloom filters. *J. Exp. Algorithmics*, **14**, 4:4.4–4:4.18.

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

Salomon,D. (1997) *Data Compression: The Complete Reference*. Springer.

Simpson,J.T. *et al.* (2009) Abyss: a parallel assembler for short read sequence data. *Genome Res.*, **19**, 1117–1123.

Warren,H.S. (2012) *Hackers Delight*. 2nd edn. Addison-Wesley Professional.

Zerbino,D.R. and Birney,E. (2008) Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, **18**, 821–829.