Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
ECCB 2020

OXFORD

Genome

A general near-exact k-mer counting method with low memory consumption enables *de novo* assembly of 106× human sequence data in 2.7 hours

Christina Huan Shi [1] and Kevin Y. Yip [1,2,3,*]

[1]Department of Computer Science and Engineering, [2]Hong Kong Bioinformatics Centre, [3]Hong Kong Institute of Diabetes and Obesity, The Chinese University of Hong Kong, Shatin, New Territories, Hong Kong SAR.

*To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Abstract

Motivation: In *de novo* sequence assembly, a standard pre-processing step is k-mer counting, which computes the number of occurrences of every length-k sub-sequence in the sequencing reads. Sequencing errors can produce many k-mers that do not appear in the genome, leading to the need for an excessive amount of memory during counting. This issue is particularly serious when the genome to be assembled is large, the sequencing depth is high, or when the memory available is limited.

Results: Here we propose a fast near-exact k-mer counting method, CQF-deNoise, which has a module for dynamically removing noisy false k-mers. It automatically determines the suitable time and number of rounds of noise removal according to a user-specified wrong removal rate. We tested CQF-deNoise comprehensively using data generated from a diverse set of genomes with various data properties, and found that the memory consumed was almost constant regardless of the sequencing errors while the noise removal procedure had minimal effects on counting accuracy. Compared with four state-of-the-art k-mer counting methods, CQF-deNoise consistently performed the best in terms of memory usage, consuming 49-76% less memory than the second best method. When counting the k-mers from a human data set with around 60× coverage, the peak memory usage of CQF-deNoise was only 10.9GB (gigabytes) for k=28 and 21.5GB for k=55. *De novo* assembly of 106× human sequencing data using CQF-deNoise for k-mer counting required only 2.7 hours and 90GB peak memory.

Introduction

The high-throughput nature and ever decreasing cost of sequencing technologies have enabled the development of experimental methods for a variety of applications (Goodwin *et al.*, 2016; Reuter *et al.*, 2015). For applications that produce a large number of sequencing reads, directly operating on the reads could be slow and memory-prohibitive when the depth-of-coverage is high. As a result, it has become common to summarize the sequencing data by the list of all k-mers (i.e., length-k sub-sequences) and their occurrence frequencies. These k-mer counts are useful for various downstream tasks, including error correction (Heo *et al.*, 2014; Lim *et al.*, 2014; Luo *et al.*, 2012), de Bruijn graph construction (Jackman *et al.*, 2017; Luo *et al.*, 2012; Souvorov *et al.*, 2018), read clustering and query (Solomon and Kingsford, 2016), and genome size estimation (Li and Waterman, 2003; Li *et al.*, 2010).

In k-mer counting, the major computational concerns include counting and querying efficiency, and memory consumption. These issues are particularly critical when the sequenced genome is large, depth-of-coverage is high, or when there is a limited amount of memory available.

One way to reduce memory consumption while allowing efficient k-mer queries is to use AMQ (Approximate Membership Query) data structures. These structures are for querying whether a particular object (a k-mer in this case) is contained in a set/multi-set. If the set contains the object, the query result is always positive, and thus these structures guarantee no false negatives; If the set does not contain the object, there is

a certain probability that the query result would still be positive, and the rate of such false positives is determined by properties of the data structure.

One widely used AMQ data structure is the Bloom filter, which is a bit vector for recording the objects stored based on a list of hash functions (Bloom, 1970). The Bloom filter can be used for object counting, by having c bits for each data slot to record how many times this slot is set, which forms a counter with a value up to $2^c - 1$ (Fan *et al.*, 2000).

The counting quotient filter (CQF) (Pandey *et al.*, 2017a) is an AMQ data structure for object counting that is more space-efficient than the counting Bloom filter, and it allows dynamic re-sizing of the data structure as more data are added. We explain CQF in detail in Materials and Methods.

A practical issue of k-mer counting is the presence of sequencing errors, which increases the number of unique k-mers by creating false k-mers that do not exist in the original sequences. Since the number of distinct random errors grows with the number of sequencing reads produced, the memory consumption of k-mer counts can increase with sequencing depth even though the number of true k-mers in the original sequences stays constant.

An important property of false k-mers is that they usually have much lower occurrence counts than the true k-mers due to the random nature of most sequencing errors. Therefore, some previous methods simply assume low-frequency k-mers are errors and discard them. For example, BFCounter (Melsted and Pritchard, 2011) and Turtle (Roy *et al.*, 2014) use a Bloom filter to detect whether a k-mer has been encountered before and a separate data structure for the actual counting of k-mers that appear at least twice. In this way, the singletons (k-mers that appear only once in the sequencing reads) will not take up space in the second data structure. Similarly, Jellyfish2 (Marçais and Kingsford, 2011) first uses a counting Bloom filter with a small number of bits per slot to identify k-mers that appear more than a pre-defined number of times, and anther data structure for counting these high-frequency k-mers.

We argue that these approaches to handling false k-mers are not ideal in two aspects. First, false k-mers take up space of the data structures during counting and thus they should be removed as early as possible. Although methods such as BFCounter attempt to minimize the memory occupied by the false k-mers by counting only the high-frequency ones in the second data structure, the first structure for identifying the high-frequency k-mers still contains both the true and false k-mers. Second, using an extra step to identify singletons leads to extra overheads in terms of running time and possibly memory consumption. It would be more preferable to combine counting and false k-mer removal in the same process.

Here we propose the CQF-deNoise method that uses the CQF data structure for counting k-mers while removing false k-mers on the fly, an idea previously explored in the problem of read indexing (Chapuis *et al.*, 2011). Based on a user-specified wrong removal tolerance threshold, CQF-deNoise automatically determines the suitable time and number of rounds of false k-mer removal. As a result, the number of unique k-mers in the CQF remains largely constant during the counting process and is much smaller than the total number of unique true and false k-mers. We show that as compared to several state-of-the-art k-mer counting methods, CQF-deNoise consumes less memory, runs competitively fast, but at the same time gives k-mer counts that are highly accurate. We further develop a *de novo* sequence assembly method using CQF-deNoise for k-mer counting, and show that it uses much less memory or running time as compared to several commonly used methods while achieving similar assembly quality.

Materials and Methods

The counting quotient filter

CQF-deNoise uses CQF for counting k-mers efficiently, with a de-noise method for removing k-mers that likely occur due to sequencing errors. Here we first explain how CQF works and how we implemented it, and then we will describe our de-noise method in the next subsection.

CQF is an AMQ data structure for object counting. It represents a multiset S by using a hash function h to map every object x to a p-bit representation, where $p = \log_2 \frac{n}{\delta}$, n is the expected maximum number of distinct objects in S, and δ is the desired false positive query rate. Unlike the Bloom filter that directly uses all p bits as the signature of an object, in CQF, the first q bits (called the quotient) are used to determine the canonical memory slot (called the "home slot") that an object should be stored in, and the remaining $r = p - q$ bits (called the remainder) are actually stored in the memory slot to indicate that the slot contains an object with that signature. The CQF thus contains a table of 2^q data slots each storing r bits of data. When collision occurs, i.e., when an object's home slot has already been taken by another object, the exact memory slot to be used for storing it is determined by a variant of linear probing, with the objects assigned to adjacent slots following the same order as their hash values. Object insertion, query and deletion are all assisted by some additional metadata that contain information about whether the CQF has stored any object with a particular quotient and where each run of data slots of objects with the same quotient ends.

In the implementation of CQF in Squeakr (Pandey *et al.*, 2017b), object counts are stored in three different ways based on their values. For an object that has appeared only once, no additional information is stored. For an object that has appeared twice, an additional slot is assigned right after the original slot, and it also stores the same remainder value of the object. For an object that has appeared three or more times, right after the original slot assigned to the object, one or more additional slots are assigned as the counter of the object, followed by another slot storing the remainder of the object again to signify the end of the counter. The number of slots assigned to the counter can be dynamically modified, such that CQF can handle object counts of very different magnitudes at the same time. To distinguish between a slot storing a remainder and one storing a counter, the first slot assigned as part of the counter must have a value smaller than the remainder of the object. This is sufficient for indicating that the slot is a counter, because by definition objects in the same run are stored in ascending order of their remainder values. The encoding scheme for counters also employs some additional rules to make sure that all combinations of remainder and counter values can be stored and retrieved correctly (to be explained below).

Our implementation of CQF, with a more space-efficient counter encoding scheme

We adopted the implementation of CQF in Squeakr on the basis of its efficient C++ code and multi-threading option, but we made two important changes. First, since we mainly applied it to DNA sequences, we chose the ntHash function specifically designed for nucleotide sequences, which was shown to perform better than several mainstream hash functions (Mohamadi *et al.*, 2016). Second, we proposed a new, more efficient encoding scheme based on Most-Significant Bit (MSB). The basic idea is to use the most significant bit to indicate whether the current counter still occupies additional slot(s), rather than storing the remainder of the object again after the last counter slot.

Specifically, suppose the occurrence count of an object with remainder x is C, and each remainder occupies r bits. As in the case of the original encoding scheme of CQF, how the occurrence count is stored in CQF-deNoise depends on the values of C and r. If $C = 1$, a single slot is assigned that stores x as its value. If $C > 1$, multiple slots are assigned, with the first slot storing x as its value and the remaining slots storing an encoding of the counter. Since such counter slots are used only when $C > 1$, we store $C - 1$ instead of C. To determine how $C - 1$ is represented, we convert it into binary form and count the number of bits

required. Suppose b bits are needed, then $\lceil \frac{b}{r-1} \rceil$ slots are allocated, and the last $r-1$ bits of each of these slots together store the binary form of $C-1$. After that, for all the counter slots except the last one, the most significant bit among its r bits is set to 1, to indicate that it is not the last slot of this counter. Finally, if the number stored in the first of these counter slots is larger than x, a slot storing the value zero is added at the beginning as an additional escape value.

There are several major differences between the original encoding scheme of CQF and the encoding scheme of CQF-deNoise:

- CQF stores x both before and after the counter slots, while CQF-deNoise only stores it before the counter slots. Instead, every counter slot reserves the most significant bit to indicate whether there are more counter slots to come or not.
- CQF reserves the values 0 and x for special meanings, such that counter values need to be encoded in a way that depends on x. In contrast, counter values in CQF-deNoise can be easily determined by simply ignoring the most significant bit of each counter slot.
- CQF needs to specially handle the case $x = 0$ since it requires putting a value smaller than x in the first counter slot. CQF-deNoise does not need to consider $x = 0$ as a special case, since it only requires putting a value smaller than or equal to x in the first counter slot.

Table 1 compares the two encoding schemes using some examples.

Table 1. Comparing the two encoding schemes. Examples are shown to show how the original CQF and CQF-deNoise encode the occurrence count C of an object with remainder x containing $r = 5$ bits.

x	C	Original CQF encoding	CQF-deNoise encoding
4	1	00100	00100
4	2	00100,00100	00100,00001
4	3	00100,00001,00100	00100,00010
4	6	00100,00000,00101,00100	00100,00000,00101
4	17	00100,00000,10000,00100	00100,00000,10001,00000
4	64	00100,00011,00010,00100	00100,00000,10011,01111
4	128	00100,00000,00110,00111,00100	00100,00000,10111,01111
0	1	00000	00000
0	2	00000,00000	00000,00000,00001
0	3	00000,00000,00000	00000,00000,00010
0	4	00000,00001,00000,00000	00000,00000,00011
0	17	00000,01110,00000,00000	00000,00000,10001,00000
0	64	00000,00010,11110,00000,00000	00000,00000,10011,01111
0	128	00000,00101,00001,00000,00000	00000,00000,10111,01111

From Table 1, we can see that the encoding scheme of CQF-deNoise rarely uses more slots than the scheme of CQF. In fact, by not having the remainder stored twice before and after the counting slots, the encoding scheme of CQF-deNoise usually consumes less space. Table 2 compares the number of slots required by the two encoding schemes for all possible occurrence counts within two practical ranges in genomic applications. In these two settings, the encoding scheme of CQF-deNoise consumes less memory in 61% and 22% of the cases, respectively, while it consumes more memory in only 0% and 3% of the cases.

The de-noise method: overview

The main idea of our de-noise procedure is to identify low-frequency k-mers and remove them from the CQF during the counting process. The objectives are: 1) to remove false k-mers as early as possible, and 2) to avoid wrongly removing true k-mers. Intuitively, false k-mers can be more confidently identified at the late stage of counting, since at that time the true k-mers should have clearly higher counts than the false ones. In contrast, at the early stage of counting, even true k-mers may also have low counts, making them indistinguishable from the false k-mers. There is thus a trade-off between our two objectives. We handle it by taking a user-specified parameter of the tolerable ratio of true k-mers being wrongly removed, to determine the number of rounds of noise removal and the suitable time for performing each round.

Table 2. Space efficiency of the two encoding schemes. The numbers of counter values C within the specified ranges for which the encoding scheme of CQF-deNoise requires one fewer (-1), the same (0) or one more (1) counter slot as compared to the original CQF encoding scheme are shown, considering all possible remainders that contain $r = 8$ bits.

Range of C	Number of CQF-deNoise slots - number of CQF slots					
	-1		0		1	
$[1, 2^8]$	40,255	(61%)	25,279	(39%)	2	(0%)
$[1, 2^{16}]$	2,731,697	(22%)	12,541,391	(75%)	504,128	(3%)

Specifically, CQF-deNoise removes suspected false k-mers m times during the counting process. In each round, all singleton k-mers (i.e., those having an occurrence count of one at that time) are removed as suspected false k-mers. As a result, any k-mer with a total occurrence count larger than m is guaranteed to remain in the CQF at the end of the counting process, although its final count can be smaller than its actual count by a difference up to $m-1$. On the other hand, k-mers with an occurrence count equal to or smaller than m may or may not remain in the CQF at the end, depending on whether it appears exactly once between every two rounds of noise removal.

The number of noise removal rounds, m, is determined as follows. First, define m' as the largest integer such that the fraction of true k-mers with an occurrence count of m' or less is smaller than the user-specified threshold. In other words, m' serves as a conservative estimate of the maximum number of noise removal rounds that can be performed, and it can be estimated based on the genome size, sequencing depth and sequencing error rate as explained below. On the other hand, the size of the CQF depends on the number of noise removal rounds, and different numbers of rounds could lead to the same CQF size. For instance, if m' rounds and $m'-1$ rounds would both lead to the same CQF size, it is better to use $m'-1$ rounds because the rate of wrongly removing true k-mers would be smaller but the memory requirement stays the same. Therefore, m is chosen as the smallest integer such that the CQF size would be the same as the one with m' rounds of noise removal.

The de-noise method: estimating the true-to-false k-mer ratio

Suppose the genome size is G and sequencing reads each of length l have been generated to an average genome-wide depth-of-coverage of d. The total number of k-mers on these reads is $N = \frac{Gd}{l}(l-k+1)$. Suppose that among these k-mers, the ratio of true k-mers that come from the genome to false k-mers that occur due to errors is R, the number of true k-mers will be $\frac{NR}{R+1}$. Finally, if the number of unique true k-mers is u, their average occurrence count will be $\frac{NR}{(R+1)u}$. Accordingly, the occurrence counts of the true k-mers are expected to follow a Poisson distribution with parameter $\lambda = \frac{Gd(l-k+1)R}{l(R+1)u}$, and $m' = \lfloor F^{-1}(w) \rfloor$, where F is the cumulative distribution function of the Poisson distribution and w is the user-specified tolerable ratio of true k-mers being wrongly removed.

In the above formula, l and d are properties of the sequencing data, k and w are user parameters, G is either prior knowledge supplied by the user or estimated using an efficient method such as ntCard (Mohamadi *et al.*, 2017), which provides basic statistics of k-mers but cannot give their exact occurrence counts. The total number of unique true k-mers, u, is not known *a priori*, but an upper bound of it can be used, the value of

which can be obtained by assuming all k-mers in the genome are unique. This leaves us with the last variable, the true-to-false k-mer ratio, R.

Here we describe an algorithm that can compute this ratio based on the error profile of sequencing reads, i.e., the base error rate of each read position. The error profile is platform dependent. For example, for Illumina short reads, the base error rate is highest at the beginning and the end of each read. For simplicity, we assume that whenever an error occurs in a k-mer, the resulting k-mer is always not present in the genome, although in reality a small portion of these error-containing k-mers can actually be found in the genome.

The main difficulty of this calculation is that there are spatial dependencies in two ways. First, if a base error appears at a read position, it turns all k-mers that overlap this position into false k-mers at the same time. Second, if multiple base errors appear at nearby positions, together they create a smaller number of false k-mers (with some false k-mers containing multiple errors) than when they are far apart.

To handle these spatial dependencies, we use a dynamic programming algorithm to compute the probabilities of error positions. Suppose the error profile is given in the form of a length-l vector of base error probabilities e, where l is the length of each sequencing read. We define a two-dimensional table $V(i,j)$ to denote the probability that among the k consecutive positions ending at position i (where i ranges from k to l), the j-th of them is the last position with a base error. For example, if $k=4$, $V(6,2)$ is the probability that among read positions 3, 4, 5 and 6, the last error occurs at position 4 (which is the second position among these positions). We also define $V(i,0)$ as the probability that among the k consecutive positions ending at position i, there is not a base error.

Table V is initialized by considering the first row, $i=k$:

$$V(k,j) = \begin{cases} e_k & \text{if } j=k \\ e_j \prod_{j'=j+1}^{k}(1-e_{j'}) = e_j V(k,j+1)\frac{1-e_{j+1}}{e_{j+1}} & \text{if } j \in [1..k-1] \\ \prod_{j'=1}^{k}(1-e_{j'}) = V(k,1)\frac{1-e_1}{e_1} & \text{if } j=0 \end{cases}$$

The remaining rows of V can be filled in according to the values in the previous row:

$$V(i,j) = \begin{cases} e_i & \text{if } j=k \\ V(i-1,j+1)(1-e_i) & \text{if } j \in [1..k-1] \\ [V(i-1,1)+V(i-1,0)]\,(1-e_i) & \text{if } j=0 \end{cases}$$

Since we assume that a k-mer is a true k-mer if and only if there is not a base error in any of the k positions, the expected number of true k-mers in a read is $\sum_{i=k}^{l} V(i,0)$. Therefore, the true-to-false k-mer ratio is given by $R = \frac{\sum_{i=k}^{l} V(i,0)}{l-k+1-\sum_{i=k}^{l} V(i,0)}$.

The V table contains $(l-k+1)(k+1)$ entries, each requiring a constant amount of time to fill in. Therefore, the time complexity of the algorithm is $O((l-k)k)$, which is also a constant with fixed k and l. The space complexity is $O(k)$, since once a row has been filled in, all entries in the previous row can be discarded.

If the detailed error profile is not available and every read position is assumed to have the same base error rate of e_0, it can be easily proved that the true-to-false k-mer ratio is $R = \frac{(1-e_0)^k}{1-(1-e_0)^k}$.

To test our algorithm, we generated a simulated data set using ART (Huang *et al.*, 2011) assuming the Illumina MiSeq v3 protocol. Using the *F. vesca* reference genome FraVesHawaii_1.0, we generated reads of l=250bp to an average genome-wide depth-of-coverage of $d=50\times$. By aligning the reads to the reference, the average base error rate was found to be 0.4%, and the actual ratio of true k-mers to false k-mers was 8.93 when k was set to 28. When we assumed every position to have the same base error rate of $e_0=0.4\%$, we obtained an estimated true-to-false k-mer ratio of $R=8.41$ based on the above formula. When we instead ran our dynamic programming algorithm using the platform-specific error profile, we obtained an estimated true-to-false k-mer ratio of $R=8.87$, which is closer to the actual value of 8.93.

We also compared the true-to-false k-mer ratios estimated by the two methods using sequencing data from the haploid hydatidiform mole CHM1 (SRA accession: SRR642626), the haploid nature of which ensures that there are no heterozygous variations. We aligned the sequencing reads to a high-quality assembly (GenBank accession: GCA_001297185.2) (Vollger *et al.*, 2019), and obtained the real true-to-false k-mer ratio to be 15.75. When we assumed every position on a sequencing read to have the same error rate, estimated using BBMap (v38.86) (Bushnell, 2020), we obtained an estimated true-to-false k-mer ratio of 6.97. When we allowed each position to have a separate error rate instead, we obtained an estimated ratio of 9.73. These results again show that using a position-specific error profile can lead to a more accurate estimate.

If the error profile involves indels, which are common for some platforms such as PacBio SMRT sequencing, our dynamic programming algorithm can be extended to handle additional insertion and deletion states between read positions. We do not pursue this in the current study.

The de-noise method: time to perform de-noise

With the true-to-false k-mer ratio R estimated, the number of rounds of k-mer removal m can be computed accordingly as explained. The next question is when these m rounds of k-mer removal should be carried out. Based on the variables defined above, the total number of false k-mers is $\frac{N}{R+1}$. To keep the size of the CQF at its minimum, it would be the best to remove these false k-mers evenly across the m rounds of removal. Based on this idea, in the worst case, if all the true k-mers have already been encountered before the target number of false k-mers is reached, the CQF will contain $u+\frac{N}{(R+1)m}$ unique k-mers in total. This value is used as the threshold to trigger the next round of k-mer removal.

Although in the above discussion we have ignored repeat sequences in the genome, as shown in the Results section, our procedure is still effective in removing false k-mers from the resulting sequencing data.

Another limitation of the above derivations is that we have used the probability for a true k-mer to have an occurrence count no more than m to quantify wrong removals, but these true k-mers can actually stay in the CQF as long as it has two new occurrences between any two consecutive rounds of removal. Therefore, the actual wrong removal rate is usually lower than the user-defined tolerance threshold.

Comparing with other k-mer counting methods

We compared the running time and memory usage of CQF-deNoise with four state-of-the-art k-mer counting methods, namely BFCounter (Melsted and Pritchard, 2011), Jellyfish2 (Marçais and Kingsford, 2011), KMC3 (Kokot *et al.*, 2017), and Squeakr (Pandey *et al.*, 2017b). BFCounter, Jellyfish2 and Squeakr were memory-based, while KMC3 was designed to be a disk-based method, although it also provided an in-memory mode, which we used in our comparisons. These five methods were compared using five real sequencing data sets with diverse properties (Table 3). All our tests were run on a machine with Intel(R) Xeon(R) CPUs (E7-4850 v3 @2.20GHz with 112 cores and 35.8MB L3 cache), 504GB RAM and 3TB SSD. All programs were run with 16 threads. Running time was defined as the wall clock time, during which the program loaded and parsed the sequencing data, counted the k-mers, and wrote the results to output files on the disk. Memory consumption was defined as the peak resident set size (RSS). All disk operations were performed in SSD, thus giving advantages to disk-based methods in terms of running time.

All methods were tested for $k=28$ and $k=55$, which are values also used in some previous studies (Kokot *et al.*, 2017; Pandey *et al.*, 2017b). To handle the issue that sequencing reads could come from either strand,

Table 3. Data sets used for testing the performance of CQF-deNoise and comparing it with other methods. Data sets were chosen from four species with very different genome sizes. The sequencing data also had different depths of coverage of the respective genomes.

Data set	Reference genome	Genome size (Mb)	Data sets (SRA accessions)	Total read length (Gb)	Depth-of-coverage
C. elegans	WBcel235	100	SRR7693585-7693591	29.0	290×
F. vesca	FraVesHawaii_1.0	240	SRR072005-072014, 072029, 5275218, 5799056-5799057	14.1	59×
Z. mays	B73 RefGen_v4	2,100	SRR7753852-7753853, 7753855, 7753858-7753861, 7753878-7753884	128.8	61×
H. sapiens (1)	GRCh38.p12	3,257	SRR2831527-2831541	199.6	61×
H. sapiens (2)	GRCh38.p12	3,257	SRR2831454-2831489	343.9	106×

Table 4. Statistics of running CQF-deNoise on the 5 data sets with k=28. The wrong removal rate tolerance thresholds were automatically determined by CQF-deNoise.

Data set	Quotient q	Estimated mean coverage of true k-mers	Wrong removal rate tolerance threshold t	Number of noise removal rounds	Range of t that would lead to the same number of noise removal rounds	Actual wrong removal rate of true k-mers
C. elegans	29	171	8.4×10^{-9}	17	$[1.6\times10^{-51}, 1]$	1.6×10^{-51}
F. vesca	31	29	2.9×10^{-9}	2	$[1.1\times10^{-10}, 7.3\times10^{-4}]$	1.1×10^{-10}
Z. mays	33	60	6.2×10^{-10}	1	$[5.3\times10^{-25}, 6.9\times10^{-6}]$	5.3×10^{-25}
H. sapiens (1)	33	61	3.7×10^{-10}	11	$[4.3\times10^{-15}, 1]$	4.3×10^{-15}
H. sapiens (2)	33	96	3.5×10^{-10}	21	$[2.2\times10^{-20}, 1]$	2.2×10^{-20}

among each k-mer and its reverse complement, we converted the one with a larger hash value to the one with a smaller hash value before counting.

To make the comparisons fair, we ran all programs with the option of removing singleton k-mers chosen if this option was provided, as follows. For BFCounter, it had the ability to remove singleton k-mers by only counting the non-singletons in the second structure. For CQF-deNoise, we set the wrong removal rate tolerance threshold w to the conservative value of 1/genome-size, and let the algorithm determine the number of rounds of noise removal automatically. We found that the number of noise removal rounds remained unchanged for long ranges of this threshold value (Table 4), showing that the counting results would be highly stable for different values used. For Jellyfish2 and KMC3, we selected the mode to save k-mers with an occurrence count at least 2. Squeakr did not provide an option for removing singleton k-mers. Since a brute-force enumeration of all the k-mers and removal of the singletons after counting could be very slow, we did not perform it.

BFCounter and Jellyfish2 required the maximum number of unique k-mers as input, which we computed using ntCard (Mohamadi *et al.*, 2017). CQF-deNoise required the number of true k-mers as an estimation of genome size and the total number of k-mers as inputs, and Squeakr required the number of slots in the CQF as input, which were all estimated based on the output of ntCard. CQF-deNoise also used the results of ntCard to determine the fraction of singleton k-mers for estimating the uniform base error rate e_0 in the comparisons. In the time measurements, the running time of ntCard was also added to the total running time of Squeakr and CQF-deNoise but not BFCounter or Jellyfish2, since the latter two methods could also obtain their required inputs by some faster means.

Squeakr could use x86 bit manipulation instructions to speed up by a factor of 2-4 (Pandey *et al.*, 2017b). Our machine did not support these instructions. Although the resulting running time could not fully reflect the counting speed of Squeakr, CQF-deNoise was also disadvantaged in the same way, because its implementation could use these instructions as well. When running Squeakr, we used the fast ntHash function (Mohamadi *et al.*, 2016) rather than the default Murmur hash function.

Default values were used for all other parameters of the methods.

Additional tools provided in our implementation

We provide a list of additional tools for various CQF operations, including downsizing, intersection, addition, and subtraction, which are useful in different applications. These tools enable logical and arithmetic operations to be directly performed on the occurrence counts efficiently rather than the raw sequencing reads that would require a lot more time.

Downsizing is to resize a CQF that uses p_1 bits of hash values to one that uses p_2 ($p_2 < p_1$) bits. One use of it is to compress the data structure to use less space, especially when the occupancy is low. It also facilitates other operations such as taking the intersection or difference of two CQFs. Downsizing can be efficiently performed by simply iterating through all the objects in the original CQF and inserting the transformed hash values into the new CQF, keeping only the first p_2 bits of its original signature as its new signature. Since objects originally having the same quotient will still have the same quotient after the downsizing, their sort order is maintained, which avoids shifting of the contents, the step usually most time consuming during object insertion.

Intersection is to identify the objects commonly contained in two CQFs and produce two new CQFs that contain only the common objects and their corresponding counts in the original CQFs.

Addition and subtraction respectively compute the sum and difference of the object occurrence counts in two CQFs. In a subtraction, objects with a resulting non-positive count are removed from the output CQF.

De Bruijn graph-based genome assembly using CQF-deNoise for k-mer counting

We implemented a de Bruijn graph (DBG)-based *de novo* genome assembler, SH-assembly, that uses CQF-deNoise to perform k-mer counting. To construct the DBG, the k-mer in the middle of each read is sampled, with a flag added to each slot of the CQF indicating whether a k-mer stored there has already been added to the DBG. If the flag has not been set for the sampled k-mer, all four possible next k-mers will be constructed by deleting the first nucleotide of the original k-mer and appending A, C, G or T to the end. If the CQF returns a count higher than a minimum requirement for one or more of these possible next k-mers, the corresponding new edge(s) will be added to the DBG and the flags of both the original k-mer and these supported next k-mer(s) will be set. This procedure is repeated until all next k-mers have been added to the DBG or no next k-mers can be found from the CQF. This extension procedure is performed in both forward and reverse directions. The resulting DBG is then compacted such that each unitig, defined as a longest unbranched path, is represented by a single node. The compacted DBG is then simplified

using the graph simplification module in Minia (Chikhi and Rizk, 2013) for removing potentially erroneous structures such as tips and bulges. Finally, a list of contigs are produced from the simplified graph.

In our implementation of SH-assembly, k-mer counting and DBG compaction can be performed with multiple threads in parallel.

We compared the running time, peak memory/disk usage and assembly quality of SH-assembly with three commonly used DBG-based assembly methods, namely SOAPdenovo2 (Luo *et al.*, 2012) (v2.04), ABySS 2.0 (Jackman *et al.*, 2017) (v2.2.3), and Minia (Chikhi and Rizk, 2013) (v3 git commit 3eb6f54), using the *C. elegans* and *H. sapiens* (2) data sets (Table 3). In terms of k-mer counting, SOAPdenovo2 uses hash tables, ABySS2.0 uses Bloom filter (also to represent the DBG in order to save space), and Minia 3 builds upon the graph compaction method BCALM 2 (Chikhi *et al.*, 2016) for k-mer counting and unitig constructions.

We compared the assemblers up to the step of contig construction, since assembling the contigs into scaffolds does not require k-mer counting. For SOAPdenovo2, the "pregraph" and "contig" modules were run for contig assembly. ABySS 2.0 could be run with or without using Bloom filter, and we tested both settings. When running it using Bloom filter, we estimated the number of hash functions, H, to $-\log_2 \epsilon$, where ϵ was the false positive rate (FPR) and we set it to 5% according to the recommendation of the authors that it should be no more than 5%. Accordingly, the number of hash functions used was $H = 4$. In addition, ABySS 2.0 had a parameter, B, that defined the total size of two Bloom filters, one for counting k-mers and one for tracing the k-mers in the contigs, with the former eight times the size of the latter according to the user manual. We set the size of the latter to $m = \frac{-n*\ln \epsilon}{(\ln 2)^2}$, where n was the total number of unique k-mers obtained from ntCard. The value of B was then set accordingly to $9m$. Minia 3 was run with maximum disk space of 500GB using the "-max-disk" option and different maximum memory settings, as to be explained in the Results section, using the "-max-memory" option.

All the assemblers were run with k=47 and the minimum occurrence count of usable k-mers set to 2, using 40 threads in single-end mode.

We used Quast (Gurevich *et al.*, 2013) (v5.0.2) with default parameter values to evaluate assembly quality. The reference genomes of *C. elegans* and *H. sapiens* used were WBcel235 and GRCh38.p12, respectively.

Code availability

The source codes of CQF-deNoise and SH-assembly are available at `https://github.com/Christina-hshi/CQF-deNoise.git` and `https://github.com/Christina-hshi/SH-assembly.git`, respectively, both under the BSD 3-Clause license.

Results

CQF-deNoise has a low wrong removal rate of true k-mers

Since CQF-deNoise removes potential false k-mers based on their low occurrence frequencies, it could accidentally remove some true k-mers. We tested how many true k-mers (those that appear in the reference genome) were wrongly removed by CQF-deNoise using the *C. elegans* data set, which had the highest depth of coverage and thus the highest expected false-to-true k-mer ratio in the data caused by the sequencing errors.

When we did not perform noise removal, the frequency distribution of the full set of k-mers was clearly multi-modal (Figure 1a), with a peak at around 160 that likely corresponded to the median coverage of the true k-mers, and another peak at 1 that should contain mostly false k-mers. Since a local minimum was observed at 50, we used it as the demarcation point and manually set the number of noise removal rounds of CQF-deNoise for different values from 0 to 50 to inspect the change of counting results. We set r to 8 based on the expected maximum k-mer occurrence count, and determined the values of p and q according to the estimated number of unique (true and false) k-mers as explained above (Table 5).

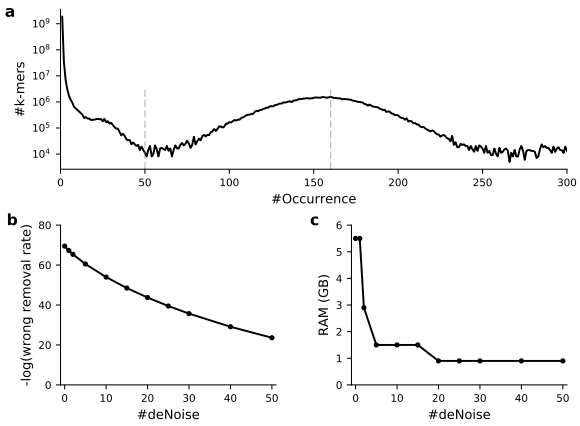

Fig. 1. Trade-off between memory consumption and wrong removal of true k-mers based on the *C. elegans* data set with $k = 28$. **a** Distribution of k-mer occurrence counts without noise removal. **b-c** Estimated wrong removal rate (b) and memory consumption (c) at different numbers of k-mer removal rounds. The estimated wrong removal rate was defined as the fraction of true k-mers having an occurrence count no larger than the number of rounds of noise removal, estimated by the Poisson distribution.

Table 5. Parameter values of CQF in CQF-deNoise in the analysis of wrong removal of true k-mers. Variables m, p, q and r respectively denote the number of k-mer removal rounds, length of hash value signatures, length of the quotient part, and length of the remainder part, as previously defined.

m	p	q	r
0,1	40	32	8
2	39	31	8
5,10,15	38	30	8
20,25,30,40,50	37	29	8

As the number of noise removal rounds increased, as expected more and more true k-mers were wrongly removed, but the rate remained low, reaching the maximum of only $10^{-23.6}$ with 50 rounds of noise removal (Figure 1b). As to be discussed below, in real situations the number of noise removal rounds determined automatically by CQF-deNoise is usually much smaller than 50, and thus the fraction of true k-mers being removed is very small in practice.

At the same time, the memory consumption rapidly dropped from 5.5GB to 0.9GB as the number of noise removal rounds increased from 0 to 20, which was then stabilized thereafter (Figure 1c). To interpret this memory consumption objectively, we performed the following conceptual analysis. Assuming that most true k-mers in the genome are unique and there is a uniform read coverage of the whole genome in the data set, there would be around 100 million unique true k-mers in the C. elegans genome, each with an occurrence count of 160. Since two counting slots are required to store this average occurrence count when the remaining parameter, r, is equal to 8 in addition to a slot storing the remainder (Materials and Methods; Table 5), the total number of slots in the CQF would be at least 300 millions. Further, since the number of slots in the CQF has to be a power of 2, the smallest number of slots is 2^{29}, which was exactly the number of slots in the CQF we constructed, showing that our memory usage of 0.9GB was optimal in this case.

CQF-deNoise has high counting accuracy

In addition to wrong removal of true k-mers, having identical hash values for different k-mers can also introduce errors to k-mer counts. We evaluated the counting accuracy of CQF-deNoise using the *C. elegans* data set in two ways, again by setting the number of rounds of k-mer removal manually. First, since singleton k-mers are mostly false k-mers, we evaluated the proportion of singleton k-mers that remained in the CQF (the "singleton survival rate") and the proportion of non-singleton k-mers that did not remain in the CQF (the "non-singleton removal rate"). Singletons can survive due to the intrinsic nature of CQF that a low-occurrence k-mer may share the same hash value with other k-mers, and thus their counts add up. A non-singleton can be removed if it occurs no more than once between every two consecutive rounds of removal. From the results, the singleton survival rate remained low across the different numbers of noise removal rounds, with only around 0.1%-0.2% of the singletons remained in the CQF at the end of counting (Figure 2a). For the non-singletons, less than 2.5% of them were removed (Figure 2b), and many of these non-singletons are expected to be false k-mers based on the shape of the multi-modal k-mer count distribution (Figure 1a).

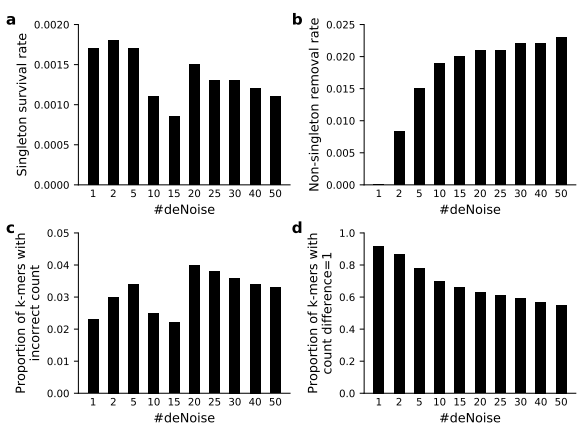

Fig. 2. Counting accuracy based on the *C. elegans* data set with different numbers of rounds of noise removal. **a** Singleton survival rate. **b** Non-singleton removal rate. **c** Proportion of k-mers in CQF with an incorrect count. **d** Among all the k-mers in CQF with an incorrect count, the proportion of k-mers with count difference being 1.

Second, for the k-mers that remained in the CQF at the end of counting, we evaluated the correctness of their counts. We observed that only 2%-4% of these k-mers had an incorrect count (Figure 2c), and among them, more than half had a difference of only 1 between the actual count and the count in the CQF (Figure 2d), showing that the noise removal procedure of CQF-deNoise had minimal effects on counting accuracy.

CQF-deNoise uses less memory than other k-mer counting methods

To benchmark the computational performance of CQF-deNoise, we compared it with the four other k-mer counting methods. We first compared the memory consumption of the different methods based on the five data sets. From the results (Figure 3a,b), CQF-deNoise consistently consumed the smallest amount of memory on all data sets for both values of k tested. Compared to the next method with the lowest memory consumption, CQF-deNoise had a memory usage reduction from 49% (*F. vesca* data set, $k = 55$) to 76% (*C. elegans* data set, $k = 28$).

The running time of CQF-deNoise was similar to that of Jellyfish2 and Squeakr and was consistently faster than BFCounter (Figure 3c,d). It was

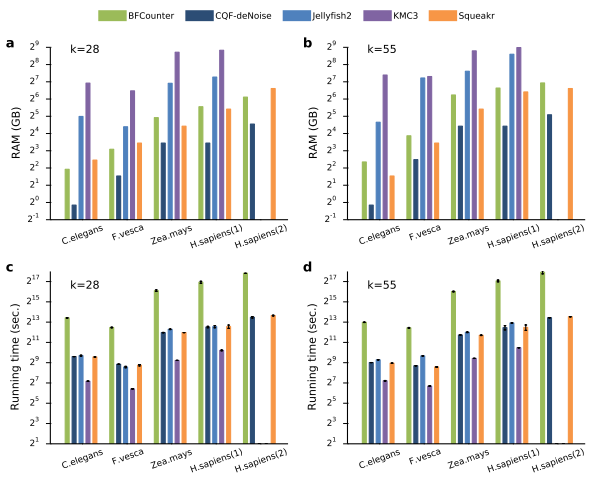

Fig. 3. Comparison of the k-mer counting methods. **a**-**b** Memory usage when k=28 (a) and k=55 (b). **c**-**d** Running time when k=28 (c) and k=55 (d). Both Jellyfish2 and KMC3 failed to complete counting the k-mers in the *H. sapiens* (2) data set.

not faster than KMC3, but the short counting time of KMC3 came with the cost of a much higher memory consumption, which was 21 (*Z. mays* data set, $k = 55$) to 187 (*C. elegans* data set, $k = 55$) times of the memory consumption of CQF-deNoise.

To evaluate the scalability of the methods, we sub-sampled reads of the *C. elegans* data set at different depths to form multiple sub-data sets. The corresponding k-mer counting results show that CQF-deNoise used the least amount of memory at all depth values (Figure 4a). Importantly, the increase in memory consumption with respect to sequencing depth was smallest for CQF-deNoise, since it was largely unaffected by the additional false k-mers. In terms of running time, the order of the different methods remained unchanged at the different depths of coverage (Figure 4b).

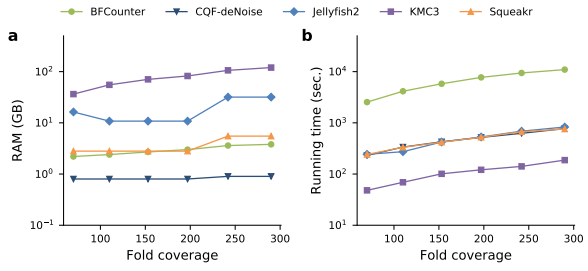

Fig. 4. Scalability of the k-mer counting methods. **a**-**b** Memory consumption (a) and running time (b) of the different methods on random subsets of reads from the *C. elegans* data set at 70× to 290× coverage.

Taken together, these results show that CQF-deNoise performs k-mer counting with lower memory consumption while running as fast as the other memory-based methods.

K-mer count querying using CQF-deNoise is efficient

Another important performance indicator is the time needed to query the occurrence counts of k-mers after the counting process, when all the counts are already loaded into memory. We compared the different methods using the *C. elegans* and *F. vesca* data sets only due to the excessive amount of time needed by some of the published methods when querying from the larger data sets. For each of the two data sets, we queried both k-mers that

existed in the sequencing reads and were contained in all the counting data structures as well as k-mers that did not exist in the reads.

The results (Table 6) show that CQF-deNoise completed these queries using the smallest amount of time among all the methods for both data sets and for both types of k-mers. The high query efficiency of CQF-deNoise was likely due to the compactness of its data structure.

Table 6. Query performance of the different methods. For each species, the running time of querying a) random k-mers that existed in the sequencing reads and were contained in the data structures ("Exist") and b) random k-mers that did not exist in the sequencing reads ("Not exist") are reported. Each query set contained approximately 100 million and 550 million k-mers in the case of the *C. elegans* data set and *F. vesca* data set, respectively.

Running time	*C. elegans*		*F. vesca*	
(sec.)	Exist	Not exist	Exist	Not exist
BFCounter	70	51	353	294
CQF-deNoise	47	40	210	197
Jellyfish2	153	156	620	778
KMC3	95	160	429	1299
Squeakr	55	50	269	254

Fast k-mer counting by CQF-deNoise enables accurate and efficient genome assembly

We performed *de novo* genome assemblies of the sequencing reads in the two data sets with the highest depth of coverage, *C. elegans* (290×) and *H. sapiens* (2) (106×), using SH-assembly and the other three assemblers.

For the *C. elegans* assembly, SH-assembly was the most efficient method in terms of both memory usage and running time (Table 7). It used 2.6GB peak memory, which was 19% of the memory usage of the second best memory-based method (ABySS 2.0 with Bloom filter, 13.6GB). As for the disk-based method Minia 3, by default its maximum memory setting ("-max-mem") was set to 5GB, and in this assembly it used 6.6GB peak memory, which was 2.5 times that of SH-assembly, together with 28.9GB of disk space. In terms of running time, SH-assembly used 7.8 minutes, which was only 11% of the other two memory-based methods (SOAPdenovo2, 70.6 minutes; ABySS 2.0 with Bloom filter, 70.2 minutes) and 41% of the default setting of Minia 3. To see whether the running time of Minia 3 could be reduced by allowing it to use more memory, we changed its maximum memory setting to 500GB. The resulting running time was only slightly reduced (from 18.9 minutes to 16.8 minutes), at the expense of the use of a lot more memory (123.3GB). We note that disk operations were already sped up by the use of SSD in our tests.

Regarding assembly quality, SH-assembly was consistently one of the best methods, in terms of NG50, NGA50, total lengths of contigs and aligned contigs, and genome coverage of aligned contigs. Since SH-assembly uses some functions of Minia 3 to perform assembly, the numbers of misassemblies of the two methods (as evaluated by QUAST (Gurevich *et al.*, 2013)) were similar. However, the two methods did have some differences at the unitig level. Specifically, Minia 3 produced more unitigs than SH-assembly (6.4 millions and 3.5 millions, respectively), but a larger proportion of the unitigs produced by Minia 3 were short (only 15,075 of them were 500bp or longer, as compared to 20,450 of them produced by SH-assembly). This was likely due to false k-mers caused by sequencing errors, many of which were removed by CQF-deNoise in SH-assembly.

For the *H. sapiens* assembly, both the memory-based methods ABySS 2.0 without Bloom filter and SOAPdenovo2 could not finish within 72 hours (Table 8). SH-assembly finished the assembly in 2.7 hours, which was 8% of the time needed by ABySS 2.0 with Bloom filter. SH-assembly used 90.0GB peak memory, which was substantially lower than ABySS 2.0 (with Bloom filter, 227.3GB) and SOAPdenovo2 before it was terminated (451.5GB). As for Minia 3, when its maximum memory was set to 5GB and 500GB, it finished the assembly in 9.1 and 6.9 hours, respectively, corresponding to 337% and 256% of the running time of-SH assembly. The memory usage of Minia 3 was low in its default setting (23.7GB), but it used quite a lot of disk space (222.3GB), which could have incurred an overhead to the running time if disk access was slow.

The assembly produced by SH-assembly was the best in terms of NG50, NGA50, total contig length, total contig aligned and genome coverage, closely followed by Minia 3.

Overall, the assembly results produced by SH-assembly were competitive to the other three methods, while it consistently used the least amount of time and usually the least amount of memory.

Discussion and conclusion

In this study, we have proposed a memory-efficient k-mer counting method, CQF-deNoise, that uses less memory than other state-of-the-art k-mer counting methods but runs as fast as the memory-based methods. Its low memory consumption is due to a procedure that removes potential false k-mers caused by sequencing errors. When compared to the next method with the lowest memory consumption, CQF-deNoise used 49%-76% less memory when tested across different data sets and k-mer sizes.

The false k-mer removal procedure of CQF-deNoise automatically determines the time and number of rounds of removal based on a user-specified maximum tolerable rate of wrongly removing true k-mers. We have shown that it was effective in removing false k-mers, with the counting accuracy of true k-mers only minimally affected. Although some other k-mer counting methods can also remove potential false k-mers, they usually have one structure that stores all true and false k-mers, causing an increase of memory usage as the sequencing depth (and number of false k-mers) increases. In comparison, the dynamic removal procedure of CQF-deNoise leads to virtually constant memory usage at different sequencing depths.

We have also developed a *de novo* genome assembly, SH-assembly, which uses CQF-deNoise for fast k-mer counting. Comparing with three other commonly used methods, it required less running time and usually less memory without affecting assembly quality. When assembling the contigs of a human genome based on a data set with 106× average genome coverage, SH-assembly finished in 2.7 hours using only 90GB peak memory. The low computational requirements of SH-assembly make it a good candidate assembler for low-cost and portable sequencers.

We provide CQF-deNoise and SH-assembly as open-source packages. The main program of CQF-deNoise contains additional options for users who want to have more control of the counting process, such as specifying the exact number of noise removal rounds to be performed. The package also comes with a number of extra tools for manipulating CQFs in general.

One limitation of CQF-deNoise is that some k-mers that remain in the CQF can actually have a lower occurrence count than some k-mers completely removed from the CQF. This is because whether a k-mer would be removed depends not only on its occurrence count, but also on the timing of its different occurrences. A k-mer remains if and only if there are two or more occurrences of it within the time periods between any two consecutive noise removal rounds. This would not affect the removal of singleton k-mers, but by chance some false k-mers do occur more than once and are not removed in this way. When it is crucial to have as few false k-mers remaining in the CQF as possible, one possible remedy is to construct a histogram of k-mer occurrence frequencies after counting, use it to determine occurrence counts that likely belong to the false k-mers by inspecting the distribution (as was done in Figure 1), and finally perform a post-processing round of noise removal using the identified threshold.

Table 7. Comparing the assemblers based on the *C. elegans* data set. Disk usage is reported only for the disk-based method Minia 3. Statistics related to contig lengths and alignments involve only contigs of 500bp or longer.

Assembly	Memory usage (GB)	Disk usage (GB)	Running time (min.)	NG50 (bp)	NGA50 (bp)	Total contig length (bp)	Total aligned contigs (bp)	Genome coverage (%)	Misassemblies per Mb
ABySS 2.0 (no Bloom filter)	22.6	-	346.9	3,117	2,859	92,817,982	88,153,905	87.2	0.09
ABySS 2.0 (with Bloom filter)	13.6	-	70.2	3,069	2,845	92,529,312	88,037,316	87.1	0.09
Minia 3 ("-max-mem 5G")	6.6	28.9	18.9	9,437	8,157	94,562,769	89,665,796	89.2	0.50
Minia 3 ("-max-mem 500G")	123.3	29.2	16.8	9,437	8,151	94,556,501	89,659,574	89.2	0.50
SH-assembly	2.6	-	7.8	9,329	8,251	94,660,023	89,794,709	89.3	0.49
SOAPdenovo2	65.3	-	70.6	2,133	2,017	88,897,024	85,644,561	84.8	0.13

Table 8. Comparing the assemblers based on the *H. sapiens* (2) data set. Disk usage is reported only for the disk-based method Minia 3. Statistics related to contig lengths and alignments involve only contigs of 500bp or longer. ABySS 2.0 without using Bloom filter and SOAPdenovo 2 both could not finish the assembly within 72 hours.

Assembly	Memory usage (GB)	Disk usage (GB)	Running time (hr.)	NG50 (bp)	NGA50 (bp)	Total contig length (bp)	Total aligned contigs (bp)	Genome coverage (%)	Misassemblies per Mb
ABySS 2.0 (no Bloom filter)	$\geq$518.2	-	>72.0	N/A	N/A	N/A	N/A	N/A	N/A
ABySS 2.0 (with Bloom Filter)	227.3	-	32.0	1,521	1,518	2,194,175,948	2,192,295,093	71.6	0.09
Minia 3 ("-max-mem 5G")	23.7	222.3	9.1	1,984	1,977	2,278,869,717	2,275,829,647	74.4	0.97
Minia 3 ("-max-mem 500G")	352.6	222.9	6.9	1,984	1,977	2,278,794,715	2,275,760,368	74.4	0.98
SH-assembly	90.0	-	2.7	2,048	2,041	2,284,444,349	2,281,280,705	74.6	1.18
SOAPdenovo2	$\geq$451.5	-	>72.0	N/A	N/A	N/A	N/A	N/A	N/A

Acknowledgements

We thank members in the Yip Lab for helpful discussions. KYY was partially supported by the CUHK Young Researcher Award and Outstanding Fellowship, the Hong Kong Research Grants Council General Research Funds 14145916, 14170217, Collaborative Research Funds C4054-16G, C4045-18WF, C4057-18EF and Theme-based Research Scheme T12C-714/14-R. This work was also supported by the Hong Kong Epigenomics Project (EpiHK).

References

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, **13**(7), 422–426.

Bushnell, B. (2020). Bbmap: Short read aligner for dna and rna-seq data. [Online; accessed 8-July-2020].

Chapuis, G. *et al.* (2011). Parallel and memory-efficient reads indexing for genome assembly. In *PPAM 2011: Parallel Processing and Applied Mathematics*, pages 272–280.

Chikhi, R. and Rizk, G. (2013). Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, **8**(1), 22.

Chikhi, R. *et al.* (2016). Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, **32**(12), i201–i208.

Fan, L. *et al.* (2000). Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, **8**(3), 281–293.

Goodwin, S. *et al.* (2016). Coming of age: Ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, **17**(6), 333–351.

Gurevich, A. *et al.* (2013). QUAST: Quality assessment tool for genome assemblies. *Bioinformatics*, **29**(8), 1072–1075.

Heo, Y. *et al.* (2014). BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, **30**(10), 1354–1362.

Huang, W. *et al.* (2011). ART: A next-generation sequencing read simulator. *Bioinformatics*, **28**(4), 593–594.

Jackman, S. D. *et al.* (2017). ABySS 2.0: Resource-efficient assembly of large genomes using a bloom filter. *Genome Research*, **27**(5), 768–777.

Kokot, M. *et al.* (2017). KMC 3: Counting and manipulating k-mer statistics. *Bioinformatics*, **33**(17), 2759–2761.

Li, R. *et al.* (2010). The sequence and de novo assembly of the giant panda genome. *Nature*, **463**(7279), 311–317.

Li, X. and Waterman, M. S. (2003). Estimating the repeat structure and length of DNA sequences using l-tuples. *Genome Reearch*, **13**(8), 1916–1922.

Lim, E.-C. *et al.* (2014). Trowel: A fast and accurate error correction module for illumina sequencing reads. *Bioinformatics*, **30**(22), 3264–3265.

Luo, R. *et al.* (2012). SOAPdenovo2: An empirically improved memory-efficient short-read de novo assembler. *Gigascience*, **1**(1), 18.

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

Melsted, P. and Pritchard, J. K. (2011). Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, **12**(1), 333.

Mohamadi, H. *et al.* (2016). ntHash: Recursive nucleotide hashing. *Bioinformatics*, **32**(22), 3492–3494.

Mohamadi, H. *et al.* (2017). ntCard: A streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, **33**(9), 1324–1330.

Pandey, P. *et al.* (2017a). A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 775–787.

Pandey, P. *et al.* (2017b). Squeakr: An exact and approximate k-mer counting system. *Bioinformatics*, **34**(4), 568–575.

Reuter, J. A. *et al.* (2015). High-throughput sequencing technologies. *Molecular Cell*, **58**(4), 586–597.

Roy, R. S. *et al.* (2014). Turtle: Identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, **30**(14), 1950–1957.

Solomon, B. and Kingsford, C. (2016). Fast search of thousands of short-read sequencing experiments. *Nature Biotechnology*, **34**(3), 300–302.

Souvorov, A. *et al.* (2018). SKESA: Strategic k-mer extension for scrupulous assemblies. *Genome Biology*, **19**(1), 153.

Vollger, M. R. *et al.* (2019). Long-read sequence and assembly of segmental duplications. *Nature Methods*, **16**(1), 88–94.