Lighter: fast and memory-efficient error correction without counting

Li Song[1] Liliana Florea[2] Ben Langmead*[3]

May 3, 2014

Abstract

Lighter is a fast and memory-efficient tool for correcting sequencing errors in high-throughput sequencing datasets. Lighter avoids counting k-mers in the sequencing reads. Instead, it uses a pair of Bloom filters, one populated with a sample of the input k-mers and the other populated with k-mers likely to be correct based on a simple test. As long as the sampling fraction is adjusted in inverse proportion to the dataset's average coverage, the Bloom filter size can be held constant while maintaining near-constant accuracy. Lighter is easily applied to very large sequencing datasets. It is parallelized, uses no secondary storage, and is both faster and more memory-efficient than competing approaches while achieving comparable accuracy. Lighter is free open source software available from https://github.com/mourisl/Lighter/.

Introduction

The cost and throughput of DNA sequencing have improved rapidly in the past several years [6], with recent advances reducing the cost of sequencing a single human genome at 30-fold coverage to around \$1,000 [7]. With these advances has come an explosion of new software for analyzing large sequencing datasets. Sequencing error correction is a basic need for many of these tools. Removing errors at the outset of an analysis can improve accuracy of downstream tools such as variant callers [12]. Removing errors can also improve the speed and memory-efficiency of downstream tools, particularly for analyses involving de novo assembly with De Bruijn graphs since the graph's size increases with the number of distinct k-mers in the dataset [3, 18].

To be useful in practice, error correction software must make economical use of time and memory even when input datasets are extremely large (billions of reads) and when the genome under study is also large (billions of nucleotides). Several methods have been proposed, covering a wide tradeoff space between accuracy, speed and memory- and storage-efficiency. SHREC [20] and HiTEC [9] build a suffix index of the input reads and locate errors by finding instances where a substring is followed by a character less often than expected. Coral [19] and ECHO [11] find overlaps among reads and use the resulting multiple alignments to detect and correct errors. Reptile [23] and Hammer [15] detect and correct errors by examining each k-mer's neighborhood in the dataset's k-mer Hamming graph.

The most practical and widely used error correction methods descend from the spectral alignment approach introduced in the earliest De Bruijn graph based assemblers [3, 18]. These methods count the number of times each k-mer occurs (its *multiplicity*) in the input reads, then apply a threshold such that reads with multiplicity exceeding the threshold are considered *solid*. Solid k-mers are unlikely to have been altered by sequencing errors. k-mers with low multiplicity (*weak* k-mers) are systematically edited into solid k-mers using a dynamic-programming solution to the spectral alignment problem [3, 18] or, more often, a fast heuristic approximation. Quake [12], the most widely used error correction tool, uses a hash-based k-mer counter called Jellyfish [14] to determine which k-mers are solid. CUDA-EC [21] was the first to use a Bloom filter as a space-efficient alternative to hash tables for counting k-mers and for representing the set of solid k-mers. More recent tools such as Musket [13] and BLESS [8] use a combination of Bloom filters and hash tables to count k-mers or to represent the set of solid k-mers.

Lighter (LIGHTweight ERror corrector) is also in the family of spectral alignment methods, but differs from previous approaches in that it avoids counting k-mers. Rather than count k-mers, Lighter samples k-mers randomly, storing the sample in a Bloom filter. Lighter then uses a statistical test applied to each position of each read to compile a set of solid k-mers, stored in a second Bloom filter. These two Bloom filters are the only sizable data structures used by Lighter.

Lighter's crucial advantage is that its parameters can be set such that the memory footprint and the accuracy of the approach are near-constant with respect to coverage. That is, no matter how deep the coverage, Lighter can allocate the same sized Bloom filters and achieve nearly the same (a) Bloom filter occupancy, (b) Bloom filter false positive rate, and (c) error correction accuracy. Lighter does this without using any disk space or other secondary memory. This is in contrast to BLESS and Quake/Jellyfish, which use secondary memory to store some or all of the k-mer counts. Lighter's accuracy is comparable to competing tools, as we show both in simulation experiments where false positives and false negatives can be measured, and in real-world experiments where read alignment scores and assembly statistics can be measured. Lighter is also very simple and fast, faster than all other tools tried in our experiments. These advantages make Lighter quite practical compared to previous counting-based approaches, all of which require an amount of memory or secondary storage that increases with depth of coverage.

Method

Lighter's workflow is illustrated in Figure 1. Lighter makes three passes over the input reads. The first pass obtains a sample of the k-mers present in the input reads, storing the sample in Bloom filter A. The second pass uses Bloom filter A to identify solid k-mers, which it stores in Bloom filter B. The third pass uses Bloom filter B and a greedy procedure to correct errors in the input reads.

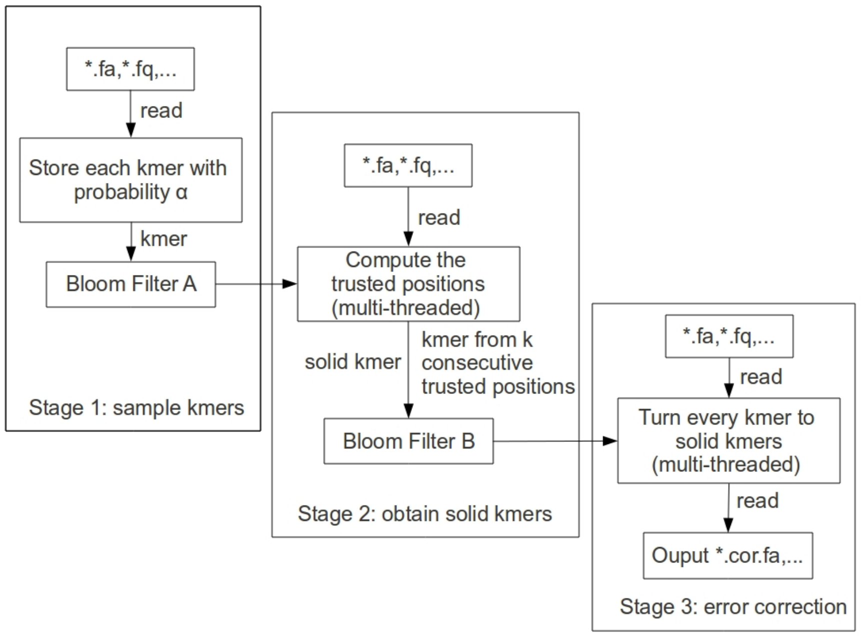
.fa,.fq,...
read
Store each kmer with probability α
kmer
Bloom Filter A
Stage 1: sample kmers
.fa,.fq,...
read
Compute the trusted positions (multi-threaded)
solid kmer
kmer from k consecutive trusted positions
Bloom Filter B
Stage 2: obtain solid kmers
.fa,.fq,...
read
Turn every kmer to solid kmers (multi-threaded)
read
Ouput *.cor.fa,...
Stage 3: error correction

Figure 1: The framework of Lighter

Bloom filter

A Bloom filter [1] is a compact probabilistic data structure representing a set. It consists of an array of m bits, initialized to 0. To add an object o, h independent hash functions $H_0(o), H_1(o), ..., H_{h-1}(o)$ are applied. Each maps o to an integer in $[0, m)$ and the corresponding h array bits are set to 1. To test if object q is a member, the same hash functions are applied to q. q is a member if all corresponding bits are set to 1. A false positive occurs when the corresponding bits are set to 1 "by coincidence," that is, because of objects besides q that were added previously. Assuming the hash functions map objects to bit array elements with equal probability, the Bloom filter's false positive rate is approximately $(1 - e^{-h\frac{n}{m}})^h$, where n is the number of distinct objects added, which we call the *cardinality*. Given n, which is usually determined by the dataset, m and h can be adjusted to achieve a desired false positive rate. Lower false positive rates can come at a cost, since greater values of m require more memory and greater values of k require more hash function calculations. Many variations on Bloom filters have been proposed that additionally permit compression of the filter, storage of count data, representation of maps in addition to sets, etc [22]. Bloom filters and variants thereon have been applied in various bioinformatics settings, including assembly [17], compression [10], k-mer counting [16], and error correction [21].

By way of contrast, another way to represent a set is with a hash table. Hash tables do not yield false positives, but Bloom filters are far smaller. Whereas a Bloom filter is an array of bits, a hash table is an array of buckets, each large enough to store a pointer, key, or both. If chaining is used, lists associated with buckets incur additional overhead. While the Bloom filter's small size comes at the expense of false positives, these can be tolerated in many settings including in error correction.

As Figure 1 shows, the efficiency of bloom filter affects the running time of Lighter a lot. For a standard Bloom filter, the h hash functions may map o to any places in the bit array. The bit array is usually very large, so all the h accesses will likely to cause cache miss. In [cite], the authors proposed blocked Bloom filter which can decrease the number of cache misses. Given a block size b, it will use $H_0(o)$ to decide the start position on the bit array, and then map $H_0(o), H_1(o), ..., H_{h-1}(o)$ onto the block starting from that position. It choose b about the size of a cache line, then the h accesses will not cause h cache misses. The drawback of using blocked Bloom filter is that we have to use a bit larger h and m to have the same FPR when using the standard bloom filter. To estimate the FPR of blocked Bloom filter, we can consider each of the possible $m - b + 1$ blocks. for the i-th block, the FPR within this block is $(b'_i/b)^h$, where b'_i is the number of bits set to 1 in block i. So the overall FPR is $\dfrac{\sum_i (b'_i/b)^h}{m - b + 1}$.

In our method, the objects to be stored in the Bloom filters are k-mers. Because we would like to treat genome strands equivalently for counting purposes, we will always *canonicalize* a k-mer before adding it to, or using it to query a Bloom filter. A canonicalized k-mer is either the k-mer itself or its reverse complement, whichever is lexicographically prior.

Sequencing model

We use a simple model for the sequencing process (including errors) and for Lighter's subsampling. The sequencing model resembles one suggested previously [**?**]. Let K be the total number of k-mers obtained by the sequencer. We say a k-mer is *incorrect* if its sequence has been altered by one or more sequencing errors. Otherwise it is *correct*. Let ϵ be the fraction of k-mers that are incorrect. We assume ϵ does not vary with the depth of sequencing. The sequencer obtains correct k-mers by sampling independently and uniformly from k-mers in the genome. Let the number of k-mers in the genome be G, and assume all are distinct. If κ_c is a random variable for the multiplicity of a correct k-mer in the input, κ_c is binomial with success probability $1/G$ and number of trials $(1-\epsilon)K$: $\kappa_c \sim Binom((1-\epsilon)K, 1/G)$. Since the number of trials is large and the success probability is small, the binomial is well approximated by a Poisson: $\kappa_c \sim Pois(K(1-\epsilon)/G)$

A sequenced k-mer survives subsampling with probability α. If κ'_c is a random variable for the number of times a correct k-mer appears in the subsample, $\kappa'_c \sim Binom((1-\epsilon)K, \alpha/G)$, which is approximately $Pois(\alpha K(1-\epsilon)/G)$.

The model for incorrect k-mers is similar. The sequencer obtains incorrect k-mers by sampling independently and uniformly from k-mers "close to" a k-mer in the genome. We might define these as the set of all k-mers with low Hamming distance from some genomic k-mer. If κ_e is a random variable for the multiplicity of an incorrect k-mer, κ_e is binomial with success probability $1/H$ and number of trials ϵK: $\kappa_e \sim Binom(\epsilon K, 1/H)$, which is approximately $Pois(K\epsilon/H)$. It is safe to assume $H \gg G$, but we need not specify H precisely here. $\kappa_e' \sim Pois(\alpha K \epsilon/H)$ is a random variable for the number of times an incorrect k-mer appears in the subsample.

Others have noted that, given a dataset with deep and uniform coverage, incorrect k-mers occur rarely while correct k-mers occur many times, proportionally to coverage [3, 18].

Stages of the method

First pass. In the first pass, Lighter examines each k-mer of each read. With probability $1-\alpha$, the k-mer is ignored. Otherwise, it is canonicalized and added to Bloom filter A.

Say a distinct k-mer K occurs a total of N_K times in the dataset. If subsampling discards all N_K occurrences, the k-mer is never added to A and A's cardinality is reduced by one. Thus, reducing α can in turn reduce A's cardinality. Because correct k-mers are more numerous, incorrect k-mers tend to be discarded from A before correct k-mers as α decreases. Later we show that if K increases but αK is held constant, the cardinality of A and the fraction of correct k-mers in A both remain nearly constant.

The subsampling fraction α is set by the user. We suggest adjusting α in inverse proportion to depth of sequencing. For our experiments, we set $\alpha = 0.05$ when the average coverage is 70-fold. That is, we set α to $0.05\frac{70}{C}$ where C is fold coverage.

Second pass. A read position is overlapped by up to x k-mers, $1 \le x \le k$, where x depends on how close the position is to either end of the read. For a position altered by sequencing error, the overlapping k-mers are all incorrect and are unlikely to appear in A. We apply a threshold such that if the number of k-mers overlapping the position and appearing in Bloom filter A is less than the threshold, we say the position is *untrusted*. Otherwise we say it is *trusted*. Each instance where the threshold is applied is called a *test case*. When one or more of the x k-mers involved in two test cases differ, we say the test cases are distinct.

Let $P^*(\alpha)$ be the probability an incorrect k-mer appears in A, taking the Bloom filter's false positive rate into account. If random variable $B_{0,x}$ represents the number of k-mers appearing in A for an untrusted position overlapped by x k-mers, $B_{e,x} \sim Binom(x, P^*(\alpha))$. We define thresholds y_x, for each possible value of x in $[1, k]$. Each y_x is the minimum integer such that $p(B_{0,x} \le y_x - 1) \ge 0.995$.

Initially we ignore false positives and model the probability of a sequenced a k-mer having been added to A as $P(\alpha) = 1 - (1-\alpha)^{f(\alpha)}$. We define $f(\alpha) = max\{2, 0.1/\alpha\}$. That is, we assume the multiplicity of a weak k-mer is at most $f(\alpha)$, which will often be a conservative assumption, especially for small α. It is also possible to define $P(\alpha)$ in terms of random variables κ_e and κ_e', but we avoid this for simplicity.

A property of this threshold is that when α is small, $P(\alpha/z) = 1-(1-\alpha/z)^{0.1z/\alpha} \approx 1-(1-\alpha)^{0.1/\alpha} = P(\alpha)$, where z is a constant greater than 1 and we use the fact that $(1-\alpha/z)^z \approx 1-\alpha$.

For $P^*(\alpha)$, we additionally take A's false positive rate into account. If the false positive rate is β, then $P^*(\alpha) = P(\alpha) + \beta - \beta P(\alpha)$.

Once all positions in a read have been marked *trusted* or *untrusted* using the threshold, we find all instances where k trusted positions appear consecutively. The k-mer made up by those positions is added to Bloom filter B.

Third pass. In the third pass, Lighter applies a simple, greedy error correction procedure similar to that used in BLESS [8]. A read r of length $|r|$, contains $|r| - k + 1$ k-mers. k_i denotes the k-mer starting at read position i, $1 \le i \le |r| - k + 1$. We first identify the longest stretch of consecutive k-mers in the read that appear in Bloom filter B. Let k_b and k_e be the k-mers at the left and right extremes of the stretch. If $e < |r| - k + 1$, we examine successive k-mers to the right starting at $k_e + 1$. For a k-mer k_i that does not appear in B, we assume the nucleotide at offset $i + k - 1$ is incorrect. We consider all possible ways of substituting for the incorrect nucleotide. We try each possible substitution, then count how many consecutive k-mers starting with k_i appear in Bloom filter B. We choose the substitution that creates the longest stretch of consecutive k-mers in B. If more than one candidate substitution is equally good, we call position $i + k - 1$ ambiguous and resume the rightward scan at k-mer $k_i + k$. The procedure is illustrated in Figure 2. If k_b is not the leftmost k-mer in the read, we apply a similar procedure moving leftward.

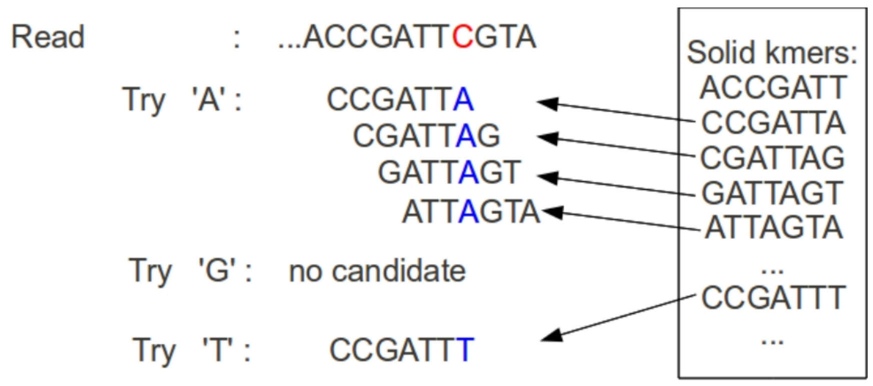

Figure 2: An example of the greedy error correction procedure. k-mer CCGATTC does not appear in Bloom filter B, so we attempt to substitute a different nucleotide for the C shown in red. We select A since it yields the longest stretch of consecutive k-mers that appear in Bloom filter B.

If the error is located near the end of the read, for each candidate substitution, we will extend reads using the kmer reported in Bloom filter A by one path that has the least lexicographical order. Thus we can solve some ties that are more likely to happend near the end of a read due to insufficient extension.

Scaling with coverage

Lighter's accuracy is near-constant as the depth of sequencing K increases and its memory footprint is held constant. The basic idea is that as K increases, we adjust α in inverse proportion. That is, we hold αK constant. For concreteness, consider two scenarios: scenario I, where coverage is K_1 and subsampling fraction is α_1, and scenario II where coverage is $K_2 = zK_1$ and subsampling fraction is $\alpha_2 = \alpha_1/z$.

Contents of Bloom filter A. The occupancy of Bloom filter A, as well as the fraction of correct k-mers in A, are approximately the same in both scenarios. This follows from the fact that $\kappa'_c \sim Pois(\alpha K(1-\epsilon)/G)$, $\kappa'_e \sim Pois(\alpha K\epsilon/H)$, and αK, ϵ, G, and H are constant across scenarios. This is also supported by our experiments, as seen in Table 2. Because the occupancy does not change, we can hold the Bloom filter's size constant while achieving the same false positive rate.

Accuracy of trusted / untrusted classifications. Also, if a read position and its neighbors within $k - 1$ positions on either side are error-free, then the probability it will be called trusted does not change between scenarios. We mentioned that when α is small, $P(\alpha_1) \approx P(\alpha_1/z) = P(\alpha_2)$. We also showed that the false positive rate of the bloom filter is approximately constant between scenarios, so $P^*(\alpha_1) \approx P^*(\alpha_1/z) = P^*(\alpha_2)$. Thus, the thresholds y_x will also remain unchanged. $p_c = (p(\kappa'_c \geq 1))/(p(\kappa_c \geq 1))$ is the probability a correct k-mer is in the subsample given that it was sequenced. $p_c = (1 - e^{-\alpha(1-\epsilon)K/G})/(1 - e^{-(1-\epsilon)K/G}) \approx 1 - e^{-\alpha(1-\epsilon)K/G}$, since $(1-\epsilon)K/G$ is large. p_c is constant across scenarios since αK, ϵ, and G are constant. Since p_c is constant, the parameters of the $B_{e,x}$ distribution are constant and the probability a correct position will be called trusted is also constant.

Now we consider an incorrect read position. We ignore false positives from Bloom filter A for now. $p_e = p(\kappa'_e \geq 1)/p(\kappa_e \geq 1) = (1 - e^{-\alpha\epsilon K/H})/(1 - e^{-\epsilon K/H})$ is the probability an incorrect k-mer is in the subsample given that it was sequenced. Since $\epsilon K/H$ is close to 0, $e^{-\epsilon K/H} \approx 1 - \epsilon K/H$ and $p_e \approx (\alpha\epsilon K/H)/(\epsilon K/H) = \alpha$. Say an incorrect read position is covered by x k-mers; if $B_{e,x}$ is a random variable for the number of k-mers overlapping the position that appear in Bloom filter A, then $B_{e,x} \sim Binom(x, p_e) \approx Binom(x, \alpha)$. The probability of falsely trusting a position is therefore: $p(B_{e,x} \geq y_x) = \sum_{i=y_x}^{x} \binom{x}{i} p_e^i (1 - p_e)^{x-i} \approx \sum_{i=y_x}^{x} \binom{x}{i} \alpha^i (1-\alpha)^{x-i}$. If we omit the $(1-\alpha)^{x-i}$ term in the sum, what remains is an upper bound, i.e. $\sum_{i=y_x}^{x} \binom{x}{i} \alpha^i (1-\alpha)^{x-i} \leq \sum_{i=y_x}^{x} \binom{x}{i} \alpha^i$. Since $\alpha_2 = \alpha_1/z$, the upper bound in scenario II is lower by a factor of at least $1/z$ relative to the upper bound in scenario I. So an upper bound on the probability of labeling an incorrect position as trusted decreases by a factor of at least z. When K increases, the number of distinct test cases for incorrect positions increases by a factor of at most z. Thus, we expect the total number incorrect positions labeled as trusted to remain approximately constant.

When α is small, the false positive rate β may dominate the probability p_e. In practice, however, the false positive rate is usually small enough that the probability of a incorrect position being labeled as trusted due to false positives is extremely low. For example, for many of the experiments reported in this study, the k-mer length $k = 17$, the false positive rate of Bloom A ≈ 0.004, the threshold $y_{2k-1} = 6$, and $\alpha = 0.05$. In this situation, $p(B_{e,x} \geq y_x) \approx 5 \cdot 10^{-11}$.

The above is not an exhaustive analysis, since we have not examined the case where a read position is error-free but not all of its neighbors within $k-1$ positions on either side are error-free. In this case, whether the threshold is passed depends chiefly on the whereabouts of the nearby errors.

Contents of Bloom filter B. Given the analysis in the previous section, we expect that the collection of k-mers drawn from the stretches of trusted positions in the reads will not change much across scenarios and, therefore, the contents of Bloom filter B will not change much. This conclusion is also supported by our experiments, as seen in Table 2.

Quality score

Lighter can make use of base quality values. Specifically, a low quality value at a certain position can force that position to be untrusted, even if the overlapping k-mers indicate it is trusted. First, Lighter scans the first 1 million reads in the input, recording the quality value at the last position in each read. Lighter then chooses the 5th-percentile quality value; that is, the value such that 5% of the values are less than or equal to it say t_1. Use the same idea, we get another 5th-percentile quality, say t_2 value for the first 1 million reads' first base. When Lighter decides whether a position is trusted or not, if its quality score is less or equal to $min\{t_1, t_2 - 1\}$, then call it untrusted regardless of how many of the overlapping k-mers appear in Bloom filter A.

Parallization

As shown in Figure 1, Lighter works in three passes: (1) populating Bloom filter A with a k-mer subsample, (2) applying the per-position test and populating Bloom filter B with likely-correct k-mers, and (3) error correction. For pass 1, because α is usually small, most time is spent scanning the input reads. Consequently, we found little benefit to parallelizing pass 1. Pass 2 is parallelized by using concurrent threads handle subsets of input reads. Because Bloom filter A is only being queried (not added to), we need not synchronize accesses to A. Accesses to B are synchronized so that additions of k-mers to B by different threads do not interfere. Since it is typical for the same correct k-mer to be added repeatedly to B, we can save synchronization effort by first checking whether the k-mer is already present and adding it (synchronously) only if necessary. Pass

Coverage		35×		70×		140×	
Error rate		1%	3%	1%	3%	1%	3%
α for lighter		0.1	0.1	0.05	0.05	0.025	0.025
Recall	quake	89.59	48.77	89.64	48.82	89.59	48.78
	musket	92.61	92.04	92.60	92.05	92.60	92.03
	bless	98.68	97.29	98.69	97.48	98.65	97.47
	lighter	**99.42**	**98.03**	**99.36**	**98.93**	**99.39**	**98.99**
Precision	quake	**99.99**	**99.99**	**99.99**	**99.99**	**99.99**	**99.99**
	musket	99.78	99.63	99.78	99.63	99.78	99.63
	bless	98.90	98.59	98.88	98.62	98.88	98.61
	lighter	99.10	99.14	99.08	99.18	99.07	99.18
F-score	quake	94.51	65.56	94.54	65.61	94.51	65.57
	musket	96.06	95.68	96.05	95.69	96.05	95.68
	bless	98.79	97.94	98.78	98.04	98.77	98.04
	lighter	**99.26**	**98.58**	**99.22**	**99.06**	**99.23**	**99.09**
Gain	quake	89.58	48.76	89.64	48.82	89.59	48.78
	musket	92.40	91.70	92.39	91.71	92.39	91.69
	bless	97.58	95.90	97.57	96.11	97.54	96.09
	lighter	**98.52**	**97.17**	**98.44**	**98.12**	**98.46**	**98.18**

Table 1: Accuracy measures for simulated rate(%) for each table for different coverages

3 is parallelized by using concurrent threads to handle subsets of the reads; since Bloom filter B is only being queried, we need not synchronize accesses.

Evaluation

Simulated data set

Accuracy on simulated data. We compared Lighter's performance with Quake [12], Musket [13] and Bless [8]. We generated collection of reads simulated from the reference genome for the K12 strain of *E. coli* using Mason v0.1.2 [**?**]. We let k-mer size $k = 17$ for all programs unless otherwise noted. Li: (done)please fill in the proper Mason version number as well as version numbers for the three error correctors. Also, please specify full set of Mason parameters somewhere and mention at least the read length in the main text.

We simulated six distinct datasets whose reads length are all 101bp, varying average coverage (35x, 75x 140x) and average error rate (1% and 3%). For a given error rate e we specify Mason parameters `-qmb` $e/2$ `-qme` $3e$, so that the average error rate is e but errors are more common toward the 3' end, as in real datasets. (done)Li: please correct what I say here, which I don't think is quite what you do

We then ran all three tools on all six datasets, with results presented in Table 1. In these comparisons, a true positive (TP) is an instance where an error is successfully corrected, i.e. with the correct base substituted. A false positive (FP) is an instance where a spurious substitution is made at an error-free position. A false negative (FN) is an instance where we either fail to detect an error or an incorrect base is substituted. As in previous studies, we report the following summaries: recall = TP/(TP+NP), precision = TP/(TP+FP), F-score = 2×recall×precision/(recall+precision) and gain = (TP-FP)/(TP+FN).

Unlike the other tools, Quake both trims the untrusted tails of the reads, and discards reads that it cannot correct. This leads to very high precision relative to other tools, though at the expense of discarded data. Of the remaining tools, Lighter and Musket achieve the highest precision, with Musket achieving slightly higher precision. Lighter achieves the highest recall, F-score and gain in all experiments.

Coverage	Bloom A	Bloom B
10×	41.563	19.843
20×	41.555	32.765
35×	41.555	33.802
70×	41.580	33.906
140×	41.577	33.881
280×	41.571	33.903

Table 2: Occupancy rate(%) for each table for different coverages

Scaling with depth of simulated sequencing. We also used Mason to generate a series of datasets with 1% error, similar to those used in Table 1, but for 10×, 20×, 35×, 70×, 140× and 280× average coverage. We ran Lighter on each and measured final occupancies (fraction of bits set) for Bloom filters A and B. If our assumptions and scaling arguments are accurate, we expect the final occupancies of the Bloom filters to remain approximately constant for relatively high levels of coverage. As seen in Table 2, this is indeed the case. Note that when coverage is quite low (10×), the occupancy of table B is significantly lower, since distributions of multiplicities of correct and incorrect k-mers become too similar to distinguish clearly.

Cardinality of Bloom filter B. We also measured the number of k-mers added to table B, i.e. its cardinality. We used the Mason dataset with 70x coverage and 1% error rate. The *E. coli* genome has 4,553,699 distinct k-mers, and 4,553,653 (99.999%) of them are in table B. The missing k-mers are likely due to low coverage at the extreme ends of the chromosome. Li: Is 4,553,653 the number of *correct* k-mers in the filter, or just the number of k-mers? (4553653 are the number of correct k-mers. Also The 4553699 is the number of total distinct kmers not distinct canonicalized kmers.)

We conducted a similar experiment with Mason configured to simulate reads from a diploid version of the *E. coli* genome. Specifically, Mason was configured to introduce heterozygous SNPs at 0.1% of the reference positions. Mason then sampled the same numbers of reads from both haplotypes, making a dataset with 70x average coverage. Of the 159,117 simulated k-mers overlapping a position with a heterozygous SNP, table B held 158,987 (99.918%) of them at the end of the run. Li: Need to explain how we configured *E. coli* better. 0.1% SNP is HETs? HOMs? Both? Also, need to have the exact command-line arguments used written down somewhere. (They are are HET. I make sure the corresponding positions of the two references are different.)

Effect of varying α. In a series of experiments, we measured how different settings for the subsampling fraction α affected Lighter's accuracy (recall, precision, F-score and gain) as well as the occupancies of Bloom filters A and B. We three datasets simulated by Mason with 35×, 70× and 140× coverage. The simulated error rate was 1% in all cases.

As shown in Figures 3 and 4, only a fraction of the correct k-mers are added to A when α is very small, causing many correct read positions to fail the threshold test. Lighter attempts to "correct" these error-free positions, decreasing accuracy. This also has the effect of reducing the number of consecutive stretches of k trusted positions in the reads, leading to a smaller fraction of correct k-mers added to B, and ultimately to lower accuracy. When α grows too large, the y_x thresholds grow to be greater than k, causing all positions to fail the threshold test, as seen in Figure 4's right-hand side. This also leads to a dramatic drop in accuracy as seen in Figure 3. Between the two extremes, we find a broad range of values for α (from 0.06 to 0.45) that yield high accuracy. Li: Figure 3 is too hard to read. We might either supplement with a table, or make a second plot zoomed in on y = [95, 100] or something. (Replaced by a figure from [85-100])

Effect of varying k. A key parameter of Lighter is the k-mer length k. When k is smalls, there is a higher probability that a k-mer affected by a sequencing error also appears elsewhere in the genome. When k is

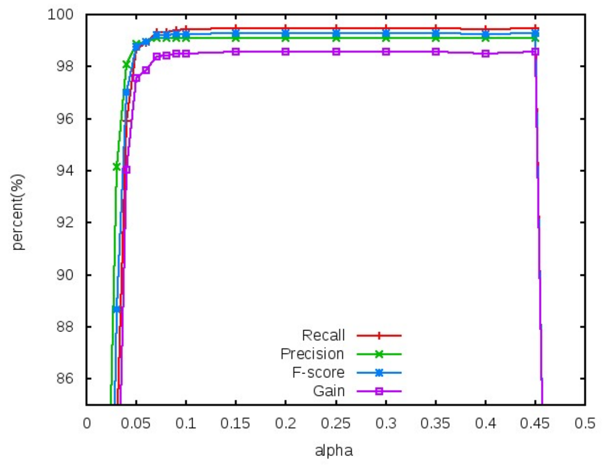

Figure 3: The effect of α on accuracy using the simulated 35× dataset.

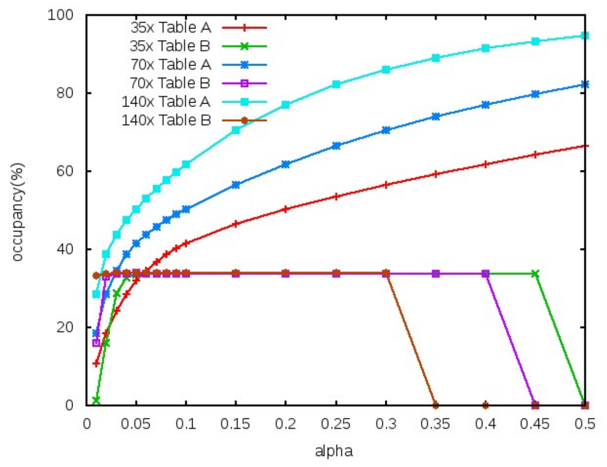

Figure 4: The effect of α on occupancy of Bloom filters A and B using simulated 35×, 70× and 140× datasets.

too large, there may be relatively few error-free k-mers, making Bloom filter A incomplete. We measured how different settings for k affect accuracy. Results are shown in Figure 5.

Real datasets

E. coli. Here we analyze ERR022075, a real sequencing dataset containing very deep coverage of the K-12 strain of the *E. coli* genome. We again used Quake, Musket, Bless and Lighter to correct errors in the dataset. To obtain a level of coverage more reflective of other projects, we randomly subsampled the reads in the dataset to obtain roughly 75x coverage (3.5M reads). The reads are paired-end with the ends having lengths 100 nt and 102 nt. Because Bless cannot handle paired-end reads with different length, we truncate the last 2 bases from the 102 nt end.

We measured that out of 4,553,699 distinct k-mers in the *E. coli* K12 reference genome, 4,519,538

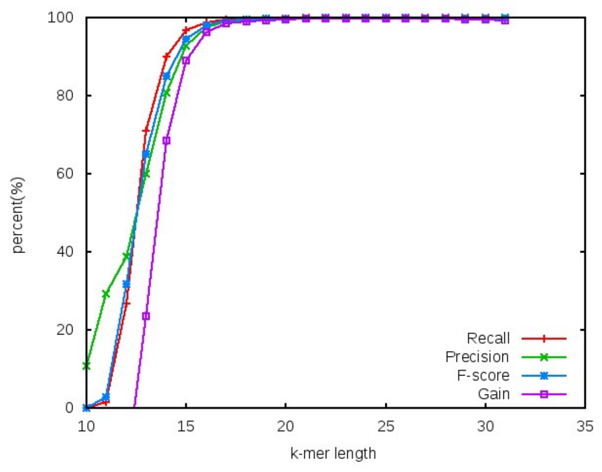

Figure 5: The effect of k-mer length k on accuracy.

	Read Level			Base Level	
	Total Reads	Mapped Reads	Gain(%)	Match/Read	Gain(%)
Original	3,497,616	3,464,140	-	99.038	-
Quake	3,369,584	3,368,911	-2.75	97.305	-1.75
Musket	3,497,616	3,467,878	0.11	99.601	0.57
Bless	3,497,616	3,472,979	0.26	99.611	0.58
Lighter	3,497,616	3,476,425	0.35	99.611	0.58

Table 3: Alignment statistics for the 75× *E. coli* data set, before error correction (Original row) and after error correction (Quake, Musket, Bless and Lighter rows). The first "Gain" column shows percent increase in reads aligned. The second "Gain" column shows percent increase in average number of matching positions per aligned read.

(99.250%) appear in table B.

Since these data are not simulated, we cannot measure accuracy directly. We can measure it indirectly, as other have done [8], by measuring how error correction affects read alignment statistics. We use Bowtie2 [**?**] to align the original reads and the corrected reads to the reference genome, then count the total number of the matched positions in the alignments in both cases. The results are shown in Table 3. Lighter yields the greatest improvement in fraction of reads aligned and in fraction of alignment positions that are matches. As before, Quake is hard to compare to the other tools because it both trims reads and suppresses reads it cannot correct. This leads to negative values in the "Gain" columns.

Also, for each tool we took the reads that aligned without indels and without trimming (in the case of Quake) and calculated the fraction of matching bases at each read position. These fractions are plotted in Figure 6. An unusual feature of this dataset is that many of the reads begin with an "N" indicating that the sequencer was unable to make a base call at that position. Li: These are paired-end reads. Are we plotting end 1 or end 2, or both?(Both) Also, is the fact that many of the reads begin with N affecting the results in an important way? That might put us at a disadvantage, since we're less able to correct an error at the extreme end of the read than a tool that counts k-mers. We could pick another dataset, or exclude reads with Ns when we subsample.(But the results of alignment and assembly looks very nice, so I think it is fine.)

To further assess accuracy, we assembled the reads before and after error correction and measured relevant assembly statistics. We used Velvet 1.2.10 [**?**], a De Bruijn graph-based assembler for second-generation sequencing reads. A key parameter of Velvet is the graph's k-mer length. To avoid being overly affected by

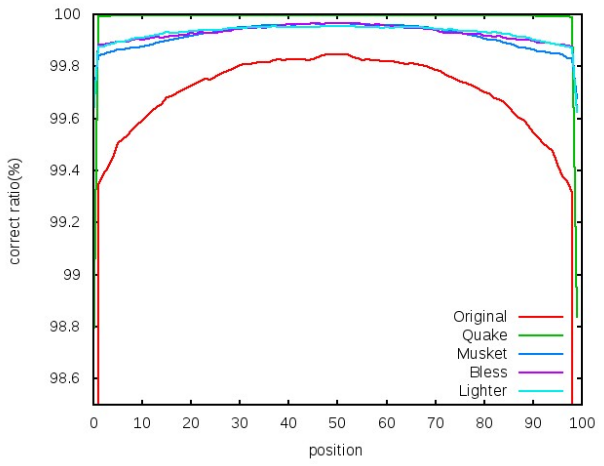

Figure 6: The matching ratio for each base in *E. coli* data set

	N50	NG50	Edits / 100kbps	Misassemblies	Coverage(%)
Original	100,039	100,039	4.18	0	97.412
Quake	125,252	125,252	4.23	8	97.450
Musket	99,995	95,384	6.96	3	97.579
Bless	104,682	99,991	4.45	3	97.352
Lighter	105,574	104,682	5.02	5	97.548

Table 4: De novo assembly of E.Coli data set

poor choice of k-mer length, for each dataset we ran Velvet with several k-mer lengths and reported statistics for the assembly with the best N50 contig size. For each assembly, we then evaluated the assembly's quality using Quast [**?**]. Since assemblers often generate many very short contigs, we only evaluate contigs with length $\geq$ 100 bp. Li: I'm unclear on what exactly we aren't doing with the contigs less than 100 bp(By default, Quast will throw away all the contigs less than 500. I think it is too harsh, so I choose 100bp which is the default threshold used by SOAPdenovo2.). Please fill in the version of Velvet used.(done)

N50 is the length such that the total length of the contigs no shorter than the N50 cover at least half the assembled genome. NG50 is similar, but with the requirement that contigs cover half the reference genome rather than half the assembled genome. Edits per 100kbps is the number of mismatches or indels per 100kbps when aligning the contigs to the reference genome. Misassembly is a position of a contig that its left sequence and right sequence are mapped far away on the genome or overlap too much. The coverage are the portion of the reference genome that are covered by contigs. Li: I'm unclear on the meaning of misassembly here.

Human Chr14

We also evaluated Lighter's effect on alignment and assembly using a dataset from the GAGE project [**?**]. The dataset consists of real 101 × 101 bp paired-end reads covering human chromosome 14 to 35× average coverage. We specified a k-mer length of 19 for Lighter Li: We specified k-mer length 19 for all tools including both error correctors and assemblers?(done)

Error correction's effect on Bowtie 2 alignment statistics are shown in Table 5. Here Bowtie 2 is aligning the reads to an index consisting of just the human chromosome 14 sequence of hg19. Li: Correct? Which assembly?

	Read Level			Base Level	
	Total Reads	Mapped	Gain(%)	Match/Read	Gain(%)
Original	36,504,800	35,993,149	-	99.492	-
Quake	32,560,518	32,546,450	-9.58	93.411	-6.11
Musket	36,504,800	36,276,990	0.79	100.104	0.62
Bless	36,504,800	36,297,288	0.84	100.192	0.70
Lighter	36,504,800	36,280,350	0.80	100.109	0.62

Table 5: Alignment of chr14 data set

	N50	NG50	Edits / 100kbps	Misassemblies	Coverage(%)
Original	8096	5940	134.53	1097	78.490
Quake	7935	5809	143.91	2110	77.303
Musket	6261	4613	128.97	811	78.984
Bless	8397	6201	131.87	1425	78.747
Lighter	8370	6211	133.22	1416	78.858

Table 6: De novo assembly of chr14 data set

We also tested error correction's effect on de novo assembly using Velvet for assembly and Quast to evaluate the quality of the assembly. Results are shown in Table 6. Overall, Lighter's accuracy on real data is comparable with other error correction tools.

Speed, space usage, and scalability

We compared Lighter's memory usage, disk usage, and running time with Quake, Musket and Bless. These experiments use a computer with 48 2.1GHz AMD Opteron processors, 512G memory and operating system Red Hat 4.1.2-52. Li: Please also give OS with version, and give CPU version.(done) The input data is on the simulated data set with different coverage and with 1% error rate used in the previous section and the chr14 real data from Gage. Li: The simulated data is simulated from *E. coli*?(yes) How do you measure peak memory usage? Is it peak virtual memory usage?(the resident memory)

	35×		70×		140×		chr14	
	memory	disk	memory	disk	memory	disk	memory	disk
Quake	2.8G	3.3G	7.1G	6.0G	14G	12G	48G	57G
Musket	139M	0	160M	0	241M	0	1.9G	0
Bless	10M	661M	11M	1.3G	13M	2.6G	600M	15G
Lighter	31M	0	31M	0	31M	0	510M	0

Bless and Lighter achieve constant memory footprint across sequencing depths. While Musket uses less memory than Quake, it uses more than either Bless or Lighter. Bless achieves constant memory footprint across sequencing depths, this comes at the cost of disk consumption. Note that Bless can be configured to trade off between peak memory footprint and peak disk usage.

To assess scalability, we also compared running time for Quake, Musket and Lighter using a different number of threads. For these experiments we used the simulated data set with 70× coverage and 1% error in Figure 7. Li: simulated *E. coli* dataset?(yes) Note that Musket requires at least 2 threads due to its master-slave design. Bless can only be run with one thread and its running time is 1475s, which is slower than Quake.

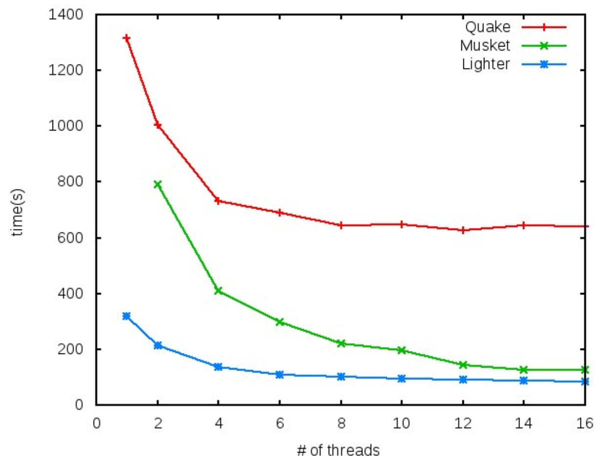

Figure 7: Running times of Quake, Musket and Lighter on 70× simulated data set with increasing number of threads

Discussion

At Lighter's core is a method for obtaining a set of correct k-mers from a large collection of sequencing reads. Unlike previous methods, Lighter does this without counting k-mers. By setting its parameters appropriately, its memory usage and accuracy can be held almost constant with respect to depth of coverage. It is also quite fast and memory-efficient, and requires no temporary disk space to operate.

Though we demonstrate Lighter in the context of sequencing error correction, Lighter's counting-free approach could be applied in other situation where a collection of solid k-mers is desired. For example, one tool for scaling metagenome sequence assembly uses of a Bloom filter populated with solid k-mers as a memory-efficient, probabilistic representation of a De Bruijn graph [17]. Other tools use counting Bloom filters [2, 5] or the related CountMin sketch [4] to represent De Bruijn graphs for compression [10] or digital normalization and related tasks [24]. We expect Ideas from Lighter could be useful in reducing the memory footprint of these and other tools.

Lighter is free open source software released under the XXX license. The software and its source are available from https://github.com/mourisl/Lighter/.

Acknowledgements

The authors thank Jeff Leek for helpful discussions.
Funding: National Science Foundation grant DBI-1159078 to LF and a Sloan Research Fellowship to BL.
Conflicts of Interest: none declared.

References

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

[2] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *Algorithms–ESA 2006*, pages 684–695. Springer, 2006.

[3] M. Chaisson, P. Pevzner, and H. Tang. Fragment assembly with short reads. *Bioinformatics*, 20(13):2067–2074, 2004.

[4] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[5] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[6] T. C. Glenn. Field guide to next-generation dna sequencers. *Molecular Ecology Resources*, 11(5):759–769, 2011.

[7] E. C. Hayden. Is the $1,000 genome for real? *Nature News*, 2014.

[8] Y. Heo, X.-L. Wu, D. Chen, J. Ma, and W.-M. Hwu. Bless: Bloom-filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, page btu030, 2014.

[9] L. Ilie, F. Fazayeli, and S. Ilie. Hitec: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3):295–302, 2011.

[10] D. C. Jones, W. L. Ruzzo, X. Peng, and M. G. Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic acids research*, 40(22):e171–e171, 2012.

[11] W.-C. Kao, A. H. Chan, and Y. S. Song. Echo: a reference-free short-read error correction algorithm. *Genome research*, 21(7):1181–1192, 2011.

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

[13] Y. Liu, J. Schröder, and B. Schmidt. Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. *Bioinformatics*, 29(3):308–315, 2013.

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

[15] P. Medvedev, E. Scott, B. Kakaradov, and P. Pevzner. Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*, 27(13):i137–i141, 2011.

[16] P. Melsted and J. K. Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC bioinformatics*, 12(1):333, 2011.

[17] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown. Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[18] P. A. Pevzner, H. Tang, and M. S. Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[19] L. Salmela and J. Schröder. Correcting errors in short reads by multiple alignments. *Bioinformatics*, 27(11):1455–1461, 2011.

[20] J. Schröder, H. Schröder, S. J. Puglisi, R. Sinha, and B. Schmidt. Shrec: a short-read error correction method. *Bioinformatics*, 25(17):2157–2163, 2009.

[21] H. Shi, B. Schmidt, W. Liu, and W. Müller-Wittig. A parallel algorithm for error correction in high-throughput short-read data on cuda-enabled graphics hardware. *Journal of Computational Biology*, 17(4):603–615, 2010.

[22] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *Communications Surveys & Tutorials, IEEE*, 14(1):131–155, 2012.

[23] X. Yang, K. S. Dorman, and S. Aluru. Reptile: representative tiling for short read error correction. *Bioinformatics*, 26(20):2526–2533, 2010.

[24] Q. Zhang, J. Pell, R. Canino-Koning, A. C. Howe, and C. T. Brown. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *arXiv preprint arXiv:1309.2975*, 2013.