A Probablistic Space-efficient Method of Obtaining Solid Kmers with An Appliction of Error Correction

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

Abstract

Text for this section.

Introduction

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 it in Bloom filter A. The second pass uses Bloom filter A to identify "solid" k-mers and stores these in Bloom filter B. The third pass uses Bloom filter B to detect and correct errors in the input reads.

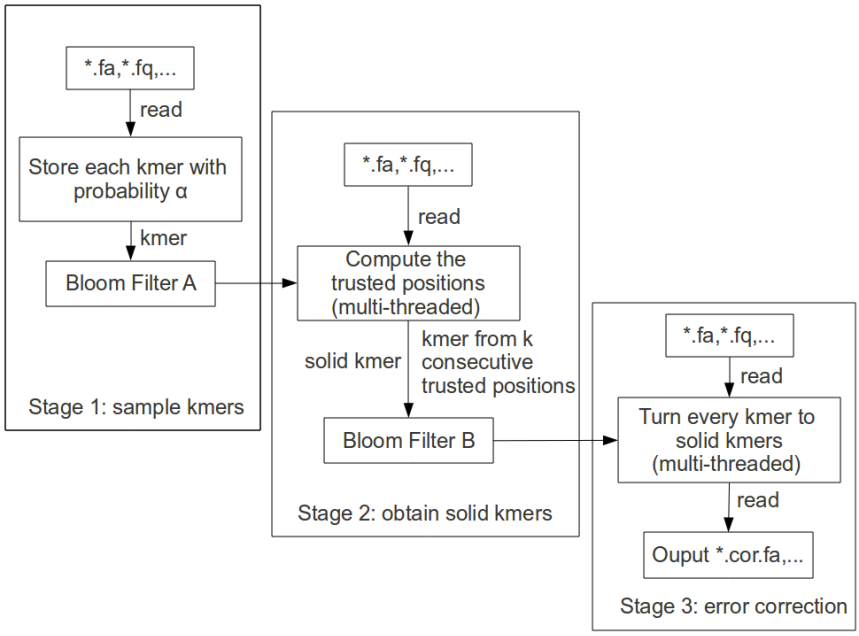

Figure 1: The framework of Lighter

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, also called the *occupancy*. 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 incur more memory usage and greater values of k require more hash function calculations. Many variations on Bloom filters have been proposed that additionally permit, for example, compression of the filter, storage of count data, representation of maps in addition to sets, etc [8]. Bloom filters and variants thereon have been applied in various bioinformatics settings, including assembly [5], compression [3], k-mer counting [4], and error correction [7].

By way of contrast, another way to represent a set is with a hash table. These have the advantage of never yielding 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.

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 less than or equal to the other.

First pass: downsampling

Consider a k-mer drawn from a read. The k-mer is *incorrect* if its sequence has been altered by one or more sequencing errors. Otherwise it is *correct*. Others have noted that, given a dataset with deep and uniform coverage, incorrect k-mers occur rarely (usually just once) while correct k-mers occur many times, proportionally to coverage [2, 6].

In the first pass over the input reads, Lighter examines each k-mer of each read, canonicalizing and storing it in Bloom filter A with probability α, where α is an adjustable parameter. Say a particular k-mer sequence occurs a total of N_k times in the dataset. If the α filter discards all N_k occurrences, the k-mer is never added to A and A's occupancy is reduced by one. Thus, reducing α in turn reduces A's occupancy. Because correct k-mers are more numerous, incorrect k-mers tend to be discarded from A before correct k-mers as α decreases.

Assume that, regardless of average coverage, there is always a value of α for which $n \approx G$, where G is the length of the sequenced genome, and the k-mers stored in A are almost all correct. If this is the case, we can fix m and h and adjust α to achieve a desired false positive rate. This is in contrast to fixing n and adjusting m and h as we did before. We do not argue here that the assumption is true, but we do show through a series of experiments that, in practice, α can be set to achieve desirable combinations of m, k and false positive rate for a range of average coverage values.

Second pass: obtaining solid k-mers

For each position in a read, it can affect up to x kmers where $1 \leq x \leq k$. If that position is wrong, these x kmers will not show up many times and they are mostly likely not in A. Suppose, a kmer with frequency less than f is very weak, then the probability of such kmer be in A is less than $P = 1 - (1 - \alpha)^f$. For the x kmers containing the specific position, we can assume that the number of those stored in A follows a

binomial distribution with parameter (x, P). We can decide whether the nucleotide of a position is trusted by one-tail hypothesis testing. The null hypothesis is that this position is wrong and the p-value can be get from test again (x, P) using the actual number of kmers in A containing that position. We use y to denote this actual number of kmers. In our method, suppose y' is the first number whose p-value is smaller than 0.005, if $y > y'$, we reject the null hypothes and the position is trusted. The benefit of using hypothesis testing instead of using a fixed threshold is that we can handle the case where $x < k$ in a uniform way and it can tolerate the ill-chosen α. Notice that, the position near the left and right boundary or near the error may never be trusted.

To show that we can keep the size of A nearly constant regardless of the coverage. Considering in one case 1, we have the coverage C. And in the next case 2, we double the coverage. Theoretically, the occurence time of error-free kmers should be doubled and so is the counting of the kmers showing up only one or two times. In case 1, we sample the kmers with probability α_1, and α_2 in case 2. In case 1, A kmer is solid if it showed up t times; a kmer is weak if it showed up less than f times. We can set f is proportional to t and is much smaller than t, like $t = 0.05f$. So in case 2, a solid kmer showed up at least about $2t$ times while a weak kmer showed up at most about $2f$ times. If we want the same null hypothesis binomial distribution for the same position in these two cases, we need $P_1 = P_2$ as the base, where $P_1 = 1-(1-\alpha_1)^f$, $P_2 = 1-(1-\alpha_2)^{2f}$.

$$P_2 = 1 - (1-\alpha_2)^{2f} = 1 - (1 - 2\alpha_2 + \alpha_2^2)^f = P_1 = 1 - (1-\alpha_1)^f$$

$$\Leftrightarrow 1 - 2\alpha_2 + \alpha_2^2 = 1 - \alpha \Leftrightarrow \alpha_2 = 1 \pm \sqrt{1-\alpha_1}$$

Since α_2 is no more than 1, $\alpha_2 = 1 - \sqrt{1-\alpha_1}$

Similarly, we can show the same relationship for the hypothesis distribution, P_1' and P_2', where $P_1 = 1 - (1 - 1 - \alpha_1)^t$, $P_2 = 1 - (1 - 1 - \alpha_2)^{2t}$. So we can change the α to make the hypothesis testing from different coverage data set equivalent.

In practice, α_1 is very small, so α_2 is also very small. If we ignore this quadratic term α_2^2, we have $\alpha_2 = \alpha_1/2$. Considering the kmers showing up only once, they takes most of the space and the number of them sampled into table A is the same in case 1 and 2 with parameter α_1 and $\alpha_2/2$ respectively. This shows that our method has the property of keeping the space complexity near constant.

After marking each positions, if there are consecutive k trusted positions, we store the corresponding kmer, which is called solid kmer, in another bloom filter B. Because these kmers are likely from the real genome, the size of B also only depends on the genome size.

In both table A and B, we only store the cannonical form of a kmer, that is the kmer itself or its rever complement depending on who has smaller lexicographical order. We know that there is nearly linear relationship between $1/\alpha$ and f the frequency for the weak kmers, in Lighter, we set $f = max\{2, 0.1/\alpha\}$ and the user will choose α. A rule of thumb for choosing α is by applying the nearly linear relationship between $1/\alpha$ and the coverage, and in this paper we choose $\alpha = 0.05$ when the coverage is 70×. So given the coverage C, we can set α to $0.05\frac{70}{C}$.

Third pass: error correction

To show that our method of getting and storing solid kmers is reliable, we designed an error correction method just using the kmers stored in bloom filter B.

The correction is done read by read. For a read of length $|r|$, we have $|r| - k + 1$ kmers and k_i denote the kmer starts at position i where $1 \leq i \leq |r| - k + 1$. We start the correction by finding a longest consecutive kmers that are stored in B, and then do the correction by scanning towards left and right. Since the correction towards left and right are symmetric, we only demonstrate the correction for the right-hand scanning. Ideally, when we move from the solid kmers towards right, the first time we see an weak kmer, say k_i, there must be an error at position $i + k - 1$. There are three candidates or four(if the position is N) to replace this position. For each candidate, we see how many consecutive kmers is solid starting from k_i with this new character. We choose the candidate creating most solid kmers. Then we resume scanning towards right starting from the next unstored kmer. If none of the candidate makes k_i solid or more than

one candidate create most solid kmers, we say position $i+k-1$ is ambiguous and stop there. If $i+k-1$ is far from the end of the read, we do the error correction on the substring from $i+k-1$ to the end of a read.

The scheme of this greedy error correction method is shown in Figure 2, where we choose found the error because there is no kmer CCGATTC. We choose A to replace C since it gives us most consecutive solid kmers up to the end of the read.

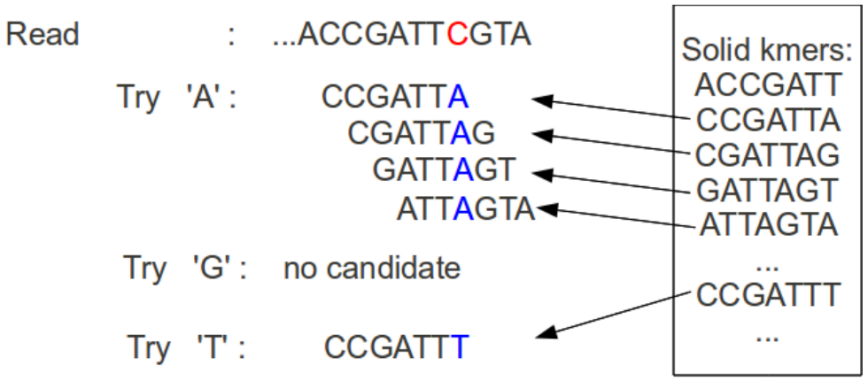

Figure 2: The framework of Lighter

Parallization

As shown in Figure 1, Lighter works in three stages: sampling kmers, obtaining solid kmers and error correction. For the sampling kmers stage, because α is very small, most of the time is spent on scanning the reads namely reading the files. As a result, the overhead of parallilize this part will dominate the benefits, so we decide to remain this stage serial. In the second stage, each thread will handle different reads independently. Reading bloom filter in parallel is very efficient since there is no data race. When a thread find a solid kmer, it firstly tests whether this kmer is stored in table B or not. If this kmer is not in table B, then we write this kmer into the table. In this way, we can avoid many writing locks for table B because a solid kmer shows up many times. The thrid stage can parallized embarrsingly.

Evaluation

Simulated data set

We compared the our result with other released programs, Quake[cite], Musket[cite] and Bless[cite]. We use Mason[cite] to generate a set of data sets with ecoli as the reference genome. The K12 strain of ecoli is fully sequenced and has the decent length for us to generated different scenarios easily.

In the simulated data set, we considered the coverage of 35x, 75x, 140x, and the average error rate of 1% and 3%. So there are totally 6 data sets. Because Mason can also specifiy the error rate of the first and the last base repsectively, suppose the average error rate is e, we set the error rate of the first base to $e/2$ and the error rate of the last base is $3e$. This setting is more realistic than a uniform error rate model.

Coverage	Table A	Table B
35×	41.35	35.33
70×	41.37	35.44
140×	41.37	35.41

Table 1: Occupancy rate(%) for each table for different coverages

We say a base is true positive(TP) if we identify the error and correct it into the right one; a base is false positive(FP) if we correct an original error-free base; a base is false negative(FN) if it is wrong and we either fail to detect its error or we make a wrong correction. As tradition[cite], we use four measurement to evaluate the error correction methods: recall=TP/(TP+NP), precision=TP/(TP+FP), F-score=2×recall×precision/(recall+precision) which is a kind of average of recall and precision and Gain=(TP-FP)/(TP+FN) which measure the benefit of using error correction.

Coverage		35×		70×		140×	
Error rate		1%	3%	1%	3%	1%	3%
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	99.40	97.99	99.33	98.71	99.37	98.87
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	99.11	99.13	99.08	99.16	99.07	99.17
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	99.25	98.56	99.20	98.94	99.22	99.02
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	98.50	97.14	98.41	97.88	98.43	98.04

The α value for the 35×, 70×, 140× is 0.1, 0.05 and 0.025 respectively.

For Quake, we only measured the performance of the reported reads. Because Quake trimmed the untrusted tail of a read and throw away unfixable reads, it has a particularly high precision. But even if we only count the original errors in those reported reads, its sensitivity is very low. Lighter is among the best on different measurement and scenarios. This shows that our sampling method is able to distinguish most solid and weak kmer, and the error correction method is very effective.

For the three data sets with different coverage, we can show Lighter can achieve near constant space usage if the portion of set bits, namely occupancy rate, in the bloom filters stays almost the same. And this is true as shown in Table 1.

To further test the performance of Lighter, we showed how many kmers from the reference genome is stored in table B. We used the simulated data with 70x coverage and 1% error rate. There are totally 4,553,699 distinct kmers from the genome on one strand, and 4,553,653 of them are in table B which is almost all of them. We lose some kmers from the genome mostly due to the low coverage at the two ends of the chromosome.

The 6 simulated data sets are for haploid case, there are many species with a pair of chromosome in a cell, like human. Here we consider the case of diploid. We still use the e.coli genome as the reference genome and then introduce 0.1% SNPs to generate the second reference genome. Mason then will sample the same amount of reads from the two genomes, totally form an 70x coverage data set. There are 159,117 kmers

containing the SNP position from one strand, and table B holds 158,972 of them. Though the performance is not as good as haploid case, it still hold almost all of them.

The sampling rate α is the key in Lighter which ensures the near constant space complexity. We did an evaluation on the simulated data set with 35× coverage and 1% error rate. And in this evaluation, we fix $f = 2$ instead of $f = max\{2, 0.1/\alpha\}$. The result is shown in Figure 3.

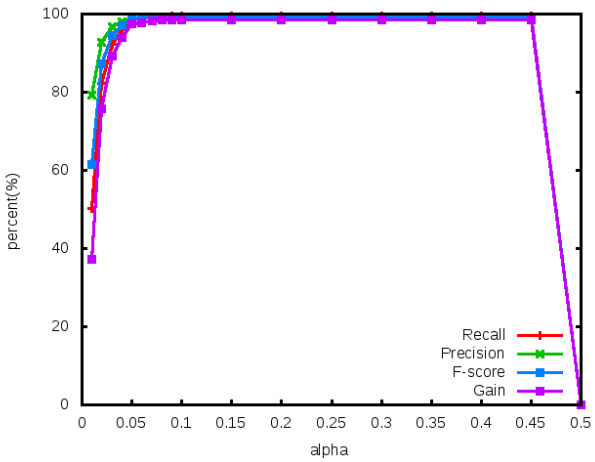

Figure 3: The effect of α

If α is too small, too much solid kmers will be failed to get in table A and too many positions will fail the hypothesis test. As a result, Lighter will try to correct those error-free region which also brings down the precision. And if α become even smaller, then there is no kmer in B, so Lighter will fail to correct all the reads because there is no candidate kmers.

If α is larger, the hypothesis test is an adaptive threshold and handle this case nicely even if there are too many elements in A and the false positive rate of A is higher. However, when α is too large, number k will have a p-value larger than 0.005, and all the positions will be regarded as untrusted. We can observe this from the dramatic drop in Figure 3.

This explanation can also be verified by looking at the occupancy rate of table A and B when changing α shown in Figure 4.

Real data set

E.Coli data set

We use ERR022075 data set which is a very deep coverage sequencing data set of ecoli K-12 strain genome. We still used Quake, Musket, Bless and Lighter to run this data set. Since usually we do not need that much of coverage, we uniformly sampled some reads from this data set and got a data set with roughly 75x coverage. The reads in this data set is paired-end and the read length is 100 and 102. Because Bless can not handle paired-end reads with different length, we truncate the last 2 base to form a 100-length paired-end data set.

Since the genome of E.Coli K-12 strain is fully sequenced, we measured that 4,519,535 out of 4,553,699 kmers from the genome is stored in table B, which is more than 99%. This number can be slightly improved using larger α.

We can also measure the error corrections' affect on the alignment. We use Bowtie2[cite] to align the reads to the reference genome and count the total number of the matched position.

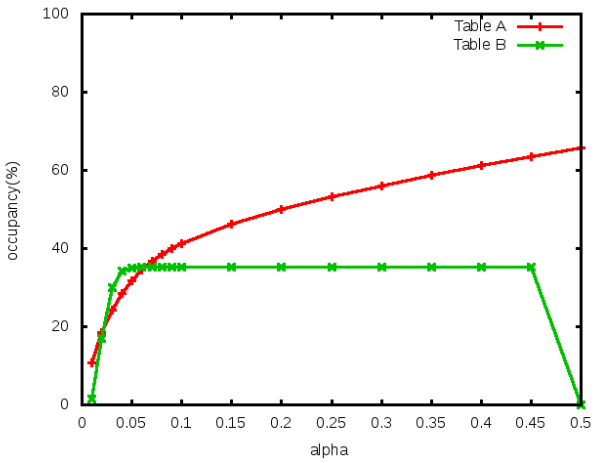

Figure 4: The effect of α

	Total Matched Pos	Improvement(%)	Total pairs	Mapped
Original	343,081,567	-	1,748,808	99.04%
Quake	327,813,255	-4.45%	1,684,792	99.98%
Musket	345,403,149	0.68%	1,748,808	99.15%
Bless	345,947,230	0.84%	1,748,808	99.30%
Lighter	346,273,158	0.93%	1,748,808	99.39%

For Quake, the number is very low mainly due to unreported reads and trimmed 3' end. It turns out Lighter gives the most improvement. We use samtools to do the SNP calling.

For those reads not got trimmed and alinged to the genome without indels (cigar field like 100M in the SAM file), we measure the matched ratio for each position shown in Figure 5. Since many of the reads in this data set start with N, the first base's matching ratio of the original data set is very low.

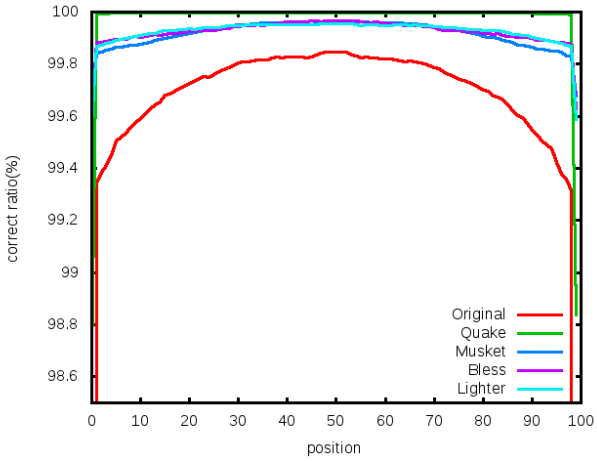

Figure 5: The matching ratio for each base in E.Coli data set

Another common measurement for error correction is to run the de novo assembly and see the change of

	contig N50	scaffold N50
Original	53584	80924
Quake	54854	89802
Musket	47826	89873
Bless	50573	84353
Lighter	48777	86836

Table 2: De novo assembly of E.Coli data set

	Total Matched Pos	Improvement(%)	Total pairs	Mapped
Original	3581034345		18252400	98.60%
Quake	3040192216	-15.10	16280259	99.96%
Musket	3631477901	1.41	18252400	99.38%
Bless	3636682539	1.55	18252400	99.43%
Lighter	3631446457	1.41	18252400	99.38%

Table 3: Alignment of chr14 data set

the N50 of the contigs and scaffolds. We use SOAPDenovo2[cite] to assemble the reads. SOAPDenovo2 is a de novo assembly software based on De Brujin graph. The parameter for SOAPDenovo2 especially the kmer size in the De Brujin graph is very important to the performance. We ran the SOAPdenovo2 on different parameters and choose the one with best N50 contig size.

Human Chr14

So far, we only tested the data on the genome of E.Coli, which is a pretty small. In this section, we use a data set from Gage[cite] on human chromosome 14. This data set is about 35x coverage and 101bp pair-end reads.

Like for the E.Coli data set, we evaluate the result using both Bowtie2 and SOAPdenovo2.

The result of Bowtie2 is shown in Table 3.

And the result

The results of the de novo assembly of the two real data set show that Lighter's result is comparable with other methods.

Running Time and Space Usage

The programs were run on a machine with 48 2.1GHz processors, 512G memory and 74T disk space.

Musket, Bless and Lighter are all aimming for low memory consumption. So we compared the peak memory usage as well as the peak disk consumption on the simulated data set with different coverage and with 1% error rate and the chr14 real data from Gage.

	contig N50	scaffold N50
Original	3372	9219
Quake	3203	8727
Musket	3760	6142
Bless	3763	9254
Lighter	3757	9012

Table 4: De novo assembly of chr14 data set

	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	26M	0	26M	0	26M	0	354M	0

From the table, we can see that Bless and Lighter can achieve nearly constant memory consumption as claimed. Musket took much less memory comparing with Quake but is much worse than Bless and Lighter.

For Bless, we increase the number of temporary files along with the coverage of the data set to achieve the constant memory consumption. Bless uses secondary memory to save the memory consumption, and the overall space consumption is much larger than Musket and Lighter.

For the chr14 data set, The memory consumption of Bless can be decreased if we create more temporary files.

We compared the running time between Quake, Musket and Lighter using different number of threads on the simulated data set with 70× coverage and 1% error in Figure 6. Musket requires at least 2 threads due to its master-slave style parallelism, so its figure start at when number of threads is 2.

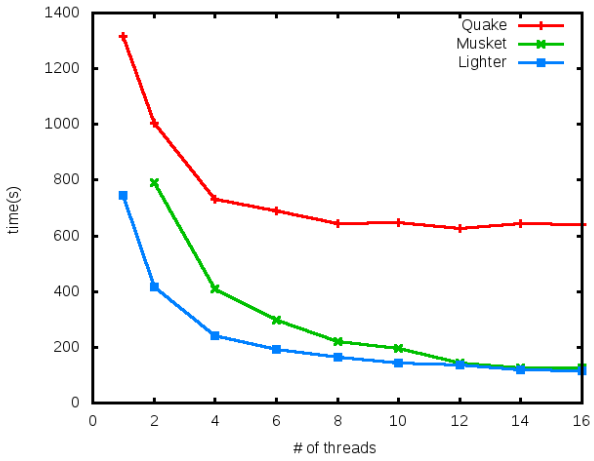

Figure 6: The running time on simulated data set with different number of threads

So far, Bless can only run in single-thread mode and it takes 1475s to finish which is much slower than Musket and Lighter.

The result shows that Lighter is very efficient and scalable.

Discussion

In this paper, we propose a probabilitic method of obtaining solid kmers without counting. It is the first method that is able to find solid kmer with near constant space complexity and it is very easy to implement.

We use this method to implement an error correction program Lighter for DNA-seq reads, which is comparable with other programs. This shows that our method of finding solid kmer is quite reliable.

Another immediate application of our method is getting solid kmers' count, which can be done by scanning the reads again to get the real count for the solid kmers stored in table B.

Author's contributions

Text for this section ...

Acknowledgements

Text for this section ...

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.