Quality-Aware, Parallel, Multistage Detection and Correction of Sequencing Errors using Storm

Lakshmisha Bhat (lbhat1@jhu.edu)
Department of Computer Science, Johns Hopkins University

Abstract

The sequence data produced by next-generation sequencing technologies are error-prone and has motivated the development of a number of short-read error correctors in recent years. The majority of methods focus on the correction of substitution errors, which are the dominant error source in data produced by Illumina sequencing technology. Our efforts are also aligned towards the same goal. We design a streaming pipeline that takes a stream of sequence reads, builds a massively distributed abundance histogram assisted by a distributed sketch and use this information to detect which read nucleotides are likely to be sequencing errors, all within the Storm ecosystem. Then, using Naive Bayes and maximum likelihood approach, we correct errors by incorporating quality values on realistically simulated reads.

1 Introduction

We saw earlier in the course that Sanger reads, typically 700 – 1000 bp in length, are long enough for overlaps to be reliable indicators of genomic co-location, which are used in the overlap-layout-consensus approach for genome assembly. However, the de novo inference of a genome without the aid of a reference genome, is a hard problem to solve. The overlap-layout-consensus approach does poorly with the much shorter reads of second-generation sequencing platforms; for e.g. DNA sequence reads from Illumina sequencers, one of the most successful of the second-generation technologies, range from 35 to 125 bp in length. In this context, de Bruijn graph [5] based formulations that reconstruct the genome as a path in a graph perform better due to their more global focus and ability to naturally accommodate paired read information. As a result, it has become de facto model for building short-read genome assemblers. In this setting it is worth talking about the correctness of these second-generation sequencers. It is well known that the sequence fidelity of these sequencers are high as they have a relatively low substitution error rate of 0.5–2.5 (empirically shown in [4]). Errors are attributed to templates getting out of sync, by missing an incorporation or by incorporating 2 or more nucleotides at once. These errors increase in the later sequencing cycles as proportionally more templates fall out of sync and hence their frequency at the 3' ends of reads is higher. This, as we shall see later in section 2.1, becomes an important property to be considered for maximum likelihood estimation during error correction.

2 Background

2.1 Error Correction

Error correction has long been recognized as a critical and difficult part of the so called graph-based assemblers (de Bruijn graph). It also has significant impact on alignment and in other next-generation sequencing applications such as re-sequencing. Sequencing errors complicate analysis, which normally requires that reads be aligned to each other during assemble or to a reference genome for single-nucleotide polymorphism (SNP) detection. Mistakes during the overlap computation in genome assembly may leave gaps in the assembly, while false overlaps may create ambiguous paths [4]. In genome re-sequencing projects, reads are aligned to a reference genome, usually allowing for a fixed number of mismatches due to either SNPs or sequencing errors. In most cases, the reference genome and the genome being newly sequenced will differ, sometimes substantially. Variable regions are more difficult to align because mismatches from both polymorphisms and sequencing errors occur, but if errors can be eliminated, more reads will align and the sensitivity for variant detection will improve.

Fortunately, the low cost of second-generation sequencing makes it possible to obtain highly redundant coverage of a genome, which can be used to correct sequencing errors in the reads before assembly or alignment. But, this significantly increases the size of the dataset. In order to deal with the computational challenges of working with such large data sets, a number of methods have been proposed for storing k-mers efficiently. Most de Bruijn graph assemblers store k-mers using 2 bits to encode each nucleotide, so that each k-mer takes $k/4$ bytes. The k-mers are then stored in a hash table, usually with some associated information such as coverage and neighborhood information in the de Bruijn graph. A complementary strategy for reducing memory usage is based on the observation that in current data sets, a large fraction of the observed k-mers may arise from sequencing errors. Most of these occur uniquely in the data, and hence they greatly increase the memory requirements of de novo assembly without adding much information. For this reason, it is frequently helpful to either discard unique k-mers prior to building the graph, or to attempt to correct them if they are similar to other, much more *abundant*, k-mers. A number of related methods have been proposed to perform this error correction step, all guided by the goal of finding the minimum number of single base edits (edit distance) to the read that make all k-mers trusted.

A common approach of error correction of reads is to determine a threshold and correct k-mers whose multiplicities fall below a threshold [3, 4, 10]. Choosing an appropriate threshold is crucial since a low threshold will result in too many uncorrected errors, while a high threshold will result in the loss of correct k-mers. The histogram of the multiplicities of k-mers will show a mixture of two distributions that of the error-free k-mers, and that of the erroneous k-mers. When the coverage is high and uniform, these distributions are centered far apart and can be separated without much loss using a cutoff threshold; such methods therefore achieve excellent results [4].

There are some papers that develop methods to correct errors without an assumption of non-uniform coverage. One such method is Hammer [7] Consider two k-mers that are within a small Hamming distance and present in the read dataset. If our genome does not contain many non-exact repeats, then it is likely that both of these k-mers were generated by the k-mer among them that has higher multiplicity. In this way, we can correct the lower multiplicity k-mer to the higher multiplicity one, without relying on uniformity. This can be further generalized by considering the notion of the Hamming graph, whose vertices are the distinct k-mers (annotated with their multiplicities) and edges connect k-mers with a Hamming distance less than or equal to a parameter . Hammer is based on a combination of the Hamming graph and a simple probabilistic model described in great detail in [7].

2.2 Bloom-Filter

A Bloom filter [2], in simple terms, is a data structure designed to tell us, rapidly and memory-efficiently, whether an element is present in a set. The price paid for this efficiency is that a Bloom filter is probabilistic in nature: it tells us that the element is either definitely not in the set or 'may be' in the set.

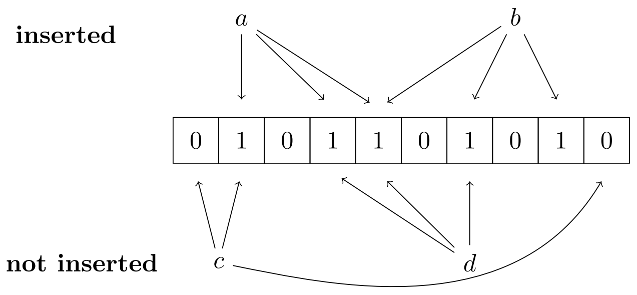

Fig. 1: An example of a Bloom filter with three hash functions. The k-mers a and b have been inserted, but c and d have not. The Bloom filter indicates correctly that k-mer c has not been inserted since not all of its bits are set to 1. k-mer d has not been inserted, but since its bits were set to 1 by the insertion of a and b, the Bloom filter falsely reports that d has been seen already.

More precisely, a Bloom filter is a method for representing a set $A = \{a_1, a_2, \ldots, a_n\}$ of n elements to support membership queries. The main idea is to allocate a vector v of m bits, initially all set to 0, and then choose k independent hash functions, $h_1, h_2, \ldots, h_k$, each with range $\{1, \ldots, m\}$. For each element $a \in A$, the bits at positions h1(a), h2(a), ..., hk(a) in v are set to 1. Given a query for b we check the bits at positions $h_1(b), h_2(b), ..., h_k(b)$. If any of them is 0, then certainly b is not in the set A. Otherwise we conjecture that b is in the set although there is a certain probability that we are wrong. This is called a *false positive*. The parameters k and m should be chosen such that the probability of a false positive is acceptable. Perhaps the most critically acclaimed feature of Bloom filters is that there is a clear trade-off between m and the probability of a false positive. Observe that after inserting n keys into a table of size m, the probability that a particular bit is still 0 is exactly

$$\left(1 - \frac{1}{m}\right)^{kn}.$$

Hence the probability of a false positive in this situation is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{kn/m}\right)^k.$$

The right hand side is minimized for $k = \ln 2 \times m/n$, in which case it becomes

$$\left(\frac{1}{2}\right)^k = (0.6185)^{m/n}.$$

In practice, k must be an integer, and a smaller, suboptimal k might be preferred, since this reduces the number of hash functions that have to be computed.

2.3 Storm

Storm [6] is the *hadoop* of realtime-processing. MapReduce, Hadoop, and related technologies made it possible to store and process data at scales previously unthinkable. But, MapReduce doesn't provide a continuous state abstraction and is tedious to program and configure a pipeline where you need to process events *as they arrive*, compute some form of a sketch [2] which probabilistically represents the data and then persist raw events. Moreover, Storm enables doing a continuous query on data streams (for example, measure the confidence of the Illumina sequencer in different regions of the read as it generates sequences) and streaming the results into clients (who could correct errors on the fly or provide feedback to the sequencer), parallelizing the entire process by providing a fine granular control to the programmer. It is scalable, provides exactly once processing semantics, guarantees no data loss and is extremely robust, unlike *hadoop*, which is a pain to manage and administer.

2.3.1 Components of Storm

There are two kinds of nodes on a Storm cluster: the master node and the worker nodes. The master node runs a daemon called *Nimbus* that is similar to Hadoop's *JobTracker*. Nimbus is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures. Each worker node runs a daemon called the *Supervisor*. The supervisor listens for work assigned to its machine and starts and stops worker processes as necessary based on what Nimbus has assigned to it. Each worker process executes a subset of a topology; a running topology consists of many worker processes spread across several machines.

2.3.2 Storm abstractions and first class citizens

The core abstraction in Storm is the *stream*. A stream is an unbounded sequence of tuples. Storm provides the primitives for transforming a stream into a new stream in a distributed and reliable way. The basic primitives Storm provides for doing stream transformations are *spouts* and *bolts*. Spouts and bolts have interfaces that you implement to run your application-specific logic. A spout is a source of streams while a bolt consumes any number of input streams, does some processing, and possibly emits new streams. Bolts can do anything from run functions, filter tuples, do streaming aggregations, do streaming joins, talk to databases, and more.

3 Methods and Software

3.1 Storm Topology

The figure gives a schematic view of the pipeline we have used for error correction. The nodes in green are the spouts, the diamond shaped nodes are the bolts. Streams are represented by arrows and are annotated with the source and destination cardinalities. These cardinalities correspond to the number of *executors* performing a task in parallel. The process of detecting and correcting errors using this topology is described below.

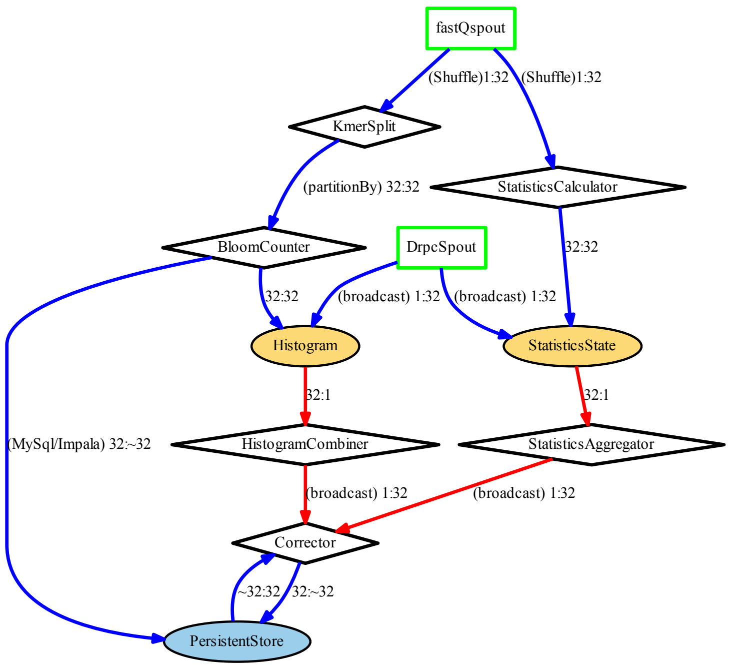

Fig. 2: The storm pipeline used for error correction. The nodes in green are the spouts, the diamond shaped nodes are the bolts. Streams are represented by arrows and are annotated with the source and destination cardinalities.

3.2 Storing and Counting k-mers using Bloom Filter

To count all non-unique k-mers we use a Bloom filter B and a simple hash table T to store k-mers. The Bloom filter keeps track of k-mers we have encountered so far and acts as a *staging area*, while the hash table stores all the k-mers seen at least twice so far. The idea is to use the memory-efficient Bloom filter to store implicitly all k-mers seen so far, while only inserting non-unique k-mers into the hash table. Initially both the Bloom filter and the hash table are empty. All k-mers are generated sequentially from the sequencing reads. For each k-mer, x, we check if x is in the Bloom filter B. If it is not in B then we update the appropriate bits in B to indicate that it has now been observed. If x is in B, then we check if it is in T and bump its count, and if not, we add it to T.

All these k-mer statistics (the sketch, some nucleotide counts per position in the read and the HashTable) are stored within an in-memory state and continuously updated as the stream arrives. The reads are then persisted in a back-end store (single node MySql, 3-node Impala). The stream rate was configured at 100,000 reads per second. We were mostly able to keep up with the stream with 32/64 executors spread across 4 storm nodes (The scale-up wasn't great when using 64 executors due to MySql connection sharing). Occasionally, the network glitches caused storm to replay the messages from the transactional spouts because of which we had to make our state updater idempotent. Not everything was perfect with storm - storm is built to be fail fast - and worker nodes restart as soon as they have a problem. This increases the chances of losing the memory mapped state unless we persist them, perhaps to a fast column store. This can be done easily in storm $BaseStateUpdater < State >$'s commit() method. It should also be noted that configuring storm batches appropriately is extremely important. When we chose a small batch size, the cost of acknowledgments was much higher and thus did not allow us to throttle the spouts as well as we wanted.

This scheme guarantees that all k-mers with a coverage of 2 or more are inserted into T. However a small proportion of unique k-mers will be inserted into T due to false positive queries to B. After the first pass through the sequence data, one can re-iterate over T to obtain exact counts of the k-mers in T and then simply delete all unique k-mers or perhaps all the k-mers with $multiplicity < m$. The time spent on the second round is at most $O(G)$, and tends to be less since hash table lookups are generally faster than insertions.

In our implementation, we use a distributed Bloom Filter that only counts a *partition* of the so-called k-mer stream.

3.3 Localizing errors

We choose a cutoff (multiplicity = 5, in our experiments) to separate trusted and untrusted k-mers based on our intuition. But in general this approach is flawed because the datasets need not have uniform coverage or our genome may contain many non-exact repeats. A highly sophisticated approach to choosing this cutoff is discussed in [3,4]. Once we choose the cutoff, all reads containing an untrusted k-mer become candidates for correction. In most cases the pattern of untrusted k-mers will localize the sequencing error to a small region. The figure 3 from [4] makes this notion very clear. The reader is advised to look it up to find more details.

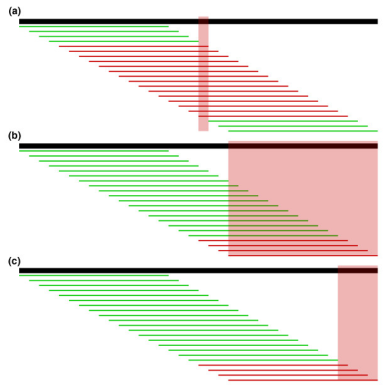

Fig. 3: Trusted (green) and untrusted (red) 15-mers are drawn against a 36 bp read. In (a), the intersection of the untrusted k-mers localizes the sequencing error to the highlighted column. In (b), the untrusted k-mers reach the edge of the read, so we must consider the bases at the edge in addition to the intersection of the untrusted k-mers. However, in most cases, we can further localize the error by considering all bases covered by the right-most trusted k-mer to be correct and removing them from the error region as shown in (c).

3.4 Sequencing error probability model : Naive Bayes

The Naive Bayes Classifier technique is based on the so-called Bayesian theorem and is particularly suited when the dimensionality of the inputs is high, often the case in Natural Language Processing. Despite its simplicity, Naive Bayes can often outperform more sophisticated classification methods. However, despite its simplicity it is very easy to fall into certain pitfalls due to the poor understanding of the central assumption behind

Naive Bayes, namely conditional independence. We take our chances and jump on the bandwagon and throw this model at the sequencing error correction problem. Of course, we make some highly inappropriate assumptions to ensure our correction model is slick and fast. First, we must define the likelihood of a set of corrections. Let the dataset contain n reads. Let O $= O_1, O_2, ..., O_n$ represent the observed nucleotides at a given position in reads $1, 2, ..., n$ respectively, and A $= A_1, A_2, ..., A_n$ the actual nucleotides at the same position in reads $1, 2, ..., n$ of the DNA. Then

$$P(A = a_k | O_1, \ldots, O_n) = \frac{P(A = a_k)\ p(O_1, \ldots, O_n | A)}{p(O_1, \ldots, O_n)}.$$

In plain English the above equation can be written as

$$\text{posterior} = \frac{\text{prior} \times \text{likelihood}}{\text{evidence}}.$$

Now we make the assumption that position p_k in read r_i is conditionally independent of position p_j in read r_j given A. So, now

$$P(A = a_k | O_1, \ldots, O_n) = \frac{1}{Z} P(A = a_k) \prod_{i=1}^{n} p(O_i | A = a_k)$$

where Z (the evidence) is a scaling factor dependent only on $O_1, \ldots, O_n$, that is, a constant if the values of the observed variables O_i are known. Because we compare likelihoods, $P(O_i = o_i)$ is the same for all assignments to A and is ignored. $P(A_i = a_k)$ is defined by the GC% of the genome, which we estimate by counting Gs and Cs in the sequencing reads or by referring to the reference genome. In case of E.Coli, the GC% is 51% which we hard-code for our testing. The $P(O_i | A = a_k)$ is estimated from the *trusted* portions of the read. In machine learning terminology, the trusted k-mers (we can probably relax this to contain all k-mers to get a better generalization) act as our *labeled training set.* That is, to estimate $P(O_i | A_i)$ for all values A_i can take namely A, C, T, G, we set the value $P(O_i = o_i | A_i = o_i) = p_i$ in the conditional probability table. We distribute the rest of the probability i.e. $(1 - p_i)$ among the other $A_i \neq o_i$ according to their prior probabilities (estimated from their GC content).

4 Results

In this paper, we pose a question; is sequencing error correction a streaming application? We argue and empirically prove that Storm, a continuous event processing system, is more than capable of handling these kind of loads and may pave a way to solve several more problems in real time. With Storm's capability to parallelize different parts to the topology to different degrees, we can fine tune our computation to get *optimal* results. We also present a method that identifies all the k-mers that occur more than once in a DNA sequence data set. Our method does this using a Bloom filter, a probabilistic data structure that stores all the observed k-mers implicitly in memory with greatly reduced memory requirements. For our test data sets (E. coli K12 substrain MG1655 [SRA:SRX000429]), we see up to 50% savings in memory usage compared to naive approach, with modest costs in computational speed. This approach may reduce memory requirements for any algorithm that starts by counting k-mers in sequence data with errors.

We also propose to use a straight forward but highly competitive probabilistic model in Naive Bayes to model sequencing errors. We have seen some interesting and positive results of employing this model. However, we aren't quite confident about the completeness of our tests. Unfortunately, due to the time constraint, we have not been able to complete all the tests we hoped to run. Hence we leave this for future work.

5 Conclusions

Counting k-mers from sequencing data is an essential component of many recent methods for genome assembly from short read sequence data. They are also found to be useful in alignment and variant detection. However, in current data sets, it is frequently the case that more than half of the reads contain errors and are observed just once. Since these error-containing k-mers are so large in number, they can overwhelm the memory capacity of available high-performance machines, and they increase the computational complexity of downstream analysis. In this paper, we describe a straightforward application of the Bloom filter to help identify and store the reads that are present more than more than m times in a data set, and hence likely to be trusted. Although one might claim that such methods trades off reduced memory usage for an increase in processing time, our use of fast, simple but sufficiently strong hash function aka. murmur alleviates any such concerns.

6 Advisors

- Ben Langmead (langmea@cs.jhu.edu)

References

[1] Andrew McGregor Ely Porat Alexander Andoni, Assaf Goldberger. Homomorphic fingerprints under misalignments: Sketching edit and shift distances. *STOC'13*, 2013.

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

[3] Rayan Chikhi and Paul Medvedev. Informed and automated k-mer size selection for genome assembly. *BioInformatics*, pages 1–7, 2013.

[4] Steven L Salzberg David R Kelley, Michael C Schatz. Quake: quality-aware detection and correction of sequencing errors. *Genome Biology*, 11(11):13, 2010.

[5] Waterman MS. Idury RM. A new algorithm for dna sequence assembly. *PubMed Central*, 2(2):291–306, 1995.

[6] Nathan Marz. Storm, distributed and fault-tolerant realtime computation. *https://github.com/nathanmarz/storm/wiki/Tutorial*, 2012.

[7] Paul Medvedev, Eric Scott, Boyko Kakaradov, and Pavel Pevzner. Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*, 27(13):i137–i141, 2011.

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

[9] Karin S. Dorman Xiao Yang and Srinivas Aluru. Reptile: representative tiling for short read error correction. *BioInformatics*, 26(20):2526–2533, 2010.

[10] Jan Schrder Yongchao Liu and Bertil Schmidt. Musket: a multistage k-mer spectrum based error corrector for illumina sequence data. *BioInformatics*, 2012.