A Probabilistic Model for Distributed Read Error Correction

Frank Austin Nothaft, Anthony D. Joseph, and David A. Patterson

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA
{fnothaft, adj, pattrsn}@berkeley.edu

Abstract. Read error correction is a key step in many genome assembly protocols, but is computationally expensive. Recent papers have proposed several strategies for approximating steps in the error correction process. In this paper, we derive a rigorous generative model for identifying and correcting errors in short reads. We implement our algorithm using the Apache Spark distributed computing platform. Our implementation demonstrates a performant and exact approach to read error correction that can scale efficiently to mammalian genomes.

Keywords: genome assembly, read error correction, distributed computing

1 Introduction

Read error correction is a key step in most genome assembly protocols, but is computationally expensive [4]. Recent work has explored mechanisms for performing error correction with probabilistic data structures which consume less memory [9, 8, 5, 3]. However, we assert that these techniques are unnecessary with the rise of distributed, in-memory computing frameworks that scale easily on commodity hardware [12].

In this paper, we present a distributed algorithm for performing read error correction, which is built using the ADAM libraries for distributed genomics [6]. Our algorithm trains a model for read errors using the abundance spectrum of quality score weighted k-mers (known as q-mers [4]). Given these probabilities, we estimate transition probabilities across classes which contain correlated errors. Finally, per read, we perform error correction via a coordinate ascent process. As our algorithms are implemented on top of the Apache Spark in-memory MapReduce system [12], we are able to achieve both scale up and scale out performance. Our software is open source under the Apache 2 license.

We make the following contributions:

1. We implement a MapReduce based, in-memory k/q-mer counting engine.
2. We demonstrate a probabilistically rigorous, generative model for read error correction.
3. Our model is implemented in a scalable fashion on an efficient, MapReduce engine.

2 Background and Related Work

3 Implementation

3.1 Distributed *k*-mer Counting

On top of a MapReduce platform, k-mer counting reduces down to the classical "word count" problem. Our k-mer counting implementation uses the following simple code:

```
def adamCountKmers(kmerLength: Int): RDD[(String, Long)] = {
  rdd.flatMap(r => {
    // cut each read into k-mers, and attach a count of 1L
    r.getSequence
        .toString
        .sliding(kmerLength)
        .map(k => (k, 1L))
  }).reduceByKey((k1: Long, k2: Long) => k1 + k2)
}
```

This MapReduce job is implemented on top of the Apache Spark platform, which provides the resilient distributed dataset (RDD) abstraction [11]. This abstraction presents a view of an array which is distributed as partitions across machines. In the code above, we first perform a `flatMap` transformation which cuts reads into k-mers of multiplicity 1. These intermediate k-mers are then persisted in memory as an intermediate result. We then perform a `reduceByKey`, where the k-mer string is used as the key. This stage opportunistically performs the reduction on local keys, before globally "shuffling" data between machines.

As noted in the rest of the paper, we use "q-mers", which were introduced by Kelley et al [4]. q-mers represent quality score weighted k-mers. Thus, while the multiplicity of a k-mer seen in a read is 1, the multiplicity of a q-mer is bounded between $[0, 1]$. The implementation of our q-mer counter is a slight modification of our k-mer count implementation. Conceptually, the q-mer counting process provides expectations for the *true* counts of all k-mers, which makes it highly useful for detecting erroneous k-mers.

3.2 Erroneous *k*-mer Detection

Given the q-mer spectrum, we apply an unsupervised clustering model to learn the classifications of q-mers. For a sample with ploidy of m, we model q-mers as being drawn from a selection of $m + 1$ Gamma distributions. $m - 1$ of these distributions model "true" q-mers which are drawn from heterozygous sites, one distribution models "true" q-mers drawn from homozygous sites, and one distribution models erroneous q-mers. While prior work by Kelley et al. [4] has used the Normal distribution to model the counts of "true" q-mers, it is preferable to use the Gamma distribution, whose support is limited to $[0, \infty)$.

We describe the process we use for counting q-mers in §3.1. Once we have counted q-mers, we then fit our mixture model using the expectation-maximization (EM) algorithm introduced by Almhana et al. [1] We considered the optimized algorithm for fitting mixtures of Gammas that was proposed by Schwander and Nielsen [7], but chose the Almhana EM algorithm because the Schwander k-MLE algorithm is slower for mixtures that contain fewer than eight components. We describe the implementation of this algorithm in §3.3.

Once the mixture model has been fit, each k-mer can be assigned an error probability. This probability is given in equation (1), and is defined by the soft-max likelihood that a k-mer is drawn from the error distribution, given the q weight of the k-mer.

$$P(k \text{ is erroneous}) = \frac{\mathcal{L}(s = 0|k)}{\sum_{i=0}^{m+1} \mathcal{L}(s = i|k)} \tag{1}$$

Once the error probabilities have been calculated for all k-mers, we then filter the set of k-mers to obtain the set of trusted k-mers. We apply the following filters, and place all k-mers that satisfy these filters into a trie:

1. The k-mer is kept if it's error probability is below a user provided error threshold, and
2. The k-mer does not contain any IUPAC disambiguation codes.

3.3 Distributed EM Algorithms

The Spark MapReduce platform was designed to enhance the performance of iterative MapReduce algorithms, which are commonplace in machine learning and related disciplines [12]. Spark achieves this by providing the ability to hold datasets in memory between iterations. This improves performance when compared to Hadoop, where intermediate results must be spilled to disk after every map phase. Through the effective use of caching, we are able to implement high performance, distributed EM on top of Spark, without the use of any approximations.

As mentioned in §3.2, we use the EM algorithm for mixtures of Gamma distributions which was introduced by Almhana et al [1]. To fit a mixture of $i \in [0, \ldots, n-1]$ Gamma distributions with shape α_i and rate (inverse of scale) β_i to n points, we perform the following M updates during step t:

$$\pi_i^{(t+1)} = \frac{1}{n} \sum_{j=1}^{n} p(i|x_j, \Theta^{(t)}) \tag{2}$$

$$\beta_i^{(t+1)} = \frac{\alpha_i^{(t)} \sum_{j=1}^{n} p(i|x_j, \Theta^{(t)})}{\sum_{j=1}^{n} x_j p(i|x_j, \Theta^{(t)})} \tag{3}$$

$$\alpha_i^{(t+1)} = \alpha_k^{(i-1)} + \frac{1}{t} G_{\alpha_i}(X, \Theta^{(t)}) \tag{4}$$

The $G_{\alpha_i}(X, \Theta^{(t)})$ expression is defined as follows:

$$G_{\alpha_i}(X, \Theta^{(t)}) = \frac{1}{n}\sum_{j=1}^{n}[\log(x_j) - \log(\beta_i^{(t)}) - \psi(\alpha_i^{(t)})]p(i|x_j, \Theta^{(t)}) \tag{5}$$

We initialize our distributions before running EM by running k-means, using the MLLib/MLI implementation of k-means [10]. We associate each point to the cluster centroid it is closest to, and compute the mean (μ) and standard deviation (σ) of the points associated with each centroid. Given the μ and σ per centroid, we initialize a Gamma distribution by solving $\alpha = \frac{\mu}{\sigma}$ and $\beta = \frac{\alpha}{\mu}$. In pseudocode, our distributed implementation of EM is as follows:

Algorithm 1 Run Distributed EM

$data \leftarrow$ input dataset
$\Theta \leftarrow$ initial distribution parameters in mixture
$\pi \leftarrow$ initial distribution weights
Require: $data$ is cached
$iteration \leftarrow 1$
repeat
  $pointProbabilities \leftarrow data$.map$(x_j \Rightarrow p(i|x_j, \Theta, \pi) \forall i)$
  $pointProbabilities$.cache()
  $nonNormalizedWeights \leftarrow pointProbabilities$.aggregate$((a, b) \Rightarrow a + b)$
  $weightsAndPoints \leftarrow pointProbabilities$.zip$(data)$
  $pointProbabilities$.uncache()
  $weightsAndPoints$.cache()
  $betaDenominator \leftarrow weightsAndPoints$.map$((\tau_j, x_j) \Rightarrow x_j\tau_j)$
    .aggregate$((a, b) \Rightarrow a + b)$
  $G \leftarrow weightsAndPoints$.map$((\tau_j, x_j) \Rightarrow \tau_j^T[\log(x_j) - \log(\beta_i) - \psi(\alpha_i), \forall i])$
    .aggregate$((a, b) \Rightarrow a + b)$
  $weightsAndPoints$.uncache()
  $\pi \leftarrow$ softmax$(nonNormalizedWeights)$
  for all $(\alpha, \beta) \in \Theta, g \in G, b \in betaDenominator$ **do**
    $\beta \leftarrow \frac{\alpha nonNormalizedWeight}{b}$
    $\alpha \leftarrow \alpha + \frac{g}{iteration}$
  end for
  $iteration \leftarrow iteration + 1$
until converged
return Θ

During this algorithm, we calculate the expected complete log likelihood (ECLL) of the mixture. We run this algorithm until either a maximum iteration limit has been reached, or until we are no longer making "significant" improvements in the ECLL. We provide the user with the ability to set the early termination cut-off threshold for ECLL improvement.

3.4 Base Transition Probability

Due to well-known biases in the sequencing process, the four nucleotides have different probabilities of being sequenced incorrectly. Errors are known to be correlated within specific nucleotides, the position of the base in the read, and empirical quality score values. These three *covariates* are used in the GATK's base quality score recalibration (BQSR) process which measures the "true" error probability that corresponds to a base quality score [2].

While base quality scores are a useful prior for the GATK's main goal (specifically, variant calling), their binary error/success nature is not as useful of a prior for read error correction. Instead, since we are trying to predict whether a true base b was incorrectly sequenced as b', where $b, b' \in \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$[1], we desire our prior to be the transition probabilities from $b' \to b$. These transition probabilities can be represented by categorical random variables with four categories. We want to estimate these distributions per error covariate, where the error covariate is defined by the sequenced base, the empirical base quality score, and the position of the base in the read.

The transition probability for error covariate $C_{b',i,q}$ can be estimated via maximum likelihood. Specifically, for each base b' sequenced with quality q at position i in the read, we can define the likelihood that this base was actually b by looking at the error probabilities of all k-mers that cover this base.

$$\mathcal{L}(b) = \frac{\prod_{j=1}^{k} p(b|b', i, q) P(\mathrm{kmer}_j^{b,i} \text{ is true})}{\sum_{\hat{b}=\{A,C,G,T\}} \prod_{j=1}^{k} p(\hat{b}|b', i, q) P(\mathrm{kmer}_j^{\hat{b},i} \text{ is true})} \tag{6}$$

In equation (6), we use $\mathrm{kmer}_j^{b,i}$ as a function that appropriately substitutes base b into the jth k-mer that covers the base at position i. Also, note in practice that bases at the start and end of the read will not be covered by k k-mers; for example, the first base of each read is covered by a single k-mer. The $P(k\text{-mer is true})$ probabilities are evaluated using the trie collected by the erroneous k-mer detection process described in §3.2. We note that invalid k-mers are not present in that tree; when we search for a k-mer that is not resident in the trie, we substitute the user provided probability cutoff as an upper bound on the truth probability of that k-mer.

To achieve a most accurate estimate for the transition probabilities of error covariate $C_{b',i,q} \to p(b|b', i, q)$, we would ideally run an EM algorithm over all bases in $C_{b',i,q}$. However, this is not tractable as within a read we cannot separately evaluate the transitions for bases $i-1$, i, and $i+1$. To run such an EM algorithm, we would need to evaluate all transitions for all bases in all reads, which would be computationally intractable; we discuss this in more detail in §3.5. Instead, we estimate the transition probabilities by treating the probability estimates of all bases in $C_{b',i,q}$ as the expected outcomes from multinomial

[1] In practice, the observed base, b', may be represented by any of the IUPAC base disambiguation codes. While we handle these cases in our implementation, we do not include them here for notational simplicity. The corrected base, b, is constrained to not be a disambiguation code.

trials, and apply an MLE estimator. This is equivalent to running a single expectation phase of the canonical EM algorithm for a mixture of categorical random variables.

3.5 Read Refinement

From §3.4, we have derived the likelihood equations for base calls (equation (6)), and an algorithm for finding the priors for bases in a given error covariate $C_{b',i,b}$. Given the base likelihood equation and the prior probabilities, we would like to make corrections to low quality base calls such that we maximize the probability of each read. We define the probability of each read as:

$$P(\mathbf{r}|\mathbf{r}') = \prod_{i=1}^{l} \mathcal{L}(b = r_i) \tag{7}$$

Where $\mathcal{L}(b)$ is defined in (6). If we consider $\mathbf{r}$ and $\mathbf{r}'$ to be vectors such that each base of the read exists in it's own coordinate plane, error correction can be performed via coordinate ascent in the read space. We define the following recursive algorithm to perform coordinate ascent, where we attempt to correct the read, and only accept this correction if it improves the probability that the current read sequence is correct:

Algorithm 2 Error Correction via Coordinate Ascent

```
read ← current read sequence
p_read ← current read probability
τ ← current priors
candidates ← sorted list of bases which are candidates for correction
if candidates is empty then
  return (read, τ)
else
  candidate ← candidates.first
  newRead ← correctRead(read, candidate)
  for all i ∈ read do
    if k-mers from i overlap candidate then
      τ'_i ← p(b|read_i)τ_i^b, ∀b ∈ {A, C, G, T}
    else
      τ'_i ← τ_i
    end if
    p'_read ← probability(τ')
    if p'_read < p_read then
      return recurse(read, p_read, τ, candidates.dropFirst)
    else
      return recurse(newRead, p'_read, τ', computeNewCandidates(newRead, τ'))
    end if
  end for
end if
```

For the first iteration of the algorithm, we use the base transition probabilities from §3.4 as prior probabilities when computing base likelihoods. After the first iteration, we use the likelihoods from the last accepted step as the priors for the next step. At each step, we define a base as a candidate for correction if the probability that the base is the "true" base at that site in the read is below a user defined probability (default is Phred 10 $\rightarrow p(b) < 0.9$), and if we have not tried this correction *since* the last accepted correction.[2] As such, we only need to calculate the correction candidates on the initial iteration, and after every accepted iteration.

Once the correction algorithm has completed, we perform a final two steps:

1. We convert the final base probabilities from the coordinate ascent algorithm into Phred-scaled base quality scores, and
2. We trim low quality bases off of the start and end of the read, where "low quality" is defined by a user provided threshold (default is Phred 10).

We only attempt to correct a single base per iteration. As mentioned in §3.4, it is intractable to exhaustively search for correction candidates. This is because a sequence of length l has $3\binom{l}{e}$ correction candidates if we allow up to e errors to be corrected in a single step. The constant factor of 3 arises because we can correct each base to any of the other four bases.

4 Results

4.1 Assembly Accuracy

4.2 Variant Calling Accuracy

4.3 Performance and Scaling

5 Conclusion

5.1 Availability and Reproducibility

This read error correction software is made available as part of the `ADAM` libraries and command line interface (CLI) [6], which are open source under the Apache 2 license. Our source code is available on GitHub[3]. Additionally, we distribute our libraries through the Apache Maven dependency system's central repository, with group ID `org.bdgenomics.adam` and artifact ID `adam-core`.

Our tests were run on the Amazon Elastic Compute (EC2) cloud. The scripts used to run our performance and accuracy tests are publicly accessible via version control on GitHub in the `bdg-recipes`[4] repository, and are tagged with the `adam-ec-recomb-15` tag.

[2] To prevent oscillating corrections, we also do not allow bases that have been previously edited to be considered as candidates.

[3] https://www.github.com/bigdatagenomics/adam

[4] https://www.github.com/bigdatagenomics/bdg-recipes

References

1. Almhana, J., Liu, Z., Choulakian, V., McGorman, R.: A recursive algorithm for gamma mixture models. In: IEEE International Conference on Communications (ICC '06). vol. 1, pp. 197–202. IEEE (2006)
2. DePristo, M.A., Banks, E., Poplin, R., Garimella, K.V., Maguire, J.R., Hartl, C., Philippakis, A.A., del Angel, G., Rivas, M.A., Hanna, M., et al.: A framework for variation discovery and genotyping using next-generation DNA sequencing data. Nature genetics 43(5), 491–498 (2011)
3. Heo, Y., Wu, X.L., Chen, D., Ma, J., Hwu, W.M.: BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads. Bioinformatics p. btu030 (2014)
4. Kelley, D.R., Schatz, M.C., Salzberg, S.L., et al.: Quake: quality-aware detection and correction of sequencing errors. Genome Biology 11(11), R116 (2010)
5. Liu, Y., Schröder, J., Schmidt, B.: Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data. Bioinformatics 29(3), 308–315 (2013)
6. Massie, M., Nothaft, F., Hartl, C., Kozanitis, C., Schumacher, A., Joseph, A.D., Patterson, D.A.: ADAM: Genomics formats and processing patterns for cloud scale computing. Tech. rep., Technical Report UCB/EECS-2013-207, EECS Department, University of California, Berkeley (2013)
7. Schwander, O., Nielsen, F.: Fast learning of gamma mixture models with k-MLE. In: Similarity-Based Pattern Recognition, pp. 235–249. Springer (2013)
8. Shi, H., Schmidt, B., Liu, W., Müller-Wittig, W.: 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)
9. Song, L., Florea, L., Langmead, B.: Lighter: fast and memory-efficient error correction without counting. bioRxiv (2014)
10. Sparks, E.R., Talwalkar, A., Smith, V., Kottalam, J., Pan, X., Gonzalez, J., Franklin, M.J., Jordan, M.I., Kraska, T.: MLI: An API for distributed machine learning. In: 2013 IEEE 13th International Conference on Data Mining (ICDM '13). pp. 1187–1192. IEEE (2013)
11. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI '12). p. 2. USENIX Association (2012)
12. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10). p. 10 (2010)