avocado: A Variant Caller, Distributed

Frank Austin Nothaft, Peter Jin, Brielin Brown
Department of Electrical Engineering and Computer Science
University of California, Berkeley
{fnothaft,phj,brielin}@berkeley.edu

ABSTRACT

In this paper, we present `avocado`, a distributed variant caller built on top of ADAM and Spark. `avocado`'s goal is to provide both high performance and high accuracy in an open source variant calling framework. To achieve this, we implement both local assembly and pileup-based single nucleotide polymorphism (SNP) calling. A key innovation presented in our work involves the development of heuristics for when to choose more expensive assembly-based methods instead of pileup-based methods. Additionally, we introduce the concept of "significant statistics," a tool for performing incremental joint variant calling.

Categories and Subject Descriptors

Applied Computing [**Life and Medical Sciences**]: Computational Biology—*Sequencing and Genotyping Technologies*; Applied Computing [**Genomics**]: Computational Genomics; Computing Methodologies [**Distributed Computing Methodologies**]: Distributed Algorithms—*MapReduce Algorithms*

General Terms

Algorithms, Performance

Keywords

Variant Calling, Genotyping, Genomics Pipeline, Local Assembly, Distributed Computing, MapReduce

1. INTRODUCTION

Modern genomics processing pipelines can be divided into four primary ordered stages:

1. **Sequencing:** Gathering of read data from DNA
2. **Alignment:** Alignment of read data against reference genome
3. **Variant Calling:** Statistical determination of differences against reference genome
4. **Variant Annotation:** Annotation of impact of variation

Currently, to run a genomics pipeline end-to-end for a single high coverage genome[1] consumes approximately 100 hours [?]. Of this 100 hour figure, both alignment and variant calling consume approximately 50 hours each.

Although some applications that use genomic data are latency insensitive (for example, population genomics), many medical applications like genomic medicine, or genomic classification of viral outbreaks [?] are latency sensitive. However, it is unacceptable to sacrifice accuracy in the pursuit of speed. Recent work has focused on the problem of accelerating alignment [?]; in this paper, we address accelerating variant calling.

As noted above, it is unacceptable to sacrifice accuracy for performance. To achieve improved performance, we implement several enhancements:

- Current pipelines are penalized by I/O performance, we address this by using an in-memory MapReduce framework [?] to reduce I/O pressure
- Additionally, we leverage the new ADAM data format [?], a high performance file format for distributed genomics
- Finally, we achieve high accuracy at a low performance cost by using high fidelity assembly-based methods only on complex segments of the genome

In this paper, we discuss this system, related work, and perform a performance analysis. We start with a discussion of the related work in §2. We then describe our architecture and algorithms in §3. Finally, we analyze the performance of our system in §4, and propose future research directions in §5.

[1]High coverage refers to having on average $>30\times$ bases aligned to each location in the reference genome.

2. RELATED WORK

There has been significant work related to variant calling, and towards accelerating the genomic processing pipeline. In this section, we discuss other variant calling pipelines, and tools that we use in our evaluation.

2.1 ADAM

ADAM [?] is a new data format for genomics meant to replace the Sequence/Binary Alignment/Map (SAM/BAM) formats for read data [?], and the Variant Call Format (VCF) for variant/genotype data [?]. The original SAM/BAM/VCF formats were designed for single-node processing, and do not easily distribute across several machines. Although a library was designed for processing BAM/VCF data in Hadoop [?], this API does not scale well past 8 nodes. ADAM achieves scalability beyond 100 machines by eliminating the central file header, and by using the Parquet data store which is optimized for parallel data access [?].

In the process of developing `avocado`, we contributed 2,500 lines of code (LOC) to the ADAM project. This contribution comprised the variant and genotype format, code for calculating normalized variant data from genotypes, and converters to/from the VCF format.

2.2 The Genome Analysis Toolkit

[?, ?]

2.3 Samtools Mpileup

[?]

2.4 FreeBayes

[?]

2.5 SNAP

2.6 SMaSH

3. ARCHITECTURE

3.1 Local Assembly

Given a partition of reads, we can group them by their starting locus in intervals of W, creating regions of length $W + L - 1$ where L is the read length. Within each region, we can evaluate the likelihood of observing the reads given the reference haplotype:

$$\begin{aligned}\mathcal{L}(H^{\text{ref}}) &\equiv \mathbf{P}(\{r_i\}|H^{\text{ref}}) \qquad (1)\\ &= \prod_i \mathbf{P}(r_i|H^{\text{ref}})\end{aligned}$$

where $\mathbf{P}(r|H)$ is obtained from aligning the read and the candidate haplotype by a pairwise HMM alignment. Note that, in practice, all probabilities are computed in units of logarithm base 10, so products become sums, etc.

If a reference haplotype likelihood is below a fixed threshold, the region corresponding to the haplotype is marked *active*. Each *active region* is assembled independently and in parallel.

The assembly of an active region starts by splitting all reads assigned to the region into k-mers, where k is a fixed parameter for all assemblies. A read generates $L - k + 1$ total k-mers. Each k-mer is uniquely identified by the substring of its originating read. Because of coverage overlap and sequence repeats, some k-mers will be duplicates; these are consolidated, and the duplication factor is recorded as the k-mer multiplicity.

The k-mers define edges in the completed k-mer assembly graph. Within a read, each adjacent pair of k-mers have an overlapping substring of length $k-1$; these are seeded as the initial vertices in the k-mer graph. Because there are duplicated k-mers, some vertices will be "merged," connecting the graph. Unlike an exact de Bruijn graph, which connects all overlaps between k-mers, we only connect the overlaps found in the reads, performing a simple form of read threading.

Once the k-mer graph is complete, we perform a depth-first traversal with an upper bound on the total path multiplicity, defined as the sum of the edge multiplicities, to enumerate a set of possible paths. The traversal begins at a graph source, and a completed path must also end at a sink. Each path is an assembled haplotype to be evaluated.

Since assembly depends on the exact sequences of the input data, the quality of the reads is critical for performing an accurate assembly. One error correction method is spectral filtering, which depends on the observation that the distribution of k-mers from all reads with respect to multiplicity is bimodal, one due to the Poisson sampling of reads (with high multiplicity), the other due to errors (with low multiplicity), so that splitting the reads between the two modes and keeping the reads near the mode with higher multiplicity serves to prune the poor quality reads. Errors in the read data lead to *spurs*, which are spurious short sections of the graph connected to the source or the sink. We do not remove spurs, although it may slightly improve performance. Empirically, we have found during other work that utilizing mate pair information greatly improves the quality of an assembly. We also do not employ mate pair threading, which requires collecting the insert size distribution of the data, but we expect that implementing it has the potential to vastly improve the accuracy of variant calls.

From the assembled haplotypes, we order them according to the haplotype likelihood:

$$\mathcal{L}(H_j) = \prod_i \mathbf{P}(r_i|H_j). \qquad (2)$$

Among the ordered haplotypes, we pick the top scoring haplotypes and ignore the low scoring ones. The likelihood of observing the reads $\{r_i\}$, given a pair of haplotypes H_j and $H_{j'}$, is defined to be:

$$\begin{aligned}\mathcal{L}(H_j, H_{j'}) &\equiv \mathbf{P}(\{r_i\}|H_j, H_{j'}) \qquad (3)\\ &= \prod_i \left[\frac{\mathbf{P}(r_i|H_j)}{2} + \frac{\mathbf{P}(r_i|H_{j'})}{2}\right].\end{aligned}$$

We compute the posterior probability of observing the pair of haplotypes H_j and $H_{j'}$ from the haplotype pair likelihood and a haplotype pair prior probability:

$$\mathbf{P}(H_j, H_{j'}|\{r_i\}) = \frac{1}{Z}\mathcal{L}(H_j, H_{j'})\mathbf{P}(H_j, H_{j'}) \qquad (4)$$

where Z is a normalization:

$$Z = \sum_{j} \sum_{j'} \mathcal{L}(H_j, H_{j'}) \mathbf{P}(H_j, H_{j'})$$

and where we obtain the prior $\mathbf{P}(H_j, H_{j'})$ by aligning the haplotype pair with the same pairwise HMM alignment as above, and taking the product of the prior probabilities for each SNP and indel event.

We choose the maximum a priori estimate among haplotypes with any variants as the called non-reference maternal and paternal haplotype pair:

$$(H_{\text{mat}}^{\text{nonref}}, H_{\text{pat}}^{\text{nonref}}) = \arg \max_{H_j, H_{j'} : n_{\text{var}}(H_j, H_{j'}) > 0} \mathbf{P}(H_j, H_{j'} | \{r_i\}). \tag{5}$$

Similarly, we may define the reference haplotype pair as $(H^{\text{ref}}, H^{\text{ref}})$. The error probability of calling the non-reference haplotype pair is:

$$\begin{aligned} &\mathbf{P}_{\text{error}}(H_{\text{mat}}^{\text{nonref}}, H_{\text{pat}}^{\text{nonref}}) \qquad (6) \\ &= \frac{\mathbf{P}(H^{\text{ref}}, H^{\text{ref}})}{\mathbf{P}(H_{\text{mat}}^{\text{nonref}}, H_{\text{pat}}^{\text{nonref}}) + \mathbf{P}(H^{\text{ref}}, H^{\text{ref}})}. \end{aligned}$$

The quality score of all variants present in the nonreference haplotype pair is defined as the Phred scaling of $\mathbf{P}_{\text{error}}$.

3.2 SNP Calling

3.3 Joint Variant Calling

3.4 Algorithm Selection

4. EVALUATION

4.1 Accuracy

4.2 Performance

5. FUTURE WORK

6. CONCLUSION

APPENDIX

A. AVAILABILITY

`avocado` is open source and is licensed under the Apache 2 license. The source code is available at:

`http://www.github.com/bigdatagenomics/avocado`