AVOCADO: A Variant Caller, Distributed

Peter Jin, Brielin Brown, Frank Austin Nothaft

Abstract

Although the accuracy of modern variant callers is improving, their performance is still poor. We propose a variant caller that is designed for distributed data systems that should achieve greatly improved performance, without sacrificing accuracy. From a computer systems perspective, this variant caller will improve performance by employing efficient data serialization/deserialization methods, data structures that can easily be accessed and filtered, and in-memory processing techniques that scale well to large compute clusters. We will improve the performance and accuracy of our variant calling by using filtering to select the optimal algorithm to use for local segments of the genome. Data filtering allows us to use less computationally intensive algorithms when processing low-complexity segments of DNA, and to choose algorithms that are optimized for specific structural variants that we encounter.

1 Introduction

Enabled by the rapid drop in the cost of sequencing a genome [8], many fields of medicine are embracing genomic data in both research and treatment practice. After this data is collected, it must be processed through an alignment and variant calling pipeline. Although the performance of current variant callers like SAMtools [4] and GATK [7] is improving, it is not maintaining pace with the increased performance and reduced cost of sequencing tools. We will add more detail to this section as we implement the algorithm.

We propose a new variant caller architecture that is easy to extend, and that is optimized for high performance and scalability. We employ efficient data serialization/deserialization techniques [6] and an in-memory distributed computing architecture [15] to reduce I/O costs during variant calling. Additionally, we leverage the filtering techniques introduced by GQL [3] to eliminate irrelevant data and reduce the size of the dataset that we are processing. We believe that data filtering will not have an adverse impact on accuracy, and may even reduce the number of spurious variant calls; we discuss the rationale for this further in §2.2. A significant advantage enabled by filtering is that we can tailor the algorithm we are using to call variants to the complexity of the adjacent location in the genome. This allows us to use less computationally expensive algorithms to call variants in low complexity areas. We discuss the heuristics that we will use for filtering in the subsections of §3.

Our project will contribute the following to the state-of-the-art in variant calling:

- The use of extensive data filtering operations that allows us to select local assembly (§3.1) for variant calling in complex regions and to select less computationally complex algorithms in other regions. This will improve system performance, and will hopefully not impact accuracy.
- The introduction of sufficient statistics for joint variant calling (§4). This technique allows us to replace repetitive computation with a database access.
- The introduction of the new ADAM data formats (§2.1). These formats present significant improvements over the SAM/BAM/VCF formats in performance, size, and semantics and will enable improved distributed system scalability.
- The creation of a variant calling pipeline that is both modular and open source. The current state-of-the-art variant calling toolchain (GATK) is mixed-source and does not have peer-reviewed literature available on all of its algorithms.

2 Architecture

The system architecture is described in figure 1. This architecture is designed specifically to enable the use of multiple different variant calling algorithms that are tailored to specific forms of structural variants (e.g. single SNP, short indel, long indel). To achieve this, our system contains a fork that allows us to use both read-based and pileup-based variant calling algorithms.

Logically, the system can be decomposed into three functions: input/output, data filtering, and variant calling. The rest of this section will give an overview of each of these three functions. The algorithms that support

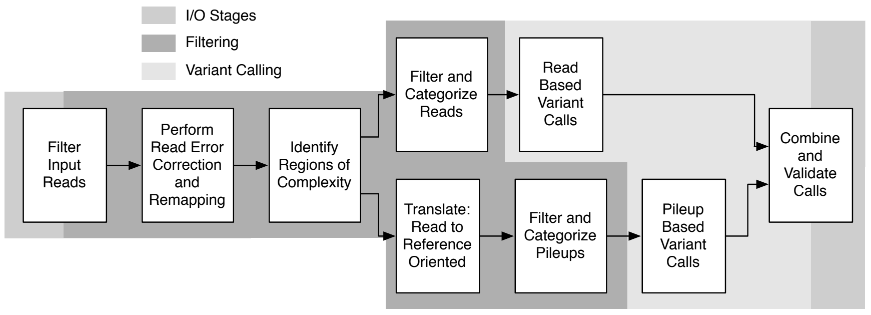
I/O Stages
Filtering
Variant Calling
Filter Input Reads
Perform Read Error Correction and Remapping
Identify Regions of Complexity
Filter and Categorize Reads
Read Based Variant Calls
Translate: Read to Reference Oriented
Filter and Categorize Pileups
Pileup Based Variant Calls
Combine and Validate Calls

Figure 1: System architecture

filtering and variant calling are not discussed in this section — instead, we will discuss them further in §3. We will also discuss how the architecture of the system lends itself to easy extension.

2.1 Input/Output

Although most variant calling tools take Sequence Alignment/Map (SAM[1] files as input and return Variant Call Format (VCF) files as output, we will use ADAM instead. ADAM provides several data structures for genomics that are implemented under Avro and Parquet. Parquet is a columnar data store that stores data in compressed form. When deserializing data, Parquet allows us to use predicates to only load relevant data. When these predicates are being used, Parquet only decompresses the relevant field that must be checked — if the predicate identifies the record, Parquet then decompresses the full record. This predicate pushdown allows us to perform basic filtering when reading in the aligned reads (e.g. only load reads whose quality is above a certain threshold), and should reduce the amount of I/O that we are performing.

We gain several additional advantages through the use of Avro. Avro allows us to define data structures using a markup language. Once the data structure has been defined using the markup, it can be reused across most major programming languages. By defining a straightforward data structure that is easily serialized and deserialized, we also eliminate the overhead of needing to parse SAM/BAM files. We plan to update SNAP to

[1]SAM is a plain-text based file format.In practice, most tools use the Binary Alignment/Map (BAM) format. See Li et al.[5] for a discussion of the file formats.

support output using ADAM datatypes; in the meanwhile, we can convert SAM/BAM files to ADAM through the use of the ADAM toolkit [6].

A notable feature that we are implementing in ADAM for AVOCADO is pileup compression. Normally, reference oriented data is stored in a conservative manner that retains full statistics for each base at the reference position. Instead, we plan to reduce the data for like bases down into a single record. As a critical application for this variant caller is high-coverage ($60\times$) cancer genomes, this may provide a significant reduction in the amount of data that needs to be distributed in our system, and may lead to significant performance improvement.

2.2 Data Filtering

One of the critical findings of the GQL project was that we can achieve a significant speedup in variant calling by eliminating data that is unlikely to contain a structural variant [3]. In AVOCADO, we look to extend this approach further by treating data filtering and segregation as a key part of successful variant calling. This work is predicated under several assumptions:

- With the exception of 10% of the genome that contains high redundancy, structural variants can be called without using based assembly techniques [12]. For these remaining structural variants, there is a negligible accuracy tradeoff that comes from using algorithms that increase statistical rigor at the cost of computation.
- Within the remaining 90% of the genome, the majority of loci do not contain evidence of structural variants and can be discarded without significantly impacting accuracy [3].
- For the remaining regions that may contain evidence of variants, we can determine what sort of structural variant they are likely to contain (SNP, short index), and optimize both accuracy and computational cost by picking algorithms tailored to those specific structural variants.

We provision three stages for the filtering of data. As alluded to in §2.1, we perform basic filtering on the input reads. This first round of filtering will be based on the read quality. The second round of filtering will look to identify segments of the genome which need to be called using read-assembly based methods. After this round of filtering, we segregate the genome into two pipelines — one for read-based identification, and another for pileup-based variant identification. In each of these pipelines, we perform a third round of filtering that serves to pair reads/pileups with the algorithm that is most likely to successfully identify their structural variant.

It is useful to note that data filtering may actually improve the accuracy of the variant calling system. In the preliminary work done on the BIGGIE system, the authors found that filtering on complexity led to them making fewer incorrect SNP calls [13]. This conclusion has logical backing: by looking for patterns in the base pair data, we can match pileups/reads that exhibit evidence for a specific form of variant with algorithms that are optimized for correctly detecting those variants.

2.3 Variant Calling

As discussed in §2.2, we plan to include several variant calling algorithms. At the moment, we plan to implement the algorithms identified in Table 1. A further discussion of these algorithms can be found in §3.

Table 1: Variant Calling Algorithms

Read Based	**Reference Based**
Local Assembly	SNP Calling
Long Indel	
Short Indel	

A consequence of having several different algorithms is that we will need to join the output of all algorithms at the end. A naïve approach is to pick the result generated by the "most rigorous" variant calling algorithm. However, since the purpose of the filtering steps discussed in §2.2 is to send disparate segments of the genome to different variant calling algorithms, there should not be any conflicting variant calls. More succinctly, if VC_k is the set of all variant calls made by algorithm k, we expect $VC_i \cap VC_j = \emptyset$ for all combinations of i, j.

2.4 Extensibility and Scalability

One of the inherent goals of this project is to produce a variant caller that has well defined interfaces and can easily be extended. We hope that this will enable the rapid development and prototyping of future variant calling algorithms. From a software engineering standpoint, we will achieve this by presenting clear interfaces for defining new filters and variant calling algorithms for both read and pileup oriented data.

3 Filtering and Calling Heuristics

3.1 Local Assembly

Local assembly will be used in areas of high genomic complexity[2] to perform very accurate structural variant calling. It is useful to note that this the highest accuracy method used in the GATK's HaplotypeCaller.

Unfortunately, there is little published information about the specific algorithm used by HaplotypeCaller, making it difficult to reproduce in our system for comparison purposes. We hope to gain some insight into their methods via our discussion with Broad engineers. There are several local assembly algorithms that can be used in this step, for instance the MULTIBRIDGING algorithm of Bressler [1], and the active region methods presented by Carnevali et al [**?**]. We intend to explore the tradeoffs between different local assembly algorithms, perhaps choosing the best one based on criteria such as read length, coverage depth, and repetitiveness. We hope to arrive at an algorithm that blends the strong points of already known algorithms with more advanced heuristic k-mer graph techniques like read threading, mate pair threading, and spur removal.

There are several heuristics that can be used for identifying high genomic complexity. A primary heuristic is read mapping quality — if there are many reads that are poorly mapped, this can indicate that the location has high similarity to other regions in the genome and is hard to differentiate, or that the reads are mapping poorly to the reference due to a structural anomaly in the sequenced genome. As a base heuristic, we will look for sliding windows of n-bases with low Phred scores. These parameters can be adjusted by the user — as defaults, we can set $n = 150$ and Phred $= 30$.

3.2 Long Indel

For long deletion detection, we may be able to implement an algorithm similar to the split-read[3] based algorithm implemented by BreakDancer [2]. This algorithm provides highly accurate detection of long deletions, at a lower computational cost than local assembly. However, currently SNAP does not perform split-read alignment [14]. Later in the project, we may look to extend SNAP, or to evaluate the algorithm with aligned reads from an aligner that performs split-read based alignment.

[2]These can be areas with long sequence repeats, areas that are redundant/repeated in the genome, or areas with large structural variants.

[3]Reads where there is no contiguous alignment/mapping.

3.3 Short Indel

Here, we define a short indel as an insertion or deletion less than 100 base pairs long. As indicated in Chen et al. [2], the Smith-Waterman sequence alignment algorithm (see [9]) can be used for accurately identifying short indels. To call these variants, we perform a consensus Smith-Waterman:

1. Using primary read alignments, find areas with evidence of short indels.
2. Find all reads with a primary alignment that overlaps this area, along with an additional padding of w. The total overlapping region will be from $b - w$ to $e + w$, where b, e are the beginning and end of the area where there is evidence for an indel.
3. Per read, create a scoring matrix against the reference (S_i).
4. Reduce all scoring matrices by performing an elemental sum, weighted by the error probability of the read (ϵ_i):
$$S[x, y] = \sum_i (1 - \epsilon_i) * S_i[x, y], \forall x, y$$
5. Perform conventional Smith-Waterman backtracking on this final scoring matrix.

As has been noted, the Ukkonen algorithm for string matching is more efficient than the Smith-Waterman algorithm [11]. This is the algorithm used in SNAP [14]. We are currently looking into the SNAP code to identify whether any algorithmic changes would need to be made to use Ukkonen's algorithm in place of Smith-Waterman. At this moment, we have not decided on a gap penalty to use. As the goal for the platform is easy extensibility (see §2.4) we will design the short indel detection module to accommodate several different gap scoring schemes including constant, affine, and profile scoring. This will also allow us to explore the gap scoring design space.

3.4 SNP Calling

To perform SNP calling, we first identify the likely genotype class[4] using equation (2) from Li et al [4]. We will filter pileups on the basis of whether

[4]For diploid human, classes include homozygous matching reference, heterozygous with one base matching reference, or no bases matching reference.

they contain any mismatches — only pileups that contain a mismatch will be evaluated. This algorithm itself can be used for both single sample or joint variant calling. If we are performing a joint call, since we are not calculating the population statistics needed for performing joint calling, we will call out to a database of population statistics described in §**??**.

One of the significant contributions of this algorithm is that we will rely on a database[5] for prior probabilities, instead of performing this calculation at runtime as is done in GATK and SAMtools [7, 4]. However, to evaluate the performance tradeoffs for this decision, we will also implement a second SNP calling algorithm that is based on equation (19) from Li et al [4]. This caller will operate on multiple samples, which will also require a change in the filter that is being used. Specifically, we will group all genomes at a specific reference value, identify sections that show SNP evidence in at least one sample, and perform the SNP call across all samples.

4 Sufficient Statistics Collection

In the current implementation of GATK and SAMTools, it is recommended that the user download sets of reads from other genomes to process with the genome of interest. The performance of both the HaplotypeCaller and mpileup improves greatly when able to jointly process multiple samples. In recent years, the number of available samples has grown to the point where there are more available samples than can be efficiently analyzed together.

As part of this work, we plan on creating a database wherein the results of computing genotype statistics on many available genomes are stored. When a new genome of interest is to be analyzed, the content of the reads covering each location will be compared against the known genomes. This will allow faster determination of whether the observed variation is true or due to sequencing error and fast fast computation of the confidence of the called genotype. If external information about the genome of interest is known, such as population (e.g. Sub-Saharan African, Northern European) or sequencing platform, this can be taken into account by comparing against genomes from the same population, and correcting likelihoods for platform based errors.

Initially, we plan on extending the work of Li et al [4]. Though GATK's methods work for SNPs and indels, the published calculations are only for SNPs. If time permits, we plan to extend the model to include both SNPs and indels, though we imagine the latter will be much more difficult.

[5]See §4.

5 Experimental Validation

We plan to validate this tool using the SMASH benchmarking suite [10]. This benchmarking suite will allow us to compare both performance and accuracy against other high quality variant calling engines.

5.1 Parallel vs. Distributed

This system can be implemented either as a traditional parallel program on a single machine, or as a distributed application running on a cluster. It is useful to note that SNAP [14] chose to run on a single machine. Our general implementation is task parallel between the read/reference-oriented calling, and also between algorithms within these two sections. For the variant calling algorithms themselves, the Short Indel and SNP callers are clearly data parallel.

Since our algorithms have a high level of parallelism, our scaling on parallel and distributed system should be determined solely by the amount of data that needs to be moved between processors/machine. We hope that the filtering steps we employ will minimize data transfer and improve scalability.

We will likely encounter a tradeoff between efficiency and performance when scaling from a parallel system running on a single machine to a distributed system running on a cluster. This opens us up to several tradeoffs that we can investigate — specific tradeoffs of interest might include finding the minimal cost configuration for variant calling, or finding the system that provides the lowest end-to-end latency (independent of cost or scaling efficiency).

5.2 Computational Cost of Joint Variant Calling

As discussed previously, we plan to calculate population statistics offline and catalog them in a database. The GATK [7] and SAMtools [4] both implement population statistics calculation as an online operation. The impact of performing joint variant calling in this manner is that if a single sample is added to a population, the entire population must be reanalyzed before calling variants on this single sample. This has significant cost for large populations, and we hope to replace this large cost that grows with population size with a single, fixed latency cost. With this feature, we will need to characterize the tradeoff at which it becomes more performant to perform offline calculation, as well as any accuracy impact of performing offline analysis. As part of this analysis, we will have to create an additional SNP caller. This is discussed in §3.4.

5.3 Order of Work

We plan to conduct our work in the following order:

- In parallel, we will implement:
  1. SNP Calling (see §3.4)
  2. Local assembly (see §3.1)
  3. Sufficient statistics calculation (see §4)
- Joint variant calling for SNPs using sufficient statistics (see §**??**)
- Short Indel detection (see §3.3)
- Long Indel detection (see §3.2) using a split-read aligner
- Enhancements to SNAP to add the ability to perform split-read alignment
- Joint variant calling for structural variants (see §**??**)

A Other Collaborators

In addition to the three students working on this project, we will be assisted by:

- Ameet Talwalkar (Postdoc in AMPLAB, statistical/machine learning advice for sufficient statistics in §4)
- Matt Massie (Software Engineering Lead in AMPLAB, creator of ADAM [6], assistance with tools/APIs for I/O in §2.1)
- Chris Hartl (Visiting Scientist in AMPLAB from MIT/Harvard Broad, advice on variant calling/genotyping algorithms)

B Resources Needed

This project will use computing resources provided through the AMPLAB including computing clusters located at Berkeley and through Amazon Elastic Compute 2 (EC2).

References

[1] Bresler, G., Bresler, M., and Tse, D. Optimal assembly for high throughput shotgun sequencing. *BMC Bioinformatics 14*, S18 (2013).

[2] Chen, K., Wallis, J. W., McLellan, M. D., Larson, D. E., Kalicki, J. M., Pohl, C. S., McGrath, S. D., Wendl, M. C., Zhang, Q., Locke, D. P., et al. Breakdancer: an algorithm for high-resolution mapping of genomic structural variation. *Nature methods 6*, 9 (2009), 677–681.

[3] Kozanitis, C., Heiberg, A., Varghese, G., and Bafna, V. Using genome query language to uncover genetic variation. *Bioinformatics* (2013).

[4] Li, H. A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data. *Bioinformatics 27*, 21 (2011), 2987–2993.

[5] Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., et al. The sequence alignment/map format and SAMtools. *Bioinformatics 25*, 16 (2009), 2078–2079.

[6] Massie, M. ADAM: Datastore Alignment Map. `http://www.github.com/massie/adam`.

[7] McKenna, A., Hanna, M., Banks, E., Sivachenko, A., Cibulskis, K., Kernytsky, A., Garimella, K., Altshuler, D., Gabriel, S., Daly, M., et al. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome research 20*, 9 (2010), 1297–1303.

[8] NHGRI. DNA sequencing costs. `http://www.genome.gov/sequencingcosts/`.

[9] Smith, T. F., and Waterman, M. S. Comparison of biosequences. *Advances in Applied Mathematics 2*, 4 (1981), 482–489.

[10] Talwalkar, A., Liptrap, J., Newcomb, J., Hartl, C., Terhorst, J., Curtis, K., Bresler, M., Song, Y., Jordan, M. I., and Patterson, D. SMASH: A benchmarking toolkit for variant calling.

[11] Ukkonen, E. Algorithms for approximate string matching. *Information and control 64*, 1 (1985), 100–118.

[12] Xia, R., Sheehan, S., Zhang, Y., Talwalkar, A., Zaharia, M., Terhorst, J., Jordan, M., Song, Y. S., Fox, A., and Patterson, D. BIGGIE: A distributed pipeline for genomic variant calling. 2078–9.

[13] Xia, R., Sheehan, S., Zhang, Y., Talwalkar, A., Zaharia, M., Terhorst, J., Jordan, M., Song, Y. S., Fox, A., and Patterson, D. BIGGIE: A distributed pipeline for genomic variant calling.

[14] Zaharia, M., Bolosky, W. J., Curtis, K., Fox, A., Patterson, D., Shenker, S., Stoica, I., Karp, R. M., and Sittler, T. Faster and more accurate sequence alignment with SNAP. *arXiv preprint arXiv:1111.5572* (2011).

[15] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), pp. 10–10.