AVOCADO: A Variant Caller, Distributed

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 [7], 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 [3] and GATK [6] 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 [5] and an in-memory distributed computing architecture [14] to reduce I/O costs during variant calling. Additionally, we leverage the filtering techniques introduced by GQL [2] 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.

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.

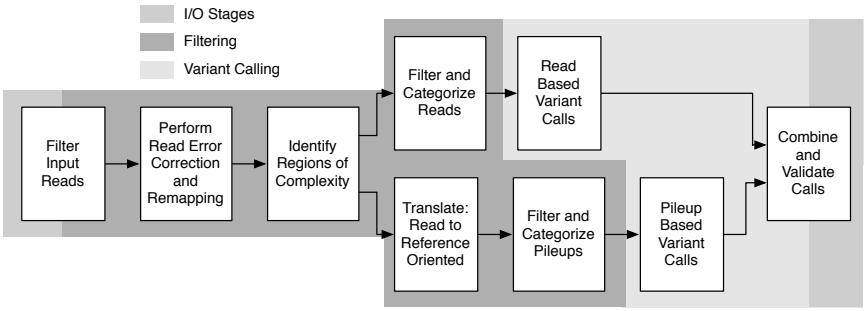

Figure 1: System architecture

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 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 support output using ADAM datatypes; in the meanwhile, we can convert SAM/BAM files to ADAM through the use of the ADAM toolkit [5].

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 [2]. 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 [11]. 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.

[1]SAM is a plain-text based file format.In practice, most tools use the Binary Alignment/Map (BAM) format. See Li et al.[4] for a discussion of the file formats.

- 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 [2].
- 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 [12]. 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.

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 [1]. This algorithm provides highly accurate detection of long deletions, at a

[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.

lower computational cost than local assembly. However, currently SNAP does not perform split-read alignment [13]. 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.

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. [1], the Smith-Waterman sequence alignment algorithm (see [8]) 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 [10]. This is the algorithm used in SNAP [13]. I am 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.

3.4 SNP Calling

To perform SNP calling, we first identify the likely genotype class[4] using equation (2) from Li et al [3]. 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.

4 Experimental Validation

We plan to validate this tool using the SMASH benchmarking suite [9]. This benchmarking suite will allow us to compare both performance and accuracy against other high quality variant calling engines.

4.1 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 [13] 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).

4.2 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)
- 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

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

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

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

[9] 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.

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

[11] 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.

[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.

[13] 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).

[14] 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.