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 [27]. 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 [26] 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 [30]; 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 [31] to reduce I/O pressure
- Additionally, we leverage the new ADAM data format [19], 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× 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 callers, and tools that we use in our evaluation.

2.1 ADAM

ADAM [19] is a new data format for genomics meant to replace the Sequence/Binary Alignment/Map (SAM/BAM) formats for read data [18], and the Variant Call Format (VCF) for variant/genotype data [8]. 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 [22], 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 [29].

In the process of developing `avocado`, we contributed 3,500 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. Additionally, this contribution included code for translating between read and reference oriented views of data.

2.2 Samtools Mpileup

Samtools Mpileup is a tool for single nucleotide polymorphism (SNP) calling and genotyping aligned read data. Given a set of reads from several individuals aligned to reference chromosomal position (a *pileup*), Mpileup determines

- Is there statistically significant evidence that some individuals in the population have a non-reference allele at this position? (SNP calling)
- Given that there is a SNP, which individuals are homozygous for the reference base, which are heterozygous, and which are homozygous for the non-reference base? (genotyping)

Since reads from a single individual will contain sequencing errors that do not represent true genetic variation, samtools leverages the alignment and read quality from several individuals, and calls a site an variant if the probability that all individuals in the sample are homozygous to the reference is small enough [16].

2.3 GATK

The Genome Analysis Toolkit (GATK) [21, 9] is a variant calling framework released by the Broad Institute of Harvard and MIT. GATK was designed for multi sample calling of SNPs and short insertions and deletions, as well as genotyping, for human genomes, and appropriately it can use existing knowledge of human genetic variants from repositories like dbSNP [24] or HapMap [28]. Originally, GATK used a purely statistical model for genotyping, along similar lines as Mpileup, although it has since moved to performing local assembly to resolve haplotypes. Architecturally, GATK consists of a parallelizable pipeline for transforming short read data into VCF output, where pipeline stages are generically called "walkers" and are invoked on the read data in a MapReduce style.

In practice, the performance of GATK suffers due to a combination of architectural and engineering deficiencies:

- Actual parallelism among GATK walkers is limited.
- There are significant overheads due to disk I/O between pipeline stages and due to poor algorithm or data structure design.
- The GATK requires expensive read processing stages like indel realignment, and base quality score recalibration.

Nonetheless, at the time we were developing `avocado`, GATK remained the state of the art in variant calling human data.

2.4 FreeBayes

FreeBayes is a variant caller that uses pileup based calling methods, and incorporates read phasing information when calling variants [11]. Instead of performing graph based assembly, the authors describe a heuristic approach that narrows down the the interval on which phasing is captured by selectively eliminating locations where alleles that have unlikely evidence appear to segregate.

FreeBayes has several advantages over other variant calling tools:

- FreeBayes does not have any limitations in terms of the ploidy, nor does it put any assumptions on the ploidy. This is important for working with samples with large deletions, and is also a significant advantage for working with plant data. Plant genomes tend to have high ploidy, and may have variable ploidy between samples. This presents a significant advantage when compared to the GATK, which assumes diploidy.
- Additionally, FreeBayes does not assume biallelism. This is an improvement over Samtools mpileup, which assumes biallelism when genotyping [16]. However, it is worth noting that there are few sites where SNPs segregate into more than two alleles [14].

FreeBayes has been indicated to have better accuracy than the GATK by researchers at Harvard, but the authors of FreeBayes do not report accuracy nor performance numbers.

2.5 SNAP

SNAP is a high performance short-read aligner that is optimized for longer read lengths, and for distributed computing [30]. At the current moment, we have not integrated with SNAP, but long term, we plan to incorporate SNAP as the aligner in our read-alignment and variant calling pipeline. This is significant, as variant callers typically optimize to correct for the error models of the aligners that they coexist with.

SNAP leverages the increasing length of reads to build a large alignment index, which is similar to the method used by BLAST [2], and which is dissimilar to the Burrows-Wheeler transform based methods used by BWA [17]. Aligners which use the Burrows-Wheeler transform perform very well in the absence of mismatching data—however, they cannot handle mismatches or inserts well. BWA has bad failure modes for reads with mismatches within the first 20 bases of the read, and Bowtie [15] does not handle insertions when aligning. As SNAP will have better performance when aligning indels, it is likely that we will be able to omit the local realignment stage from ADAM [19]—this is significant as local realignment is the most expensive phase of read processing before variant calling.

2.6 SMaSH

SMaSH is a benchmarking suite for alignment and variant calling pipelines [27], and was a key tool used for the evaluation of `avocado`.

Traditionally, variant calling pipelines have been evaluated on concordance[2], and through using venn diagrams [12]. This is because genomics rarely has access to *ground truth*: typical sequencing methods have insufficient fidelity to detect all variants clearly, and extremely high fidelity sequencing methods are too expensive/slow to be used in clinical practice. However, this method is fraught with risk: concordance is not a good metric to use if variant callers or aligners are making similar systemic errors.

To address this problem, SMaSH leverages synthetic data which by definition has known ground truth, and rigorously verified mouse and human genomes. The human genomes and validation data come from the 1000 Genomes project, a project which surveyed the genomes of 1000 individuals using multiple sequencing technologies [25]. On top of this curated data, SMaSH provides a VCF based interface for determining the precision of a variant calling pipeline. Novel to this benchmarking suite, the authors introduced a "rescue" phase, which is used to resolve ambiguities that are caused by the VCF specification.

It is worth noting that SMaSH does not include any datasets that are designed for joint variant calling. Because of this, we fall back on concordance as a metric for evaluating our joint variant calling algorithms which are described in §3.3.

3. ARCHITECTURE

When architecting `avocado`, we made a conscientious decision to prioritize modularity and extensibility. There are several reasons behind this design choice:

- Current variant calling pipelines are not meant to be extended. Because of this, anyone looking to prototype a new variant calling algorithm must implement their own variant calling infrastructure, which is a significant impediment to variant calling research.
- Although we have limited specialization in our current pipeline (we specialize towards local assembly and pileup based SNP calling), long term we plan to add increasingly specialized variant calling algorithms.
- Similarly, it is known that modern variant callers perform poorly when calling structural variants (SVs). This is a critical area that we hope to attack through specialized variant calling algorithms.

[2]Identifying overlap between the call set of multiple variant calling pipelines.

To improve modularity, we pushed as much functionality into the ADAM stack as possible [19]. ADAM implements several important transformations that are used on the read processing frontend, including sort-by-reference position, duplicate marking, base quality score recalibration (BQSR), and local realignment (see §2.5). Additionally, we have moved portions of the variant calling stack into ADAM—after genotyping samples, we use the ADAM pipeline to transform genotypes into variant calls. We include a brief discussion of the merits of this approach in Appendix B.

A diagram of the `avocado` architecture is included in Figure 1.

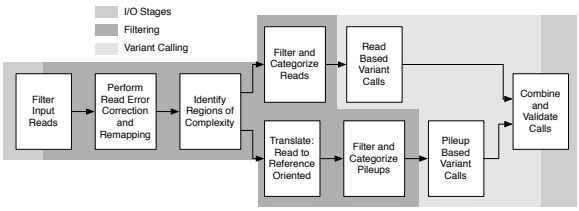

Figure 1: System Architecture

`avocado` is roughly divided into four stages:

1. **Read pre-processing:** Applies transformations from the ADAM pipeline, including sort, duplicate marking, BQSR, and local realignment. Operates strictly on read data.
2. **Filtering:** Selects the best algorithm for calling variants on a segment of data; transforms data to be reference-oriented if necessary. This is discussed in §3.4.
3. **Variant calling:** Calls genotypes for samples in input set on either read-oriented or reference-oriented data. These algorithms are discussed in §3.1 and §3.2.
4. **Variant filtering:** Here, variant data is calculated from genotypes using the ADAM API [19]. Additionally, variants can be filtered for quality.

All stages are designed to be configurable, and easy to replace. Specifically, the filtering and variant calling stages are designed with a specific eye towards modularity. For variant calls, we provide a base class, and two specialized subclasses for variant calls that operate on either read or reference oriented data. A variant calling algorithm can be added by implementing one of these class interfaces, and then registering the call with a filter. For filters, we provide two filtering stages: the first operates on read data, and is used to implement the filter described in §3.4. The

second filtering stage operates on pileup (reference oriented) data—currently, we use this to filter out pileups that have no mismatch evidence. For both of these filters, we provide a simple interface for developers to implement.

To improve pipeline performance, we made a significant optimization to the reads-to-pileup transformation step. At the start of transforming reads into reference oriented data, the reads are sorted by reference position. However, due to the partitioning used by Spark [31], after a naïve transformation of reads to single reference oriented bases, the reference oriented bases are no longer sorted. This makes the ensuing grouping of bases by position very expensive due to significant disk shuffle I/O. Instead, we perform a quasi-streaming transform on the read data. Here, the sorted read blocks are chunked together. All of the reads in this window are then converted into pileup bases, and then grouped together into rods. This leads to significantly better performance, but has tricky edge cases: reads that border the end of a group must be duplicated into both read groups.

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:

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

where $\mathbf{P}(r|H)$ is obtained from aligning the read and the candidate haplotype by a pairwise HMM alignment [10]. 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. Note that our definition of an active region is similar in spirit but not in implementation to the "active regions" defined by the Complete Genomics variant caller [5] and current versions of GATK [9].

The assembly of an active region is a kind of k-mer graph or "Eulerian" approach [23]. We start 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.

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

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

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 [1]:

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

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 [1]:

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

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 [1]:

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

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:

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

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 Genotype Calling

For one sample at a site, we can estimate the genotype likelihood based on the number of reads that match the reference genome and the quality of each of these reads. Let the number of reads at site a be k and the ploidy be m. Without loss of generality assume the first $l \leq k$ bases match the reference, and the rest do not. Let ϵ_j be the error probability

of the jth read base. We have that,

$$\mathcal{L}(g) = \frac{1}{m^k} \prod_{j=1}^{l} [(m-g)\epsilon_j + g(1-\epsilon_j)] \times \prod_{j=l+1}^{k} [(m-g)(1-\epsilon_j) + g\epsilon_j] \quad (7)$$

Here we are only leveraging the data of a single sample. Performance can be improved by considering statistics associated with other samples. If a study involves sequencing many individuals from a population, it is beneficial to run them jointly in order to determine population parameters like per-locus allele frequency spectrum (AFS) and minor allele frequency (MAF) (§ 3.3). However, when genotyping a single individual computational time can be saved by looking these parameters up in a database such as dbSNP. If the population MAF at site a is ϕ_a, the likelihood can be compensated by the prior probability of seeing a non-reference allele, and the genotype is

$$\hat{g}_a = \arg\max_g \frac{\mathcal{L}(g)P[g|\phi_a]}{\sum_g \mathcal{L}(g)P[g|\phi_a]} \quad (8)$$

where $P[g|\phi] = \binom{m}{g}\phi^g(1-\phi)^{m-g}$ is the pmf of the binomial distribution.

3.3 Joint Variant Calling

When genotyping several individuals one may wish to genotype them jointly while determining population allele statistics, especially when said individuals are from a specific population of interest. In this case, we can use the EM procedure of Samtools Mpileup. Given the data for several individuals and using the likelihood in (7), we can infer the population MAF per site via iteration of

$$\phi_a^{(t+1)} = \frac{1}{M}\sum_{i=1}^{n} \frac{g\mathcal{L}(g)P[g|\phi_a^{(t)}]}{\sum_g \mathcal{L}(g)P[g|\phi_a^{(t)}]} \quad (9)$$

where n is the number of individuals, $M = \sum_{i=1}^{n} m_i$ is the total number of chromosomes and P is the binomial likelihood described above. This population MAF can then be used in genotyping as above.

3.4 Algorithm Selection

As discussed in §3, we seek to improve the performance of our variant caller without reducing accuracy by directing the variant caller to use higher accuracy methods in areas that show increased complexity. Loosely, we define a complex region as an area that is highly similar to other areas of the genome, or where it is likely that a complex variant (such as an indel) is segregating. To identify those regions, we use the following heuristics:

- Areas that are highly similar to other areas of the genome can be distinguished by low mapping quality. Reduced mapping quality indicates that alignment found several areas where the read could map with similar quality.
- Complex variants lead to a change in coverage over the effected area. Deletions will see reduced coverage, and insertions lead to increased coverage.

We implemented our filter by stepping a window across the reference genome. The window was stepped by 1000 base pairs. In each 1000 base pair window, we would compute the average coverage, and mapping quality. If a window violated either of the mapping quality or the coverage threshold, we would flag it as high complexity and pass it to the assembler. If a window did not violate either of those two thresholds, we built pileups out of the data on this interval, and then used the genotyping methods described in §3.2.

We provided several tuning parameters to the end user. Specifically, the end user could set the target mapping quality and coverage deviation percentage (percent change from mean coverage). Table 1 summarizes the sensitivity of these parameters.

Table 1: % Reads in High Complexity Region

MapQ,Cov	0.2	0.4	0.6
40	88%	54%	20%
50	90%	56%	22%
60	98%	91%	88%

As can be noted, mapping complexity begins to saturate as the mapping quality threshold increases to 60[3], and as the coverage variation coefficient decreases to 0.2.

4. EVALUATION

In this section, we present an evaluation of `avocado`. We evaluate the accuracy of our variant calls using SMaSH, which was introduced in §2.6. All of our variant calling was run on Amazon Elastic Compute 2 (EC2), using the machine types listed in Table 2.

Table 2: EC2 Machine Types Used

Machine	CPU	Memory[4]	Storage	Cost[5]
`m2.4xlarge`	8	68.4GiB	2×840GB	$1.64
`cr1.8xlarge`	32	244GiB	2×120GB	$3.50
`hs1.8xlarge`	16	117GiB	20×2,048GB	$4.60

The storage drive types were different between the two machines: the `m2.4xlarge` machines were backed by hard disk drives (HDD), while the `cr1.8xlarge` machines are backed by solid state drives.

The datasets used in testing `avocado` are listed in Table 3.

The Venter dataset is high coverage data (30×) coverage, and was generated using simNGS [20]. The NA12878 dataset

[3]Phred-scaled 60 is equivalent to a probability of $P = 0.999999$ that the read is correctly mapped.
[4]One Gibibyte (GiB) is equal to 2^{30} bytes, as opposed to a Gigabyte (GB) which is equal to 10^9 bytes.
[5]Cost is per machine per hour.

Table 3: Datasets Used

Dataset	Type	Size	From
Venter	Synthesized Human	99GB	[27]
NA12878	Human Whole Genome	15GB	[25]
	Chromosome 11	673MB	[25]
	Chromosome 20	297MB	[25]
NA18507	Chromosome 20	236MB	[25]

is low coverage (5×) data, and was sequenced on an Illumina HiSeq2000. We chose these datasets because they allowed us to view several different areas in the design space—specifically, we have different coverage depths, as well as different file sizes. Long term, we are interested in adding the high coverage (50×) NA12878 dataset to our experiments, so that we can identify how strongly we perform on low vs. high coverage reads for the same dataset. Additionally, we plan to use `avocado` on cancer genomes starting in February of next year, when the datasets become available.

4.1 Accuracy

Due to an incompatibility with Hadoop-BAM and ADAM [19, 22], we were unable to export a VCF file from our variant calling pipeline. This precludes us from running SMaSH on our calls, and prevents us from determining accuracy and recall numbers against validated data. We are currently working very closely with the Hadoop-BAM team to resolve this issue—specifically, the ADAM Variant and Genotype data can be converted over to the internal representation used for the VCF, but the internal representation for VCF data cannot be written out to disk in Spark from Scala. It appears that this issue stems from the Hadoop Job configuration, and we hope to have this issue resolved by the beginning of January.

However, we are able to obtain concordance data from running joint variant calling. We ran our standalone variant caller on NA12878, and our joint variant caller on NA12878 and NA18507. The data on callset overlap from this run is contained in Table 4.

Table 4: Joint Variant Calling Concordance

NA12878	Overlap	Joint Calling
14.8M	13.7M	15.8M
92.5%	—	86.7%

This concordance data is promising—however, it is worth noting that since we do not currently support joint variant calling on the regions that we assemble, the variant calls resulting from assembly will be unchanged after joint calling, and will thusly always have 100% concordance. We use the default filtering settings for this run, which means that 22% of the genome will be assembled (see Table 1). However, due to the way we emit variant calls for assembled regions, assembly accounts for less than 22% of the variant calls. There are several other important items to note about this data:

- The lack of joint assembly is partly accountable for the 13.3% discordance from joint variant calling. Additionally, we do not filter out the variant calls that are called solely on NA18507, and not on NA12878. These calls will not overlap with the calls made in the single sample run using NA12878.
- We hope to experiment more with this data at a later point, with more samples. Joint calling methods depend significantly on the sample size used; by default, we seed the single sample calling pipeline with data from dbSNP [24]. dbSNP contains a significantly larger set of data than used in this trial; it would be interesting to see how many samples are necessary to approach 100% concordance between calling against dbSNP and traditional joint calling.

We view these results as promising, but note that this section requires additional progress. As soon as the ADAM to VCF pipeline is complete, we plan to run the following experiments:

- Use SMaSH to benchmark performance against validated datasets.
- Perform concordance measurements against established pipelines including the GATK and Mpileup.

While the SMaSH benchmarking numbers will provide us with feedback about the true accuracy of our pipeline, it is also important to validate concordance with other established tools. This is because low concordance would likely point to issues with the way we are formatting our variant calls. This would not be captured in SMaSH, as the rescue phase will mark these calls as correct [27]. While these calls may be correct, it is important to be syntactically similar to existing tools, as variant annotation pipelines expect to see data that is formatted in these styles.

4.2 Performance

In this section, we review the runtime performance of `avocado` on varying dataset sizes, and machine configurations. From this, we present areas for optimization, and recommendations for people using the system. Additionally, we highlight the factors that contribute to the performance characteristics of the system.

4.2.1 Multi-node Scaling

A sample test case that demonstrates `avocado`'s ability to scale to multiple nodes is illustrated in figure 2. In this test case, we performed SNP-only genotyping on the NA12878, chromosome 11 dataset. All tests used a single Spark master node, and the number of computing slaves used was scaled from 1 to 8 `m2.4xlarge` nodes.

As can be noted, we see superlinear speedup for the 4 and 8 node cases. This occurs due to two reasons:

1. The 4 and 8 node cases exercise higher disk parallelism when reading and writing results, because the number of HDFS replicas is increased from 1 to 3 once there are more than three nodes [3].

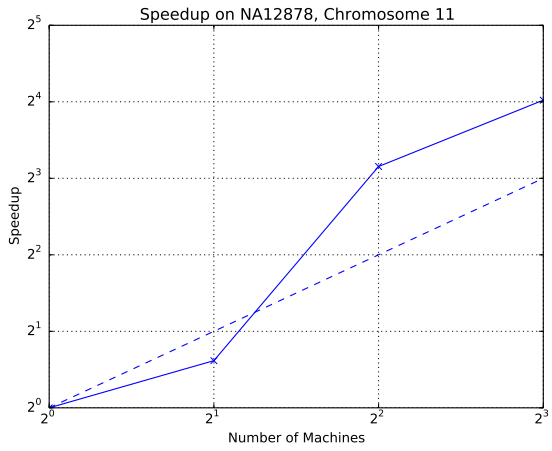

Figure 2: Multi-Node Processing: Speedup on NA12878, Chromosome 11

2. Additionally, scaling beyond 2 nodes reduces memory backpressure. In Spark, memory backpressure causes data to be written to disk. Also, by increasing the number of disks in the system, we increase the aggregate disk throughput, which improves shuffle performance.

As was reported in the ADAM technical report [19], single node performance is improved the furthest by using a machine with a very large memory, and with many disks. Specifically, for processing a large file on a single node, we recommend using the `hs1.8xlarge` machine with the HDDs configured in redundant array of inexpensive disks (RAID) 0.

4.2.2 *Scaling Across Datasets*

Our performance changes as the size of data scales. This is demonstrated in Table 4.2.2.

Sample	Time	Machines
Venter	6hr	20×`cr1.8xlarge`
NA12878, chr11	0.47hr	4×`m2.4xlarge`
NA12878, chr20	0.25hr	4×`m2.4xlarge`

We note the following about how our algorithm scales with data size:

- For filtering, we must perform three large reductions at the immediate start of variant calling. On the whole genome Venter case, we noted several large stragglers during these reductions. Additionally, Spark fails to pipeline these reductions.
- We see greater scalability than expected when moving from chromosome 20 on NA12878 to chromosome 11. Although chromosome 11 is 2.26× larger than chromosome 20, it's execution is only 1.8× slower. This is due to limited parallelism at the beginning of the pipeline.

We are currently investigating the reduction issue—although we are not sure of why there is significant skew, we can reduce the impact of this skew by manually pipelining the three reductions, as the operate on the same dataset. The limited parallellism at the start of the pipelines is only a limitation for smaller datasets. This limitation arises from Parquet, and is due to the parameters passed to ADAM when converting BAM files to the ADAM format. Currently, we opt to increase the parallelism by coalescing the data into more partitions. We choose to at least match the core count of the machines we are running on. This increases parallelism, but comes at a cost: this coalesce operation requires data to be shuffled on disk.

4.2.3 *Joint Variant Calling Performance*

We tested our joint variant caller by jointly calling chromosome 20 of both NA12878 and NA18507; these two chromosomes account for 533 MB of data. In this test, we were able to joint call the two samples in 0.45 hours. Readers may note that this number is lower than expected, based off of the data from Table 4.2.2; in this table, we illustrate that we can call 673 MB of data in 0.47 hours. This discrepancy can be explained by the inclusion of the EM algorithm described in §3.3. This algorithm requires several iterations to converge, and requires the creation of a matrix that is significantly larger than the vector used for genotyping. It is quite likely that this would lead to decreased performance. Additionally, sort consumes more time for the two genome case. We believe that this is because the act of appending two sets of read data together eliminates any advantages that are obtained from having the read data come in with reasonable local sorting. Thusly, we require additional shuffling when sorting the union of the data.

5. FUTURE WORK

Several points for future work are noted elsewhere in this paper; the most significant necessary future work is noted in §4.1. This section covers other areas that we hope to address in a future revision of `avocado`.

As a project, `avocado` has a well defined path for future research:

- A major driving goal for this project is to be able to call variants across a full, very high coverage genome within a couple of hours. Currently, we call variants in 6 hours on 20 machines on a 30× coverage genome. Within several months, we hope to call variants on a 60× coverage genome in less time.
- Although `avocado` currently is designed for germline variant calling, we hope to use this tool for calling variants on tumor genomes. A full discussion of the changes necessary for tumor calling is out of the scope of this paper, rather, we refer interested readers to the Cibulskis et al, which discusses an adaptation of the GATK for mutation calling [7].

From a timeline perspective, we hope to integrate `avocado` with SNAP [30] by the end of January 2014. We expect that this project will reach maturity in the spring of 2014, and to use it on patient data by the middle/end of 2014.

5.1 Performance Bottlenecks

Currently, there exist several significant performance bottlenecks in our pipeline. We need to address these bottlenecks in order to hit the performance targets outlined above. These performance bottlenecks include the following:

- Although read to pileup creation time has been reduced, there is still a significant latency hit taken because these steps appear to emit unsorted pileups. We have tried to locally sort the pileups during creation, but this does not impact the output. We believe this is due to using an unsorted Spark partitioner. This is important because it is desirable to emit sorted variants—additionally, we would prefer to have our variants sorted, as this minimizes shuffle traffic when performing variant filtering. One approach we are considering to ameliorate this issue involves refactoring the mechanism we use for distributing data between the calling methods. We hope that we can change the signature for our variant callers to operate on partitions of read data, which would allow us to assign calls to sorted read partitions, and perform the calling in situ. As long as the calls emitted per partition are sorted, this would maintain sorted order on the variants.
- As discussed in §4.2.2, we struggle when computing basic statistics on larger datasets. We propose a solution to the problem of reductions not being pipelined in that section, but we also must point out that key skew remains an issue. What may help to address this issue is to perform these reductions after reads have been reduplicated and sorted. The sorting process has optimizations that reduce key skew across partitions. These optimizations came about due to problems encountered when sorting sets of reads that include unmapped reads [19].
- Our current genotype to variant conversion code requires a group-by on the genotype data. As noted two bullets above, this is problematic because we do not currently guarantee ordering on the genotype data, which leads to poor group-by performance. One approach that may improve this is to refactor this code to not require coordination between the variant context object, the variant, and the genotype. This would eliminate a group-by, which would lead to a significant performance gain when emitting variants.

We hope to address these performance issues in release 0.0.2 at the beginning of 2014.

5.2 Structural Variant Calling

Structural variant calling presents a great area for improvement in variant calling; modern structural variant callers are correct less than 25% of the time [27]. Long term, we would like to improve on this performance by using specialized algorithms, as this is a key benefit of the modular toolkit we have designed. We see several significant opportunities:

- As demonstrated by GQL [13], there is significant opportunity to use algorithms like BreakDancer [6] along with filtering. BreakDancer is an algorithm for detecting long deletions and translocations using a split-read aligner. To support BreakDancer, we would need to have aligner support—this would require modifications to SNAP [30].
- Another opportunity could come from using a full *de novo* assembler for areas where indels are segregating. De novo assembly does not rely on a reference[6], and is a more expensive formulation of the local assembly problem. Although de novo assembly is more expensive, it presents much higher accuracy for complex portions of the genome, such as near telomeres and centromeres [4].

By integrating these algorithms into `avocado`, we hope to achieve significantly higher structural variant calling accuracies than existing pipelines.

5.3 Improving Local Assembly

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 [23], 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. 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.

5.4 Improved Region Filtering

There are several efforts underway to improve variant calling outcomes through the filtering of reads. These methods are similar to the methods we implement in §3.4, but provide deeper formulations for their work. These methods include:

- **Similar Region Detection:** In this unreleased work, the reference sequence is evaluated to find regions that expose high similarity. This is done by forming a contig graph, and identifying highly connected regions. This detection is not performed at runtime; rather, it is performed once per region. The output of this tool includes regions where alignments cannot necessarily be trusted. To incorporate this, we would add a filtering block that would identify reads which group into similar regions, and then we would assemble the requisite number of haplotypes from this region, using the reference sequence as anchors.
- **Changepoint Detection:** This work uses an improved, more statistically rigorous set of heuristics to

[6]Some de novo assemblers are *reference assisted*, which means that they take some hints from a reference genome assembly.

determine whether a region presents high complexity. Similar to our work, we would use assembly in these regions. We hope that this will improve the filtering, as we do not always catch indels in our filtering stage. However, it may be cheaper to use a simple heuristic and to then revert a block to assembly if we encounter indel evidence.

Thanks to the modular decomposition of our system, these two filtering implementations should be straightforward to implement. Additionally, the similar region detection system may necessitate the addition of a new schema to the ADAM toolkit for describing ranges in a genome.

6. CONCLUSION

In this paper, we presented `avocado`, a distributed variant caller implemented using Spark and ADAM [31, 19]. We illustrate the performance of the system across a variety of datasets including both high coverage sampled human genomes and low coverage human genomes. Major contributions of this system include the development of an open source local assembler, the use of filtering to determine when it is efficient to use local assembly instead of pileup based calling methods, the development of a system which can approximate joint genotyping with a single sample, and the development of a modular, open source platform for developing and prototyping new variant calling algorithms. We review the performance of this system on computing clusters containing up to 20 nodes; from these measurements, we identify areas for performance optimization, and plot a course for the continued development of this toolkit.

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`

B. GENOTYPE/VARIANT REFACTORING

Traditionally, variant calling has involved genotyping n samples, and then determining which of the genotypes of these samples contain true variants. This involves the calculation of a *variant quality* score[7], and filtering variants.

Semantically, all of the information necessary for the creation of variant data can be extracted from the genotypes of the samples. Therefore, the step of packing/unpacking genotype data to create variant calls is unnecessary. To reduce the likelihood of making errors, we have migrated code to do this into the ADAM [19] framework. Then, to get variant calls, we must just genotype the samples we see, and call the ADAM library.

This does not preclude joint variant calling—although joint variant calling implies that multi-sample data is being used to influence the filtering of variants, this is a bit of a misnomer. Practically, as discussed in §3.3, joint variant calling involves the use of data from multiple samples to refine genotype probabilities across a population to support genotypes that are frequently seen in this population.

[7]Loosely defined as the likelihood that there is *at least* one genotype with this variant out of all haplotypes seen in the call set. However, notably, FreeBayes [11] uses a different definition for variant quality.

C. REFERENCES

[1] C. A. Albers, G. Lunter, D. G. MacArthur, G. McVean, W. H. Ouwehand, and R. Durbin. Dindel: Accurate indel calls from short-read data. *Genome Research*, 21:961–973, 2011.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[3] D. Borthakur. The hadoop distributed file system: Architecture and design, 2007.

[4] M. Bresler, S. Sheehan, A. H. Chan, and Y. S. Song. Telescoper: de novo assembly of highly repetitive regions. *Bioinformatics*, 28(18):i311–i317, 2012.

[5] P. Carnevali, J. Baccash, A. L. Halpern, I. Nazarenko, G. B. Nilsen, K. P. Pant, J. C. Ebert, A. Brownley, M. Morenzoni, V. Karpinchyk, B. Martin, D. G. Ballinger, and R. Drmanac. Computational techniques for human genome resequencing using mated gapped reads. *Journal of Computational Biology*, 19(3):279–292, 2012.

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

[7] K. Cibulskis, M. S. Lawrence, S. L. Carter, A. Sivachenko, D. Jaffe, C. Sougnez, S. Gabriel, M. Meyerson, E. S. Lander, and G. Getz. Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples. *Nature biotechnology*, 31(3):213–219, 2013.

[8] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry, et al. The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158, 2011.

[9] M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. del Angel, M. A. Rivas, M. Hanna, et al. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nature genetics*, 43(5):491–498, 2011.

[10] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge Univ Press, 1998.

[11] E. Garrison and G. Marth. Haplotype-based variant detection from short-read sequencing. *arXiv preprint arXiv:1207.3907*, 2012.

[12] T. Jiang, G. Sun, Y. Wu, W. Wang, J. Hu, P. Bodily, L. Tian, H. Hakonarson, W. E. Johnson, et al. Low concordance of multiple variant-calling pipelines: practical implications for exome and genome sequencing. 2013.

[13] C. Kozanitis, A. Heiberg, G. Varghese, and V. Bafna.

Using Genome Query Language to uncover genetic variation. *Bioinformatics*, 2013.

[14] L. Kruglyak et al. The use of a genetic map of biallelic markers in linkage studies. *Nature genetics*, 17(1):21–24, 1997.

[15] B. Langmead, C. Trapnell, M. Pop, S. L. Salzberg, et al. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol*, 10(3):R25, 2009.

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

[17] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

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

[19] M. Massie, F. A. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson. ADAM: Genomics formats and processing patterns for cloud scale computing. Technical Report UCB/EECS-2013-207, EECS Department, University of California, Berkeley, Dec 2013.

[20] T. Massingham. simNGS and simLlibrary. `http://www.ebi.ac.uk/goldman-srv/simNGS/`, 2012.

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

[22] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä, E. Korpelainen, and K. Heljanko. Hadoop-BAM: directly manipulating next generation sequencing data in the cloud. *Bioinformatics*, 28(6):876–877, 2012.

[23] P. A. Pevzner, H. Tang, and M. S. Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[24] S. T. Sherry, M.-H. Ward, M. Kholodov, J. Baker, L. Phan, E. M. Smigielski, and K. Sirotkin. dbSNP: the NCBI database of genetic variation. *Nucleic Acids Research*, 29(1):308–311, 2001.

[25] N. Siva. 1000 genomes project. *Nature biotechnology*, 26(3):256–256, 2008.

[26] E. S. Snitkin, A. M. Zelazny, P. J. Thomas, F. Stock, D. K. Henderson, T. N. Palmore, J. A. Segre, et al. Tracking a hospital outbreak of carbapenem-resistant Klebsiella pneumoniae with whole-genome sequencing. *Science translational medicine*, 4(148):148ra116–148ra116, 2012.

[27] A. Talwalkar, J. Liptrap, J. Newcomb, C. Hartl, J. Terhorst, K. Curtis, M. Bresler, Y. S. Song, M. I. Jordan, and D. Patterson. SMASH: A benchmarking toolkit for variant calling. *arXiv preprint arXiv:1310.8420*, 2013.

[28] The International HapMap Consortium. The international HapMap project. *Nature*, 426:6968, 2003.

[29] Twitter and Cloudera. Parquet. `http://www.parquet.io`.

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

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