Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category

OXFORD

Subject Section

Distributed Variant Calling with Avocado

Frank Austin Nothaft [1,2]*, David A. Patterson [1,2] and Anthony D. Joseph [1]

[1] AMPLab, University of California, Berkeley, 94720, USA and
[2] ASPIRE Lab, University of California, Berkeley, 94720, USA.

*To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: With the increasing size of genomic datasets, computational analysis is a rate limiting step. Variant calling is a particularly expensive step, especially when working with whole genome data. To improve performance, we implement variant calling on top of the horizontally scalable Apache Spark distributed computing framework.
Results: Avocado achieves linear scalability with the size of the computing cluster, while achieving state-of-the-art accuracy in SNP calling, and competitive accuracy at INDEL calling.
Availability: Avocado is open source software, released under the Apache 2 license. The Avocado source code is hosted at https://github.com/bigdatagenomics/avocado.
Contact: fnothaft@berkeley.edu

1 Introduction

While many modern short-read based variant callers can achieve greater than 99% accuracy when calling single nucleotide variants (SNVs), accuracy decreases when calling insertion/deletion (INDEL) variants. For INDELs that are shorter than the read length, some degradation in accuracy is caused by incorrect mapping of reads containing INDELs. However, this is not the primary culprit. While most reads map correctly, they are frequently locally misaligned. In short, these reads have been placed in the correct area of the reference genome, but their local alignment makes them appear to represent a different sequence variant then they truly contain.

There are two general approaches that can be used to eliminate the effect of incorrect local alignment on variant calling accuracy: we can either use a local sequence alignment algorithm that is less likely to misrepresent INDEL variants when aligning individual reads, or we can try to improve the alignments of all reads that have mapped around a genomic locus by pooling and jointly realigning these reads. Most tools have focused on the second approach, which further bifurcates into realignment-only and reassembly-with-realignment algorithms. Research has focused on realignment-based approaches because realignment-based approaches obliviate the major problem with traditional local sequence alignment algorithms (Smith and Waterman, 1981; Ukkonen, 1985; Landau and Vishkin, 1986), which by their probabilistic or dynamic nature cannot be guaranteed to emit consistent alignments for all reads that contain an INDEL variant. Realignment algorithms eliminate this problem by identifying all possible INDELs in the reads at the site, scoring these variants at the site, and realigning the reads to contain a consistent representation of the top scoring variant(s).

Although realignment and reassembly algorithms have high accuracy, they also have high computational cost. The traditional formulation of a realignment algorithm has $\mathcal{O}(n^4)$ runtime complexity, while most local sequence alignment algorithms have runtime complexity of $\mathcal{O}(n^2)$ or lower. Additionally, realignment approaches require all of the reads covering a genomic locus to be materialized which necessitates a shuffle of the input dataset and can have significant memory requirements. While the end-to-end runtime can be decreased by only reassembling a portion of the genome, the performance improvements available through these approaches are bound similarly to Amdahl's law: local assembly is sufficiently expensive that an approach that prefilters 90% of the genome only achieves a $3\times$ improvement in end-to-end runtime (Bloniarz *et al.*, 2014).

As an alternative to traditional realignment and reassembly approaches, we introduce a local sequence alignment algorithm that is inspired by colored de Bruijn graphs (Iqbal *et al.*, 2012). This algorithm has several desirable properties: it has linear runtime complexity and it produces provably canonical pairwise alignments. When used for locally realigning previously mapped reads, we can determine if an individual read is already canonically aligned prior to realigning, which allows us to aggressively limit the number of reads realigned. Our algorithm uses motifs that we have identified in the structure of the sequence flanking a bubble in a colored de Bruijn graph. Our algorithm has an efficient implementation and we can prove strong guarantees about the local alignments that our algorithm generates. We have implemented this algorithm in Avocado, a

parallel variant caller implemented using Apache Spark (Zaharia *et al.*, 2012, 2010) and the ADAM library for parallel genomic analysis (Massie *et al.*, 2013; Nothaft *et al.*, 2015). Our approach achieves a $10\times$ speedup over conventional INDEL realignment tools when run on a single node while maintaining variant calling accuracy, and can be parallelized across a cluster of computers. Our implementation is released as open source code under an Apache 2 license and is available from `https://github.com/bigdatagenomics/avocado`.

2 Approach

The accuracy of insertion and deletion (INDEL) variant discovery has been improved by the development of variant callers that couple local reassembly with haplotype-based statistical models to recover INDELs that were locally misaligned (Albers *et al.*, 2011). Now, several prominent variant callers such as the Genome Analysis Toolkit's (GATK) HaplotypeCaller (DePristo *et al.*, 2011), Scalpel (Narzisi *et al.*, 2014), and Platypus (Rimmer *et al.*, 2014). Although haplotype-based methods have enabled more accurate INDEL and single nucleotide polymorphism (SNP) calls (Bao *et al.*, 2014), this accuracy comes at the cost of end-to-end runtime (Talwalkar *et al.*, 2014). Several recent projects have been focused on improving reassembly cost either by limiting the percentage of the genome that is reassembled (Bloniarz *et al.*, 2014) or by improving the performance of the core algorithms used in local reassembly (Rimmer *et al.*, 2014).

The performance issues seen in haplotype reassembly approaches derives from the high asymptotic complexity of reassembly algorithms. Although specific implementations may vary slightly, a typical local reassembler performs the following steps:

1. A de Bruijn graph is constructed from the reads aligned to a region of the reference genome,
2. All valid paths (*haplotypes*) between the start and end of the graph are enumerated,
3. Each read is realigned to each haplotype, typically using a pair Hidden Markov Model (HMM, see Durbin *et al.* (1998)),
4. A statistical model uses the read↔haplotype alignments to choose the haplotype pair that most likely represents the variants hypothesized to exist in the region,
5. The alignments of the reads to the chosen haplotype pair are used to generate statistics that are then used for genotyping.

In this paper, we focus on steps one through three of the local reassembly problem, as there is wide variation in the algorithms used in stages four and five. Stage one (graph creation) has approximately $\mathcal{O}(rl_r)$ time complexity, and stage two (graph elaboration) has $\mathcal{O}(h\max(l_h))$ time complexity. The asymptotic time cost bound of local reassembly comes from stage three, where cost is $\mathcal{O}(hrl_r\max(l_h))$, where h is the number of haplotypes tested in this region[1], r is the number of reads aligned to this region, l_r is the read length[2], and $\min(l_h)$ is the length of the shortest haplotype that we are evaluating. This complexity comes from realigning r reads to h haplotypes, where realignment has complexity $\mathcal{O}(l_r l_h)$.

[1] The number of haplotypes tested may be lower than the number of haplotypes reassembled. Several tools (see DePristo *et al.* (2011); Garrison and Marth (2012)) allow users to limit the number of haplotypes evaluated to improve performance.

[2] For simplicity, we assume constant read length. This is a reasonable assumption as many of the variant callers discussed target Illumina reads that have constant length.

In this paper, we introduce the indexed de Bruijn graph and demonstrate how it can be used to reduce the asymptotic complexity of reassembly. An indexed de Bruijn graph is identical to a traditional de Bruijn graph, with one modification: when we create the graph, we annotate each k-mer with the index position of that k-mer in the sequence it was observed in. This simple addition enables the use of the indexed de Bruijn graph for $\Omega(n)$ local sequence alignment with canonical edit representations for most edits. This structure can be used for both sequence alignment and assembly, and achieves a more efficient approach for variant discovery via local reassembly.

Current variant calling pipelines depend heavily on realignment based approaches for accurate genotyping (Li, 2014). Although there are several approaches that do not make explicit use of reassembly, all realignment based variant callers use an algorithmic structure similar to the one described above. In non-assembly approaches like FreeBayes (Garrison and Marth, 2012), stages one and two are replaced with a single step where the variants observed in the reads aligned to a given haplotyping region are filtered for quality and integrated directly into the reference haplotype in that region. In both approaches, local alignment errors (errors in alignment *within* this region) are corrected by using a statistical model to identify the most likely location that the read could have come from, given the other reads seen in this area.

Although the model used for choosing the best haplotype pair to finalize realignments to varies between methods (e.g., the GATK's IndelRealigner uses a simple log-odds model (DePristo *et al.*, 2011), while methods like FreeBayes (Garrison and Marth, 2012) and Platypus (Rimmer *et al.*, 2014) make use of richer Bayesian models), these methods require an all-pairs alignment of reads to candidate haplotypes. This leads to the runtime complexity bound of $\mathcal{O}(hrl_r\min(l_h))$, as we must realign r reads to h haplotypes, where the cost of realigning one read to one haplotype is $\mathcal{O}(l_r\max(l_h))$, where l_r is the read length (assumed to be constant for Illumina sequencing data) and $\max(l_h)$ is the length of the longest haplotype. Typically, the data structures used for realignment ($\mathcal{O}(l_r\max(l_h))$ storage cost) can be reused. These methods typically retain *only* the best local realignment per read per haplotype, thus bounding storage cost at $\mathcal{O}(hr)$.

For non-reassembly based approaches, the cost of generating candidate haplotypes is $\mathcal{O}(r)$, as each read must be scanned for variants, using the pre-existing alignment. These variants are typically extracted from the CIGAR string, but may need to be normalized (Li, 2014). de Bruijn graph based reassembly methods have similar $\mathcal{O}(r)$ time complexity for building the de Bruijn graph as each read must be sequentially broken into k-mers, but these methods have a different storage cost. Specifically, storage cost for a de Bruijn graph is similar to $\mathcal{O}(k(l_{\text{ref}}+l_{\text{variants}}+l_{\text{errors}}))$, where l_{ref} is the length of the reference haplotype in this region, l_{variants} is the length of true variant sequence in this region, l_{errors} is the length of erroneous sequence in this region, and k is the k-mer size. In practice, we can approximate both errors and variants as being random, which gives $\mathcal{O}(kl_{\text{ref}})$ storage complexity. From this graph, we must enumerate the haplotypes present in the graph. Starting from the first k-mer in the reference sequence for this region, we perform a depth-first search to identify all paths to the last k-mer in the reference sequence. Assuming that the graph is acyclic (a common restriction for local assembly), we can bound the best case cost of this search at $\Omega(h\min l_h)$.

The number of haplotypes evaluated, h, is an important contributor to the algorithmic complexity of reassembly pipelines, as it sets the storage and time complexity of the realignment scoring phase, the time complexity of the haplotype enumeration phase, and is related to the storage complexity of the de Bruijn graph. The best study of the complexity of assembly techniques was done by Kingsford et al. (Kingsford *et al.*, 2010), but is focused on *de novo* assembly and pays special attention to resolving repeat structure. In the local realignment case, the number of

haplotypes identified is determined by the number of putative variants seen. We can naïvely model this cost with (1), where f_v is the frequency with which variants occur, ϵ is the rate at which bases are sequenced erroneously, and c is the coverage (read depth) of the region.

$$h \sim f_v l_{\text{ref}} + \epsilon l_{\text{ref}} c \tag{1}$$

This model is naïve, as the coverage depth and rate of variation varies across sequenced datasets, especially for targeted sequencing runs (Fang *et al.*, 2014). Additionally, while the ϵ term models the total number of sequence errors, this is not completely correlated with the number of *unique* sequencing errors, as sequencing errors are correlated with sequence context (DePristo *et al.*, 2011). Many current tools allow users to limit the total number of evaluated haplotypes, or apply strategies to minimize the number of haplotypes considered, such as filtering observed variants that are likely to be sequencing errors (Garrison and Marth, 2012), restricting realignment to INDELs (IndelRealigner, DePristo *et al.* (2011)), or by trimming paths from the assembly graph. Additionally, in a de Bruijn graph, errors in the first k or last k bases of a read will manifest as spurs and will not contribute paths through the graph. We provide (1) solely as a motivating approximation, and hope to study these characteristics in more detail in future work.

3 Methods

To use Avocado to call variants, we run two applications, each of which has several sub-stages:

1. **INDEL Reassembly:** Here, we clean up all reads that are aligned near INDEL variants. We do this as a two step process:
   a. We make a pass over all reads, using our "bubble flank motif" algorithm to extract INDEL variants. These INDEL variants are collected on a single node, and used as inputs to the next stage.
   b. We run ADAM's (Massie *et al.*, 2013; Nothaft *et al.*, 2015) INDEL realigner, using the discovered INDELs from stage one as "known INDELs" to realign to.
2. **Variant Calling:** In this phase, we discover all SNVs and INDELs, score them using the reads, and emit either called variants or genotype likelihoods in genome VCF (gVCF) format. This runs as a four step process:
   a. We extract all variants from the aligned reads by parsing the alignments.
   b. Using these variants, we compute all read/variant overlaps, and compute the likelihood that each read represents a given variant that it overlaps. In gVCF mode, we also calculate the likelihood of the reference allele at all locations covered by a read.
   c. We merge all of the per-read likelihoods per variant. This gives us final genotype likelihoods per each variant.
   d. Finally, we apply a standard set of hard filters to each variant.

All of these stages are implemented as a parallel application that runs on top of Apache Spark (Zaharia *et al.*, 2010, 2012), using the ADAM library (Massie *et al.*, 2013; Nothaft *et al.*, 2015).

3.1 INDEL Reassembly

As opposed to traditional realignment based approaches, we canonicalize INDELs in the reads by looking for "bubble flank motifs." In a colored de Bruijn graph, a bubble refers to a location where the graph diverges between two samples. In §3.1.1, we demonstrate how we can use the reconvergence of the de Bruijn graph in the flanking sequence around a bubble to define provably canonical alignments of the bubble between two sequences. For a colored de Bruijn graph containing reads and the reference genome, this allows us to canonically express INDEL variants in the reads against the reference. In §3.1.2, we then show how this approach can be implemented efficiently without building a de Bruijn graph per read, or even adding each read to a de Bruijn graph. Once we have extracted a canonical set of INDELs, we realign the reads to each INDEL sequence using ADAM's INDEL realigner, in known INDELs mode. For a full description of the INDEL realignment process, see Nothaft (2015).

3.1.1 Preliminaries

Our method relies on an *indexed de Bruijn* graph, which is a slight extension of the colored de Bruijn graph (Iqbal *et al.*, 2012). Specifically, each k-mer in an indexed de Bruijn graph knows which sequence position (index) it came from in its underlying read/sequence. To construct an indexed de Bruijn graph, we start with the traditional formulation of a *de Brujin* graph for sequence assembly:

Definition 1 (de Bruijn Graph). *A de Bruijn graph describes the observed transitions between adjacent k-mers in a sequence. Each k-mer s represents a k-length string, with a $k-1$ length prefix given by prefix(s) and a length 1 suffix given by suffix(s). We place a directed edge ($\rightarrow$) from k-mer s_1 to k-mer s_2 if prefix$(s_1)^{\{1,k-2\}}$ + suffix(s_1) = prefix(s_2).*

Now, suppose we have n sequences $\mathcal{S}_1, \ldots, \mathcal{S}_n$. Let us assert that for each k-mer $s \in \mathcal{S}_i$, then the output of function $\text{index}_i(s)$ is defined. This function provides us with the integer position of s in sequence $\mathcal{S}_i$. Further, given two k-mers $s_1, s_2 \in \mathcal{S}_i$, we can define a distance function $\text{distance}_i(s_1, s_2) = |\text{index}_i(s_1) - \text{index}_i(s_2)|$. To create an indexed de Bruijn graph, we simply annotate each k-mer s with the $\text{index}_i(s)$ value for all $\mathcal{S}_i, i \in \{1, \ldots, n\}$ where $s \in \mathcal{S}_i$. This index value is trivial to log when creating the original de Bruijn graph from the provided sequences.

Let us require that all sequences $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are not repetitive, which implies that the resulting de Bruijn graph is acyclic. If we select any two sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ from $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that share at least two k-mers s_1 and s_2 with common ordering ($s_1 \rightarrow \cdots \rightarrow s_2$ in both $\mathcal{S}_i$ and $\mathcal{S}_j$), the indexed de Bruijn graph G provides several guarantees:

1. If two sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ share at least two k-mers s_1 and s_2, we can provably find the maximum edit distance d of the subsequences in $\mathcal{S}_i$ and $\mathcal{S}_j$, and bound the cost of finding this edit distance at $\mathcal{O}(nd)$,[3]
2. For many of the above subsequence pairs, we can bound the cost at $\mathcal{O}(n)$, *and* provide canonical representations for the necessary edits,
3. $\mathcal{O}(n^2)$ complexity is restricted to aligning the subsequences of $\mathcal{S}_i$ and $\mathcal{S}_j$ that exist *before* s_1 or *after* s_2.

Let us focus on cases 1 and 2, where we are looking at the subsequences of $\mathcal{S}_i$ and $\mathcal{S}_j$ that are between s_1 and s_2. A trivial case arises when both $\mathcal{S}_i$ and $\mathcal{S}_j$ contain an identical path between s_1 and s_2 (i.e., $s_1 \rightarrow s_n \rightarrow \cdots \rightarrow s_{n+m} \rightarrow s_2$ and $s_{n+k} \in \mathcal{S}_i \wedge s_{n+k} \in \mathcal{S}_j \forall k \in \{0, \ldots, m\}$). Here, the subsequences are clearly identical. This determination can be made trivially by walking from vertex s_1 to vertex s_2 with $\mathcal{O}(m)$ cost.

However, three distinct cases can arise whenever $\mathcal{S}_i$ and $\mathcal{S}_j$ diverge between s_1 and s_2. For simplicity, let us assume that both paths are independent (see Definition 2). These three cases correspond to there being either a canonical substitution edit, a canonical INDEL edit, or a non-canonical (but known distance) edit between $\mathcal{S}_i$ and $\mathcal{S}_j$.

Definition 2 (Path Independence). *Given a non-repetitive de Bruijn graph G constructed from $\mathcal{S}_i$ and $\mathcal{S}_j$, we say that G contains independent*

[3] Here, $n = \max(\text{distance}_{\mathcal{S}_i}(s_1, s_2), \text{distance}_{\mathcal{S}_j}(s_1, s_2))$.

paths between s_1 and s_2 if we can construct two subsets $\mathcal{S}'_i \subset \mathcal{S}_i, \mathcal{S}'_j \subset \mathcal{S}_j$ of k-mers where $s_{i+n} \in \mathcal{S}'_i \forall n \in \{0, \ldots, m_i\}, s_{i+n-1} \rightarrow s_{i+n} \forall n \in \{1, \ldots, m_i\}$, $s_{j+n} \in \mathcal{S}'_j \forall n \in \{0, \ldots, m_j\}, s_{j+n-1} \rightarrow s_{j+n} \forall n \in \{1, \ldots, m_j\}$, and $s_1 \rightarrow s_i, s_j; s_{i+m_i}, s_{j+m_j} \rightarrow s_2$ and $\mathcal{S}'_i \bigcap \mathcal{S}'_j = \emptyset$, where $m_i = distance_{\mathcal{S}_i}(s_1, s_2)$, and $m_j = distance_{\mathcal{S}_j}(s_1, s_2)$. This implies that the sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ are different between s_1, s_2,

We have a canonical substitution edit if $m_i = m_j = k$, where k is the k-mer size. Here, we can prove that the edit between $\mathcal{S}_i$ and $\mathcal{S}_j$ between s_1, s_2 is a single base substitution k letters after index(s_1):

Proof regarding Canonical Substitution. Suppose we have two non-repetitive sequences, $\mathcal{S}'_i$ and $\mathcal{S}'_j$, each of length $2k + 1$. Let us construct a de Bruijn graph G, with k-mer length k. If each sequence begins with k-mer s_1 and ends with k-mer s_2, then that implies that the first and last k letters of $\mathcal{S}'_i$ and $\mathcal{S}'_j$ are identical. If both subsequences had the same character at position k, this would imply that both sequences were identical and therefore the two paths between s_1, s_2 would not be independent (Definition 2). If the two letters are different *and* the subsequences are non-repetitive, each character is responsible for k previously unseen k-mers. This is the only possible explanation for the two independent k length paths between s_1 and s_2.

To visualize the graph corresponding to a substitution, take the two example sequences `CCACTGT` and `CCAATGT`. These two sequences differ by a `C` $\leftrightarrow$ `A` edit at position three. With k-mer length $k = 3$, this corresponds to the graph in Figure 1.

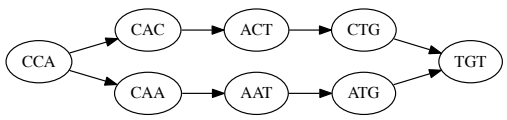

Fig. 1. Subgraph Corresponding To a Single Nucleotide Edit

If $m_i = k - 1, m_j \geq k$ or vice versa, we have a canonical INDEL edit (for convenience, we assume that $\mathcal{S}'_i$ contains the $k - 1$ length path). Here, we can prove that there is a $m_j - m_i$ length insertion[4] in $\mathcal{S}'_j$ relative to $\mathcal{S}'_i$, $k - 1$ letters *after* index(s_1):

Lemma 1 (Distance between k length subsequences). Indexed de Bruijn *graphs naturally provide a distance metric for k length substrings. Let us construct an indexed de Bruijn graph G with k-mers of length k from a non-repetitive sequence $\mathcal{S}$. For any two k-mers $s_a, s_b \in \mathcal{S}, s_a \neq s_b$, the $distance_{\mathcal{S}}(s_a, s_b)$ metric is equal to $l_p + 1$, where l_p is the length of the path (in k-mers) between s_a and s_b. Thus, k-mers with overlap of $k - 1$ have an edge directly between each other ($l_p = 0$) and a distance metric of 1. Conversely, two k-mers that are adjacent but not overlapping in $\mathcal{S}$ have a distance metric of k, which implies $l_p = k - 1$.*

Proof regarding Canonical INDELs. We are given a graph G which is constructed from two non-repetitive sequences $\mathcal{S}'_i$ and $\mathcal{S}'_j$, where the only two k-mers in both $\mathcal{S}'_i$ and $\mathcal{S}'_j$ are s_1 and s_2 and both sequences provide independent paths between s_1 and s_2. By Lemma 1, if the path from $s_1 \rightarrow \cdots \rightarrow s_2 \in \mathcal{S}'_i$ has length $k - 1$, then $\mathcal{S}'_i$ is a string of length $2k$ that is formed by concatenating s_1, s_2. Now, let us suppose that the path from $s_1 \rightarrow \cdots \rightarrow s_2 \in \mathcal{S}'_j$ has length $k + l - 1$. The first l k-mers after s_1 will introduce a l length subsequence $\mathcal{L} \subset \mathcal{S}'_j, \mathcal{L} \not\subset \mathcal{S}'_i$, and then the remaining $k - 1$ k-mers in the path provide a transition from $\mathcal{L}$ to s_2. Therefore, $\mathcal{S}'_j$ has length of $2k + l$, and is constructed by concatenating $s_1, \mathcal{L}, s_2$. This provides a canonical placement for the inserted sequence $\mathcal{L}$ in $\mathcal{S}'_j$ between s_1 and s_2.

To visualize the graph corresponding to a canonical INDEL, take the two example sequences `CACTGT` and `CACCATGT`. Here, we have a `CA` insertion after position two. With k-mer length $k = 3$, this corresponds to the graph in Figure 2.

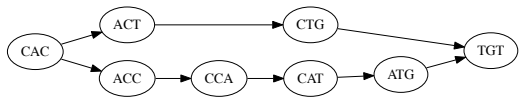

Fig. 2. Subgraph Corresponding To a Canonical INDEL Edit

Where we have a canonical allele, the cost of computing the edit is set by the need to walk the graph linearly from s_1 to s_2, and is therefore $\mathcal{O}(n)$. However, in practice, we will see differences that cannot be described as one of the earlier two canonical approaches. First, let us generalize from the two above proofs: if we have two independent paths between s_1, s_2 in the de Bruijn graph G that was constructed from $\mathcal{S}_i, \mathcal{S}_j$, we can describe $\mathcal{S}_i$ as a sequence created by concatenating $s_1, \mathcal{L}_i, s_2$.[5] The canonical edits merely result from special cases:

- In a canonical substitution edit, $l_{\mathcal{L}_i} = l_{\mathcal{L}_j} = 1$.
- In a canonical INDEL edit, $l_{\mathcal{L}_i} = 0, l_{\mathcal{L}_j} \geq 1$.

Conceptually, a non-canonical edit occurs when two edits occur within k positions of each other. In this case, we can trivially fall back on a $O(nm)$ local alignment algorithm (e.g., a pairwise HMM or Smith-Waterman, see Durbin *et al.* (1998); Smith and Waterman (1981)), *but* we only need to locally realign $\mathcal{L}_i$ against $\mathcal{L}_j$, which reduces the size of the realignment problem. However, we can further limit this bound by limiting the maximum number of INDEL edits to $d = |l_{\mathcal{L}_i} - l_{\mathcal{L}_j}|$. This allows us to use an alignment algorithm that limits the number of INDEL edits (e.g., Ukkonen's algorithm (Ukkonen, 1985)). By this, we can achieve $O(n(d + 1))$ cost.

3.1.2 Implementation

As alluded to earlier in this section, we can use this indexed de Bruijn concept to canonicalize INDEL variants without needing to first build a de Bruijn graph. The insight behind this observation is simple: any section of a read alignment that is an exact sequence match with length greater than our k-mer length maps to a section of the indexed de Bruijn graph where the read and reference paths have converged. As such, we can use these segments that are perfect sequence matches to anchor the bubbles containing variants (areas where the read and reference paths through the graph diverge) without first building a graph. We can perform this process simply by parsing the CIGAR string (and MD tags) for each read (Li *et al.*, 2009a). We do this by:

- Iterating over each operator in the CIGAR string. We coalesce the operators into a structure that we call an "alignment block":
- If the operator is a sequence match (CIGAR `=`, or CIGAR `M` with MD tag indicating an exact sequence match) that is longer than our k-mer length, we can create an alignment block that indicates a convergence in the indexed de Bruijn block (a sequence match block).
- If the sequence match operator is adjacent to an operator that indicates that the read diverges from the reference (insertion, deletion, or

[4] This is equivalently an $m_j - m_i$ length deletion in $\mathcal{S}'_i$ relative to $\mathcal{S}'_j$.

[5] This property holds true for $\mathcal{S}_j$ as well.

sequence mismatch), we then take k bases from the start/end of the matching sequence and append/prepend the k bases to the divergent sequence. We then create an alignment block that indicates that the read and reference diverge, along with the two diverging sequences, flanked by k bases of matching sequence on each side. We call these blocks realignment blocks.

- We then loop over each alignment block. Since the sequence match blocks are exact sequence matches, they do not need any further processing and can be directly emitted as a CIGAR = operator. If the block is a realignment block, we then apply the observations from §3.1.1. Again, we can apply our approaches without building de Bruijn graphs for the bubble. Specifically, both of the canonical placement rules that we formulate in §3.1.1 indicate that the variant in a bubble can be recovered by trimming any matching flanking sequence. We begin by trimming the matching sequences from the reference and read, starting from the right, followed by the left. We then emit a CIGAR insertion, deletion, or sequence mismatch (`X`) operator for this block, along with a match operator if either side of the flanking sequence was longer than k.

This process is very efficient, as it can be done wholly with standard string operators in a single loop over the read. To avoid the cost of looking up the reference sequence from a reference genome, we require that all reads are tagged with the SAM `MD` tag. This allows us to reconstruct the reference sequence for a bubble from the read sequence and CIGAR.

One problem with this method is that it can be misled by sequencing errors that are proximal to a true variant. As can be seen in §4.1, solely using our bubble flanking algorithm to clean up INDEL alignments leads to lower accuracy than the state-of-the-art toolkit. However, if the INDEL variant in a read that is discovered is a true variant, it is a good candidate to be used as an input to a local realignment scheme. To implement this approach, we used our bubble flanking algorithm to canonicalize INDEL variants, and then we used our variant discovery algorithm (see §3.2.1) with filtration disabled to collect all canonical INDELs. We then fed these INDELs and our input reads into ADAM's INDEL realignment engine (Massie *et al.*, 2013; Nothaft *et al.*, 2015). This tool is based on the `GATK`'s INDEL realigner (DePristo *et al.*, 2011), and calculates the quality-score weighted Hamming edit distance between a set of reads, a consensus sequence (a haplotype containing a potential INDEL variant), and the reference sequence. If the sum weighted edit distance between the reads and the consensus sequence represents a suffient improvement over the sum weighted edit distance between the reads and the reference genome, the read alignments are moved to their lowest weighted edit distance position relative to the consensus sequence. A detailed description of this algorithm can be found in Nothaft (2015). As seen in §4.1, coupling local realignment with our INDEL canonicalization scheme improves SNP calling accuracy to comparable with the state-of-the-art, while improving INDEL calling accuracy by 2–5%.

3.2 Genotyping

Avocado performs genotyping as a several stage process where variants are discovered from the input reads and filtered, joined back against the input reads, and then scored. We use a biallelic likelihood model to score variants (Li, 2011), and run all stages in parallel. Our approach does not rely on the input reads being sorted, and as such, is not unduly impacted by variations in coverage across the genome. This point is critical in a parallel approach, as coverage can vary dramatically across the genome (Pinard *et al.*, 2006). If the input reads must be sorted, this can lead to large work imbalances between nodes in a distributed system, which negatively impacts strong scaling. An alternative approach is to use previously known data about genome coverage to statically partition tasks into balanced chunks Chiang *et al.* (2015). Unlike the static partitioning approach used by SpeedSeq that discards regions with very high coverage, this allows us to call variants in regions with very high coverage. However, as is also noted in the SpeedSeq paper, variant calls in these regions are likely to be caused by artifacts in the reference genome that confound mapping and thus are uninformative or spurious, and are hard filtered by our pipeline (see §3.2.3).

3.2.1 Variant Discovery and Overlapping

To identify a set of variants to score, we scan over all of the input reads, and generate a set of variants per read where each variant is tagged with the mean quality score of all bases in the read that were in this variant. We then use Apache Spark's `reduceByKey` functionality to compute the number of times each variant was observed with high quality. We do this to discard sequence variants that were observed in a read that represent a sequencing error, and not a true variant. In our evaluation, we set the quality needed to consider a variant observation as high quality to Phred 18 (equivalent to a error probability of less than 0.016), and we require that a variant is seen in at least 3 reads.

To score the discovered variants, we use an "overlap join" primitive to find all of the variants that a single read overlaps. An overlap join is a relational join where the row equality function is defined as whether two objects overlap in the genomic coordinate space (Nothaft *et al.*, 2015). This primitive can be implemented in a distributed system as both a broadcast join (the smaller of the two datasets is sent to every node in the cluster), or as a sort-merge join, where the dataset is sorted. Our implementation uses a broadcast strategy, as the set of variants to score is typically small and this approach eliminates the work imbalance problem introduced earlier.

Our broadcast overlap join implementation starts by sorting the candidate variants by genomic locus. We collect the variants to the leader node, and then broadcast a sorted array of variants to each node in the cluster. To find all of the variants that overlap a single read, we run a binary search across the sorted array of variants. We prefer this strategy to building an indexed datastructure (such as an interval tree, see Kozanitis and Patterson (2016)) because sorting can be efficiently parallelized across the Apache Spark cluster, while building an indexed structure would typically need to be done sequentially on a single node. Additionally, a flat array of sorted variants is simpler to serialize and broadcast across the cluster than an indexed structure. When we query into the sorted array using binary search, the binary search algorithm will give us a variant that is overlapped by the read. Since we actually want to run a combined join-and-group query, we then search outwards from this first hit to identify all of the variants that overlap the read alignment.

One of the reasons that we filter out variant sites that are not supported by many high quality reads is an engineering limitation currently in Avocado. As we decrease the stringency of the filters and allow more variants to be detected, we increase the amount of variants that we need to broadcast between nodes. This causes the size of data that we must serialize to grow beyond the size of the maximum individual item that we can serialize (limited to 2GB due to the Java Virtual Machine). We are working to eliminate this limitation. There are several possible strategies. A simple strategy would be to reduce the amount of data written to the serialization buffer by compressing the data before streaming it into the serialization buffer. However, our sorted array currently stores the genomic coordinate of a variant separately from the variant itself, which causes a minor amount of data duplication in memory. By eliminating this data duplication, we should be able to eliminate this engineering constraint.

3.2.2 Genotyping Model

Once we have joined our reads against our variants, we score each read using the biallelic genotyping model proposed by Li (2011). For each variant, we check to see if the variant allele is present in the read at the

appropriate position in the alignment. If the variant is present, we treat the read as positive evidence supporting the variant. If the read contains the reference allele at that site, we treat the read as evidence supporting the reference. If the read neither matches the variant allele nor the reference, we do not use the read to calculate the genotype likelihoods, but we do use the read to compute statistics (e.g., for calculating depth, strand bias, etc.) about the genotyped site. We calculate the genotype likelihood for the genotype in log space, using Equation (2). Equation (2) is not our contribution and is reproduced from Li (2011), but in log space.

$$\log \mathcal{L}(g) = -mk \sum_{i=0} j l_r(g, m-g, \epsilon_i) \sum_{i=j+1}^{k} l_r(m-g, g, \epsilon_i) \tag{2}$$

$$l_r(c_r, c_a, \epsilon) = \text{logsum}(\log c_r + \log \epsilon, \log c_a + \text{logm1}(\log \epsilon)) \tag{3}$$

In Equation (2), g is the genotype state (number of reference alleles), m is the copy number at the site, k is the total number of reads, j is the number of reads that match the reference genome, and ϵ is the error probability of a single read base, as given by the harmonic mean of the read's base quality, and the read's mapping quality, if present. The logsum function adds two numbers that are in log space, while logm1 computes the additive inverse of a number in log space. These functions can be implemented efficiently while preserving numerical stability (Durbin *et al.*, 1998). By doing this whole calculation in log space, we can eliminate issues caused by floating-point underflow. Additionally, since ϵ is derived from Phred scaled quantities and is thus already in log space (base ten), while g and $m - g$ are constants that can be pre-converted to log space. For all sites, we also compute a reference model that can be used in joint genotyping in a gVCF approach. Additionally, we support a gVCF mode where all sites are scored, even if they are not covered by a putative variant.

We compute the likelihoods for each read in parallel. This function maps over all of the reads, and emits a set of records describing each observation. In addition to storing the likelihood vector per read/variant pair, this record contains data necessary to compute several genotype annotations that are used for variant filtration (such as strand bias observations, mapping quality, etc., see §3.2.3). We use Apache Spark's `reduceByKey` function to merge all of the observations for a given locus. Once we have merged all of the observations for a given site, we call the genotype state by taking the genotype state with the highest likelihood. In single sample mode, we assume no prior probability. We support a joint variant calling mode that computes reference allele frequency for use in a binomial prior probability distribution.

3.2.3 Variant Filtration

Once we have called variants, we pass the calls through a hard filtering engine. First, unless we are in gVCF mode, we discard all homozygous reference calls and low quality genotype calls (default threshold is Phred 30). Additionally, we provide several hard filters that retain the genotype call, but mark the call as filtered. These include:

1. Quality by depth: the Phred scaled genotype quality divided by the depth at the site. Default value is 2.0 for heterozygous variants, 1.0 for homozygous variants. The value can be set separately for INDELs and SNPs.
2. Root-mean-square mapping quality: Default value is 30.0 for SNPs. By default, this filter is disabled for INDELs.
3. Depth: We filter out genotype calls below a minimum depth, or above a maximum depth. By default, the minimum depth is 10, and maximum depth is 200. This value can be set separately for INDELs and SNPs.

Currently, we do not support filtering variant sites in joint genotyping mode. However, we will add this functionality soon.

4 Results

4.1 Accuracy

To benchmark Avocado's accuracy, we used the high coverage, PCR-free whole genome sequencing (WGS) run of NA12878 from the 1,000 Genomes project (1000 Genomes Project Consortium, 2015). We chose this dataset because NA12878 has extensive orthogonal verification data that is available through the National Institute for Standards and Time's (NIST's) Genome-in-a-Bottle (GIAB) project (Zook *et al.*, 2015), and the WGS preparation of NA12878 for the 1,000 Genomes project[6] is more representative of a typical sequencing dataset than the GIAB 300× coverage WGS data for NA12878. We verified our calls using the Global Alliance for Genomics and Health's (GA4GH's) benchmarking suite[7] As our input reads are from an Illumina sequencing platform, we limited our evaluation to variants that the GIAB's orthogonal validation mechanism were able to observe on an Illumina sequencing platform.

We processed the data through Avocado, and the GATK's HaplotypeCaller (DePristo *et al.*, 2011). We ran the HaplotypeCaller with the "Best Practices" settings, using Toil (Vivian *et al.*, 2016)[8]. Table 1 compares the accuracy of the two tools. We include two columns for Avocado: Avocado 1 uses only the bubble flanking realignment code, while Avocado 2 uses the bubble flanking realigner to generate candidate variants that are aligned using the ADAM INDEL realigner. These two modes are described in §3.1.2. As mentioned in §3.1.2, we require our input data to have MD tags. As the aligned dataset that we used for evaluation is missing MD tags on some reads, we recomputed the full set of MD tags using ADAM's `transform` command prior to running Avocado.

Table 1. Accuracy on NA12878

	GATK	**Avocado 1**	**Avocado 2**
Unfiltered			
SNP Recall	99.9%	97.4%	99.7%
SNP Precision	98.8%	97.3%	99.4%
INDEL Recall	99.2%	84.3%	88.4%
INDEL Precision	98.5%	86.3%	89.0%
Filtered			
SNP Recall	99.9%	97.2%	99.6%
SNP Precision	99.0%	98.2%	99.8%
INDEL Recall	99.2%	79.4%	83.5%
INDEL Precision	98.6%	91.2%	94.0%
Single Node Runtime	35h49m	19h55m	—
Cluster Runtime	–	21m	46m

Avocado and the GATK HaplotypeCaller have comparable SNP calling accuracy, with Avocado having higher (0.8%) precision and the HaplotypeCaller having slightly higher (0.3%) recall. The

[6] The high coverage NA12878 data is available from `ftp://ftp-trace.ncbi.nlm.nih.gov/1000genomes/ftp/phase3/data/NA12878/high_coverage_alignment/NA12878.mapped.ILLUMINA.bwa.CEU.high_coverage_pcr_free.20130906.bam`.

[7] Available from `https://github.com/ga4gh/benchmarking-tools`. Also, see Paten *et al.* (2015).

[8] The Toil scripts used can be found at `https://github.com/BD2KGenomics/toil-scripts/tree/master/src/toil_scripts/gatk_germline`.

HaplotypeCaller outperforms in INDEL calling. However, this is due to a bug in Avocado that causes us to erroneously call homozygous alternate INDELs as heterozygous sites, even in the presence of read evidence strongly supporting a heterozygous INDEL call. We are currently working to eliminate this bug, which affects approximately 10% of INDELs in the GIAB callset (31,107 out of 339,091 total INDELs). This bug negatively impacts both INDEL calling precision and accuracy, as the call is treated as both a false negative (missing homozygous call) and a false positive (heterozygous call not in validation set). By addressing this error, our INDEL recall will improve up to 96.6%, and our INDEL precision will improve up to 97.2%.

4.2 Performance

As illustrated in Table 1, Avocado is approximately 45% faster than the GATK HaplotypeCaller when run on a single node, using the fast realignment (bubble canonicalization only mode). We did not have the opportunity to run Avocado's full realignment mode on a single node. However, this mode is approximately 2.2× slower than the bubble canonicalization mode when run on a cluster. As such, we expect it to be approximately 20% slower than the GATK when run on a single node. However, Avocado can be efficiently parallelized across more than 1,000 cores.

Figure 3A demonstrates Avocado's strong scaling. In this experiment, we ran Avocado's INDEL realigner and genotyper on a constant dataset (the high coverage NA12878 dataset from 1,000 Genomes (1000 Genomes Project Consortium, 2015)), while varying the size of the cluster that Avocado was run on. This experiment was run on our in-house cluster, which contains 64 machines connected by a full-bisection bandwidth 10 gigabit ethernet network. Each machine in this cluster runs a 16-core Intel Xeon E5-2670 at 2.6GHz, with 256GB of memory. Each node has four 1TB hard disk drives, and data was stored in the Apache Hadoop Distributed File System (HDFS, (Shvachko *et al.*, 2010)). The cluster resources are managed by Apache YARN (Vavilapalli *et al.*, 2013). We ran Apache Hadoop 2.6.2, and Apache Spark 1.6.2. Due to cluster maintainance, several nodes were out of commission, limiting us to 896 cores for our experiments.

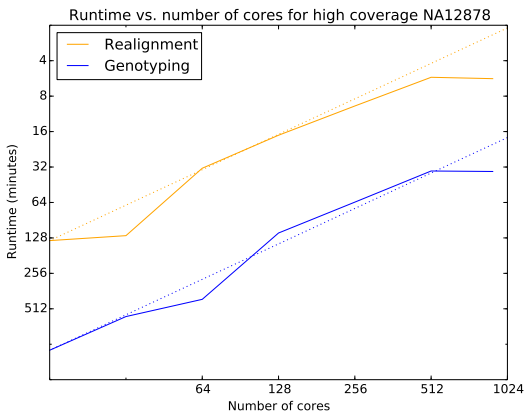

Fig. 3. Runtime of the INDEL realignment and genotyping algorithms as the number of cores is changed. Note that the times on the Y-axis are descending; as the number of cores made available to Avocado is doubled, runtime decreases. The dotted lines represent ideal scaling, where runtime halves when the number of cores in the cluster is doubled.

Avocado demonstrates linear strong scaling out to the full size of our cluster. This is due to the even distribution of work across all of the nodes in our cluster. This can be seen in Figure 3B, which shows that task completion times are fairly uniform within each stage of the realignment and genotyping process.

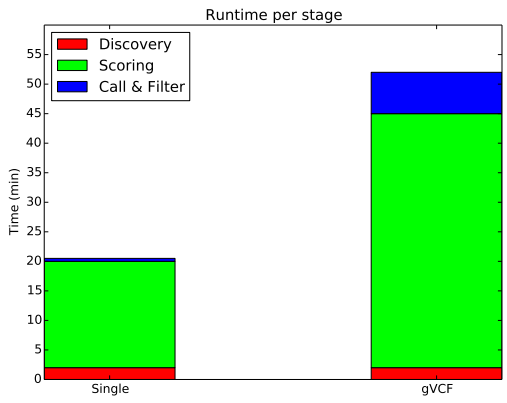

Fig. 4. Runtime of the different stages in the genotyper core.

Figure 4 contains a performance comparison of the genotyping engine when running in both single sample and gVCF mode. This comparison excludes the realignment stages, which are outside of the genotyper core, and which are common to both stages. Loading/filtering the input reads and discovering variants is a small fraction of the total runtime in each mode. Scoring the variants against the reads consumes the majority of the runtime in both modes. As we move from single sample to gVCF mode, the time this stages takes to complete increases by 2.3× from 18 minutes to 42 minutes. This increase is expected, as we need to merge scores across all of the sites in the genome in the gVCF mode, which drastically increases the amount of computation we must do. This is also reflected in the final stage, where we emit the final calls and filter variants. The runtime of this stage increases by 14× from half a minute up to seven minutes.

5 Discussion

In addition to Avocado, there have been several other attempts to develop variant callers that run using Apache Hadoop/Spark. An early approach was Crossbow (Langmead *et al.*, 2009a), which used Apache Hadoop to invoke SoapSNP (Li *et al.*, 2009b) and Bowtie (Langmead *et al.*, 2009b) on a cloud computing cluster. Unlike Crossbow, which used Apache Hadoop to orchestrate single node tools on a cluster, Avocado is written to natively use Apache Spark's core abstractions to run variant calling. Additionally, there are several other variant callers under development that use Apache Spark. These include Guacamole[9], which performs both germline and somatic variant calling, and the development of the new GATK4[10].

Although this manuscript has focused on single sample variant calling in Avocado, we also support joint variant calling across samples. When joint calling, we support using gVCF data as input, where each site includes likelihoods for the reference genome. We load in these likelihood observations, and filter all sites that were called as a variant in at least one sample. Similar to how we run genotyping (see §3.2), we overlap the variant sites against the likelihood observations by running a join. Once we have completed this, we use the expectation-maximization (E-M)

[9] `https://github.com/hammerlab/guacamole/`
[10] `https://github.com/broadinstitute/gatk`

procedure described in Li (2011) to compute the major allele frequency. We run for a fixed number of iterations of the E-M loop before using the major allele frequency with a binomial distribution to compensate the genotype likelihoods. These compensated genotype likelihoods are then used to emit a "squared-off" genotype matrix. We have not yet validated our joint variant caller, and consider it as future work.

6 Conclusion

In this paper, we have introduced Avocado, a distributed variant caller built on top of Apache Spark (Zaharia *et al.*, 2010, 2012) and the ADAM library for parallel genomic analysis (Massie *et al.*, 2013; Nothaft *et al.*, 2015). Avocado is open source software released under the permissive Apache 2 license, and makes use of a novel variant canonicalization engine that reduces the runtime complexity of the local reassembly algorithms that are commonly used for variant discovery and calling. By combining our novel variant canonicalization engine with efficient parallelization of tasks across a distributed system, we are able to reduce the latency of calling variants on a single human genome to under one hour on a 896 core cluster of commodity computers. We do this while achieving SNP precision and recall that is competitive with the current state of the art, and high INDEL precision.

Acknowledgements

The authors would like to thank Timothy Danford and John St. John, who have contributed to the architecture of Avocado through their work on somatic variant calling, and Michael Heuer, who has been instrumental in leading the development and refinement of ADAM's variant and genotype schemas and APIs.

Funding

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, NIH BD2K Award 1-U54HG007990-01, NIH Cancer Cloud Pilot Award HHSN261201400006C and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC, Ericsson, Facebook, Guavus, Huawei, Intel, Microsoft, NetApp, Pivotal, Samsung, Splunk, Virdata, VMware, and Yahoo!. Author FAN is supported by a National Science Foundation Graduate Research Fellowship.

References

1000 Genomes Project Consortium (2015). A global reference for human genetic variation. *Nature*, **526**(7571), 68–74.

Albers, C. A., Lunter, G., MacArthur, D. G., McVean, G., Ouwehand, W. H., and Durbin, R. (2011). Dindel: accurate indel calls from short-read data. *Genome research*, **21**(6), 961–973.

Bao, R., Huang, L., Andrade, J., Tan, W., Kibbe, W. A., Jiang, H., and Feng, G. (2014). Review of current methods, applications, and data management for the bioinformatics analysis of whole exome sequencing. *Cancer informatics*, **13**(Suppl 2), 67.

Bloniarz, A., Talwalkar, A., Terhorst, J., Jordan, M. I., Patterson, D., Yu, B., and Song, Y. S. (2014). Changepoint analysis for efficient variant calling. In *Research in Computational Molecular Biology*, pages 20–34. Springer.

Chiang, C., Layer, R. M., Faust, G. G., Lindberg, M. R., Rose, D. B., Garrison, E. P., Marth, G. T., Quinlan, A. R., and Hall, I. M. (2015). SpeedSeq: Ultra-fast personal genome analysis and interpretation. *Nature methods*, **12**(10), 966–968.

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

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

Fang, H., Wu, Y., Narzisi, G., O'Rawe, J. A., Barrón, L. T. J., Rosenbaum, J., Ronemus, M., Iossifov, I., Schatz, M. C., and Lyon, G. J. (2014). Reducing INDEL calling errors in whole genome and exome sequencing data. *Genome Med*, **6**, 89.

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

Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., and McVean, G. (2012). De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature genetics*, **44**(2), 226–232.

Kingsford, C., Schatz, M. C., and Pop, M. (2010). Assembly complexity of prokaryotic genomes using short reads. *BMC bioinformatics*, **11**(1), 21.

Kozanitis, C. and Patterson, D. A. (2016). GenAp: A distributed SQL interface for genomic data. *BMC bioinformatics*, **17**(1), 63.

Landau, G. M. and Vishkin, U. (1986). Efficient string matching with k mismatches. *Theoretical Computer Science*, **43**, 239–249.

Langmead, B., Schatz, M. C., Lin, J., Pop, M., and Salzberg, S. L. (2009a). Searching for SNPs with cloud computing. *Genome biology*, **10**(11), R134.

Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. L. (2009b). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, **10**(3), R25.

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

Li, H. (2014). Towards better understanding of artifacts in variant calling from high-coverage samples. *arXiv preprint arXiv:1404.0929*.

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

Li, R., Li, Y., Fang, X., Yang, H., Wang, J., Kristiansen, K., and Wang, J. (2009b). SNP detection for massively parallel whole-genome resequencing. *Genome research*, **19**(6), 1124–1132.

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

Narzisi, G., O'Rawe, J. A., Iossifov, I., Fang, H., Lee, Y.-h., Wang, Z., Wu, Y., Lyon, G. J., Wigler, M., and Schatz, M. C. (2014). Accurate de novo and transmitted indel detection in exome-capture data using microassembly. *Nature methods*, **11**(10), 1033–1036.

Nothaft, F. A. (2015). Scalable genome resequencing with adam and avocado. Technical report, UCB/EECS-2015-65, EECS Department, University of California, Berkeley.

Nothaft, F. A., Massie, M., Danford, T., Zhang, Z., Laserson, U., Yeksigian, C., Kottalam, J., Ahuja, A., Hammerbacher, J., Linderman, M., Franklin, M., Joseph, A. D., and Patterson, D. A. (2015). Rethinking data-intensive science using scalable analytics systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM.

Paten, B., Diekhans, M., Druker, B. J., Friend, S., Guinney, J., Gassner, N., Guttman, M., Kent, W. J., Mantey, P., Margolin, A. A., *et al.* (2015). The NIH BD2K center for Big Data in Translational Genomics. *Journal of the American Medical Informatics Association (JAMIA)*, **22**(6), 1143–1147.

Pinard, R., de Winter, A., Sarkis, G. J., Gerstein, M. B., Tartaro, K. R., Plant, R. N., Egholm, M., Rothberg, J. M., and Leamon, J. H. (2006). Assessment of whole genome amplification-induced bias through high-throughput, massively parallel whole genome sequencing. *BMC Genomics*, **7**(1), 216.

Rimmer, A., Phan, H., Mathieson, I., Iqbal, Z., Twigg, S. R., Wilkie, A. O., McVean, G., Lunter, G., WGS500 Consortium, *et al.* (2014). Integrating mapping-, assembly-and haplotype-based approaches for calling variants in clinical sequencing applications. *Nature genetics*, **46**(8), 912–918.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The Hadoop distributed file system. In *Proceedings of the Symposium on Mass Storage Systems and Technologies (MSST '10)*, pages 1–10. IEEE.

Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of molecular biology*, **147**(1), 195–197.

Talwalkar, A., Liptrap, J., Newcomb, J., Hartl, C., Terhorst, J., Curtis, K., Bresler, M., Song, Y. S., Jordan, M. I., and Patterson, D. (2014). SMASH: A benchmarking toolkit for human genome variant calling. *Bioinformatics*, page btu345.

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

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., *et al.* (2013). Apache Hadoop YARN: Yet

another resource negotiator. In *Proceedings of the Symposium on Cloud Computing (SoCC '13)*, page 5. ACM.

Vivian, J., Rao, A., Nothaft, F. A., Ketchum, C., Armstrong, J., Novak, A., Pfeil, J., Narkizian, J., Deran, A. D., Musselman-Brown, A., *et al.* (2016). Rapid and efficient analysis of 20,000 RNA-seq samples with Toil. *bioRxiv*, page 062497.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: cluster computing with working sets. In *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10)*, page 10.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*, page 2. USENIX Association.

Zook, J. M., Chapman, B., Wang, J., Mittelman, D., Hofmann, O., Hide, W., and Salit, M. (2015). Integrating human sequence data sets provides a resource of benchmark SNP and INDEL genotype calls. *Nature*, **201**, 4.