Scalable Systems and Algorithms for Genomic Variant Analysis

by

Frank Austin Nothaft

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Anthony Joseph, Chair
Professor David Patterson, Co-chair
Professor Haiyan Huang

Fall 2017

The dissertation of Frank Austin Nothaft, titled Scalable Systems and Algorithms for Genomic Variant Analysis, is approved:

Chair ____ Date ______________

Co-chair ____ Date ______________

____ Date ______________

University of California, Berkeley

Scalable Systems and Algorithms for Genomic Variant Analysis

Abstract

Scalable Systems and Algorithms for Genomic Variant Analysis

by

Frank Austin Nothaft

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Anthony Joseph, Chair

Professor David Patterson, Co-chair

With the cost of sequencing a human genome dropping below $1,000, population-scale sequencing has become feasible. With projects that sequence more than 10,000 genomes becoming commonplace, there is a strong need for genome analysis tools that can scale across distributed computing resources while providing reduced analysis cost. Simultaneously, these tools must provide programming interfaces and deployment models that are easily usable by biologists.

In this dissertation, we describe the ADAM system for processing large genomic datasets using distributed computing. ADAM provides a decouped stack-based architecture that can accomodiate many data formats, deployment models, and data access patterns. Additionally, ADAM defines schemas that describe common genomic datatypes. ADAM's schemas and programming models enable the easy integration of disparate genomic datatypes and datasets into a single analysis.

To validate the ADAM architecture, we implemented an end-to-end variant calling pipeline using ADAM's APIs. To perform parallel alignment, we developed the Cannoli tool, which uses ADAM's APIs to automatically parallelize single node aligners. We then implemented GATK-style alignment refinement as part of ADAM. Finally, we implemented a biallelic genotyping model, and novel reassembly algorithms in the Avocado variant caller. This pipeline provides state-of-the-art SNV calling accuracy, along with high (97%) INDEL calling accuracy. To further validate this pipeline, we reanalyzed 270 samples from the Simons Genome Diversity Dataset.

Contents

Part I

Introduction and Principles

Chapter 1

Introduction

The rapid decrease in sequencing cost has made large scale sequencing tractible. The dramatic improvement in sequencing cost since the Human Genome Project has enabled a human whole genome sequence (WGS) to be generated for under $1,000 in wetlab costs [36]. This trend will continue for the forseeable future, as sequencing vendors like Illumina unveil new sequencers such as the NovaSeq that provide even higher throughput while also decreasing cost, and as radically new sequencing technologies like Oxford Nanopore come online [17]. The reduced cost of sequencing enables the use of genome sequencing in population health research projects and clinical practice. As a result, the total volume of sequencing data produced is expected to exceed that of YouTube by 2021 [50].

The massive scale of the sequencing data this is being acquired enables novel insight into biological phenomena. The Exome Aggregation project [23]–now gnomAD–provides an especially powerful demonstration: by sequencing more than 60,000 exomes, we have been able to better understand the impact of genomic variation on prion disease [32] and cardiovascular disease [56], and to better characterize the effect of structural variation [43]. However, this scale of data solves biological problems at the cost of techincal and logistical problems. Data storage and transfer has become a serious problem, and the focus of many researchers [14, 21] and standards organizations [40]. Not only is the volume of data large, but expensive processing is needed to analyze the data. Due to historical design decisions, much of this processing is currently restricted to single node architectures that assume POSIX storage APIs. As a result, it can take well upwards of 100 hours to analyze the raw read data from a single genome.

We believe that distributed computing architectures are a good match for genomic data analysis. Horizontally scalable storage architectures can simultaneously provide increased data storage capacities, data access throughput, and reduced storage cost. Because most genomic analyses are centered on analyzing the genomic data at disparate genomic loci without coordination betwen loci, most genomic analysis tasks can be executed in parallel. Even more importantly, these analysis patterns cleanly map onto quasi-relational primitives that are powerful and can be executed in parallel. Finally, by building upon widely used open-source distributed processing architectures like Apache Spark [57] and Hadoop [3], genomics can benefit from engineering contributions from a broader swath of engineering.

In this thesis, we introduce ADAM, an application programming interface (API) for processing genomic data using Apache Spark. ADAM is based around a novel stack-oriented architecture that uses schemas to define the narrow waist in the stack. On top of the schemas, we provide high-level APIs that allow computational biologists and bioinformaticians to manipulate collections of genomic data in a parallel fashion. The high level APIs extend Apache Spark's Resilient Distributed Dataset (RDD, see Zahaira et al. [57]) abstraction with genomics-specific functionality, and eliminates the low level "walker" pattern [31] that is common in genomics. At lower levels in the stack, we provide efficient implementations of the common genomics query models. By having clearly defined APIs between each level of the stack, we are able to exchange layers to optimize query performance for a given query, input data type, or cluster/cloud configuration.

Our work on ADAM has resulted in the broad ecosystem of projects depicted in Figure 1.1. We refer to the tools built on ADAM as the "Big Data Genomics" (BDG) project. In this dissertation, we will limit our focus to ADAM's architecture and APIs, and the tools and algorithms that form the core components of the BDG variant calling pipeline:

- Cannoli, which parallelizes single node genomic data processing tools. Cannoli is used in our pipeline for alignment.
- The ADAM read transformations, which correct for errors in the aligned reads.
- Avocado, a fully parallelized variant caller.

This pipeline is able to call variants on a high coverage (60×) whole genome in under one hour when running on commodity cloud computing resources. This represents a dramatic improvement in performance over the widely used Genome Analysis Toolkit (GATK, see DePristo et al. [11]), which needed over 100 hours to call variants on the same sample. These tools demonstrate how ADAM's APIs enable bioinformatics analyses to be written at a high level, while also allowing for the reuse of code from legacy bioinformatics tools.

In this dissertation, we begin by describing the deluge of genomic data and the tools that are used to process, manipulate, and store genomic data. In part two, we then describe the requirements for a distributed genomic data analysis framework and introduce ADAM's architecture. Part three describes how we built the Cannoli-ADAM-Avocado variant calling pipeline on top of ADAM's architecture, and the novel algorithms and architectural refinements that were needed. We then validate the accuracy of this pipeline in part four using ground truth datasets [60] and a large scale sequencing project [29]. We conclude by describing open problems and future directions for this research, as well as the impact of the Big Data Genomics/ADAM project.

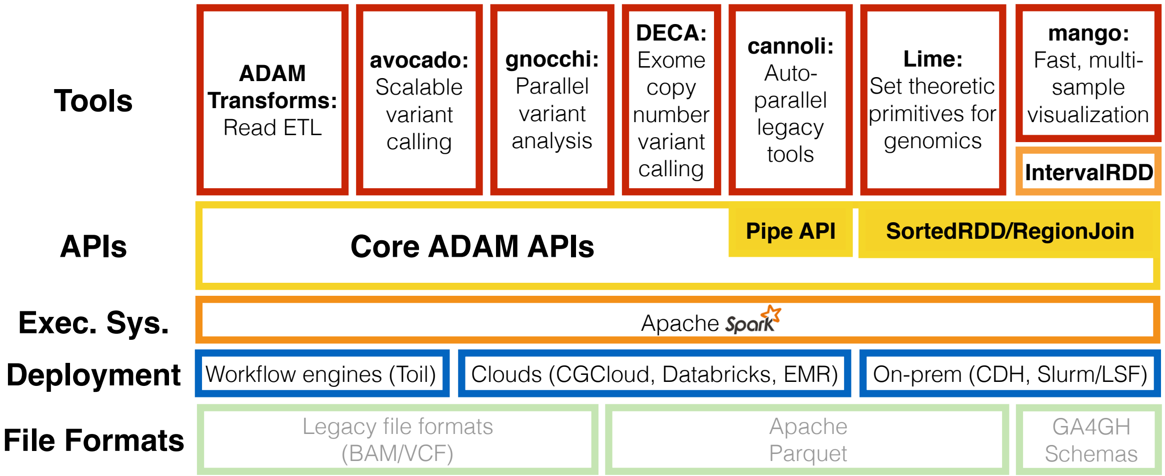
Tools
ADAM Transforms: Read ETL
avocado: Scalable variant calling
gnocchi: Parallel variant analysis
DECA: Exome copy number variant calling
cannoli: Auto-parallel legacy tools
Lime: Set theoretic primitives for genomics
mango: Fast, multi-sample visualization
IntervalRDD
APIs
Core ADAM APIs
Pipe API
SortedRDD/RegionJoin
Exec. Sys.
Apache Spark
Deployment
Workflow engines (Toil)
Clouds (CGCloud, Databricks, EMR)
On-prem (CDH, Slurm/LSF)
File Formats
Legacy file formats (BAM/VCF)
Apache Parquet
GA4GH Schemas

Figure 1.1: The Big Data Genomics ecosystem. Our work on ADAM has built a framework for scalable genomics using external projects like Apache Spark [57] and Apache Parquet [4]. On top of ADAM's core APIs, we have built a broad ecosystem of tools [53, 33, 28] demonstrating how Apache Spark can accelerate genomic data analysis. To enable the reproducible use of Apache Spark in scientific data analysis workloads, we have contributed to novel, cloud-native workflow systems [55].

1.1 Economic Trends and Population Scale Sequencing

- Genomic data is often only meaningful when viewed in aggregate.
  - The association between genotype and phenotype is often weak, unless the variant under study is strongly pathogenic.
  - Many diseases are not driven by a single genotype, but rather by the combined effect of multiple genotypes.
  - For example, in AML there is a spread spectrum of mutations which fall into several clearly distinct disease subtypes. What is the impact of a single mutation?
- Technical innovation has made large scale sequencing tractible.
  - NHGRI "Moore's Law" plot.
  - While the sequencing technology used in the Human Genome Project was expensive, current whole genome sequencing technology is inexpensive.
  - Additionally, exome and targeted capture techniques can further reduce costs.

- Population scale sequencing projects are no longer unprecidented.
  - The 1,000 Genomes project is now small data; there are a multitude of projects that have sequenced at the 10,000+ sample scale.
  - These include the UK10K, Exome Aggregation Consortium, and Genomics England, among others.
- Finally, large scale sequencing projects are moving out of research and into practice.
  - Large scale sequencing projects can inform both drug design and risk models.
  - GSK/NHS public/private model for Genomics England
  - Modeling risk via sequencing for cancer at Color Genomics
  - Regeneron/Geisenger tie-up

1.2 The Case for Distributed Computing for Genomic Analysis

- Most genomic analysis tasks map naturally to distributed computing.
  - Heavyweight analyses either typically work on unaligned data, or on a sorted stream across aligned data.
  - These patterns typically can be parallelized without significant communication.
  - Additionally, there are many queries that map directly onto relational primitives.
- We believe we need to clean slate re-architect genomics for distributed computing.
  - Genomics tools are typically designed assuming a flattened stack running on a single node, or on a HPC-style cluster.
  - There have been several attempts to retrofit tools onto distributed computing, e.g., CloudBurst/Crossbow using Hadoop Streaming.
  - There have been several attempts to retrofit genomics specific file formats onto distributed query architectures, e.g., SegPig, BioPig.
  - However, these implementations provide either poor programming costs or inefficient and limited query mechanisms.
  - By doing a clean-slate rearchitecture, we can eliminate architectural problems and provide better user-facing query models with better performance.

1.3 Mapping Genomics onto Distributed Computing using ADAM

- To this end, we propose ADAM.
  - Define schemas for genomic datatypes, which provide data independence.
  - These schemas form the basis of a narrow waisted stack, which yields APIs that support both genomic query and metadata management, and which can be used across multiple languages.
  - We implement this architecture on top of Apache Spark, one of the most widely used distributed computing frameworks.
- To demonstrate ADAM, we have built an end-to-end variant calling pipeline.
  - This pipeline includes distributed implementations of alignment, read preprocessing, and variant calling.
  - The pipeline can run end-to-end on a 60× coverage whole genome in under an hour, at a cost of <$15 on cloud computing.
  - This pipeline provides results comparable to state of the art for SNV calling, and high accuracy (97%) for INDEL calling.
- ADAM improves over conventional genomics tools by providing:
  - Schemas which can support loading data from a large variety of formats.
  - High level, quasi-relational APIs for manipulating genomic data in both single node and cluster environments.
  - Parallel I/O across genomics file formats.
  - A simple API for parallelizing single node genomic tools with a minimal amount of code.
- ADAM has enabled a large ecosystem of work

Chapter 2

Background and Related Work

- This dissertation focuses on the “genome resequencing” pipeline.
  - I.e., given a known genome assembly, identify the edits between this individual and the assembly.
  - We assume short reads.
  -

2.1 Genome Sequencing Technologies

- How are reads sequenced?
  - Illumina uses a sequencing-by-synthesis approach.
  - Dyes are attached to nucleotides.
  - The dyes are imaged, washed off, and new dyes are attached.
  - Image to go here.
- Where does the DNA come from?
  - Sample prep and extraction...
  - Details to be added, depending on amount of detail suggested.
- Data characteristics:
  - Relative error rates, bias patterns, etc...
  - Differences between whole genome, whole exome.

2.2 Genomic Analysis Tools and Architectures

Genomic Data Representations

- Widely used formats were developed mostly during the 1,000 Genomes project.
  - The Sequence Alignment/Mapping (SAM) format was developed as a way to represent genomic reads.
  - The Variant Call Format (VCF) format was defined to store variants and genotypes.
  - Both are tab delimited text file formats that store semistructured data.
- These formats begat later binary versions that provide improved compression and performance.
  - SAM/VCF were supplanted by binary variants (BAM/BCF)
  - Additionally, there was significant interest in compressed storage formats for genomic data (most significantly, CRAM)
- Schemas for representing genomic data:
  - GA4GH APIs
  - OpenCB APIs

Genomic Analysis Architectures

- The main genomic analysis architecture out there is the GATK.
  - Uses an iterator-based model called a “walker” to traverse over data aligned to reference genome coordinates.
  - Puports a map-reduce style API, but historically only provided single node execution (multithreaded).
  - Multi-node execution was provided through the Queue workflow manager.
  - Revisit this in the context of the GATK4.
- Several alternative approaches have included the Google Genomics and OpenCB approaches.
  - Google Genomics is built heavily on top of BigQuery.
- Workflow management as an alternate paradigm?
  - How much can genomics be “parallel-by-sample”?
  - See GATK Queue.
  - Toil, Cromwell, CWL, WDL, NextFlow...

Variant Calling Approaches

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 [1]. Now, several prominent variant callers such as the Genome Analysis Toolkit's (GATK) HaplotypeCaller [11], Scalpel [35], and Platypus [42]. Although haplotype-based methods have enabled more accurate INDEL and single nucleotide polymorphism (SNP) calls [7], this accuracy comes at the cost of end-to-end runtime [51]. Several recent projects have been focused on improving reassembly cost either by limiting the percentage of the genome that is reassembled [8] or by improving the performance of the core algorithms used in local reassembly [42].

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 [12]),
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 improving the algorithmic efficiency steps one through three of the local reassembly problem. We do not focus algorithmically on accelerating stages four and five, as there is wide variation in the algorithms used in stages four and five. However, we do provide an parallel implementation of a widely used statistical model for genotyping [24]. 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

[1] The number of haplotypes tested may be lower than the number of haplotypes reassembled. Several tools (see Depristo et al [11] and Garrison and Marth [15]) 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.

complexity comes from realigning r reads to h haplotypes, where realignment has complexity $\mathcal{O}(l_r l_h)$.

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. To further improve the efficiency of this approach, we demonstrate in §7.1 how we can implement the canonicalization scheme that we demonstrate using indexed de Bruijn graphs without constructing a de Bruijn graph that contains both sequences.

Current variant calling pipelines depend heavily on realignment based approaches for accurate genotyping [25]. 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 [15], 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 [11], while methods like FreeBayes [15] and Platypus [42] 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 [25]. 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. [18], 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 (2.1), 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.

$$h \sim f_v l_{\text{ref}} + \epsilon l_{\text{ref}} c \tag{2.1}$$

This model is naïve, as the coverage depth and rate of variation varies across sequenced datasets, especially for targeted sequencing runs [13]. 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 [11]. 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 [15], restricting realignment to INDELs (IndelRealigner, [11]), 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 (2.1) solely as a motivating approximation, and hope to study these characteristics in more detail in future work.

2.3 Distributed Computing Platforms

Distributed Genomic Analysis Tools

- Tools retrofitted on top of distributed computing:
    - CloudBurst
    - CrossBow
    - CloudScale-BWAMem
    - Halvalde
- Genomics tools designed for distributed computing:

  - SparkSeq
  - VariantSpark
  - Query models for Genomics on distributed computing: SeqPig, BioPig
  - OpenCB
  - GATK4
  - Hail

- Genome assembly and HPC architectures:
  - Won't discuss much in this paper, but genome assembly has different access patterns, more amenable to HPC
  - AbYSS on MPI
  - PGAS approaches to de Bruijn graph traversal

Chapter 3

Design Principles for Scalable Genomics

- Initially, ADAM had several goals:
    - Provide clean APIs for writing large scale genomic data analyses
    - Raise abstraction by centering data manipulation around schemas instead of file formats
    - Allow these APIs to be exposed across commonly used languages
- To do this, ADAM used a decoupled, stack-oriented architecture
- Over time, these goals grew in scope to include:
    - Support coordinate-space joins with genomic data
    - Support exploratory data analysis on genomic datasets
    - Allow people to reuse their existing genomic analysis tools on Spark with minimal modifications
- Supporting these goals was enabled by ADAM's decoupled architecture

3.1 A Stack Architecture for Scientific Data Processing

The processing patterns being applied to scientific data shift widely as the data itself ages. Because of this change, we want to design a scientific data processing system that is flexible enough to accommodate our different use cases. At the same time, we want to ensure that the components in the system are well isolated so that we avoid bleeding functionality across the stack. If we bleed functionality across layers in the stack, we make it more difficult to adapt our stack to different applications. Additionally, as we discuss in §**??**, improper separation of concerns can actually lead to errors in our application.

These concerns are very similar to the factors that led to the development of the Open Systems Interconnection (OSI) model and Internet Protocol (IP) stack for networking services [59]. The networking stack models were designed to allow the mixing and matching of different protocols, all of which existed at different functional levels. The success of the networking stack model can largely be attributed to the "narrow waist" of the stack, which simplified the integration of a new protocol or technology by ensuring that the protocol only needed to implement a single interface to be compatible with the rest of the stack.

Unlike conventional scientific systems that leverage custom data formats like BAM or SAM [27], or CRAM [14], we believe that the use of an explicit schema for data interchange is critical. In our stack model shown in Figure **??**, the schema becomes the "narrow waist" of the stack. Most importantly, placing the schema as the narrow waist enforces a strict separation between data storage/access and data processing. Additionally, this enables literate programming techniques which can clarify the data model and access patterns. The seven layers of our stack model are decomposed as follows, and are numbered in ascending order from bottom to top:

1. **Physical Storage:** This layer coordinates data writes to physical media.
2. **Data Distribution:** This layer manages access, replication, and distribution of the files that have been written to storage media.
3. **Materialized Data:** This layer encodes the patterns for how data is encoded and stored. This layer determines I/O bandwidth and compression.
4. **Data Schema:** This layer specifies the representation of data, and forms the narrow waist of the stack that separates access from execution.
5. **Evidence Access:** This layer provides us with primitives for processing data, and allows us to transform data into different views and traversals.
6. **Presentation:** This layer enhances the data schema with convenience methods for performing common tasks and accessing common derived fields from a single element.
7. **Application:** At this level, we can use our evidence access and presentation layers to compose the algorithms to perform our desired analysis.

A well defined software stack has several other significant advantages. By limiting application interactions with layers lower than the presentation layer, application developers are given a clear and consistent view of the data they are processing, and this view of the data is independent of whether the data is local or distributed across a cluster or cloud. By separating the API from the data access layer, we improve flexibility. With careful design in the data format and data access layers, we can seamlessly support conventional whole file access patterns, while also allowing easy access to small slices of files. By treating the compute substrate and storage as separate layers, we also drastically increase the portability of the APIs that we implement.

As we discuss in more detail in §??, current scientific systems bleed functionality between stack layers. An exemplar is the SAM/BAM and CRAM formats, which expect data to be sorted by genomic coordinate. This order modifies the layout of data on disk (level 3, Materialized Data) and constrains how applications traverse datasets (level 5, Evidence Access). Beyond constraining applications, this leads to bugs in applications that are difficult to detect.[1] These views of evidence should be implemented at the evidence access layer instead of in the layout of data on disk. This split enforces independence of anything below the schema.

The idea of decomposing scientific applications into a stack model is not new; Bafna et al [6] made a similar suggestion in 2013. We borrow some vocabulary from Bafna et al, but our approach is differentiated in several critical ways:

- Bafna et al consider the stack model specifically in the context of data management systems for genomics; as a result, they bake current technologies and design patterns into the stack. In our opinion, a stack design should serve to abstract layers from methodologies/implementations. If not, future technology trends may obsolete a layer of the stack and render the stack irrelevant.

- Bafna et al define a binary data format as the narrow waist in their stack, instead of a schema. While these two seem interchangeable, they are not in practice. A schema is a higher level of abstraction that encourages the use of literate programming techniques and allows for data serialization techniques to be changed as long as the same schema is still provided.

- Notably, Bafna et al use this stack model to motivate GQL [19]. While a query system should provide a way to process and transform data, Bafna et al instead move this system down to the data materialization layer. We feel that this inverts the semantics that a user of the system would prefer and makes the system less general.

Deep stacks like the OSI stack [59] are generally simplified for practical use. Conceptually, the stack we propose is no exception. In practice, we combine layers one and two, and layers five and six. There are several reasons for these mergers. First, in `Hadoop`-based systems, the system does not have practical visibility below layer two, thus there is no reason to split layers one and two except as a philosophical exercise. Layers five and six are commingled because some of the enriched presentation objects are used to implement functionality in the evidence access layer. This normally happens when a key is needed, such as when repartitioning the dataset, or when reducing or grouping values.

[1] The current best-practice implementations of the BQSR and Duplicate Marking algorithms both fail when processing certain corner-case alignments. These errors are caused because of the requirement to traverse reads in sorted order.

Part II

Architecture and Infrastructure

Chapter 4

The ADAM Architecture

Due to both the growing volume of genomic sequencing data and the large size of these datasets, sequencing centers face the increasingly difficult task of turning around genomic data analyses in a reasonable timeframe [44, 49]. While the per-run latency of current genomic pipelines such as the `GATK` can be improved by manually partitioning the input dataset and distributing work, native support for distributed computing is not provided. As a stopgap solution, projects like `Cloudburst` [45] and `Crossbow` [22] have ported individual analytics tools to run on top of `Hadoop`. While this approach has served well for proofs of concept, it provides poor abstractions for application developers and makes it difficult to create novel distributed genomic analyses, and does little to attack sources of inefficiency or incorrectness in distributed genomics pipelines.

To address these problems, we need to reconsider how to build software for processing genomic data. Modern genome analysis pipelines are built around monolithic architectures and flat file formats. These architectures are designed to efficiently run current algorithms on single node processing systems, but impose significant restrictions. These restrictions include:

- These implementations are locked to a single node processing model. Even the `GATK`'s "map-reduce" styled `walker` API [31] is limited to natively support processing on a single node. While these jobs can be manually partitioned and run in a distributed setting, manual partitioning can lead to imbalance in work distribution and makes it difficult to run algorithms that require aggregating data across all partitions, and lacks the fault tolerance provided by modern distributed systems such as `Hadoop` or `Spark` [57].

- Most of these implementations *assume* invariants about the sorted order of records on disk. This "stack smashing" (specifically, the layout of data is used to "accelerate" a processing stage) can lead to bugs when data does not cleanly map to the assumed sort order. Additionally, since these sort order invariants are rarely explicit and vary from tool to tool, pipelines assembled from disparate tools can be brittle. We discuss this more in §**??** and footnote 1.

- Additionally, while these invariants are intended to improve performance, it is not clear that these invariants *actually* improve performance. There are two common sort invariants used in genomics: sort by reference position and sort by read name. Changing between these two sort orders entails a full shuffle and resort of the dataset. Additionally, a sort is required after alignment to establish a sort order.

As noted above, current implementations are locked to a single node model. Projects like `Hadoop-BAM` [37], `SeqPig` [46], and `BioPig` [38] have attempted to build a distributed processing environment on top of current single node genomics APIs. However, several problems must be solved in order to make distributed processing of genomic data productive:

- Current genomics data formats rely on a centralized header for storing experiment metadata. Since the metadata is centralized, it must be replicated to all machines.
- A simple map-reduce or SQL-like API is insufficient for implementing genomic analyses. Rather, to enhance bioinformatician productivity, we need to define APIs that allow developers to conveniently express algorithms.

In `ADAM`, we have taken a more aggressive approach to the design of APIs for processing genomic data in a distributed system. Although modern genomics pipelines are built as monolithic applications, we have chosen a layered decomposition for `ADAM`, that uses a schema as a "narrow waist". Instead of using a flat file format, as is traditional in genomics, we are using this schema with `Parquet` (a commodity columnar store [4]) to store genomic data in a way that both allows efficient distributed read/write and that achieves high compression. On top of this, we have added primitives that implement common genomic traversals in a distributed manner. We have then used `ADAM` to implement common preprocessing stages from commonly used genomics pipelines. `ADAM`'s preprocessing stages are between 1-5× faster than the equivalent `GATK` preprocessing stages, and achieve linear scaling out to 128 nodes.

4.1 Realizing A Decoupled Stack Architecture In ADAM

- Implementing a stack architecture requires careful API design
  - If APIs are too strictly specified, stack layers can't actually be exchanged
  - If APIs are too vaguely specified, system is unreliable
- ADAM ultimately extends two important APIs:
  - The ADAMContext: the entrypoint to loading all data
  - The GenomicRDD abstraction: a wrapper for genomic datasets

- How are these APIs important in realizing the stack vision?
  - Layers 1 and 2 are largely deferred to Apache Spark
  - Layer 3 is essentially a view; the ADAMContext provides the view on load, GenomicRDD provides the view on save
  - Layer 4 is enclosed in the GenomicRDD
  - Layers 5 and 6 are implemented in the GenomicRDD; users can specialize a GenomicRDD for their specific application/query pattern
  - Layer 7 is where the user builds their code, and this is built by using the ADAMContext to load in a GenomicRDD, which is then transformed and saved.

4.2 Query Patterns for Genomic Data Analysis

There are a wide array of experimental techniques and platforms in genome informatics, but many of these methods produce datapoints that are tied to locations in the genome through the use of genomic coordinates. Each cell contains a copy of the genome with one molecule per chromosome. Each molecule is a collection of DNA polymers coated with (and wrapped around) proteins and packed into the nucleus in a complex 3-dimensional shape. In practice, computational biologists abstract this complexity by storing a single long string that represents the nucleotides of the chromosome. We can then connect a datapoint or observation to the genome by associating the data with the chromosome name and a point or interval on a 1-dimensional space.

A platform for scientific data processing in genomics needs to understand these 1-dimensional coordinate systems because these become the basis on which data processing is parallelized. For example, when calling variants from sequencing data, the sequence data that is localized to a single genomic region (or "locus") can be processed independently from the data localized to a different region, as long as the regions are far enough apart.

Beyond parallelization, many of the core algorithms and methods for data aggregation in genomics are phrased in terms of geometric primitives on 1-D intervals and points where we compute distance, overlap, and containment. An algorithm for calculating quality control metrics may try to calculate "coverage," a count of how many reads overlap each base in the genome. A method for filtering and annotating potential variants might assess the validity of a variant using the quality characteristics of all reads that overlap the putative variant.

To support these algorithms, we provide a "region" or "spatial" join primitive. The algorithm used is described in algorithm 1 and takes as input two sets (RDDs, see Zaharia et al [57]) of `ReferenceRegions`, a data structure that represents intervals along the 1-D genomics coordinate space. It produces the set of all overlapping `ReferenceRegion` pairs. The *hulls* variable contains the set of convex hulls and is broadcasted to all compute nodes during the join.

To find the maximal set of non-overlapping regions, we must find the convex hull of all regions emitted. We present a distributed algorithm for finding convex hulls in Appendix

Algorithm 1 Partition And Join Regions via Broadcast

```
  $left \leftarrow$ input dataset; left side of join
  $right \leftarrow$ input dataset; right side of join
  $regions \leftarrow left$.map($data \Rightarrow$generateRegion($data$))
  $regions \leftarrow regions$.groupBy($region \Rightarrow region.name$)
  $hulls \leftarrow regions$.findConvexHull()
  $hulls$.broadcast()
  $keyLeft \leftarrow left$.keyBy($data \Rightarrow$getHullId($data$, $hulls$))
  $keyRight \leftarrow right$.keyBy($data \Rightarrow$getHullId($data$, $hulls$))
  $joined \leftarrow keyLeft$.join($keyRight$)
  $truePositives \leftarrow joined$.filter($r1, r2 \Rightarrow r1$.overlaps($r2$))
  return $truePositives$
```

6.2. The distributed convex hull computation problem is important because it is used both for computing regions for partitioning during a region join and for performing INDEL realignment.

While the join described above is a broadcast join, a region join can also be implemented via a straightforward shuffle-based approach, which is described in Algorithm 2. The `partitionJoinFn` function maintains two iterators (one each from both the left and right collections), along with a buffer. This buffer is used to track all key-value pairs from the right collection iterator that *could* match to a future key-value pair from the left collection iterator. We prune this buffer every time that we advance the left collection iterator. For simplicity, the description of Algorithm 2 ignores the complexity of processing keys that cross partition boundaries. In our implementation, we replicate keys that cross partition boundaries into both partitions.

Algorithm 2 Partition And Join Regions via Shuffle

```
  $left \leftarrow$ input dataset; left side of join
  $right \leftarrow$ input dataset; right side of join
  $partitions \leftarrow left$.getPartitions()
  $left \leftarrow left$.repartitionAndSort($partitions$)
  $right \leftarrow right$.repartitionAndSort($partitions$)
  $joined \leftarrow left$.zipPartitions($right$, partitionJoinFn)
  return $joined$
```

These joins serve as a core that we can use to build other abstractions with. For example, self-region joins and multi-region joins are common in genomics, and can be easily implemented using the above implementations. We are currently working to implement further parallel spatial functions such as sliding windows, using techniques similar to the shuffle-based join. We are working to characterize the performance differences between the two join strategies described above. In the future, we hope to enable the use of the region join in a SQL based system such as Spark SQL [5].

4.3 Schema Design for Representing Genomic Data

A common criticism of bioinformatics as a field surrounds the proliferation of file formats. Short read data alone is stored in four common formats: FASTQ [10], SAM [27], BAM, and CRAM [14]. While these formats all represent different layouts of data on disk, they tend to be logically harmonious. Due to this logical congruency of the different formats, we chose to build ADAM on top of a logical schema, instead of a binary format on disk. While we do use Apache Parquet [4] to materialize data on disk, the Apache Avro [2] schema is used as a narrow waist in the system, that enables "legacy" formats to be processed identically to data stored in Parquet with modest performance degradation.

We made several high level choices when designing the schemas used in ADAM. First, the schemas are fully denormalized, which reduces the cost of metadata access and simplifies metadata distribution. We are able to get these benefits without greatly increasing the cost of memory access because our backing store (Parquet) makes use of run length and dictionary encoding, which allows for a single object to be allocated for highly repetitive elements on read. Another key design choice was to require that all fields in the schema are nullable; by enforcing this requirement, we enable arbitrary user specified projections. Arbitrary projections can be used to accelerate common sequence quality control algorithms such as Flagstat [30, 39].

We have reproduced the schemas used to describe reads, variants, and genotypes below. ADAM also contains schemas for describing assembled contigs, genomic features, and variant annotations, but we have not included them in this section.

Listing 4.1: ADAM read schema

```
record AlignmentRecord {
  /** Alignment position and quality */
  Contig contig;
  long start;
  long oldPosition;
  long end;

  /** read ID, sequence, and quality */
  string readName;
  string sequence;
  string qual;

  /** alignment details */
  string cigar;
  string oldCigar;
  int mapq;
  int basesTrimmedFromStart;
  int basesTrimmedFromEnd;
  boolean readNegativeStrand;
  boolean mateNegativeStrand;
  boolean primaryAlignment;
  boolean secondaryAlignment;
  boolean supplementaryAlignment;
  string mismatchingPositions;
  string origQual;

  /** Read status flags */
  boolean readPaired;
  boolean properPair;
  boolean readMapped;
  boolean mateMapped;
  boolean firstOfPair;
  boolean secondOfPair;
  boolean failedVendorQualityChecks;
  boolean duplicateRead;

  /** optional attributes */
  string attributes;

  /** record group metadata */
  string recordGroupName;
  string recordGroupSequencingCenter;
  string recordGroupDescription;
  long recordGroupRunDateEpoch;
  string recordGroupFlowOrder;
  string recordGroupKeySequence;
  string recordGroupLibrary;
  int recordGroupPredictedMedianInsertSize;
  string recordGroupPlatform;
  string recordGroupPlatformUnit;
  string recordGroupSample;

  /** Mate pair alignment information */
  long mateAlignmentStart;
  long mateAlignmentEnd;
  Contig mateContig;
}
```

Our read schema maps closely to the logical layout of data presented by `SAM` and `BAM`.

The main modifications relate to how we represent metadata, which has been denormalized across the record. All of the metadata from the sequencing run and prior processing steps are packed into the record group metadata fields. The program information describes the processing lineage of the sample and is expected to be uniform across all records, thus it compresses extremely well. The record group information is not guaranteed to be uniform across all records, but there are a limited number number of record groups per sequencing dataset. This metadata is string heavy, which benefits from column-oriented decompression and makes proper deserialization from disk important. Although the information consumes less than 5% of space on disk, a poor deserializer implementation may replicate a string per field per record, which greatly increases the amount of memory allocated and the garbage collection (GC) load.

Listing 4.2: ADAM variant and genotype schemas

```
enum StructuralVariantType {
  DELETION,
  INSERTION,
  INVERSION,
  MOBILE_INSERTION,
  MOBILE_DELETION,
  DUPLICATION,
  TANDEM_DUPLICATION
}

record StructuralVariant {
  StructuralVariantType type;
  string assembly;

  boolean precise;
  int startWindow;
  int endWindow;
}

record Variant {
  Contig contig;
  long start;
  long end;

  string referenceAllele;
  string alternateAllele;
  StructuralVariant svAllele;

  boolean isSomatic;
}
```

```
enum GenotypeAllele {
  Ref,
  Alt,
  OtherAlt,
  NoCall
}

record VariantCallingAnnotations {
  float variantCallErrorProbability;

  array<string> variantFilters;

  int readDepth;
  boolean downsampled;
  float baseQRankSum;
  float clippingRankSum;
  float fisherStrandBiasPValue = null;
  float haplotypeScore;
  float inbreedingCoefficient;
  float rmsMapQ;
  int mapq0Reads;
  float mqRankSum;
  float variantQualityByDepth;
  float readPositionRankSum;

  array<int> genotypePriors;
  array<int> genotypePosteriors;

  float vqslod;
  string culprit;
  boolean usedForNegativeTrainingSet;
  boolean usedForPositiveTrainingSet;

  map<string> attributes;
}

record Genotype {
  Variant variant;
  VariantCallingAnnotations variantCallingAnnotations;

  string sampleId;
  string sampleDescription;
  string processingDescription;
  array<GenotypeAllele> alleles;

  float expectedAlleleDosage;
  int referenceReadDepth;
  int alternateReadDepth;
  int readDepth;
  int minReadDepth;
  int genotypeQuality;
  array<int> genotypeLikelihoods;
  array<int> nonReferenceLikelihoods;
  array<int> strandBiasComponents;

  boolean splitFromMultiAllelic;

  boolean isPhased;
  int phaseSetId;
  int phaseQuality;
}
```

The variant and genotype schemas present a larger departure from the representation used by the Variant Call Format (VCF). The most noticeable difference is that we have migrated away from VCF's variant oriented representation to a matrix representation. Instead of the variant record serving to group together genotypes, the variant record is embedded within the genotype. Thus, a record represents the genotype assigned to a sample, as opposed to a VCF row, where all individuals are collected together. The second major modification is to assume a biallelic representation[1]. This differs from VCF, which allows multiallelic records. By limiting ourselves to a biallelic representation, we are able to clarify the meaning of many of the variant calling annotations. If a site contains a multiallelic variant (e.g., in VCF parlance this could be a 1/2 genotype), we split the variant into two or more biallelic records. The sufficient statistics for each allele should then be computed under a reference model similar to the model used in genome VCFs. If the sample does contain a multiallelic variant at the given site, this multiallelic variant is represented by referencing to another record via the OtherAlt enumeration.

These representations achieve high compression versus the legacy formats. We provide a detailed breakdown of compression in §??. ADAM data stored in Parquet achieves an approximately 25% reduction in file size over compressed BAM for read data, and a 66% reduction over GZIPped VCF for variant data.

[1] In a biallelic representation, we describe the genotype of a sample at a position or interval as the composition of a reference allele and a single alternate allele. If multiple alternate alleles segregate at the site (e.g., there are two known SNPs in a population at this site), we create multiple biallelic variants for the site.

Part III

Algorithms and Tools

Chapter 5

Automatic Parallelization of Legacy Tools with Cannoli

- Will never be able to eliminate single node tools
  - Most/all current widely used tools are designed for a single node
  - Certain tools (e.g., aligners) pay a steep performance penalty moving out of C/C++
  - Diversity of genomic workflows is too large (>100 sequencing assays)
- Several pipelines rely on manually chunking up tools to run in parallel:
  - GATK Queue
  - SpeedSeq/FreeBayes Parallel
- Several tools have custom Hadoop Streaming/Spark wrappers:
  - BWA through Seal, CS-BWAMEM, BWASpark
  - RNA-Rail
  - CrossBow, Cloudburst
- We should be able to automate this process:
  - Many genomics tools are built around a streaming paradigm
  - If we can provide automated chunking and process setup, then we are good to go
- Two part architecture:
  - `pipe` API in ADAM: auto-parallelization architecture
  - Tool wrappers in Cannoli

- Is it a truly general approach?
  - No.
  - Tools that need an all-reduce aren't a good match (e.g., Kallisto/Sailfish/Salmon, CNV callers).
  - However, this approach works well for the majority of tools.

5.1 Accomodating Single-node Tools in ADAM With the `pipe` API

- The PIPE API provides a simple API that autoparallelizes a command across a cluster that supports Apache Spark:
  - User specifies command, files to copy locally, environment settings
  - ADAM infers partitioning from attached sequences
  - Piped formats are specified at compile-time, optionally at runtime
- Implementation:
  - For data aligned to a reference genome, uses fixed size chunking, built on region join infrastructure (§4.2); user can specify flank size
  - Once data is partitioned, we open up a subprocess, and connect to stdin/out of this process.
  - Data is formatted using In/Out formatters:
    - Converts data from ADAM schemas into stream in legacy format, and vice versa
    - ADAM supports a broad range of codecs, including...

5.2 Packaging Parallelized Single-node Tools in Cannoli

- Approach:
  - Delegate to `pipe` API as much as possible
  - Transparently support both local and "remote" reference files
  - Support native executables, but default to Docker for convenience of packaging
- Tools supported:

- Aligners:
  - BWA
  - Bowtie2
  - SNAP
- Variant callers:
  - FreeBayes
  - SAMTools mpileup
- Annotation tools:
  - SNPEff
  - BEDTools

Chapter 6

Scalable Alignment Preprocessing with ADAM

In `ADAM`, we have implemented the three most-commonly used pre-processing stages from the `GATK` pipeline [11]. In this section, we describe the stages that we have implemented, and the techniques we have used to improve performance and accuracy when running on a distributed system. These pre-processing stages include:

1. **Duplicate Removal:** During the process of preparing DNA for sequencing, reads are duplicated by errors during the sample preparation and polymerase chain reaction stages. Detection of duplicate reads requires matching all reads by their position and orientation after read alignment. Reads with identical position and orientation are assumed to be duplicates. When a group of duplicate reads is found, each read is scored, and all but the highest quality read are marked as duplicates.

   We have validated our duplicate removal code against Picard [52], which is used by the GATK for Marking Duplicates. Our implementation is fully concordant with the Picard/GATK duplicate removal engine, except we are able to perform duplicate marking for chimeric read pairs.[1] Specifically, because Picard's traversal engine is restricted to processing linearly sorted alignments, Picard mishandles these alignments. Since our engine is not constrained by the underlying layout of data on disk, we are able to properly handle chimeric read pairs.

2. **Local Realignment:** In local realignment, we correct areas where variant alleles cause reads to be locally misaligned from the reference genome.[2] In this algorithm, we first identify regions as targets for realignment. In the `GATK`, this identification is done by traversing sorted read alignments. In our implementation, we fold over partitions where we generate targets, and then we merge the tree of targets. This process allows us to eliminate the data shuffle needed to achieve the sorted ordering. As part of this

[1] In a chimeric read pair, the two reads in the read pairs align to different chromosomes; see Li et al [26].

[2] This is typically caused by the presence of insertion/deletion (INDEL) variants; see DePristo et al [11].

fold, we must compute the convex hull of overlapping regions in parallel. We discuss this in more detail later in this section.

After we have generated the targets, we associate reads to the overlapping target, if one exists. After associating reads to realignment targets, we run a heuristic realignment algorithm that works by minimizing the quality-score weighted number of bases that mismatch against the reference.

3. **Base Quality Score Recalibration (BQSR):** During the sequencing process, systemic errors occur that lead to the incorrect assignment of base quality scores. In this step, we label each base that we have sequenced with an *error covariate*. For each covariate, we count the total number of bases that we saw, as well as the total number of bases within the covariate that do not match the reference genome. From this data, we apply a correction by estimating the error probability for each set of covariates under a beta-binomial model with uniform prior.

   We have validated the concordance of our BQSR implementation against the GATK. Across both tools, only 5000 of the ~180B bases ($< 0.0001\%$) in the high-coverage NA12878 genome dataset differ. After investigating this discrepancy, we have determined that this is due to an error in the GATK, where paired-end reads are mishandled if the two reads in the pair overlap.

In the rest of this section, we discuss the high level implementations of these algorithms.

6.1 BQSR Implementation

Base quality score recalibration seeks to identify and correct correlated errors in base quality score estimates. At a high level, this is done by associating sequenced bases with possible error covariates, and estimating the true error rate of this covariate. Once the true error rate of all covariates has been estimated, we then apply the corrected covariate.

Our system is generic and places no limitation on the number or type of covariates that can be applied. A covariate describes a parameter space where variation in the covariate parameter may be correlated with a sequencing error. We provide two common covariates that map to common sequencing errors [34]:

- *CycleCovariate*: This covariate expresses which cycle the base was sequenced in. Read errors are known to occur most frequently at the start or end of reads.
- *DinucCovariate*: This covariate covers biases due to the sequence context surrounding a site. The two-mer ending at the sequenced base is used as the covariate parameter value.

To generate the covariate observation table, we aggregate together the number of observed and error bases per covariate. Algorithms 3 and 4 demonstrate this process.

Algorithm 3 Emit Observed Covariates

```
$read \leftarrow$ the read to observe
$covariates \leftarrow$ covariates to use for recalibration
$sites \leftarrow$ sites of known variation
$observations \leftarrow \emptyset$
for $base \in read$ do
    $covariate \leftarrow \text{identifyCovariate}(base)$
    if $\text{isUnknownSNP}(base, sites)$ then
        $observation \leftarrow \text{Observation}(1, 1)$
    else
        $observation \leftarrow \text{Observation}(1, 0)$
    end if
    $observations.\text{append}((covariate, observation))$
end for
return $observations$
```

Algorithm 4 Create Covariate Table

```
$reads \leftarrow$ input dataset
$covariates \leftarrow$ covariates to use for recalibration
$sites \leftarrow$ known variant sites
$sites.\text{broadcast}()$
$observations \leftarrow reads.\text{map}(read \Rightarrow \text{emitObservations}(read, covariates, sites))$
$table \leftarrow observations.\text{aggregate}(\text{CovariateTable}(), \text{mergeCovariates})$
return $table$
```

In Algorithm 3, the `Observation` class stores the number of bases seen and the number of errors seen. For example, `Observation(1, 1)` creates an `Observation` object that has seen one base, which was an erroneous base.

Once we have computed the observations that correspond to each covariate, we estimate the observed base quality using equation (6.1). This represents a Bayesian model of the mismatch probability with Binomial likelihood and a Beta(1, 1) prior.

$$\mathbf{E}(P_{err}|cov) = \frac{\texttt{\#errors}(cov) + 1}{\texttt{\#observations}(cov) + 2} \tag{6.1}$$

After these probabilities are estimated, we go back across the input read dataset and reconstruct the quality scores of the read by using the covariate assigned to the read to look into the covariate table.

6.2 Indel Realignment Implementation

Although global alignment will frequently succeed at aligning reads to the proper region of the genome, the local alignment of the read may be incorrect. Specifically, the error models used by aligners may penalize local alignments containing INDELs more than a local alignment that converts the alignment to a series of mismatches. To correct for this, we perform local realignment of the reads against consensus sequences in a three step process. In the first step, we identify candidate sites that have evidence of an insertion or deletion. We then compute the convex hull of these candidate sites, to determine the windows we need to realign over. After these regions are identified, we generate candidate haplotype sequences, and realign reads to minimize the overall quantity of mismatches in the region.

Realignment Target Identification

To identify target regions for realignment, we simply map across all the reads. If a read contains INDEL evidence, we then emit a region corresponding to the region covered by that read.

Convex-Hull Finding

Once we have identified the target realignment regions, we must then find the maximal convex hulls across the set of regions. For a set R of regions, we define a maximal convex hull as the largest region $\hat{r}$ that satisfies the following properties:

$$\hat{r} = \cup_{r_i \in \hat{R}} r_i \tag{6.2}$$

$$\hat{r} \cap r_i \neq \emptyset, \forall r_i \in \hat{R} \tag{6.3}$$

$$\hat{R} \subset R \tag{6.4}$$

In our problem, we seek to find all of the maximal convex hulls, given a set of regions. For genomics, the convexity constraint described by equation (6.2) is trivial to check: specifically, the genome is assembled out of reference contigs[3] that define disparate 1-D coordinate spaces. If two regions exist on different contigs, they are known not to overlap. If two regions are on a single contig, we simply check to see if they overlap on that contig's 1-D coordinate plane.

Given this realization, we can define Algorithm 5, which is a data parallel algorithm for finding the maximal convex hulls that describe a genomic dataset.

The `generateTarget` function projects each datapoint into a Red-Black tree that contains a single region. The performance of the fold depends on the efficiency of the merge function. We achieve efficient merges with the tail-call recursive `mergeTargetSets` function that is described in Algorithm 6.

[3] *Contig* is short for *contiguous sequence*. In alignment based pipelines, reference contigs are used to describe the sequence of each chromosome.

Algorithm 5 Find Convex Hulls in Parallel

```
  $data \leftarrow$ input dataset
  $regions \leftarrow data$.map($data \Rightarrow$generateTarget($data$))
  $regions \leftarrow regions$.sort()
  $hulls \leftarrow regions$.fold($r_1, r_2 \Rightarrow$ mergeTargetSets($r_1, r_2$))
  **return** $hulls$
```

Algorithm 6 Merge Hull Sets

```
  $first \leftarrow$ first target set to merge
  $second \leftarrow$ second target set to merge
**Require:** $first$ and $second$ are sorted
  **if** $first = \emptyset \wedge second = \emptyset$ **then**
    **return** $\emptyset$
  **else if** $first = \emptyset$ **then**
    **return** $second$
  **else if** $second = \emptyset$ **then**
    **return** $first$
  **else**
    **if** last($first$) $\cap$ head($second$) $= \emptyset$ **then**
      **return** $first + second$
    **else**
      $mergeItem \leftarrow$ (last($first$) $\cup$ head($second$))
      $mergeSet \leftarrow$ allButLast($first$) $\cup mergeItem$
      $trimSecond \leftarrow$ allButFirst($second$)
      **return** mergeTargetSets($mergeSet$, $trimSecond$)
    **end if**
  **end if**
```

The set returned by this function is used as an index for mapping reads directly to realignment targets.

Candidate Generation and Realignment

Once we have generated the target set, we map across all the reads and check to see if the read overlaps a realignment target. We then group together all reads that map to a given realignment target; reads that don't map to a target are randomly assigned to a "null" target. We do not attempt realignment for reads mapped to null targets.

To process non-null targets, we must first generate candidate haplotypes to realign against. We support several processes for generating these consensus sequences:

- *Use known INDELs*: Here, we use known variants that were provided by the user to generate consensus sequences. These are typically derived from a source of common variants such as dbSNP [47].

- *Generate consensuses from reads*: In this process, we take all INDELs that are contained in the alignment of a read in this target region.
- *Generate consensuses using Smith-Waterman*: With this method, we take all reads that were aligned in the region and perform an exact Smith-Waterman alignment [48] against the reference in this site. We then take the INDELs that were observed in these realignments as possible consensuses.

From these consensuses, we generate new haplotypes by inserting the INDEL consensus into the reference sequence of the region. Per haplotype, we then take each read and compute the quality score weighted Hamming edit distance of the read placed at each site in the consensus sequence. We then take the minimum quality score weighted edit versus the consensus sequence and the reference genome. We aggregate these scores together for all reads against this consensus sequence. Given a consensus sequence c, a reference sequence R, and a set of reads $\mathbf{r}$, we calculate this score using equation (6.5).

$$q_{i,j} = \sum_{k=0}^{l_{r_i}} Q_k I[r_I(k) = c(j+k)] \forall r_i \in \mathbf{R}, j \in \{0, \ldots, l_c - l_{r_i}\} \tag{6.5}$$

$$q_{i,R} = \sum_{k=0}^{l_{r_i}} Q_k I[r_I(k) = c(j+k)] \forall r_i \in \mathbf{R}, j = \mathrm{pos}(r_i|R) \tag{6.6}$$

$$q_i = \min(q_{i,R}, \min_{j \in \{0, \ldots, l_c - l_{r_i}\}} q_{i,j}) \tag{6.7}$$

$$q_c = \sum_{r_i \in \mathbf{r}} q_i \tag{6.8}$$

In (6.5), $s(i)$ denotes the base at position i of sequence s, and l_s denotes the length of sequence s. We pick the consensus sequence that minimizes the q_c value. If the chosen consensus has a log-odds ratio (LOD) that is greater than 5.0 with respect to the reference, we realign the reads. This is done by recomputing the CIGAR and MDTag for each new alignment. Realigned reads have their mapping quality score increased by 10 in the Phred scale.

6.3 Duplicate Marking Implementation

Reads may be duplicated during sequencing, either due to clonal duplication via PCR before sequencing, or due to optical duplication while on the sequencer. To identify duplicated reads, we apply a heuristic algorithm that looks at read fragments that have a consistent mapping signature. First, we bucket together reads that are from the same sequenced fragment by grouping reads together on the basis of read name and record group. Per read bucket, we then identify the 5' mapping positions of the primarily aligned reads. We mark as duplicates all read pairs that have the same pair alignment locations, and all unpaired reads that map to the same sites. Only the highest scoring read/read pair is kept, where the score is the sum of all quality scores in the read that are greater than 15.

Chapter 7

Rapid Variant Calling with Avocado

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

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 indexed de Bruijn algorithm to extract INDEL variants. These INDEL variants are collected on a single node, and used as inputs to the next stage.
    b) Optionally, we run ADAM's [30, 39] INDEL realigner, using the discovered INDELs from stage one as "known INDELs" to realign to. This improves variant calling accuracy over solely using the indexed de Bruijn algorithm.
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 [58, 57], using the ADAM library [30, 39].

7.1 INDEL Reassembly

As opposed to traditional realignment based approaches, we canonicalize INDELs in the reads by looking for bubbles flanked by read vs. reference sequence matches. In a colored de Bruijn graph, a bubble refers to a location where the graph diverges between two samples. In §7.1, 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 §7.1, 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 §6.2.

Preliminaries

Our method relies on an *indexed de Bruijn* graph, which is a slight extension of the colored de Bruijn graph [16]. 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 $\mathcal{S}_1, \dots, \mathcal{S}_n$. 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 $\mathcal{S}_i, i \in \{1, \dots, n\}$ 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 $\mathcal{S}_1, \dots, \mathcal{S}_n$ 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 $\mathcal{S}_1, \dots, \mathcal{S}_n$ 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 $\mathcal{O}(nd)$,[1]

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., $s_1 \to s_n \to \cdots \to s_{n+m} \to s_2$ 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 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 $s_{i+n} \in \mathcal{S}'_i \forall n \in \{0, \ldots, m_i\}, s_{i+n-1} \to s_{i+n} \forall n \in \{1, \ldots, m_i\}$, $s_{j+n} \in \mathcal{S}'_j \forall n \in \{0, \ldots, m_j\}, s_{j+n-1} \to s_{j+n} \forall n \in \{1, \ldots, m_j\}$, and $s_1 \to s_i, s_j; s_{i+m_i}, s_{j+m_j} \to s_2$ 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` ↔ `A` edit at position three. With k-mer length $k = 3$, this corresponds to the graph in Figure 7.1.

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

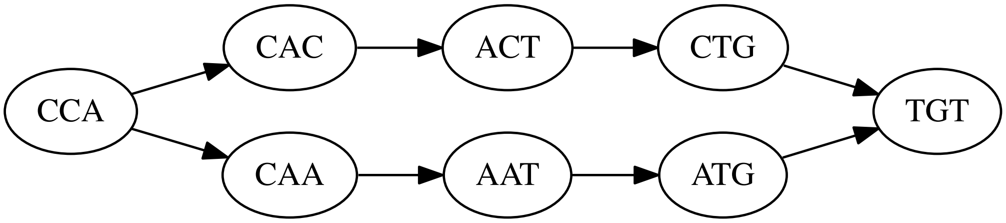

Figure 7.1: Subgraph Corresponding To a Single Nucleotide Edit

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[2] 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 $s_1 \to \cdots \to s_2 \in \mathcal{S}'_i$ 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 $s_1 \to \cdots \to s_2 \in \mathcal{S}'_j$ 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 7.2.

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 $s_1, \mathcal{L}_i, s_2$.[3] The canonical edits merely result from special cases:

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

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

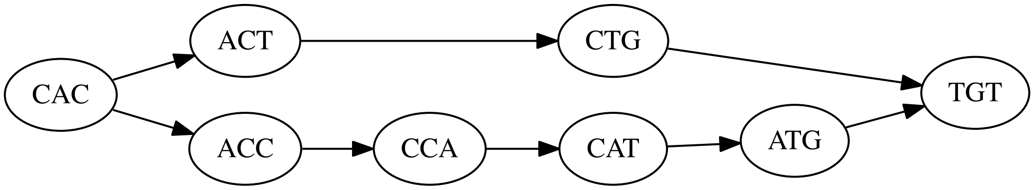

Figure 7.2: Subgraph Corresponding To a Canonical INDEL Edit

- 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 [12] or Smith and Waterman [48]), *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 [54]). By this, we can achieve $O(n(d+1))$ cost. Alternatively, we can decide to not further canonicalize the site, and to express it as a combined insertion and deletion. For simplicity and performance, we use this approach in Avocado.

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 [27]. 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 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 §7.1. 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 §7.1 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 §**??**, solely using our indexed de Bruijn 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 indexed de Bruijn algorithm to canonicalize INDEL variants, and then we used our variant discovery algorithm (see §7.2) with filtration disabled to collect all canonical INDELs. We then fed these INDELs and our input reads into ADAM's INDEL realignment engine [30, 39]. This tool is based on the algorithms used in the `GATK`'s INDEL realigner [11], 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 §6.2. As seen in §**??**, 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%.

7.2 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 [24], 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 [41]. 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 [9]. 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 §7.2).

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 [39]. 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 [20]) 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, which is used by Apache Spark). 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.

Genotyping Model

Once we have joined our reads against our variants, we score each read using the biallelic genotyping model proposed by Li [24]. 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 (7.1). Equation (7.1) is not our contribution and is reproduced from Li [24], but in log space.

$$\log \mathcal{L}(g) = -mk \sum_{i=0} j l_r(g, m-g, \epsilon_i) \sum_{i=j+1}^{k} l_r(m-g, g, \epsilon_i) \tag{7.1}$$

$$l_r(c_r, c_a, \epsilon) = \text{logsum}(\log c_r + \log \epsilon, \log c_a + \text{logm1}(\log \epsilon)) \tag{7.2}$$

In Equation (7.1), 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 [12]. 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 §7.2). 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.

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.

Part IV

Evaluation

Chapter 8

Benchmarking the ADAM Stack

Chapter 9

The Simons Genome Diversity Dataset Recompute

Part V

Conclusion and Future Work

Chapter 10

Future Work

- What were our overarching the goals?
  - Make genomic analysis easier to scale
  - Improve variant calling accuracy
  - Make genomic data analysis faster
- Several future directions:
  - Further improving query performance in ADAM
  - Extending variant calling algorithms in Avocado
  - Hardware acceleration for genomics

10.1 Further Query Optimization in ADAM

- Preserving sorted knowledge in ADAM
  - Needed to improve performance of region joins
  - Currently implemented, but refactoring to extend to legacy genomic datatypes
  - Store sorted state as metadata in the GenomicRDD.
- Extending what transformations can be optimized by partitioning:
  - Currently just accelerate joins
  - Aggregation queries naturally fall out of this as well
- Exposing other primitives as well:
  - Convex hull merge

10.2 Extensions to Avocado

- Deep learning for variant calling:
  - Current model for INDEL discovery is intolerant of read errors
  - Doesn't model error distribution over technologies
  - Google DeepVariant uses deep learning but in an unglamorous way
  - Would the Avocado INDEL canonicalization algorithm be a better fit?
- Somatic variant calling:
  - Avocado currently only models a sample with known ploidy
  - Can we accomodiate pooled or somatic variant calling?
  - WIP towards porting over MuTect

10.3 Hardware Acceleration for Genomic Data Processing

- With improved I/O, single thread performance remains an issue
  - As seen with Cannoli, hard to outperform C++ from Java
  - However, marginal benefit to jumping to C++ is unclear
  - Can we go directly to hardware?
- WIP on Genie library
- How can we make this even more accessible?
  - SQL as target language is a great opportunity
  - Integrate Genie with Apache Arrow
  - Allow generation of FPGA from Spark SQL plan
- How can we easily accomodate UDFs?

Chapter 11

Conclusion

Bibliography

[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(6):961–973, 2011.

[2] Apache. Avro. `http://avro.apache.org`.

[3] Apache. Hadoop. `http://hadoop.apache.org`.

[4] Apache. Parquet. `http://parquet.apache.org`.

[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, 2015.

[6] V. Bafna, A. Deutsch, A. Heiberg, C. Kozanitis, L. Ohno-Machado, and G. Varghese. Abstractions for genomics. *Communications of the ACM*, 56(1):83–93, 2013.

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

[8] A. Bloniarz, A. Talwalkar, J. Terhorst, M. I. Jordan, D. Patterson, B. Yu, and Y. S. Song. Changepoint analysis for efficient variant calling. In *Research in Computational Molecular Biology (RECOMB '14)*, pages 20–34. Springer, 2014.

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

[10] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic acids research*, 38(6):1767–1771, 2010.

[11] 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.

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

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

[14] M. H.-Y. Fritz, R. Leinonen, G. Cochrane, and E. Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Research*, 21(5):734–740, 2011.

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

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

[17] M. Jain, S. Koren, J. Quick, A. C. Rand, T. A. Sasani, J. R. Tyson, A. D. Beggs, A. T. Dilthey, I. T. Fiddes, S. Malla, et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. *bioRxiv*, page 128835, 2017.

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

[19] C. Kozanitis, A. Heiberg, G. Varghese, and V. Bafna. Using Genome Query Language to uncover genetic variation. *Bioinformatics*, 30(1):1–8, 2014.

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

[21] C. Kozanitis, C. Saunders, S. Kruglyak, V. Bafna, and G. Varghese. Compressing genomic sequence fragments using SlimGene. *Journal of Computational Biology*, 18(3):401–413, 2011.

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

[23] M. Lek, K. J. Karczewski, E. V. Minikel, K. E. Samocha, E. Banks, T. Fennell, A. H. O'Donnell-Luria, J. S. Ware, A. J. Hill, B. B. Cummings, et al. Analysis of protein-coding genetic variation in 60,706 humans. *Nature*, 536(7616):285–291, 2016.

[24] 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.

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

[26] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[27] 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.

[28] M. D. Linderman, D. Chia, F. Wallace, and F. A. Nothaft. DECA: Scalable XHMM exome copy-number variant calling with ADAM and Apache Spark. *bioRxiv*, 2017.

[29] S. Mallick, H. Li, M. Lipson, I. Mathieson, M. Gymrek, F. Racimo, M. Zhao, N. Chennagiri, S. Nordenfelt, A. Tandon, et al. The Simons Genome Diversity Project: 300 genomes from 142 diverse populations. *Nature*, 538(7624):201–206, 2016.

[30] M. Massie, F. 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, 2013.

[31] 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.

[32] E. V. Minikel, S. M. Vallabh, M. Lek, K. Estrada, K. E. Samocha, J. F. Sathirapongsasuti, C. Y. McLean, J. Y. Tung, P. Linda, P. Gambetti, et al. Quantifying prion disease penetrance using large population control cohorts. *Science translational medicine*, 8(322):322ra9–322ra9, 2016.

[33] A. Morrow. Distributed visualization for genomic analysis. Master's thesis, EECS Department, University of California, Berkeley, May 2017.

[34] K. Nakamura, T. Oshima, T. Morimoto, S. Ikeda, H. Yoshikawa, Y. Shiwa, S. Ishikawa, M. C. Linak, A. Hirai, H. Takahashi, et al. Sequence-specific error profile of Illumina sequencers. *Nucleic acids research*, page gkr344, 2011.

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

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

[37] 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.

[38] H. Nordberg, K. Bhatia, K. Wang, and Z. Wang. BioPig: a Hadoop-based analytic toolkit for large-scale sequence data. *Bioinformatics*, 29(23):3014–3019, 2013.

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

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

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

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

[43] D. M. Ruderfer, T. Hamamsy, M. Lek, K. J. Karczewski, D. Kavanagh, K. E. Samocha, M. J. Daly, D. G. MacArthur, M. Fromer, S. M. Purcell, et al. Patterns of genic intolerance of rare copy number variation in 59,898 human exomes. *Nature genetics*, 48(10):ng–3638, 2016.

[44] E. E. Schadt, M. D. Linderman, J. Sorenson, L. Lee, and G. P. Nolan. Computational solutions to large-scale data management and analysis. *Nature Reviews Genetics*, 11(9):647–657, 2010.

[45] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[46] A. Schumacher, L. Pireddu, M. Niemenmaa, A. Kallio, E. Korpelainen, G. Zanetti, and K. Heljanko. SeqPig: simple and scalable scripting for large sequencing data sets in Hadoop. *Bioinformatics*, 30(1):119–120, 2014.

[47] 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.

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

[49] L. D. Stein et al. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[50] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: astronomical or genomical? *PLoS biology*, 13(7):e1002195, 2015.

[51] 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 human genome variant calling. *Bioinformatics*, page btu345, 2014.

[52] The Broad Institute of Harvard and MIT. Picard. `http://broadinstitute.github.io/picard/`, 2014.

[53] E. Tu. Interactive exploration on large genomic datasets. Master's thesis, EECS Department, University of California, Berkeley, May 2016.

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

[55] J. Vivian, A. Rao, F. A. Nothaft, C. Ketchum, J. Armstrong, A. Novak, J. Pfeil, J. Narkizian, A. D. Deran, A. Musselman-Brown, et al. Toil enables reproducible, open source, big biomedical data analyses. *Nature biotechnology*, 35(4):314, 2017.

[56] R. Walsh, K. L. Thomson, J. S. Ware, B. H. Funke, J. Woodley, K. J. McGuire, F. Mazzarotto, E. Blair, A. Seller, J. C. Taylor, et al. Reassessment of mendelian gene pathogenicity using 7,855 cardiomyopathy cases and 60,706 reference samples. *Genetics in Medicine*, 19(2):192–203, 2016.

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

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

[59] H. Zimmermann. OSI reference model–the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

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