Scalable Genome Resequencing with `ADAM` and `avocado`

by

Frank Austin Nothaft

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David Patterson, Chair
Professor Anthony Joseph
Assistant Professor Nir Yosef

Spring 2015

The thesis of Frank Austin Nothaft, titled Scalable Genome Resequencing with `ADAM` and `avocado`, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

Scalable Genome Resequencing with `ADAM` and `avocado`

Abstract

Scalable Genome Resequencing with `ADAM` and `avocado`

by

Frank Austin Nothaft

Master of Science in Computer Science

University of California, Berkeley

Professor David Patterson, Chair

The decreased cost of genome sequencing technologies has made genome sequencing a viable tool for clinical and populations genomics applications. The efficiency of genome sequencing has been further improved through large projects like the Human Genome Project, which have assembled reference genomes for medically/agriculturally important organisms. These reference quality assemblies have enabled the creation of *genome resequencing pipelines*, where the genome of a single sample is computed by computing the *difference* between a given sample and the reference genome for the organism.

While sequencing cost has decreased by more than 10,000× since the Human Genome Project concluded in 2003, resequencing pipelines have struggled to keep pace with the growing volume of genomic data. These tools suffer from limited parallelism because they were not designed to use parallel or distributed computing techniques, and are limited by asymptotically inefficient algorithms. In this thesis, we introduce two tools, `ADAM` and `avocado`. `ADAM` provides an efficient framework for performing distributed genomic analyses, and `avocado` implements efficient local reassembly to discover genomic variants.

Contents

Chapter 1

Genome Processing Pipelines

1.1 Introduction

Since the completion of the Human Genome Project in 2003, genome sequencing costs have dropped by more than $10,000\times$ [36]. The rapidly declining cost of sequencing a single human genome has enabled large sequencing projects like the 1000 Genomes Project [43] and the Cancer Genome Atlas (TCGA, [50]). As these large sequencing projects perform analysis that process terabytes (TB) to petabytes (PB) of genomic data, they have created a demand for genomic analysis tools that can efficiently process these scales of data [40, 46].

Over a similar time range, commercial needs led to the development of horizontally scalable analytics systems. The development and deployment of MapReduce at Google [9, 10] spawned the development of a variety of distributed analytics tools and the Hadoop ecosystem [3]. In turn, these systems led to other systems that provided a more fluent programming model [53] and higher performance [56]. The demand for these systems has been driven by the increase in the amount of data available to analysts, and has coincided with the development of statistical systems that are accessible to non-experts, such as `Scikit-learn` [38] and `MLI` [45].

With the rapid drop in the cost of sequencing a genome, and the accompanying growth in available data, there is a good opportunity to apply modern, horizontally scalable analytics systems to genomics. New projects such as the 100K for UK, which aims to sequence the genomes of 100,000 individuals in the United Kingdom [14] will generate three to four *orders of magnitude* more data than prior projects like the 1000 Genomes Project [43]. These projects use the current "best practice" genomic variant calling pipelines [5], which take approximately 120 hours to process a single, high-quality human genome using a single, beefy node [47]. To address these challenges, scientists have started to apply computer systems techniques such as MapReduce [26, 32, 41] and columnar storage [16] to custom scientific compute/storage systems. While these systems have improved analysis cost and performance, current implementations incur significant overheads imposed by the legacy formats and codebases that they use.

In this thesis, we demonstrate ADAM, a genomic data processing and storage system built using Apache Avro, Parquet, and Spark [2, 4, 56], and avocado, a variant caller built on top of ADAM. This pipeline achieves a 50× increase in throughput over the current best practice pipeline, while reducing analysis cost by 50%. In the process of creating ADAM, we developed a "narrow waisted" layering model for building scientific analysis systems. This narrow waisted stack is inspired by the OSI model for networked systems [57]. However, in our stack model, the data schema is the narrow waist that separates data processing from data storage. Our stack solves the following three problems that are common across current scientific analysis systems:

1. Current scientific systems improve the performance of common patterns by changing the data model (often by requiring data to be stored in a coordinate-sorted order).
2. Legacy data formats were not designed with horizontal scalability in mind.
3. The system must be able to efficiently access shared metadata, and to slice datasets for running targeted analyses.

We solve these problems with the following techniques:

1. We make a schema the "narrow waist" of our stack to enforce data independence and devise algorithms for making common genomics patterns fast.
2. To improve horizontal scalability, we use Parquet, a modern parallel columnar store based off of Dremel [33] to push computation to the data.
3. We use a denormalized schema to achieve O(1) parallel access to metadata.

We introduce the stack model in figure 1.1 as a way to decompose scientific systems.

While the abstraction inversion used in genomics to accelerate common access patterns is undesirable because it violates data independence, we also find that it sacrifices performance and accuracy. The current Sequence/Binary Alignment and Map (SAM/BAM [30]) formats for storing genomic alignments apply constraints about record ordering to enable specific computing patterns. Our implementation (described in §1.3) identifies errors in two current genomics processing stages that occur *because* of the sorted access invariant. Our implementations of these stages do not make use of sort order, and achieve higher performance *while* eliminating these errors.

Additionally, this thesis describes the variant discovery and genotyping algorithms implemented in avocado. avocado introduces a new algorithm for local reassembly that eliminates the expensive step of realigning reads to candidate haplotypes. Additionally, avocado introduces a novel statistical model for genotyping that eliminates errors caused by statistical models that optimistically assume the local independence of genomic loci.

All of the software (source code and executables) described in this thesis are available free of charge under the permissive Apache 2 open-source license. ADAM is available at

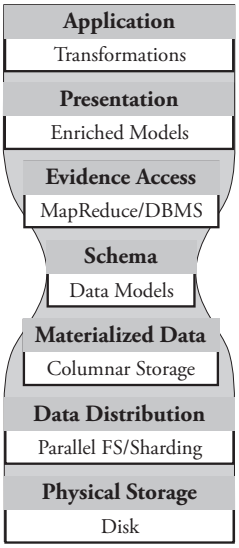

Figure 1.1: A Stack Model for Scientific Computing

`https://www.github.com/bigdatagenomics/adam`, and `avocado` is available at `https://www.github.com/bigdatagenomics/avocado`.

1.2 Background

This work is at the intersection of computational biology, data management, and processing systems. As such, our architectural approach is informed by recent trends in both areas. The design of large scale data management systems has changed dramatically since the papers by Dean and Ghemawat [9, 10] describing Google's `MapReduce` system. Over a similar timeframe, genomics has arisen due to improvements in data acquisition technologies. For example, since the Human Genome Project finished in 2001 [25], the price of genomic sequencing has dropped by 10,000× [36]. This drop in cost has enabled the capture of petabytes of sequence data, which has (in turn) enabled significant population genomics experiments like the 1000 Genomes project [43] and The Cancer Genome Atlas (TCGA, [50]).

Although there has been significant progress in the development of systems for processing large datasets—the development of first generation MapReduce systems [9], followed by iterative MapReduce systems like Spark [56], as well as parallel and columnar DBMS [1, 24]—the uptake of these systems in genomics has been slow. `MapReduce`'s impact has been limited to tools that use the map-reduce programming model as an inspiration for API design [32], or have been limited systems that have used `Hadoop` to naïvely parallelize existing toolkits [26, 41]. These approaches are perilous for several reasons:

- A strong criticism levied against the MapReduce model is that the API is insufficiently expressive for describing complex tasks. As a consequence of this, tools like the GATK [32] that adopt MapReduce as a programming model force significant restrictions on algorithm implementors. For example, a GATK `walker`[1] is provided with a single view over the data (a sorted iterator over a specified region), and is allowed limited reduce functionality.

- A major contribution of systems like MapReduce [10] and Spark [56, 55] is the ability to reliably distribute parallel tasks across a cluster in an automated fashion. While the GATK uses MapReduce as a programming abstraction (i.e., as an interface for writing `walker`s), it does not use MapReduce as an execution strategy. To run tools like the GATK across a cluster, organizations use workflow management systems for sharding and persisting intermediate data, and managing failures and retries. This approach is not only an inefficient duplication of work, but it is also a source of inefficiency during execution: the performance of iterative stages in the GATK is bottlenecked by I/O performance.

- The naïve Hadoop-based implementations in Crossbow [26] and Cloudburst [41] use scripts to run unmodified legacy tools on top of Hadoop. This approach does achieve speedups, but does not attack overhead. Several of the methods that they parallelize incur high overhead due to duplicated loading of indices (for fast aligners, loading of large indices can be a primary I/O bottleneck) and poor broadcasting of data.

Recent work by Diao et al [12] has looked at optimizations to MapReduce systems for processing genomic data. They adapt strategies from the query optimization literature to reorder computation to minimize data shuffling. While this approach does improve shuffle traffic, several preprocessing stages cannot be transposed. For instance, reversing the order of indel realignment and base quality score recalibration (see §1.3) will change the inferred quality score distribution. Additionally, we believe that the shuffle traffic that Diao et al observe is an artifact caused by an abstraction inversion present in many genomics tools. This abstraction inversion requires that all genomic data is processed in sorted order, which necessitates frequent shuffles. As we demonstrate in §1.3, these penalties can be eliminated by restructuring the pre-processing algorithms.

One notable area where modern data management techniques have been leveraged by scientists is in the data storage layer. Due to the storage costs of large genomic datasets, scientists have introduced the CRAM format that uses columnar storage techniques and special compression algorithms to achieve a 30% reduction in size over the original BAM format [16]. While CRAM achieves high ($\gg 50\%$) compression, it imposes restrictions on the ordering and structure of the data, and does not provide support for predicates or projection. We perform a more comprehensive comparison against CRAM in §4.2.

[1] The GATK provides `walker`s as an interface for traversing regions of the genome.

One interesting trend of note is the development of databases specifically for scientific applications. The exemplar is SciDB, which provides an array based storage model as well as efficient linear algebra routines [7]. While arrays accelerate many linear algebra based routines, they are not a universally great fit. For many genomics workloads, data is semistructured and may consist of strings, boolean fields, and an array of tagged annotations. Other systems like the Genome Query Language [22] have extended SQL to provide efficient query semantics across genomic coordinates. While GQL achieves performance improvements of up to 10× for certain algorithms, SQL is not an attractive language for many scientific domains, which make heavy use of user designed functions that may be cumbersome in SQL.

1.3 Pipeline Structure

This thesis targets the acceleration of *variant calling*, which is a statistical process to infer the sites at that a single individual varies from the reference genome.[2] Although there are a variety of sequencing technologies in use, the majority of sequence data used for variant calling and genotyping comes from the Illumina sequencing platform, which uses a "sequencing-by-synthesis" technique to generate short read data [34]. Short read refers to sequencing run will generate many reads that are between 50 and 250 bases in length. In addition to adjusting the length of the reads, we can control the amount of the data that is generated by changing the amount of the genome that we sequence, or the amount of redundant sequencing that we perform (the average number of reads that covers each base, or *coverage*). A single human genome sequenced at 60× coverage will produce approximately 1.4 billion reads, which is approximately 600 GB of raw data, or 225 GB of compressed data. For each read, we also are provided *quality scores*, which represent the likelihood that the base at a given position was observed. In a variant calling pipeline, we perform the following steps:

1. **Alignment:** For each read, we find the position in the genome that the read is most likely to have come from. As an exact search is too expensive, there has been an extensive amount of research that has focused on indexing strategies for improving alignment performance [29, 31, 54]. This process is parallel per sequenced read.

2. **Pre-processing:** After reads have been aligned to the genome, we perform several preprocessing steps to eliminate systemic errors in the reads. This may involve recalibrating the observed quality scores for the bases, or locally optimizing the read alignments. We will present a description of several of these algorithms in §1.3; for a more detailed discussion, we refer readers to DePristo et al [11].

3. **Variant calling:** Variant calling is a statistical process that uses the read alignments and the observed quality scores to compute whether a given sample matches or diverges

[2]The reference genome represents the "average" genome for a species. The Human Genome Project [25] assembled the first human reference genome.

from the reference genome. This process is typically parallel per position or region in the genome.

4. **Filtration:** After variants have been called, we want to filter out false positive variant calls. We may perform queries to look for variants with borderline likelihoods, or we may look for clusters of variants, which may indicate that a local error has occurred. This process may be parallel per position, may involve complex traversals of the genomic coordinate space, or may require us to fit a statistical model to all or part of the dataset.

This process is very expensive in time to run; the current best practice pipeline uses the BWA tool [29] for alignment and the GATK [11, 32] for pre-processing, variant calling, and filtration. Current benchmark suites have measured this pipeline as taking between 90 and 130 hours to run end-to-end [47]. Recent projects have achieved 5–10× improvements in alignment and variant calling performance [39, 54], which makes the pre-processing stages the performance bottleneck. Our experimental results have corroborated this, as the four pre-processing stages take over 110 hours to run on a clinical quality human genome when run on an Amazon EC2 `cr1.8xlarge` machine.

For current implementations of these read processing steps, performance is limited by disk bandwidth [12]. This bottleneck exists because the operations read in a SAM/BAM file, perform a small amount of processing, and write the data to disk as a new SAM/BAM file. We achieve a performance bump by performing our processing iteratively in memory. The four read processing stages can then be chained together, eliminating three long writes to disk and an additional three long reads from disk. Additionally, by rethinking the design of our algorithms, we are able to reduce overhead in several other ways:

1. Current algorithms require the reference genome to be present on all nodes. This assembly is then used to look up the reference sequence that overlaps all reads. The reference genome is several gigabytes in size, and performing a lookup in the reference genome can be costly due to its size. Instead, we leverage the `mismatchingPositions` field in our schema to embed information about the reference in each read. This optimization allows us to avoid broadcasting the reference, and provides O(1) lookup.

2. Shared-memory genomics applications tend to be impacted significantly by false sharing of data structures [54]. Instead of having data structures that are modified in parallel, we restructure our algorithms so that we only touch data structures from a single thread, and then merge structures in a reduce phase. The elimination of sharing improves the performance of covariate calculation during BQSR and the target generation phase of local realignment.

3. In a naïve implementation, the local realignment and duplicate marking tasks can suffer from stragglers. The stragglers occur due to a large amount of reads that either do not associate to a realignment target, or that are unaligned. We pay special attention

to these cases by manually randomizing the partitioning for these reads. This resolves load imbalance and mitigates stragglers.

4. For the Flagstat command, we are able to project a limited subset of fields. Flagstat touches fewer than 10 fields, which account for less than 10% of space on disk. We discuss the performance implications of this further in §4.2.

These techniques allow us to achieve a $> 50\times$ performance improvement over current tools, and scalability beyond 128 machines. We perform a detailed performance review in §4.1.

1.4 Layering

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 §1.3, 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 [57]. 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/SAM [30], or CRAM [16], we believe that the use of an explicit schema for data interchange is critical. In our stack model shown in Figure 1.1, 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 §1.3, 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 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.[3] These views of evidence should be implemented at the evidence access layer instead of in the layout of data on disk. This 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

[3]The current best-practice implementations of the BQSR and Duplicate Marking algorithms both fail in certain corner-case alignments. These errors are caused because of the limitation on traversing reads in sorted order.

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 [22]. 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.

By using Parquet as a storage format, we are able to process genomic data using many Hadoop-based systems. We implement high performance batch and interactive processing with Spark, and can delegate to systems like Impala and Spark-SQL for warehousing.

1.5 Related Work

Several variant analysis toolkits exist, with the most well known analysis toolkit being the `GATK` [11]. Additional toolkits include `HugeSeq` [23], `STORMSeq` [21], and `SpeedSeq` [8]. These tools combine alignment, variant calling, and filtration into an easy to use package, and may also orchestrate work distribution across a set of distributed machines. For example, the `GATK` and `HugeSeq` make use of an improvised map-reduce model, while `STORMSeq` uses a grid engine to distribute work according to a provided partitioning function. These tools delegate to either the `GATK`'s `HaplotypeCaller` or `UnifiedGenotyper`, or `FreeBayes` [17] for calling germline point events and INDELs. `Platypus` [39] is an additional notable toolkit that directly integrates alignment with variant calling to improve computational efficiency.

Although earlier methods such as the `mpileup` caller assumed the statistical independence of sites [27] post-alignment, current variant calling pipelines depend heavily on realignment based approaches for accurate genotyping [28]. These methods take two different approaches to generate candidate sequences for realignment:

1. *Realignment-only:* Putative INDELs are extracted directly from the aligned reads, and the reads are locally realigned,

2. *Reassembly:* The aligned reads are reassembled into haplotypes, which the reads are aligned against.

The realignment-only approach is used in `UnifiedGenotyper`[4] and `FreeBayes`, while `HaplotypeCaller`, `Platypus`, and `Scalpel` [35] make use of reassembly. In both cases, we perform the following algorithmic steps:

1. Candidate haplotypes are generated for realignment,

[4]When used following `IndelRealignment`

2. Each read is realigned to each haplotype, typically using a pair Hidden Markov Model (HMM, see [13]),

3. 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,

4. The alignments of the reads to the chosen haplotype pair are used to generate statistics that are then used for genotyping.

In haplotype reassembly, step 1 is broken down into two further steps:

1. A *de Bruijn* graph is constructed from the reads aligned to a region of the reference genome,

2. All valid paths between the start and end of the graph are enumerated.

In both the *realignment* and *reassembly* 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. These approaches are algorithmically different from *global alignment* because they make use of local context when picking the sequence to align to, and the alignment search space is much smaller, which enables the use of more expensive alignment methods.

De novo assembly provides another promising approach to variant discovery. In the *de novo* formulation of assembly, the reads are not aligned to a reference genome. Instead, "novel" contiguous fragments of sequence are assembled from the reads. Variants are called by aligning these assemblies to the reference genome, and by realigning the reads against the novel assemblies. Several implementations of *de novo* variant calling exist, most notably the `Cortex` [20] and `Discovar` [51] assemblers. Although *de novo* assembly solves several important issues seen by traditional variant callers (reference bias, structural variant detection), *de novo* assembly is currently too computationally expensive for widespread use in genotyping.

Chapter 2

Genomic Data Storage and Preprocessing Using ADAM

2.1 Distributed Architectures for Genomics

ADAM provides an API for storing and processing genomic data in a distributed system using Apache Spark [55], Avro [2], and Parquet [4]. At a core level, ADAM couples with Apache Spark to implement an API for expressing common genomic patterns on a distributed system. This provides several important advantages over current frameworks, such as the `walker` interface in the GATK:

- ADAM leverages the full Spark API, which is a more flexible and all-encompassing API than the map-reduce style API provided by the GATK,
- Current genomics APIs generally enforce limitations on the layout of data on disk. We provide a data independent API, which eliminates errors that occur when processing chimeric reads.
- By building on top of efficient distributed computing technologies, ADAM is able to scale out horizontally to more than 128 compute nodes.

2.2 Schema Design for Genomics

2.3 Read Preprocessing Algorithms

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 [48], 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 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 fold, we must compute the convex hull of overlapping regions in parallel. We discuss this in more detail in Appendix 2.3.

   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:

$$\mathbf{E}(P_{err}|cov) = \frac{\texttt{\#errors}(cov) + 1}{\texttt{\#observations}(cov) + 2} \tag{2.1}$$

   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

[1] In a chimeric read pair, the two reads in the read pairs align to different chromosomes; see Li et al [29].
[2] This is typically caused by the presence of insertion/deletion (INDEL) variants; see DePristo et al [11].

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.

BQSR Implementation

Base Quality Score Recalibration is an important early data-normalization step in the bioinformatics pipeline, and after alignment it is the next most costliest step. Since quality score recalibration can vastly improve the accuracy of variant calls — particularly for pileup-based callers like the UnifiedGenotyper or Samtools mpileup. Because of this, it is likely to remain a part of bioinformatics pipelines.

BQSR is also an interesting algorithm in that it doesn't neatly fit into the framework of map reduce (the design philosophy of the GATK). Instead it is an embarrassingly parallelizable aggregate. The ADAM implementation is:

```
def computeTable(rdd: Records, dbsnp: Mask) :
  RecalTable = {

  rdd.aggregate(new RecalTable)(
    (table, read) => { table + read },
    (table, table) => { table ++ table })
}
```

The ADAM implementation of BQSR utilizes the MD field to identify bases in the read that mismatch the reference. This enables base quality score recalibration to be entirely reference-free, avoiding the need to have a central Fasta store for the human reference. However, dbSNP is still needed to mask out positions that are polymorphic (otherwise errors due to real variation will severely bias the error rate estimates).

Indel Realignment Implementation

Indel realignment is implemented as a two step process. In the first step, we identify regions that have evidence of an insertion or deletion. After these regions are identified, we generate candidate haplotypes, and realign reads to minimize the overall quantity of mismatches in the region. The quality of mismatches near an indel serves as a good proxy for the local quality of an alignment. This is due to the nature of indel alignment errors: when an indel is misaligned, this causes a temporary shift in the read sequence against the reference sequence. This shift manifests as a run of several bases with mismatches due to their incorrect alignment.

Realignment Target Identification

Realignment target identification is done by converting our reads into reference oriented "rods"[3]. At each locus where there is evidence of an insertion or a deletion, we create a *target* marker. We also create a target if there is evidence of a mismatch. These targets contain the indel range or mismatch positions on the reference, and the range on the reference covered by reads that overlap these sites.

After an initial set of targets are placed, we merge targets together. This is necessary, as during the read realignment process, all reads can only be realigned once. This necessitates that all reads are members of either one or zero realignment targets. Practically, this means that over the set of all realignment targets, no two targets overlap.

The core of our target identification algorithm can be found below.

```
def findTargets (reads: RDD[ADAMRecord]):
    TreeSet[IndelRealignmentTarget] = {

  // convert reads to rods
  val processor = new Read2PileupProcessor
  val rods: RDD[Seq[ADAMPileup]] = reads.flatMap(
      processor.readToPileups(_))
    .groupBy(_.getPosition).map(_._2)

  // map and merge targets
  val targetSet = rods.map(
      IndelRealignmentTarget(_))
    .filter(!_.isEmpty)
    .keyBy(_.getReadRange.start)
    .sortByKey()
    .map(new TreeSet()(TargetOrdering) + _._2)
    .fold(new TreeSet()(TargetOrdering))(
      joinTargets)

  targetSet
}
```

To generate the initial unmerged set of targets, we rely on the ADAM toolkit's pileup generation utilities (see S**??**). We generate realignment targets for all pileups, even if they do not have indel or mismatch evidence. We eliminate pileups that do not contain indels or mismatches with a filtering stage that eliminates empty targets. To merge overlapping targets, we map all of the targets into a sorted set. This set is implemented using Red-Black trees. This allows for efficient merges, which are implemented with the tail-call recursive *joinTargets* function:

[3]Also known as pileups: a group of bases that are all aligned to a specific locus on the reference.

```
@tailrec def joinTargets (
  first: TreeSet[IndelRealignmentTarget],
  second: TreeSet[IndelRealignmentTarget]):
    TreeSet[IndelRealignmentTarget] = {

  if (!TargetOrdering.overlap(first.last,
      second.head)) {
    first.union(second)
  } else {
    joinTargets (first - first.last +
      first.last.merge(second.head),
      second - second.head)
  }
}
```

As we are performing a fold on an RDD which is sorted by the starting position of the target on the reference sequence, we know a priori that the elements in the "first" set will always be ordered earlier relative to the elements in the "second" set. However, there can still be overlap between the two sets, as this ordering does not account for the end positions of the targets. If there is overlap between the last target in the "first" set and the first target in the "second" set, we merge these two elements, and try to merge the two sets again.

Candidate Generation and Realignment

Candidate generation is a several step process:

1. Realignment targets must "collect" the reads that they contain.
2. For each realignment group, we must generate a new set of candidate haplotype alignments.
3. Then, these candidate alignments must be tested and compared to the current reference haplotype.
4. If a candidate haplotype is sufficiently better than the reference, reads are realigned.

The mapping of reads to realignment targets is done through a tail recursive function that performs a binary search across the sorted set of indel alignment targets:

```
@tailrec def mapToTarget (read: ADAMRecord,
  targets: TreeSet[IndelRealignmentTarget]):
    IndelRealignmentTarget = {

  if (targets.size == 1) {
    if (TargetOrdering.equals (targets.head, read)) {
```

```
      targets.head
    } else {
      IndelRealignmentTarget.emptyTarget
    }
  } else {
    val (head, tail) = targets.splitAt(
      targets.size / 2)
    val reducedSet = if (TargetOrdering.lt(
        tail.head, read)) {
      head
    } else {
      tail
    }
    mapToTarget (read, reducedSet)
  }
}
```

This function is applied as a groupBy against all reads. This means that the function is mapped to the RDD that contains all reads. A new RDD is generated where all reads that returned the same indel realignment target are grouped together into a list.

Once all reads are grouped, we identify new candidate alignments. However, before we do this, we left align all indels. For many reads that show evidence of a single indel, this can eliminate mismatches that occur after the indel. This involves shifting the indel location to the "left"[4] by the length of the indel. After this, if the read still shows mismatches, we generate a new consensus alignment. This is done with the *generateAlternateConsensus* function, which distills the indel evidence out from the read.

```
def generateAlternateConsensus (sequence: String,
  start: Long, cigar: Cigar): Option[Consensus] = {
  var readPos = 0
  var referencePos = start

  if (cigar.getCigarElements.filter(elem =>
        elem.getOperator == CigarOperator.I ||
        elem.getOperator == CigarOperator.D
      ).length == 1) {
    cigar.getCigarElements.foreach(cigarElement =>
      { cigarElement.getOperator match {
        case CigarOperator.I => return Some(
          new Consensus(sequence.substring(readPos,
            readPos + cigarElement.getLength),
            referencePos to referencePos))
        case CigarOperator.D => return Some(
```

[4] To a lower position against the reference sequence.

```
          new Consensus("",
          referencePos until (referencePos +
            cigarElement.getLength)))
        case _ => {
          if (cigarElement.getOperator
                .consumesReadBases &&
              cigarElement.getOperator
                .consumesReferenceBases
              ) {
            readPos += cigarElement.getLength
            referencePos += cigarElement.getLength
          } else {
            return None
          }
        }
      }
    })
    None
  } else {
    None
  }
}
```

From these consensuses, we generate new haplotypes by inserting the indel consensus into the reference sequence. The quality of each haplotype is measured by sliding each read across the new haplotype, using *mismatch quality*. Mismatch quality is defined for a given alignment by the sum of the quality scores of all bases that mismatch against the current alignment. While sliding each read across the new haplotype, we aggregate the mismatch quality scores. We take the minimum of all of these scores and the mismatch quality of the original alignment. This sweep is performed using the *sweepReadOverReferenceForQuality* function:

```
def sweepReadOverReferenceForQuality (
    read: String,reference: String,
    qualities: Seq[Int]): (Int, Int) = {
  var qualityScores = List[(Int, Int)]()

  for (i <- 0 until (reference.length -
      read.length)) {
    qualityScores = (
      sumMismatchQualityIgnoreCigar(
        read,
        reference.substring(i, i + read.length),
          qualities),
```

```
      i) :: qualityScores
  }

  qualityScores.reduce ((p1: (Int, Int),
     p2: (Int, Int)) => {
    if (p1._1 < p2._1) {
      p1
    } else {
      p2
    }
  })
}
```

If the consensus with the lowest mismatch quality score 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.

Convex-Hull Finding

A frequent pattern in our application is identifying the maximal convex hulls across sets 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:

$$\hat{r} = \cup_{r_i \in \hat{R}} r_i \tag{2.2}$$

$$\hat{r} \cap r_i \neq \emptyset, \forall r_i \in \hat{R} \tag{2.3}$$

$$\hat{R} \subset R \tag{2.4}$$

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 (2.2) is trivial to check: specifically, the genome is assembled out of reference contigs[5] 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 the data-parallel algorithm 1 to find the maximal convex hulls that describe a genomic dataset.

The `generateTarget` function projects each datapoint into a Red-Black tree which 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 which is described in algorithm 2.

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

Algorithm 1 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 2 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

For a region join (see §**??**), we would use the maximal convex hull set to define partitioning for the join. Alternatively, for INDEL realignment (see §1.3), we use this set as an index for mapping reads directly to targets.

Duplicate Marking Implementation

The following ADAM code, reformatted for this report, expresses the algorithm succinctly in 42 lines of Scala code.

```
for (((leftPos, library), readsByLeftPos) <-
    rdd.adamSingleReadBuckets()
       .keyBy(ReferencePositionPair(_))
       .groupBy(leftPositionAndLibrary);
```

```
buckets <- {

leftPos match {
 // These are all unmapped reads.
 // There is no way to determine if
 // they are duplicates
 case None =>
    markReads(readsByLeftPos.unzip._2,
      areDups = false)
 // These reads have their left position mapped
 case Some(leftPosWithOrientation) =>
   // Group the reads by their right position
   val readsByRightPos = readsByLeftPos.groupBy(
     rightPosition)
   // Find any reads with no right position
   val fragments = readsByRightPos.get(None)
   // Check if we have any pairs
   // (reads with a right position)
   val hasPairs = readsByRightPos.keys
     .exists(_.isDefined)
   if (hasPairs) {
     // Since we have pairs,
     // mark all fragments as duplicates
     val processedFrags = if (fragments.isDefined
     ) {
       markReads(fragments.get.unzip._2,
         areDups = true)
     } else {
       Seq.empty
     }
     val processedPairs =
         for (buckets <- (readsByRightPos - None)
           .values;
             processedPair <-
                scoreAndMarkReads(buckets.unzip._2))
        yield processedPair
     processedPairs ++ processedFrags
   } else if (fragments.isDefined) {
     // No pairs. Score the fragments.
     scoreAndMarkReads(fragments.get.unzip._2)
   } else {
     Seq.empty
   }
};
```

```
read <- buckets.allReads) yield read
```

For lines 1-4, all reads with the same record group name and read name are collected into buckets. These buckets contain the read and optional mate or secondary alignments. Each read bucket is then keyed by 5 ′ position and orientation and grouped together by left (lowest coordinate) position, orientation and library name.

For lines 8-41, we processed each read bucket with a common left 5′ position. Unmapped reads are never marked duplicate as their position is not known. Mapped reads with a common left position are separated into paired reads and fragments. Fragments, in this context, are reads that have no mate or should have a mate but it doesn't exist.

If there are pairs in a group, all fragments are marked duplicates and the paired reads are grouped by their right 5′ position. All paired reads that have a common right and left 5′ position are scored and all but the highest scoring read is marked a duplicate.

If there are no pairs in a group, all fragments are scored and all but the highest scoring fragment are marked duplicates.

Chapter 3

Variant Calling via Reassembly Using `avocado`

3.1 Modular Approaches for Variant Calling

3.2 Reference Threaded *de Bruijn* Graphs

The local assembler in `avocado` is a de bruijn graph based assembler. The assembler creates a "reference threaded" assembly. In this approach, we label k-mers that appear in the reference genome with their reference position. The advantage of this approach is that we can determine whether a k-mer in the graph can be mapped to the reference genome or if it represents a divergence from the reference (an alternate allele). Additionally, for bubbles off of reference arcs, we can identify the exact allele represented by the bubble without needing to enumerate haplotypes which we then align against the reference sequence. This approach has the following benefits:

- For a region that contains n variants, we only need to evaluate n variant arcs, as opposed to n^2 variant haplotypes.
- We can emit statistics for computing genotype likelihoods directly from the de bruijn graph. This is more efficient than emitting all haplotypes, scoring haplotype pairs, and then realigning reads to the top scoring haplotype pair.

Formulation

First, we start with the traditional formulation of a *de bruijn* graph for sequence assembly:

- 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 $\text{prefix}(s_1)^{\{1,k-2\}}+\text{suffix}(s_1) = \text{prefix}(s_2)$.

Additionally, we add the following:

- There is a set $\mathcal{R}$ which contains all of the k-mers that are in the reference genome that covers a certain region.
- If k-mer $s \in \mathcal{R}$, then the output of function $\text{refPos}(s)$ is defined. This function provides us with the integer position of s in the reference genome.
- For two k-mers $s_1, s_2 \in \mathcal{R}$, we can define a distance function $\text{distance}(s_1, s_2) = |\text{refPos}(s_1) - \text{refPos}(s_2)|$.

As stated above, this condition assumes that there is a 1-dimensional reference (e.g., a single assembled reference). In reality, genomic assemblies provide a 2-dimensional space, where the second dimension is defined by the chromosome that we are on. In practice, we will reassemble reads from a region which is restricted to a single chromosome.

To discover alternate alleles from this graph without performing a search for all haplotypes, we need to address the k-mers that are not in $\mathcal{R}$. These can be processed with path-based methods. First, let us classify paths into two types:

- **Spurs:** A spur is a set S of n k-mers $\{s_1, \ldots, s_n\}$ where either s_1 or $s_n \in \mathcal{R}$ and all other k-mers are $\notin \mathcal{R}$, and where $s_i \rightarrow s_{i+1} \forall i \in \{1, \ldots, n-1\}$. If $s_1 \in \mathcal{R}$, then s_n must not have a successor. Alternatively, if $s_n \in \mathcal{R}$, than s_1 is required to not have a predecessor.
- **Bubbles:** A bubble is a set S of n k-mers $\{s_1, \ldots, s_n\}$ where both s_1 and $s_n \in \mathcal{R}$ and all other k-mers are $\notin \mathcal{R}$, and where $s_i \rightarrow s_{i+1} \forall i \in \{1, \ldots, n-1\}$.

Currently, we do not process spurs. Spurs typically result from sequencing errors near the start or end of a read. Additionally, given a spur, we cannot put a constraint on what sort of edit it may be from the reference.

However, it is easy to define constraints for bubbles so that we can determine what variant has been seen at that site. For many variants, we can even canonically determine the variant. First, let us make three definitions:

- The *bubble length* is the number of non-reference k-mers that appear in the bubble.
- The *gap length* is the distance between s_1 and s_n in the reference genome. Formally, $l_{\text{gap}} = \text{distance}(s_1, s_n)$.
- The *allele length* is the absolute difference between the bubble length and the gap length.

With these definitions, we can define the allelic composition of bubbles as follows:

- A bubble contains a single/multiple nucleotide variant (S/MNV) if $l_{\text{allele}} = 0$.
- A bubble contains a canonical insert if $l_{\text{bubble}} = l_{\text{gap}} + l_{\text{allele}}$.
- A bubble contains a canonical deletion if $l_{\text{bubble}} < k$, where k is the k-mer size, and $\text{distance}(s_1, s_n) > 0$.
- If a bubble does not meet any of the above constraints, it is a non-canonical indel.

We discuss how to process these alleles in §3.2. It is also worth noting that this approach can be used for detecting structural variants. A simple extension involves jointly processing multiple reference regions (e.g., $\mathcal{R}_1$ and $\mathcal{R}_2$). If s_1 and s_n are members of different reference sets, then this bubble identifies a possible structural variant breakpoint. While this section has just provided intuition about the canonical representation of alleles from a de bruijn graph, we provide full proofs in §3.4 at the end of this chapter.

Reassembly Implementation

Reassembly is performed as a two step process. In the first step, we build a reference threaded de bruijn graph using the sequence of reference region $\mathcal{R}$ and the reads from sample n. Once we have built a reference threaded de bruijn graph, we then elaborate the graph and identify the allelic content of all bubbles.

Graph Construction

A reference threaded de bruijn graph is constructed per sample. We start by extracting the reference k-mers. This algorithm—`addReferenceKmers`—is tail recursive.

We use a similar algorithm for processing the k-mers from the read dataset. However, this algorithm filters out k-mers that contain `N`s, checks to see whether the k-mer sequence has been seen before, and also stores information about the mapping quality, base quality, and read strand.

Graph Elaboration

To elaborate the graph (the `toObservations` class method of a `KmerGraph`), we rely on a tail-recursive state machine that pushes state onto a stack at branch points. Our machine has the following states:

- `R`, `Reference`: We are on a run of k-mers that are mapped to a position in $\mathcal{R}$.
- `A`, `Allele`: We are on a run of k-mers that have diverged off of the reference. We have a divergence start point, but have not connected back to the reference yet. This could be either a bubble or a spur.

Algorithm 3 Incorporate Reference k-mers: `addReferenceKmers(iter, pos, lastKmer, kmerMap)`

```
iter ← iterator across k-mer strings from the reference
pos ← current reference position
lastKmer ← the last k-mer seen
kmerMap ← a map from k-mer strings → objects
if iter ≠ ∅ then
  ks ← iter.next
  if lastKmer ≠ ∅ then
    newKmer ← Kmer( ks, pos, predecessors = {lastKmer})
    lastKmer.successors ← lastKmer.successors + newKmer
  else
    newKmer ← Kmer( ks, pos)
  end if
  newPos ← pos + 1
  kmerMap ← kmerMap + (ks, newKmer)
  addReferenceKmers( iter, newPos, newKmer, kmerMap)
end if
```

- `C`, `ClosedAllele`: We were on a run of k-mers that had diverged off of the reference, but have just reconnected back to the reference and now know the start and end positions of the bubble, as well as the alleleic content of the bubble.

The following state transitions are allowed:

- `R` → `R`: We are continuing along a reference run.
- `R` → `A`: We were at a reference mapped k-mer, and have seen a branch to a non-reference k-mer.
- `A` → `A`: We are continuing along a non-reference run.
- `A` → `C`: We were on a non-reference run, and have just connected back to a reference k-mer.
- `C` → `R`: We have just closed out an allele, and are back at a reference position.

We initialize the state machine to 'R' with the first k-mer from $\mathcal{R}$. Per k-mer, we evaluate the state transition per successor k-mer. If the successor set contains a single k-mer, we continue to that state. If the successor set contains multiple k-mers, we choose a successor state to transition to, and push the branch context of all other successor states onto our stack. If the successor set is empty, we pop a branch context off of the stack, and

switch to that context. We stop once we reach a k-mer whose successor set is empty, and our branch context stack is empty.

The implementation of the `R` and `A` states are trivial and largely amount to bookkeeping. The `C` state though, is more complex. First, let us discuss how to canonicalize the canonical alleles:

- For a canonical S/MNV bubble, the SNV sites can be determined by calculating the Hamming distance of the bubble sequence versus the reference sequence. Evidence for SNVs exist at any site where an edit is made.
- For a canonical insert, sequence is inserted $k-1$ bases from the bubble start. There will be l_{allele} bases inserted. We can derive the inserted bases by computing the bubble sequence and trimming the first $k-1$ bases.
- For a canonical deletion, we guarantee that the bubble sequence is derived from the start and end of the reference sequences that the bubble overlaps. To canonicalize this, we can perform a greedy match of the bubble sequence, where we match bases to the front of the deletion until we see a mismatch, and then align all remaining bases to the end of the deletion.

For a complex variant, we take the bubble sequence and align it against the reference sequence from the bubble gap using a pairwise HMM [13]. From the HMM alignment, we emit observations according to the observed alignment states. The HMM uses an infinite padding penalty to ensure that the ends of the sequences are matched.

3.3 Statistical Models for Genotyping

Given a set of observed alleles and reads spanning these alleles, we can put together a graph. Each allele is given a node. We connect two nodes with an edge if and only if a read is observed to cover both nodes. In Figure 3.1, we present a sequence where the reference is `ACCCTATCGCTCACA`. Evidence of a `C` $\rightarrow$ `G` single nucleotide polymorphism (SNP) is present at position 3, and evidence of a deletion of bases 8–9 (`GC`) is present.

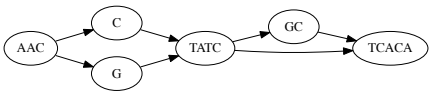

Figure 3.1: Allele Graph

Between two nodes, we can define a conditional probability $P(x = X|y = Y)$, which is the probability that node x has copy number X and node y has copy number Y:

$$P(x = X|y = Y) = \frac{1}{Y^k} \prod_{i=1}^{j} (XP_i + (Y - X)(1 - P_i)) \prod_{i=j+1}^{k} ((Y - X)P_i + XP_i) \tag{3.1}$$

In equation (3.1), we have the following terms:

Table 3.1: Equation (3.1) Terms

Term	Definition
X	The copy number of the x allele.
Y	The copy number of the y allele.
k	The number of reads in the y allele that span *any* allele in the x direction.
i	Reads $\in k$ that support a transition from $y \to x$.
j	Reads $\in k$ that do *not* support a transition from $y \to x$.
P_i	The probability that read i supports a transition from $y \to x$.

While equation (3.1) is useful, it only covers a single, uninteresting scenario. Specifically, it does not cover the case where we have a fork in the graph. In this case, we can define a further equation:

$$P(x = X|y_1 = Y_1, \ldots, y_n = Y_n) = \sum_{X_1+\cdots+X_n=X, X_i \geq 0} P(X_1, \ldots, X_n|\frac{Y_1}{\sum Y}, \ldots, \frac{Y_n}{\sum Y}) \prod_{i=1}^{n} P(x = X|y_i = Y_i) \tag{3.2}$$

where $P(X_1, \ldots, X_n|\frac{Y_1}{\sum Y}, \ldots, \frac{Y_n}{\sum Y})$ is multinomial.

With equation (3.2) defined, we can calculate full conditional probabilities for all nodes in the graph. Once we've done this, we can marginalize out all the conditional probabilities by running elimination from the root of the graph. We do make a single assertion here: specifically, we assume that all spurs are trimmed from the allele graph, and we have a clearly defined "root" for the graph. This may be easy to provide in a local assembly scenario, where we have a clearly defined start to the active region for reassembly; for *de novo* assembly, we may be able to define this by finding a cut of the graph that minimizes a certain function.

3.4 Proofs of Allele Canonicality

To construct an *indexed de Bruijn* graph, we start with the traditional formulation of a *de Brujin* graph for sequence assembly:

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

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

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

1. If two sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ share at least two k-mers s_1 and s_2, we can provably find the maximum edit distance d of the subsequences in $\mathcal{S}_i$ and $\mathcal{S}_j$, and bound the cost of finding this edit distance at $\mathcal{O}(nd)$,[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., $s_1 \rightarrow s_n \rightarrow \cdots \rightarrow s_{n+m} \rightarrow s_2$ and $s_{n+k} \in \mathcal{S}_i \wedge s_{n+k} \in \mathcal{S}_j \forall k \in \{0, \ldots, m\}$). Here, the subsequences are clearly identical. This determination can be made trivially by walking from vertex s_1 to vertex s_2 with $\mathcal{O}(m)$ cost.

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

Definition 2 (Path Independence)**.** *Given a non-repetitive* de Bruijn *graph G constructed from $\mathcal{S}_i$ and $\mathcal{S}_j$, we say that G contains independent paths between s_1 and s_2 if we can construct two subsets $\mathcal{S}'_i \subset \mathcal{S}_i, \mathcal{S}'_j \subset \mathcal{S}_j$ of k-mers where $s_{i+n} \in \mathcal{S}'_i \forall n \in \{0, \ldots, m_i\}, s_{i+n-1} \rightarrow s_{i+n} \forall n \in \{1, \ldots, m_i\}$, $s_{j+n} \in \mathcal{S}'_j \forall n \in \{0, \ldots, m_j\}, s_{j+n-1} \rightarrow s_{j+n} \forall n \in \{1, \ldots, m_j\}$, and $s_1 \rightarrow s_i, s_j; s_{i+m_i}, s_{j+m_j} \rightarrow s_2$ and $\mathcal{S}'_i \bigcap \mathcal{S}'_j = \emptyset$, where $m_i = \text{distance}_{\mathcal{S}_i}(s_1, s_2)$, and $m_j = \text{distance}_{\mathcal{S}_j}(s_1, s_2)$. This implies that the sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ are different between s_1, s_2,*

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

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 3.2.

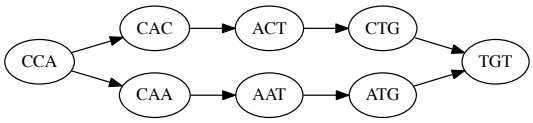

Figure 3.2: Subgraph Corresponding To a Single Nucleotide Edit

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

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

$k+l-1$. 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 3.3.

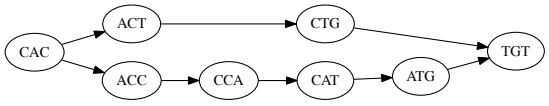

Figure 3.3: Subgraph Corresponding To a Canonical INDEL Edit

Where we have a canonical allele, the cost of computing the edit is set by the need to walk the graph linearly from s_1 to s_2, and is therefore $\mathcal{O}(n)$. However, in practice, we will see differences that cannot be described as one of the earlier two canonical approaches. First, let us generalize from the two above proofs: if we have two independent paths between s_1, s_2 in the *de Bruijn* graph G that was constructed from $\mathcal{S}_i, \mathcal{S}_j$, we can describe $\mathcal{S}_i$ as a sequence created by concatenating $s_1, \mathcal{L}_i, s_2$.[3] The canonical edits merely result from special cases:

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

Conceptually, a non-canonical edit occurs when two edits occur within k positions of each other. In this case, we can trivially fall back on a $O(nm)$ local alignment algorithm (e.g., a pairwise HMM or Smith-Waterman, see [13, 44]), *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 [49]). By this, we can achieve $O(n(d+1))$ cost.

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

Chapter 4

Performance and Accuracy Analysis

Thus far, we have discussed ways to improve the performance of scientific workloads that are being run on commodity MapReduce systems by rethinking how we decompose and build algorithms. In this section, we review the improvements in performance that we are able to unlock. We achieve near-linear speedup across 128 nodes for a genomics workload, and achieve a 3× performance improvement over the current best MPI-based system for the Montage astronomy application. Additionally, both systems achieve 25-50% compression over current file formats when storing to disk.

4.1 Genomics Workloads

Table 4.1 previews our performance versus current systems. The tests in this table are run on the high coverage `NA12878` full genome BAM file that is available from the 1000 Genomes project.[1] These tests have been run on the EC2 cloud, using the instance types listed in Table 4.2. We compute the cost of running each experiment by multiplying the number of instances used by the total wall time for the run and by the cost of running a single instance of that type for an hour, which is the process Amazon uses to charge customers.

Table 4.2 describes the instance types. Memory capacity is reported in Gibibytes (GiB). Storage capacities are not reported in this table because disk capacity does not impact performance, but the number and type of storage drives is reported because aggregate disk bandwidth does impact performance. In our tests, the `hs1.8xlarge` instance is chosen to represent a workstation. Network bandwidth is constant across all instances.

As can be seen from these results, our pipeline is at best three times faster than current pipelines when running on a single node; at worst, we are approximately at parity. Additionally, `ADAM` achieves speedup that is close to linear. This point is not clear from Table 4.1, as we change instance types when also changing the number of instances used. Figure 4.1 presents speedup plots for the NA12878 high coverage genome.

[1] The file used for these experiments can be found on the 1000 Genomes ftp site, `ftp.1000genomes.ebi.ac.uk` in directory `/vol1/ftp/data/NA12878/high_coverage_alignment/` for NA12878.

Table 4.1: Summary Performance on NA12878

Sort			
Software	**EC2 profile**	**Time**	**Cost**
Picard	1 `hs1.8xlarge`	17h 44m	$81.57
ADAM	1 `hs1.8xlarge`	8h 56m	$41.09
ADAM	128 `m2.4xlarge`	9m	$31.49
Mark Duplicates			
Software	**EC2 profile**	**Time**	**Cost**
Picard	1 `hs1.8xlarge`	20h 22m	$93.68
ADAM	1 `hs1.8xlarge`	9h	$41.40
ADAM	128 `m2.4xlarge`	19m	$64.48
BQSR			
Software	**EC2 profile**	**Time**	**Cost**
GATK	1 `hs1.8xlarge`	31h 18m	$143.98
ADAM	1 `hs1.8xlarge`	34h	$156.40
ADAM	128 `m2.4xlarge`	54m	$188.93
INDEL Realignment			
Software	**EC2 profile**	**Time**	**Cost**
GATK	1 `hs1.8xlarge`	42h 49m	$196.88
ADAM	1 `hs1.8xlarge`	12h 58m	$82.80
ADAM	128 `m2.4xlarge`	24m	$83.97
Flagstat			
Software	**EC2 profile**	**Time**	**Cost**
SAMtools	1 `hs1.8xlarge`	25m 24s	$1.95
ADAM	1 `hs1.8xlarge`	7m 4s	$1.95
ADAM	128 `cr1.8xlarge`	1m 53s	$5.35

When testing on NA12878, we achieve linear speedup out through 1024 cores; this represents 128 `m2.4xlarge` nodes. In this test, our performance is limited by several factors:

- Although columnar stores have very high read performance, their write performance is low. Our tests exaggerate this penalty; since a variant calling pipeline will consume a large read file, but then output a variant call file that is approximately two orders of magnitude smaller, the write penalty will be reduced. In practice, we also use in-memory caching to amortize write time across several stages.
- Additionally, for large clusters, straggler elimination is an issue. However, we have made optimizations to both the `Mark Duplicates` and `INDEL Realignment` code to

Table 4.2: AWS Machine Types

Machine	Cost	Description
`hs1.8xlarge`	$4.60/hr	16 proc, 117GiB RAM, 24× HDD
`m2.4xlarge`	$1.64/hr	8 proc, 68.4GiB RAM, 2× HDD

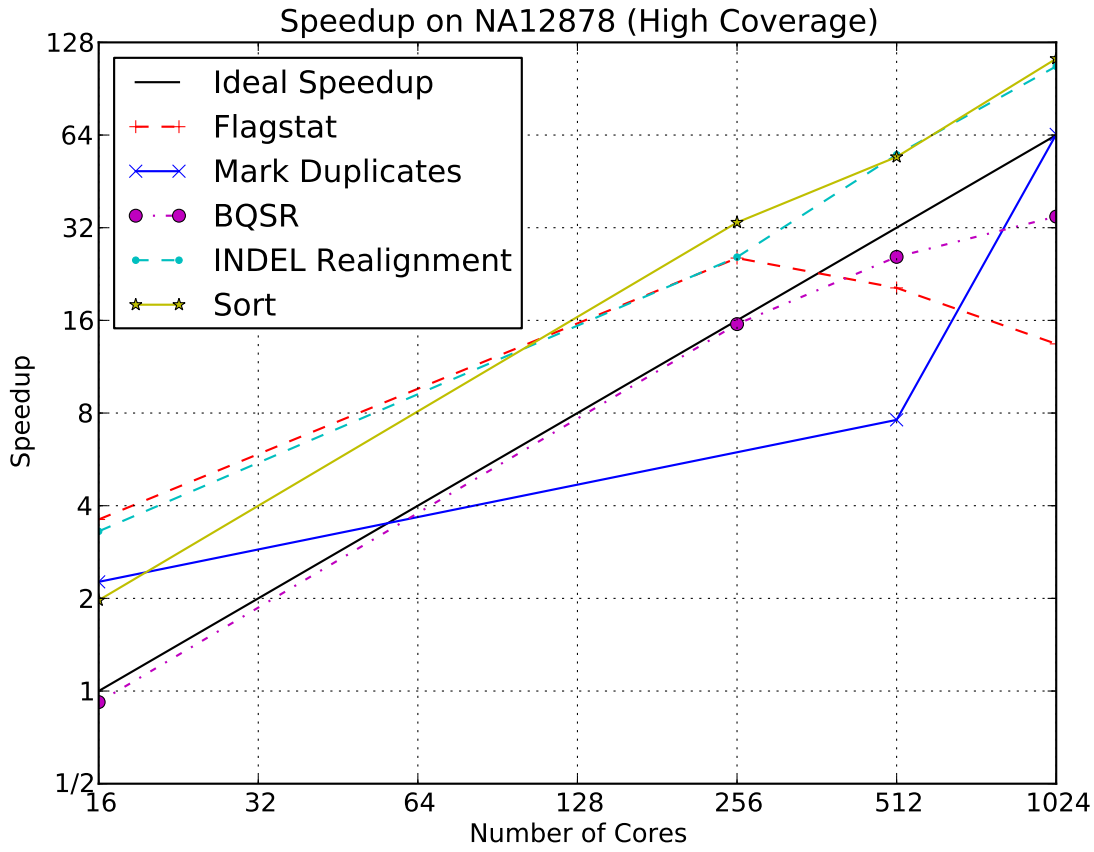

Figure 4.1: Speedup on NA12878

eliminate stragglers by randomly rebalancing reads that are unmapped/do not map to a target across partitions.

We do note that the performance of `flagstat` degrades going from 32 to 128 `m2.4xlarge` nodes. It is worth noting that `flagstat` executes in one minute on 32 nodes. By increasing the number of machines we use to execute this query, we increase scheduling overhead, which

leads to degraded performance.[2]

4.2 Column Store Performance

Earlier in this paper, we motivated the use of a column store as it would allow us to better push processing to the data. Specifically, we can use predicate pushdown and projections to minimize the amount of I/O that we perform. Additionally, column stores provide compressed storage, which allows us to minimize both the required I/O bandwidth and space on disk. In this section, we'll look at the performance that our columnar store achieves in terms of read performance and compression. We will not look extensively at write performance; for genomic data, write performance is not a bottleneck because our workflow computes a *summarization* of a large dataset. As a result, our output dataset tends to be O(100 MB) while our input dataset is in the range of O(10 GB)–O(100GB).

Compression

The Parquet columnar store [4] supports several compression features. Beyond traditional block-level compression, Parquet supports run length encoding for repeated values, dictionary encoding, and delta encoding. Currently, we make use of run length encoding to compress highly repeated metadata value, and dictionary encoding to compress fields that can take a limited range of values. Dictionary encoding provides substantial improvements for genomic data; specifically, the majority of genomic sequence data can be represented with three bits per base.[3] This is an improvement over our in-memory string representation which allocates a byte per base.

In Table 4.3, we look at the compression we achieve on the `NA12878` and `HG00096`[4] human genome sequencing samples. We compare against the GZIP compressed BAM [30] format, and the CRAM format [16]. We achieve approximately a 1.25× improvement in storage. This is not as impressive as the result achieved by the CRAM project, but the CRAM project applies specific compression techniques that make use of the read alignment. Specifically, CRAM only stores the read bases that *do not* appear in the reference genome. As we only expect a genomic variant at one in every 1000 bases, and a read error at one in every 50 bases, this allows them to achieve significant compression of the sequenced bases.

For genomic datasets, our compression is limited by the sequence and base quality fields, which respectively account for approximately 30% and 60% of the space spent on disk.

[2]While we have tested against the SAMtools/Picard/GATK pipeline, we do note that new implementations have come out recently (e.g., SAMBAMBA and SAMBLASTER [15]) that focus on fast duplicate marking. We have not compared to them due to time limitations, but will compare to them in a later revision of this paper.

[3]Although DNA only contains four bases (A, C, G, and T), *sequenced* DNA uses disambiguation codes to indicate that a base was read in error. As a result, we cannot achieve the ideal two-bits per base.

[4]A link to the `NA12878` dataset was provided earlier in this paper. The `HG00096` dataset is available from `ftp.1000genomes.ebi.ac.uk` in directory `/vol1/ftp/data/HG00096/alignment/`.

Table 4.3: Genomic Data Compression

NA12878		
Format	**Size**	**Compression**
BAM	234 GB	—
CRAM	112 GB	2.08×
Parquet	185 GB	1.26×
HG00096		
Format	**Size**	**Compression**
BAM	14.5 GB	—
CRAM	3.6 GB	4.83×
Parquet	11.4 GB	1.27×

Quality scores are difficult to compress because they are high entropy. We are currently looking into computational strategies to address this problem; specifically, we are working to probabilistically estimate the quality scores *without* having observed quality scores. This would be performed via a process that is similar to the base quality score recalibration algorithm presented earlier in this paper.

Horizontal Scalability

The representation Parquet uses to store data to disk is optimized for horizontal scalability in several ways. Specifically, Parquet is implemented as a hybrid row/column store where the whole set of records in a dataset are partitioned into row groups which are then serialized in a columnar layout. This provides us with two additional benefits:

1. We are able to perform parallel access to Parquet row groups without consulting metadata or checking for a file split.

2. Parquet achieves very even balance across partitions. On the HG00096 dataset, the average partition size was 105 MB with a standard deviation of 7.4 MB. Out of the 116 partitions in the file, there is only one partition whose size is not between 105–110MB.

Parquet's approach is preferable when compared to Hadoop-BAM [37], a project that supports the direct usage of legacy BAM files in Hadoop. Hadoop-BAM must pick splits, which adds non-trivial overhead. Additionally, once Hadoop-BAM has picked a split, there is no guarantee that the split is well placed. It is only guaranteed that the split position will not cause a *functional error*.

Projection and Predicate Performance

We use the `flagstat` workload to evaluate the performance of predicates and projections in Parquet. We define three projections and four predicates, and test all of these combinations. In addition to projecting the full schema (see Appendix **??**), we also use the following two projections:

1. We project the read sequence *and* all of the flags (40% of data on disk).
2. We only project the flags (10% of data on disk).

Beyond the null predicate (which passes every record), we evaluate the following three predicates:

1. We pass only uniquely mapped reads (99.06% of reads).
2. We pass only the first pair in a paired end read (50% of reads).
3. We pass only *un*mapped reads (0.94% of reads).

Our performance is documented in Table 4.4. Projections are arranged in the columns of the table while predicates are assigned to rows.

Table 4.4: Predicate/Projection Speedups

	0	1	2
0	—	1.7	1.9
1	1.0	1.7	1.7
2	1.3	2.2	2.6
3	1.8	3.3	4.4

We achieve a 1.7× speedup by moving to a projection that eliminates the deserialization of our most complex field (the quality scores that consume 60% of space on disk), while we only get a 1.3× performance improvement when running a predicate that filters 50% of records. This can be partially attributed to overhead from predicate pushdown; we must first deserialize a column, process the filter, and then read all records who passed the pushdown filter. If we did not perform this step, we could do a straight scan over all of the data in each partition.

Chapter 5

Conclusion

5.1 Discussion

5.2 Future Work

Our genomics work leverages columnar storage to improve performance and compression of data on disk, with special emphasis on repetitive fields that can be run length encoded. While this improves disk performance, it has the side effect of making data consume significantly more space in memory than on disk. We are currently investigating techniques that leverage the immutability of data in our applications to reduce memory consumption. We have changed Parquet and Avro's deserialization codec to reuse allocated objects. For every value that is RLE'd, we only allocate the value once in memory. We then share the value across all records which contained that value. This is especially important since we maintain a lot of string metadata which is RLE'd on disk.

It is worth noting that there are many significant scientific applications (such as genome assembly) that are expressed as traversal over graphs. Recent work by Simpson et al (ABySS, [42]) and Georganas et al [18] has focused on using MPI or Unified Parallel C (UPC) to implement their own distributed graph traversal. Both systems find that synchronization via message passing is a significant cost; specifically, the ABySS assembler experiences scaling problems because it thrashes portions of the graph across nodes during traversal. By building our system using Spark, we are able to leverage the GraphX processing library [19, 52]. We are in the process of developing a genome assembler using this library system, and believe that we can achieve improved performance through careful graph partitioning. This partitioning involves algorithmic changes to the graph creation and traversal phases to bypass "knotted" sections of the graph that correspond to highly repetitive areas of the genome, which cause the major performance issues in MPI based assemblers.

5.3 Conclusion

In this paper, we have advocated for an architecture for decomposing the implementation of a scientific system, and then demonstrated how to efficiently implement genomic and astronomy processing pipelines using the open source Avro, Parquet, and Spark systems [2, 4, 56]. We have identified common characteristics across scientific systems, like the need to run queries that touch slices of datasets and the need for fast access to metadata. We then enforced data independence through a layering model that uses a schema as the "narrow waist" of the stack, and used optimizations to make common, coordinate-based processing fast. By using Parquet, a modern columnar store, we use predicates and projections to minimize I/O, and are able to denormalize our schemas to improve the performance of accessing metadata.

By rethinking the architecture of scientific data management systems, we have been able to achieve 22–131× performance improvements over conventional genomics processing systems, along with linear strong scaling and a 2× cost improvement. On the astronomy workload, we achieve speedup between 2.8–8.9× speedup over the current best MPI-based solution at various scales. By applying our techniques to both astronomy and genomics, we have demonstrated that the techniques are applicable to both traditional matrix-based scientific computing, as well as novel scientific areas that have less structured data.

Bibliography

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of Data (SIGMOD '06)*, pages 671–682. ACM, 2006.

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

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

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

[5] G. A. Auwera, M. O. Carneiro, C. Hartl, R. Poplin, G. del Angel, A. Levy-Moonshine, T. Jordan, K. Shakir, D. Roazen, J. Thibault, et al. From FastQ data to high-confidence variant calls: The Genome Analysis Toolkit best practices pipeline. *Current Protocols in Bioinformatics*, pages 11–10, 2013.

[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] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, pages 963–968. ACM, 2010.

[8] 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. *bioRxiv*, page 012179, 2014.

[9] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04)*. ACM, 2004.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[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] Y. Diao, A. Roy, and T. Bloom. Building highly-optimized, low-latency pipelines for genomic data analysis. In *Proceedings of the 7th Conference on Innovative Data Systems Research (CIDR '15)*, 2015.

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

[14] G. England. 100,000 genomes project. `https://www.genomicsengland.co.uk/`.

[15] G. G. Faust and I. M. Hall. Samblaster: fast duplicate marking and structural variant read extraction. *Bioinformatics*, page btu314, 2014.

[16] 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.

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

[18] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. Parallel de bruijn graph construction and traversal for de novo genome assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, 2014.

[19] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 16th Symposium on Operating System Design and Implementation (OSDI '14)*. ACM, 2014.

[20] 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.

[21] K. J. Karczewski, G. H. Fernald, A. R. Martin, M. Snyder, N. P. Tatonetti, and J. T. Dudley. STORMSeq: An open-source, user-friendly pipeline for processing personal genomics data in the cloud. *PloS one*, 9(1):e84860, 2014.

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

[23] H. Y. Lam, C. Pan, M. J. Clark, P. Lacroute, R. Chen, R. Haraksingh, M. O'Huallachain, M. B. Gerstein, J. M. Kidd, C. D. Bustamante, et al. Detecting and annotating genetic variations using the HugeSeq pipeline. *Nature biotechnology*, 30(3):226–229, 2012.

[24] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[25] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[26] 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.

[27] 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.

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

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

[30] 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.

[31] Y. Li, A. Terrell, and J. M. Patel. WHAM: A high-throughput sequence alignment method. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*, SIGMOD '11, pages 445–456, New York, NY, USA, 2011. ACM.

[32] 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.

[33] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[34] M. L. Metzker. Sequencing technologies—the next generation. *Nature Reviews Genetics*, 11(1):31–46, 2009.

[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] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

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

[40] 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.

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

[42] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol. ABySS: a parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.

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

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

[45] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. MLI: An API for distributed machine learning. In *13th IEEE International Conference on Data Mining (ICDM '13)*, pages 1187–1192. IEEE, 2013.

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

[47] 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.

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

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

[50] J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. M. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, J. M. Stuart, C. G. A. R. Network, et al. The cancer genome atlas pan-cancer analysis project. *Nature Genetics*, 45(10):1113–1120, 2013.

[51] N. I. Weisenfeld, S. Yin, T. Sharpe, B. Lau, R. Hegarty, L. Holmes, B. Sogoloff, D. Tabbaa, L. Williams, C. Russ, et al. Comprehensive variation discovery in single human genomes. *Nature genetics*, 2014.

[52] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[53] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

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

[55] 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 9th USENIX conference on Networked Systems Design and Implementation (NSDI '12)*, page 2. USENIX Association, 2012.

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

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