Rethinking Data-Intensive Science Using Scalable Analytics Systems

ABSTRACT

"Next generation" data acquisition technologies are allowing scientists to collect exponentially more data at a lower cost. These trends are broadly impacting many scientific fields, including genomics, astronomy, and neuroscience. We can attack the problem caused by exponential data growth by applying horizontally scalable techniques from current analytics systems to accelerate scientific processing pipelines.

In this paper, we demonstrate an example genomics pipeline that leverages open-source MapReduce and columnar storage techniques to achieve a 22–130× speedup over current genomics pipelines, while reducing cost by 50%. From building this system, we were able to distill a set of techniques for implementing scientific analyses efficiently using commodity "big data" systems. To demonstrate the generality of our architecture, we then implement a scalable astronomy image processing system which achieves a 2.8–8.9× improvement over the state-of-the-art MPI-based system.

Categories and Subject Descriptors

L.4.1 [**Applied Computing**]: Life and medical sciences—*Computational biology*; H.1.3.2 [**Information Systems**]: Data management systems—*Database management system engines, parallel and distributed DBMSs*; E.3.2 [**Software and its Engineering**]: Software creation and management—*Software Development Process Management*

General Terms

Design

Keywords

Analytics, MapReduce, Genomics, Scientific Computing

1. INTRODUCTION

Major improvements in scientific data acquisition techniques have increased scientific data storage and processing needs [8, 36]. In fields like neuroscience [15] and genomics [41], particle physics, and astronomy, scientists routinely perform analysis that use terabytes (TB) to petabytes (PB) of data. While traditional scientific computing platforms are optimized for fast linear algebra, many emerging domains make heavy use of statistical learning techniques and user defined functions (UDFs) on top of semistructured data. This move towards statistical techniques has been driven by the increase in the amount of data available to scientists. At the same time, commercial needs have led to the development of horizontally-scalable analytics systems like MapReduce [9, 10] and Spark [51], as well as statistical systems that are accessible to non-experts, such as `Scikit-learn` [34] and `MLI` [40].

Since the amount of scientific data being generated is growing so quickly, there is a good opportunity to apply modern, horizontally scalable analytics systems to science. New scientific projects such as the 100K for UK, which aims to sequence the genomes of 100,000 individuals in the United Kingdom [13] will generate three to four *orders of magnitude* more data than prior projects like the 1000 Genomes Project [39]. These projects use the current "best practice" genomic variant calling pipeline [4], which takes approximately 120 hours to process a single, high-quality human genome using a single, beefy node [42]. To address these challenges, scientists have started to apply computer systems techniques such as MapReduce [24, 28, 37] 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 paper, we demonstrate a genomic data processing system built using Apache Avro, Parquet, and Spark [2, 3, 51], that achieves a 50× increase in throughput over the current best practice pipeline, while reducing analysis cost by 50%. In the process of creating this system, we developed a "narrow waisted" layering model for building similar scientific analysis systems. This narrow waisted stack is inspired by the OSI model for networked systems [53]. 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 and enforce data independence. We then devise algorithms for making common scientific processing patterns fast (e.g., coordinate-space joins, see §5.1).
2. To improve horizontal scalability, we use Parquet, a modern parallel columnar store based off of Dremel [29] to push computation to the data.
3. We use a denormalized schema to achieve O(1) parallel access to metadata.

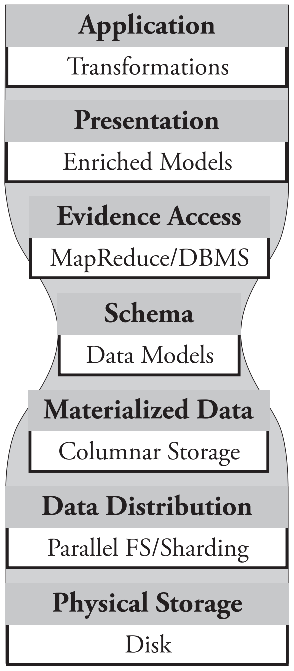

Figure 1: A Stack Model for Scientific Computing

We introduce the stack model in figure 1 as a way to decompose scientific systems. In addition to the genomics application described above, we demonstrate the generality of this model by using it to implement a system for processing astronomy images, which achieves a 2.8–8.9× performance improvement over a state-of-the-art Message Passing Interface (MPI) based pipeline.

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 [26]) formats for storing genomic alignments apply constraints about record ordering to enable specific computing patterns. Our implementation (described in §4.1) 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.

We have made all of the software (source code and executables) described in this paper available free of charge under the permissive Apache 2 open-source license.

2. BACKGROUND

Our work is at the intersection of computational science, 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 has changed dramatically since the papers by Dean and Ghemawat [9, 10] that described Google's `MapReduce` system. Over a similar timeframe, scientific fields have moved to take advantage of improvements in data acquisition technologies. For example, since the Human Genome Project finished in 2001 [23], the price of genomic sequencing has dropped by 10,000× [32]. 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 [39] and The Cancer Genome Atlas (TCGA, [45]). These changes are not unique to genomics; indeed, fields such as neuroscience [8] and astronomy [19, 44, 48] are experiencing similar changes.

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 [51], as well as parallel and columnar DBMS [1, 22]—the uptake of these systems in the scientific world has been slow. Most implementations have either used MapReduce as an inspiration for API design [28], or have been limited systems that have used MapReduce to naïvely parallelize existing toolkits [24, 37]. 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 [28] 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 [51, 50] 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),

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

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 [24] and Cloudburst [37] use scripts to run unmodified legacy tools on top of Hadoop. This approach does achieve speedups, but does not attack any 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 §4.1) will change the inferred quality score distribution. Additionally, we believe that the shuffle traffic that Diao et al observe is an artifact caused by the abstraction inversion discussed in §1. As we demonstrate in §4.1, these penalties can be eliminated by restructuring the preprocessing 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 §6.3.

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 [6]. 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 [21] have extended SQL to provide efficient query semantics across genomic coordinates. While GQL achieves performance improvements of up to $10\times$ 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.

3. CHARACTERISTICS OF SCIENTIFIC ANALYSIS SYSTEMS

Most prior work on scientific computing has been focused on linear algebra and other problems that can be structured as a matrix or network. However, in several of the emerging data-driven scientific disciplines, data is less rigorously structured. As discussed in §2, scientists have been developing ad hoc solutions to process this data. In this section, we discuss the common characteristics of workloads in these emerging scientific areas. Given these characteristics, we describe a way to decompose data processing and storage systems so that we can efficiently implement important processing patterns while providing a wide range of data access methods.

3.1 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 §4.1, 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 [53]. 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 [26], 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, 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 §4.1, 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.[2] To resolve this conflict, we demonstrate several ways to efficiently implement conventional scientific traversals in §5. These traversals are implemented in the evidence access layer, and are independent of anything below the schema.

The idea of decomposing scientific applications into a stack model is not new; Bafna et al [5] 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.

[2]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.

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

- Notably, Bafna et al use this stack model to motivate GQL [21]. 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.

Our stack enables us to serve the use cases we outline in §3.2. By using Parquet as a storage format, we are able to process the data in 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.

3.2 Workloads

There are several common threads that unify the diverse set of applications that make up scientific computing. When looking at the data that is used in different fields, several trends pop out:

1. Scientific data tends to be rigorously associated with coordinates in some domain. These coordinate systems vary, but can include:
   - Time (e.g., fMRI data, particle simulations)
   - Chromosomal position (e.g., genomic read alignments and variants)
   - Position in space (imaging data, some sensor datasets)
2. For aggregated data, we frequently want to slice data into many different views. For example, for time domain data aggregated from many sensors, scientists may want to perform analyses by slicing across a single point in time, or by slicing across a single sensor. In genomics, we frequently aggregate data across many people from a given population. Once we have done this first aggregation, we may want to then slice the data by subsets of the population, or by regions of the genome (e.g., specific genes of interest).

There are two important consequences of the characteristics above. First, since data is attached to a coordinate system, the coordinate system itself may impose logical processing patterns. For example, for time domain data, we may frequently need to run functions that pass a sliding window across the dataset (e.g., for convolution). Second, the slicing of aggregated data is frequently used to perform analyses across subsets of a larger dataset. This is common if we want to study a specific phenomenon, like the role of a gene in a disease (a common analysis in the TCGA [45]), or the measured activity in a single lobe of the brain while performing a task. Since the datasets we are processing are large,[3] it may be uneconomical to colocate data with processing nodes, because of either the number of nodes that would need to be provisioned, or the amount of storage that would need to be provisioned per node.

For scientific fields that process very large datasets, the exact processing techniques and algorithms vary considerably, but common processing trends do exist:

1. There is increasing reliance on statistical methods. The `Thunder` pipeline makes heavy use of the MLI/MLLib statistical libraries [15, 40], and tools like the GATK perform multiple rounds of statistical refinement [11].
2. Data parallelism is very common. This varies across applications; in some applications (like genomics), we may leverage the independence of sites across a coordinate system and process individual coordinate regions in parallel. For other systems, we may have matrix calculations that can be parallelized [40], or we may be able to run processing in parallel across samples or traces.

Additionally, there are several different emerging use cases for scientific data processing and storage systems. These different use cases largely correspond to different points in the lifecycle of the data:

- **Batch processing:** After the initial acquisition of raw sensor data (e.g., raw DNA reads, brain electrode traces, telescope images), we use a batch processing pipeline (e.g., `Thunder` or the GATK) to perform some dimensionality reduction/statistical summarization of the data. This is generally used to extract notable features from the data, such as turning raw genomic reads into variant alleles, or identifying areas of activity in neuroscience traces. These tasks are unlikely to have any interactive component, and are likely to be long running compute jobs.
- **Ad hoc exploration:** Once the batch processing has completed, there is often a need for exploratory processing of the results. For example, when studying disease genetics, a geneticist may use the variant/genotype statistics to identify genomic sites with statistically significant links to the disease phenotype. Data exploration tasks have a significant user facing/interactive nature, and are generally performed by scientists who may be programming laypeople.
- **Data warehousing:** In large scientific projects, it is common to make data available to the members of the scientific community through some form of warehouse service (e.g., the Cancer Genomics Hub, CGHub, for the TCGA). As is the case for all data warehousing, this implies that queries must be made reasonably efficient, even though the data is expected to be cold. To reduce the cost of storing data, we may prioritize compression here; this has led to the creation of compressed storage formats like CRAM [16].

[3]For example, the Acute Myeloid Leukemia subset of the TCGA alone is over 4 TB in size, and is only one of 20 cancers in the TCGA.

In this paper, we design a system that can achieve all of the above goals. The genomics and astronomy pipelines we demonstrate achieve improvements in batch processing performance, and allow for interactive/exploratory analysis through both Scala and Python. Through the layering principles we lay out in the next section and the performance optimizations we introduce in §5.2, we make our system useful for warehousing scientific data.

4. CASE STUDIES

To validate our architectural choices, we have implemented pipelines for processing short read genomic data and astronomy image processing. Both of these pipelines are implemented using Spark [51], Avro [2], and Parquet [3]. We have chosen these two applications as they fit in different areas in the design space. Specifically, the genomics pipeline makes heavy use of statistical processing techniques over semistructured data, while the astronomy application has a traditional matrix structure.

Corresponding to the stack model that was introduced in Figure 1, we use the following technologies to implement both of our applications:

- **Physical Storage:** We have designed our system to run on top of local or distributed drives, as well as block stores.
- **Data Distribution:** Our system is designed to operate on top of the Hadoop Distributed File System (HDFS), or to perform it's own data distribution over HTTP or by reaching out to an Amazon S3 bucket. We describe these optimizations in §5.2.
- **Materialized Data:** We store data using the open source Parquet columnar store.
- **Schema:** We manage our schemas (and data serialization) via the Avro serialization framework [2]. Our schemas are described in Appendix B.
- **Evidence Access:** We use Spark's Resilient Distributed Dataset (RDD, [50]) abstraction to provide parallel processing over the data. We enhance this with the join patterns we describe in §5.1.
- **Presentation:** In our genomics application, we provide several rich datatypes that implicitly wrap our schemas to provide convenience methods for metadata access. This is not as crucial in the astronomy application.

In the remainder of this section, we describe the two applications that we have implemented, and the optimizations we have made to improve the horizontal scalability of these algorithms.

4.1 Genomics Pipeline

Contemporary genomics has been revolutionized by "next generation" sequencing technologies (NGS), which have driven a precipitous drop in the cost of running genomic assays [32]. Although there are a variety of sequencing technologies in use, the majority of sequence data comes from the Illumina sequencing platform, which uses a "sequencing-by-synthesis" technique to generate short read data [30]. 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.

One of the most common genomic analyses is *variant calling*, which is a statistical process to infer the sites at that a single individual varies from the reference genome.[4] To call variants, 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 [25, 27, 49]. 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 §4.1; 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 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. While we do not present work on variant filtration in this paper, variant filtration has motivated the coordinate space joins presented in §5.1.

[4]The reference genome represents the "average" genome for a species. The Human Genome Project [23] assembled the first human reference genome.

This process is very expensive in time to run; the current best practice pipeline uses the BWA tool [25] for alignment and the GATK [11, 28] 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 [42]. Recent projects have achieved 5–10× improvements in alignment and variant calling performance [35, 49], 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. We have focused on implementing the four most-commonly used pre-processing stages, as well as `flagstat`, a command that is used at the end of pre-processing for validating the quality of an aligned/pre-processed sample. In the remainder of this section, we describe the stages that we have implemented, and the techniques we have used to improve performance and accuracy.

1. **Sorting:** This phase sorts all reads by the position of the start of their alignment. The implementation of this algorithm is trivial, as Spark provides a sort primitive [51]; we solely need to define an ordering for genomic coordinates, which is well defined.[5]

2. **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 [43], 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.[6] 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.

3. **Local Realignment:** In local realignment, we correct areas where variant alleles cause reads to be locally misaligned from the reference genome.[7] 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

[5]In practice, an explicit sort is unnecessary when using the rest of our MapReduce-based pipeline. We have included sort to enable the use of legacy tools that require sorted input.

[6]In a chimeric read pair, the two reads in the read pairs align to different chromosomes; see Li et al [25].

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

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 C.

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.

4. **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:

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

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 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.

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 [49]. 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 §6.3.

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 §6.1.

4.2 Astronomy Image Processing

The Montage [20] application is an astronomy image processing pipeline that builds "mosaic" images by combining small image tiles obtained from telescopes. Montage has the requirement of preserving the energy quantity and position of each pixel between the input and output images. The pipeline has the following four phases:

1. **Tile Reprojection:** The raw input images are reprojected with the scale and rotation required for the final mosaic.

2. **Background Modeling:** This phase smoothes out the background levels between each pair of overlapped images and fits a plane to each of them. The phase can be further divided into overlap calculation, difference image creation, and plane-fitting coefficient calculation.

3. **Background Matching:** The background matching phase performs background removal on the reprojected images. The best solution derived from the previous phase is used to smooth out the overlap between tiled images.

4. **Tile Mosaicing:** The tile masoning phase merges all corrected images into a aggregated mosaic file, after applying background matching to the reprojected images. This phase also includes a metadata processing stage prior to mosaicing.

The tile reproduction, background modeling, and background matching phases are embarrassingly parallel, with each task (both computation and I/O) running independent from other tasks in the same phase. The tile mosaicing phase aligns the tiles of the matrix and applies a function (average, median, or count) to the overlapped pixels. All functions can be applied to the overlapped pixels in an embarrassingly parallel manner.

We care the most about the tile mosaicing phase because the MPI implementation requires a preceding stage to summarize the metadata of all corrected images to produce a metadata table containing the tile positioning information. This particular stage is inefficient because it has to read all input files into memory, but only accesses a small portion of the file. In addition, the current implementation parallelizes the computation with each matrix row as the element, which results in an inefficient replication of input images when executed in a distributed environment. We address the metadata issue by explicitly integrating the metadata into the renormalized image data schema. We then store the images in a columnar store which allows all input images to be loaded into memory a single time. All subsequent computation proceeds in memory.

5. DATA ACCESS OPTIMIZATIONS FOR SCIENTIFIC PROCESSING

In §3.2, we discussed several processing patterns that were important for scientific data processing. In this section, we introduce optimizations for two important use cases. First, we present a join pattern that enables processing that traverses a coordinate system. Region joins enable distributed implementations of important genomics algorithms. Second, we implement an efficient method for applying predicates and projections into data that is stored in a remote block store. Enabling predicates and projections on remote data allows us to defer as much remote data movement as is possible and improves the efficiency of accessing remotely staged data.

5.1 Coordinate System Joins

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

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

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

To support these algorithms, we provide a "region" or "spatial" join primitive. The algorithm used is described in algorithm 1 and takes as input two sets (RDDs, see Zaharia et al [50]) of `ReferenceRegions`, a data structure that represents intervals along the 1-D genomics coordinate space. It produces, as output, the set of all `ReferenceRegion` pairs where a region from the first RDD overlaps with a region from the second RDD.

Algorithm 1 Partition And Join Regions

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

To find the maximal set of non-overlapping regions, we must find the convex hull of all regions emitted. We present a distributed algorithm for finding convex hulls in Appendix C. The distributed convex hull computation problem is important because it is used both for computing regions for partitioning during a region join and for performing INDEL realignment. We are currently working to improve our algorithm so that we can eliminate a reduce phase. This optimization will allow us to improve performance by eliminating a synchronization phase caused by Spark's Bulk Synchronous Parallel (BSP) execution model.

5.2 Loading Remote Data

Another challenge faced by scientific systems is where to store the initial data files and how to load them efficiently. Today, Spark is usually run in conjunction with the HDFS portion of the Hadoop stack—HDFS provides data locality, access to local disk on each node of the Spark cluster, and robustness to node failure. However, HDFS imposes significant constraints on running a Spark system in virtualized or commodity computing (e.g. "cloud") environments. It is easy to scale an HDFS-based system up to larger numbers of nodes, but harder to remove nodes when the capacity is no longer needed.

If we are willing to forgo the advantages of local disk and data locality provided by HDFS, however, we may be able to relax some of these other restrictions and build a Spark-based cluster whose size is more easily adjusted to the changing demands of the computation. By storing our data in higher-latency, durable, cheaper block storage (e.g., S3) we can also exploit the varying requirements of data availability—not all datasets need to be kept "hot" in HDFS at all times, but can be accessed in a piecemeal or parallelized manner through S3 interfaces.

Spark provides a particularly convenient abstraction for writing these new data access methods. By implementing our own data-loading RDD, we are able to allow a Spark cluster to access Parquet files stored in S3 in parallel (each partition in the RDD reflects a row group in the corresponding Parquet file). For Parquet files containing records that reflect known genomics datatypes (that are mapped to genomic locations, for example) we generate simple index files for each Parquet file. Each index file lists the complete set of row groups for the Parquet file, as well which genomic regions contain data points within each row group. Our parallelized data loader reads this index file and restricts the partitions in the data loading RDD it creates to only those Parquet row groups that possibly contain data relevant to the user's query.

6. PERFORMANCE

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.

6.1 Genomics Workloads

Table 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.[8] These tests have been run on the EC2 cloud, using the instance types listed in Table 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 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, `SparkG` achieves speedup that is close to linear. This point is not clear from Table 1, as we change instance types when also changing the number of instances used. Figure 2 presents speedup plots for the NA12878 high coverage genome.

Table 1: Summary Performance on NA12878

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

Table 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

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 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 go-

[8]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. For anonymity, we refer to our system as `SparkG` in this table.

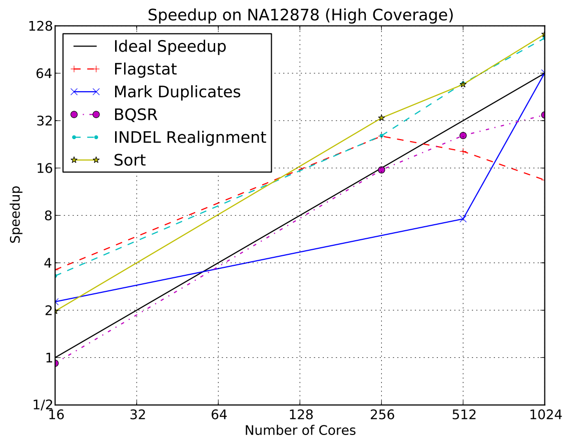

Figure 2: Speedup on NA12878

ing 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.[9]

6.2 Astronomy Workloads

To evaluate the mosaicing application, we use the 2MASS data[10] and the Montage test case of 3x3 degree mosaicing with Galaxy m101 as the center. The tile mosaicing phase converts 1.5 GB of input data into a 1.2 GB aggregated output file. We compare the `Spark-mAdd` performance against the High Performance Computing (HPC) styled MPI-based parallel implementation from Montage v3.3 (`MPI-mAdd`). We performed the test on 1, 4, and 16 Amazon `c3.8xlarge` instances. We chose the `c3.8xlarge` instances for this test because they provided HPC-optimized networking, which is a prerequisite for good MPI performance. We use OrangeFS v2.8.8—a successor of PVFS [7]—as the shared file system when running `MPI-mAdd`. All 32 cores on each instance are used for both `Spark-mAdd` and `MPI-mAdd`.

As shown in Figure 3, `Spark-mAdd` runs 2.8x, 5.7x, 8.9x faster than MPImAdd on 1, 4, and 16 instances. In the single machine case, `MPI-mAdd` achieves a cost of \$0.17 per analysis. `Spark-mAdd` costs \$0.06 to run on a single instance, which is a 2.8× improvement in cost. `Spark-mAdd` is still cheaper than `MPI-mAdd` by a factor of 2.2× when running on four nodes. `Spark-mAdd` is only more expensive than a single node of `MPI-mAdd` when running on 16 nodes, and even then is only 40% more expensive than the lowest cost `MPI-mAdd` run while providing a 8.9× performance improvement over the fastest `MPI-mAdd` run.

[9]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 [14]) 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.

[10]Available from `http://irsa.ipac.caltech.edu/applications/2MASS/IM/`.

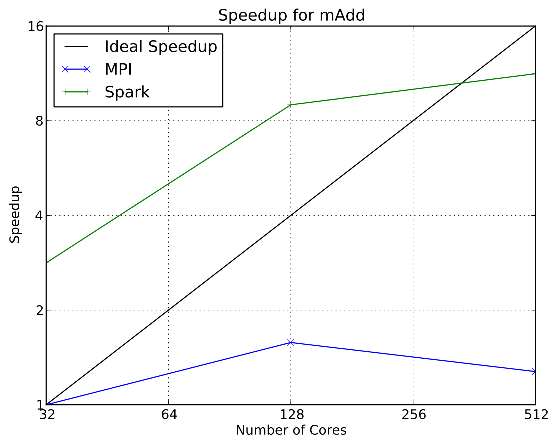

Figure 3: Speedup when running *mAdd* using MPI and Spark

The performance improvement is caused by multiple factors. We are able to reduce the amount of I/O performed, while also reducing contention in the I/O system and improving data locality. By denormalizing the metadata into our data schema, we are able to combine the metadata processing stage with the mAdd stage. This allows us to only load the input dataset a single time. Parquet also compresses the input and output data, which reduces the volume of I/O performed. Additionally, MPI is bound by contention when trying to write all output to a single file in a shared file system, while Parquet writes output files into HDFS in a contention free manner. Finally, Spark allows the computation to benefit from data locality, while MPI distributes the computation across the available resources without optimizing for data placement.

While the dataset used is a small dataset, larger datasets are commonplace. For example, the Large Synoptic Survey Telescope (LSST, [19]) has been used to collect terabyte sized datasets [31]. As future work, we plan to tackle these very large astronomy datasets using our framework.

6.3 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).

6.3.1 Compression

The Parquet columnar store [3] 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.[11] This is an improvement over our in-memory string representation which allocates a byte per base.

In Table 3, we look at the compression we achieve on the `NA12878` and `HG00096`[12] human genome sequencing samples. We compare against the GZIP compressed BAM [26] 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.

Table 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×

We achieve greater compression on the astronomy datasets. We compare against the legacy FITS [46] format in Table 4. We measure the aggregate compression of the image files provided as input to our system, and the compression of our pipeline output.

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. 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.

Table 4: Astronomy Data Compression

	Input Dataset	
Format	**Size**	**Compression**
FITS	1.5 GB	—
Parquet	0.55 GB	2.75×
	Output Dataset	
Format	**Size**	**Compression**
FITS	1.2 GB	—
Parquet	0.88 GB	1.35×

6.3.2 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 [33], 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*.

6.3.3 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 B.1), 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).

[11] 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.

[12] 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/`.

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 5. Projections are arranged in the columns of the table while predicates are assigned to rows.

Table 5: 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.

7. DISCUSSION AND FUTURE WORK

Similar to what we propose, the `Thunder` system was developed as a novel MapReduce-based system for processing terabytes of neuroscience imaging data [15]. `Thunder` performs a largely statistical workload, and the significant tasks in terms of execution time are clustering and regression. The system is constructed using Spark and Python and is designed to process datasets larger than 4 TB, and leverages significant functionality from the MLI/MLLib libraries [40]. Similar to our system, they are able to use Spark's in-memory caching to amortize load time across several pipeline stages. Additionally, they use Spark's filtering primitives to allow scientists to cut problems into subproblems. This is a common trend across scientific analyses, and is one of the reasons that we advocate for the use of a columnar store with efficient predicate pushdown. `Thunder` is an example pipeline that demonstrates the power of using horizontally scalable systems to enable novel scientific analyses.

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, [38]) and Georganas et al [17] 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 [18, 47]. 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.

8. 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, 3, 51]. 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.

APPENDIX

A. REFERENCES

[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. Parquet. `http://parquet.incubator.apache.org`.

[4] 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.

[5] 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.

[6] 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.

[7] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. PVFS: a parallel file system for linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference*, pages 28–28. USENIX Association, 2000.

[8] J. P. Cunningham. Analyzing neural data at huge scale. *Nature Methods*, 11(9):911–912, 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] G. England. 100,000 genomes project. `https://www.genomicsengland.co.uk/`.

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

[15] J. Freeman, N. Vladimirov, T. Kawashima, Y. Mu, N. J. Sofroniew, D. V. Bennett, J. Rosen, C.-T. Yang, L. L. Looger, and M. B. Ahrens. Mapping brain activity at scale with cluster computing. *Nature Methods*, 11(9):941–950, 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. 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.

[18] 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.

[19] Z. Ivezic, J. Tyson, E. Acosta, R. Allsman, S. Anderson, J. Andrew, R. Angel, T. Axelrod, J. Barr, A. Becker, et al. LSST: from science drivers to reference design and anticipated data products. *arXiv preprint arXiv:0805.2366*, 2008.

[20] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*, 4(2):73–87, 2009.

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

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] M. Moyers, E. Soroush, S. C. Wallace, S. Krughoff, J. Vanderplas, M. Balazinska, and A. Connolly. A demonstration of iterative parallel array processing in support of telescope image analysis. *Proceedings of the VLDB Endowment*, 6(12):1322–1325, 2013.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

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

[40] 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.

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

[42] 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.

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

[44] M. J. Turk, B. D. Smith, J. S. Oishi, S. Skory, S. W. Skillman, T. Abel, and M. L. Norman. yt: A multi-code analysis toolkit for astrophysical simulation data. *The Astrophysical Journal Supplement Series*, 192(1):9, 2011.

[45] 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.

[46] D. Wells, E. Greisen, and R. Harten. FITS—a flexible image transport system. *Astronomy and Astrophysics Supplement Series*, 44:363, 1981.

[47] 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.

[48] D. G. York, J. Adelman, J. E. Anderson Jr, S. F. Anderson, J. Annis, N. A. Bahcall, J. Bakken, R. Barkhouser, S. Bastian, E. Berman, et al. The sloan digital sky survey: Technical summary. *The Astronomical Journal*, 120(3):1579, 2000.

[49] 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.

[50] 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.

[51] 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.

[52] Z. Zhang, D. S. Katz, M. Wilde, J. M. Wozniak, and I. Foster. MTC Envelope: Defining the capability of large scale computers in the context of parallel scripting applications. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '13)*, pages 37–48. ACM, 2013.

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

B. SCHEMAS

Here, we present the schemas that we have used for these two systems. We make several simplifications to the schemas for clarity; specifically, we have grouped all of our fields into a simpler logical schema ordering, and we have also removed the Avro syntactic sugar used to ensure that all fields are nullable. We require that all fields are nullable, as this allows users to specify arbitrary projections. These schemas are implemented using Avro [2], and data is stored to disk via Parquet [3]. In-memory (de-)serialization is provided via a custom wrapper around Avro's serialization framework.

B.1 Genomics Schema

The schema used for storing genomics data is described below:

```
record Contig {
  string contigName;
  long contigLength;
  string contigMD5;
  string referenceURL;
  string assembly;
  string species;
}

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

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

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

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

  /** optional attributes */
  string attributes;

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

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

All of the metadata from the sequencing run and prior processing steps are packed into the record group metadata fields. The program information describes the processing lineage of the sample and is expected to be uniform across all records, thus it compresses extremely well. The record group information is not guaranteed to be uniform across all records, but there are a limited number number of record groups per sequencing dataset.[13] This metadata is string heavy, which makes proper deserialization from disk important. Although the information consumes less than 5% of space on disk, a poor deserializer implementation may replicate a string per field per record, which greatly increases the amount of memory allocated and the garbage collection (GC) load.

In our system, we have defined common projections and predicates to operate on these records. For example, tools (like the `flagstat` command) that perform quality control for sequenced data commonly only access the read status flags from the schema. We can reduce the amount of I/O we perform by projecting only these fields. Additionally, it is common to run predicates on the read position, or whether the read is mapped or not. We have several predicates that are optimized for these queries, and have devised a system that allows us to apply these predicates to legacy datasets that do not support direct predicate pushdown. In the legacy case, we only get the *functionality* of the predicate, not the performance improvement.

B.2 Astronomy Schema

We use the following schema for storing astronomy pixel values:

```
record PixelValue {
  /** pixel position */
  int xPos;
  int yPos;

  /** pixel value */
  float value;

  /** file metadata */
  int start;
  int end;
  int offset;
  int height;
}
```

This schema is derived from the legacy Flexible Image Transport System (FITS, [46]), which defines an interchange format for astronomy images. During the `mAdd` processing kernel described in §6.2, we access the file metadata from each pixel. In current systems, metadata access becomes a significant performance bottleneck as we are performing metadata access across thousands of files [52].

C. 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:

[13]The record group reflects the way the samples were sequenced; increasing the parallelism of the sequencing increases the number of record groups.

$$\hat{r} = \cup_{r_i \in \hat{R}} r_i \tag{2}$$

$$\hat{r} \cap r_i \neq \emptyset, \forall r_i \in \hat{R} \tag{3}$$

$$\hat{R} \subset R \tag{4}$$

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) is trivial to check: specifically, the genome is assembled out of reference contigs[14] 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 2 to find the maximal convex hulls that describe a genomic dataset.

Algorithm 2 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 $hulls$
```

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 3.

Algorithm 3 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 §5.1), we would use the maximal convex hull set to define partitioning for the join. Alternatively, for INDEL realignment (see §4.1), we use this set as an index for mapping reads directly to targets.

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