Massively Parallel Processing of Whole Genome Sequence Data: An In-Depth Performance Study

Abhishek Roy[⋆], Yanlei Diao[⋆♦], Uday Evani[▲], Avinash Abhyankar[▲]
Clinton Howarth[▲], Rémi Le Priol[♦*], Toby Bloom[▲]
[⋆]University of Massachusetts Amherst, USA [▲]New York Genome Center, USA [♦]École Polytechnique, France
[⋆]{aroy,yanlei}@cs.umass.edu
[▲]{usevani,avabhyankar,chowarth,tbloom}@nygenome.org [♦]{remi.le-priol@polytechnique.edu}

ABSTRACT

This paper presents a joint effort between a group of computer scientists and bioinformaticians to take an important step towards a general big data platform for genome analysis pipelines. The key goals of this study are to *develop a thorough understanding of the strengths and limitations of big data technology for genomic data analysis, and to identify the key questions that the research community could address* to realize the vision of personalized genomic medicine. Our platform, called GESALL, is based on the new "Wrapper Technology" that supports existing genomic data analysis programs in their native forms, without having to rewrite them. To do so, our system provides several layers of software, including a new Genome Data Parallel Toolkit (GDPT), which can be used to "wrap" existing data analysis programs. This platform offers a concrete context for evaluating big data technology for genomics: we report on super-linear speedup and sublinear speedup for various tasks, as well as the reasons why a parallel program could produce different results from those of a serial program. These results lead to key research questions that require a synergy between genomics scientists and computer scientists to find solutions.

1. INTRODUCTION

Recently, the development of high-throughput sequencing (or next-generation sequencing) has transformed genomics into a new paradigm of data-intensive computing [4]: sequencing instruments are now able to produce billions of short reads of a complete DNA sequence in a single run, raising the potential to answer biomedical questions with unprecedented resolution and speed. For instance, a human genome has approximately 3 billion bases and each base has a letter of 'A', 'C', 'G' or 'T'. The first whole human genome was completed in 13 years' time at a cost of $3.8 billion. Since then, the total amount of sequence data produced has doubled about every seven months [35], far exceeding Moore's Law. Today, for a human genome sample a sequencer can produce one billion short reads of 200-1000 bases each, totaling 0.5–1 TB of data, within three days and at a cost of a few thousand dollars. As a result, there is general excitement that technology advance will soon enable population-scale sequencing and *personalized genomic medicine* as part of standard medical care, as declared in the recent Precision Medicine Initiative by the White House.

However, the unprecedented growth of whole genome sequence data has led to major computation challenges:

(1) The flood of data needs to undergo complex processing to mine high-level biological information from vast sets of short reads while handling numerous errors inherent in the data. Genomic data analysis pipelines typically consist of a large number (at least 15-20) of steps, from the initial alignment for mapping short reads to the reference genome, to data cleaning for fixing various issues introduced by noisy input data, to variant calling for detecting small or large structure variants in the test genome.

(2) Most genomic analysis tools were developed based on a wide range of algorithms and heuristics to deal with the complexity of analysis and idiosyncrasies of data as a result of sequencing limitations. These algorithms, however, were not designed for efficiency on enormous data sets: most analysis algorithms can be run only on a single computer, e.g., `Bwa` [17], Bowtie [14], and Wham for alignment [18], PicardTools for data cleaning [28], NovoSort for sorting [24], and GATK for small variant calling [7]. Some of these algorithms are further limited to single-threaded execution.

(3) As a result, most genomic analysis pipelines take significant time to run. For instance, at the New York Genome Center (NYGC), which supports user pipelines from a dozen hospitals and medical institutes, current pipelines can take 3 to 23 days to complete for a human sample based on our initial profiling [8]. The variation of running time depends on the complexity of analysis. Some algorithms, such as Mutect [5] and Theta [25] for complex cancer analysis, alone can take days or weeks to complete on whole genome data. *Such long-running time lags far behind the desired turnaround time of 1-2 days for clinic use, e.g., to assist in timely diagnosis and treatment.*

To respond to the computing challenges in genomic data analysis, a number of recent projects have attempted to scale the processing across multiple machines. We make the following observations about these projects:

Isolated Implementations: Recently, the bioinformatics community has sought to parallelize existing analysis programs using MapReduce. These efforts often come with isolated

*Work performed during internship at NYGC.

SIGMOD'17, May 14 - 19, 2017, Chicago, IL, USA
 ISBN 978-1-4503-4197-4/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3035918.3064048

implementations of a single component in an overall large, complex infrastructure. For example, GATK asks the user to *manually* divide data into subsets and then runs parallel instances over these subsets [20, 7]. Crossbow [13] runs Hadoop jobs without porting the standard SAM/BAM format [31] for genomic data into HDFS. HadoopBAM supports the SAM/BAM format in HDFS but does not support logical partitioning to ensure correct execution of genomic analysis programs [22]. These systems lack an integrated solution to the storage, logical partitioning, and runtime support of large genome analysis pipelines.

Change of Existing Software: ADAM [19] provides new formats, APIs, and implementations for several genome processing stages based on Spark [37] and Parquet [27]. It requires complete reimplementation of genomic analysis methods using Scala and custom in-memory fault-tolerant data structures. This approach does not suit a genome center that supports many analysis pipelines for different species, population groups, and diseases. Since the owner of each pipeline can choose to use any analysis methods he trusts based on prior research, a genome center cannot afford to reimplement the numerous analysis programs in its pipelines, or afford the possible consequences of different variant (mutation) detection results due to reimplementation.

Big Data versus HPC Technology: A recent study [34] compares the Hadoop-based Crossbow implementation to a single node of conventional high-performance computing (HPC) resources for alignment and small variant calling. The study shows that Hadoop outperforms HPC by scaling better with increased numbers of computing resources. However, this study does not answer the question whether the Hadoop performance is optimal in terms of key metrics for parallel computing, such as speedup and utilization.

Objectives. Based on the above observations, we set two objectives in our study between database researchers and bioinformaticians at NYGC: (1) *The bioinformatics community still lacks a general big data platform for genome analysis pipelines*, which can support many analysis programs in their original implementations. We aim to provide such a big data platform with an integrated solution to storage, data partitioning, and runtime support, as well as high quality results of parallel analysis. (2) *Both bioinformatics and database communities still lack a thorough understanding of how well big data technology works for genomics and hence how we can improve it for this new application.* Our study aims to enable such understanding in terms of running time, speedup, resource efficiency, and accuracy.

More specifically, our contributions include the following:

1. A New Platform (Section 3): We design a general big data platform for genome data analysis, called GESALL, by leveraging massive data parallelism. Most notably, we depart from existing approaches by introducing the **Wrapper Technology** that supports existing genomic data analysis programs in their native forms, without changing the data model, data access behavior, algorithmic behavior, heuristics, etc., used in those programs. To do so, our wrapper technology provides several layers of software that can be used to "wrap" existing data analysis programs. Of particular interest is a new Genome Data Parallel Toolkit (GDPT) that suits the complex data access behaviors in genome analysis programs.

2. In-Depth Performance Analysis (Section 4): Our platform provides a concrete technical context for us to analyze the strengths and limitations of big data technology for genomics. We compare our parallel platform to the GATK best practices [10], a multi-threaded pipeline known as the "gold standard" for genomic data analysis. Known systems such as Galaxy [1], Genepattern [30], AWS-GATK [2] provide interface to run the same set of analysis programs as recommended by GATK best practices. Although some of these systems may offer a cluster for use and basic scheduling tools like scatter-gather, they leave the actual parallelization of a pipeline to the user.

Our performance study is centered on three questions:

(1) *When does big data technology offer linear or superlinear speedup?* Our analysis of the CPU-intensive `Bwa` alignment program shows that the rich process-thread hierarchy of Hadoop allows us to increase the degree of parallelism without incurring the overhead of the multi-threaded `Bwa` implementation. Given 24-threaded `Bwa` as baseline, which is a common configuration in existing genomic pipelines, our parallel platform can achieve super-linear speedup.

(2) *When does big data technology offer sublinear performance?* We also benchmarked a range of shuffling-intensive programs. On one hand, our parallel platform reduced the running time of these programs significantly; for instance, we enabled the first parallel algorithm for `Mark Duplicates` which reduced the running time from 14.5 hours to 1.5 hours. On the other hand, these shuffling-intensive tasks experienced sublinear speedup and low resource efficiency (<50%). The key overheads in the parallel implementation include (a) data shuffling due to different degrees of parallelism permitted in different steps; (b) data shuffling due to the change of logical partitioning criteria in different steps; and (c) data transformation between Hadoop and external programs. The first two factors are intrinsic properties of genomic pipelines; in particular, the change of degree of parallelism across steps (e.g., from 90 to 1 to 23) is less a concern in other domains such as relational processing.

(3) *Why does data parallelism produce different results from serial execution?* To the best of our knowledge, our study is the first to analyze why data parallelism produces different results from serial execution. Contrary to the common belief that genomic programs such as `Bwa` alignment are "embarrassingly parallel", a parallel program for `Bwa`, which is the first step in the genomic pipeline, already produces results different from the serial execution. The ultimate accuracy measure, as proposed by our bioinformatics team, is the impact on final variant calls, which is around 0.1% differences for our parallel pipeline. Our in-depth analysis further reveals that data partitioning in our parallel pipeline does not increase error rates or reduce correct calls. The differences are more likely from the non-determinism of genome algorithms for hard-to-analyze regions, for which both serial and parallel versions give low-quality results.

While the above results were obtained from a Hadoop platform, most of them reflect fundamental characteristics of genomic workloads. Hence, we expect them to carry value beyond the specific platform used.

3. Future Research Questions (Appendix C): The above results lead to a set of research questions that call for a synergy between bioinformaticians and computer scientists to improve big data technology for genomics. The questions include (1) automatic safe partitioning of genomic analysis programs, (2) a rigorous framework for keeping track of errors in a deep genomic pipeline, (3) data parallelism with

Term	Explanation
`(Short) read`	a sequence of 100-250 nucleotides (bases)
`Paired reads`	forward and reverse reads of a DNA fragment, usually with a known distance in between
`Read name`	the unique identifier of a pair of reads
`Mate`	the other read that pairs with a given read
`Base call`	letter 'A', 'C', 'G' or 'T' returned by a sequencer for each base, together with a quality score
`Variant call:`	a change (mutation) from the reference genome
a) `SNP`	single nucleotide polymorphism (1-base change)
b) `Indel`	small insertion/deletion (multi-base change)

Table 1: Basic terminology for genomic data analysis.

high resource efficiency, and (4) a pipeline optimizer that can best configure the execution plan of a deep pipeline to meet both user requirements on running time and a genome center's requirements on throughput or efficiency.

2. BACKGROUND

In this section, we provide background on genome data analysis pipelines. The commonly used genome terms are summarized in Table 1, while the detailed data format is deferred to the subsequent technical sections.

2.1 Scope of Our Genome Data Analysis

For simplicity, our discussion in this paper focuses on human genomes and *paired-end* (PE) sequencing, a common practice in next-generation sequencing. A human genome has approximately 3 billion bases and each base has a letter of 'A', 'C', 'G' or 'T'. Given a test genome, lab processes are used to break the DNA into small fragments. Then a PE sequencer "reads" each double-stranded DNA fragment from both sides, and aligns the forward and reverse reads as *paired reads*, as shown in Fig. 1(a), with the distance between them known for a specific sequencer. Then data analysis proceeds in three phases [21].

Primary Analysis. The first analysis phase is base calling by the sequencer, a machine-specific yet highly automated process today. For a given sample, a sequencer parses raw data to produces billions of paired reads, totaling 0.5-1TB of data. Each read consists of 100-250 bases, depending on the sequencer. Furthermore, each base has a specific call, 'A', 'C', 'G' or 'T', and a quality score capturing the likelihood that the base call is correct – sequencing errors may cause a base call to differ from the true base in the test genome. All of the reads are stored in the file format called FASTQ, where paired reads share the same *read name*.

Secondary Analysis. The next stage of analysis is the most resource intensive, including over a dozen steps for alignment, data cleaning, and variant calling. We built a pipeline based on GATK best practices [10]. Table 2 lists the subset of the algorithms that are used in this study.

Alignment: The first main task is to align the short reads against a reference genome, as shown in Fig. 1(b). Each aligned read is mapped to one or multiple positions on the reference genome together with mapping quality scores. Our pipeline uses the `Bwa` algorithm [16] because it allows a read to be mapped to multiple positions in the face of mapping ambiguity, and assigns a mapping quality score to each position. The alignment results are encoded in the SAM/BAM format [31]: The text-based SAM format includes a record for each mapping of a read, i.e., m records for a read that is mapped to m locations on the reference. SAM records

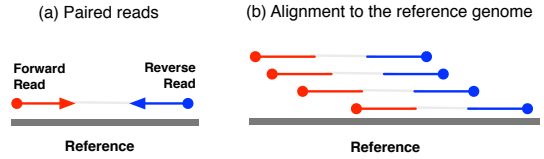


Figure 1: Paired-end sequencing and alignment.

are then converted to a binary, compressed indexed format called BAM, which will be detailed in Section 3.

Data cleaning: After alignment, the pipeline goes through a large number of steps to remove the effect of sequencing errors from subsequent analysis. Among them, the time-consuming steps include: `Fix Mate Info` (step 5 in Table 2) shares the alignment information and makes other information consistent between a pair of reads, which is needed due to the limitations of alignment software. `Mark Duplicates` (step 6) flags those paired reads mapped to exactly the same start and end positions as duplicates. These reads are unlikely to occur in a random sampling-based sequencing process, and often originate erroneously from DNA preparation methods. They will cause biases that skew variant calling and hence should be removed. We implemented the above steps using the PicardTools [28]. `Base Recalibrator` (steps 11-12): The sequencer returns a quality score for each base to represent the probability that the base was called incorrectly. The likelihood of an incorrect call is affected by a number of factors, such as the position of a base in the read (for example, bases at the ends of reads tend to be lower quality). The base recalibrator uses the alignment results, along with these other factors, to adjust the quality scores to better reflect the actual likelihoods of incorrect base calls. These steps are implemented using GATK [10].

Variant calling aims to detect the variants (mutations) from the reference genome based on all the reads aligned to a specific position. Our pipeline aims to include a large number of such algorithms to serve different clinical or research needs. Small variant calls include single nucleotide polymorphism (SNP) and small insertions/deletions (INDELs). Our pipeline uses the `Unified Genotyper` and `Haplotype Caller` [10] for these purposes. Large structure variants span thousands of bases or across chromosomes. We are currently testing GASV [33] and somatic mutation algorithms (e.g., [5, 25] for cancer analysis) for our pipeline.

Tertiary Analysis. The detected variants can then be loaded into databases, statistical tools, or visualization tools for annotation, interpretation, and exploratory analysis, referred to as tertiary analysis. Multiple samples may to be brought together, integrated with phenotype and other experimental data, and compared to existing databases of genomic variation. Genomic analysis for clinical purposes usually attempts to identify likely pathogenic mutation(s) that account for a specific phenotype. Unlike the previous phases that process *billions* of reads per sample, the data volume in this phase has been reduced to *millions* of mutations per sample, sometimes manageable on a single computer.

In this work, we focus on **secondary** genome analysis, which takes FASTQ data and produces variant calls. *This phase is the main "data crunching" part of genome data analysis and a necessary ETL (extract-transform-load) process to enable tertiary analysis.* This phase causes a significant processing delay for analyzing urgent samples in clinic use, or a substantial computing cost for research use. More broadly speaking, our secondary analysis results in calling genomic

Secondary Analysis: Alignment, Data Cleaning, and Variant Calling		Time (hrs)
1. `Bwa (mem)`	Aligns the reads to the positions on the reference genome	**26.3**
2. `Samtools Index`	Creates the compressed BAM file and its index	**14.3**
3. `Add Replace Groups`	Fixes the ReadGroup field of every read, adds information to header	**14.8**
4. `Clean Sam`	Fixes Cigar and mapping quality fields, removes reads that overlap two chromosomes	8.5
5. `Fix Mate Info`	Makes necessary information consistent between a pair of reads	**23.1**
6. `Mark Duplicates`	Flags duplicate reads based on the same position, orientation, and sequence	**14.5**
...	...	...
11. `Base Recalibrator`	Finds the empirical quality score for each covariate (each specific group of base calls)	**34.8**
12. `Print Reads`	Adjusts quality scores of reads based on covariates	**46.6**
13.v_1 `Unified Genotyper`	Calls both SNPs and small ($\leq$ 20 bases) insertion/deletion variants	**20.4**
13.v_2 `Haplotype Caller`	Like Unified Genotyper, but a newer version of the algorithm	**33.2**
...	...	

Table 2: A pipeline of algorithms based on GATK best practices, and single-server performance for a human genome.

variants in a single genome, and the existence of a variant in any one genome from a population or disease cohort is used in later analyses to assess the quality of variant calls in other genomes within the cohort. Thus, our work will be applicable to a much broader range of genomic analysis and the downstream application to clinical impact.

Regarding architectural choices, most practice for secondary analysis is based on GATK best practices exploiting only multi-core technology. A few recent projects [19, 29] scale out to clusters but require reimplementing analysis methods (which raise concerns to large genome centers). In contrast, tertiary analysis is subject to broader architectural choices including database systems such as SciDB [36] and Paradigm4 [26] (currently used at NYGC); distributed NoSQL systems such as GMQL-GeCo [12] built on Spark and Flink, and DeepBlue on MongoDB [3]. The extension of our work to tertiary analysis is left to future work.

2.2 Challenges and Requirements

To illustrate computation challenges in secondary analysis, we ran the pipeline shown in Table 2 for a human genome sample NA12878 [6] on a server with 12 Intel Xeon 2.40GHz cores, 64GB RAM, and 7200 RPM HDD. We turned on multithreading whenever the analysis program allowed so. The pipeline took about two weeks to finish with the running times of different steps shown in the last column of Table 2.

We aim to address such computation challenges using a parallel approach. To begin with, we summarize the requirements and constraints posed by NYGC.

Performance Goals. To assist in timely diagnosis and treatment, the desired turnaround time for processing a whole human genome is 1-2 days (including complex cancer analysis). While this indicates an objective of minimizing *running time*, NYGC is also concerned with *resource efficiency* or *throughput* (in number of Gigabases analyzed each day), because its compute farm is intensively shared among many analysis pipelines from hospitals and research institutes.

Parallelizing the pipelines is subject to a few constraints:

Any Analysis Methods. The platform for parallelizing the genome pipelines must permit any analysis methods that a user needs, without requiring rewriting of these methods.

Standard Data Formats. Until such time that new genome data formats are agreed upon by standards groups, deviation from existing standards such as SAM/BAM formats is unlikely to gain adoption in the bioinformatics community.

Assumption of In-Memory Processing. The size of individual genomes (0.5-1TB each) and the need to process a batch of genomes together in some analysis steps make in-memory processing often untenable, or at most as optimization of "local" parts of the computation. A general platform for genomic pipelines still needs to be designed with full capacity to process most data to and from disks.

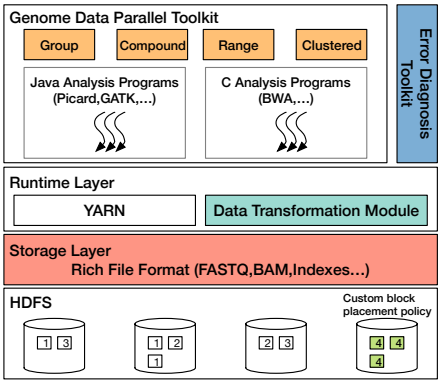


Figure 2: Gesall Architecture

3. BIG DATA PLATFORM FOR GENOMICS

In this section, we present our design of a big data platform, called Gesall, that offers a suite of new features to support correct and efficient execution of genomic pipelines based on data parallelism. Most notably, our approach departs from existing approaches by introducing the **Wrapper Technology** that aims to support existing genomic data analysis programs in their native forms, without changing the data model, data access behavior, algorithmic behavior, heuristics, etc., used in those programs. This technology also frees genomic programmers from the burden of rewriting tens to hundreds of genomic data analysis programs when porting these programs to a parallel platform.

To do so, our wrapper technology provides several layers of software that can be used to "wrap" existing data analysis programs. In this section, we provide an overview of these software layers, which sets the technical background for our performance study in the next section. In particular, we put an emphasis on our new Genome Data Parallel Toolkit (GDPT), which involves more complex hashing and range partitioning techniques than traditionally considered for relational queries due to the complex data access behaviors in genomic data analysis programs. As general system background, we leverage HDFS [11] for storage, YARN [11] for resource allocation, and the MapReduce (MR) runtime engine [11] for executing genomic pipelines.

3.1 Distributed Storage of Genomic Datasets

The lowest layer of software that Gesall provides is a distributed storage substrate for genomic data datasets encoded in the standard SAM/BAM format [31]. Our distributed storage substrate lies between HDFS and genomic analysis

programs. It is designed to ensure correct execution of genomic analysis programs on HDFS, because these programs were developed based on the assumption that they can access the whole genome data encoded in the BAM format from a local file system. Since we want to run these programs directly on subsets of genome data on different nodes, it is incorrect to let HDFS split a BAM file into physical blocks and distribute them to the nodes. This naive approach not only breaks the correct BAM format assumed in the analysis programs, but also violates assumptions made in the programs about the accessibility of all relevant data, causing incorrect results. We address these issues by adding two new features in the storage substrate.

1. Distributed Storage of BAM Files. Alignment results of all the reads are encoded first in a text-based SAM file format. The file starts with a header with metadata such as the alignment program, the reference sequence name, the sorting property, etc. Then it contains a series of SAM records. For I/O efficiency, a SAM file is always converted to a binary, compressed BAM format to write to disk. The BAM construction method takes a fixed number of bytes from the SAM file, converts the contained records to the binary format, compresses it using BGZF compression, and appends the resulting (variable-length) chunk to the existing BAM file.

Our goal is to partition a BAM file and store it using HDFS, and later present the local data on each node as a proper BAM file to the analysis programs. When a BAM file is uploaded from an external file system or written from an analysis program to HDFS, the file is split into fixed-sized HDFS blocks (default: 128MB), which are distributed and replicated to different nodes. Each HDFS block can contain multiple, variable-length BAM compressed chunks. In the splitting process, the last BAM chunk in a HDFS block may span across the boundary of a block. We provide a custom implementation of Hadoop's `RecordReader` class to read BAM chunks properly from HDFS blocks. Then to iterate over read records in BAM chunks, we provide a utility class that is called with a list of BAM chunks as input. The utility class will fetch the header from the first chunk of the BAM file, and provide an iterator, of the same class as in PicardTools, over the read records in compressed chunks. This allows us to have only one-line modification of single-node programs to switch the reading from local disk to HDFS.

2. Storage Support for Logical Partitions. The above solution enables correct access to BAM records stored in HDFS. However, it alone does not guarantee the correctness of running analysis programs on subsets of data on each node. For instance, the `Bwa` and `Mark Duplicates` algorithms assume to have both the reads of a pair in the input data. We call such partitioning criteria the "*logical partitioning*" of data. Our storage substrate adds the feature of logical partitions on top of HDFS. It can be used when a user uploads an external dataset with logical partitions into HDFS, or when a MapReduce job rearranges data according to a logical condition and writes such logical partitions back to HDFS. To support this feature, we implement a custom `BlockPlacementPolicy`, which assigns all blocks of a logical partition file to one data node.

3.2 Genome Data Parallel Toolkit

As stated above, logical partitioning of a large genomic dataset enables an analysis program to run independently on the logical partitions. Since most existing analysis programs were designed for running over a complete genomic dataset, we next analyze the data access patterns of common analysis programs and the logical partitioning schemes that we propose for these programs. We encode these logical partitioning schemes in a library, called *Genome Data Parallel Toolkit*, which can be used later to wrap an existing program that matches the specific data access pattern.

We first summarize the key attributes in SAM records that are used for partitioning in Fig. 3. Recall that there is a SAM record for each aligned position of a read. Within a record, the `QNAME` attribute gives the name of a read, which is the same for its mate in the pair. `POS` is the leftmost mapping position of the read on the reference genome. `PNEXT` refers to the mapping position of the mate of this read.

We next categorize the partitioning schemes as follows:

1. Group Partitioning The simplest data access pattern is to require data to be grouped by a logical condition. Examples include `Bwa` alignment and Fix Mate Info (partitioning by read name), and Base Recalibrator (partitioning by user-defined covariates). Below, we describe `Bwa` in detail.

Alignment: The `Bwa` algorithm requires unaligned reads to be grouped by the read name. The input FASTQ files of unaligned reads are already sorted by the read name, with one file for each of the forward and reverse reads of a pair. We first merge them to a single sorted file of read pairs. Then we split this file into a set of logical partitions and load them into our HDFS storage system, which ensures that read pairs in a logical partition are not split to different nodes when running `Bwa` in parallel. However, the result of our parallel program differs slightly from that of a single-node program due to the inherent uncertainty in genome data analysis, which we detail in Section 4.5.

2. Compound Group Partitioning refers to the case that an analysis program imposes two grouping conditions on a dataset. In general, it is not possible to partition data once while meeting both conditions. But if the two conditions are related, we may be able to analyze data partitions under both grouping conditions simultaneously. Such compound group partitioning is unique to the genomic data model where reads are produced as pairs: grouping by individual reads and grouping by read pairs are both required and exhibit correlated data access patterns.

Mark Duplicates: Recall from Section 2.1 that duplicates are a set of paired reads that are mapped to exactly the same start and end positions. Since they often originate erroneously from DNA preparation methods, they should be removed to avoid biases in variant calling.

First, the `Mark Duplicates` algorithm defines duplicates on a derived attribute, called the $5'$ unclipped end, which is unclipped start position of each read. Note that the alignment algorithm may "clip" the end of a read, where quality may be lower, to get a better alignment of the remaining bases. Mark Duplicates needs to use the unclipped start of each read to find true duplicates. For this work, it is sufficient to consider the $5'$ unclipped end as a derived attribute of a read record, which can be computed from the `POS` and `CIGAR` attributes in that record, as shown in Fig. 3.

Second, the algorithm detects duplicates in both *complete matching pairs* (both reads of a pair are successfully mapped to the reference genome), and in *partial matching pairs* (one read in a pair is unmapped), using different criteria.

Attribute	Type	Brief Description
QNAME	String	Read name
SEQ	String	Read sequence of size k
QUAL	String	Base call quality scores of size k
POS	Int	Leftmost mapping position
MAPQ	Int	Mapping quality
CIGAR	String	Mapping details including clipping
PNEXT	Int	Mapping position of the mate
TLEN	Int	Read pair length l
5′ unclipped end	Int	Computed from POS and CIGAR

Figure 3: Basic attributes (in blue) and derived attributes (in red) of a sam read record.

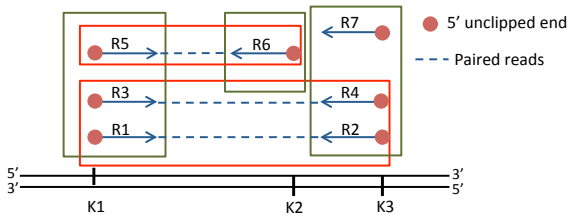


Figure 4: Mark Duplicates partitions enclosed in boxes. One of (R1, R2) and (R3, R4) pairs is marked as a duplicate. The partial matching, R7, is marked as a duplicate.

Criterion 1: For complete matching pairs, each read pair is marked by the 5′ unclipped ends of both the reads. Then to detect duplicates among different read pairs, we need to group reads by a compound key composed from *both* unclipped 5′ ends of the reads in the pair.

Criterion 2: To detect duplicates in partial matchings, only the 5′ end of the mapped read in a pair is compared to the 5′ ends of other reads, which may belong to *either complete or partial matchings*. Although the reads of complete matching are not marked as duplicates, they are required to detect reads from partial matching as duplicates.

Thus, reads in complete matching pairs require **two partitioning functions**: the first based on the 5′ ends of the paired reads, and the second based on the 5′ end of an individual read. Fig. 4 shows an example of 3 complete matching pairs and 1 partial matching. They require generating 5 partitions including 2 paired read partitions, marked by the red boxes, and 3 individual read partitions, marked by the green boxes. Between (R1, R2) and (R3, R4) pairs, the one with a lower quality score is marked as a duplicate. The partial matching, R7, is also marked as a duplicate because it coincides with reads R2 and R4 in complete matching pairs.

Parallel Algorithms. We propose a parallel implementation for `Mark Duplicates` from PicardTools, which is the first in the literature to the best of our knowledge. It uses a full round of MapReduce to realize two partitioning functions. In the map phase, the input data must be grouped by the read name so that the two reads in a pair are read together. The map function processes the reads in each pair, and generates two partitioning keys as described above. Encoding two partitioning functions leads to more data shuffled than the input data. However, to detect duplicates from the reads of partial matching pairs, it suffices to have only one such read from complete matching pairs. We use this intuition to add a map-side filter that emits only one read from complete matching pairs for each unclipped 5′ position. At each reducer, reads belonging to different keys are grouped together and sorted in order to call `Mark Duplicates`, which requires reads to be sorted by the mapped position. We refer to this implementation as $\texttt{MarkDup}^{\texttt{reg}}$ (regular).

As an optimization, we can precompute a bloom filter in a previous MapReduce round, which records the 5′ unclipped positions of all the reads in partial matching pairs. The '1' bit for a genome position tells that it is necessary to apply the second partitioning function to the complete matching pairs mapped to that position, and unnecessary otherwise. This method, referred to as $\texttt{MarkDup}^{\texttt{opt}}$ (optimized), has the potential to decrease the number of records shuffled.

3. Range Partitioning Many analysis programs also consider read pairs as intervals over the reference genome. The analysis work can be partitioned by dividing the reference genome into segments, which may be non-overlapping in the simple case or overlapping in more complex cases. Each partition consists of all the reads overlapping with a given genome segment. The reads that overlap with two partitions are replicated. While such range partitioning is reminiscent of temporal databases and other systems that model data as intervals, genomics presents another level of complexity regarding how segments are defined and how much overlap is required to ensure best quality of the output of a parallel program. Two examples are given below.

Unified Genotyper calls single-nucleotide polymorphism (SNP) and small insertion-deletion (Indel) variants. Bioinformaticians at NYGC suggest partitioning reads based on the chromosome to which they are mapped. Hence, we can use a (non-overlapping) range partitioning scheme where a range is a chromosome, permitting 23 partitions to run in parallel. However, quality control tests show that even chromosome-level partitioning gives slightly different results from single-node execution, demonstrating that most genome analysis programs are non-trivial to partition.

Haplotype Caller is a newer algorithm for small variant detection. To increase the degree of parallelism, it is possible to design a fine-grained partitioning scheme that leverages the knowledge of the internal data access pattern within a chromosome. Most notably, Haplotype Caller employs a greedy segmentation method during a sequential walk of the reference genome. Specifically, it walks across all the positions of the genome performing three operations: 1) compute a statistical measure over all the reads that overlap with the current position; 2) greedily define the current segment (called an active window) based on the trend of the statistical measures in the recent positions and the minimum and maximum length constraints of an active window; 3) detect mutations inside each active window. We call this access pattern *greedy sequential segmentation*. In particular, this second operation prevents us from simply partitioning the data by position. In our work, we have designed an overlapping partitioning scheme that can determine the appropriate overlap between two genome segments and bound the probability of errors produced by this scheme.

However, the top priority in parallel data processing is quality. Even coarse-grained partitioning based on chromosomes gives (slightly) different results from single-node execution. Further diagnosis is needed; only after we understand why differences occur, can more advanced algorithms be accepted into production genomic pipelines.

3.3 Runtime Data Transformation

Our system also offers runtime support for running analysis programs coded in C or Java in the Hadoop framework.

	Cluster A (Research)	Cluster B (Production)
Nodes	1 name node + 15 data nodes	1 name data + 4 data nodes
CPU	24 cores@2.66GHz	16 cores@2.4GHz with hyper-threading
Memory	64GB	256GB
Disk	1 x 3TB, 140MB/sec	6 x 1TB, 100MB/sec
Network	1Gbps	10 Gbps

Table 3: Two clusters with comparable total memory.

Round 1: Alignment		
Input files	15	4800
Avg. file size (MB)	38420.5	120.1
Wall clock time (s)	11699	14266
Round 3: MarkDuplicates		
Input files	30	510
Avg. file size (MB)	8139.6	478.8
Wall clock time (s)	13172	10548

Table 4: Running time with varied logical partitions.

We highlight the key data transformation issues below, but leave additional details to Appendix A.1.

We use Hadoop Streaming to transfer the data to/from analysis programs coded in C, such as the `Bwa` aligner. In our implementation of a map-only task for `Bwa`, two external programs are piped together for the map function – multi-threaded `Bwa`, and single-threaded `SamToBam`. Fig. 8 shows the dataflow through different programs and pipe buffers. Even if an analysis program is coded in Java, there is still a data format mismatch between Hadoop data objects, in the form of key-value pairs, and the input BAM format required by external programs. We presented an instance of this mismatch in Section 3.1 where HDFS blocks are transformed to a complete BAM file in the map phase. Similarly for the reduce phase, we provide utility classes that can transform a list of read records to an in-memory BAM file with the fetched header. Such transformation allows us to run single-node programs, with minimal changes, in the Hadoop framework.

3.4 Error Diagnosis Toolkit. Another feature of Gesall is an extensive error-diagnosis toolkit for parallel genomic pipelines, which keeps tracks of errors through the pipeline, taking into account various biological measures and the impact on the final pipeline output. We describe it in detail in the validation section (§4.5.2).

4. EXPERIMENTS & ANALYSIS

We next evaluate the strengths and limitations of Gesall for genomic data analysis in terms of running time, speedup, resource efficiency, and accuracy. In particular, we compare our parallel platform to the GATK best practices [10], a multi-threaded pipeline known as the "gold standard" for genomic data analysis.

4.1 Experiment setup

Our experiments were run on two clusters, a dedicated research cluster, referred as Cluster A, and a production cluster at NYGC, referred to Cluster B. They have comparable aggregate memory but otherwise different hardware characteristics, as listed in Table 3. Both clusters run Apache Hadoop 2.5.2. On both clusters, the input data to the pipeline was NA12878 whole genome sample with 64x coverage [6]. This sample has 1.24 billion read pairs in two FASTQ files. The two uncompressed FASTQ files are 282GB per file, and 220GB in total with compression.

We ran a pipeline with five MapReduce (MR) rounds, with the detailed diagrams shown in Appendix A.2.

Round 1 is map-only for `Bwa` alignment and `SamToBam`.
Round 2 runs `Add Replace Groups` and `Clean Sam` in the mapper, then shuffling, and `Fix Mate Info` in the reducer.
Round 3 runs data extraction in the mapper, then shuffling, and `Sort Sam` and `Mark Duplicates` in the reducer.
Round 4 sorts the dataset for variant calling. It runs data extraction in the mapper, then range partitioning, finally sorting and building the BAM file index in the reducer.
Round 5 runs `Haplotype Caller` on the partitioned, sorted, and indexed BAM files to detect variants.

We use four metrics to capture performance:
(1) **Wall clock time** taken by the program to finish.
(2) **Speedup** $= \frac{\text{time taken by the single node program}}{\text{time taken by the parallel implementation}}$, which considers state-of-the-art single node programs.
(3) **Resource efficiency** $= \frac{\text{Speedup}}{\text{number of cpu cores used}}$, capturing how effectively extra resources are used for a job.
(4) **Serial slot time** is the amount of time for which the tasks of a job are holding on to one or multiple cores. It is the sum of wall clock time for each task multiplied by the number of cores requested by each task.

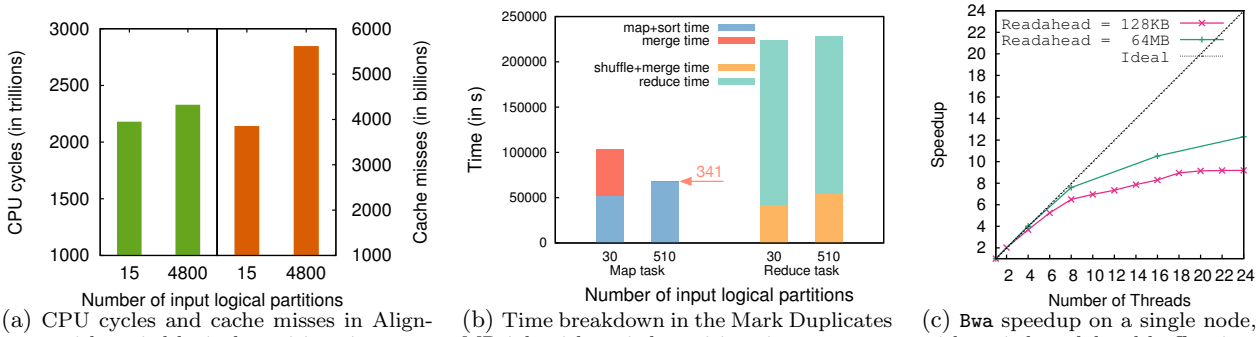
We use a large suite of tools to measure and analyze the performance. For each data node, we use the `sar` monitoring tool to report on CPU utilization, IO wait, swap space usage, network traffic, etc. For a single process, we use `perf stat` to get performance statistics like CPU cycles and cache misses. To capture the call stacks of a single process, we use `gdb` and `perf record`. For Hadoop jobs, we have written parsers to extract relevant data from log files generated at the end of each job. We use the MapReduce History API to generate the job progress plots (as shown in Fig. 7).

4.2 Exploring the Parameter Space

We begin by considering the first three MR rounds (jobs), while deferring the additional rounds to the later experiments. To run these jobs, we need to first determine the configuration of a fairly large parameter space, including (1) *granularity of scheduling*, defined to be the size of input logical partitions of each MR job; (2) *degree of parallelism*, defined to be the numbers of concurrent mappers and reducers per node in each job. We start by using the research cluster A with 15 nodes.

Granularity of scheduling. Many analysis steps have logical partitioning requirements on input data: both `Bwa` and `Mark Duplicates` require input data to be grouped by read name. The default Hadoop block size is 128MB, which can be used as the logical partition size. We next show that the optimal sizes of logical partitions for genomic analysis differ far from the default Hadoop configuration and require significant understanding of the workloads.

We first vary the input partition size to the map-only `Bwa` job. It was run on 15 data nodes with 1 map task of 6 threads per node. We use two input sizes: (1) Using 15 logical partitions of 38 GB each, we can finish alignment with one wave of map tasks on each node. (2) We also use 4800 logical partitions with 120 MB each (close to the default setting). As Table 4 shows, the run time is higher with the partition size = 120 MB. The reason is that the mappers for alignment need to each load the reference genome index into memory. Such loading is done by 15 mappers given 15 logical



(a) CPU cycles and cache misses in Alignment with varied logical partition sizes.

(b) Time breakdown in the Mark Duplicates MR job with varied partition sizes.

(c) `Bwa` speedup on a single node, with varied readahead buffer sizes

Figure 5: Performance of MR jobs with varied logical partition sizes, and limitations of the multi-threaded Bwa.

	MapReduce Round 3: `Mark Duplicates`^{opt}			MapReduce Round 3: `Mark Duplicates`^{reg}		
Number of data nodes	**Wall clock time (s)**	**Speedup**	**Resource efficiency**	**Wall clock time (s)**	**Speedup**	**Resource efficiency**
1 (Gold Standard)	94680	–	–	94680	–	–
5	10539	8.984	0.299	21967	4.310	0.144
10	5655	16.743	0.279	10846	8.729	0.145
15	(4065) 3724	(23.292) 25.424	(0.259) 0.282	7031	13.466	0.150

Table 5: Scale up to 15 nodes and 90 parallel tasks with reduced running time and similar resource efficiency.

partitions, but 4800 times given 4800 logical partitions. The overheads of repeatedly starting mappers and loading the reference genome in each mapper are shown by the CPU cycles and cache misses in Fig. 5(a). So large input partition sizes amortize per-mapper overheads better.

However, we observe a different trend for Mark Duplicates. To illustrate, consider an MR round running the `MarkDup`^{opt} algorithm on 5 data nodes with 6 concurrent map or reduce tasks per node. The map phase requires to group the input by read name to generate a compound key for shuffling data to reducers. Table 4 shows that the run time for 30 and 510 input logical partitions. Here, the larger partition size gives worse performance. The time breakdown in Figure 5(b) shows that the key difference between the two configurations is the time taken by the map side merge.

This is because Hadoop uses the sort-merge algorithm to sort all shuffled key-value pairs by the key, for which each mapper performs a local sort of its output. The sort buffer `mapreduce.task.io.sort.mb` is set to the maximum Hadoop value of 2GB. If the sort buffer gets filled, data is first sorted in-memory and then spilled to disk. When the map function finishes, a map-side merge phase is run to produce a single sorted, partitioned file for the reducers. With large partition sizes there is a significant overlap of the map-side merge phases of concurrent map tasks on the disk. This results in increased workload on disk and hence more time to complete the map-side merge task.

In summary, our system enables better performance by tuning the input logical partition size for each job. One should use large partition sizes when aiming to amortize per-partition overheads specific to genomic analysis, and medium sizes for better sort-merge performance in data shuffling.

Degree of Parallelism: We next consider the effect of scaling the degree of parallelism up to 15 nodes. We show the results of running the MR job for Mark Duplicates, for up to 15 data nodes. Recall that we have two implementations, `MarkDup`^{reg}and `MarkDup`^{opt}, as described in §3.2. We run both up to 15 nodes, with 6 concurrent map or reduce tasks per node. Since each mapper/reducer must be given 10GB of memory to hold its working set, 6 tasks are the most we can run on one node. As each node has only 1 disk, we defer the study of parallel disks to §4.5.

First consider `MarkDup`^{opt}. The number of records shuffled is 1.03 times the input records and the shuffled data in terms of bytes (with Snappy compression) is 375 GB. Table 5 shows the performance with different numbers of nodes: The job run time reduces with more nodes, but the resource efficiency decreases slightly with more nodes (in particular, see the efficiency in parentheses for 15 nodes). This is because when using more nodes, each node gets less intermediate shuffle data. By default, Hadoop overlaps the shuffle phase of reduce tasks with the map tasks. The parameter `mapreduce.job.reduce.slowstart.completedmaps` sets the fraction of the number of map tasks which should be completed before reduce tasks are scheduled. The default value of this parameter is 5%. As the amount of shuffled data is reduced with 15 nodes, these tasks end up occupying and wasting resources on each node while waiting for more map output. To improve resource efficiency, we configure Hadoop to start shuffle phase only after 80% of map tasks finish for 15 nodes. The new resource efficiency with 15 nodes increases from 0.259 to 0.282, enabling constant resource efficiency (although the number is low). Next consider `MarkDup`^{reg}, which presents a workload with large intermediate shuffle data: it shuffles 1.92 times the input records and 785 GB in bytes (with Snappy compression). Table 5 again shows constant resource efficiency (at a low number).

In summary, our system can scale up to 15 nodes and 90 concurrent tasks with reduced running time and similar resource efficiency. Hence, we use the maximum degree of parallelism permitted as the default setting. We did not consider more nodes because the real computing environment at NYGC requires us to consolidate parallel tasks on a relatively small number of nodes, as we detail in §4.5. We further investigate the overall low resource efficiency in §4.4.

4.3 Linear / Superlinear Speedup

We next seek to understand how well a Hadoop-based platform supports CPU-intensive jobs, for which we consider the map-only alignment job. To cope with the high compu-

	Single node	Cluster A with 15 data nodes				
	Wall clock time (s)	Processes per node	Wall clock time (s)	Speedup	Resource efficiency	Serial slot time (s)
Round 1: Bwa, SamToBam	99240	1 (24 threads)	7200	13.78	**0.92**	104106
	(24 threads)	6 (4 threads)	4007	24.77	**1.65**	336090
Round 2: AddRepl,CleanSam,FixMate	154228	6	3612	42.70	**0.47**	308985
Round 3: SortSam, MarkDuplicates^{opt}	94680	6	4065	23.29	**0.26**	293830

Table 6: Performance of the three MapReduce rounds on Cluster A, compared to the single-node performance.

tation needs, the `Bwa` aligner already comes with a multi-threaded implementation. As described in §3, we run alignment via Hadoop streaming between the Hadoop Framework and native `Bwa`, encoded in C, and `SamToBam`. This gives us freedom to explore the degree of parallelism by changing the number of nodes, number of mappers per node, and number of threads per mapper. Since Cluster A has 24 cores per node, we can run up to 24 threads per node (number of mappers per node × number of threads per mapper). We use 90 input partitions so all mappers finish in one wave.

Table 6 shows the metrics for the multi-threaded single node program and our parallel implementation (in the row named "Round 1"). We found that (1) resource efficiency is more when running 6 mappers with 4 threads each, than 1 mapper with 24 threads, per node; (2) resource efficiency is greater than 1 when we use 6 mappers.

The reason is that multi-threaded `Bwa` does not use resources efficiently when we increase the number of threads. The initial speedup of multi-threaded `Bwa` program is shown in Fig. 5(c) labeled with Readahead = 128KB. We did intensive profiling and found out that `Bwa` has a synchronization point in the form of file read and parse function. To solve this problem, we set the read ahead buffer size to 64MB. As the file is accessed in a sequential pattern, the Linux kernel is able to keep up with the program usage by prefetching 64MB of file at each time. The new speedup is labeled with Readahead = 64MB in Fig. 5(c). There are more bottlenecks in the program, which impede the linear speedup, e.g. the computation threads wait for all other threads to finish before issuing a common read and parse request. We did not spend more resources fixing these issues, because Hadoop provides a very flexible process-thread hierarchy for us to explore. This hierarchy allowed us to utilize the multi-process model, rather than only multi-threads within a process, to increase the degree of parallelism per node.

As can be seen, the rich process-thread hierarchy of Hadoop allows us to increase the degree of parallelism without incurring the overhead of the `Bwa` multi-threaded implementation. Given the 24-threaded `Bwa` as baseline, we can even achieve super-linear speedup as shown in Table 6. However, if we take 1-threaded `Bwa` as baseline, our speedup is sub-linear, 190.58 as opposed to the ideal case of 24 x 15 = 360. This is due to the overhead of data transformation between Hadoop and external C programs via Hadoop streaming.

4.4 Sublinear Performance

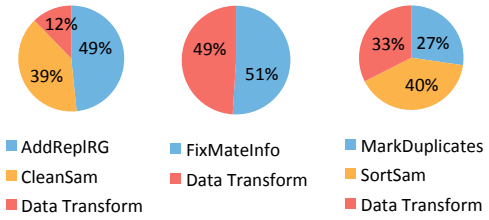
We next seek to understand how well a Hadoop-based platform works for shuffling-intensive jobs. We first consider Round 2 (data cleaning) and Round 3 (Mark Duplicates) jobs as described before, which both require shuffling data from mappers to reducers to rearrange data based on a new logical condition. Table 6 shows the performance of the two MR jobs. We observe poor resource efficiency, especially for Mark Duplicates. We examine the reasons below.

1. Data shuffling: Data shuffling from mappers to reducers incurs a huge overhead for genome data because there is no reduction of data. As noted in Table 6, the job running `MarkDup`^{opt} with bloom filters takes 1 hour 7 minutes, where 72 million more records than the input are shuffled due to the compound partitioning scheme. Without the bloom filter, `MarkDup`^{reg} shuffles 1.92 times the input data and the job run time increases to 1 hour 57 minutes. To measure the shuffling overhead, we moved Sort Sam and Mark Duplicates programs from reduce() to map(), and the execution time of this map-only round was only 43 minutes. The 24-min (74-min) difference for `MarkDup`^{opt} (`MarkDup`^{reg}) shows the cost of data shuffling to reducers.

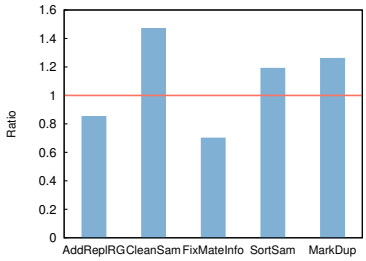
2. Data transformation between Hadoop and external programs: To run existing external programs without modifying their source code, we have added data transformation steps between the Java objects used in the Hadoop framework and the in-memory BAM files taken as input by the external programs. We incur the additional data transformation cost at each time when we provide input to an external program or read the output from an external program. Fig. 6(a) shows the time spent in external programs and the time in data transformation for the different map and reduce tasks of the pipeline. The time taken for data transformation varies between 12%-49%. We have to pay this *copy-and-convert* overhead as the external programs work with only SAM/BAM file formats.

3. Overhead of calling an external program repeatedly: We found that the time taken by repeated calling of an external program on partitions of data in the Hadoop platform can be significantly higher than the time taken by calling the program on the complete dataset, where Fig. 6(b) shows the ratio of these two time measures. A ratio of more than one means that external programs are taking more time to finish the job in the Hadoop platform. For example, the sum of running times of `Clean Sam` in our parallel implementation is 11 hours 3 minutes, while the running time of single-node `Clean Sam` on complete input data is 7 hours 33 minutes. In addition, we expected `Sort Sam` and `Mark Duplicates` to take more time as they are processing 72 million extra records (1.03 times the input data) compared to the single node version of the programs, but the increase in running time is greater than the corresponding increase in input data due to the overhead of repeated program calling. Overall, this overhead can be attributed to many factors, including input data being able to fit in memory (rather than spilling to disk), cache behavior, and fixed startup overheads.

4. Change in degree of parallelism: For some algorithms, there may not exist fine-grained partitioning scheme that generates the same results as those from a single node. For instance, bioinformaticians so far only accept a coarse-grained partitioning scheme based on chromosomes for Haplotype Caller. The program further requires reads in sorted order of the mapped position. To run it in parallel, we split the data into 23 partitions (for 23 chromosomes). MR Round



(a) Time breakdown for analysis and data transformation programs in the map and reduce tasks.



(b) Ratio of time taken in Hadoop versus single-node for external programs.

Figure 6: Profiling results to show overheads in shuffling-intensive jobs.

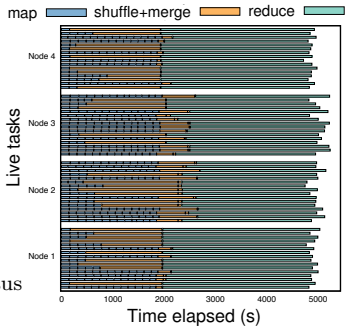


Figure 7: Task progress of MarkDup^{opt} with 1 disk in Cluster B at NYGC

Configuration	Wall Clock Time	Avg Map Time	Avg Shuffle + MergeTime	Avg Reduce Time
Align:Hadoop 4x4x4	4hrs, 57 mins, 16 sec	1 hrs, 10 mins, 40 sec		
Align:Hadoop 4x16x1	3hrs, 45 mins, 24 sec	3 hrs, 38 mins, 8 sec		
Align:in_house 4x16x1	3hrs, 44 mins, 38 sec	3 hrs, 16 mins, 18 sec		
MarkDup^{reg}:Hadoop 1 disk	4 hrs, 43 mins, 26 sec	5 mins, 57 sec	1hrs, 13 mins, 32 sec	2 hrs, 14 mins, 16 sec
MarkDup^{reg}:Hadoop 2 disks	3 hrs, 24 mins, 2 sec	4 mins	45 mins, 18 sec	1 hrs, 46 mins, 18 sec
MarkDup^{reg}:Hadoop 3 disks	3 hrs, 7 mins, 31 sec	3 mins, 51 sec	36 mins, 40 sec	1 hrs, 43 mins, 44 sec
MarkDup^{reg}:Hadoop 6 disks	2 hrs, 55 mins, 36 sec	3 mins, 54 sec	29 mins, 50 sec	1 hrs, 43 mins, 40 sec
MarkDup^{opt}:Hadoop 1 disk	1 hrs, 27 mins, 36 sec	2 mins, 25 sec	17 mins	45 mins, 57 sec
MarkDup^{opt}:Hadoop 6 disks	1 hrs, 22 mins, 40 sec	2 mins, 26 sec	15 mins, 34sec	43 mins, 58sec
MarkDup:in_house 1x1x1	14 hrs, 26 mins, 42 sec			

Table 7: Wall clock time and time breakdown in the production cluster (Cluster B) at NYGC.

4, which sorts the data by range-partitioning and creates an index on each partition, took 1 hour 1 minute. We have to pay this overhead to shuffle the data when the next analysis requires re-partitioning data. Then with 23 partitions as input, Round 5 which calls Haplotype Caller ran for 7 hours 14 minutes. During this run our resources were severely underutilized. Even though we had resources to run up to 90 parallel tasks, we could not use all of them with coarse-grained partitioning. This shows that such programs that dramatically change the degree of parallelism and data arrangement in the pipeline prevent us from fully utilizing resources.

4.5 Validation at New York Genome Center

We next present the performance and accuracy evaluation at NYGC, which has a compute farm of servers purchased specially for genomic analysis. To support this study, NYGC took 5 *production servers* from its compute farm to set up a dedicated cluster, named Cluster B, whose hardware specification is shown in Table 3. As noted before, the compute farm of NYGC is intensively shared among different analysis pipelines. We expect a small number of high-end servers to be the most likely computing environment for one or a batch of samples. This means that, instead of spreading an MR job to many nodes, we need to consolidate a large number of parallel tasks on a few servers.

Our goals here are to (1) validate our previous results on these production servers; (2) identify new performance characteristics on production servers; (3) assess the accuracy of our parallel pipeline using rigorous biological metrics.

4.5.1 *Performance Validation*

The performance of our Hadoop programs and an existing in-house solution at NYGC is summarized in Table 7.

1. Alignment (`Bwa` and `SamToBam`). We begin by discussing the change of Hadoop configuration at Cluster B. This cluster has much more memory per node than Cluster A, and hence can accommodate more parallel tasks per node. For our map-only program for alignment, each mapper needs 13GB memory so we can run 16 concurrent mappers per node, denoted as 4 (nodes)×16 (mappers/node)×1 (thread/mapper) in Table 7. (We turn off hyper-threading because it is an issue orthogonal to our study here.) Since the `Bwa` aligner allows multi-threading, for comparison we also run 4 mappers with 4 threads each per node, denoted as 4×4×4. We also run the in-house parallel alignment program at NYGC with the same numbers of parallel tasks.

Our main results from Table 7 are: (1) 16 mappers per node work better than 4 mappers per node, even if the total number of threads is the same. As reported above, running independent mappers (processes) overcome the limitations of the multi-threaded `Bwa`. (2) The in-house solution at NYGC also achieves the best performance using our suggested configuration of 16 independent tasks per node and shows similar performance. Even with the overhead of passing data between Hadoop and C programs, our implementation offers comparable performance to the in-house solution for alignment, the most parallelizable step in the pipeline.

2. Mark Duplicates. We again use a full MR job as before. Cluster B has sufficient memory to run 16 concurrent map or reduce tasks on each node, with 13GB memory allocated to each task. Recall that our two implementations for Mark Duplicates: `MarkDup`^{reg} and `MarkDup`^{opt} produce intermediate shuffle data of sizes 785 GB and 375 GB respectively (with Snappy compression of map output). We use the two implementations to represent workloads of different intermediate data sizes for shuffling. Since Cluster B requires us to consolidate a large number (16) of tasks on each server, we observe new performance characteristics regarding the relation between shuffling data sizes, the number of disks, and the number of reducers.

Due to space constraints, we summarize our results as follows: (1) When we run a full MR job for genomic data, there is a large amount of data shuffled and merged. Disk becomes the bottleneck when we run 16 reducers per node. Therefore, there must be a correct ratio between shuffled

data and disks, with our observation to be 1 disk per 100 GB of data shuffled and as many reducers as possible. (2) The in-house solution can only run this step in a single thread on a single node, with the running time about 14.5 hours, while our parallel program completes in less than 1.5 hours. Details can be found in Appendix B.1.

4.5.2 Accuracy Validation

Besides performance, we must ensure that parallelizing the analysis programs does not reduce the quality of resulting data. With the bioinformatics team at NYGC, we developed an extensive error diagnosis toolkit involving many biological metrics. We then performed a thorough analysis of the different immediate and final results of our parallel pipeline against those of a serial pipeline.

Error Diagnosis Toolkit. We denote a serial pipeline with k programs as $P = \{O_1, O_2, \ldots, O_k\}$, and the corresponding parallel pipeline as $\bar{P} = \{\bar{O}_1, \bar{O}_2, \ldots, \bar{O}_k\}$. We use R_i to refer the output of the pipeline P up to the step O_i, and $\bar{R}_i$ to refer to the output of $\bar{P}$ up to $\bar{O}_i$. Since we generally do not have the ground truth for the correct output of each step, our error diagnosis focuses on the difference between R_i and $\bar{R}_i$, for $i = 1, \ldots, k$. In particular, we define:

- $\Phi_i^+ = R_i \cap \bar{R}_i$, called the **concordant result set**;
- $\Phi_i^- = R_i \cup \bar{R}_i - R_i \cap \bar{R}_i$, the **discordant result set**;
- $|\Phi_i^-|$, the **discordant count** (**D_count**).

For all the discordant counts, we consider the reads having the quality score greater than zero.

Further discussion with bioinformaticians revealed that the final output produced at the end of our pipeline, the variant calls, are the most important measure of accuracy. As long as the final variants remain same through parallelization, the immediate discordant results do not have any *impact* on the detection results or any further analysis. To reflect this intuition, we define another measure that captures the impact of a parallel pipeline $\bar{P}_i$ (up to step i) on the final variant calls. Formally, we build a hybrid pipeline, $\tilde{P} = \{\bar{O}_1, \ldots, \bar{O}_i, O_{i+1}, \ldots, O_k\}$, which runs parallel pipeline $\bar{P}_i$ (up to step i) and then serial pipeline from step $i+1$ to the final step k. Denote its output as $\tilde{R}_k^i$. Then define:

- $\Psi(\bar{P}_i) = R_k \cup \tilde{R}_k^i - R_k \cap \tilde{R}_k^i$, the **discordant variant set** caused by $\bar{P}_i$;
- $|\Psi(\bar{P}_i)|$, the **discordant impact** (**D_impact**) of $\bar{P}_i$.

Furthermore, not all discordant results (immediate or final) are equal. We observed that many genome analysis programs aim to filter out the records with low quality scores. So we want to weigh D_count and D_impact based on the quality scores. Our weighting function $\mathcal{F}$ is a generalized *logistic function* which allows flexibility to handle a range of quality scores. For alignment, our function weights the aligned reads by the mapping quality (MAPQ), an attribute in each SAM read record. MAPQ is a log-scaled probability that the read is misaligned, and takes the value between 0 and 60. Our logistic weighting function assigns the weight 0 to reads with MAPQ ≤ 30 and weight 1 to those with MAPQ ≥ 55 (based on the filtering behaviors of genome analysis programs), and other weights between 0 and 1 for $30 <$ MAPQ < 55 following the curve of a logistic function. Our system can accept other customized functions as well. We also designed a similar weighting function for variant quality scores. We call the resulting measures the **weighted D_count / D_impact**.

We have written MapReduce programs to compute all the D_count and D_impact measures and their weighted versions for our parallel pipeline.

Major Results: In this experiment, we took three parallel pipeline fragments: $\bar{P}_1$ runs up to parallel `Bwa`; $\bar{P}_2$ runs up to parallel Mark Duplicates; $\bar{P}_3$ runs up to parallel Haplotype Caller. Table 8 shows the measures.

The discordant count for `Bwa` is 71,185 (out of 2,504,895,008 reads). The weighted D_count metric, which weighs the discordant alignment results by the quality scores, is 0.0002%. To measure D_impact of $\bar{P}_1$, we pass the parallel `Bwa` output through the serial versions of Mark Duplicates and Haplotype Caller. The weighted D_impact on the variants is only 0.0837%.

For parallel Mark Duplicates, the D_count metric appears to be higher. This can be attributed to our calculation which sums the differences in number of duplicates for each pair. Simply, the Mark Duplicates algorithm can mark read pairs as duplicates at random when pairs are of equal quality, which inflates our metrics. We also found that the difference between $\bar{P}_2$ and P_2 output in terms of number of duplicates is only 259. To measure D_impact of $\bar{P}_2$, we pass the parallel Mark Duplicates output through the serial Haplotype Caller. We obtain the same D_impact for $\bar{P}_2$ as $\bar{P}_1$. To calculate the D_impact value for parallel `Bwa`, we run serial Mark Duplicates, followed by serial Haplotype Caller. To calculate the same metric for parallel Mark Duplicates, we run serial Haplotype Caller. Since the output of parallel Mark Duplicates is same as that of serial Mark Duplicates when run on same input data, we do not see any differences in the results of serial Haplotype Caller, which runs after Mark Duplicates.

The parallel Haplotype Caller row in Table 8 shows discordant counts for the complete parallel pipeline. The absolute D_count value for parallel Haplotype Caller is 8,710 variants, which is slightly higher than the D_impact value (or, size of discordant variant set) of 8,489 for Mark Duplicates. This is because D_impact values are computed by running a single node Haplotype Caller as the last program while the D_count values for Haplotype Caller are computed by running its parallel version on partitioned data.

Further diagnosis of discordant results. We conducted a detailed error diagnosis study of `Bwa` and its output records to understand why we get discordant results. Due to space limitations, we only present a summary of our analysis. The reader can refer to Appendix B.2 for the detailed analysis. Our analysis showed that most of the differences between the serial and parallel implementations of `Bwa` lie in the reads with low quality scores. In addition, our analysis provided an interpretation of the differences from the biological viewpoint: a large proportion of discordant reads are gathered around "hard-to-map" regions, known to be anomalous and highly repetitive genome fragments. Further we found that `Bwa` embeds data-dependent heuristics, such as making alignment decisions based on statistics from a local batch of reads, which can change with data partitions. `Bwa` also makes random choices when different alignment options have the same quality score.

Further diagnosis of discordant impact. We further analyze in detail the discordant impact (D_impact) on variant calling caused by introduction of a parallel version of `Bwa` and Mark Duplicates. To calculate the impact, we used two pipelines: a hybrid pipeline by following the parallel

	D_count	Weighted D_count	Weighted D_count (%)	D_impact	Weighted D_impact	Weighted D_impact (%)
`Bwa`	71,185	36,427	0.0002	8,489	4,202	**0.0837**
`Mark Duplicates`	1,618,863	176,974	0.0547	8,489	4,202	**0.0837**
`Haplotype Caller`	8,710	4,225	**0.0841**	-	-	-

Table 8: Discordant counts (D_count) and discordant impact (D_impact) for the parallel pipeline up to different steps.

pipeline with a serial Haplotype Caller, and a serial pipeline with Haplotype Caller as the last step. We compare the output of the two pipelines in detail in Tables 9 and 10, where the discordant impact is 8,489 variants while we have a total of 5,017,886 concordant variants.

We next seek to analyze the properties of the discordant variant set and the biological interpretation of discordance. We report on several alignment and variant quality metrics used in bioinformatics analysis. While we refer the reader to Appendix B.3 for the definitions of these metrics and their values, we summarize the main observations as follows:

The evaluation of different quality metrics indicates that the variant calls found **only** by one of the pipelines have the following properties: (1) They are a small fraction of the total number of variant calls. This fraction is close to 0.1%. (2) They are of low quality, relative to the concordant variant set. (3) They are of low quality as shown by six quality metrics in Tables 9 and 10. These results indicate that the concordant variant set comprises the high-quality, likely-correct variants, and that the serial and hybrid pipelines differ in only low-confidence calls. We also compared the variant output of both pipelines with gold standard dataset for NA12878 sample provided by Genome in a Bottle Consortium [38] and did not observe any significant difference, as reported in Appendix B.3. Thus, data partitioning in our parallel pipeline does not increase error rates or reduce correct calls. The differences are more likely from the non-determinism of genome algorithms for hard-to-analyze regions and poor quality data.

5. RELATED WORK

Parallel processing for secondary analysis. GATK [20, 7] can parallelize genomic analysis within a single multi-threaded process, or asks the user to *manually* divide data and then run multiple GATK instances. BioPig [23] and SeqPig [32] offer low-level libraries for I/O operations, SAM record access, and simple counting tasks, but not complete programs such as Alignment and Mark Duplicates. Hadoop-BAM [22] can store binary, compressed BAM data in HDFS, but without advanced features such as logical partitioning for executing a pipeline of analysis programs. Seal [29] supports three specific processing steps, by significantly modifying or reimplementing the code, and does not support logical partitioning schemes. ADAM [19] provides a set of formats, APIs, and processing step implementations, but requires reimplementing analysis methods using Scala and RDD's (in-memory fault-tolerant data structures in Spark). Crossbow [13, 34] implements a Hadoop-based parallel pipeline for Bowtie alignment and SOAPsnp for SNP calling. It is the only parallel system that does not require changing existing analysis programs. But it does not support standard SAM/BAM formats in HDFS or logical partitioning schemes for correctness. Moreover, its supported algorithms do not overlap with those suggested by NYGC, making it hard for us to compare empirically. Finally, our recent study profiled genomic pipelines on a single node to motivate the design of a parallel platform [8], but did not present design details or evaluation results.

Cloud computing for secondary analysis. In prior work, a team at the Broad Institute ported into the Amazon cloud the Picard primary pipeline [28], the GATK unified genotyper, and GenomeStrip structure variant caller. This effort, however, only utilizes multi-core technology or manual partitioning of data across multiple nodes, while our work investigates the use of Hadoop-based big data technology to run full genomic pipelines across a cluster.

Tertiary analysis. A variety of tools can be used for tertiary analysis operations on the detected variants. The GenBase [36] benchmark study compares the performance of storage systems including column stores and array databases on typical tertiary analysis tasks. Systems like GMQL-GeCo [12] provide an integrated, high-level query language for performing algebraic operations over genomic regions and their metadata. Online data servers (e.g. DeepBlue [3]) allow users to search and retrieve additional genomic and epigenomic data. FireCloud [9] is a cloud-based platform that allows users to perform cancer genome analysis. It includes pre-loaded data, workspaces, and tools tools like MuTect to detect cancer mutations.

6. SUMMARY AND FUTURE WORK

We presented the design and evaluation of a data parallel platform for genome data analysis Our main results include:

1) *Superlinear speedup*: The rich process-thread hierarchy of Hadoop allows us to increase the degree of parallelism for CPU-intensive `Bwa` alignment without the overheads of the multi-threaded `Bwa` implementation, achieving super-linear speedup over a common configuration of 24-threaded `Bwa`.

2) *Sublinear performance*: While our parallel platform reduced the running time of shuffling-intensive steps significantly, we observed sublinear speedup and limited resource efficiency (<50%) due to the overheads associated with different degrees of parallelism permitted in different steps, different logical partitioning criteria across steps, and data transformation between Hadoop and external programs.

3) *Accuracy of parallel programs*: Starting from `Bwa` alignment, the parallel pipeline produces different results from a serial pipeline. It is due to the nondeterministic behaviors of `Bwa` for low quality mappings and hard-to-map genome regions. The ultimate accuracy measure is the impact on final variant calls, which is around 0.1% differences for our parallel pipeline. Finally, data partitioning in our parallel pipeline does not increase error rates or reduce correct calls. The differences are more likely from the non-determinism of genome algorithms for hard-to-analyze regions, for which both serial and parallel versions give low-quality results.

Our results lead to a number of areas for future exploration, which are summarized in Appendix C.

Acknowledgments: We would like to thank Prashant Shenoy for his feedback. This work was supported in part by the National Science Foundation under the grant DBI-1356486, and the IDEX chair from Université Paris-Saclay.

7. REFERENCES

[1] E. Afgan, D. Baker, et al. The galaxy platform for accessible, reproducible and collaborative biomedical analyses. *Nucleic Acids Research*, 44(W1):W3–W10, 2016.

[2] E. Afgan, B. Chapman, and J. Taylor. Cloudman as a platform for tool, data, and analysis distribution. *BMC bioinformatics*, 13(1):1, 2012.

[3] F. Albrecht et al. Deepblue epigenomic data server: programmatic data retrieval and analysis of epigenome region sets. *Nucleic Acids Research*, 44(W1):W581, 2016.

[4] M. Baker. Next-generation sequencing: adjusting to data overload. *Nature Method*, 7(7):495–499, 2010.

[5] K. Cibulskis, M. S. Lawrence, et al. Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples. *Nature Biotechnology*, 31(3):213–219, 2013.

[6] . G. P. Consortium. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, Oct. 2010.

[7] M. A. DePristo, E. Banks, et al. A framework for variation discovery and genotyping using next-generation dna sequencing data. *Nat Genet*, 43(5):491–498, 2011.

[8] Y. Diao, A. Roy, and T. Bloom. Building highly-optimized, low-latency pipelines for genomic data analysis. In *CIDR*, 2015.

[9] Firecloud by broad institute. https://software.broadinstitute.org/firecloud.

[10] Best practice variant detection with gatk. http://http://www.broadinstitute.org/gatk/.

[11] Hadoop: Open-source implementation of mapreduce. http://hadoop.apache.org.

[12] A. Kaitoua, P. Pinoli, et al Framework for supporting genomic operations. *IEEE Transactions on Computers*, 66(3):443–457, March 2017.

[13] B. Langmead, M. Schatz, et al. Searching for SNPs with cloud computing. *Genome Biology*, 10(11):R134+, 2009.

[14] B. Langmead, C. Trapnell, et al. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3), 2009. R25.

[15] B. Li, E. Mazur, et al. Scalla: A platform for scalable one-pass analytics using mapreduce. *ACM Trans. Database Syst.*, 37(4):27:1–27:43, Dec. 2012.

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

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

[18] Y. Li, A. Terrell, and J. M. Patel. Wham: a high-throughput sequence alignment method. In *SIGMOD*, 445–456, 2011.

[19] M. Massie, F. Nothaft, et al. Adam: Genomics formats and processing patterns for cloud scale computing. Technical Report UCB/EECS-2013-207, University of California, Berkeley, Dec 2013.

[20] A. McKenna, M. Hanna, et al. The genome analysis toolkit: A mapreduce framework for analyzing next-generation dna sequencing data. *Genome Research*, 20(9):1297–1303, 2010.

[21] S. Moorthie, A. Hall, and C. F. Wright. Informatics and clinical genome sequencing: opening the black box. *Genet Med*, 15(3):165–171, 03 2013.

[22] M. Niemenmaa, A. Kallio, et al. Hadoop-BAM: directly manipulating next generation sequencing data in the cloud. *Bioinformatics*, 28(6):876–877, Mar. 2012.

[23] H. Nordberg, K. Bhatia, et al. Biopig: A hadoop-based analytic toolkit for large-scale sequence data. *Bioinformatics*, 29(23):3014–9, December 2013.

[24] Custom designed multi-threaded sort/merge tools for bam files. http://www.novocraft.com/products/novosort/.

[25] L. Oesper, A. Mahmoody, and B. Raphael. Theta: Inferring intra-tumor heterogeneity from high-throughput dna sequencing data. *Genome Biology*, 14(7):R80, 2013.

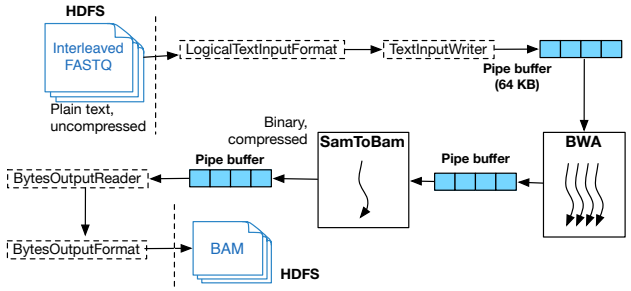


Figure 8: A map-only task for Alignment via Hadoop Streaming, allowing custom data transformation.

[26] A commercial dbms for scalable scientific data management. http://www.paradigm4.com/.

[27] Parquet: a columnar storage format for the hadoop ecosystem. http://parquet.incubator.apache.org.

[28] Picard tools: Java-based command-line utilities for manipulating sam files. http://picard.sourceforge.net/.

[29] L. Pireddu, S. Leo, and G. Zanetti. Mapreducing a genomic sequencing workflow. In *MapReduce '11*, 67–74, 2011.

[30] M. Reich, T. Liefeld, Jet al. Genepattern 2.0. *Nature genetics*, 38(5):500–501, 2006.

[31] Sam: a generic format for storing large nucleotide sequence alignments. http://samtools.sourceforge.net/.

[32] A. Schumacher, L. Pireddu, et al. Seqpig: simple and scalable scripting for large sequencing data sets in hadoop. *Bioinformatics*, 30(1):119–20, 2014.

[33] S. S. Sindi, S. Onal, et al. An integrative probabilistic model for identification of structural variation in sequence data. *Genome Biology*, 13(3), 2012.

[34] A. Siretskiy, T. Sundqvist, et al. A quantitative assessment of the hadoop framework for analyzing massively parallel dna sequencing data. *GigaScience*, 4(26), June 2015.

[35] Z. D. Stephens, S. Y. Lee, et al. Big data: Astronomical or genomical? *PLoS Biology*, 13(7):1117–1123, 2015.

[36] R. Taft, M. Vartak, et al. Genbase: A complex analytics genomics benchmark. In *SIGMOD*, 2014.

[37] M. Zaharia, M. Chowdhury, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI'12, 2012.

[38] J. M. Zook, D. Catoe, et al. Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Scientific data*, 3, 2016.

APPENDIX

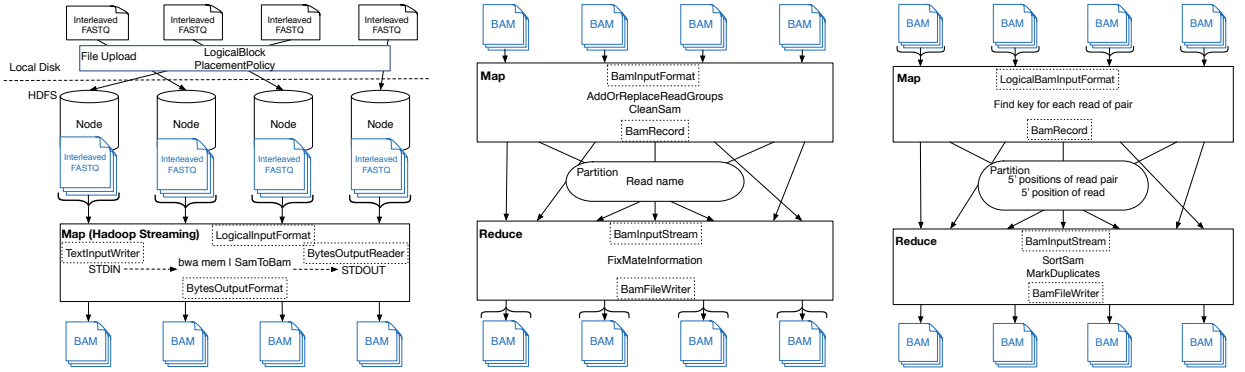
A. TECHNICAL DETAILS

A.1 Runtime Data Transformation

We perform multiple transformations which allow us to run single node programs coded in different programming languages, with minimal changes, in the Hadoop framework. We use Hadoop Streaming framework as the interface between Hadoop and analysis programs coded in C, such as `Bwa` and `SamToBam`, as shown in Fig. 8.

A.2 Building MapReduce Rounds for a Pipeline

Given a long pipeline of analysis programs, we arrange them into a series of MapReduce jobs as follows: as soon as the partitioning scheme of the next analysis program differs from (or, not compatible with) that of the previous program, we start a new round of MapReduce. If the map phase of the new job (e.g., `Mark Duplicates`) can benefit from the partitioning scheme of the previous step (e.g., data arranged by the read name in `Fix Mate Info`), we use logical partitions to feed the output of the previous step to the mappers of



(a) Alignment map-only round with Hadoop Streaming

(b) Picard Tools cleaning steps MapReduce round

(c) Mark Duplicates MapReduce round

Figure 9: Illustration of the first three MapReduce rounds of the workflow

the new job. Otherwise, the mappers simply read physical blocks. The shuffling phase of the new job is used to rearrange data according to the new partitioning requirement of the next analysis program. Based on partitioning requirements and order of steps, we designed the MapReduce (MR) rounds for the pipeline. The first three MR rounds are shown in Figure 9.

B. MORE EXPERIMENTAL RESULTS

B.1 Shuffling-Intensive Tasks at NYGC

We observe new performance characteristics regarding shuffling intensive tasks on the production servers at NYGC.

Relation between shuffling data sizes and disks: We first consider the workload of running $\texttt{MarkDup}^{\texttt{reg}}$. Profiling results show that the CPU utilization fluctuates widely due to the fact that one disk is maxed out, as shown in Fig. 10(a). This is because 1.96 times the input records needs to be shuffled from mappers to reducers, and Hadoop uses a sort-merge algorithm to sort all the key-value pairs by the key. As a reducer receives small files of sorted key-value pairs, it uses a disk-based merge to bring all data into a sorted run. Hence the temporary disk used for the merge process is maxed out. In this case, each temporary disk is handling approximately 200 GB of intermediate shuffled data. Therefore, we increase the number of disks for shuffle and merge from 1 disk to 6 disks, with the performance shown in Table 7. The 6 disk configuration gives the best performance for $\texttt{MarkDup}^{\texttt{reg}}$, and fixes the CPU and disk utilization issues as shown in Fig. 10(b).

Then, we run $\texttt{MarkDup}^{\texttt{opt}}$. Using 1 disk per node, we do not see the single disk maxing out (Fig. 10(c)). Here, each disk is handling the merge of approximately 100 GB of shuffled data, and is able to sustain the load of disk-based merge. This behavior can be explained using the the multipass merge model developed [15], which shows that the number of bytes read and written during the reduce side merge operation depends on the square of intermediate data handled per disk. This explains why the running time is so sensitive to the amount of data shuffled and merged. Empirically, we observe that one disk can sustain up to 100 GB of shuffled and merged data. Fig. 7 further shows that with 1 disk, the progress of reducers is already quite even. When we further increase to 6 disks per node, the reduce progress is very even with no stragglers.

Impact of reducers on shuffling: We also test the impact of changing the number of reducers. Both $\texttt{MarkDup}^{\texttt{reg}}$ with 2 disks per node and $\texttt{MarkDup}^{\texttt{opt}}$ with 1 disk per node handle approximately 100 GB of shuffled data per disk. From Table 7, we observe that it takes less time to shuffle and merge data in $\texttt{MarkDup}^{\texttt{opt}}$. In $\texttt{MarkDup}^{\texttt{opt}}$, the shuffle buffers of 16 reducers are used in the multi-pass merge algorithm, whereas in $\texttt{MarkDup}^{\texttt{reg}}$, shuffle buffers of only 8 reducers are available per disk. Since the number of bytes read and written during the reduce side merge operation is inversely proportional to the number of reducers per disk [15], it is beneficial to set the maximum number of reducers as allowed by the physical constraints (CPU, memory) of each node.

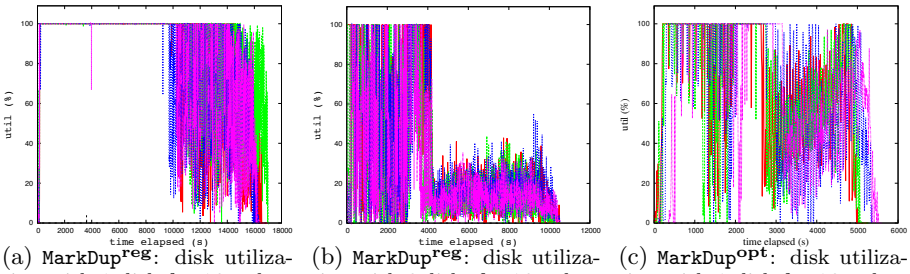
B.2 Error Diagnosis for Bwa Alignment

In this section we address the differences in results arising between `Bwa` run on single machine and multiple machines. In particular, we make the following observations:

1. *Majority of disagreeing reads have low mapping quality.* We plotted the distribution of the mapping quality of the two primary alignments of each read, one from the serial execution and the other from the parallel execution, under the condition that these two alignments differ. Figure 11(b) shows the mapping quality distribution where x-axis is the mapping quality in the single node output and y-axis is the mapping quality in the output of parallel version.

2. *A large proportion of disagreeing reads are gathered around "hard-to-map" regions.* First, on each chromosome we know the locations of centromeres. These regions are made of repetitive DNA and lack complete information in the reference genome. As a result, these regions and their neighborhoods are hard to map . Second, each chromosome also has a number of blacklisted regions due to their low mappability or other reasons known to biologists. In Figure 11(a) we plot the coverage of disagreeing pairs on chromosomes 1, 2, 3 and 4 in logarithmic scale. In Figure 11(a), the red stripes are centromeres and black stripes are blacklisted regions (obtained from ENCODE data at UCSC). We observe that a large proportion of disagreeing reads fall in these sensitive regions, as observed in the spikes.

Regarding the effect on downstream analysis, note that many algorithms consider only the reads with good mapping quality score (>30). In addition, some variant detection algorithms ignore the blacklisted regions or the output variant list is filtered to ignore such regions in a post-processing step.

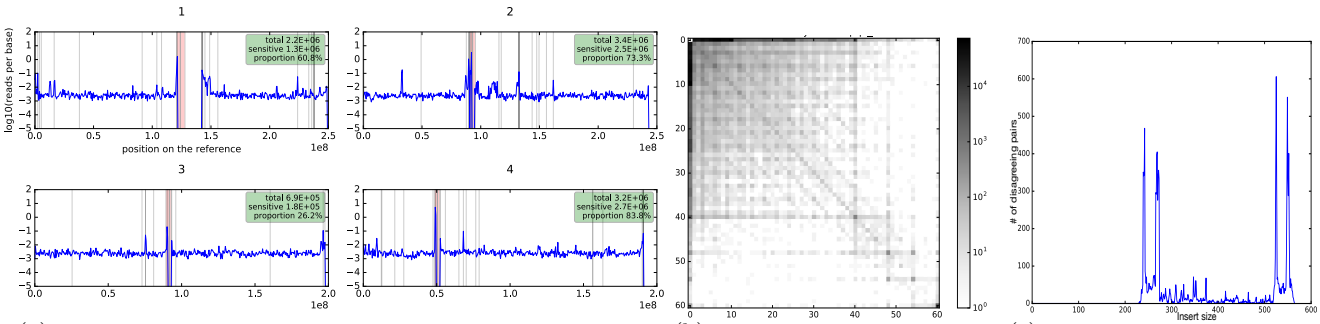


(a) `MarkDup`$^{\texttt{reg}}$: disk utilization with 1 disk for 16 reducers per node.

(b) `MarkDup`$^{\texttt{reg}}$: disk utilization with 6 disks for 16 reducers per node.

(c) `MarkDup`$^{\texttt{opt}}$: disk utilization with 1 disk for 16 reducers per node.

Figure 10: Profiling results of the experiments in the production cluster (Cluster B) at NYGC.



(a) Coverage of disagreeing pairs on chromosomes 1, 2, 3 and 4.

(b) Distribution of the mapping quality of the disagreeing pairs.

(c) Number of disagreeing pairs as a function of the insert size.

Figure 11: Error diagnosis study of `Bwa`.

If we apply these two filters, then the differences reduce to only 0.025% of total read pairs.

3. *Some artifacts of the `Bwa` program also cause instability of parallel results.* `Bwa` includes several implementation choices that cause different results under data partitioning: a) *The insert size distribution changes with each batch.* An optimization in the `Bwa` program computes the distribution of the *insert size* (distance between a read pair) from a batch of reads These batch statistics are then used to score the list of alignments for each read of the pair and select the best pair. The best pair is determined by a pair score. The pair score is a step function and can quickly change value when the probability of observing that pair distance is away from mean of insert size distribution is less than a fixed threshold. We plot the number of disagreeing pairs in Figure 11(c) when both reads of the pair are properly aligned (and applying the two widely used filters as described above) against insert size. We observe that the number of disagreeing pairs are higher on the edges of the insert size distribution. b) *Random choice among equal pair scoring score.* When different pairs of alignment have same alignment score then `Bwa` randomly selects a pair of alignment – this is especially common for alignment against repetitive genome regions. Thus, the batching behavior and the inherent randomness in `Bwa` implementation also prevents parallel version from reproducing exact same results.

B.3 Discordant Impact Study

In this section we analyze the discordant impact (D_impact) on variant calling caused by introduction of parallel version of `Bwa` and Mark Duplicates. To calculate the impact, we construct two pipelines - a hybrid pipeline by following the parallel pipeline with a serial Haplotype Caller and a serial pipeline with Haplotype Caller as the last step. For comparing the outputs of the two pipelines, we use *Intersection* to label the set of variants which are common to both pipelines, *Hybrid* to label the set of variants found only by the hybrid pipeline, and *Serial* to label the set of variants found only by the serial pipeline.

We evaluated the following metrics, used standardly in bioinformatics analysis, for the comparison study -
(1) **Mapping Quality (MQ):** MQ is the root mean square mapping quality of all the reads at the variant site.
(2) **Read Depth (DP):** Number of reads at variant site.
(3) **Fisher's Strand (FS):** Strand bias estimated using Fisher's Exact Test. Strand bias indicates that more reads come from one strand of DNA vs. the other, and can impact coverage assumptions.
(4) **Allele Balance (AB):** Estimates whether the data supporting a variant call fits allelic ratio expectations. For a heterozygous call, the value should be close to 0.5 whereas for a variant homozygous call should be to close to 1.0. Divergence from the expected ratio may indicate bias for one allele. AB = {# ALT reads}/{# REF + ALT reads}
(5) **Transition/Transversion Ratio (Ti/Tv):** Transitions are changes between C's and T's or A's and G's. Transversions are changes between A's and T's or C's and G's. We expect Ti/Tv ratio to be ∼2 in high quality variant calls.
(6) **Het/Hom Ratio:** The average ratio of heterozygous to homozygous variant calls across each sample.

Tables 9 and 10 show the metrics for the variants called in both pipelines or unique to a single pipeline.

The comparison of variant output of both pipelines with gold standard dataset for NA12878 sample provided by Genome in a Bottle Consortium [38] revealed no significant difference. We obtained the following precision and sensitivity values respectively (1) 99.2816% and 97.055% for serial pipeline, and (2) 99.2817% and 97.056% for hybrid pipeline.

	Intesection	Hybrid	Serial
Total	5,017,886 (99.831%)	4,287 (0.085%)	4,196 (0.083%)
Ti/Tv	1.94	0.97	0.94
SNP Het/Hom	1.8	12.55	12.51

Table 9: Variant call summary.

Quality Metric	Mean			Median		
	Intersection	Hybrid	Serial	Intersection	Hybrid	Serial
Mapping Quality (MQ)	57.97	45.26	45.06	60.00	47.31	46.49
Fisher Strand (FS)	3.98	14.74	17.45	0.85	5.36	6.25
Allele Balance (AB)	0.66	0.28	0.28	0.54	0.15	0.15
Read Depth (DP)	71.84	272.23	268.22	67.00	95.00	104.50

Table 10: Mean and median of variant quality metrics.

C. FUTURE RESEARCH DIRECTIONS

Our results thus far, in both performance and accuracy, lead to a number of areas for future exploration, some of which will require collaboration between genomics scientists and computer scientists to find solutions.

1. Automatic safe partitioning. We discovered in this study that the data partitioning method varies with data access behaviors and data dependencies in each genomic analysis program, and there is a large set of such behaviors as more algorithms are designed in bioinformatics. Finding a partitioning scheme that retains accurate results and best improves performance is a time-consuming manual process. To make our platform widely usable will require automating the partitioning process, and is a critical piece of our future work. This automation process will likely require "hints" or tags on each algorithmic step from genomics scientists, to describe the internal data dependencies that disrupt partitioning – a systematic way to do so for various data access behaviors is an interesting research topic.

2. A rigorous framework for error diagnosis in a deep pipeline. Keeping track of errors in a deep genomic analysis pipeline is extremely complex, labor-intensive at present, while it is also vitally critical when the variants finally reported by the pipeline are mutations in human genomes and such mutations will be used in diagnosis and treatment. We are particularly concerned with the differences introduced by data parallelism, that is, the differences of a parallel program from the results of a serial program, or the differences that vary with different numbers of partitions created. At the moment, there is no rigorous framework for tracking and reasoning such differences. Provenance from the database literature is a helpful concept. However, we need tuple-level provenance to diagnose how different results occur with partitioning. Tuple-level provenance is hard to collect because it requires changing the code of each analysis program substantially (e.g., in each object creation line of the code), let alone the time cost of doing so for billions of reads per genome. Interesting research directions may include (1) provenance collection at the granularity of mini-batches when such mini-batches are used as the data access pattern in genomic analysis, (2) error diagnosis for data partitioning based on such provenance, and (3) error evolution through a chain of steps.

3. Data parallelism with high resource efficiency. Our study observed that data parallelism reduced the running time of individual analysis steps, but the resource efficiency for shuffling-intensive steps can be lower than 50%. It is a critical issue for genome centers like NYGC which have a fixed budget (being non-profit organizations) and have to support many hospitals and research institutes. Often, the compute farm is intensively shared among numerous pipelines and has a long queue of jobs waiting to be submitted. In this case, wasting resources while achieving shorter running time is not desirable. More specifically, the wasted resources are due to the overheads of data parallelism, including shuffling data when the analysis steps change the partitioning criteria or degree of parallelism, and the overheads of data transformation between Hadoop and external programs. A research challenge posed here is how to achieve shorter running time through data parallelism while maintaining high resource efficiency. The key is to design new techniques that reduce the overheads, including more efficient algorithms for data shuffling, statistics collected from the earlier pipeline to reduce data needed in shuffling, or materialized views that suit different data partitioning schemes and can be reused through the pipeline.

4. A Pipeline Optimizer: Our study also demonstrates that large genomic workloads require a pipeline optimizer. (1) Even for the best studied objective of running time, the optimal configuration of our parallel execution varies with the analysis program, involves a large parameter size (the input partition size for mappers, number of concurrent tasks per nodes, number of disks used, when to start reducers, etc.), and requires capturing new cost components in genomic workloads (overhead of building an reference index per partition, disk-based sort-merge for shuffled data, etc.). The performance measures from our current tests help form a basis for the design of an optimizer that can automatically choose the optimal configuration for each analysis step. (2) The optimizer also faces the challenge of multi-objective optimization. We observed a tradeoff between running time and resource efficiency (throughput), which is expected to persist even if we reduce overheads of data shuffling and transformation. Given the heterogeneous mix of research and clinic processing pipelines at NYGC, it is an important study to determine how to optimize for running time for urgent samples while at the same time, optimize for overall throughput sustained by the center.