High-Performance Genomic Analysis Framework with In-Memory Computing

Xueqi Li, Guangming Tan, Bingchen Wang, Ninghui Sun
*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
†University of Chinese Academy of Sciences
{lixueqi,tgm,wangbinchen,snh}@ncic.ac.cn

Abstract

In this paper, we propose an in-memory computing framework (called GPF) that provides a set of genomic formats, APIs and a fast genomic engine for large-scale genomic data processing. Our GPF comprises two main components: (1) scalable genomic data formats and API. (2) an advanced execution engine that supports efficient compression of genomic data and eliminates redundancies in the execution engine of our GPF. We further present both system and algorithm-specific implementations for users to build genomic analysis pipeline without any acquaintance of Spark parallel programming. To test the performance of GPF, we built a WGS pipeline on top of our GPF as a test case. Our experimental data indicate that GPF completes Whole-Genome-Sequencing (WGS) analysis of 146.9G bases Human Platinum Genome in running time of 24 minutes, with over 50% parallel efficiency when used on 2048 CPU cores. Together, our GPF framework provides a fast and general engine for large-scale genomic data processing which supports in-memory computing.

CCS Concepts • **Applied computing → Computational genomics**; • **Software and its engineering →** *Distributed systems organizing principles*;

Keywords High-Performance Computing, Genomic Analysis Framework, In-memory Computing

ACM Reference Format:
Xueqi Li, Guangming Tan, Bingchen Wang, Ninghui Sun. 2018. High-Performance Genomic Analysis Framework with In-Memory Computing. In *PPoPP '18: Principles and Practice of Parallel Programming, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3178487.3178511

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178511

1 Introduction

Precision medicine is a global initiative aimed at using a patient's genomic information to identify and treat cancer. For example, precision medicine allows identification of somatic gene variants that act as drivers for new mutations in cancer tissue [6], and allows monitoring of gene expression changes during cancer progression or response to therapy [25]. Progress in precision medicine is driven by rapidly dropping sequencing costs and the ability to obtain valuable information about the entire human genetic code using next-generation sequencing (NGS) technology [21]. Currently, genomic data are acquired at high quality and high-throughput with the cost of 1,000 dollar per genome via HiSeqX [7], it is estimated that in the near future it should be possible to sequence entire genome at a cost of less than 100 using NovaSeq technology [8]. Due to an exponential increase in genomic data volume, the bottleneck is now changing from sequencing cost to computational power and storage requirement [11]. The growing genomic data is critically dependent on large-scale computing infrastructure. However, existing bioinformatics tools run in parallel on a single computer but are not designed to scale to clusters. Scaling-out genomic computing system faces two crucial challenges:

- *The expected surge of human genomic data will result in serious I/O problems during data analysis.* As introduced in the next section, genomic analysis consists of a series of steps, all of which are combined as a pipeline. Every two consecutive steps are connected by intermediate data, which are usually stored as disk files. For example, in the whole-genome sequencing method, read mapping consumes read samples in FASTQ format and produces aligned data in SAM/BAM format. The subsequent steps include sorting, indexing, duplicate marking and variant calling. These applications manipulate the SAM/BAM files, which are often much larger than the input data size. In fact, this type of data manipulation involves massive I/O operations. In a cluster machine connected with InfiniBand network, we ran a WGS pipeline to process 100Gb+ data while scaling the number of samples from 1 to 30. Table 1 shows that I/O time occupies more than 60% of the total running time in the case of processing multiple samples on larger-scale machines in both Lustre and

Table 1. Timing results for scaling from 1 to 30 samples on both Lustre and NFS file systems

	I/O Percent	CPU Percent
1 sample 96 cores Lustre	29%	71%
1 sample 96 cores NFS	25%	75%
30 samples 480 cores Lustre	60% ↑	40%
30 samples 480 cores NFS	74% ↑	26%

NFS file systems. This bottleneck indicates that performance will be greatly reduced when scaling to larger systems if the I/O problem is not resolved.

- *Current de facto genomic data formats and processing pipelines were not designed to support in-memory computing.* Recently, rewritting programs using in-memory computing model has been proposed as a potential solution to I/O bottleneck in many big data applications [27]. Previous studies focused on optimizing individual genomic applications via parallelism on single multicore machines [28] or multi-node scale-out computing systems [3, 10, 12, 14]. On the one hand, many of these parallelization approaches rely upon on frameworks like Hadoop MapReduce [5]. However, these frameworks lack abstraction for leveraging distributed memory, rendering them inefficient for pipelines that reuse intermediate results across different steps. On the other hand, studies that parallelize genomics data analysis applications using in-memory computing model showed an advantage over on Amazon cluster system [18, 23]. However, it only provides an incomplete framework used to improve several single modules instead of the entire pipeline.

Here, we propose a new genomic programming framework (GPF) that represents a high-performance, scalable, in-memory framework and supports in-memory genomic data format. The GPF has an advanced execution engine and a set of APIs, and complete processing stage implementations that support in-memory computing for large-scale genomic processing. More specifically, we make the following contributions:

- We develop a genomic programming framework (GPF) that can be used to build personalized genomic analytics pipelines. Genomic data format of GPF allows to use in-memory computing for standard data structures used for genomic analysis. Also, GPF abstracts the whole genomic analysis pipeline into three phases: **`Aligner`**, **`Cleaner`**, and **`Caller`**. It also provides in-memory computing programming interfaces.
- We build a fast and advanced execution engine to run genomic processing pipeline efficiently across large-scale cluster. The execution engine eliminates redundancy operations in the pipeline and stores large genomic object in serialized form.
- We evaluate our framework through building a WGS pipeline on top of GPF and demonstrate that GPF achieve parallel efficiencies of more than 50% on a 2048-cores cluster system. We find that system I/O is not the performance factor in terms of GPF. The fact indicates that scale-out parallel system can be built for large-scale genomic applications.

GPF is freely available: *https://github.com/fhyxz/GPF.git*. This paper is organized as follows: Section 2 provides background information on genomic analytics pipelines. Section 3 explains the APIs of the GPF, and Section 4 describes the execution engine of GPF. Section 5 evaluates our solution on a 2048 cores compute cluster and analyzes the most important factors to the performance of our GPF. Finally, we discuss related work in Section 6 before concluding in Section 7.

2 Background

Genomic analysis is the process during which sequencing data are processed into biological or clinical knowledge. For example, NGS-based cancer sequencing methods enable researchers to detect rare somatic variants (e.g., single-nucleotide variants (SNVs), insertion-deletion (indels), structural variants (SVs), and copy number variants (CNVs) to a specific tumor sample.

2.1 WGS Pipeline

Whole-genome sequencing using NGS technology is the most comprehensive method for analyzing genome. WGS processing consists of a number of steps, namely: read alignment application, sorting, and variant calling. Through investigating the WGS pipelines, we abstract the main computational applications into three phases as shown in Figure 1: (i) align reads to a reference genome (**`Aligner`**), (ii) manipulating the read alignments by sorting/indexing and eliminating duplicates (**`Cleaner`**) and (iii) various variants detection (**`Caller`**). The programming interfaces of in-memory computing framework are derived from the abstraction at this level.

- **`Aligner`**: This processing stage is referred to as short read mapping, which determines the location in the reference genome to which each read maps. The aligner employs a Burrows-Wheeler transform (BWT) algorithm [15] to index genome sequences reads (FASTQ files) to be aligned by reads. It executes the BWT-based mapping tool (bwa-0.7.12) [16], and then generates raw alignment records in SAM format [17].
- **`Cleaner`**: In order to improve accuracy of alignment and variant calling, most pipelines deploy an optional

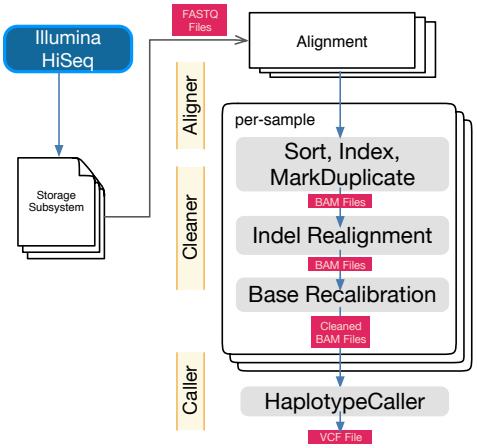

Figure 1. WGS analysis pipeline.

intermediate processing stage called data cleaning. The pipeline uses programs from either Picard (Picard-1.119) [2] or SAMtools package (Samtools-1.3) [17], including `Mark Duplicate`, `Indel Realignment` and `BaseRecalibration`. For example, `Mark Duplicate` marks reads with identical position and orientation, since there exist duplicate reads are created during sequencing whenever the number of sample molecules is too low.

- **`Caller`**: Once the sequencing data have been pre-processed as described above, the preceding steps are usually followed by variant calling using statistical methods. Common variant calling tools is: `HaplotypeCaller` in GATK [20].

3 The Genomic Programming Model

In this section we first define two basic concepts – `Process` and `Resource` (Sec 3.1), which are the abstract objects to be instantiated in a specific analysis algorithm. We then introduce the programming interfaces of GPF (Sec 3.2), and lastly provide an example for how to construct a genomic analytics pipeline using GPF APIs.

3.1 Process and Resource

To describe procedure and data dependency of genomic analysis pipelines, our genomic programming framework (GPF) uses two terms – `Process` and `Resource`. `Process` defines an execution instance which is involved in data input, data processing, and data output. `Resource` is the abstraction of data which are referred to as number, string, RDD and other specified objects.

Figure 2 depicts the state machine of both `Process` and `Resource`. A `Resource` changes its state between defined and undefined. The former state denotes that the `Resources` content has been filled and the latter is empty. A `Process` can be scheduled to execute only if all of its dependent `Resources` are defined. Importantly, `Process` goes through three states – *Blocked, Ready, and Running*. The key idea behind the *Ready* state is to perform a robust dependency analysis, so that redundant shuffle operations can be efficiently eliminated (See Section 4).

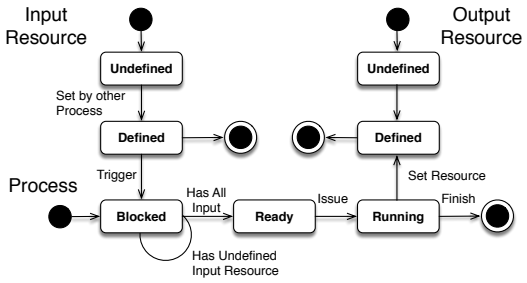

Figure 2. The state transition of `Process` and `Resource`.

3.2 Data Format and API

Our GPF consists of three genomic data formats for storing different original reads from the sequencer (FASTQ format), aligned-read after **`Aligner`** (SAM/BAM format), and the mutation site information (VCF format). These data formats are compatible with the most commonly used genomic formats, SAM/BAM and VCF. We introduce a set of interfaces that leverage Spark primitives to perform parallel operations so that users are able to build a genomic analysis pipeline without first learning Spark parallel programming. Users only need to define instances of both `Process` and `Resource` according to the sequential analysis algorithm. In this part, we provide definitions for the data representations and brief introduction to the APIs.

```
// Set up environment for Process and Resource
val sc = new SparkContext(conf)
val pipeline = Pipeline("myPipeline",sc)

// Load pair-end FASTQ to RDD
val fastqPath1 = "…/1.fastq"
val fastqPath2 = "…/2.fastq"

val fastqPairRdd = FileLoader.loadFastqPairToRdd(sc, fastqPath1, fastqPath2)
val fastqPairBundle = FASTQPairBundle.defined("fastqPair", fastqPairRdd)

// Add Aligner Process into the Pipeline
val alignedSAMBundle = SAMBundle.undefined("alignedSam", new SamHeaderInfo.unsortedHeader())
val mappingProcess = BwaMemProcess.pairEnd("MyBwaMapping", reference, fastqPairBundle, alignedSAMBundle)
pipeline.addProcess(mappingProcess)

// Add Cleaner Process into the Pipeline
val dedupedSamBundle = SAMBundle.undefined("dedupedSam", new SamHeaderInfo.unsortedHeader())
markDuplicateProcess = MarkDuplicateProcess("MyMarkDuplicate", alignedSAMBundle, dedupedSamBundle)
pipeline.addProcess(markDuplicateProcess)

// Add Caller Process into the Pipeline
val vcfBundle = VCFBundle.undefined("ResultVCF", VcfHeaderInfo.newHeader(refContigInfo, List()))
val rodMapCall = Map(RODNames.DBSNP -> dbsnp)
val useGVCF = True
val haplotypeCallerProcess = HaplotypeCallerProcess("MyHaplotypeCaller", reference, rodMapCall,
repartitionInfoBundle, List(recaledSamBundle), vcfBundle, useGVCF)
pipeline.addProcess(haplotypeCallerProcess)

// Issue and Execute Processes
pipeline.run()
```

Figure 3. An example of GPF user programming.

In the GPF programming framework, all RDDs are represented as `Resources`. Different from other studies using new data formats (*i.e.,* column-wise) [3, 19], our GPF directly converts the original structure of FASTQ, SAM, and

Table 2. Programming interfaces for runtime system and process in Scala language. The suffix of *Bundle* represent the corresponding RDD format.

Process Name	Description
Runtime System	
Pipeline(val name: String, sc: SparkContext)	pipeline constructor
addProcess(process: Process)	add `Process` to construct an execution DAG
run()	schedule `Process` to execution in parallel
Process in Aligner Stage	
BwaMemProcess.pairEnd(name:String, referencePath: String, inputFASTQPairBundle: FASTQPairBundle, outputSAMBundle: SAMBundle)	map reads to reference genome using BWT algorithm.
Process in Cleaner Stage	
Mark DuplicateProcess(name:String, inputSAMBundle: SAMBundle, outputSAMBundle: SAMBundle)	remove redundant alignments.
IndelRealignProcess(name:String, referencePath:String, rodMap:Map(name:String, path:String), partitionInfoBundle: PartitionInfoBundle, inputSAMList:List(SAMBundle), outputSAMList:List(SAMBundle))	adjust alignment in particular position
BaseRecalibrationProcess(name:String, referencePath:String, rodMap:Map(name:String, path:String), partitionInfoBundle: PartitionInfoBundle, inputSAMList:List(SAMBundle), outputSAMList:List(SAMBundle))	adjusting quality scores
Process in Caller Stage	
HaplotypeCallerProcess(name:String, referencePath:String, rodMap:Map(name:String, path:String), partitionInfoBundle: PartitionInfoBundle, inputSAMList:List(SAMBundle), outputVCFBundle:VCFBundle, useGVCF:Boolean)	calling variants via local de-novo assembly of haplotypes in an active region based on paired-HMM algorithm.
Auxiliary Process	
ReadRepartitioner(name:String, inputSAMBundleList: List(SAMBundle), outputPartitionInfo: PartitionInfoBundle, refernceLength:List(Int),advisedPartitionLength:Int)	generate partitioned RDD files for different formats

VCF into the RDD format. In order to achieve good scalability performance, we use an auxiliary `PartitionInfo` RDD that records the mapping of reference/read positions to partitioned regions. A `ReadRepartitioner` function (which is referred to as partition `Process`) is used to generate a `PartitionInfo` RDD. The advantage of data format of GPF is that it eliminates the step of format transformation while simultaneously generating scalable RDD partition on-the-fly.

Two types of programming interfaces are used to construct a pipeline. The first type represents runtime system interfaces that manage the execution engine (Section 4). First, users call the constructor to set up the execution environment with Spark context. Next, each `Process` is added to a dynamic DAG one-by-one according to the analysis algorithm. The `run()` function is issued to parse and execute all `Processes` in parallel. The second type algorithm-specific interfaces, which are instantiated as `Processes` in pipelines. The internal implementation of these `Processes` will be described in Section 4.4. The major functions APIs are summarized in Table 2.

Figure 3 shows an example of user programming on top of GPF framework. Users do not need to handle the parallelism required for either data distribution or parallel primitive operations in Spark. According to GPF programming model, users instantiate both `Process` and `Resource`. In this simple example, the `Resources` include the instances of `FASTQPairBundle, SAMBundle, VCFBundle`. For the pipeline of `Aligner-Cleaner-Caller`, users then create the corresponding `Processes` and add them into the runtime system driver `Pipeline`, which is finally issued and scheduled to Spark execution.

4 Execution Engine

4.1 Overview

We design our engine on top of the Spark framework, which by default keeps persistent RDDs in memory. Given the considerable volume of genomic dataset, it is usually not sufficient to fit the data in the memory. Thus, decreasing memory usage via data serialization and reducing shuffle operations are two main options to build a fast and efficient engine for large-scale genomic dataset. Our GPF engine comprises two main components: genomic data compression and a DAG scheduler. The compression method aims to persist data in the serialized form used for shuffling data between worker nodes as well as when serializing RDDs to disk. As a consequence, the volume of shuffle data will reduce, and network performance will be greatly improved. The DAG scheduler analyzes the dependency between two `Processes` and construct the DAG execution order that eliminates of shuffle operations caused by redundancy computation at the `Process`-level.

4.2 Genomic Data Compression

The genomic data compression of our GPF is designed to reduce memory usage. The in-memory computing model requires that all of the data processed are located in the memory. In the process of processing genetic data, the input data and the intermediate data are large, which require a large amount of memory, and generate the overhead of garbage collection. Moreover, in the case of poor cluster network environment, shuffling large volume datasets will turn data transmission between nodes into a system bottleneck.

A more effective way to reduce memory usage is to store objects in serialized form. Our GPF stores each RDD partition as one large byte array. The original Spark programming model provides two serialization libraries: Java serialization and Kryo serialization [27]. Kyro is significantly faster and more compact than Java serialization (often as much as 10x). However, when shuffling RDDs with complex objects or string types, the Kryo compression algorithm becomes inefficient. To overcome this bottleneck, we introduce a new data compression method. Genomic applications commonly include FASTQ, SAM, and VCF data format, in which VCF consists of the small volume result file. In our framework, we define the data structures that store FASTQ records and SAM records, and their fields correspond to the FASTQ and SAM file formats. Based on the data structure, we find that the **Sequence field** and the **Quality field** account for 80% -90% of the total length of FASTQ record. Thus, our GPF maintains the original data structure and compress the **Sequence field** and the **Quality field** respectively. Below, we introduce the compression algorithm in the execution engine.

Encoding:	A:00 G:01 C:10 T:11		
Sequence:	GGTTNCCTA		
Quality Score:	CCCB#FFFF		
Sequence Conversion:	GGTTACCTA		
Quality Score Conversion:	CCCB(SOH)FFFF		
Binary Sequence:	**(00001001)**	**01011111 00101011**	**00(000000)**
	Length of Sequence	**Compressed Binary Sequence**	**Complemen tation**

Figure 4. sequence-compression

- **Sequence Field Compression:** The stored base sequence of a sequence field is composed of the 4 characters A,G,C, and T. We use 2-bit encoding to express the sequence. When encountering special characters in an individual position in the sequence, we refer to Deorowicz [4] method, which encodes special characters into **Quality Filed**. For example, the quality score of the sequence shown in Figure 4, the illegal character N in the sequence field is 1, the compression algorithm first converts the character into A and changes the corresponding quality score to 0. [1] At the time of decompression, if the algorithm encounters the character A whose quality score is 0, the algorithm will recognize that A is a special character in the particular position before compression. In addition to the 2-bit encoding of the base moiety, a byte is used to store the length of the base sequence before compression. The base sequence before and after compression is shown in Figure 4. Our results show that our GPF improves storage by approximately four times.

[1] the range of the mass score of normal Read is [33 -126].

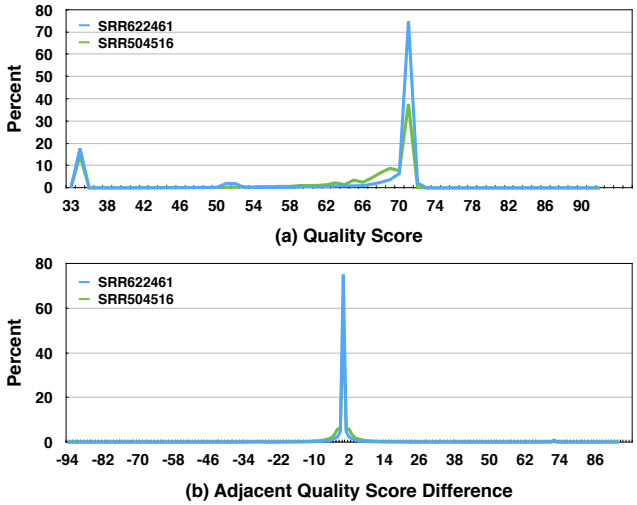

Figure 5. quality-distribute

- **Quality Field Compression:** As shown in Figure 5, the difference in the adjacent quality score of the different samples is more concentrated and easier to predict than the mass fraction itself. In addition, the vast majority of adjacent quality score differences are ranged between 0-10. Based on this feature, we convert the mass fraction sequence into the Delta sequence (the character range becomes -127 127) of the difference between the quality score, and then compress the delta sequence using Huffman coding with the end symbol of EOF (Figure 6).

Quality Score:	CCCB#FFFF
Quality Score Conversion:	CCCB(SOH)FFFF
Difference Sequence:	**67 0 0 -1 -65 -69 0 0 0**
Binary Sequence	**1101101011101101011101100101 01(EOF)**

Figure 6. quality-compression

4.3 DAG Scheduler

After users construct a pipeline with the Spark-based APIs, a driver function `Pipeline.run()` analyzes the pipeline before any committed operation. The novelty of our framework is that the execution engine performs a unified analysis and scheduling of the execution of each process, and then submit the optimized execution order. Unlike the built-in RDD dependency analysis of Spark, the dependency analysis of GPF occurs at `Process` level, which reserves the execution characteristics and algorithmic significance of each process. This allows the framework to automatically change the execution pattern of each `Process`.

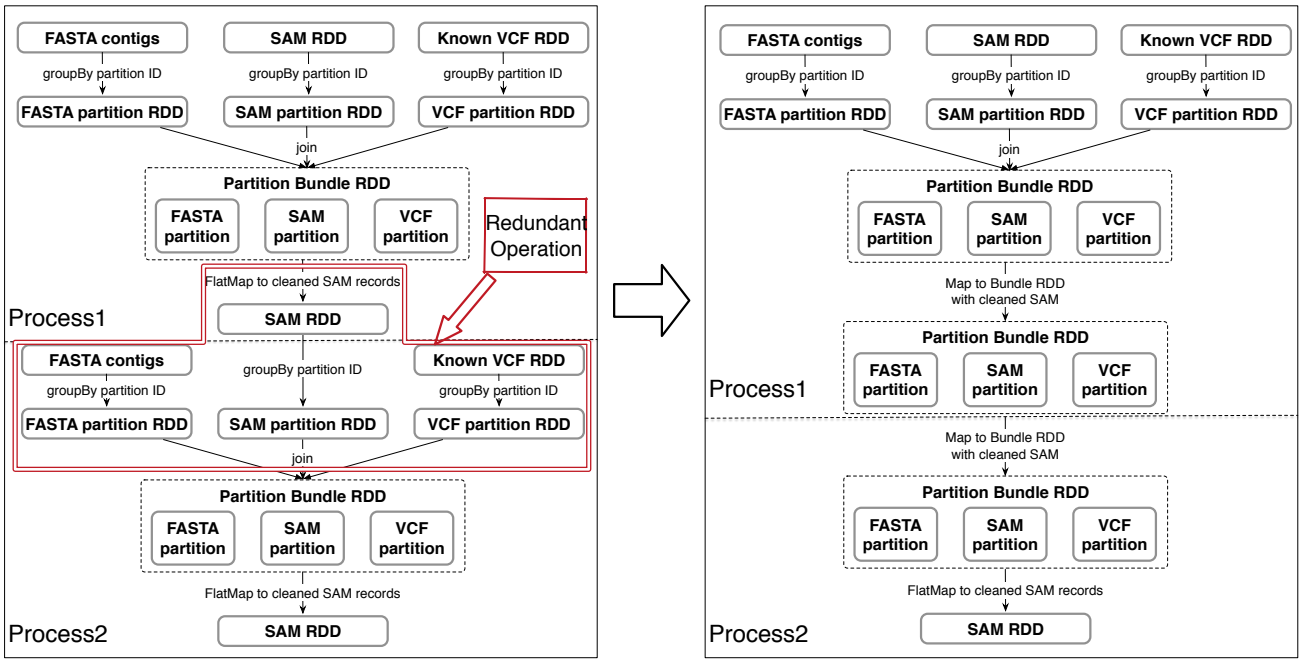

Figure 7. An illustrative example of eliminating useless shuffle and duplicated computation.

The execution engine of our GPF is designed in such manner that it builds the dependencies among processes according to the references among different resources. The algorithm 1 presents a pseudo-code of the DAG analysis. For example, when the output of `Process` A uses the same reference as the input of `Process` B, B is dependent on A. If `Process` is treated as a node and the dependency is as a directed edge from one node to another, the entire pipeline can be transformed into a DAG. The engine then performs a topology sort on the DAG to generate a proper execution order. It should be noted that the DAG may not be a connected graph. Thus, it is necessary to iterate through the topology until all processes have been executed.

Shuffle is an expensive operation since it involves disk I/O, data serialization, and network I/O. Note that most of the shuffle operations occur during partition `Process`, which prepares RDDs of either FASTA or VCF for the next stage. We observe that redundant partitions and joins exist that incur a heavy cost of during network communication. Therefore, the execution engine schedules several continuous partition `Processes` to share the same operation. The runtime system traverses DAG to identify a path where (i) each node is partition `Process`; (ii) out-degree of the start node is 1, in-degree of the end node is 1, and both in-degree and out-degree of each middle node are 1. In fact, this pattern often occurs through different stages like `Indel Realignment` and `Base Recalibration` and `HaplotypeCaller`. Figure 7 shows an

Algorithm 1 Generating execution order and detecting dependencies to schedule execution

```
1: Set<Process> unfinishedProcess = allProcess
2: List<Process> finishedProcess = {}
3: List<Resource> resourcePool= {}
4: ▷ Add Resource which has been defeined
5: for each p in unfinishedProcess do
6:     for each r in p.inputResourceList do
7:         if each r isDefined then
8:             resourcePool.add(r)
9:         end if
10:    end for
11: end for
12: while unfinishiedProcess is NotEmpty do
13:    ▷ Find out process list which can be executed in this iteration
14:    List<processToBeFinished> = {}
15:    for each p in unifishedProcess do
16:        if all input resource of the process in resource pool then
17:            processToBeFinished.add(p)
18:        end if
19:    end for
20:    ▷ None of the p can be executed
21:    if processToBeFinishedisEmpty then
22:        throw new Exception: Circular dependency
23:    end if
24:    for each p in processToBeFinished do
25:        unfinishedProcess.remove(p)
26:        finishedProcess.add(p)
27:        for each r in process.outputResourceList do
28:            resourcePool.addResource(r)
29:        end for
30:    end for
31: end while
```

example of eliminating the useless shuffles and duplicated computations.

By default, Spark submits all operations that are defined in the process (Figure 7(a)). Since FASTA and VCF are read-only, `Process 2` can reuse the FASTA partition RDD and VCF partition RDD generated by `Process 1`. In addition, `Process 1` merges the SAM results generated in each partition into a SAM RDD, which will be re-partitioned in `Process 2`. In order to aggregate several data in the form of bundled RDD, an additional join operation is required. More specifically, the execution engine parses DAGs and automatically changes the execution mode of both `Process 1` and `Process 2` to be a reduced task graph (Figure 7(b)). `Process 1` uses a map operation to convert the bundled RDD to another bundled RDD, which preserves the calculation results of FASTA, VCF, and SAM in `Process 1`. `Process 2` directly uses the bundle RDD in the next step. Thus, time-consuming redundant operations, such as FASTA, VCF partition, SAM data re-repartition, and join, have been eliminated in our GPF before the tasks are submitted to Spark runtime system.

4.4 RDD Partition

Ideally, sequencing reads are evenly distributed according to the reference. Optimal parallelism should be achieved by partitioning input data by loci for the entire workflow. However, in the 50 sequencing coverage WGS dataset [2], it is common that the depth of coverage of a targeted base is beyond $10,000x$. Thus, with the uneven distribution of coverage depth across the bases targeted in the genome, simply partitioning input data into equal length will result in load imbalance. As a result, some of the executors with heavy tasks may collapse due to lack of memory resource.

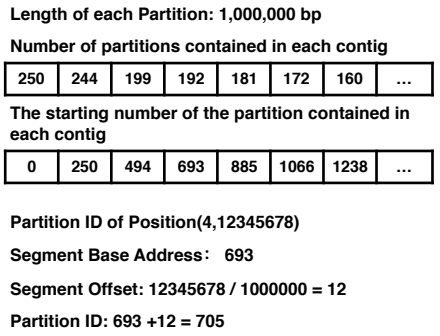

Figure 8. Mapping position into partition ID using PartitionInfo

To overcome the limitation of load imbalance, we have designed our GPF in such manner that it balances the task dynamically, namely based on the number of reads in each partition. GPF integrates a Process named `RepartitionInfoProducer` and generates `PartitionInfo` structure to describe

[2]Sequencing coverage describes the average number of reads that align to known reference bases

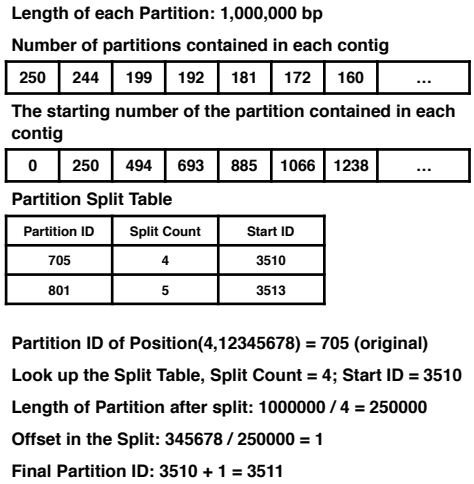

Figure 9. Using the `PartitionInfo` structure to map the location to the `partition ID`

segmentation. Here we take SAMRecord as a case to illustrate the mechanism of the `partition Process`. The `partition Process` uses a dynamic strategy consisting of three steps:

- As shown in Figure 8, at the beginning of converting SAMRecord into subregions, `RepartitionInfoProducer` equally divides the dataset into multiple subregions and generates the basic *PartitionInfo*. For an individual position (contig ID, position), we first identify the start ID of the segment based on the contig ID. Second, we obtain the offset value (position/length of partition). Finally, the sum of start ID and offset is the final partition ID result.
- Second, broadcast variables are created from the structure containing the start ID of each contig by calling SparkContext.broadcast(x). Tuple (partition id, 1) is created using transformation operation from SAMRecord RDD. Once created, *reduce* is performed to aggregate all the elements of the RDD using *collect()* primitive and returns the number of reads in each partition to the driver program. The driver program will set a segmentation threshold to cut the dataset.
- Finally, once the number of reads in a partition exceeds the segmentation threshold, the GPF will repartition the current partition. One important parameter for the new segment is the new `partition ID`. Figure 9 shows a method to create the new `partition ID`. For example, the old `partition ID` of targeted base(4,12345678) is 705, we look up the *partition split table* to find whether this partition has been split. If it is not split, the new `partition ID` is still 3510, while we find this partition was split into 4 segments. Then we calculate the length of new partition and the offset to generate the final new `partition ID` 3511.

5 Evaluation

This section reviews the speed, ease of use, scalability of our GPF. We built a typical WGS pipeline on top of GPF and evaluated our framework through experiments on large-scale clusters.

5.1 Experimental Setup and Data Sets

We use an in-memory computing environment of Apache Spark version 2.1.0. The Spark cluster was configured with up to 2048 cores, each with two Intel Xeon E5-2692v2 12-cores CPU chips at 2.2GHz and 64 GBytes of DRAM. Each node consists of 1 Seagate St9 1000640NS SATA disk (1TB, 7200 RPM). The total 240 nodes are connected by Infiniband FDR. Due to the limit of memory capacity per node for launching Spark tasks, we use up to 10 cores on each node.

In all our experiments, we use paired-end genome dataset NA12878_Platinum Genomes from Illumina HiSeq 2000, which consists of 146.9G bases, and is 500 GB in FASTQ format.[3] The known variant database used for count covariates stage is dbsnp_138.b37. We selected hg19 human genome [1] as the reference genome to which we aligned the dataset.

We evaluated GPF in terms of both performance and scalability by comparing several the state-of-the-art genomic data analysis softwares. Especially, we compared our GPF with three representative tools or frameworks:

- *Churchill* [12]: It is the few counterparts that implements a full pipeline parallelization in WGS pipeline. Since [12] has demonstrated that both HugeSeq [14] and GATK-Queue [10] showed modest improvements in speed between 8 and 24 cores (2-fold), with a maximal 3-fold speedup being achieved with 48 cores, and no additional increase in speed beyond 48 cores. We compared the performance of GPF with Churchill for cluster scalability of the whole pipeline.
- *ADAM* [19]: ADAM and GATK4 [9] present implementations on top of Spark and improve performance through in-memory caching and reducing disk I/O. However, only partial applications like `Mark Duplicate`, `BQSR`, and `INDEL Realignment` are available in ADAM. The current GATK4 is released as a beta version, implementations have not yet been fully validated for correctness nor has their performance been fully known. We compared the performance of typical GPF applications with ADAM and GATK4.
- *Persona* [3]: Different from either Hadoop or Spark in-memory computing model, Persona is a more general framework that leverages TensorFlow as a dataflow execution engine. Like ADAM, the framework only supports algorithms in either **`Aligner`** or **`Cleaner`**.

[3]The sequenced reads are stored as ASCII strings (roughly 100 characters each).

5.2 Performance Evaluation

5.2.1 Cluster Scalability

GPF can scale out to reduce the time for sequencing data analysis by distributing computation across a cluster. Thus, cluster performance is critical. Due to the limited of memory capacity per node[4], we measured scalability starting from 128 cores. As shown in Figure 10, our GPF-based pipeline achieved more than 50% parallel efficiency over 2048 cores and finished WGS analysis in 24 minutes on the NA12828 high-coverage human genome. We found that the parallel efficiency and the number of cores were much higher than those of previous studies whose parallel efficiency was approximately 15% on tens to hundreds of cores [10, 12, 14, 18, 19]. As a result, Churchill enables the whole analysis to be completed in 128 minutes with 1024 cores. Due to the chromosomal subregion is decided at the beginning of the analysis and the inherent load imbalance of the strategy mentioned in Sec 4.4, the scalability of Churchill was limited to 1024 cores. Our GPF overcame the bottleneck through in-memory caching and by reducing the disk I/O and by using the dynamic repartition mechanism. As a result, our GPF performed about three times faster than Churchill.

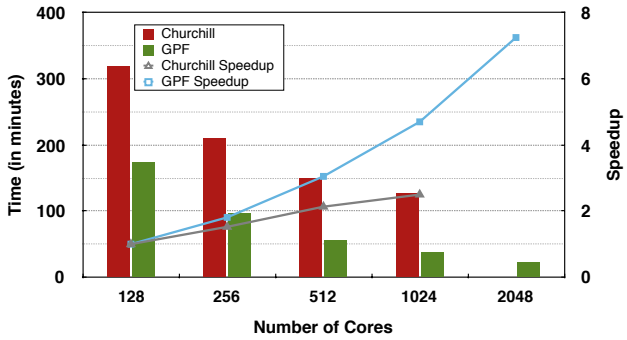

Figure 10. Execution time and scalability with the increasing number of cores.

5.2.2 Comparison with In-memory Programming Approaches

To test the efficiency of GPF with the current implementations that are used in ADAM and GATK4, we ran strong scaling experiments for the different steps of the pipeline. In this experiment, we use the high coverage genome NA12878 genomic dataset. We hold the executor configuration as detailed in Sec 5.1. Figure 11 compare these results to the ADAM and GATK4 respectively. In general, implementations of various algorithms in the pipeline were more efficient and effective than the same algorithms implemented in the ADAM and GATK. GPF outperformed the ADAM when running `Mark Duplicate` (7.3x speedup), `INDEL realignment` (7.6x

[4]64GB RAM in each node is beyond the actual capacity requirement of caching intermediate data with more than 10 cores

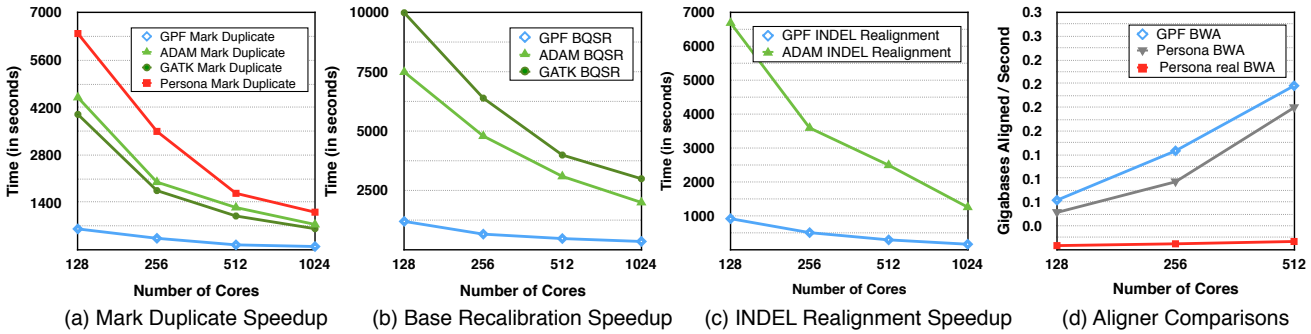

Figure 11. Strong scaling characteristics of GPF. GPF outperforms the ADAM when running equivalent implementations of BQSR (6.4x speedup), `Mark Duplicate` (7.3x speedup), and `INDEL realignment` (7.6x speedup). Further, GPF outperforms GATK4 6.3x speedup when running `Mark Duplicate` and 8.4x speedup with BQSR.

speedup), and BQSR (6.4x speedup). Compared with the latest implementation of these algorithms in the GATK4, GPF performed 6.3x speedup to the GATK when running `Mark Duplicate` and 8.4x speedup when running BQSR respectively. The `Collect` action after BQSR introduced a serial step that runs for several minutes, as the multiple gigabyte mask table was broadcast to all of the nodes in the cluster, which slowed the parallel efficiency of BQSR.

5.2.3 Comparison with Approaches Using Other Framework

In addition to Hadoop MapReduce and in-memory computing distributed programming model, studies like Persona [3] seek to embed genomic tools in dataflow framework like Google TensorFlow, which inherently allows data partitioning and distribution across clusters. However, a direct performance comparison of GPF and Persona is difficult. Persona integrated SNAP [26] as a reader aligner for cluster scaling implementation, and it used single-end reads. In the GPF, we chose BWA to align paired-end reads because paired-end reads lead to much better alignment results in terms of the biology. Thus, we based the performance comparison on implementations that were available in the repository of Persona on Github. We compared the performance of Persona-BWA and duplicate marking to our GPF, whose algorithm we have used in our implementations. Figure 11 (a) shows the results when duplicating paired-end reads, our GPF can mark duplicates up to 10 times faster than Persona. Figure 11 (d) shows the alignment throughput of Persona and GPF as a comparison. Here we use half of a paired-end whole genome dataset from NA12878. Figure 11 (d) reports the giga bases aligned per second for a single genome. BWA in our GPF aligns the genome with higher throughput than Persona-BWA. Besides, when considering data format conversion time as mentioned in Persona (FASTQ is imported to AGD data format in Persona at 360 MB/s, while the alignment result (BAM format files) are produced from AGD at 82 MB/s) [3]. Persona performed WGS alignment for a typical dataset in 16.7 seconds. Taking the platinum standard genome data as an example, time of data format conversion is approximately 3300 seconds, which is 200 times that of the alignment time. Thus, Persona should take into account the conversion time, the red line in Figure 11 (d) shows the alignment throughout of Persona-BWA with the conversion time. Practically, the real throughput of Persona-BWA was about 20 times lower than BWA of GPF.

5.2.4 Specific Advantages Caused by Optimizations

The serialization and compression algorithm in the GPF is significantly faster and more compact than Java serialization and Kryo. Also, the novel execution engine enables eliminates redundant operations to reduce data shuffle overhead. We therefore studied the performance impact of data serialization and shuffle operation here:

Effect of Genomic Data Compression: Table 3 shows the effect of data compression. GPF reduces memory consumption by 50% totally. Spark actions are executed through a set of stages, separated by distributed shuffle operations. We take three running stages as examples, which are used for not only shuffling compressed data between worker nodes but also to disk. Stage 1 has the best compression effect when shuffling FASTQ RDDs. Dataset in Stage 1 is more compact than in Stage 5 since SAM formats in Stage 5 have various fields, and these fields are not compressed. When shuffling RDDs with FASTA, SAM, and VCF formats in Stage 20, the compression rate is slightly lower but the total compression rate is essentially constant.

Table 3. Efficient compression of genomic data

Stage ID	Describtion	Orgin	Compressed
1	Load FASTQ	20.0GB	11.1GB
5	Segment SAM	22.8GB	14.4GB
20	Generate Bundle RDD	27.0GB	18.7GB

Table 4. Redundant Shuffle Operations

Pipeline	Orignal	Redundant Calculations
Running Time	21min	18mins
Stage Num.	38	22
Core Hour	74.95h	63.98h
GC Time	7.16h	6.34h
Shuffle Time	46.83min	24.29min
Shuffle Data	326.1GB	187.0GB

Effect of Redundancy Elimination: To compare the effect of redundancy elimination, we tested the performance gains of this optimization strategy. We chose a 256 cores cluster and the SRR622461 dataset (18.7G bases paired-end reads). Table 4 shows the performance gain of redundancy elimination strategy. The second and third columns indicate whether this mechanism is activated respectively. As can be seen from the above chart, the optimized cluster disk and network usage has been significantly reduced. This is because the execution engine eliminated the redundancy calculation that contains a lot of *groupBy*, *join* and Shuffle operations.

5.3 Performance Analysis

To identify whether scale-out beyond 2048 cores is feasible, we systematically identified the potential performance bottlenecks of our GPF. Given the large volume of genomic data, the probability is high that genomic data will not fit in memory. The assumption is that I/O of our GPF is still a bottleneck. Our expectation is that time blocked on either disk or network would represent the majority of job completion time.

5.3.1 I/O Behavior of GPF

Using blocked time analysis [24], we computed the improvement in job completion time if tasks did not spend any time blocked on either disk or network I/O. The improvement indicates that an upper bound of genomic applications from disk and network I/O optimization.

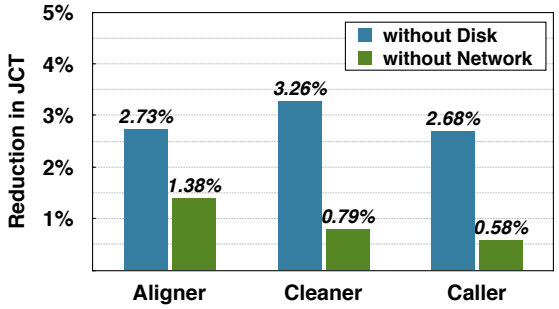

Figure 12. Improvement in job completion time (JCT) as a result of eliminating all time blocked on disk or network I/O.

Diks I/O: Time blocked on disk I/O consists of two types of points in task execution: the first is time that the task spent blocked waiting for shuffle data to be read from remote machine and the second is time that the tasks which write shuffle data to disk. Spark writes all shuffle data to disk, even when input data are read from memory. Using blocked time analysis, we found that the median improvement from eliminating all time blocked on disk is at most 2.7% across all workloads as shown in Figure 12, which includes all trials of each stage in each workload in the WGS pipeline. The fact that this improvement is non-zero indicates that even in-memory workloads store shuffle data on disk. We also found that the amount of data sent over the network is often much less than the data transferred to disk because the memory capacity is limited.

Network I/O: To test the improvement in job completion time as a result of eliminating all time blocked on network, we measured the largest possible improvement from optimizing the network. As shown in Figure 12, the largest improvement in job completion time as a result of taking out network time is 1.38%. The blocked time instrumentation for the network included time to read shuffle data over the network and the time to write output data to one local machine and two remote machines. Both of these times include disk use as well as network use because disk and network are interlaced in a manner that makes them difficult to measure separately.

5.3.2 Bounding Factor Analysis

These results eliminated the possibility that disk or network creates a bottleneck for GPF. Time blocked on disk I/O and network I/O can only improves the job completion time (JCT) by a maximum of 4.6% in our GPF. To better understand this measurement, we identified the true bounding factors by comparing resource utilization between CPU and I/O.

Figure 13 (a) and 13 (b) displays the throughput of disk read/write and network transfer, respectively. Figure 13 (a) displays aggregated disk throughput and IOPS data for the execution on 2048 cores. Evidently, the data volume of disk read/write can not saturate the disk bandwidth. At the beginning of the pipeline, the conversion of the FASTQ file to RDD format generate intensive disk and network operations. During the process of pipeline execution, the `Processes` operate on RDD in memory. However, due to the limited memory space, some shuffle operations incur scattered writes on either local disk and remote disk. In the **`Caller`** phase, a re-partition `Process` is applied to evenly distributed active regions among the task for load balance. The re-partition was designed to generate abundant serializations to disk files which are then read by the paired-HMM algorithm for traversing active regions to call variants.

The blocked time analysis is consistent with resource utilization (Figure 13 (c)). Eliminating time blocked on disk I/O can only improve job completion time by a maximum of

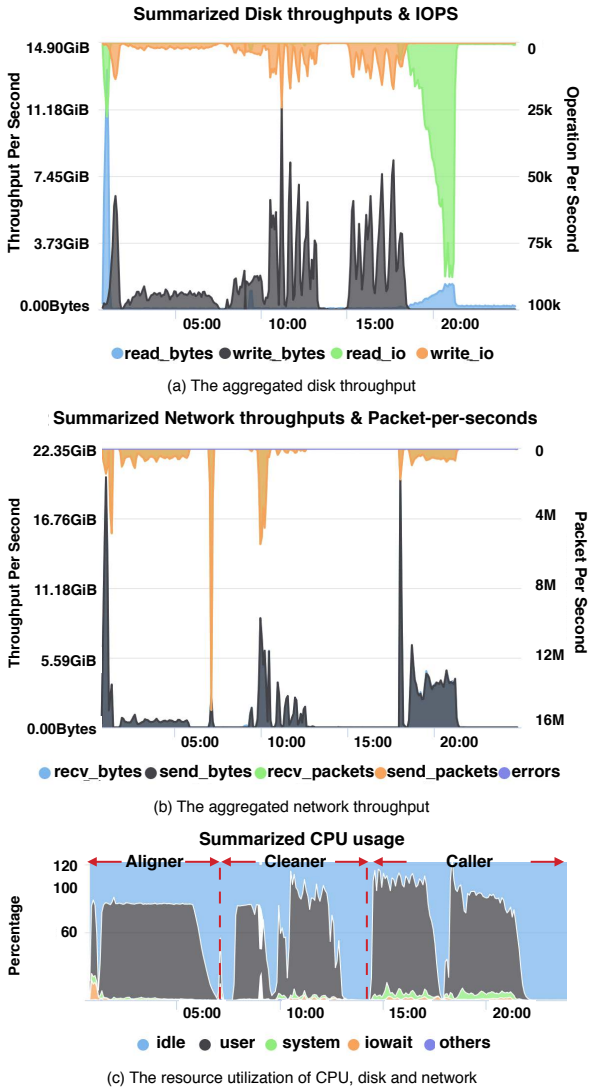

Figure 13. Performance profiling of GPF system on the 2048 cores Spark cluster.

4.6%. This suggested that jobs may be CPU bound. Because measuring when a task is using only a CPU versus when background I/O is occurring is difficult, we instead examined the utilization of both CPU and disk utilization to understand the importance of CPU use. Figure 13 (c) shows that tasks are likely blocked while waiting on computation to complete. According to the x-axis in Figure 13 (c), the largest number of CPU-intensive jobs was found in the alignment, recalibration and variant calling steps of the pipeline. Both the `BWA-MEM` and `HaplotypeCaller` are computationally intensive components of the pipeline in which CPU architecture and speed completely determine efficiency and time-to-solution. To improve I/O performance, serialization and compression algorithm in the engine decrease the I/O and increase the CPU requirements, which is one reason for the relatively high use of CPU in our GPF. However, considering the volume of genomic data, serialization and compression formats will inevitably evolve in the future. In conclusion, CPU utilization is currently much higher than disk, the utilization ration can navigate the tradeoff between CPU and I/O time when tuning the in-memory computing model.

6 Related Work

Table 5 summarizes results from our experiment or cited paper in usability and implementation perspectives. To date, most implementations use map-reduce as a programming model. [10, 12–14, 20, 22] GATK-Queue [10] provides a scatter-gather parallelization to run on multiple nodes. Both HugeSeq [14] and Churchill [12] optimize parallel performance by partitioning either reads or genomes. Churchill enables division across genomic regions with fixed boundaries. These studies use workflow management systems for sharing and persisting intermediate data, which must spill to disk. Moreover, most of these studies are based on the current Sequence/Binary Alignment/Map (SAM/BAM) formats, which was not designed to scale well to large datasets. To improve I/O performance through in-memory computing, ADAM and GATK4 implement well-established tools on top of Apache Spark. ADAM provides a set of APIs based on GATK, while only partial programs like `BQSR` and `INDEL Realignment` in **Cleaner** are available. GATK4 is released as a beta version, most of the components are still under development.

Table 5. Comparison of various platforms for genome data analysis. The numbers of GPF, GATK4, and Persona are from our experiment results. Others are from the cited papers.

	Parallel Framework	In-memory Computing	#Cores	Parallel Efficiency
GPF	full	√	2048	> 50%
Churchill [12]	full	×	768	28%
HugeSeq [14]	full	×	48	~50%
GATK-Queue [10]	full	×	48	~50%
ADAM [19]	Cleaner	√	1024	14.8%
GATK4 [9]	Cleaner&Caller	√	1024	41.6%
Persona-BWA [3]	Aligner&Cleaner	×	512	51.1%

7 Conclusion

In recent years, NGS methods enable researchers to perform whole-genome studies with large quantities of data, which requires ease of use, high performance, and reproducibility of computational tools. Our GPF allows users to write a serial program according to the protocol of bioinformatics pipeline (*think-in-serial*) while transparently take advantage of high performance in-memory computing models (*run-in-parallel*). In conclusion, we believe that our novel GPF system will render various disparate genomic applications into high-performance, easy-to-use systems. Thus, our system presents an important addition to already available tools used for handling WGS-related data and associated genome analysis. Importantly, with tools like our GPF is should be possible to move closer to the realization of personalized medicine.

Acknowledgments

This research is supported The National Key Research and Development Program of China (2016YFB0201305,2016YFB0200504, 2016YFB0200803,2016YFB0200300) and National Natural Science Foundation of China, under grant no. (61521092, 91430218, 31327901, 61472395, 61432018). We would like to express sincere thanks to our shepherd, Frank Mueller, and our reviewers for their constructive feedbacks.

References

[1] 2016. HG19 Human Genome Download. http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/. (2016).

[2] 2016. Picard: A set of command line tools (in Java) for manipulating high-throughput sequencing (HTS) data and formats such as SAM/BAM/CRAM and VCF. http://broadinstitute.github.io/picard/. (2016).

[3] Stuart Anthony Byma, Sam David Whitlock, Laura Flueratoru, Ethan Tseng, Christos Kozyrakis, Edouard Bugnion, and James Larus. 2017. Persona: A High-Performance Bioinformatics Framework. In *USENIX Annual Technical Conference 2017*.

[4] Sebastian Deorowicz and Szymon Grabowski. 2011. Compression of DNA sequence reads in FASTQ format. *Bioinformatics* 27, 6 (2011), 860–862.

[5] Apache Software Foundation. Online. Apache Hadoop. http://hadoop.apache.org/. (Online).

[6] Claudia Gonzaga-Jauregui, James R Lupski, and Richard A Gibbs. 2012. Human genome sequencing in health and disease. *Annual review of medicine* 63 (2012), 35–61.

[7] Illumina. 2012. HiSeq Sequencing System. http://www.illumina.com/. (2012).

[8] illumina. 2017. NovaSeq. https://www.illumina.com/systems/sequencing-platforms/novaseq.html. (2017).

[9] Broad Institute. Online. GATK-4. https://github.com/broadinstitute/gatk. (Online).

[10] Broad Institute. Online. GATK Queue. http://gatkforums.broadinstitute.org/discussion/1306/overview-of-queue. (Online).

[11] Scott D Kahn. 2011. On the future of genomic data. *science* 331, 6018 (2011), 728–729.

[12] Benjamin J. Kelly, James R. Fitch, Yangqiu Hu, Donald J. Corsmeier, Huachun Zhong, Amy N. Wetzel, Russell D. Nordquist, David L. Newsom, and Peter White. 2015. Churchill: an ultra-fast, deterministic, highly scalable and balanced parallelization strategy for the discovery of human genetic variation in clinical and population-scale genomics. *Genome Biology* 16, 1 (2015), 1–14. https://doi.org/10.1186/s13059-014-0577-x

[13] Patricia Kovatch, Anthony Costa, Zachary Giles, Eugene Fluder, Hyung Min Cho, and Svetlana Mazurkova. 2015. Big omics data experience. In *the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM Press, New York, New York, USA, 1–12.

[14] Hugo Y K Lam, Cuiping Pan, Michael J Clark, Phil Lacroute, Rui Chen, Rajini Haraksingh, Maeve O'Huallachain, Mark B Gerstein, Jeffrey M Kidd, Carlos D Bustamante, and Michael Snyder. 2012. Detecting and annotating genetic variations using the HugeSeq pipeline. *Nat Biotech* 30, 3 (03 2012), 226–229. http://dx.doi.org/10.1038/nbt.2134

[15] Ben Langmead, Cole Trapnell, Mihai Pop, Steven L Salzberg, et al. 2009. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biol* 10, 3 (2009), R25.

[16] Heng Li and Richard Durbin. 2009. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics* 25, 14 (2009), 1754–1760.

[17] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, et al. 2009. The sequence alignment/map format and SAMtools. *Bioinformatics* 25, 16 (2009), 2078–2079.

[18] Xueqi Li, Guangming Tan, Chunming Zhang, Xu Li, Zhonghai Zhang, and Ninghui Sun. 2016. Accelerating large-scale genomic analysis with Spark. In *Bioinformatics and Biomedicine (BIBM), 2016 IEEE International Conference on*. IEEE, 747–751.

[19] Matt Massie, Frank Nothaft, Christopher Hartl, Christos Kozanitis, André Schumacher, Anthony D Joseph, and David A Patterson. 2013. Adam: Genomics formats and processing patterns for cloud scale computing. *University of California, Berkeley Technical Report, No. UCB/EECS-2013* 207 (2013).

[20] A Mckenna, M Hanna, E Banks, A Sivachenko, K Cibulskis, A Kernytsky, K Garimella, D Altshuler, S Gabriel, and M Daly. 2010. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research* 20, 9 (2010), 1297–303.

[21] Michael L Metzker. 2010. Sequencing technologies—the next generation. *Nature reviews genetics* 11, 1 (2010), 31–46.

[22] Nabeel M Mohamed, Heshan Lin, and Wuchun Feng. 2013. Accelerating data-intensive genome analysis in the cloud. In *Proceedings of the 5th International Conference on Bioinformatics and Computational Biology (BICoB), Honolulu, Hawaii, USA*.

[23] Frank Austin Nothaft, Matt Massie, Timothy Danford, Zhao Zhang, Uri Laserson, Carl Yeksigian, Jey Kottalam, Arun Ahuja, Jeff Hammerbacher, Michael Linderman, et al. 2015. Rethinking data-intensive science using scalable analytics systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 631–646.

[24] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and V ICSI. 2015. Making Sense of Performance in Data Analytics Frameworks.. In *NSDI*, Vol. 15. 293–307.

[25] Cole Trapnell, Adam Roberts, Loyal Goff, Geo Pertea, Daehwan Kim, David R Kelley, Harold Pimentel, Steven L Salzberg, John L Rinn, and Lior Pachter. 2012. Differential gene and transcript expression analysis of RNA-seq experiments with TopHat and Cufflinks. *Nature protocols* 7, 3 (2012), 562.

[26] Matei Zaharia, William J. Bolosky, Kristal Curtis, Armando Fox, David A. Patterson, Scott Shenker, Ion Stoica, Richard M. Karp, and Taylor Sittler. 2011. Faster and More Accurate Sequence Alignment with SNAP. *CoRR* abs/1111.5572 (2011). arXiv:1111.5572 http://arxiv.org/abs/1111.5572

[27] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[28] Jing Zhang, Heshan Lin, Pavan Balaji, and Wu-chun Feng. 2013. Optimizing Burrows-Wheeler Transform-Based Sequence Alignment on Multicore Architectures. *CCGRID* (2013), 377–384.