Doppio: I/O-Aware Performance Analysis, Modeling and Optimization for In-Memory Computing Framework

Peipei Zhou,* Zhenyuan Ruan,* Zhenman Fang,*‡ Megan Shand,† David Roazen,† Jason Cong*
*University of California Los Angeles, †Broad Institute, ‡Xilinx
{memoryzpp, zainryan, zhenman, cong}@cs.ucla.edu, {mshand, droazen}@broadinstitute.org

Abstract*—In conventional Hadoop MapReduce applications, I/O used to play a heavy role in the overall system performance. More recently, a study from the Apache Spark community—state-of-the-art in-memory cluster computing framework—reports that I/O is no longer the bottleneck and has a marginal performance impact on applications like SQL processing. However, we observe that simply replacing HDDs with SSDs in a Spark cluster can have over 10x performance improvement for certain stages in large-scale production-quality genome processing. Therefore, one key question arises: *How does I/O quantitatively impact the performance of today's big data applications developed using in-memory cluster computing frameworks like Apache Spark?

In this paper we select an important yet complex application—the Spark-based Genome Analysis ToolKit (GATK4)—to guide our modeling. We first use different combinations of HDDs and SSDs to measure the I/O impact on GATK4 and change the CPU core number to discover the relation between computation and I/O access. By combining with Spark's underlying implementations, we further analyze the inherent cause of the above observations and build our model based on the analysis. Although we are building upon GATK4, our model maintains generality to other applications. Experimental results show that we can achieve a performance prediction error rate within 10% for typical Spark applications of both iterative and shuffle-heavy algorithms. Finally, we further extend our model to a broader area—that of optimal configuration selection in the public cloud. In Google Cloud, our model enables us to save 38% to 57% of cost for genome sequencing compared with its recommended default configurations. Currently, more and more companies are adopting cloud computing for specific workloads. Our proposed model can have a huge impact on their choices, while also enabling them to significantly reduce their costs.

I. Introduction

Within the past decade, there has been great success in programming frameworks that support efficient development and deployment of large-scale applications in datacenters. Examples include the pioneering MapReduce framework [1] initially proposed by Google, the open-source Hadoop MapReduce framework [2], and the more recent Apache Spark framework that improves the performance of Hadoop by up to 100x through in-memory cluster computing [3]. Due to its high performance efficiency, Spark has attracted increased attention from both academia and industry.

In such big data computing frameworks, I/O used to play an important role in system performance, and it attracted a significant amount of research [4, 5, 6, 7]. For example, Kambatla et al. reports in [4] that SSDs can deliver up to 70% performance improvement for Hadoop MapReduce workloads. On the other hand, a recent work [5] from the Apache Spark community claims that eliminating I/O accesses in Spark SQL processing workloads can reduce job completion time by a median of at most 19%. Thus, I/O tends to no longer be the bottleneck for the Spark in-memory computing framework. Such studies present quite different implications of the I/O impact based on their application domains and hardware resources; this often confuses users. As a result, the following key question arises: *How does I/O impact the big data application performance running on top of in-memory cluster computing frameworks like Apache Spark?*

Having a quantitative understanding of a complex distributed system like Spark is not trivial. Unfortunately, previous studies in modeling Spark performance [6, 8, 9] usually overlook the impact of I/O in their models. To better answer the above question, we first measure the performance impact of I/O on Spark by conducting an in-depth case study on the Spark-based production-quality Genome Analysis ToolKit (GATK4) [10]. GATK4 is one of the most important tools in computational genomics, and it has great potential for providing personalized medicine [11]. Since it involves various types of Spark operations, it is typical and complex enough for our motivation study. A brief introduction to Apache Spark and the GATK4 application will be presented in Section II.

Different from [5], we observe that I/O can still play a heavy role—even in the in-memory computing framework of Spark. In addition to the HDFS read and write for the input and output data which introduce I/O access, we make three other observations; these are analyzed in Section III. First, to avoid the time-consuming recomputation, certain RDD (resilient distributed dataset) operations like groupByKey will perform shuffle write/read to write/read intermediate data to/from storage between different Spark stages. Second, there is usually some non-cached intermediate data (RDD) as it is too large to be totally put into memory. For example, for GATK4, to cache a 30x coverage whole genome intermediate data, at least 3.2TB total CPU memory is required. It requires more memory for higher coverage genome inputs. Third, effective I/O bandwidth differs under various data request sizes for different I/O devices including HDD and SSD. For example, the bandwidth difference between HDD and SSD for the HDFS read operation in GATK4 is 3.7x; however, it is 32x for shuffle read operation. In that way, performance under HDD and SSD varies a lot for some Spark stages while not much for others.

To gain more insights into how I/O impacts the performance of Spark applications, we propose a generic I/O-aware analytical model to reason the underlying behavior of different Spark RDD operations and model their performance in Section IV. In this model, we analytically combine all the following factors together, which often have been overlooked in past studies.

1. We incorporate the effective I/O bandwidths under different data block sizes during different RDD accesses, including HDFS read/write, shuffle read/write, and persist read/write.
2. We incorporate the I/O bandwidth contention from different CPU cores, and quantify the break point (i.e., number of CPU cores) after which the I/O bandwidth is saturated.

3. We also incorporate the overlap between the CPU computation and I/O accesses from different data partitions, where we assume a simple but effective pipeline execution model.

To validate the accuracy of our proposed I/O-aware analytical model, we choose a set of representative Spark applications, including GATK4 and five other typical applications of iterative algorithms and shuffle-heavy algorithms. As detailed in Section V, experimental results show that our model achieves a performance prediction error rate within 10%, and well explains the runtime behavior of different stages in those applications.

This **quantitative** model enables us to better understand the performance of Spark applications for further optimizations. Moreover, our model can also be applied in the public cloud. In Section VI we conduct a case study to use our model to optimize the cost of genome sequencing in Google Cloud, where the cost of execution can be modeled as $Cost = f(CoreNum, DiskTypes, DiskSize_{HDFS}, DiskSize_{Spark_Local}, Time)$. Using the basic application profile and platform configuration $(CoreNum, DiskTypes, DiskSize_{HDFS}, DiskSize_{Spark_Local})$, our model can derive the application execution *Time*. Therefore, we can explore the platform configuration space to find the optimal one with the minimum cost. Experimental results demonstrate that we can save 38% to 57% of cost compared to default configurations recommended by Spark [12] and Cloudera [13].

Although we only discuss the above usages due to space limits, our model can be utilized for other purposes as well. For example, in a shared cluster environment with a job scheduler, our performance prediction model can allow the scheduler to know ahead the approximating job execution time and thus enable better job scheduling with less job waiting time.

In summary, this paper makes the following contributions:

1. A quantitative performance analysis on the Spark-based production-quality genome analysis toolkit.
2. An accurate I/O-aware performance analytic model for a broad set of Spark applications.
3. A model-driven cost optimization study for genome sequencing in the public Google Cloud.

In addition, we open source our toolset Doppio[1] [14], incorporating the I/O-aware model to the community.

II. Background

A. Apache Spark

Apache Spark [3] is a widely used in-memory large-scale data processing framework. Spark exposes a programming model to big data application developers based on resilient distributed datasets (RDDs). The RDD abstraction provides a series of *transformations* (e.g., map, filter) and *actions* (e.g., collect, count) that enable lazy evaluation of data partitions over a cluster of nodes. By caching RDDs in memory and thus reducing I/O accesses, Spark often achieves significant performance improvement over the Hadoop MapReduce [2] framework.

A typical Spark application launches its driver program on a master node and then distributes its tasks (data partitions) into a cluster of slave/worker nodes for parallel processing. Each slave node will read its data partitions from a distributed file system (e.g., HDFS [15]) and perform parallel computations. In addition, each slave node in Spark has its local storage directory (spark.local.dir) to store data, including RDDs, that persist on disk specified by a user program, or intermediate data [16, 17] preserved by the Spark framework. In the following we will interchangeably use the term Spark Local to refer to this local directory.

B. Genome Analysis ToolKit (GATK4)

Building a performance model for a complex distributed system like Spark is not trivial. Therefore, we first analyze a realistic and complex Spark application—GATK4—and use its analyzing result to guide our modeling.

GATK4 is a Spark-based production-quality genome analysis toolkit widely used in computational genomics. It includes three major stages: 1) MarkDuplicate (MD), which groups reads (small DNA fragments from the biochemical sequencing machine) by alignment information and marking duplicate reads; 2) BaseRecalibrator (BR), which builds a statistical model on how to update the quality scores of the aligned reads; and 3) SaveAsNewAPIHadoopFile (SF), which updates the quality scores and saves the analysis-ready reads into storage. In addition to the genome reads in the BAM file [18, 19], GATK4 also takes two other input files: 1) the VCF file that contains all known genome variations, and 2) the reference genome file to which all the genome reads are aligned. The output of GATK4 can be used to conduct genome-wide association studies (GWAS, also known as population studies) to discover unknown genome variations. The main data flow of GATK4 in the Spark perspective is shown in Fig. 1.

C. Experiment Setup

Table I describes the system software and hardware configuration for each slave node. Table II describes the default Spark and HDFS configuration in our cluster setting unless otherwise specified. Although we use fixed Spark configurations here, out proposed model can work for other configurations as well. We will revisit this in Section IV.

TABLE I: Software and hardware configuration

System	Linux kernel	4.4.0-104-generic
	CPU	2×Intel Xeon CPU E5-2699 v3 = 36 cores
	RAM size	128 GB
	Network	10Gb/s
HDD	Model	Western Digital 4000FYYZ-01UL1B2
	RPM	7200
	Capacity	4 TB
	max_sectors_kb	512 KB
SSD	Model	SAMSUNG MZ7LM240HCGR-0E003
	Capacity	240 GB
	max_sectors_kb	512 KB

TABLE II: Spark and HDFS configuration

Spark	version	spark-1.6.2
	SPARK_WORKER_CORES	36
	SPARK_WORKER_MEMORY	90 GB
HDFS	version	hadoop-2.6.0
	dfs.blocksize	128 MB
	dfs.replication	2
JAVA	version	jdk 1.8.0_73

As an example, we process a single whole human genome with 30x coverage sampled from a patient with breast cancer (HCC1954 [20]). This genome contains 500 million read pairs, each read with around 101 nucleotides. This input genome

[1]Doppio is the anagram for I/O-aware Performance analysis, moDeling and OPtimization for in-memory computing framework.

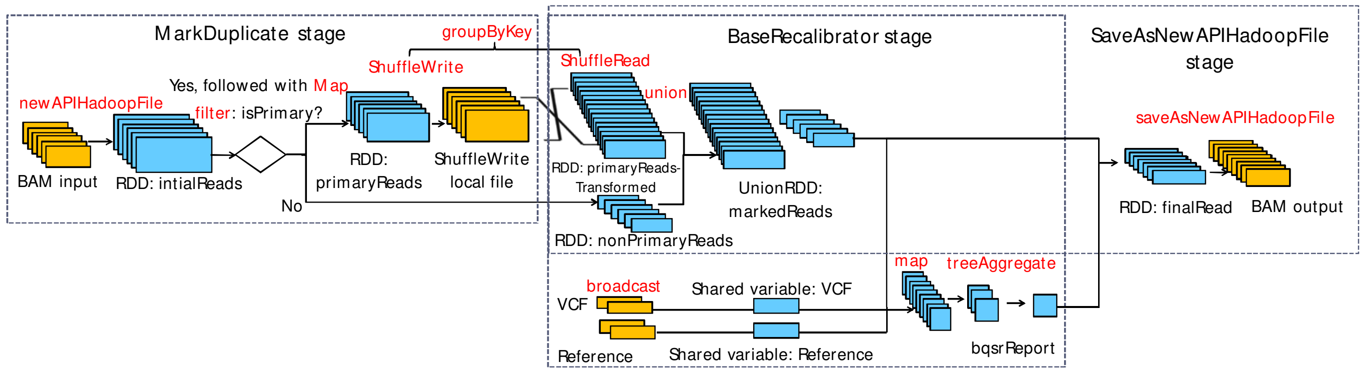


Fig. 1: The Spark RDD flow of GATK4 pipeline.

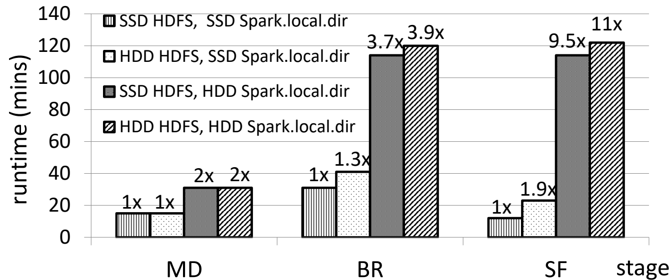


Fig. 2: Runtime for different stages in GATK4 using 500M read pairs input in four-node cluster, each with 36 executor cores.

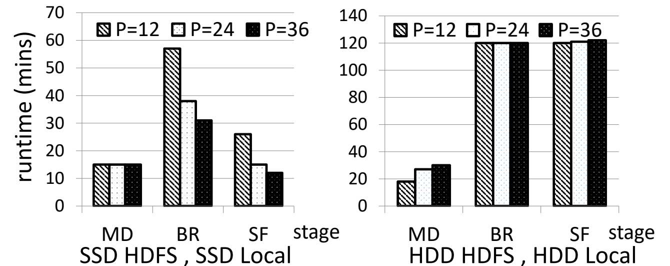


Fig. 3: Runtime for 2HDD and 2SSD cases when the number of CPU cores per node P = 12, 24, 36.

in a compressed BAM file is around 122GB, and the output analysis-ready genome file (also in compressed BAM format) is around 166GB. To measure the impact of I/O on the GATK4 performance, we test its performance under four different HDD/SSD configurations for each node, as summarized below.

TABLE III: Hybrid configurations of HDDs and SSDs

Configuration	1	2	3	4
HDFS	1 SSD	1 HDD	1 SSD	1 HDD
Local (spark.local.dir)	1 SSD	1 SSD	1 HDD	1 HDD

We leave the description of five other typical Spark applications in Section V.

III. GATK4 Performance Analysis

Using the Spark-based GATK4 as a motivational example, we first measure the impact of I/O on its performance under different execution stages with different RDD operations, and different numbers of CPU cores. After that, we summarize the observations that motivate us to build an I/O-aware performance analytical model for Spark applications. A **four-node** small cluster (one for master) is used in this example.

A. GATK4 Performance Profile Results

We break down the I/O impact into different GATK4 stages using 500M read pairs input and 36 Spark executor cores. As shown in Fig. 2, we find that the performance impact varies considerably in different stages with different RDD operations. The I/O data sizes for different stages are shown in Table IV.

TABLE IV: I/O data size (GB) in different GATK4 stages

I/O (GB)	HDFS read	Shuffle write	Shuffle read	HDFS write
MD	122	334	0	0
BR	122	0	334	0
SF	122	0	334	166

1. Changing the HDFS folder from HDD to SSD has no performance gain for the MD stage (though MD has the same I/O data access size as the BR stage according to Table IV), up to a 30% and 90% performance gain for the BR and SF stages, respectively.
2. The sensitivity to Spark Local bandwidth varies in the different stages. It is interesting that the major time-consuming stage changes from BR to SF and BR when switching Spark Local from SSD to HDD.
3. Spark Local is much more IO-sensitive than HDFS.

Finally, we evaluate the I/O impact on different stages under different numbers of CPU cores, using 500M read pairs input. As shown in Fig. 3, when the number of cores (P) increases from 12 to 36, the runtime for the BR and SF stages decreases in the 2SSD configuration but stays the same in the 2HDD configuration. For the MD stage, the runtime almost stays the same when P changes in the 2SSD as well as 2HDD configurations. We observe that when there are more cores, SSDs *might* achieve a larger performance gain than HDDs.

As observed above, I/O still plays a heavy role in the GATK4 performance. As we will demonstrate in Section V-B, even in typical Spark iterative algorithms where intermediate data is cached in memory, I/O can play an important role with a small iteration number (less than 100). The performance gap between HDD and SSD can be as large as 2x.

B. Analysis of I/O-intensive Operations

After a detailed analysis, we find there are two other major I/O-intensive operations in GATK4, in addition to the HDFS read and write of the input and output file. These two operations are discussed in detail in the following.

1) Shuffle-Heavy RDD Operations: Certain RDD operations in Spark, such as groupByKey() and repartition(), involve very time-consuming shuffle operations. During shuffling, mappers write all the intermediate data into the Spark Local.

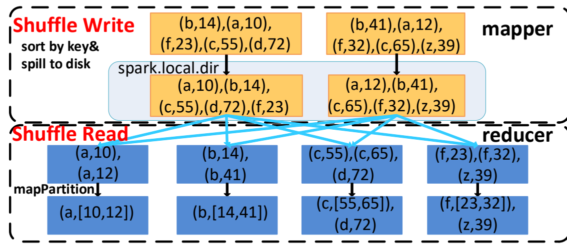


Fig. 4: An illustration of the groupByKey operation.

Later, reducers read those intermediate results from the mappers' local storage [16, 17]. Such RDD operations involve a significant amount of I/O accesses.

Fig. 4 presents an overview of the groupByKey() operation used in Spark, which has two phases: ShuffleWrite and ShuffleRead. To redistribute and group the data based on its key, the ShuffleWrite phase generates *map* tasks to sort the data based on its key and spills the output from the *mapper* side onto Spark local storage after data serialization and compression. The ShuffleRead phase generates *reduce* tasks to collect and aggregate data from all the *mappers*. In this phase, data is read from the Spark local storage and network, and then decompressed and deserialized. ShuffleRead is the phase where both disk and network are involved, and data redistribution among different data partitions occurs. In GATK4, the MD and BR stages are separated by the ShuffleWrite and ShuffleRead phases as shown in Fig. 1. Though shuffle-related operations also involve network communication, the 10Gbps network usually is not the bottleneck of Spark applications [5]. Hence, we mainly focus on the analysis and modeling of the I/O part.

2) Large RDDs NOT Cacheable in Memory: Another major source of I/O access comes from multiple references to large RDDs that consume a large amount of memory and cannot be cached in memory. One example is the UnionRDD markedReads in GATK4, which is used by both the BR and SF, as shown in Fig. 1. Since this RDD is not cached in the memory, each time the program uses and performs actions on it, it will be 1) read from the persist storage (Spark Local) if there is a persist write for this RDD, or otherwise 2) recomputed using input data from either the HDFS or Spark Local.

To explain why this UnionRDD cannot be cached in memory in GATK4, we change the original GATK4 to cache it for a small input (compressed, serialized data), and then measure its runtime memory consumption (decompressed, deserialized data). Based on this information, we find that caching this single UnionRDD for the whole genome with 122GB input requires around 870GB memory space. Assuming that around 40% of the entire Spark executor memory is used as storage memory to cache this UnionRDD, the total Spark memory required would be around 2.18TB. Each server that we use has 128GB RAM, and if we allocate 90GB for the Spark executor, we need 25 slave nodes in total, which is quite expensive for practical usage. Therefore, such large RDDs can not be fully cached in memory and need to be persisted in disk or simply be rebuilt from input anytime that a program uses them. Both of these approaches will incur a large amount of I/O access.

C. Effective I/O Bandwidth under Various Data Request Sizes

As previously analyzed, the big performance gap of different storage configurations is mainly caused by switching the Spark Local from HDD to SSD. We find that the effective I/O bandwidth in the shuffle read stage has up to a 32x gap for SSD and HDD, which is much larger than the 3.7x gap of their peak bandwidths. In this section we use 36 Spark executor cores and the whole genome input unless otherwise specified.

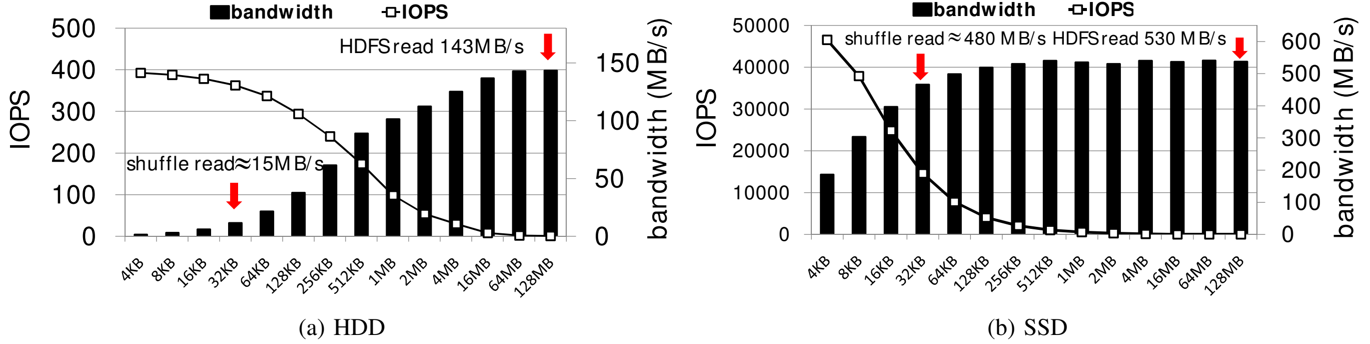
After a detailed analysis, we find that unlike the HDFS read/write that usually involves large data block accesses (e.g., 128MB), shuffle read incurs many small block size I/O accesses, where HDDs have a much lower effective bandwidth than SSDs. As a result, replacing a HDD with a SSD for the Spark Local for BR and SF stages can achieve up to 3.7x and 9.5x performance gains, respectively. In particular, we have the following observations and analysis.

1) Effective I/O Bandwidth on HDD and SSD: We use fio [21] to test the input/output operations-per-second (IOPS) and effective bandwidth on different read block sizes for HDD and SSD to simulate the shuffle read operation and HDFS read operation in Spark. As shown in Fig. 5a and 5b, when block size is 30KB, the average bandwidth is 15MB/s for HDD and 480MB/s for SSD, resulting a 32x bandwidth gap. For such a small data access size, shuffle read I/O in HDD becomes the bottleneck of the system, while shuffle read I/O in SSD does not. The bandwidth gap between HDD and SSD is higher when the block size gets smaller. When the block size is 4KB, the gap can be as high as 181x. When the block size is 128MB (default in HDFS block size), the gap is only around 3.7x.

2) Why Shuffle Read Accesses Small Data Blocks: Now we analyze why shuffle read involves numerous small data block accesses. As shown in Fig. 4, assume the mapper side has M partitions. There are M local output files that are indexed with all the reducer IDs. On the reducer side, assume there are R tasks. Each reducer reads shuffle data indexed with its own reducer ID from M separate files in the mapper side, and dynamically merges the data. To fit the data read by each reducer in memory for use in later RDD operations (e.g., unionRDD in GATK4), there is usually a fixed data size for each reducer (e.g., each reducer in GATK4 reads 27MB shuffle data). Since the fixed amount of data in each reducer comes from M mapper files, when M is large it will incur a large amount of small block size I/O accesses.

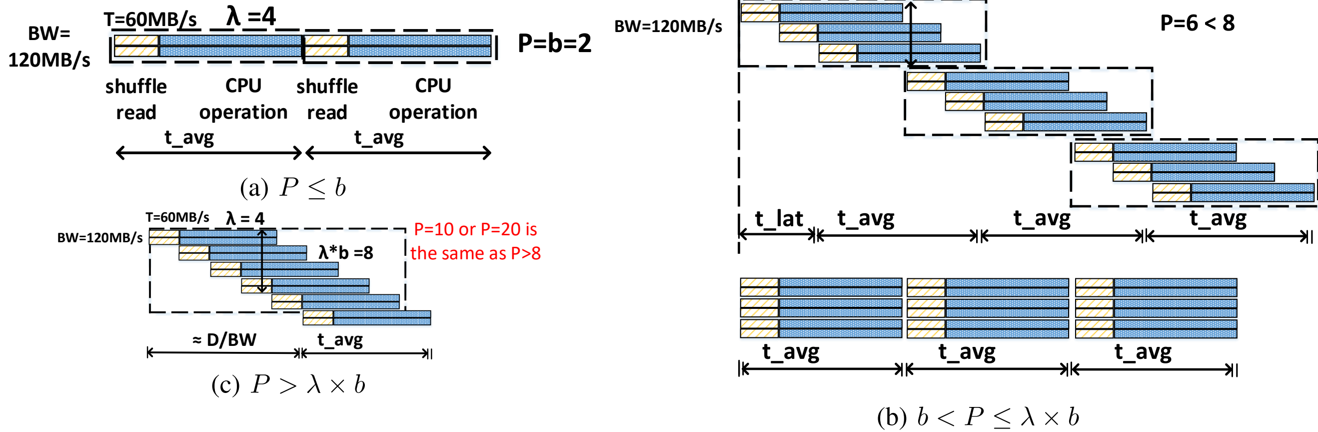
In GATK4, the number of mappers M is determined by the partition number of the RDD initialReads as shown in Fig. 1, i.e., the number of HDFS blocks of the input HDFS file. When the input is a whole genome, M = 122GB*1024(MB/GB) / 128(MB/HDFS block) = 973. The number of reducers R is set so that each reducer task reads in 27MB shuffle data as tuned in the original GATK4. On average, each reducer reads around 27MB / 973 = 30KB data from each mapper. We also use iostat to measure the average I/O request size (in sectors, 512B per sector) in the BR and SG; the average request size is 60, which corresponds to the 30KB ($\approx$512B$\times$60) block size.

3) Shuffle Performance Analysis: According to Fig. 5a and 5b, the HDD small block access bandwidth is only 15MB/s, which also matches the result of iostat. Hence, the time needed for shuffle read (334GB as in Table IV) is 334 * 1024(MB) / 3 / 15(MB/sec) / 60(sec/min) = 126mins, which perfectly matches the execution time of both BR and SF shown in Fig. 2. This further indicates that all computation time is overlapped by HDD access. One may notice in Table IV and Fig. 4, although the shuffle data size of MD is exactly the same as BR and SF, the execution time of MD is much shorter. That is because the block size of shuffle write is much larger than shuffle read—about 365MB in this case since mappers write data in sorted chunks (as described in Section III-B1).



(a) HDD (b) SSD

Fig. 5: Read bandwidths and IOPs for HDD and SSD on different block sizes.



(a) $P \leq b$

(c) $P > \lambda \times b$

(b) $b < P \leq \lambda \times b$

Fig. 6: Execution model for (a) $P \leq b$, no I/O contention; (b) $b < P \leq \lambda \times b$, I/O contention is hidden by the CPU computation; and (c) $P > \lambda \times b$, I/O becomes a bottleneck, and increasing the number of CPU cores P does not help.

IV. I/O-Aware Spark Analytical Model

To reason and quantify the underlying behavior of Spark tasks with different RDD operations, we propose an I/O-aware analytical model that considers the effective I/O bandwidths under different data request sizes and different numbers of CPU cores, and the overlap between the CPU computation and I/O access. For illustration purposes, we use Spark shuffle read as an example to explain our model, which works similarly for other I/O or computation-intensive RDD operations.

A. Model Variables Definition

Shown in Fig. 6a, we define the following variables used in our model.

1. T is the I/O (here, shuffle read) throughput per core when there is no I/O bandwidth contention. Usually we can test this T under the SSD configuration using a single core for the Spark executor. Our example in Figure 6 assumes T = 60MB/s for illustration purposes.
2. t_avg is the average execution time of a single task (for a single data partition). t_lat is the initial latency for pipelined batches of tasks, smaller than t_avg.
3. λ is the average time ratio of the entire task execution to the I/O access. Our example assumes $\lambda = 4$.
4. BW is the effective I/O bandwidth under the average data request size in the I/O operation. Our example assumes BW = 120MB/s.
5. $b = \frac{BW}{T}$ is the break point for the number of CPU cores per node, after which the CPU cores will contend for the limited I/O bandwidth. In the example shown in Fig. 6a, $b = 2$.
6. D is the entire data access size.
7. P is the number of actual launched executor cores per node.
8. N is the number of nodes.
9. M is the number of tasks (data partitions).

B. Different Execution Phases

When we gradually increase P from 1 to the number of maximum executor cores, there are three phases where the runtime model and I/O access are different.

$P \leq b$: no overlap with I/O and CPU computation. In this case, I/O is not a bottleneck as the number of executor cores does not exceed the bandwidth break point. As shown in Fig. 6a, after a batch of P tasks finish their execution, another batch of P tasks are launched. Here we only show two batches of tasks, and there is no overlap with I/O access (shuffle read) and CPU computation. Therefore, the estimated runtime formula is $\frac{M}{N*P} \times t_avg$.

$b < P \leq \lambda \times b$: overlap with I/O and CPU computation within a batch. As shown in Fig. 6b (right column of Fig. 6), P tasks are launched in a batch. Since b tasks already saturate the I/O bandwidth, the next b tasks start I/O operations after the first b tasks finish their I/O operation.[2] When $P \leq \lambda \times b$, the first b tasks in the second batch start the I/O operation (and

[2] In the real case, all P tasks are launched at the same time, where Fig. 6b draws equivalent sequential I/O access with no I/O contention for easy illustration.

also include some computation-like decompression) right after the first b tasks in the first batch finish, i.e., when the CPU cores are available. Here we show three batches of tasks. After an initial latency t_lat, each batch of tasks finishes in t_avg. Therefore, the estimated runtime formula is $\frac{M}{N*P} \times t_avg + t_lat$.

We find that in this case, the estimated runtime formula is almost the same as that of the $P \leq b$ case, since the CPU computation can hide the I/O access. *We conclude that the performance of parallel part (that is the left part of above formula) scales with the number of CPU cores P as long as $P \leq \lambda \times b$, where I/O does not hit the bandwidth break point or is hidden by the CPU computation.* We can define $\boldsymbol{B = \lambda \times b}$, as the turning point where I/O starts to become a bottleneck.

$\boldsymbol{P > \lambda \times b, i.e., P > B}$**: I/O becomes a bottleneck**. As shown in Fig. 6c, when P increases further, there is more overlapping of I/O and CPU operations. Since $\lambda \times b$ determines the maximum parallelism of CPU tasks, if P is larger than that, it means I/O becomes a bottleneck. In this case, the estimated runtime formula is $\frac{D}{N \times BW} + t_avg$, which is determined by the entire data access size and effective I/O bandwidth. *That is, further increasing the number of CPU cores P does not help the system performance when $P > B$.*

C. Generic Model

Therefore, our model can be generalized as follows: for each stage i, its runtime t_{stage} is:

$$\begin{aligned} t_{stage} &= \max\left(t_{scale}, t_{read_limit}, t_{write_limit}\right), \\ t_{scale} &= \frac{M}{N*P} \times t_avg + \delta_{scale} \\ t_{read_limit} &= \frac{D_{read}}{N*BW_{read}} + \delta_{read}, \\ t_{write_limit} &= \frac{D_{write}}{N*BW_{write}} + \delta_{write} \end{aligned} \quad (1)$$

Here t_{scale} is the execution time when none of the I/O read and write is a bottleneck (when $P < B$, t_{scale} is larger than t_{read_limit} and t_{write_limit}), thus $t_{stage} = t_{scale}$, and its parallel part scales with $N * P$. t_{read_limit} (t_{write_limit}) is the I/O read (write) time when it becomes a bottleneck, i.e., $P > B_{read}$ ($P > B_{write}$). We add a constant δ to each term to accommodate the linear part of the code. The model is built for each stage, and for the entire application, the total runtime is the sum of each stage's runtime, $t_{app} = \sum t_{stage}$.

Note that our model relates to disk bandwidth rather than disk number. Thus, it is general enough to support the multi-disk case. In addition, different hardware platforms or Spark configurations will lead to different t_avg. Therefore, our model can still correctly capture the execution time.

V. Model Evaluation Results

To demonstrate how to use our I/O-aware model to explain and predict the runtime behavior of Spark programs, we first apply it to GATK4 and validate the model accuracy. To better illustrate the generality of our model, we later apply it to typical big data applications from SparkBench [22] and BigDataBench [23]. Our experimental results demonstrate that our model can predict their performance with an error rate less than 10%, and can well explain their behaviors under different configurations. An **eleven-node** (one for master, ten for slave nodes) cluster is used for the experiments in this section. We report the average run time for five runs in the experiment results and also report error bars with positive and negative error values.

A. Applying Model to GATK4

1) MD stage: Changing the HDFS folder from an HDD to an SSD for the MD stage gives no performance gain when P = 36, as shown in Fig. 2. The HDFS read operation in MD only occupies a small portion of the task execution time. The time ratio of the entire task execution over I/O access $\lambda = 12$ is already pretty large. Although the break point b is different for HDD and SSD for HDFS read (4.3 and 16, respectively), B in both cases is larger than 36, the maximum number of executor cores per node in our setting.

When using SSD as Spark Local, the runtime is calculated as $t = t_{scale} = \frac{M}{N*P} \times t_avg + \delta_{scale}$. To be noted here, in Fig. 3 MD stage time does not scale for SSDs. This is not because the I/O is the bottleneck, it is because the garbage collection time increases with larger P and dominates the execution time of MD, which is currently not included in our model and will be dealt with in future work. When using HDD as Spark Local, shuffle write becomes the I/O bottleneck, BW_{write} = 100MB/s, $B = 10$, and runtime does not scale for P = 12, 24, 36.

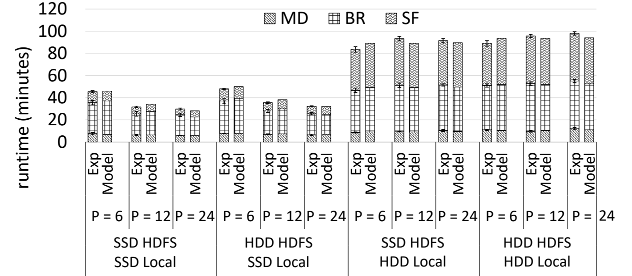
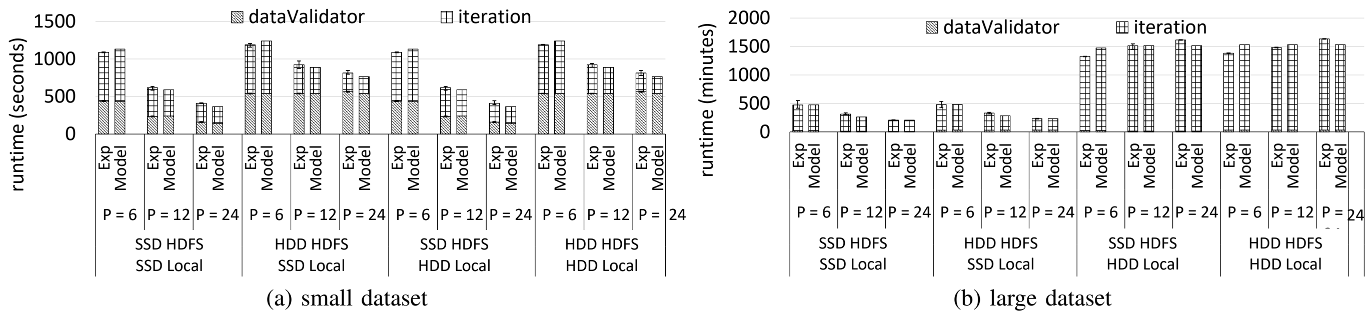


Fig. 7: Comparison of measured runtime (exp) and model predicted runtime (model) for GATK4 (max and min error bars are also shown).

2) BR and SF Stage: There are two kinds of tasks in the BR stage. One starts from the RDD nonPrimaryReads that need HDFS read, with a $\lambda = 1.3$; i.e., the CPU computation time is small compared to the I/O access time. However, since most read records are filtered out after the filter() function, as shown in Fig. 1, this task only occupies a small portion of the total BR execution time. The other task starts from the shuffle read, and the CPU computation time is long, with a $\lambda = 20$. And this task dominates the BR execution time. Due to space constraints, we will mainly illustrate the modeling for the latter task.

We first explain the case when both the Spark Local and HDFS are set to separate SSDs. For shuffle read, if there is no I/O contention, each core's read throughput T is around 60MB/s. And λ, the time ratio of the entire task execution over shuffle read task time, is 20. The SSD shuffle read bandwidth BW is around 480MB/s. Thus the break point $b = \frac{BW}{T} = 8$. In this way, the BR stage runtime scales with the number of executor cores P up to $B = 160$ cores. This matches well with the results in Fig. 3: when P increases from 12 to 24 to 36, the runtime of BR decreases accordingly.

However, when changing the Spark Local to an HDD, where HDD shuffle read bandwidth for 30KB block size is only 15MB/s, even one core suffers the I/O contention in some sense, that is, b=1. Compared to the shuffle read time in SSD,



(a) small dataset (b) large dataset

Fig. 8: Comparison of measured runtime (exp) and model predicted runtime (model) for Logistic Regression (LR).

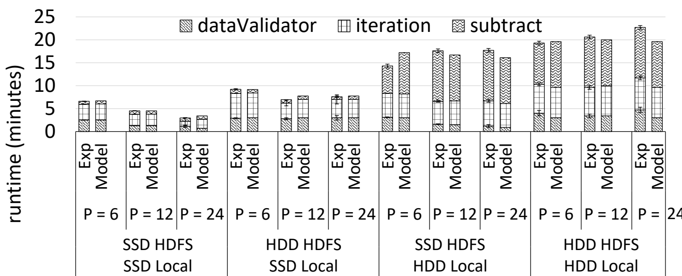


Fig. 9: Comparison of measured runtime (exp) and model predicted runtime (model) for Support Vector Machine (SVM).

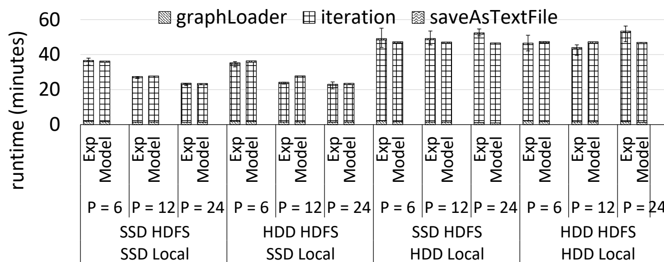


Fig. 10: Comparison of measured runtime (exp) and model predicted runtime (model) for PageRank (PR).

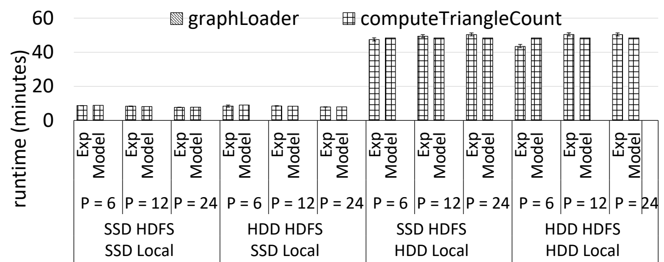


Fig. 11: Comparison of measured runtime (exp) and model predicted runtime (model) for Triangle Count (TC).

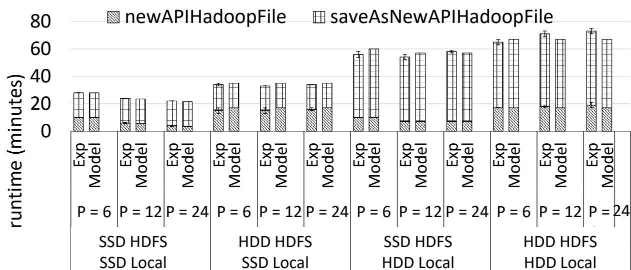


Fig. 12: Comparison of measured runtime (exp) and model predicted runtime (model) for Terasort (TS).

the shuffle read time in HDD in each core is 4x longer, which means that λ here is 5. Thus, $B = 5$; this means the runtime of BR does not further decrease when P is larger than 5. Based on the above analysis, there is no performance gap between the HDD and SSD when P is small. As $P > 5$, it further increases the performance gap between the two storage configurations on BR, as seen in Fig. 3. This also explains how the SF runtime scales with P. Since in SF λ is smaller, the performance gap when changing Spark Local from SSD to HDD starts even earlier than the BR.

3) Model Accuracy Results for GATK4: Fig. 7 presents the real measurements compared to model predicted runtime for different stages under different I/O configurations, when there are ten slave nodes, and P= 6, 12, 24. The average error rate is less than 6%, which is true for other omitted cases as well.

B. Generality of Our Model: Other Applications

1) Logistic Regression: Logistic Regression in Spark Mllib [24] is a typical iterative machine learning algorithm. It consists of two stages: dataValidator and iteration. In our experiment, We take two datasets generated by SparkBench: 1,200 (small) and 4,000 (larger) million examples, each with 20 features. The iteration number is set to 50 in this experiment. For the small dataset, the RDD parsedData generated from dataValidator can be cached in memory. For the large dataset, it is too large to be totally cached in memory and will be put in Spark Local. The sizes of RDD parsedData for the small and large datasets are 280GB and 990GB respectively. Results are shown in Fig. 8 with an average error rate of 5.3%.

2) Support Vector Machine: Support Vector Machine [25] is another typical iterative machine learning algorithm. It consists of three phases: dataValidator, iteration and subtract. Input dataset has 12 million samples, 1000 features, 1200 partitions. Iteration number is set to 10, and each iteration reads in 82GB in-memory cached RDD generated from dataValidator. The subtract phase incurs shuffle, and total shuffle size is 170GB. Results are shown in Fig. 9 with an average error rate of 8.4%.

3) PageRank: PageRank [26] in Spark GraphX [27] is an iterative graph algorithm that ranks the relative importance of webpages. It consists of three phases: graphLoader, iteration, and saveAsTextFile. We generate a dataset that has 20 million vertices, 4800 partitions (other data generator parameters are set as default). Iteration number is set to 10, and each iteration reads in cached RDD data from the last iteration and generates new RDD data for the next iteration to compute. The cached RDD total size is as large as 420GB, and it is larger than total executor storage memory space (assume 40% total executor memory is for storage) and persist in Spark Local. Results are shown in Fig. 10 with an average error rate of 5.2%.

4) Triangle Count: Triangle Count [28, 29] in Spark GraphX is a graph algorithm to count three-vertex small graphs within a large graph. It consists of two phases: graphLoader, computeTriangleCount. The computeTriangleCount phase will first repartition the graph to canonicalize [30] the graph so that there are no self loops or duplicated edges and all edges are oriented, and then compute the triangle count. We generate a dataset that has 1 million vertices, 2400 partitions. ComputeTriangleCount phase incurs 49GB in-memory cached RDD and 396GB total shuffle data. Results are shown in Fig. 11 with an average error rate of 3.6%.

5) Terasort: Terasort in Spark is another shuffle-heavy algorithm. There are two stages in Terasort: newAPIHadoopFile (NF) and saveAsNewAPIHadoopFile (SF). In the NF stage, input records are read from HDFS and sorted by ranges, and then the shuffle data is written to Spark Local. In the SF stage, each partition reads in the shuffle data that belongs to its range, sorts the record by key within the range and writes the output to HDFS. We take one example dataset generated by SparkBench: it has 10 billion records, with a total size of 930GB data. Results are shown in Fig. 12 with an average error rate of 3.9%.

Summary: For iterative algorithms, when dataset is small and cached in memory, runtime difference between HDD and SSD comes from HDFS read (write), and can be as large as 2x in LR (Fig 8a). When dataset is large and persist on disk, runtime difference mainly comes from persist read (write) on Spark Local in each iteration, as shown in iteration phases for LR (7.0x in Fig 8b) and PR (2.2x in Fig 10). For iterative algorithms with shuffle phase and shuffle-heavy algorithms like Terasort, runtime difference between using HDD and SSD as Spark Local can be modeled as shown in subtract phases in SVM (6.2x in Fig 9), TC (6.5x in Fig 11), and Terasort (2.6x in Fig 12). In summary, our model enables users to quantitatively analyze and understand application runtimes on in-memory computing frameworks like Apache Spark.

VI. Application of The Performance Model—A Case Study for Cost Optimization in Public Cloud

As reported by Broad Institute in 2017, 17 TB of new genome data is generated per day, and in total 45PB of data is managed. Moreover, according to [31], with the advancement of DNA sequencing, it is estimated that 20 exabytes of genome data will be produced every year by 2025. Huge data in genome analysis requires enormous CPU, memory, and I/O resources. Consequently, private institutions may not be able to afford the cost. Public cloud providers, e.g., Google Cloud, Amazon EC2 and Microsoft Azure, offer abundant CPU and associated memory and disk I/O resources that users may request. However, to process 20 exabytes genome data in Google Cloud means about 1.6×10^{10} CPU hours in GATK4, which is about 0.53 billion dollars for CPU cost only. Moreover, users have to pay for the requested I/O resources as well. Cloud providers support different disk I/O options. While SSD offers a much higher bandwidth compared to HDD, it is charged at a much higher price (4.2x in Table V). An important question naturally arises from such observations: *In a public cloud, how does one effectively find the optimal configuration to minimize cost for its required workload?*

This is not a trivial question. While a higher configuration can deliver shorter execution time, the cost per time unit is increased. On the other hand, although adopting a lower configuration guarantees lower cost per time unit, the total execution time rises. Hence, a balance needs to be discovered.

TABLE V: Disk price in Google Cloud platform

Type	Price (per **GB/month**)
Standard provisioned space	$0.040
SSD provisioned space	$0.170

1) Cost Modeling for HDDs and SSDs: With GATK4 as an example, we demonstrate that with our I/O-aware analytic model, users can quickly find the optimal hardware configuration from a large set of configurations in a public cloud to save on cost. In this section, all the results we report are genome sequencing with 500 million read pairs, as described in Section II-C.

In Google Cloud, users can specify the CPU instance with a different number of virtual CPU cores. The disks are also virtualized in Google Cloud. According to their datasheet [32], the virtual disk bandwidth is related to its configured size. Size and type (e.g., HDD or SSD) are the determinant factors of the disk price.

First, we do **one-time disk profiling** per data center. We create lookup tables for HDD and SSD persistent disk under different sizes. The read bandwidths for different request sizes of HDD and SSD can be found in [14].

For each application, we can perform four profiling runs to get the model variables usually under a small number of nodes N (e.g., $N = 3$) based on our models described in Equation 1. For t_{scale}, M can be obtained from one profiling run. t_avg and δ_{scale} can not be measured directly from profiling run. But we can choose two different P, measure corresponding t_{scale}, and calculate t_avg and δ_{scale}. The details are first and second sample run as the followings. And to measure BW_{read} and BW_{write} in t_{read_limit} and t_{write_limit} for read/write operations on HDFS and Spark Local separately, we need two additional sample runs, which are the third and fourth sample run as the followings. After four sample runs, we can obtain all constants in Equation 1.

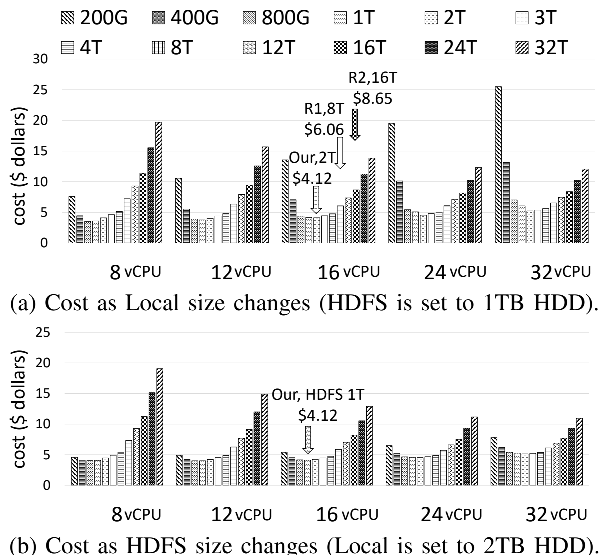
1. In the first sample run, we set P = 1, and set one SSD disk for HDFS and one for Spark Local. Both are sized at 500GB. This setting is chosen because we want to get the runtime when I/O is not the system bottleneck; that is, $P < B_{read}$ and $P < B_{write}$. Runtime for each stage t_{stage} is logged. And for each stage, we can get the number of tasks M, and data access size D_{read} and D_{write}. At the same time, iostat is used to log the average I/O request sizes RS_{read}, RS_{write} to look up the effective bandwidths BW_{read}, BW_{write}. Then we perform a sanity check that $t_{stage} > \frac{D}{N*BW}$; that is, I/O is not the bottleneck.
2. In the second sample run, we set P = 2, and set one SSD disk (500GB) for HDFS and one for Spark Local. Similarly, a sanity check that I/O is not the bottleneck is also done. Together with the first sample run, we can derive the δ_{scale} and the average execution time of a single task t_avg. δ_{scale} characterizes the serial execution time that can not be parallelized.
3. In the third sample run, we set P = 16 (according to [33], performance is more predictable for an instance with at least 16 vCPUs), and set one HDD (200GB) for Spark Local, and one SSD (500GB) for HDFS. This setting is chosen to get the runtime when I/O can be a bottleneck for I/O-related operations that need to read/write from/to Spark Local. For each stage, iostat is used to log the

I/O request sizes RS_{read}, RS_{write} to look up effective bandwidths BW_{read}, BW_{write} for Spark Local. Constant δ_{read} and δ_{write} are also calculated.

4. In the fourth sample run, we also set P as 16, and set one HDD (200GB) for HDFS, and one SSD (500GB) for Spark Local. This sample run is similar to the third sample run, except that I/O can be a bottleneck on HDFS read or write. Similarly, we can log the I/O request sizes RS_{read}, RS_{write} to look up effective bandwidths BW_{read}, BW_{write} for HDFS.

In the first sample run, the default SSD size is chosen at 500GB. If I/O is a bottleneck even for $P = 1$, we could double the requested SSD size and re-sample. For the third and fourth sample runs, the default HDD size is chosen at 200GB. If I/O is not a bottleneck even for $P = 16$, we could shrink the requested HDD size by half and re-sample.

After getting the model, the configuration selection problem is converted to minimize a discrete multivariate function *Cost* = *f (P, DiskTypes, DiskSize$_{HDFS}$, DiskSize$_{Spark_Local}$, Time)*. Here *P* denotes the core number per node and *Time* denotes the execution time which can be derived from our model. This optimization problem can be solved by the gradient descent method. We first consider *DiskTypes* as HDDs, and the optimal configuration that we get is when *P* = 16, *DiskSize$_{HDFS}$* = 1TB, *DiskSize$_{Spark_Local}$* = 2TB.



(a) Cost as Local size changes (HDFS is set to 1TB HDD).

(b) Cost as HDFS size changes (Local is set to 2TB HDD).

Fig. 13: Cost for using different sizes of HDDs

In order to give readers an idea of the trend of cost function, in Fig. 13a and Fig. 13b we present the cost results under *DiskSize$_{HDFS}$* = 1TB and *DiskSize$_{Spark_Local}$* = 2TB. There are two recommended hardware configurations for reference: R1 [12] hardware provisioning from the Apache Spark official website, suggesting 1:2 ratio of disks to CPU cores; R2 [13] hardware provisioning for Hadoop cluster from Cloudera, suggesting two hex-core machines with 12 disks (1TB), with 1:1 ratio of disks to CPU cores. If a 16 vCPU is used as a worker node, the estimated cost for R1 with 8TB disk is \$6.06, and for R2 with 16TB disk it is \$8.65. Interestingly, the estimated cost found by our model is \$4.12, which is 32% and 52% lower than R1 and R2 costs, respectively.

2) Model Verification on Google Cloud: True, the virtualized environment is much different than the physical one. However, as shown in Section IV, all of the model factors are only related to the system performance experienced at the user level, whether or not the underlying runtime is virtualized. That means the abstraction of our model is at user level—which is higher than the underlying system level. Therefore, our model can still work well in the cloud environment, and this is proved by the following experiment results.

Due to our limited Google Cloud credit, we verify our analytic model for using ten slave nodes, each with 16vCPU and 1TB as HDFS, while changing the HDD Local size. The measured runtime and predicted runtime from models are compared in Fig. 14. When HDD Local size increases from 200GB to 2TB, runtime decreases. After 2TB, runtime remains flat as expected. The average error rate is less than 4%.

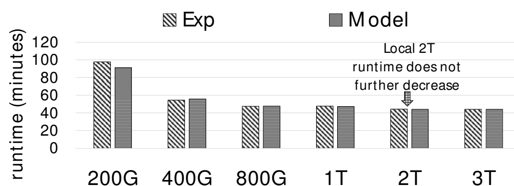


Fig. 14: 16vCPU: Comparison of measured runtime model (exp) and predicted runtime (model) for GATK4 when using different sizes HDD as Local (HDFS is set to 1TB HDD).

3) Cost Modeling for SSDs: Fig. 15 shows the case where SSD is used for Spark Local, with estimated cost and runtime for different numbers of executor cores P using different sizes SSD as Local (from 20GB to 3.2TB). The optimal cost of using SSD is less (\$3.75) which is another 1.1x as compared to \$4.12 using HDD as Spark Local. The measured runtime of using 200GB SSD as Spark Local is 43 mins, while the estimated runtime from the model is 45 mins (error rate 4.6%). Last, considering SSD as HDFS does not bring further cost savings. We omit the details due to space constraints.

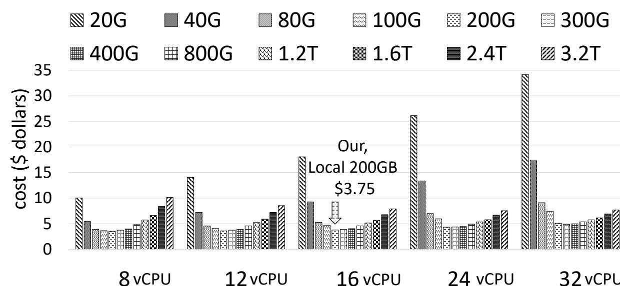


Fig. 15: Cost for using different sizes SSD as Local (HDFS is set to 1TB HDD).

4) Modeling Results: In conclusion, using 200GB SSD as Spark Local and 1TB HDD as HDFS achieves the cost-optimal hardware configuration for 16vCPU as a worker node. The cost is \$3.75, which is 38% and 57% lower than the cost of suggested configurations in R1 [12] and R2 [13] respectively.

VII. Related Work

A. Spark Performance Analysis and Modeling

K. Ousterhout et al. [5] use blocked time analysis to study the impact of network, I/O and stragglers on Spark

performance. In their paper, for the SQL workloads and the hardware setup they study, optimization on the network and I/O storage reduces the runtime by at most 2% and 19%, respectively. The conclusion on I/O part can also be explained by our model: 1) Average megabytes transferred to or from disk MB/s per node in their SQL workload is 10 MB/s (98MB/s in GATK4); 2) CPU:Disk Ratio in their cluster is 4:1 (18:1 in our cluster). By applying this number in Equation 1, I/O is not bottleneck in their application and cluster setup. On network part, A. Trivedi et al. [34] point out that moving from a 1Gbs to a 10Gbs network reduces the Spark runtime by up to 2.5x, and the network performance still matters. Others study Spark performance from an architecture perspective [35], NUMA [35, 36, 37], huge page [36], hyperthreading [35, 37], tuning JVM parameters and OS parameters [37], or from an application perspective [22].

E. Gianniti et al. [9] use Fluid Petri Nets to model the performance of MapReduce and Spark applications. It focuses on a scenario of the user-shared cluster, using previous execution information to study the distribution of the task time which is used for future prediction. Yet, our model can work in a much wider scenario, and the methodology we adopt is quite different than their statistical way. CherryPick [38] leverages Bayesian optimization and builds the non-parametric performance model. However, it picks a cost-saving configuration from a limited number (66) of predefined cloud configurations. Studies like Ernest [8] and [6] build analytic models to predict the Spark performance on iterative machine learning algorithms when there are more slave/worker nodes. However, in their models, the I/O impact on different data request sizes is not considered; this has a significant impact on performance, especially for the HDD case (as we studied in Section III-C and Section V-B).

B. Impact of I/O on Parallel and Distributed Computing

Work in [4, 39] studies how SSD and HDD impact the MapReduce performance. In [4] the runtime difference between SSDs and HDDs is compared: the number of SSD and HDD is matched as 1 to 11 on equivalent sequential I/O bandwidth. We do not match the number of disks as in [4] because the I/O bandwidth of SSDs and HDDs is not constant and varies significantly with application-requested I/O block size (as explained in Section III-C). Thus, matching on sequential I/O does not mean matching on random I/O. Other work like [40, 41] studies the SSD performance with its internal mechanism.

Work in [42, 43] characterizes I/O impact for HPC clusters and proposes job scheduling algorithms to optimize the system throughput among different applications. Opass [44] analyzes how remote and imbalanced read accesses impact system performance in distributed file systems and proposes optimization to maximize data locality and access balance. I. S. Choi et al. [45] leverage PCIe SSD to optimize the I/O performance of Spark. Work in [46] discusses how to scale Spark in HPC clusters where a global parallel file system is used instead of local disks. In summary, to the best of our knowledge, we are the first to propose an I/O-aware analytic model to quantitatively analyze and model the impact of I/O on applications running on top of the in-memory computing framework Spark.

VIII. Conclusion and Future Work

In this paper we observe that I/O can still play a heavy role—even in the in-memory cluster computing frameworks like Apache Spark. After a quantitative analysis, we find that the performance gap is mainly caused by a large number of (random) intermediate data accesses with small data blocks to the Spark local storage, where SSDs can achieve a much more effective bandwidth than HDDs. Such Spark local storage accesses are often used to avoid recomputation of time-consuming sort in shuffle operations, or persist large RDDs that consume a large amount of memory if cached. More importantly, we propose an I/O-aware analytical model to reason the performance of Spark programs, where it brings together the effective I/O bandwidth under different data access sizes and different numbers of CPU cores, and the overlap between the CPU computation and I/O accesses which has been often overlooked in past studies. Our proposed model can analytically explain and predict (within a 10% error rate) the runtime behavior of the production-quality genome analysis toolkit GATK4, and typical iterative algorithms that are computation-heavy, as well as typical shuffle-heavy algorithms. Finally, we demonstrate our model's usage by doing cost optimization in Google Cloud, which can quickly find the optimal hardware configuration in large exploration space and help us save 38% and 57% in cost for genome sequencing compared to two other recommended Spark cloud configurations.

GATK4 official release on January 2018 includes Burrows-Wheeler Aligner (BWA) and HaplotypeCaller (HC) in addition to MarkDuplicate (MD), BaseRecalibrator (BR) and SaveAsNewAPIHadoopFile (SF). The last three stages were the only stable stages available when the paper was submitted in the fall of 2017. We consider to include BWA and HC in our future work.

IX. Acknowledgment

This work is supported by Intel/NSF Innovation Transition (InTrans) Program (CCF-1436827) awarded to the Center for Domain-Specific Computing and by CDSC industrial partners Samsung and Google. The authors thank Adam Kiezun, Louis Bergelson, Geraldine Van der Auwera from Broad Institute for their help on using GATK4; Mulugeta Mammo, Yingqi Lu, George Powley, Eric Kaczmarek from Intel for their help on fixing core scaling issue with Scala; Anahita Shayesteh, Hingkwan Huen, Vijay Balakrishnan, Yang Seok Ki from Samsung for help on SSD profiling; Janice Wheeler for paper editing; Dharmesh Kakadia, Prof. Lizy Kurian John, Prof. Miryung Kim, Prof. Benjamin Lee, Prof. David Brooks for their advice. The authors also thank all peer reviewers in providing feedbacks and comments in the submissions.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, 2008.
[2] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.
[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, Berkeley, CA, USA, 2012.
[4] K. Kambatla and Y. Chen, "The truth about mapreduce performance on ssds," in *28th LISA*. USENIX Association, Nov. 2014.
[5] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *NSDI*. USENIX Association, May 2015.
[6] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," ser. HPCC-CSS-ICESS '15. IEEE Computer Society, 2015.

[7] A. J. Awan *et al.*, "How data volume affects spark based data analytics on a scale-up server," in *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer International Publishing, 2016.

[8] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Mar. 2016.

[9] E. Gianniti, A. M. Rizzi, E. Barbierato, M. Gribaudo, and D. Ardagna, "Fluid petri nets for the performance evaluation of mapreduce and spark applications," *SIGMETRICS Perform. Eval. Rev.*, vol. 44, no. 4, pp. 23–36, May 2017.

[10] B. Institute, "GATK," https://github.com/broadinstitute/gatk, 2017.

[11] A. E. Guttmacher, A. L. McGuire, B. Ponder, and K. Stefánsson, "Personalized genomic information: preparing for the future of genetic medicine," *Nature Reviews Genetics*, 2010.

[12] A. Spark, "Hardware Provisioning," http://spark.apache.org/docs/latest/hardware-provisioning.html, 2017.

[13] Cloudera, "How-to: Select the Right Hardware for Your New Hadoop Cluster," http://blog.cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster, 2013.

[14] "UCLA VAST Project Doppio," https://github.com/UCLA-VAST/Doppio, 2017.

[15] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *MSST*, May 2010.

[16] Spark, "Apache Spark Shuffle Operations," http://spark.apache.org/docs/latest/programming-guide.html#shuffle-operations, 2017.

[17] A. Spark, "Apache Spark Shuffle Operations Performance Impact," http://spark.apache.org/docs/latest/programming-guide.html#performance-impact, 2017.

[18] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin *et al.*, "The sequence alignment/map format and samtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[19] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä, E. Korpelainen, and K. Heljanko, "Hadoop-bam: directly manipulating next generation sequencing data in the cloud," *Bioinformatics*, vol. 28, no. 6, pp. 876–877, 2012.

[20] A. F. Gazdar, V. Kurvari, A. Virmani, L. Gollahon, M. Sakaguchi, M. Westerfield, D. Kodagoda, V. Stasny, H. T. Cunningham, I. I. Wistuba *et al.*, "Characterization of paired tumor and non-tumor cell lines established from patients with breast cancer," *International journal of cancer*, 1998.

[21] K. Salah, M. Al-Saba, M. Akhdhor, O. Shaaban, and M. Buhari, "Performance evaluation of popular cloud iaas providers," in *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*. IEEE, 2011, pp. 345–349.

[22] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark," ser. CF. ACM, 2015.

[23] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, 2014, pp. 488–499.

[24] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016.

[25] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, vol. 9, no. 3, pp. 293–300, 1999.

[26] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120.

[27] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '13. New York, NY, USA: ACM, 2013, pp. 2:1–2:6.

[28] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 607–614.

[29] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis, "Efficient triangle counting in large graphs via degree-based vertex partitioning," *Internet Mathematics*, vol. 8, no. 1-2, pp. 161–185, 2012.

[30] A. Spark, "Triangle Count in Spark," https://github.com/apache/spark/blob/master/graphx/src/main/scala/org/apache/spark/graphx/lib/TriangleCount.scala, 2017.

[31] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: astronomical or genomical?" *PLoS Biol*, vol. 13, no. 7, p. e1002195, 2015.

[32] G. C. Platform, "Storage Options," https://cloud.google.com/compute/docs/disks/, 2017.

[33] C. Delimitrou and C. Kozyrakis, "Hcloud: Resource-efficient provisioning in shared cloud systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 473–488.

[34] A. Trivedi, P. Stuedi, J. Pfefferle, R. Stoica, B. Metzler, I. Koltsidas, and N. Ioannou, "On the [ir]relevance of network performance for data processing," in *HotCloud*, Jun. 2016.

[35] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Architectural impact on performance of in-memory data analytics: Apache spark case study," *CoRR*, vol. abs/1604.08484, 2016.

[36] L. Wang, T. Xu, J. Wang, W. Zhang, X. Sui, and Y. Bao, "Understanding the behavior of spark workloads from linux kernel parameters perspective," in *17th International Middleware Conference*, ser. Middleware Posters and Demos '16, New York, NY, USA, 2016, pp. 1–2.

[37] T. Chiba and T. Onodera, "Workload characterization and optimization of tpc-h queries on apache spark," in *ISPASS*, April 2016.

[38] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017.

[39] S. Ahn and S. Park, "An analytical approach to evaluation of ssd effects under mapreduce workloads," *JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE*, vol. 15, no. 5, pp. 511–518, 2015.

[40] Z. Zhuang, S. Zhuk, H. Ramachandra, and B. Sridharan, "Designing ssd-friendly applications for better application performance and higher io efficiency," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, June 2016.

[41] C. Dirik and B. Jacob, "The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 279–289.

[42] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the i/o of hpc applications under congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 1013–1022.

[43] S. Herbein, D. H. Ahn, D. Lipari, T. R. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer, "Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 69–80.

[44] J. Yin, J. Wang, J. Zhou, T. Lukasiewicz, D. Huang, and J. Zhang, "Opass: Analysis and optimization of parallel data access on distributed file systems," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 623–632.

[45] I. S. Choi, W. Yang, and Y. S. Kee, "Early experience with optimizing i/o performance using high-performance ssds for in-memory cluster computing," in *2015 IEEE International Conference on Big Data (Big Data)*, Oct 2015, pp. 1073–1083.

[46] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling spark on hpc systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 97–110.