H5Spark: Bridging the I/O Gap between Spark and Scientific Data Formats on HPC Systems

Jialin Liu[1], Evan Racah[1],Quincey Koziol[1], Richard Shane Canon[1],
Alex Gittens[2], Lisa Gerhardt[1], Suren Byna[1], Mike F. Ringenburg[3], Prabhat[1].

***Abstract*— The Spark framework has been tremendously powerful for performing Big Data analytics in distributed data centers. However, using Spark to analyze large-scale scientific data on HPC systems has several challenges. For instance, parallel file systems are shared among all computing nodes, in contrast to shared-nothing architectures. Additionally, accessing data stored in commonly used scientific data formats, such as HDF5 and netCDF, is not natively supported in Spark. Our study focuses on improving I/O performance of Spark on HPC systems when reading and writing scientific data stored in HDF5/netCDF. We select several scientific use cases to drive the design of an efficient parallel I/O API for Spark on HPC systems, called H5Spark, which optimizes I/O performance and takes into account Lustre file system striping. We evaluate the performance of H5Spark on Cori, a Cray XC40 system located at NERSC.**

I. INTRODUCTION

Big Data analytics frameworks, like Spark [17], have been tremendously powerful in tackling big data problems and facilitating scientific knowledge discovery. The productive interface to distributed computation has largely reduced the development cycle of data analytics. For example, a word-count program only has 6 lines of code in Spark, while a Hadoop MapReduce implementation requires 28 lines of code [1]. Spark has been developed and optimized in commercial data centers, where the commodity hardware has a local disk attached to each compute node. Traditional HPC, on the other hand, favors the MPI programming model, which is more explicitly controllable by users in terms of parallel computing and concurrent I/O. HPC compute nodes and storage nodes are separated and typically use parallel file systems like Lustre or PVFS, instead of HDFS. Porting Spark onto Cray machines to enable efficient data analysis is an active area of exploration. When shifting this software stack onto the traditional HPC systems, the most challenging optimization points include network I/O, file I/O, file system access, and scalability. This study focuses on the I/O performance, and we highlight that reading and writing scientific data arrays is a major I/O task. Parallel HDF5 and netCDF are two common scientific data formats that are used to manage large amounts of data (e.g., many TBs), which are generated from scientific simulations or experiments.

Using the hierarchical data formats and the high performance parallel I/O APIs in these libraries scientists are able to efficiently conduct data analysis and manage millions of files. At NERSC, according to an annual workload analysis in 2014, HDF5 and netCDF are among the top 10 libraries on Edison (a Cray XC30), with about 750 unique users [2]. Loading these data formats, however, is not currently supported in Spark. The reasons include, but are not limited to the lack of an API in Spark to directly load or sub-select HDF5/netCDF datasets into its in-memory data structures, and the underlying Lustre file system is not well tuned for Spark I/O and vice versa. In order to address these limitations, translators that can map between HPC system and the Spark ecosystem are needed.

Users have been developing various conversion codes in order to fit the HDF5/netCDF data into Spark. A large amount of programming effort has been put towards dealing with the input/output and data conversion issues instead of the actual data analytics. One previous work, SciHadoop [4], has designed a hadoop plugin allowing scientists to specify logical queries over array-based data models, but is still based on HDFS and netCDF only. Another previous work, SciSpark, has been developed by NASA [15], in which a sciRDD data structure is defined to represent the netCDF data in memory. The SciSpark library has been only tested on distributed commodity machines with the data being located on openDap servers and manipulated on HDFS. Converting HDF5 files to Spark RDD has been proposed in a HDF5 blog [8], we conduct a more in-depth study and design in this work with an emphasis on high bandwidth parallel I/O on HPC system. In this study, we pick several scientific use cases to drive the design of an efficient parallel I/O API for scientific data formats in Spark, which we call H5Spark. One of the selected use cases is to conduct non-negative matrix factorization on thousands of Daya Bay nuclear datasets that have been converted to HDF5 formats. We efficiently load the datasets into Spark and form an RDD based on user required in-memory data structure, e.g., indexed row matrix. We optimize the I/O performance, taking into account Lustre file system striping. We also compare the performance of H5Spark with MPI-IO and SciSpark on the new Cori supercomputer (a Cray XC40) at NERSC.

II. MOTIVATION AND BACKGROUND

In a traditional HPC environment, users can explicitly request and have some level of control over the resources, e.g., memory allocation, file system tuning, and parallelism, etc. As we enter the data-intensive era, users are looking for more productive ways to analyze their data and answer their scientific questions. Spark is a distributed big data analytic

[1]NERSC, Lawrence Berkeley National Laboratory
[2]ICSI and Department of Statistics, UC Berkeley
[3]Cray, Inc.

framework, which started as an academia research project and now has become very popular in industry. Similar runtime systems have been considered in HPC for decades, but have been rejected due to performance issues. As a result, it is interesting to see how Spark's ecosystem could potentially speedup scientific discovery in HPC environments [5]. Spark relieves the users from complex resource management, scheduling, parallelism, and fault tolerance, etc. For example, Spark can easily parallelize the data loading, transforming and processing. A simple map(f) command can perform the function "f" on a large distributed dataset in an embarrassingly parallel way. Although the MPI programming interface is able to construct various communication patterns and support highly efficient parallel I/O, Spark is much easier to use in these kind of workloads (where data dependency is not an issue) in terms of productivity. On the other hand, HPC applications often rely on hierarchical data formats to organize files and datasets [9], [7]. The Daya Bay HDF5 files, for example, store each event as a row record, where each column records a different sensor's output. The climate data has four dimensions, which are time, elevation, latitude, and longitude, and each data point records the temperature at a specific time and pace. Consequently, simplifying access to scientific data formats like HDF and netCDF in Spark is essential to enabling scientists to more quickly exploit its capabilities.

Spark reads in the data as a resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on simultaneously. We have two ways to create RDDs: one is to parallelize an existing collection and the other is to reference a dataset in an external storage system, such as a shared data stores, like HDFS and HBase; this is based on the assumption that there is a corresponding Hadoop input format implementation for that format. HDF5 and netCDF, however, are not natively supported in Spark.

To use Spark in HPC, it is impractical to convert all existing scientific data formats into Spark-friendly data formats, like text file or columnar parquet formats that Spark has specific loading functionality for. For a text file, since it is a sequence of bytes, Spark can easily split the files into even sized blocks. For parquet formats, it has a row group, which is a logical horizontal partitioning of the data into rows. However, there is no physical structure for a row group. A row group consists of a column chunk for each column in the dataset. A column chunk is a contiguous data that lives in a particular row group. These columnar storage formats have its advantages in data compression and the cases when an aggregate needs to be computed over many rows but only for a notably smaller subset of all columns of data. However, row-oriented storage formats, like HDF5 and netCDF, are more common in HPC [6].

III. Challenges of Loading HDF5 in Spark

Scientific data formats often have a deep hierarchy, which cannot simply be treated as a sequence of bytes nor be evenly divided [12]. Spark needs extra effort to be able to interpret the data and file object. For example, as one of the major hierarchical data formats in the HPC community, HDF5, can have interleaved metadata and raw data in the file. Its efficient IO library is another benefit for loading data from parallel file systems. As such, we have the following list of questions that we will address in this paper:

- How do we transform an HDF5 dataset into an RDD?
- How do we utilize the HDF5 I/O libraries in Spark?
- How do we enable parallel I/O?
- What is the impact of Lustre striping?
- What is the effect of caching on IO in Spark?

We addressed these challenges by designing the H5Spark, which is a parallel I/O interface for loading HDF5 natively in Spark and forms various RDD structures based on the users' requirements.

IV. H5Spark Design

The H5Spark design is driven by several use cases from the NERSC user community. NERSC users typically have their data stored in Lustre, often in the form of HDF5 files. To use Spark, the need this data to be efficiently loaded from disk and converted into a format that Spark can efficiently work with in-memory (e.g. RDD files). Furthermore, they may need various in-memory formats of this data depending on the questions they are trying to explore. A common set of cases are from linear algebra parallel processing, e.g., principal component analysis(PCA) and non-negative matrix factorization(NMF), where vectors and matrices are the common data structures. This is also confirmed in the Spark ecosystem, that the 'matrix' is the most common data structure in Spark's machine learning library, MLlib [14]. The data structure in Spark MLlib includes:

- Vector
- Labeled Points
- RowMatrix
- IndexedRowMatrix
- CoordinateMatrix
- BlockMatrix

As shown in Figure 1, there are four major components in H5Spark: Metadata analyzer, RDD seeder, Hyperslab partitioner and RDD constructor. For loading a single HDF5 file, the H5Spark metadata analyzer takes the user's input file name and dataset name and triggers the first IO call to the file system to fetch the HDF5 file metadata information. The IO calls H5Fopen and H5Dopen are called to return the size of each dimension of the queried dataset. Note that the Spark partitions is an important parameter that can be set by the users to control the degree of parallelism. For example, if the user chooses 10 partitions, then there will be at most 10 parallel tasks to process the RDD. Therefore, it is essential for H5Spark to utilize partitions for parallelizing the I/O. This is achieved with the hyperslab partitioner by comparing the dataset's slowest dimension size with the number of Spark partitions. The basic idea is described in the following pseudocode (Algorithm 1).

H5Spark determines the hyperslab I/O region for each Spark task between line 1 and line 10. Starting from line 2,

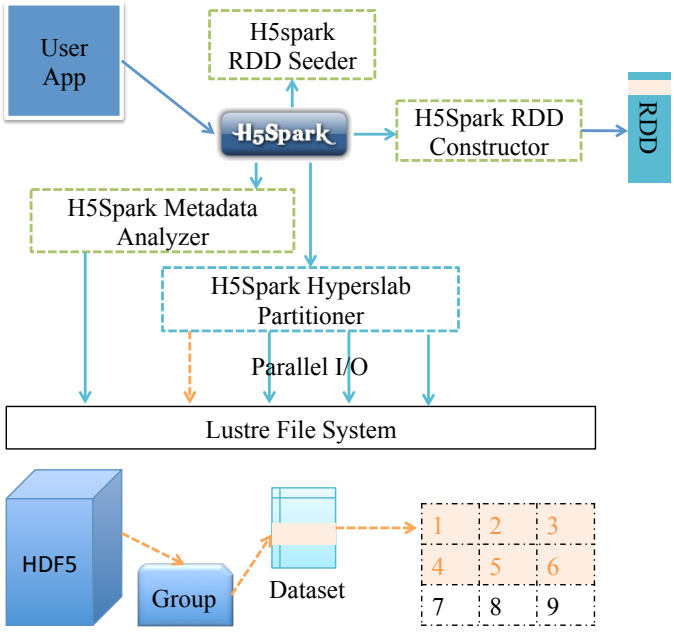

Fig. 1. An overview of H5Spark architecture

input : HDF5 File Path, f; Dataset Name, v; Spark Partition, p; Slowest Dimension ID, sid; SparkContext, sc, Dimension, dim
output: RDD, r

```
1  Hyperslab Partition;
2  if dim[sid] < p then
3  |  p = dim[sid];
4  end
5  step = dim[sid]/p;
6  i = 0;
7  while i < dim[sid] do
8  |  offset[i] = step * i;
9  |  i++
10 end
11 Rdd Seeding;
12 r_seed = sc.parallelize(offset, p);
13 Parallel I/O;
14 r = r_seed.flatmap(h5read(f, v));
15 RDD Construction;
16 r = h5transform(r);
```

Algorithm 1: From HDF5 to RDD

TABLE I
RDD CONSTRUCTOR IN H5SPARK

Function	Input	Output
h5read	sc, f, v, p	A RDD of double array
h5read_point	sc, f, v, p	A RDD of points
h5read_vec	sc, f, v, p	A RDD of vector
h5read_irow	sc, f, v, p	A RDD of indexed row
h5read_imat	sc, f, v, p	A RDD of indexed row matrix

if the size of the slowest dimension is less than the available partition, the number of partition will be adjusted to be the same as the size of the dimension. This assures that each Spark task will have a chunk to read, otherwise, there will be idle takes. Then line 7-9 evenly balances the I/O on the tasks. Each of the Spark tasks will be launched to do the I/O independently in line 14. Depending on the user's requirements, the returned RDD can be transformed to other structure, e.g., from double array to indexed row, from index row to indexed row matrix, etc.

To meet users' requirement and be consistent with the existing structure in Spark MLlib, we provide the following interface for RDD construction, Table I. Using these functions, users do not need handle all of the details that are described in Algorithm 1. Instead, simply by calling the h5_read_[option] and passing in the required arguments, the user can easily acquire the desired RDD object and continue to work on their data analytics jobs.

The major benefits of using H5Spark is that the Parallel I/O is transparently handled without users' interaction. Currently, H5Spark's IO is an MPI-like independent I/O, which means, each executor will issue the I/O independently without communicating with other executors. In the future, more advanced MPIIO features, like collective I/O, will be considered. A collective version of H5Spark IO, however, is difficult to implement and is not guaranteed to deliver high bandwidth. MPI collective IO's current implementation has two phases [16], [13]. In the case of collective read, the first phase is the I/O phase, where a selected group of processes act as I/O aggregators, to bring the data from disks into memory. Then in the second phase, which is the shuffle phase, all processes will communicate with the previous subset of processes(i.e., aggregators) to get the data. In other words, the data is re-distributed among all processes to its original desired pattern [10], [11]. Such two-phase design may not work in Spark. One reason is that there is no global synchronization among tasks in Spark, while collective I/O will at least require each process to participate in a synchronized communication to construct a global offset list. Another reason is that a lot of shuffle operations are involved in the collective I/O; however, shuffle is a costly operation in Spark. We are looking to utilize local (per worker) collective buffering to address this issue in the future.

V. H5SPARK EVALUATION

A. Experimental Setup and Datasets

In our experiments, we evaluate the H5Spark on the Cori Phase 1 system at NERSC, which is a Cray XC40 supercomputer. This cluster currently is equipped with 1600 compute nodes and 248 storage nodes. Each compute node has 32 cores with 128 GB RAM in total. The peak I/O bandwidth is 700GB/s.

One of the two datasets we used is a temperature dataset from climate science, which and is a 2D (6349676, 46715) dataset of double types. The total file size on disk is 2.2 TB. Another dataset is a 1.6 TB dataset from the Daya Bay neutrino experiment. It is originally stored as hundreds of

thousands of small HDF5 files, as they are extracted and converted from hundreds of thousands of ROOT files [3]. After combining all these small HDF5 files, we generate a single 1.6 TB large file.

B. How to Use H5Spark in Spark

Currently, we collect a few use cases from climate science, high energy physics and astronomy at NERSC. To use Spark, either Python or Scala/Java is recommended. As a first time user, we suggest to first download from the H5Spark git repository, i.e., https://github.com/valiantljk/h5spark.git, and then use a Scala/Java build tool, e.g., *sbt*, to build the H5Spark package. After that, the users can get a h5spark jar package in the 'target' directory. For using in Python/Scala codes, simply exporting the Python/Java classpath to where the H5Spark is, and then import the h5spark's read class, should have all its I/O interface available. Specifically, to use in PySpark, a simple example is:

```
from h5spark import read
from pyspark import SparkContext
sc = SparkContext()
rdd = h5read(sc,file_list_or_txt_file,
      mode='multi', partitions=2000)
```

In the above Python scripts, the user can use this h5read function to access a folder or a single HDF5 file and a single RDD will be returned which contains all the data from all input files. The users can then use the standard mechanisms in the Spark world to analyze and transform this data.

For using the H5Spark in Scala, the code will be like:

```
import org.nersc.io._
val rdd = read.h5read (sc,inputpath,
          dataset name, partition)
```

C. Profiling H5Spark I/O

The operations in Spark are lazy. For example, if we have an H5Spark read call, it will not be executed immediately. Only if an action is detected which requires reading the data, e.g., count, will Spark start to perform disk I/O. In order to profile H5Spark's I/O performance, we need to have at least one action, we used 'count' to count the number of rows in the returned RDD, and this setup is the same in all our tests.

With a count operation, we can guarantee that one H5Spark read call is able to bring all the data in memory once, which is necessary to measure the actual I/O bandwidth. But this still can not guarantee that the data are in memory at the same time, because there is no synchronization in spark's concurrent tasks, and any data block fetched by any task can be immediately thrown away after that task's count operation. It is not necessary to persist all the data in memory at the same time, as long as the users just want to perform some simple operations. But one major benefit of running iterative data analytics in Spark is to have in-memory data processing and avoid the costly disk I/O as in traditional Hadoop. Therefore, we force the data to be cached in memory after H5Spark's read and before the count operation. This means the I/O time we measured includes H5spark's read and Spark's cache and a count operation, among which the read and count are necessary, while the cache makes sense to the real user cases. Note that Spark's $sc.cache()$ (or $sc.persist()$) operation involves more cost than not caching.

D. Testing H5Spark with Lustre Striping

In Figure 2, we used 45 nodes and set the number partitions as 3000. We vary the Lustre striping counts, i.e., number of OSTs, to observe the I/O bandwidth. The 2.2 TB climate data is duplicated in different directories with different striping counts. As shown in the Figure 2, the number of OSTs we have tried is 1, 8, 24, 72, 144, 248. These striping counts are configured using the current striping recommendation commands on Cori. Users tend to follow our file system optimization suggestions to optimize their I/O. Cori's lustre file system sets the default striping to 1 for all users. For small files, we have a command called "stripe_small", which can set the striping count to 8, and correspondingly, 24 and 72 are the results of "stripe_medium" and "stripe_large". We also set the striping to be 144 and 248, which are about half and all of the storage bandwidth, respectively. We assume that those striping configurations are commonly used by our users.

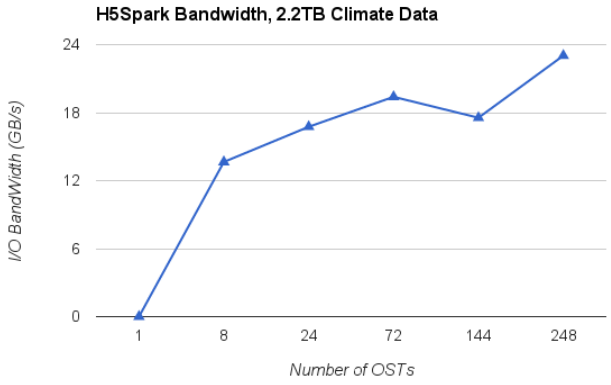

Fig. 2. H5Spark I/O Bandwidth with Lustre Striping

It is not surprising to find out that a single OST can not provide desired I/O bandwidth for the 2.2TB dataset. In our tests, H5Spark does not finish loading the 2.2TB data from this single OST, we tried with one hour wall time limit. We mark it as zero bandwidth in the figure. Then we immediately observe the increasing of I/O bandwidth as we increase the number of OSTs. For example, with 8 OSTS, we are able to load the whole 2.2TB data in 2.7 minutes with 13.65 GB/s bandwidth. And the maximum bandwidth is achieved at OST=248, which is 1.6 minutes of loading time for the 2.2 TB dataset.

In general, this is not surprising to HPC communities and traditional MPI/Lustre users. The striping effect is similar for both H5Spark and MPIIO. For H5Spark Scala version, the current implementation is equal to the MPI independent

I/O, where each executor reads a chunk/hyperslab from the HDF5 file. If the striping is not set well, e.g., only one OST is used, then all executors' tasks will be issued to the same storage node, which will cause huge I/O contention on server side. Spark is sensitive to these contentions (or delays) and after certain amounts of waiting time, the Spark driver may re-launch the tasks (which will continue to block on the I/O queue), and eventually will fail the tasks.

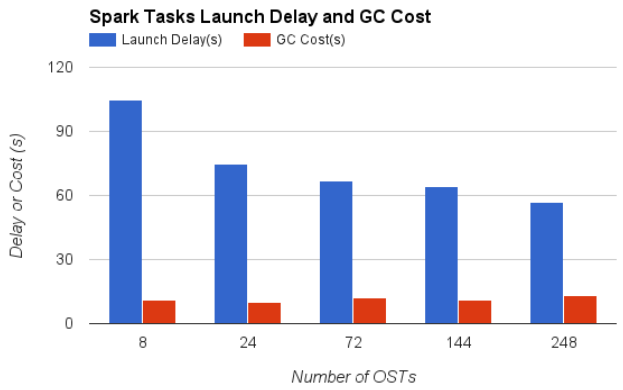

Fig. 3. H5Spark Launching Delay

We also profiled the I/O Cost in details in H5Spark. Spark uses a task scheduler to allocate the resources for the tasks. The resources are designed to be the CPU and memory only, according to Spark's scheduling implementation, but we observed some weird situations where disk I/O can also be a resource bottleneck in scheduling. As shown in Figure 3, with fewer OSTs, the tasks have longer launching delay. For example, with 8 OST, 95% of the tasks are launched at t_0, while the remaining 5% tasks are launched 1 minute later than t_0. This increased the overall I/O cost and largely reduced the I/O bandwidth. This launching delay reduction scales as we increase the number of OSTs. We currently suggest that users distribute their data on more OSTs to reduce such launching delay. We believe that storage resources are also important in the Spark scheduling on HPC. We will investigate in the future on a storage-aware H5Spark scheduling to complement the Spark's existing scheduler (where only CPU and memory are considered).

The garbage collection cost is constant in these different striping configurations. This is because the number of nodes, the total memory capacity, the way how H5Spark allocates the buffer, as well as the JVM garbage collection mechanism are all same in these tests. Storage(i.e., lustre striping) is not a factor in affecting the GC cost in Spark.

E. H5Spark Scaling

As we mentioned before, the number of Spark partitions determined the degree of parallelism. We used partitions equal to 3000 in the previous evaluation, and that is around twice of the number of available cores, i.e., 1440 (45 worker nodes, each has 32 executor cores). As shown in Figure 4, we did not see expected scalability as we increase the number

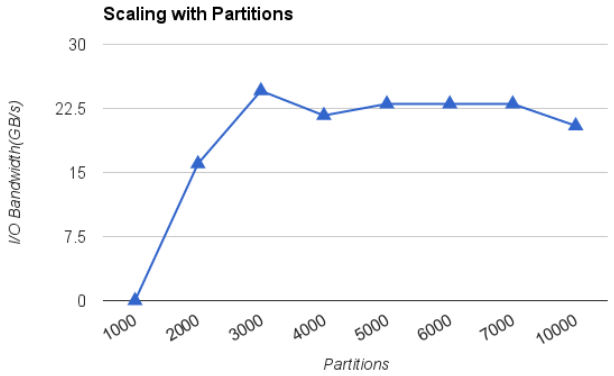

Fig. 4. H5Spark Scaling with Partitions

of partitions. For example, from partition 3000 to partition 10000, the I/O performance is about the same, which means this workflow is already I/O dominated at partition 3000. With more partitions, however, the I/O would not be scaled anymore. Instead, more partitions bring more scheduling overhead to Spark driver, which causes performance degradation. With fewer partitions, e.g., 1000, the job fails. This is due to two possible reasons. One is that if the number of partition is too small, then each executor needs to read a very large chunk of data from disk, which is relatively slow. The other reason is that the fewer the partitions, the more cores are idle in Spark.

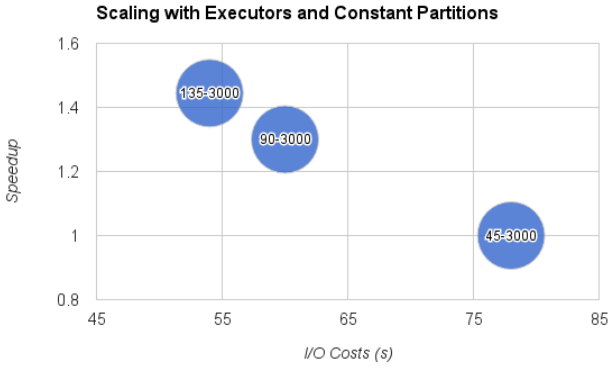

Fig. 5. H5Spark Scaling with Executors

Because of the lack of scaling from increasing the partitions, we tried to increase the actual total number of physical cores, while fixing the number of partitions. We used 45, 90 and 135 executor nodes , all with 3000 partitions and the data stored on 72 OSTs. In Figure 5, the I/O achieved 1.3X speedup when doubling the number of physical cores, i.e., 90 nodes with 3000 partitions, but higher concurrency did not result in linear scaling, e.g., with three times of nodes, the I/O performance only gets 1.4x speedup. For the three tests, we found that the number of physical cores in test 1 is only $45 \times 32 = 1440$, in which each core will need to handle $3000/1440 = 2$ tasks, while in test 2, we have

$90 \times 32 = 2880$ cores, in which each core can roughly handle 1 task. Therefore, the difference between test 1 and test 2 is reasonable. In test 3, we have $135 \times 32 = 4320$ cores, and each core can handle one task with no problems, however, 1320 cores are actually idle. This suggests that we increase both of the partitions and the number of executors. Which is confirmed in the following test.

Fig. 6. H5Spark Scaling with both Executors and Partitions

In Figure 6, we increased the number of partitions along with the number of executor nodes. In this test, we run three tests on the same data but with 144 OSTs. The first test has 45 executor nodes and 3000 partitions, the second one has 90 executor nodes with 6000 partitions, and the last one has 135 executor nodes with 9000 partitions. The last one achieved much better performance compared with the previous tests. The last two tests gets 2.73X and 3.24X speedup, which is a linear speedup. Therefore, among all the tests, the best result we get so far is shown in the Table II. Also, we included the

TABLE II
H5SPARK'S BEST PERFORMANCE OBSERVATION AND HERO RUN ON CORI, APRIL 8 2016

Size(TB)	I/O(s)	B/W(GB/s)	OSTs	Executors	Partition
2.2	37	59.7	144	135	9000
16	120	136.5	144	1522	52100

I/O number of a hero run on Cori, in which H5Spark was used as an essential plugin in loading the data. That hero run reserved all nodes on Cori and successfully loaded 16 TBs data in 2 minutes.

F. H5Spark Python vs Scala Version

Since we implemented two versions of H5Spark, Python and Scala, we are interested in comparing their performance. The dataset we used is the 2.2 TB climate data on 24 OSTS. We found that the Scala version is 1.8 times faster than the Python version. This is because in PySpark, the JVM has to spawn a Python process to run the Python scripts, while with Scala version, all processes and threads run in the same JVM without creating additional process space and generating additional copy between JVM and Python process. We also observed the difference of RDD in-memory size. In the Scala version, the cached RDD equals the raw data, while in Python, the cached RDD is only 479 GB (Spark history server shows it is 100% fully cached), which is only $1/4.61$ of original size. This might be due to the default serialization process used in Python, i.e., pickle.

TABLE III
COMPARISON OF H5SPARK PYTHON AND SCALA VERSION

Version	I/O(s)	B/W(GB/s)	Speedup	Mem(GB)	Ratio
Python	162	13.65	1	479	1
Scala	90	24.56	1.8	2210	4.61

G. H5Spark vs MPI-IO

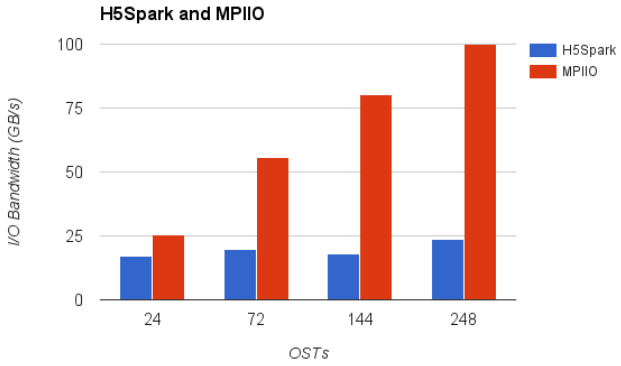

Fig. 7. H5Spark vs MPI-IO, Scaling with Storage Nodes

We varied the number of OSTs and compared MPI-IO's independent I/O with H5Spark on a 2.2 TB temperature dataset. As shown in Figure 7, MPI scales better than H5Spark when increasing storage nodes, which confirmed with our previous observation, that storage resources are not well considered in the spark tasks scheduling. The performance gap between H5Spark and MPI in this test is 1.48X to 4.25X.

We then varied the number of nodes and conducted a strong scaling test with the same 2.2TB dataset on 248 OSTS(the maximum number of OSTs on Cori). The Figure 8 shows that H5Spark demonstrates a linear speedup while MPIIO only achieves a sub-linear speedup. However, MPIIO still outperforms H5Spark with 4.25X, 2.35X and 2.45X speedup. The MPIIO's sub-linear speedup is due to its saturated bandwidth under this configuration. H5Spark's performance is encouraging in a sense that it only takes half minute to bring in 2.2TB data, when MPIIO needs 15 seconds.

H. H5Spark vs SciSpark

The last evaluation we did was to compare the existing SciSpark with our H5Spark. SciSpark provides interfaces for loading netCDF files, so to have a fair comparison, we used

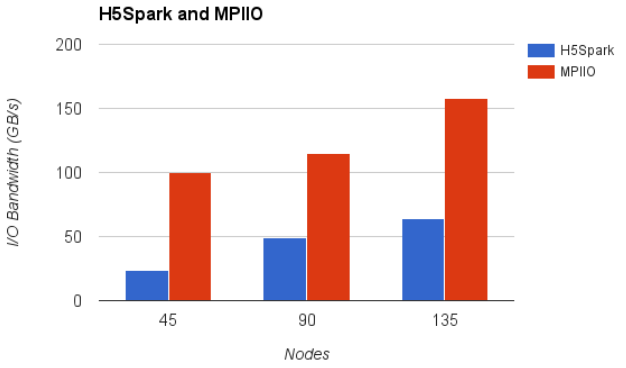

Fig. 8. H5Spark vs MPI-IO, Scaling with Compute Nodes

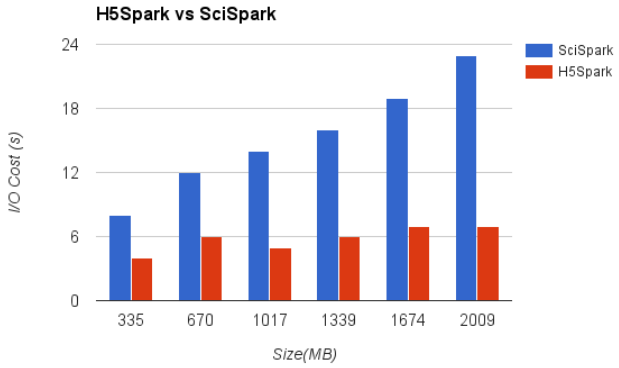

Fig. 9. H5Spark vs SciSpark

the netCDF4 format in SciSpark and HDF5 file formats in H5Spark, which will ensure the underlying format is same. We tested on a single TB of data and then a 51 GB file. SciSpark returns out-of-memory error in both cases. We checked the source code of SciSpark (the version before March 2016, new codes have been pushed in after that), we found that SciSpark used Spark's 'binaryFile' function to load all data into memory and form a RDD. This function, however, assumes that the data is a Hadoop-readable dataset, and splittable. This function requires input to be like:

- hdfs:hdfs-pathpart-00000
- hdfs:hdfs-pathpart-00001
- ...
- hdfs:hdfs-pathpart-nnnnn

Then by calling this binaryFile function, the returned RDD is like:

- (hdfs-path/part-00000, its content)
- (hdfs-path/part-00001, its content)
- ...
- (hdfs-path/part-nnnnn, its content)

Given a single large HDF5/netCDF file, however, this does not trigger the parallel I/O to the Lustre file system. We were able to test on several small files, as shown in Figure 9. The results confirm that SciSpark performs serial I/O to the file system, while H5Spark can fetch the data by chunk or hyperslab in parallel.

Note that both SciSpark and H5Spark are being actively developed. This comparison does not reflect any new and future changes.

VI. CONCLUSION

In this work, we designed H5Spark, an efficient HDF5 file loader for Spark. We implemented an MPI-like independent I/O in H5Spark and utilized the Spark's parallelism to automatically handle the I/O without much of users' interference, and such that the users can just focus on the data analysis. We evaluated H5Spark extensively, observing linear scaling when increasing the number of OSTs, and confirming that storage is also an important resource in Spark's scheduling on an HPC environment. We also achieved high I/O bandwidth by increasing the number of partitions along with the number of executor nodes. This is a valuable lesson for users to scale their codes. At last, we compared H5Spark with MPIIO and SciSpark, and found that in terms of I/O bandwidth, H5Spark gets closer to MPIIO (2X gap) and performs better than SciSpark. In the future, we would like to investigate more details in Spark's internal scheduling, and consider a storage-aware scheduling algorithm in H5Spark, we would also like to implement a collective IO mechanism in H5Spark. H5Spark's parallel write and loading balancing functionality are still under development.

ACKNOWLEDGMENT

During the development of H5Spark, we received help from Jey Kottaalam in understanding the PySpark and the spark runtime. We also thank the SciSpark team, we get help and quick response from Rahul Palamuttam, Chris Mattmann, Brian Wilson, and Renato Marroqun Mogrovejo, etc. We would like to thank Douglas Jacobsen at NERSC, Doug helps us in debugging and fixing the memory issue when running H5Spark on Cori. We thank HDF group for consulting on the HDF5 java library. We thank our users and reviewers for providing useful feedback and comments.

REFERENCES

[1] Introduction to spark, http://blog.cloudera.com/blog/2014/03/apache-spark-a-delight-for-developers/.

[2] Brian Austin. NERSC 2014 workload analysis, http://portal.nersc.gov/project/mpccc/baustin/nersc_2014_workload_analysis_v1.1.pdf, 2014.

[3] Rene Brun and Fons Rademakers. Rootan object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1):81–86, 1997.

[4] Joe B. Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott A. Brandt. Scihadoop: array-based query processing in hadoop. In Scott Lathrop, Jim Costa, and William Kramer, editors, *SC*, page 66. ACM, 2011.

[5] N. Chaimov, A. Malony, S. Canon, K. Ibrahim, C. Iancu, and J. Srinivasan. Scaling spark on hpc systems. In *The 25th International ACM Symposium on High-Performance Parallel and Distributed Computing*, 2016. in publication.

[6] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the HDF5 technology suite and its applications. In Peter Baumann, Bill Howe, Kjell Orsborn, and Silvia Stefanova, editors, *EDBT/ICDT Array Databases Workshop*, pages 36–47. ACM, 2011.

[7] Jim Gray, David T. Liu, María A. Nieto-Santisteban, Alexander S. Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

[8] Gerd Heber. From hdf5 to spark rdd, https://hdfgroup.org/wp/2015/03/from-hdf5-datasets-to-apache-spark-rdds/, 2015.

[9] Mark Howison, Quincey Koziol, David Knaak, John Mainzer, and John Shalf. Tuning HDF5 for Lustre File Systems. In *Proceedings of 2010 Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10)*, Heraklion, Crete, Greece, September 2010. LBNL-4803E.

[10] Wei keng Liao and Alok Choudhary. Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In *SC'08 USB Key*. ACM/IEEE, Austin, TX, November 2008.

[11] Wei keng Liao, Kenin Coloma, Alok N. Choudhary, Lee Ward, Eric Russell, and Sonja Tideman. Collective caching: application-aware client-side file caching. In *HPDC*, pages 81–90. IEEE, 2005.

[12] Jialin Liu, Brad Crysler, Yin Lu, and Yong Chen. Locality-driven high-level i/o aggregation for processing scientific datasets. In *2013 IEEE International Conference on Big Data*, pages 103–111, Oct 2013.

[13] Jialin Liu, Yu Zhuang, and Yong Chen. Hierarchical collective i/o scheduling for high-performance computing. *Big Data Research*, 2(3):117 – 126, 2015. Big Data, Analytics, and High-Performance Computing.

[14] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D.B. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *arXiv preprint arXiv:1505.06807*, 2015.

[15] R. Palamuttam, R. M. Mogrovejo, C. Mattmann, B. Wilson, K. Whitehall, R. Verma, L. McGibbney, and P. Ramirez. Scispark: Applying in-memory distributed computing to weather event detection and tracking. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 2020–2026, Oct 2015.

[16] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective i/o in romio. In *Frontiers of Massively Parallel Computation, 1999. Frontiers '99. The Seventh Symposium on the*, pages 182–189, Feb 1999.

[17] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.