FERMILAB-PUB-17-078-CD

Spark and HPC for High Energy Physics Data Analyses

Saba Sehrish
Fermi National Accelerator Laboratory
P.O.BOX 500
Batavia, IL 60510
Email: ssehrish@fnal.gov

Jim Kowalkowski
Fermi National Accelerator Laboratory
P.O.BOX 500
Batavia, IL 60510
Email: jbk@fnal.gov

***Abstract*—A full High Energy Physics (HEP) data analysis is divided into multiple data reduction phases. Processing within these phases is extremely time consuming, therefore intermediate results are stored in files held in mass storage systems and referenced as part of large datasets. This processing model limits what can be done with interactive data analytics. Growth in size and complexity of experimental datasets, along with emerging big data tools are beginning to cause changes to the traditional ways of doing data analyses. Use of big data tools for HEP analysis looks promising, mainly because extremely large HEP datasets can be represented and held in memory across a system, and accessed interactively by encoding an analysis using high-level programming abstractions. The mainstream tools, however, are not designed for scientific computing or for exploiting the available HPC platform features. We use an example from the Compact Muon Solenoid (CMS) experiment at the Large Hadron Collider (LHC) in Geneva, Switzerland. The LHC is the highest energy particle collider in the world. Our use case focuses on searching for new types of elementary particles explaining Dark Matter in the universe. We use HDF5 as our input data format, and Spark to implement the use case. We show the benefits and limitations of using Spark with HDF5 on Edison at NERSC.**

I. Introduction

Experimental High Energy Physics (HEP) deals with the understanding of fundamental particles and the interactions between them. Experimental HEP is a compute- and data-intensive statistical science; a large number of interactions must be analyzed to discover new particles or to measure the properties of known particles. For example, data from over 300 trillion (3×10^{14}) proton-proton collisions at the Large Hadron Collider (LHC) were analyzed for the Higgs boson discovery [14]. Future HEP experiments will bring in even more data, and processing and analyzing will be more challenging. For example, while the LHC generates up to a billion collisions per second; the High Luminosity-LHC [6] will generate 7.5 times this rate (data representing each collision will be larger). These larger data samples will be needed to obtain a deeper understanding of the Higgs boson and its implications for the fundamental laws of nature.

A typical HEP workflow to extract physics results from detector measurements consists of three steps: detector signal recording, event reconstruction, and data analysis. HEP detectors' data acquisition systems record detector signals and impose a structure on the recorded data depending on the kind of detector and particle interaction of interest. In the event reconstruction step, physics quantities of broad interest (e.g., trajectories of charged particles, particle hypothesis) are extracted from raw instrument signals. The event reconstruction step involves a variety of pattern recognition, clustering, and track finding algorithms. Reconstruction is normally performed on full raw data sets and is costly in terms of processing time. Data analysis typically involves processing the reconstructed data using selection algorithms, calculations of statistical summaries, and exploratory plotting of the relevant summaries. This step is typically I/O bound.

Analysis is an iterative process where low-latency and interactivity is key to make progress. HEP analysis consists of several processing steps to reduce data to achieve a sufficient level of interactivity. These steps can take from days to weeks; storing intermediate results in files and associated data handling software infrastructure adds to the processing time. Figure 2 shows an example of the time consuming analysis steps, and intermediate data storage requirements. We need to look at alternate approaches to in-memory data processing to enable interactive analysis to meet the needs of the next generation of HL-LHC without significant overhead of bookkeeping and intermediate files. Emerging big data tools look promising for changing the traditional ways of doing these data analyses. However, big data tools are neither designed for the scientific applications and data nor to exploit the high performance computing platforms that are available to the scientific community.

Apache Spark [10], [23] has become industry de-facto in recent years, and provides several attractive features for HEP analyses. The in-memory data processing in Spark allows for interactive analysis, especially where repeated analysis is performed on the same data sets. Since large data volume can be represented in memory, the need to create intermediate files can be eliminated. Spark provides features that can be readily used to encode the physics analysis. The implicit parallelization of the data processing algorithms provides the possibility of good performance and scaling to large numbers of cores without requiring the physicists to master complex parallel programming techniques. This provides important ease-of-use requirement for programmers beyond what can be achieved with the current tools. Their goal is rapid completion of anal-

Operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.

ysis, and not in developing specialized parallel programming skills.

In recent years, Spark has been made available for the scientific applications at National Energy Research Scientific Computing Center (NERSC) and other facilities [11] making it even more attractive for our analysis tasks. This allows Spark applications to run on immensely powerful big compute, big memory machines at NERSC. Getting access to a large well-maintained and tuned installation of Spark eliminates the need for the user to master the installation and tuning of the complex Spark system.

Our goal is to understand how well Spark performs for the data- and compute-intensive HEP statistical analyses on High Performance Computing (HPC) platforms to improve time-to-physics. We use Spark to implement an active LHC analysis, searching for Dark Matter with the Compact Muon Solenoid (CMS) detector as our use case (see Section II), and evaluate its performance on the supercomputing resources provided by NERSC. Experimental HEP is a non-traditional HPC user: only recently are there efforts to use HPC resources for HEP computing. HEP has big data, but the field has its own tools and typically uses grid resources to perform an analysis. Hence, using Spark for HEP analysis on HPC resources provides a unique approach to encoding and executing data analysis tasks.

In Section II, we explain the CMS Dark Matter search use case, and current approach and computing available to perform this analysis. In Section III, we briefly describe few fundamental concepts of Spark and HPC IO used in this paper. In Section IV, we provide details about the input data format and analysis encoding in Spark. Section V explains the experimental setup, and discusses results. We provide a list and discussion of lessons learned in Section VI. We discuss the related work in Section VII and conclusion and future work in Section VIII.

II. CMS Dark Matter Search

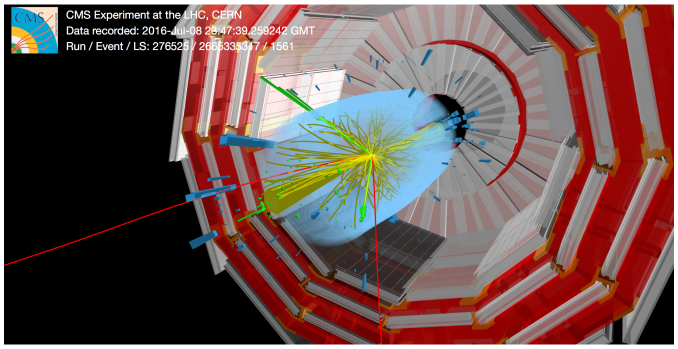

CMS Experiment at the LHC, CERN
Data recorded: 2016-Jul-08 23:47:39.259242 GMT
Run / Event / LS: 276525 / 2663333317 / 1561

Fig. 1: [19]. This figure shows a collision inside the CMS detector. Tracks that are inferred in the detector, and identified as different particles. For example, light green lines are electrons, which are hard to find, and red lines are muons.

A. Science

The CMS detector measures different properties of the particles produced in a collision, such as tracks left by charged particles and energy deposits from all particles that interact via photons and gluons. An example collision is shown in Figure 1. One collision like this is called an event, and these events are used in the analysis step to search for new particles. Our use case focuses on searching for new types of elementary particles explaining Dark Matter in the universe. We focus our search on the monoTop signature, where the detectable particle is a single, unbalanced top quark. Hence, the analysis task looks at all the event data, which includes data about all the particles that were formed as a result of collision and searches for interesting events based on signatures specified by scientists. For example, a search query may look like: *find all the events with missing energy less than 200, and have electrons with a maximum momentum of 10 and muons with a momentum of 4.*

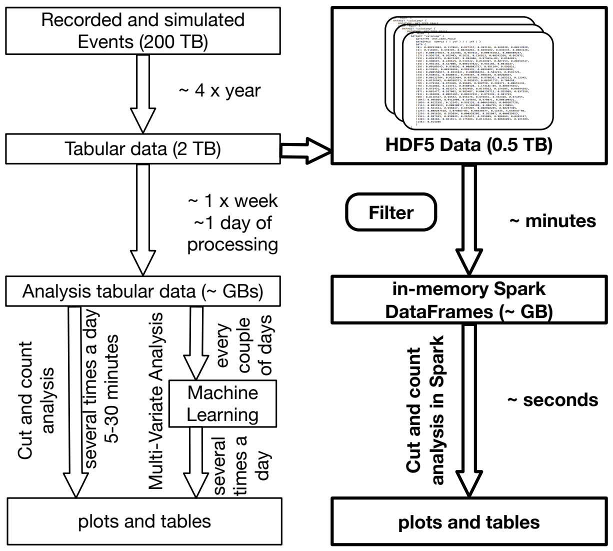


Fig. 2: This figure shows current flow of operations and data, frequency of each operation and time taken by each operation. The proposed and implemented flow is shown in bold. The arrow shows the starting point of our setup, at which point the tabular data is converted to the HDF5 data format [21]. Once data is loaded in memory in Spark DataFrames, the subsequent query operations take a couple of seconds and show potential for interactive analysis. The size of data in the first bold box doesn't represent the conversion of 2TB, rather it only includes the particle data sets needed in this particular analysis task.

B. Computing

The CMS dark matter analysis is using composite C++ objects, which describe either simulated or recorded collisions (events). The traditional user analysis workflow for CMS data uses two frameworks: CMSSW, specially designed for analyzing CMS data, and ROOT, which is a general, experiment-independent toolkit. The ROOT framework provides statistical tools and a serialization format to persist reconstructed and transformed objects in files.

The data volume to be analyzed is about 200 TB for the 2015 dataset and contains both data and Monte Carlo simulations. This is expected to grow in the future significantly in LHC run 2. This structured event data is converted to a simplified tabular data structure, called as n-tuples. These n-tuples have a simple flat structure of vectors of basic types like integers and floats. Each row of a table holds information about different particles (photons, electrons, taus) and their properties (momentum, etc) in an event. Often, the n-tuples are still too big for interactive analysis ($\approx$ 2 TB). Therefore, the contents and the number of events are reduced to couple of GBs. Eventually, quantities from the final n-tuple are aggregated and plotted as histograms.

An example of end-user analysis is to count signal and background events, called "cut-n-count". This is achieved by processing the n-tuples and making optimized selection cuts. In the end, the required plots and tables are produced to extract the physics results of the analysis. The time scale of the complete Dark Matter workflow can range from days to weeks, depending on the number of events needed for analysis. Figure 2 shows several steps in the analysis workflow, and also the time it takes for each step and how frequent each step is performed. The purpose of our study is to evaluate Spark for such analysis. The proposed and implemented workflow is shown in bold in Figure 2, discussed in Section IV.

III. Background: Spark and HDF5

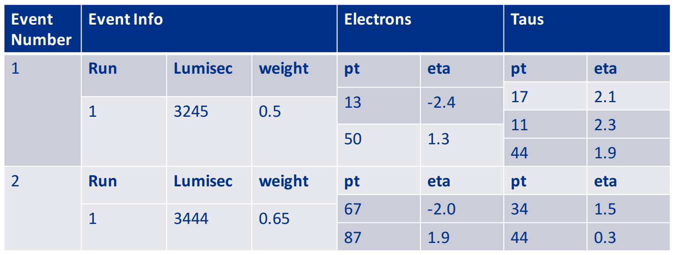
In this section, we will describe useful concepts and terms used in this paper.

Spark [10] was developed for those applications that reuse a working set of data across multiple parallel operations. It introduces a concept of Resilient Distributed Datasets (RDDs), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. Spark SQL is a Spark module for structured data processing [12]. It provides a programming abstraction called DataFrame, which is a distributed collection of rows organized into named columns [1]. A Spark DataFrame is an abstraction for selecting, filtering, aggregating and plotting structured data. These operations are optimized using the catalyst optimizer [17], [22].

Spark uses lazy evaluation by specifying two types of operations: *transformations* and *actions*. A transformation creates a new dataset from an existing one, and an action returns a value to the driver program after running a computation on the dataset. For example, `filter` is a transformation that applies a function to each element in the dataset and returns a new RDD with pass/fail status. A `reduce` is an action that applies some function to all the elements of the RDD, aggregates and returns the final result to the driver program.

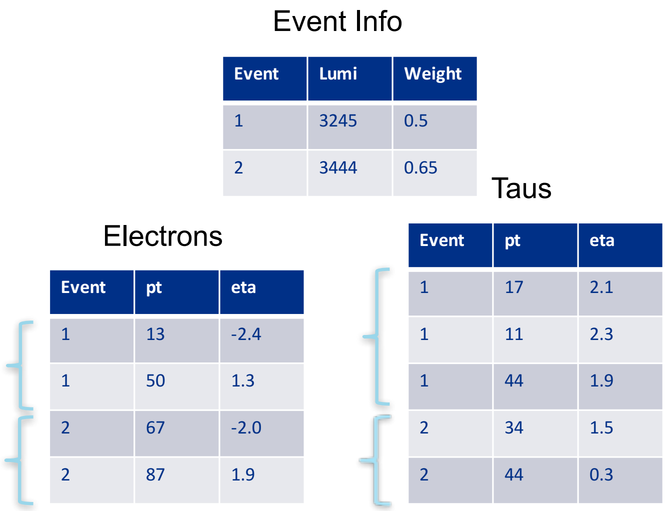
Hierarchical Data Format 5 (HDF5) [21] files organize data into groups and datasets. Each group can have have more groups and also several datasets of different types and shapes (defined by dataspaces). Please see HDF5 documentation on complete description of these concepts.

IV. Design and Implementation



Event Number	Event Info			Electrons		Taus	
1	Run	Lumisec	weight	pt	eta	pt	eta
						17	2.1
	1	3245	0.5	13	-2.4	11	2.3
				50	1.3	44	1.9
2	Run	Lumisec	weight	pt	eta	pt	eta
	1	3444	0.65	67	-2.0	34	1.5
				87	1.9	44	0.3

Fig. 3: This figure shows current data organization, where each row represents an event. Each event has two types of particles; electrons and taus. Event 1 has three electrons and 2 taus, whereas event 2 has two of each. Please note, the numbers in tables do not represent real data, their purpose is only to show data organization.



Event Info

Event	Lumi	Weight
1	3245	0.5
2	3444	0.65

Electrons

Event	pt	eta
1	13	-2.4
1	50	1.3
2	67	-2.0
2	87	1.9

Taus

Event	pt	eta
1	17	2.1
1	11	2.3
1	44	1.9
2	34	1.5
2	44	0.3

Fig. 4: This figure shows the new data organization, where each table corresponds to an HDF5 group. Each particle type group has the data about the event to use join operation later. Such an organization allows for maximal parallel processing by dealing with all different groups in parallel and combining the results in the end. Please note, the numbers in tables do not represent real data, their purpose is only to show data organization.

Figure 2 shows our proposed and implemented workflow in bold. We use HDF5 as our input data format instead of ROOT [13] n-tuples. ROOT is the most common data format in HEP. Using HDF5 allows us to work with other HPC supported programming abstractions including MPI and is well supported at NERSC. We implemented a converter to convert ROOT n-tuples to HDF5 and a reader to read in HDF5 groups into Spark DataFrames. The first bold box shows the converted data in HDF5, and the second box represents data in Spark DataFrames. The arrows show processing from one type of data to another. With the current volume, it takes about several minutes to construct the DataFrames in-memory as needed for the analysis, and a few seconds to run queries on the in-memory DataFrames.

A. *Using HDF5 as Input Data Format*

Converting to HDF5 format: Figure 3 shows the current data organization, where each row contains all the data for different particles and their properties in an event. Our approach is to represent each particle in an event as an HDF5 group, and each property of the particle as an HDF5 dataset within the group as shown in Figure 4. We create a group about general properties of an event, and then a group for each particle type and add additional datasets to identify which events this particle belong to. Such an organization with column-oriented layout allows distributed processing and joins across groups as needed. Each group can be processed independently and in parallel as much as possible, and potentially improve the performance.

We have left performance studies with other data organizations for future work. This includes the use of compound data types to represent all properties of a particle and grouping properties with same data types into N dimensional datasets.

Reading HDF5 in Spark:

HDF5 is not natively supported by Spark framework, however, there are two projects with basic implementations to support reading of HDF5 1D datasets into either RDD or DataFrame [18], [5]. Our most important requirement is to create a DataFrame per HDF5 group without using the Spark join operation, which can be very expensive. The current implementations are preliminary and inadequate for our purposes as discussed in Section VII.

We implemented a customized HDF5 reader for Spark. We used HDF5 Java API and Scala to read in an HDF5 group with specified datasets into a Spark DataFrame. The process of reading HDF5 in Spark is shown using an example in Figure 5. It is not straight forward to read data from HDF5 files into Spark; each of the HDF5 dataset can have a different type. All the data sets in a group have the same number of elements because each group represent a particle type, and datasets within a group represent different properties of each particle. We create a list of tuples (file name, begin index and end index), where begin and end are calculated based on user-defined chunk size, and represent starting and ending index within a dataset. The list is then parallelized by using `parallelize(data, number of tasks)`. By default, the number of tasks are equal to the size of the list, and data represents the list of tuples. Each task reads the same number of elements from each dataset into a row of Spark RDD. This approach allows us to have each row of different type in an RDD with contiguous data reads from HDF5 files.

HDF5 allows you to read or write to a portion of a dataset by use of hyperslab selection. A hyperslab selection can be a logically contiguous collection of points in a dataspace, or it can be a regular pattern of points or blocks in a dataspace [7]. We use hyperslabs to read in chunks to allow maximal parallelism while reading data. At the end of this read step, we essentially get an RRD with the number of rows equal to the number of datasets read, and the number of elements in each row is equal to the user-defined chunk size. Now, we have all the data we need in RDDs but not in the right format to allow for easy to use operations. Each task transposes its RDD, and maps to the appropriate DataFrame schema getting the right form. We define schema for each DataFrame corresponding to each HDF5 group. The data partition is based on the chunk size in each HDF5 dataset per group across all the input files. The expected outcome from HDF5 Spark reader is several DataFrames each representing data of a particle type, and an additional DataFrame with event properties. In future, we would like to explore more efficient mapping from HDF5 to Spark, and other data organizations within our HDF5 files.

B. *Encoding analysis problem in Spark*

The conversion step is extremely important because the structure of data defines what APIs can be used later on in the cut-n-count analysis. The next step is to encode the analysis, by first reducing the contents of the event and the number of events. We define operations for the DataFrame as a whole, which will perform selection and filtering of the data and often use join for DataFrames as we are doing event-based analysis. This requires adding new columns to the DataFrames, where each column would define a complex filter using other columns in the DataFrame. We used the User-defined Functions (UDF) provided by Spark SQL to define new column based functions. We implemented several UDFs, and a few were using inputs from tens of other columns, see [3] for details. The drawback of using UDFs is they are treated as black box code, and Spark doesn't try to optimize these [17]. In some cases, we divided required filter conditions into multiple equivalent conditions to simplify UDFs. Several filter operations per DataFrame are defined; but all these are lazy evaluated. In our filter implementation, we used several conditionals, `select` queries, `groupby`, and `aggregations (sum, count)`. After the aggregation operations, we are usually left with one particle per event in all the particles DataFrames. We used `SQL select` queries to join columns from different DataFrames. The analysis was implemented in Apache Spark 2.0 DataFrame API using Scala.

Figure 6 shows an example of creating a histogram; we can identify several distinct steps. The read operation is followed by filtering, and aggregation. After the aggregation operation, we only have one electron per event that passes given criteria.

V. Results and Discussion

We evaluated the performance of reading, and applying queries to the electron data; there are ≈ 200 million electrons for the 360 million events in our current data set. There are many events with no electrons in them, and there are many events with several electrons. There are 22 datasets in an electron group, and each dataset corresponds to one property of an electron. Each dataset is either a float or an integer (4

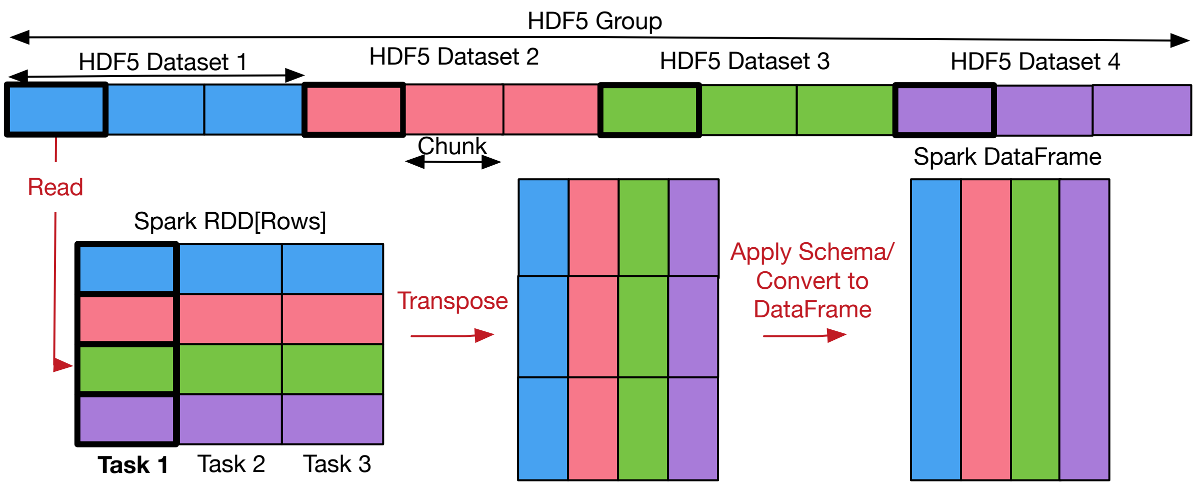


Fig. 5: This figure shows how our HDF5 custom reader creates a Spark DataFrame for an HDF5 group in a file. In this example, the HDF5 group has four datasets and each data set is divided into three chunks. Three tasks are created to read data in parallel. Each task sequentially reads in a chunk from all data sets within a group. Task 1 is highlighted for clarity. The chunks read by different tasks are not contiguously located in the file. Each chunk from a dataset is read into a row of Spark RDD. The resulting RDD is transposed and converted to a Spark DataFrame, with the same number of columns as datasets in HDF5 group.

bytes). Hence, the effective data read for this analysis is linked to electrons only. The analysis operation also involves reading information about 360 million events. The resulting DAG from Spark is also included in Figure 7; when using high level APIs in Spark, it is always challenging to fine tune an application.

The data in the files is organized such that each file consists of several groups, and each group has several 1D data sets. Each group represents one type of particle in an event. Each HDF5 file has 7 groups. Our data set consists of information about 360 million events ($\approx$ 0.5 TB). We have used an example of analyzing electron data.

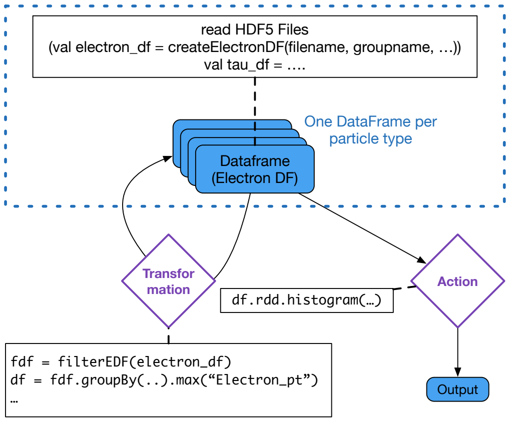
We ran tests on Edison, which is a Cray XC30 supercomputer at NERSC [2]. Each compute node consists of two 12-core Intel "Ivy Bridge" processors at 2.4 GHz. We set the number of executors to 24, the driver memory at 40GB, and the executor memory at 56 GB. The limit on executor memory is 64GB on Edison, and we can run up to 24 executors. We used the Lustre filesystem for retrieving input data.

A. *Number of tasks and partitions:*

We have the data in two forms; a large number of small compressed HDF5 files (gzip) with total 270 GB, and a few larger uncompressed HDF5 files with total of 500GB. There are 928 compressed files, each file is less than 10 GB; they are distributed across 243 OSTs at NERSC. There are 35 uncompressed files with size ranging between 2GB to 90GB. These files are distributed on 35 OSTs. We made another copy with `stripe_medium` set, and each file was distributed on 24 OSTs, using 236 OSTs in total.

For each of these three configurations, we defined four different data partition sizes; 100K means that each task has 100 thousand rows of the DataFrame, 500K means 500 thousand, oneM means one million and tenM means ten million rows. For example, for the 100K size, our partitioning scheme divides all the electron data into 22x100K elements, and 2K tasks are generated. With 24 cores per node, the number of tasks will even out as the number of nodes increase from 64 to 128 nodes. Hence, for 100K partition we expect to see performance curve flattening out.

We observed no affect of partitioning on execution time for the compressed files on 243 OSTs, and for the uncompressed files distributed on 236 OSTs. Upon further investigation, we found out that Spark is not always creating tasks based on our data partitioning but mostly it is using some system optimizations to identify what would be better data and task assignment. In short, we have no control on changing the



```
(val electron_df = createElectronDF(filename, groupname, …))
val tau_df = ….
```

```
df.rdd.histogram(…)
```

```
fdf = filterEDF(electron_df)
df = fdf.groupBy(..).max("Electron_pt")
…
```

Fig. 6: This figure shows the sequence of operations that take place for histogram calculation. The blue dotted box shows the read operation. The output is created when an action is called, and several transformations are applied on the DataFrame.

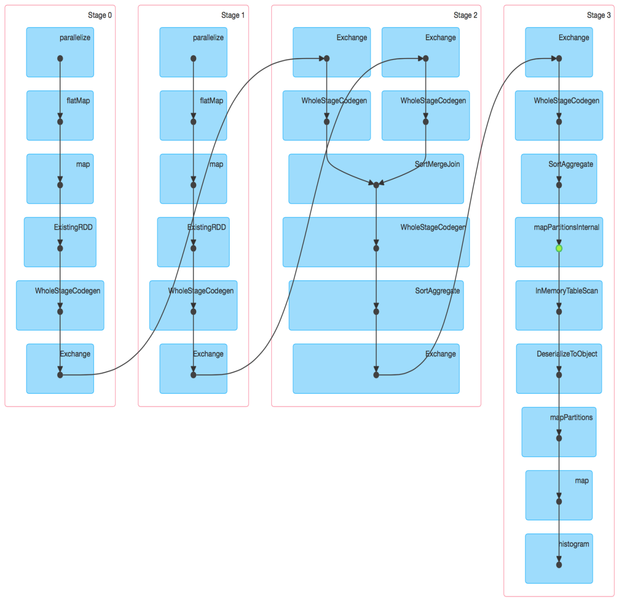


Fig. 7: This figure shows a rather complex DAG Spark creates to calculate the histogram. The first two stages correspond to the parallel read followed by a join operation and further transformations in stage 2. Stage 3 shows the DataFrame to RDD conversion for histogram calculation. The green dot in the last stage shows caching of the resulting DataFrame. The cached DataFrame is used to create histogram.

partitions size, when the data is distributed across almost all the OSTs (246 is the maximum).

In the last configuration, our dataset is using 35 OSTs, i.e. each file is residing on exactly one OST, even the larger ones. The results are shown in Figure 8. 100K and ten million appears to be either a lot of partitions or too few partitions to benefit from the existing resources. Regardless of the partitions, we observe scaling with number of cores to a point where current data size can not optimally use all the cores.

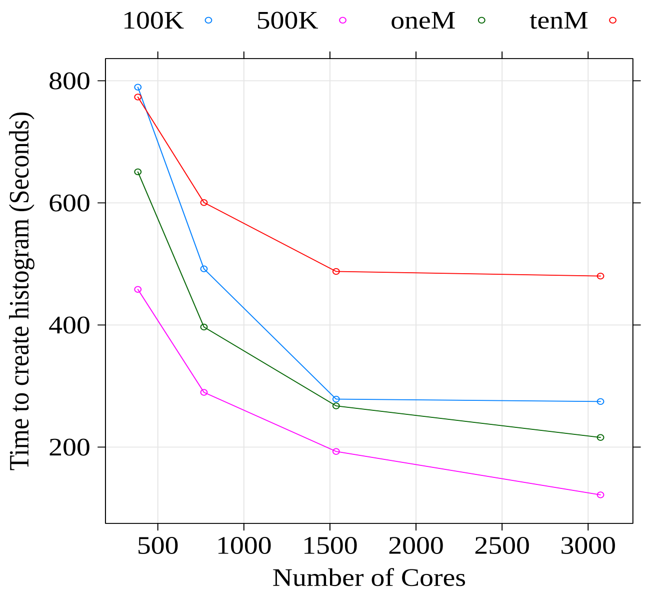


Fig. 8: This figure shows the impact of different number of partitions with varying number of cores. We used four different partition sizes: 100K means that each task has 100 thousand rows of the DataFrame, 500K means 500 thousand, oneM means one million and tenM means ten million rows.

B. *Compressed and uncompressed data:*

Using the same configurations for file distribution and sizes, and partition of 500K, we compared the execution time for the histogram query as shown in Figure 9. The test using compressed files on 243 OSTs show the best performance as compared with the tests using uncompressed files. However, we observe convergence in the performance by all three tests as the number of cores outnumber the data partitions. The better performance of compressed file is attributed to the following: decompressing data in memory is faster than reading more data from the disks. The uncompressed files on fewer OSTs, performs worse than the other two because of the OST contention, while reading the files. All three organizations show similar scaling behavior.

C. *Different Steps in Spark:*

In Spark, it is difficult to isolate different internal stages. We divided the analysis operation in several steps, and forced

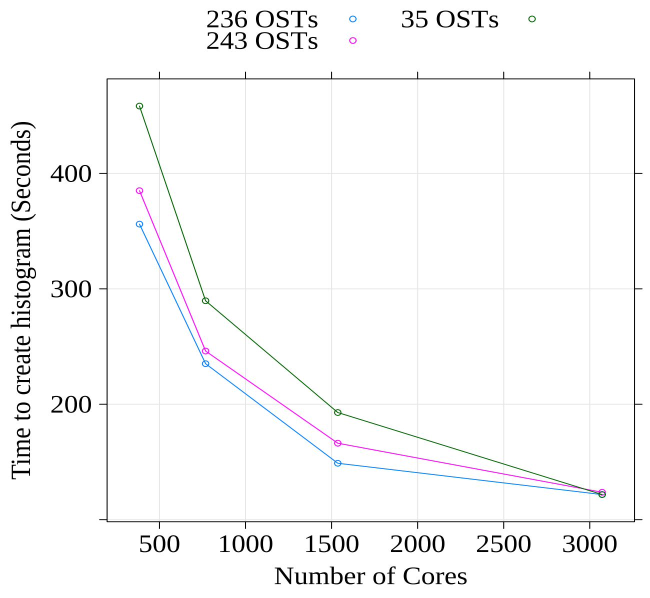


Fig. 9: This figure compares the time for histogram query using both compressed and uncompressed data. The compressed data is using 243 OSTs, and uncompressed is using 35 OSTs in one case and 236 in another.

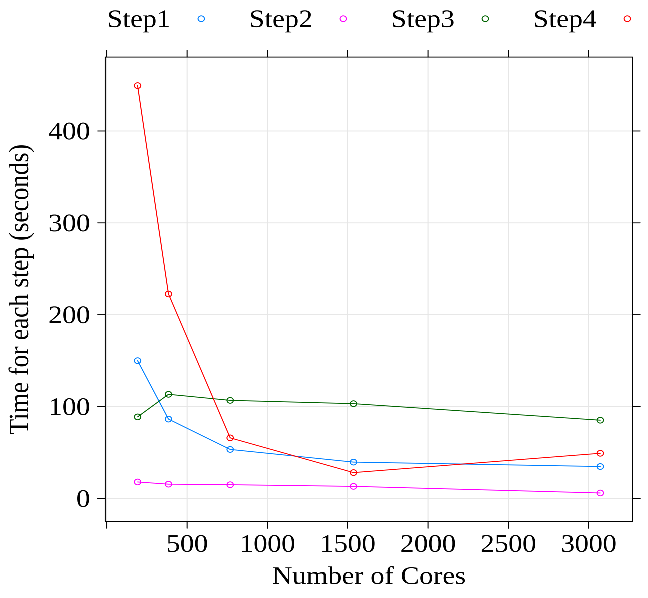


Fig. 10: Time taken by different steps in Spark for the histogram query. The read in step 1 scales, step 2 and step3 show a constant overhead regardless of scale. Step 4 is the most complicated and involves several operations; it demonstrates that once a DataFrame is created, we observe good scaling for 1534 cores and degraded performance after that.

evaluation at end of each step by using an appropriate Spark action. As already described, the analysis operation is to create histogram of transverse momentum of electrons that pass certain criteria. The task is to look at each electron independently to see if it passes the given filter, group by event, and then keep the electrons with the highest transverse momentum. The high level operations we used are:

- Step 1: Read in the electron group from HDF5 files into an unformatted RDD
- Step 2: Format the RDD to right shape (transpose)
- Step 3: Convert to DataFrame using the schema based on HDF5 group
- Step 4: Histogram of transverse momentum (Read in info group into a Spark DataFrame, join with electron DataFrame, and several operations using UDFs on the DataFrame, e.g. aggregation, filtering)

In Figure 10, step 1 shows good scaling with increase in number of cores to the point where number of cores outnumber the number of tasks, same is the case with Step 4. Step 4 is rather complex, and involves few global synchronization and several transformations, e.g. filter defined in UDFs. Once data is correctly loaded into memory, the best time for step 4 is $\approx$ 28 sec to run the user query. It shows the expected time for the similar wide range of operations. Step 2 and 3 do not involve any reading from disk operation or computation, and we see a constant overhead regardless of the number of cores used.

D. *Comparing MPI and Spark*

In this subsection, we provide preliminary results for exploring alternate approaches to Spark. It was not feasible to run the relevant parts of the original analysis code on Edison. We are using MPI as an alternate to our Spark implementation. Reading HDF5 files using MPI on Edison is extremely efficient. The availability of Python interface in MPI along with the high performance I/O makes MPI a possible approach to address this problem.

Global reduction operations are frequently done in several analysis; in this test we compared the time it takes to read a single HDF5 dataset and compute sum of its elements using Spark and MPI. Both implementations are configured to read from the uncompressed HDF5 files distributed on 236 nodes. We kept the same partitioning scheme with 500K elements for Spark. We used `agg(sum("weights"))` on the DataFrame, which has *weights* as one of the columns.

The MPI implementation used parallel read with in each file utilizing the Python interface for reading HDF5 files, `h5py` [4], with MPI/IO support (`mpi4py` [8]). In MPI, we used `NumPy` [9] arrays to hold the data in memory and `MPI.COMM_WORLD.Reduce()` with op as `MPI.SUM`. The interface provided by `h5py`, `mpi4py`, and `numpy` is simple to use, and offers a competitive approach to the high level API offered by Spark. The results are shown in Table 1. MPI implementation always performs better as compared with the Spark implementation; both are reading the same data, and doing the exact same calculation. We also observed that once data is loaded into the Spark DataFrame, the sum operation takes less than 4 seconds to complete.

Due to the size of data, we haven't seen any performance scaling beyond 384 cores for both MPI and Spark. However, we have observed good scaling behavior in Spark in general. With the MPI implementation, providing scalability is challenging and requires a different distribution of input files every time with the varying number of tasks. Additional algorithm need to be written and maintained for this implementation to work well as a system.

TABLE I: Comparing Read and Summation time for the MPI and Spark implementation

Number of Cores	**96**	**192**	**384**
Time for MPI	4.89 sec	7.01 sec	7.14 sec
Time for Spark	107 sec	69 sec	48 sec

VI. Lessons Learned

We discuss observations and lessons learned as a result of our project:

1) *Input data format and organization:* The structure and organization of data defines what APIs can be used later on in the analysis. We had the option to define data organization to allow for efficient read into Spark DataFrame and use its API efficiently. The DataFrame API provides all the features we needed to implement the use case. The CMS data is stored in the form of ROOT trees [13], which is the most common data format in HEP, which is hierarchical and complex and Spark DataFrames are tabular. Hence, we need flattened tables to effectively use the API. We used HDF5 data format to define logical tables into groups for this project. We have used RDDs in the past, but the API results in more complex code. Use of Dataset API will be explored in our future work.
2) *Operations on multiple DataFrames:* Currently we read data in multiple logical tables, e.g. all information regarding a given reconstructed particle type can be found in one table. The nature of this analysis is to combine the data from different tables and make several histograms for different particles properties. If we read data from multiple DataFrames, adding a new column to the DataFrame from another DataFrame is not supported within Spark. Spark only supports adding new columns within the same DataFrame. We use `join` or SQL select queries to add columns from a different DataFrame.
3) *High Level API:* There are functions available to perform transformations, aggregations, global reduction in a distributed environment, which can be readily used. Such a set-up provides ease-of-programming, however, it does mean that the user has to rely on system optimizations provided by Spark's implementation to improve any performance. Similarly, it is hard to understand a DAG created by Spark for SQL queries on DataFrames, and users have minimal control over optimizing the query structure, data partitions, etc.
4) *Orchestration:* We defined partition size, and number of tasks for our use case but for many use cases we can use the default partition size, which is equal to file system block size. But figuring out the best partition size for a job is challenging. Allocating tasks and data partition to the worker nodes is abstracted from the user. Users don't have to change the number of tasks to match with the number of cores available.
5) *Scaling:* The Spark implementation provided good scaling without requiring any tuning to the implementation and developer expertise in parallel algorithms. When the number of partitions is less than number of cores allocated, we are wasting resources and observe degraded performance. However, it has been challenging to control number of tasks in some cases. We anticipate good scaling behavior when we use bigger data sets ($\approx$ 200TB).
6) *Application tuning:* All the transformations in Spark are lazy, with delayed calculation of results. The transformations applied to the base dataset e.g. a file are remembered by the system and only computed when an action is carried out on the dataset. This design enables Spark to run more efficiently. For example, it can recognize that a dataset created through `map` will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset. Due to the lazy evaluation, it is hard to isolate slow-performing tasks and report timing for different stages.
7) *Using appropriate Spark flags:* Without an expert help it is challenging to identify what configuration and options to use for an application. For example, memory per node, number of cores per node, debug level, etc.
8) *Multiple Language Support:* The availability of Python and R interface for end user analyses is also a key feature desired by the HEP community.
9) *Debugging and Error Messages:* Most of the times error messages in Spark are misleading and need more work just to understand the error message before working on a fix. Use of graphical interface and history server can be useful but when using DataFrame API, understanding different stages and partitions is not straightforward.
10) *Documentation:* Spark could be used more effectively with comprehensive documentation with reasonable examples.

VII. Related Work

Using big data technologies from industry is not a new area in HEP, but we are the first group exploring the use of big data technology and HPC for HEP. In the past, we have done some exploratory work by implementing a compute intensive event classification algorithm using Spark, but the algorithm itself was unsuitable to use caching and in-memory repeated processing provided by Spark [20] resulting in poor performance as compared with the MPI implementation. Our CMS use case, however, presents us with the opportunity to fully explore the attractive features of Spark as discussed previously.

Our current work on using Spark for CMS Dark Matter use case is part of CMS big data project, which also includes using industry standard data format, a data organization that is consistent with the existing data organization in ROOT, and Hadoop ecosystem instead of HPC platforms [16].

Using MapReduce for High Energy Physics Data Analysis [15] makes use of ROOT in Hadoop eco-system. The use of MapReduce does not solve the iterative and interactive HEP analysis problems. Our approach uses Spark on HPC, which enables iterative and interactive analysis as well as the use of HDF5 which is a supported HPC format.

h5spark [18] provides the ability to read HDF5 1D datasets into Spark RDDs. We needed higher level API access to the datasets, and converting RDDs to DataFrame was an extra overhead. Also, we needed the ability to read data such that all the datasets in a group should be in one DataFrame. These two requirements made it inconvenient for us to use h5spark. hdf5-spark [5] is a Spark plugin to read HDF5 data in DataFrames, however it doesn't meet our requirement of reading all HDF5 datasets into a Spark DataFrame. The only way to achieve this requirement is to use join operation, which is extremely inefficient if we have to call join for each dataset. Our HDF5 to Spark DataFrame reader allows us to create one DataFrame per HDF5 group, but customized to our current data layout.

VIII. Conclusion and Future work

We chose the CMS Dark Matter use case because its processing stages represents an important class of problems in experimental HEP. These stages include reducing the event contents and event count by applying several selection criteria, and plotting. Here is a summary of our contributions:

- Implemented a prototype of an HDF5 Spark reader for reading several 1D HDF5 datasets within a HDF5 group into a single Spark DataFrame with user specified partitioning size across multiple files
- Provided a tabular data representation and ability to perform key operations in distributed environment beyond batch processing.
- Provided encoding of analysis in Spark environment, which eliminated the need of intermediate file storage

Spark is relatively new and emerging technology, and its use, especially in the HEP community, is in exploratory stages. The learning curve involved with its use, especially using Scala, cannot be ignored. However, the availability of APIs in R and Python improves beginner's experience. Other advantages include task distribution and user controlled data partitioning. We have seen good scaling behavior of Spark applications with increase in dataset size and the number of nodes with no extra work. Encoding analysis workflow using Scala best practices and the optimal use of DataFrame features is challenging. The documentation and error reporting should be improved. However, the ease of use, reasonable performance and good scalability makes Spark a viable candidate for our future work.

In the future, we would like to study the impact of various HDF5 data layouts and their efficient reading in Spark. We are collaborating with the data analytics team at NERSC to understand performance of our use case in Spark. We will explore the use of OST for maximal parallelism. We will also study the usability of the new Dataset APIs for our use case. We will study scalability with 100 times larger data. We will also implement the full use case using MPI, and compare ease of use and performance with the Spark implementation. We will also implement more use cases from HEP analyses and explore multi-user aspect of interactive analysis.

IX. Acknowledgments

We would like to thank our team at Fermilab: Marc Paterno, Oliver Gutsche, Matteo Cremonesi, Bo Jayatilaka, Cristina Mantilla and at Princeton University: Jim Pivarski, Alexey Svyatkovskiy for providing details on the science use case and computing model. We would like to thank Lisa Gerhardt for guidance in using Spark optimally at NERSC. This research supported through the Contract No. DE-AC02-07CH11359 with the United States Department of Energy, 2016 ASCR Leadership Computing Challenge award titled "An End-Station for Intensity and Energy Frontier Experiments and Calculations". This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] DataFrame API. http://spark.apache.org/docs/latest/sql-programming-guide.html.
[2] Edison. . http://www.nersc.gov/users/computational-systems/edison/.
[3] Github repository for the CMS Dark Matter Analysis Code using Spark. https://github.com/sabasehrish/spark-hdf5-cms.
[4] HDF5 for Python. http://www.h5py.org.
[5] HDF5 Plugin for Spark. https://github.com/LLNL/spark-hdf5.
[6] HL-LHC: High Luminosity Large Hadron Collider. http://hilumilhc.web.cern.ch.
[7] Hyperslab Tutorial. https://support.hdfgroup.org/HDF5/Tutor/select.html.
[8] MPI for Python. http://mpi4py.scipy.org.
[9] NumPy. http://www.numpy.org.
[10] Spark. https://spark.apache.org.
[11] Spark Distributed Analytics Framework at NERSC. https://www.nersc.gov/users/data-analytics/data-analytics/spark-distributed-analytic-framework.
[12] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.
[13] R. Brun and F. Rademakers. ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth.*, A389:81–86, 1997.
[14] Serguei Chatrchyan et al. Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC. *Phys. Lett.*, B716:30–61, 2012.
[15] Fabian Glaser, Helmut Neukirchen, Thomas Rings, and Jens Grabowski. Using MapReduce for High Energy Physics Data Analysis. In *Proceedings of the 2013 International Symposium on MapReduce and Big Data Infrastructure (MR.BDI 2013), 03-05 December 2013, Sydney, Australia 2013*, December 2013.
[16] Oliver Gutsche, Matteo Cremonesi, Bo Jayatilaka, Jim Kowalkowski, Saba Sehrish, Cristina Mantilla Suarez, Nhan Tranl, Peter Elmer, Jim Pivarski, and Alexey Svyatkovskiy. Big Data in HEP: A comprehensive use case study. *To be published in the proceedings of CHEP 2016.*

[17] Jacek Laskowski. Mastering Apache Spark 2.0. https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark/details.

[18] Jialin Liu, Evan Racah, Quincey Koziol, and Richard Shane Canon. H5Spark: Bridging the I/O Gap between Spark and Scientific Data Formats on HPC Systems. *Cray User Group (CUG'16)*.

[19] Thomas Mc Cauley. Higgs boson produced via vector boson fusion event recorded by CMS (Run 2, 13 TeV). CMS Collection., Aug 2016.

[20] Saba Sehrish, Jim Kowalkowski, and Marc Paterno. Exploring the performance of spark for a scientific use case. *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 00(undefined):1653–1659, 2016.

[21] The HDF Group. Hierarchical Data Format, version 5, 1997-2017.

[22] Rishi Yadav. *Spark Cookbook*. Packt Publishing, 2015.

[23] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.