FERMILAB-CONF-16-072-CD

Exploring the Performance of Spark for a Scientific Use Case

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

Marc Paterno
Fermi National Accelerator Laboratory
P.O.BOX 500
Batavia, IL 60510
Email: paterno@fnal.gov

***Abstract*—We present an evaluation of the performance of a Spark implementation of a classification algorithm in the domain of High Energy Physics (HEP). Spark is a general engine for in-memory, large-scale data processing, and is designed for applications where similar repeated analysis is performed on the same large data sets. Classification problems are one of the most common and critical data processing tasks across many domains. Many of these data processing tasks are both computation- and data-intensive, involving complex numerical computations employing extremely large data sets. We evaluated the performance of the Spark implementation on Cori, a NERSC resource, and compared the results to an untuned MPI implementation of the same algorithm. While the Spark implementation scaled well, it is not competitive in speed to our MPI implementation, even when using significantly greater computational resources.**

I. Introduction

In High Energy Physics (HEP), datasets from both experiments and simulations are increasingly large and complex, broadening the gap between what analyses can and should be done to advance the scientific discovery process. Analyses are constrained by the computation and storage capabilities available to the scientists, and by the computing model common in HEP. The techniques used to organize applications and algorithms contribute to the limitations of this model, including large memory space requirements (~ 2 GB/process) and lack of parallelism (typically singly threaded processes are used). These in turn impact how much data can be used conveniently in analysis. Smaller analyses can be done efficiently using today's technologies, but there are many tasks that take days or weeks to complete; this high latency is detrimental to exploratory analysis. In particular, the computing model makes it difficult to make use of large-scale multicore computing facilities.

Most HEP data processing involves tasks such as data classification, integration, pattern matching and parameter estimation (optimization, minimization and fitting). At the scale of petabyte datasets, such analyses can't be done with low-latency, nor in near-real-time, without leveraging parallel computing technologies and hardware.

Classification is the process of assigning objects or events to a set of possible discrete classes. In HEP, the typical classification problem involves relating an observed signal in a detector to the physical particles or processes that resulted in the production of that detector signal. For example, identifying the observation of a high-energy muon by recognizing the pattern of light flashes observed in a scintillating material caused by the passage of the muon, as opposed to the pattern that might be left by a different type of particle, such as an electron. [8]. In this paper, we present our experience of implementing a classification algorithm used by the NOvA [4] experiment, used to identify the type of neutrino interaction responsible for leaving an observed set of energy deposits in the NOvA detector, using Apache Spark. [5], [11]

Spark allows for in-memory data processing, and is an attractive approach for the cases where repeated analysis is performed on the same data sets. Spark provides implicit parallelization of the physicists' data processing algorithms, thus providing the possibility of good scaling to large numbers of cores without requiring the physicists to master complex parallel programming techniques. This provides important ease of use for "casual" programmers, for whom the goal is rapid performance of analysis, who are usually not interested in developing specialized parallel programming skills. Additionally, a supported and tuned installation of Spark is available at NERSC [6], thus eliminating the need for the user to master the installation and tuning of the complex Spark system.

The goal of this exploratory work was to evaluate Spark for an HEP analysis use case, using a sufficiently large data sample and involving a typically complex set of compute-intensive calculations, and to compare that solution with a parallel programming technology more traditionally used in HPC: MPI.

In Section II, we present the algorithm. In Section III, we provide details about implementation in Spark and MPI. Section IV explains the experimental setup, and describes results. We provide conclusion and future work in Section V.

II. NOvA Library Event Matching Algorithm (LEM)

The NOvA problem is to classify an unknown event by comparing it to a large number of known simulated events. The metric used for comparison is loosely based on the electrostatic energy comparison for two systems of point charges laid on top of each each other. The charges are taken to be the recorded energy deposits (hits) in the NOvA detector. Once the very best

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

matches are found (here, the best 10^4 of all simulated events), their known properties are used to estimate the properties of the event to be classified. The details of the matching criteria calculations are described in the paper [3].

We only describe a very simple high level equation here, which we used to implement the algorithm. An event A to be classified is matched with an event B in the simulated events; the match score is defined as $E = \frac{1}{2}E_{AA} + \frac{1}{2}E_{BB} - E_{AB}$ where E_{AA} is the self-energy (repulsion) of event A's charges, E_{BB} is the self-energy of event B's charges, and E_{AB} is the (negative) energy due to the the A/B attraction. The self energy of the simulated events, E_{BB} is pre-calculated. The E_{AB} calculation is shown in the algorithm 1. The input event and template is the same for E_{BB} and E_{AA}.

Algorithm 1: Algorithm to calculate electrostatic energy, this function accepts an event and a template, each event and template has the following information per hit: cells (c), planes (p) and energy (e), and the number of hits (N).

function calcEEnergy (A, B);
Input : Event A and template B;
N= the number of hits in the event to be classified;
p_i, c_i, e_i= plane and cell coordinates and energy of hit i in the event to be classified; $i \in N - 1$;
N_k= the number of hits in the template k;
$p_{k,i}, c_{k,i}, e_{k,i}$= plane and cell coordinates and energy of hit i in the template k; $i \in N_k - 1$;
Output: The metric value for measuring the closeness of template k to the event.
$\beta = 0.5$;
T =function of the distance (motivated by the electrostatic analogy) between the hits; see [3] for details;
$E_{AB} = \sum_{i=0}^{N-1} \sum_{j=0}^{N_k-1} {e_i}^{\beta} {e_{k,j}}^{\beta} T(p_i - p_{k,j}, c_i - c_{k,j})$;
return E_{AB};

III. Implementation

A. Using Spark

1) Background: Spark [5] 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 [7]. It provides a programming abstraction called DataFrame, which is a distributed collection of rows organized into named columns [2]. A Spark DataFrame is an abstraction for selecting, filtering, aggregating and plotting structured data. These operations are optimized using the catalyst optimizer. The *Catalyst Optimizer* leverages Scala's pattern matching and quasiquotes for logical optimization, physical planning, and run-time code generation. Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in the main program (called the driver program). We used Spark DataFrame API (in Java) to implement the classification algorithm.

2) Implementation Details: **Input Data Format** The NOvA data is stored in the form of ROOT trees [10], which is the most common data format in HEP. The ROOT I/O format is not directly accessible in Spark unless there is a ROOT I/O reader and writer implemented for use with Spark and HDFS. The data format is relatively simple for this problem. Data is kept in two types of files. The first has the properties of an event, e.g. number of prongs, number of hits, total PE, etc. The second includes cells, planes, etc per event, which are used to calculate the score. We used two approaches to deal with the data format; the first approach was to use the ASCII tabular form in text files, and the second approach was to use the JSON format. The first approach was used for RDD implementation, and its disadvantage was the extensive and repeated parsing of the data to retrieve the fields for calculation.

The second approach was used for DataFrame implementation, and the data was directly read into DataFrame thus allowing direct operations on a single row of the DataFrame. We will only discuss the second approach in this paper. Spark SQL can automatically infer the schema of a JSON dataset and load it as a DataFrame [5]. We used *jsonFile*, which loads data from a directory of JSON files where each line of the files is a JSON object.

Limitations/Constraints: To make the data ready for Spark implementation, we encountered the following limitations. Our solutions are also described.

1) *Multi-line JSON form not supported:* Spark does not support typical JSON file format, rather it expects each line must contain a separate, self-contained valid JSON object. As a consequence, a regular multi-line JSON file does not work. We made sure that JSON files we create consist of one JSON object per line.
2) *Adding a new column to the DataFrame:* To perform the score calculations and classification, we needed one DataFrame with all the required fields from both metadata and data files. Hence, adding a new column to the DataFrame from another DataFrame was a natural choice. However, it was discovered that Spark DataFrame doesn't support adding a column from a different DataFrame, it only supports columns within the same DataFrame.
3) *Using `join` operation instead:* In order to address the previous hurdle we attempted to use inner join, but the overhead of the `join` operation was extremely overwhelming that we had to pre-process the data, and make amendments to data files before copying them to Cori
4) *Getting data out of DataFrame by index of column:* Another minor issue with using data frame API was getting data for a particular row given the column name. It works with a column index, and not the column name. e.g. `getDouble(int index)` is correct but `getDouble("columnX")` is not.

The schema of final JSON file we used to create DataFrames is as follows:

```
root
 |-- c : array (nullable = true)
 |    |-- element: long (containsNull = true)
 |-- EventID : long (nullable = true)
 |-- e : array (nullable = true)
 |    |-- element: long (containsNull = true)
 |-- p : array (nullable = true)
 |    |-- element: long (containsNull = true)
 |-- ccnc: long (nullable = true)
 |-- mode: long (nullable = true)
 |-- N : long (nullable = true)
 |-- EBB: double (nullable = true)
 |-- theta1: double (nullable = true)
 |-- theta2: double (nullable = true)
```

`c`, `p`, `e` and `N` are used in the algorithm 1 to calculate electrostatic energy. `ccnc` and `mode` are used to identify the type of an event. `theta1`, `theta2` and `N` are used in the filter operation specified in Section 3.3. Spark team does not recommend JSON format for production level analyses, we will investigate alternate formats in the future work.

There are two types of RDD 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, `map` is a transformation that applies a function to each element in the dataset and returns a new RDD representing the results. In our implementation, we used a `map` function to calculate score for each match between event to be classified and events in the library. We use DataFrames and convert them to RDDs before running transformations. 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. We used `top` method to aggregate the best N matches. Figure 1 shows the transformation and action in our implementation.

Our implementation consist of Java classes and methods for the matching score calculation. Then from the Spark API, we used the `map` method to calculate E_{AB} for each library event with event to be classified. We formed tuples out of the DataFrames, where each tuple was of the form `Tuple2<Long, Tuple2<Long, String> >`. The first `Long` corresponds to the Event ID of the library event, needed for the look-up to find properties of the event. The second `Long` is for the score, and the `String` is for the properties of an event. Since the `map` method can not be directly applied to a DataFrame, a conversion to RDD is needed before using `map`. Since `map` is a transformation, the score calculation is not performed until an action is called. We used the `top` method to return the top matches and provided custom implementation of a comparator to work on the tuples we created. Figure 1 shows the sequence of `map` and `top` for each event classification.

Each transformed RDD is recomputed each time an action is run on it. We used the `cache()` feature available in Spark to persist the initial data set in the memory. Spark keeps the elements in memory for much faster access the next time. Spark system creates a DAG for executing user program and divides the program execution into different stages as shown in Figure 2. The DAG created for execution shows a green dot as an indication of data persisted in memory. There is also support for persisting RDDs on disk, or replicated across multiple nodes [5]. In our case, we didn't persist the computed RDDs because computations vary per event to be classified as shown in the equation for calculating E_{AB}.

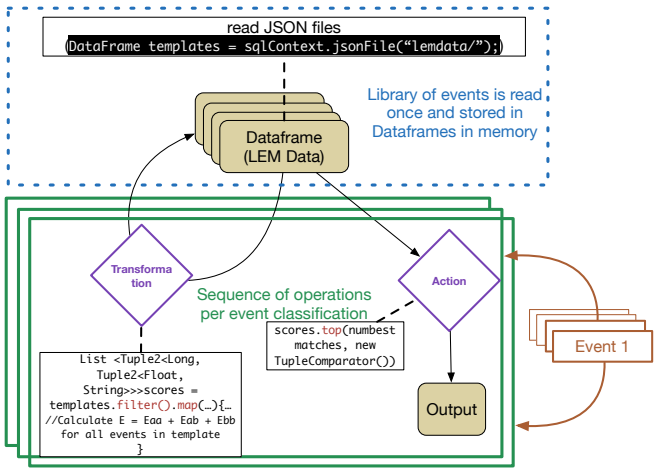

Fig. 1: Flow of operations and data in Spark. This figure shows the sequence of operations that take place per event classification. The green boxes correspond to the program flow for an event to be classified. The dotted blue box represents the load/file read operation that happens only once, and the DataFrame is loaded in the memory, and used for event classification.

Optimization The number of E_{AB} calculations are directly proportional to the number of events in the template library, hence we used an optimization to reduce the number of events to look at thereby reducing the number of E_{AB} calculations. The `filter` operation is used to implement this functionality. Our filter is based on two characteristics of the events, one is number of hits and other is the angle theta along x and y direction. The filter limits the events to have number of hits in the range $IH/2$ to $2 \times IH$, where IH is the number of hits in the event to be classified.

B. Using MPI

Input Data Format. As mentioned earlier, the NOvA source data is stored in the form of ROOT trees [10]. The MPI implementation transforms the trees into data structures similar to the Spark JSON schema: events are represented by a struct containing relevant attributes and arrays for cell position, plane position, and cell energy. A second data structure is maintained with the entire set of metadata attributes for each event. A major difference is that the arrays use fundamental types (shorts for position and floats for energy). A custom output routine writes the event structure directly into binary disk files. The result of the conversion is 200 files containing the entire LEM dataset with the number of events per file matching the original ROOT files. A custom input routines reassembles the in-memory representation from the binary data

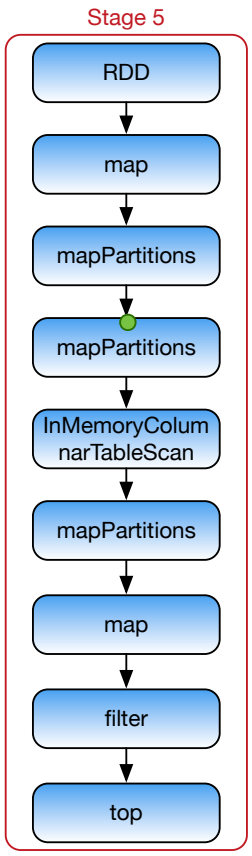

Fig. 2: DAG showing one of the stages. Each blue box corresponds to one individual operation (transformation or action). The first three boxes represent getting the input data i.e. event to be matched and the templates. The last three boxes represent two transformations and one action performed on the data. The cached DataFrame is denoted by the green highlight.

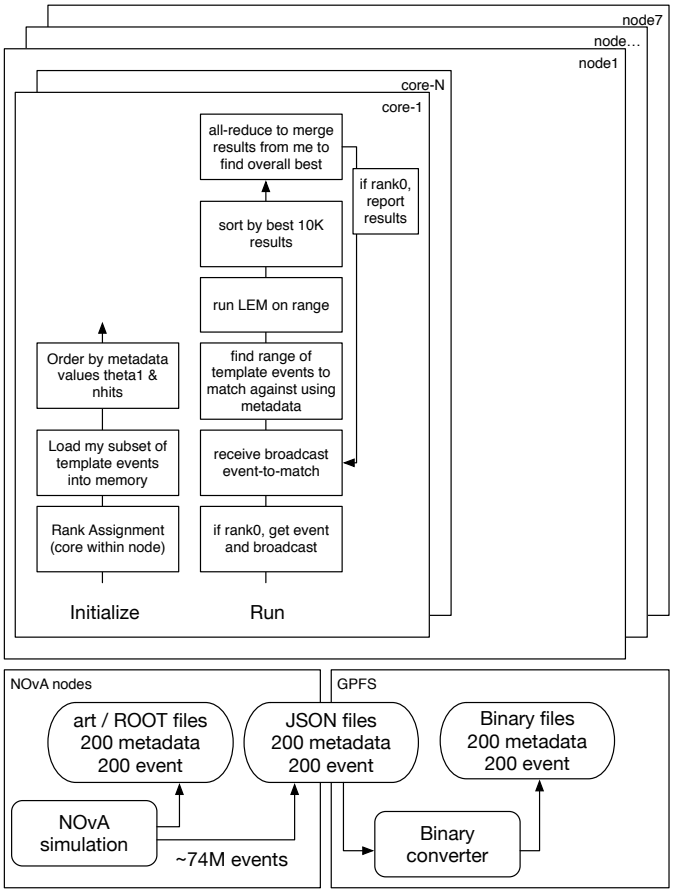

Fig. 3: Flow of operations and data in the MPI implementation. This figure shows the sequence of operations that take place per event classification.

files. The original ROOT dataset did not have equal number of events or equal file size; the transformed binary dataset echoes this feature and does not attempt to correct it.

The data distribution scheme among the processing ranks is rigid and simple: each rank consumes (TotalFiles / TotatRanks) number of files and build a private in-memory dataset using these events. The size of data files vary from hundred of MB to less than five GB. Therefore the only run represented in this paper is 200 files and 200 ranks or one file per rank. The output data (the 10K best matches) is returned as an array eight byte integers. The integers contain of the score, the event ID and its type information. Such a compact form permits for an efficient implementation of the merge sort operator used in the reduction stage.

Calculations. The calculations required by the algorithm were easy to implement using C++ and MPI. Several high performance numeric and data structure libraries are readily available for C++. Armadillo [1] and the C++ STL were used to implement the computations. The LEM MPI implementation uses datatypes and structures that can be directly manipulated by these libraries. Armadillo can make good use of available vector units and perform calculations using LAPACK and BLAS when necessary [1]. For example, we used Armadillo `dot` product in the LEM calculation and STL's std::partial_sort for sorting results.

The operations in main body of the application are divided the following stages, also shown in Figure 3: `MPI_Bcast` to hand out an event to be classified, `scoring` to find the best 10K matches, `sorting` to sort the 10K matches, and `MPI_reduce` to collect the best 10K across the all ranks (custom reduce operator is used).

Optimization. The MPI implementation reduces the number of events considered using the number of hits. Library events for scoring on each rank are only considered if its hit count is in the range (IH/2, IH*2) where IH is the hit count of the event to be classified.

IV. Results and Discussion

We evaluated the event classification rate (i.e. number of events classified per second) by comparing a set of events with a library of $\sim$ 74 million events. The library data is in 200 files in the range of couple of hundred MBs to couple

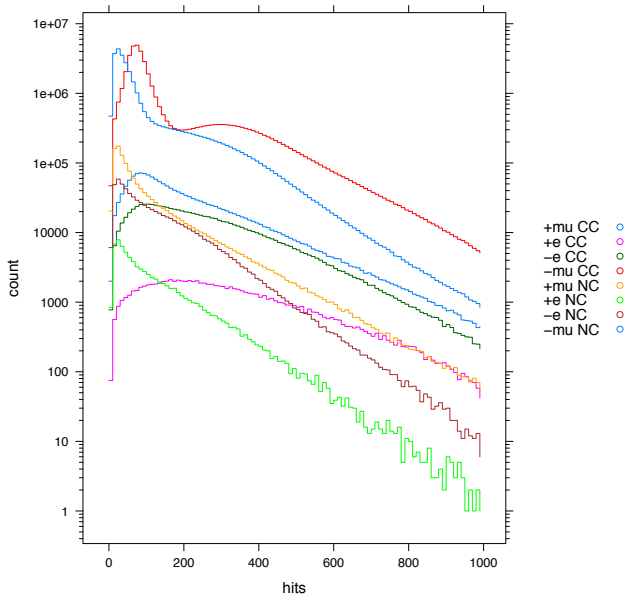

Fig. 4: This graph shows the distribution of the number of hits per event in the template library. Each line corresponds to a different type of event. Most of the events in the library have fewer number of hits.

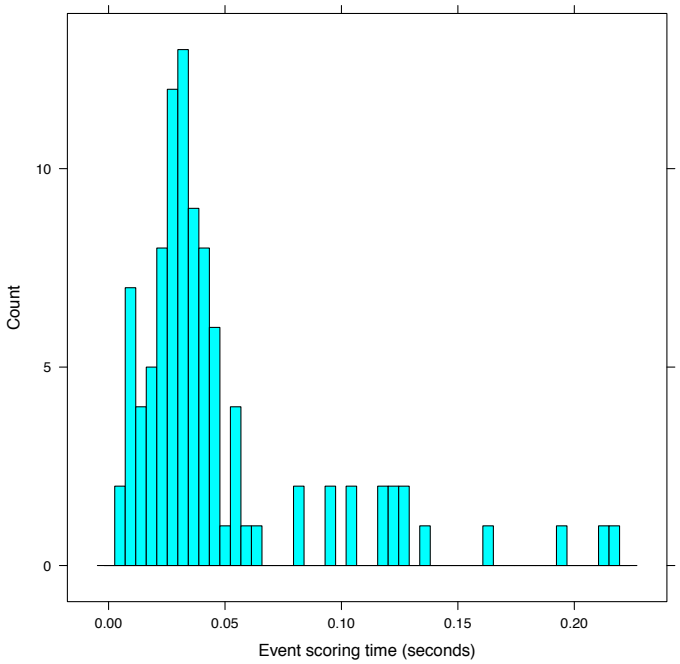

Fig. 5: The distribution of score times for the sample of 100 events to be classified using MPI with 7 nodes on Cori.

of GBs. We used varying number of hits for each event to be classified, ranging from 20 to 250. The reason for using this range is that the exploration of data sample revealed that the common number of hits per event is less than 100, also shown in the Figure 4.

For MPI, we classified 100 events. To keep the configuration simple, 200 cores were used by specifying 7 nodes with 32 cores per node in the job allocation request. Each MPI rank reads one file regardless of its size. Each file size is in the range of hundreds of MBs to a few GBs. Figure 5 shows the scoring time, in seconds, for each of the 100 events being classified. The scoring time is the wall clock time to calculate the scores of all the templates that pass the filtering condition. Most of the events take less than 50 milliseconds. Figure 6 shows the time to gather the best 10 thousand scores; this is the time for the local sort and the global reduction. All the events took between 1.8 and 2.2 milliseconds to perform sorting and reduction. No event took more than 0.4 seconds to classify, as can be seen in Figure 7.

The rigid file distribution among MPI ranks, along with the uneven files sizes and the filtering optimizations resulted in uneven load across ranks. However, the initial performance results with this untuned MPI implementation were promising enough to be used as performance baseline, hence no further tuning was done for load balancing or numerical optimization. Figures 5, 6, and 7 also show that the score calculation in MPI application is a costly operation as compared with the sorting and gathering best 10 thousand scores. Hence, if we needed to improve a phase that phase would be the score calculation. Measurement of the time for each stage, easily done in the

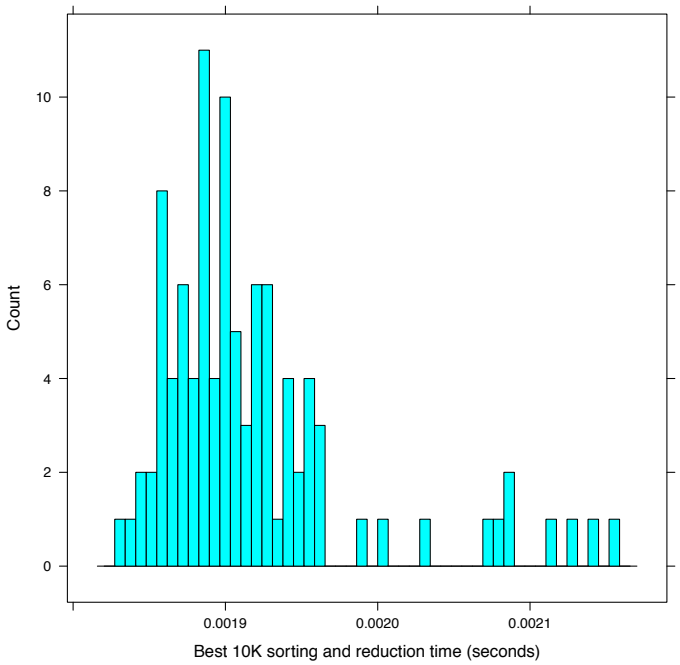

Fig. 6: The distribution of time taken to obtain the best 10 thousand scores, using MPI.

MPI implementation, is not possible in the Spark application.

For Spark, the results show the total time to classify an event. Due to the lazy evaluation, we were unable to isolate the timing results of different execution phases. The figure of merit is the number of events that can be classified per second.

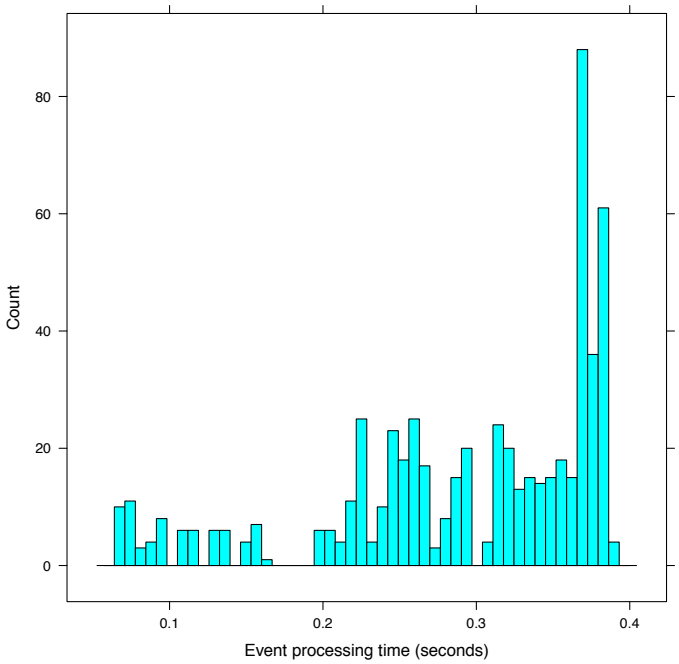

Fig. 7: The distribution of the total time to classify each event, using MPI.

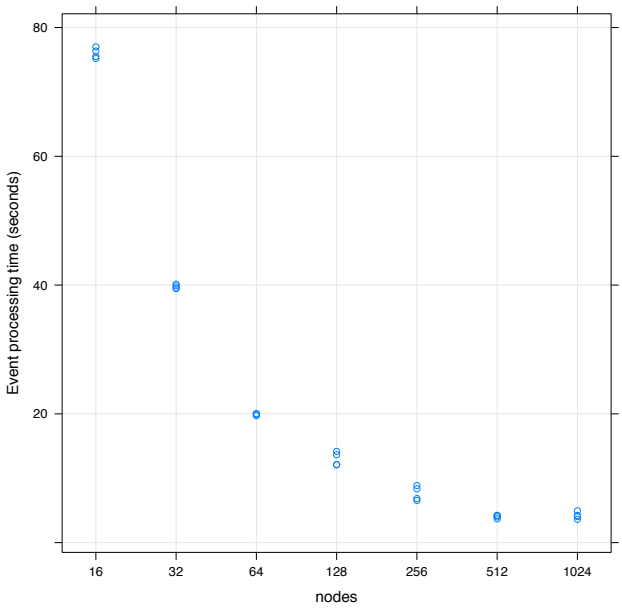

Fig. 8: The dependence of the event processing time on the number of nodes, using Spark.

We configured Spark to use 28 GB per node, and changing it to a larger value did not improve the performance. Number of cores was left at the default value (32 cores), and the number of nodes was varied. There is an overhead associated with caching that appears in the run time for the first event, these numbers are omitted from the graphs as it is a one time cost and caching reduces the event classification time of the subsequent events. Using 16 nodes it takes ≈ 76 seconds to classify an event. Investigation of the distributed tasks showed that some nodes finished in a few seconds while the slowest took almost 76 seconds. Hence both the MPI and the Spark solutions have a load imbalance. However, in Spark we can not distinguish between the time spent on filtering or score calculation or reduction.

The Spark implementation provided us with the opportunity to perform a scaling study as shown in Figure 8. The number of nodes was varied from 16 to 1024. We observed linear scaling until 512 nodes and adding more nodes after 512 does not provide any additional benefits. The untuned MPI implementation using 7 nodes is 10 times faster than the Spark implementation using 512 nodes (more than 2.5 events/s as compared with ≈ 0.25 events/s).

In the following discussion, we present a comparison of the two implementations and also explain a few reasons behind the Spark performance.

1) *Orchestration:* In the MPI implementation, the data placement, task assignment, and all the communication steps to perform local sorting and operators for global reduction was done manually. In Spark, however, data distribution and task assignment is abstracted from the user. There are functions available to perform global reduction in a distributed environment, which can be readily used. Such a set-up provides ease-of-programming in a distributed environment. It does mean, however, that the user has to rely on system optimizations provided by Spark's implementation to improve any performance.
2) *Scaling:* The Spark implementation provided good scaling without requiring any tuning to the implementation and developer expertise in parallel algorithms. 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.
3) *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. In the MPI implementation, the user has complete control of the task assignments and data placement. All the steps are well-defined, hence, performance measurement at each stage is possible as shown in Figures 5, 6, and 7. Hence, it is straight forward in the MPI implementation to tune the application.
4) *Wrapped types:* We used the Java DataFrame API, and

most of the available operations for our implementation are available through the Java wrapped types. A lot of time was spent in boxing and unboxing. It is well known that wrapped types perform much slower than the primitive types, as explained in [9]. "... Changing the declaration of sum from Long to long reduces the run-time from 43 seconds to 6.8 seconds on my machine. The lesson is clear: prefer primitives to boxed primitives, and watch out for unintentional autoboxing ..." [9]. We did a test on Cori to compare the performance of Float and float by running a simple Java program based on the calculations of the classification algorithm, and observed a performance difference of 28% in using unboxed types. Hence, we can achieve better performance with Spark if DataFrames used primitive types. In the MPI implementation, only primitive types and arrays of those types were used. Therefore, there are no performance penalties due to using unnecessary object types.

5) *Vectorized linear algebra library:* We experienced slow performance for repetitive numerical computations in Spark due to unavailability of high performance linear algebra library. In the MPI implementation, we used Armadillo, which is a high quality C++ linear algebra library. The use of advanced libraries without data conversions, and the ability to use vectorized hardware allowed MPI implementation to perform exceedingly well as compared with Spark.

V. Conclusion and Future work

We chose the LEM use case because its computational tasks and processing stages are representative many common analysis patterns in experimental HEP. These stages include matrix algebra over a filtered range of data, sorting, and summarization across the problem space. Spark and MPI are capable of implementing these kinds of use cases, as demonstrated in this paper. Each requires a different computing model than is currently in use in HEP.

Experimental HEP's current computing model requires single-process (and mainly single core) applications running at grid sites, operating on large files that have been staged by data handling systems. Both the Spark and the MPI solution require a different computing model, including a distributed dataset that ultimately resides in memory, and a run-time configurable number of compute resources. The current model also does not adequately address multi-core scheduling.

The Spark implementation has shown excellent scaling given our limited dataset size. We find that Spark also has a good programming model that hides many of the difficulties in achieving good scaling. We were able to achieve nearly perfect scaling up to 512 nodes (or 16K cores) on Cori, without being concerned about the intricacies of parallel programming. However, the relative efficiency as compared to the MPI solution is abysmal. The per-core performance of the untuned MPI application is approximately 500 times greater that the best performance we observed from the Spark implementation.

For this technology to benefit our science, we would need improved floating-point performance available to us within Spark, especially but not only for linear algebra. We also need to be able to leverage the vectorization available on many modern CPUs. In addition, we need better facilities for profiling performance to guide our optimization efforts. We suspect that the use of wrapped types leads to the inability to make direct use of native high-performance libraries, which impacts the overall performance. Comparison with our MPI implementation shows what the hardware is capable of. Increasing the computational efficiency of Spark while retaining the simple programming model would make Spark an attractive approach to HEP's traditional programming model.

References

[1] Armadillo - C++ linear algebra library. http://arma.sourceforge.net.

[2] Dataframe API. http://spark.apache.org/docs/latest/sql-programming-guide.html.

[3] Library Event Matching event classification algorithm for electron neutrino interactions in the NOvA detectors. http://arxiv.org/abs/1501.00968v2.

[4] NOvA Neutrino Experiment. http://www-nova.fnal.gov.

[5] Spark. https://spark.apache.org.

[6] Spark Distributed Analytics Framework at NERSC. https://www.nersc.gov/users/data-analytics/data-analytics/spark-distributed-analytic-framework.

[7] 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.

[8] Pushpalatha C. Bhat. Advanced analysis methods in high-energy physics. *AIP Conf. Proc.*, 583:22–30, 2001. [,22(2001)].

[9] Joshua Bloch and Guy L. Steele. *Effective Java : programming language guide*. The Java Series. Addison-Wesley, Boston (Mass.), Toronto, Paris, 2001. La couv. porte : Foreword by Guy Steele.

[10] R. Brun and F. Rademakers. ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth.*, A389:81–86, 1997.

[11] 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.