FERMILAB-CONF-17-437-CD
ACCEPTED

Python and HPC for High Energy Physics Data Analyses

S. Sehrish
Fermi National Accelerator Laboratory
Batavia, Illinois
ssehrish@fnal.gov

J. Kowalkowski
Fermi National Accelerator Laboratory
Batavia, Illinois
jbk@fnal.gov

M. Paterno
Fermi National Accelerator Laboratory
Batavia, Illinois
paterno@fnal.gov

C. Green
Fermi National Accelerator Laboratory
Batavia, Illinois
greenc@fnal.gov

ABSTRACT

High level abstractions in Python that can utilize computing hardware well seem to be an attractive option for writing data reduction and analysis tasks. In this paper, we explore the features available in Python which are useful and efficient for end user analysis in High Energy Physics (HEP). A typical vertical slice of an HEP data analysis is somewhat fragmented: the state of the reduction/analysis process must be saved at certain stages to allow for selective reprocessing of only parts of a generally time-consuming workflow. Also, algorithms tend to to be modular because of the heterogeneous nature of most detectors and the need to analyze different parts of the detector separately before combining the information. This fragmentation causes difficulties for interactive data analysis, and as data sets increase in size and complexity ($O10$ TiB for a "small" neutrino experiment to the $O10$ PiB currently held by the CMS experiment at the LHC), data analysis methods traditional to the field must evolve to make optimum use of emerging HPC technologies and platforms. Mainstream big data tools, while suggesting a direction in terms of what can be done if an entire data set can be available across a system and analysed with high-level programming abstractions, are not designed with either scientific computing generally, or modern HPC platform features in particular, such as data caching levels, in mind.

Our example HPC use case is a search for a new elementary particle which might explain the phenomenon known as "Dark Matter". Using data from the CMS detector, we will use HDF5 as our input data format, and MPI with Python to implement our use case.

CCS CONCEPTS

• **Applied computing → Physics**; • **Software and its engineering → Software performance**; *Software usability*;

KEYWORDS

HEP analysis, MPI, Python, HPC, HDF5, numpy, pandas, mpi4py

PYHPC2017, Nov 2017, Denver, Colorado USA

ACM ISBN 978-1-4503-5124-9…$15.00
https://doi.org/10.1145/3149869.3149877

ACM Reference Format:
S. Sehrish, J. Kowalkowski, M. Paterno, and C. Green. 2017. Python and HPC for High Energy Physics Data Analyses. In *PyHPC'17: PyHPC'17: 7th Workshop on Python for High-Performance and Scientific Computing, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3149869.3149877

1 INTRODUCTION

The field of experimental High Energy Physics (HEP) is concerned with the design, creation, running and analysis of data from machines which will improve our understanding of the fundamental particles and the forces between them. The interactions studied are statistical in nature, so we have always been relatively demanding of computational, data storage and transfer resources in order to measure properties of the particles known exist, or to discover those which have hitherto only been predicted by theory. As the field advances, these demands have only increased as we attempt to discover and characterise ever-more-elusive particles and interactions. From the discovery of the Higgs boson at the Large Hadron Collider (LHC) in Switzerland where over 300 trillion (3×10^{14}) proton-proton collisions were required for analysis [5], the upcoming high luminosity run [2] (dubbed "HL-LHC") will generate interactions for study at nearly 8 times the rate of the previous run, each of which will require significantly more data to describe, allowing us to push for a deeper understanding of the Higgs boson and its implications for the fundamental laws of nature.

An HEP data analysis workflow can generally be subdivided into Data Acquisition (DAQ), event reconstruction, and data analysis stages, the results of each being saved to storage separately. Dedicated hardware-adjacent systems are generally responsible for the collection of data and "event building"—structuring the data for each particle bunch interaction—and the initial saving of the data to the storage system for subsequent stages. These raw data are refined during the event reconstruction phase to produce physically-significant information such as particle trajectories, particle identification, and energy measurements. This step generally employs time-consuming algorithms and may actually produce a larger data set than the original raw data from which it is derived. Reconstruction, like DAQ, is usually a process coordinated across the experimental collaboration and while the algorithms may evolve over time, there is often only one such workflow per experiment. Data analysis is usually tailored for an individual physical interaction process, result or paper, but generally includes data reduction

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

via selection and statistical summarization, and the creation of exploratory plots prior to the production of the final result. There are thus often many data analysis workflows per experiment, and they are individually generally more demanding of I/O than compute cycles, with the notable exception of deep learning training operations for certain analysis algorithms.

A given data analysis workflow will usually have multiple data reduction steps necessary to produce a data set suitable for interactive analysis. The execution time of such workflows vary in duration up to weeks, and multiple iterations of this reduction process may be necessary during the refinement of an analysis. Figure 1 describes these steps for the example use case, and the data storage requirements for the intermediate stages. Our overarching aim is to enable interactive in-memory analysis of data volumes of the order of that expected from HL-LHC while minimizing where possible costs of intermediate storage and the need for the handling of large numbers of files. High performance Python looks promising for our purpose: it provides a familiar programming environment to the physicist, it provides efficient ways to deal with the data we have formatted for the HPC platforms, and it provides efficient vectorized operations to doing these data analyses.

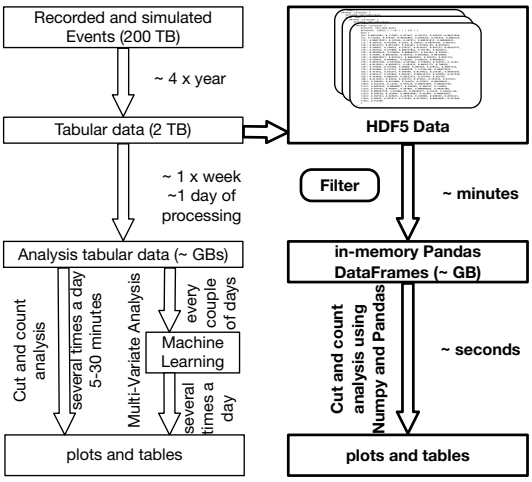

Figure 1: The left-hand side shows the current flow of operations and data, the frequency of each operation, and time taken by each operation, for the CMS Dark Matter analysis. Our proposed design is shown on the right-hand side [9].

A secondary goal of the study is to close in on a paradigm that will allow physicist programmers to construct and refine an analysis workflow without the need to develop expertise in parallel programming. This notional toolbox will have different aspects for different styles of analysis, but the common thread will be ease of use for the physicist whose primary interest is developing their analysis rather than any broad or deep knowledge of HPC programming techniques and technologies.

In Section 2, we explain the CMS Dark Matter search use case, and current approach and computing available to perform this analysis. In Section 3, we briefly describe the fundamental concepts of HPC I/O used in this paper, and provide details about the input data format and analysis encoding. Section 4 explains the experimental setup, and discusses results. We provide a list and discussion of lessons learned in Section 5. We discuss the conclusion and future work in Section 6.

2 SCIENCE USE CASE: CMS DARK MATTER SEARCH

The CMS detector is a physically large, heterogeneous detector capable of measuring various properties of particle collision products, such as charged particle tracks, the energy depositions of charged and neutral particles in calorimeters, collision product identity and mass. An example of a particle bunch collision ("event") in CMS is shown in figure 2. Due to the probabilistic nature of the science behind such collisions, huge numbers of them are required to be able to make statistical observations about the physical processes these events depict. The particular use case studied in this work is a the search for a hypothetical particle that may be responsible for Dark Matter in the universe, the gross effects of which can be seen in the rotation of galaxies, and the dynamics of intergalactic collisions. One possible indicator of the existence of such a particle is the production of events containing a single top quark. Given a description of the detectable signature of such an event, the analysis must search the entire data set to obtain statistically significant evidence for discovery, or based on the amount of data examined, to place limits on the likelihood of the existence of the postulated particle. One would reduce the data based on attributes of the postulated signature such as expected spatially-isolated electrons within a particular momentum range, accompanied by higher momentum muons and a relatively low measurement of the "missing energy", which is a measure of the imbalance of momentum in the particles observed in the detector.

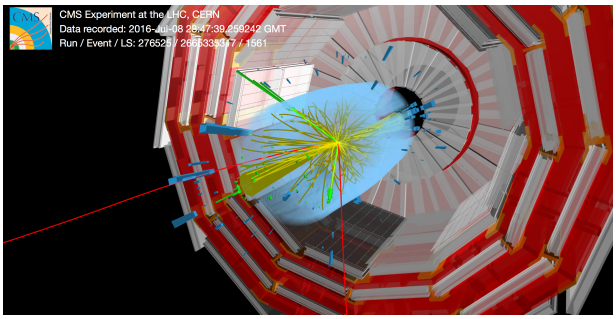

Figure 2: A collision inside the CMS detector. Identified particle tracks are color coded, with red for muons and green for electrons [7].

The data to be analyzed is from the 2015 data set of about 200 TiB, comprising both DAQ-obtained data and simulated events produced with Monte Carlo techniques. With HL-LHC, the data volume available for the analysis will be significantly greater. For our analysis, the required components of the data are converted to "n-tuples", flat data structures of homogeneous collections of mainly integers and floating point numbers. Each row of the n-tuple contains a

metadata description and identity of a particle, and its properties. Because each event may contain several identified particles of a given type, multiple rows may be necessary to completely describe an event. Data reduction filters are applied to reduce further the data size from ≈ 2 TiB to a few GiB, suitable for interactive analysis. Quantities from this reduced n-tuple are then summarized, combined and plotted to look for evidence of the postulated particle against the representative background of simulated data where the postulated particle is known to be absent. The whole workflow can take from days to weeks to execute depending on the amount of data examined and the iterations required to refine the analysis.

3 DESIGN AND IMPLEMENTATION

Our proposed approach for the CMS analysis is contrasted with the current CMS approach in Figure 1. The input data in the original workflow is stored using ROOT [4], which is the most common data format in HEP. We have chosen to use the Hierarchical Data Format, version 5 (HDF5) [10] because of the widespread availability and support of optimized installations at HPC centers, and because of the wide variety of analysis and visualization tools that can efficiently access data stored in this format.

The Python library pandas [8] provides a popular high-level set of facilities for transformation and statistical analysis of data. The pandas library provides a class DataFrame that represents a table of data. The use of DataFrames, which in turn use numpy ndarrays, allows us to make use of vectorized operations in data transformations. In addition, the Python library mpi4py [3] provides a high-level interface to MPI that makes it sufficiently easy to use for programmers without expertise in MPI, which is our target community.

We discuss the input data, its representation in HDF5, reading and analysis implementation in the following subsections.

3.1 Tabular data storage using HDF5

The HDF5 libraries can be used in a wide variety of manners; it does not mandate any single usage pattern. Because we are interested in writing tabular data, and in reading that data using a variety of programming language tools, we have chosen to use a specific format that makes both writing and reading of the data simple and efficient.

In each HDF5 file, we store one or more tables. The top-level *root node* of the HDF5 files we write contain only these tables. This makes discovery of the number and names of tables in our files simple.

Each table is implemented as an HDF5 *group*. This allows us to attach any relevant metadata, and to have an arbitrary number of columns in the table. Our own data writing API makes sure that all columns in a given table have the same length. This organization makes it simple to discover the number and types of the columns in a table.

Each column is implemented as an HDF5 *dataset*. Our format allows for datasets that contain multi-dimensional arrays. For the use case described in this paper, such arrays are not needed; however, for other use cases they are necessary. Because the dataset carries information about the dimensionality of the arrays stored in it, the code reading the dataset does not have to have knowledge of what had been written.

The combination of choices in this design allows us to write generic code, in a variety of languages supported by HDF5, to read any file written in our format. The code does not need to have been written with knowledge of the names of tables, nor with knowledge of the number and types of columns in each table.

Figure 3 shows the organization of the CMS data in the ROOT files, where each row contains all the data for different particles and their properties in an event. In the HDF5 format, we store the data for each particle type in a table named for that particle type, and each property of the particle as a column in that table, as shown in Figure 4. Since, in this format, the rows do not correspond to individual events, we also need to store a column that identifies to which event each particle belongs.

In addition to the particles, the general general properties of an event are also stored in a different table. Each table can be processed independently and in parallel. Additionally, column-oriented transformations and reductions can be executed in parallel.

Event ID	Event Info		Electrons		Taus	
1	met	weight	pt	eta	pt	eta
	150	0.5	130	0.4	17	0.1
			50	1.3	55	0.3
					44	1.9
2	met	weight	pt	eta	pt	eta
	210	0.65	67	-0.5	34	1.5
			87	1.9	44	0.3

Figure 3: An example of the CMS data organization in the current ROOT format. Each row represents an event. Each event has two types of particles; electrons and taus [9].

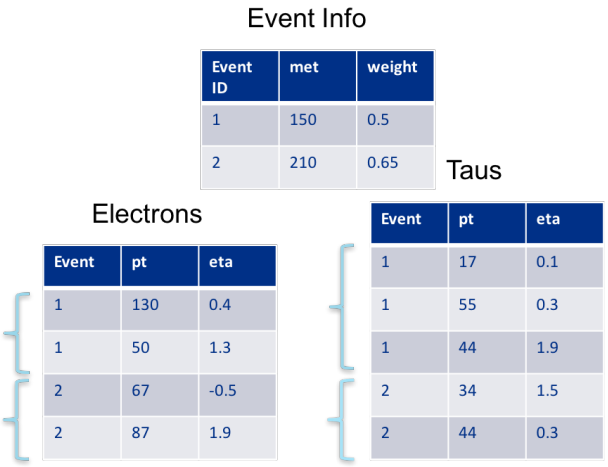

Figure 4: The data of figure 3, in our HDF5 organization [9].

3.2 Reading tabular data into pandas DataFrames

Our HDF5 file format can store data that pandas is unable to represent; in particular, our columns can carry not only scalar values, but also multi-dimensional arrays. Thus we are unable to use the facilities that pandas provides to read HDF5 files. However, using the Python package h5py [1], we are able to read the columns into numpy ndarray objects. When, as in this use case, the columns contain only scalar values, the set of ndarrays read from the columns of one table can be efficiently converted into a DataFrame. The h5py package is capable of using HDF5's MPI I/O support, which we take advantage of.

Each column in a table is divided into equal chunks of a size that depends on the number of MPI processes; using mpi4py [3] makes this task simple. The analysis task requires that each MPI rank has a portion of all the columns needed; each rank has different rows, and no data are shared. Using h5py, each MPI rank reads a contiguous portion of each column in a table into a numpy ndarray. We put all the ndarrays corresponding to columns of a table into a Python dictionary. This dictionary is converted to a pandas DataFrame. Figure 5 shows an example of this read operation.

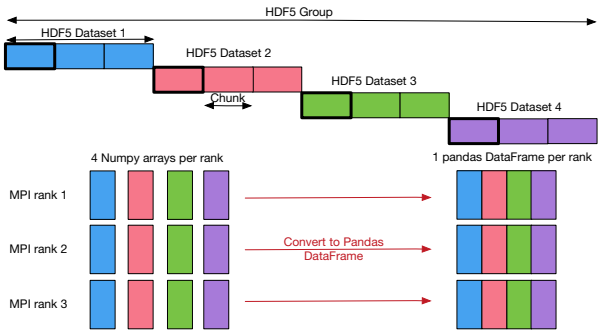

Figure 5: An example of parallel reading and creation of a pandas DataFrame, using three MPI ranks. There are four columns in the example table; a portion of each column is read into a numpy ndarray by each of MPI ranks. Each MPI rank creates one pandas DataFrame from its ndarrays.

3.3 Encoding Analysis

The organization of data determines what APIs can be used to efficiently implement the analysis of those data. Our analysis task involves mutations (defining new quantities using one or more columns in a DataFrame), filtering operations across multiple columns within a DataFrame, and reduction operations on the resulting data. Use of column-based (numpy vectorized) operations is critical for good performance. Parallel execution of the transformations and filtering does not require writing of any additional code, because each MPI rank is processing an independent DataFrame. Parallel execution of reduction operations is made easy by the high-level mpi4py API (we note in section 6 how it can be made even easier).

Listing 1 shows an example of parallel filtering code; the full analysis requires tens of such filtering operations. Note that the parallelism is implicit: no special code needs to be written to parallelize the operation because each MPI rank is processing data independently of every other rank.

```
def filter_electrons(df):
    f =  ((iso < 0.126*df.pt)
          |(dEtaIn < 0.01520)
          |(df.sieie < 0.01140))
          & f_scEta0
    return df[(abs(df.eta) < 2.5)
              &    (df.pt >= 10)
              &    f]
```

Listing 1: Example of the filtering criteria for electrons.

In this example, `df` is the electrons DataFrame. `sieie`, `eta` and `pt` are three of the columns in the DataFrame. `iso`, `dEtanln`, `f_scEta0` are new quantities calculated by transformations omitted from the code for clarity. For such transformation and filtering, no MPI synchronization or communication is needed, and so we would expect excellent scaling.

Listing 2 shows the code for filling a histogram. Each rank calculates a local histogram; all have the same binning, so combination of the local histograms into a single global histogram can be done with an MPI reduction that does a bin-by-bin addition of the counts in each histogram. The code shows the two-step process: first, filling of the local histogram, and second the reduction. While this does not require much MPI expertise, we view this as a place where we can make improvement for our user community.

```
fdf = filter_electrons(df)
bins = [0, 20, 30, 40, 50, 100, 500]
myhist, _ = np.histogram(fdf['pt'],
                         bins = bins)
hist = MPI.COMM_WORLD.reduce(myhist,
                             op = MPI.SUM,
                             root = 0)
```

Listing 2: Histogramming of electron transverse momenta.

4 RESULTS AND DISCUSSION

We ran tests on both Cori Phase I (Haswell compute nodes) and Phase II (KNL compute nodes). We used 2–64 nodes on Cori Phase I, and 2–32 nodes on Cori Phase II, in all cases using one physical core per MPI rank. Each Haswell node has two sockets, and each socket is populated with a 16-core Intel Xeon Processor E5-2698 v3 (Haswell) with a clock speed of 2.3 GHz. Each node has 128 GiB DDR4 2133 MHz memory (four 16 GiB DIMMs per socket). Each core has its own L1 (64 KiB) and L2 (256 KiB) caches, and a 40 MiB shared L3 cache per socket. Each KNL node has 68 cores and 96 GiB DDR4 2400 MHz memory per node. Each node has 16 GiB of on-package, high-bandwidth multi-channel DRAM (MCDRAM). We have used both of the flexible memory modes available, *cache mode* (effectively an L3 cache) and *flat mode* (a unique NUMA domain, separate from DDR4).

The CMS data are stored uncompressed in one 506 GiB file. The file is striped using 40 Object Storage Targets on the Lustre file system. The data consists of information about all the particles found in the events. In this example, however, we only use electrons, along with one event property. There are a total of 360 million events containing 180 million electrons, so we need to hold ≈ 60 GiB of data in memory for this analysis. The analysis we implemented creates a histogram of the transverse momentum of the leading electron (the electron with the greatest transverse momentum) in events that passed the filtering criteria. The filtering criteria are applied to tens of properties of electrons, and several new quantities are created as part of the processing. We extensively used the pandas-supplied logical operators on columns in the DataFrames.

Figure 6 shows the scaling behavior of the reading of the CMS data from the HDF5 file, as we vary the number of nodes used in the processing. In each case, we measured the time taken to read the same data file. For the Cori Phase I and Cori Phase II (cache) configurations, we repeated each measurement three times; due to time constraints we were able to complete only one such set of measurements for the Cori Phase II (flat) configuration. Each panel shows the data for a different platform. We plot the processing speed (the number of records processed per second) versus the number of nodes used. If the scaling were perfect, each of the panels would show a straight line going through the origin. The line drawn on each panel is a linear fit to the data, for a line with an intercept of zero. The dominant feature of these plots is that the read times are wildly variable: for a given number of nodes, reading speeds can vary by more than an order of magnitude. We suspect a significant contribution to the difference is something that users of HPC systems can not, in general, control: the load on the global filesystem. Verification of this would require access to performance data for the system at the exact time at which the program in question is running. The variability is greater on the Haswell nodes than on the KNL nodes. The maximum reading speed achieved by the Haswell nodes is much greater than that achieved on the KNL nodes. This is consistent with the observations of [6], which, however, concentrates primarily on writing performance.

The variability of the results on the Haswell nodes was such that we performed additional studies, described below. For the Cori Phase II (cache) nodes, the scaling appears to be worse than linear, especially after 8 nodes. For the flat mode, we have insufficient data to judge. The KNL nodes in cache mode show dramatically poorer read performance than do the Haswell nodes. In flat mode, the KNL node performance improves, and is plausibly equivalent to the Haswell nodes, indicating that the I/O stack may be taking advantage of the special memory available in this mode. We note that nothing in our own code was specialized to take advantage of this.

Figure 7 shows the scaling behavior of the analysis portion of the task, which is the in-memory processing. For each rank, this is after all the reading is done; however, it is possible and even likely for one rank to finish reading and move on to analysis before another rank has finished reading (when no barriers are used). We note that most of the in-memory work is done independently for each rank; only the reduction of each ranks's histogram to produce the resulting single histogram requires communication between ranks. The layout of the plot is similar to that of figure 6.

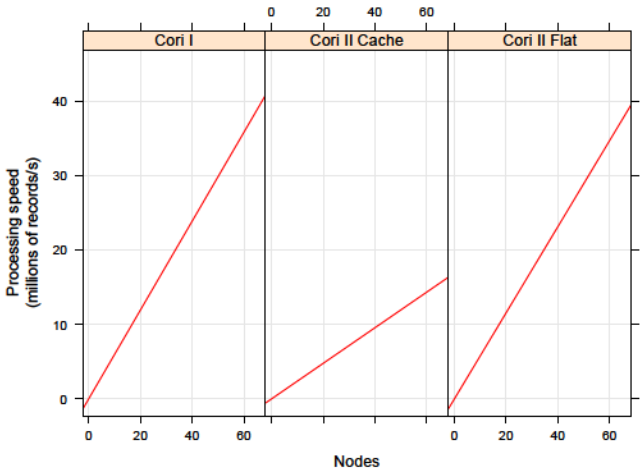

Figure 6: Scaling performance of reading CMS data from and HDF5 file, for Cori Phase I and Phase II.

Comparison between these two figures shows that the reading is much slower than the analysis, in all cases. The analysis scales reasonably well out to 32 nodes for both Haswell and KNL, and for both memory modes of the KNL nodes. At 64 nodes, the variability in performance is large. Comparing the KNL cache and flat modes, we see that for analysis the cache mode provides uniformly superior performance, albeit by a small margin. Unlike the reading task, nothing in the software stack takes advantage of the special memory made available in flat mode; thus the larger cache available in cache mode yields the improved performance.

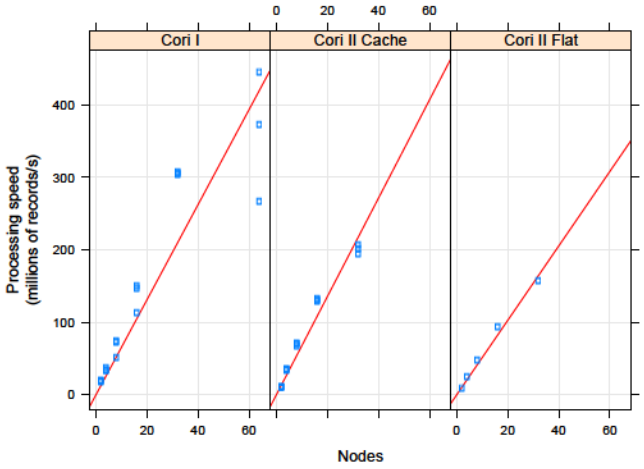

Figure 7: Scaling of processing speed for analysis of CMS data in memory, for Cori Phase I and Phase II.

Figure 8 shows another view of the same data, but displayed in a fashion to highlight any deviations from perfect scaling. This figure shows how the total computing work done (measured in node-seconds, rather than the more common CPU-hours) changes with the number of nodes. For a program that scales perfectly, the

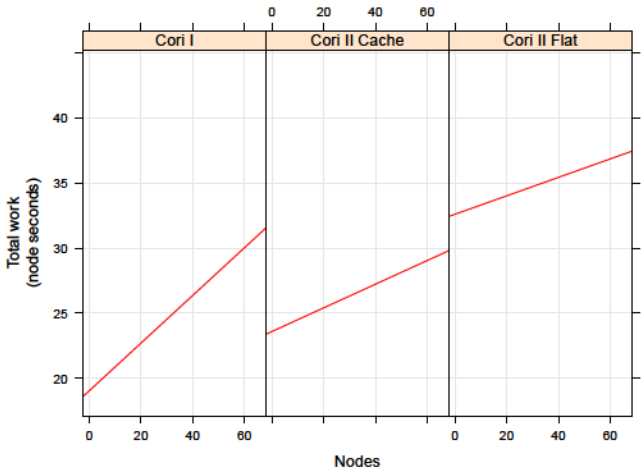

Figure 8: Scaling total work done for analysis of CMS data in memory, for Cori Phase I and Phase II.

amount of work that needs to be done does not change as the number of nodes used for the task is increased; thus a program that scales perfectly would yield performance data that would form a horizontal line. For the Haswell nodes, the variability of the data make it difficult to reach a clear conclusion, except that the performance at 64 nodes is clearly worse than that at 32 nodes. For the KNL nodes, in both modes, using either too few or too many nodes results in an increase in the amount of work that needs to be done. This is especially dramatic when only two nodes are used. The improved memory locality obtained by using an appropriate number of nodes provides superior performance. In comparing the cache and flat modes, we also see the clear advantage of the additional local memory available in the cache mode.

Because of the wide variations in read performance seen in figure 6, we studied the read performance on the Haswell nodes in more detail. First, in order to avoid any migration of processes between cores, we used the `--cpu_bind=rank` option of the `srun` command. We also explored whether the lack of coordination between ranks might have been responsible for the large variations in read performance. We chose three strategies for controlling synchronization by placement of MPI barriers. The first strategy, labeled *nobarrier*, employed no barriers; this reproduced our original running conditions. The second strategy, labeled *1barrier*, employed a single MPI barrier placed before the reading of the data. This ensured that the start-up time of different ranks (observed to be up to $\sim$ 60 seconds) did not cause the read step to begin at different times for different ranks of the same program. The third strategy (labeled *2barrier*) used barriers before and after the reading; this ensured that the in-memory processing was not begun by any rank until all ranks had finished reading. For each of these strategies, we measured the performance of both 8- and 32-node jobs. We chose these jobs because the 8-node jobs were the largest that showed relatively modest variation in read time, and the 32-node jobs showed very dramatic variation. For each of these configurations, we ran two batches (each submitted as an independent batch job) of 8 runs each. Finally, we instrumented the program to collect data on all ranks of the program independently (the previous runs collected data only on rank 0 of each program). We note that all of the runs of a given batch were executed within 30 minutes of each other. The different batches were separated in time by much larger intervals.

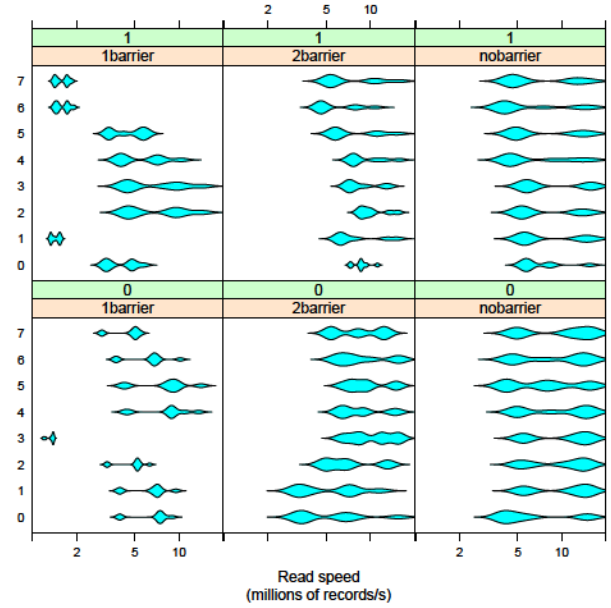

Figure 9: Distribution of read speeds for each MPI rank, for 8-node runs on Cori I (Haswell nodes). The green ribbon shows the batch job number, and the tan ribbon shows the synchronization strategy. Each panel corresponds to a single batch job. Each line corresponds to a single program run, and shows the distribution of reading speeds for each of 256 ranks. Note the x-axis is logarithmic.

Figure 9 shows the results of these measurements for the 8-node runs, and figure 10 show the results for the 32-node runs. Each numbered panel in the plots shows the results of one of the 8 runs. The data are displayed as a *violinplot* [11], which shows the distribution of processing speeds (as estimated by a Gaussian kernel density estimate). Each "violin" denotes 256 measurements (one per rank) for the 8-node runs and 1024 measurements for the 32-node runs. Our first observation is that all of the distributions, save one, are multi-modal; such data are not well-described by a single statistic such as a mean or median. For the 8-node data, the *1barrier* runs show somewhat worse performance than do either of the other configurations, and dramatically so for a few of the runs.

For the 32-node data, the difference between the *1barrier* and *2barrier* results is even more dramatic, but the *1barrier* reads are faster. Most remarkable is the observation that the variation between ranks within a single program execution was neglible for the *1barrier* strategy. Figure 11 shows the details of one of these runs; note the extremely small range of the x-axis.

Using the performance data we collected for the 8- and 32-node runs, we looked in more detail at the in-memory processing. Rather than measuring the speed for the entire in-memory processing,

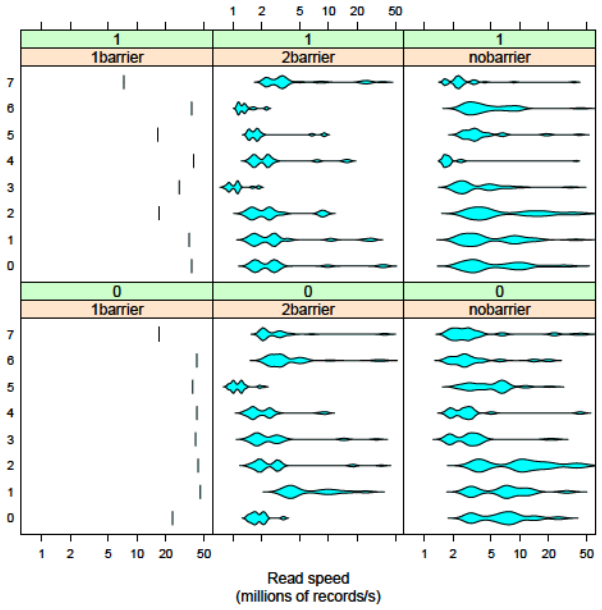

Figure 10: Distribution of read speeds for each MPI rank, for 32-node runs on Cori I (Haswell nodes); see figure 9 for details. In this figure, there are 1024 ranks per program run.

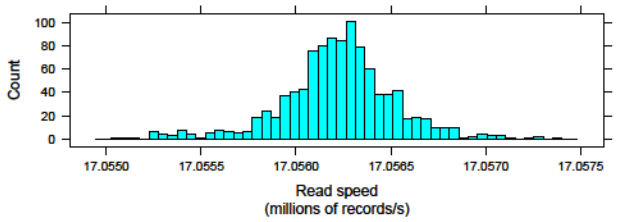

Figure 11: Distribution of read speeds for each MPI rank, for run 2, batch 1, of the 32-node run on Cori I (Haswell nodes), for the *1barrier* strategy.

we measured each step (merging of DataFrames, selection of electrons, per-rank histogramming, and reduction to a single histogram) separately. We observed the slowest of the steps was the merging.

Figures 12 and 13 shows the measurements of the speed for the merging step. These data are single-modal, and show a much smaller fractional variation than do the reading speeds. We observe no significant difference between the different syncronization strategies. The 32-node runs process at approximately 4 times the rate of the 8-node runs, consistent with the excellent scaling show in figure 7.

5 LESSONS LEARNED

Input data format and organization: We now have data organized within the file to facilitate reading for analysis. A wide variety of

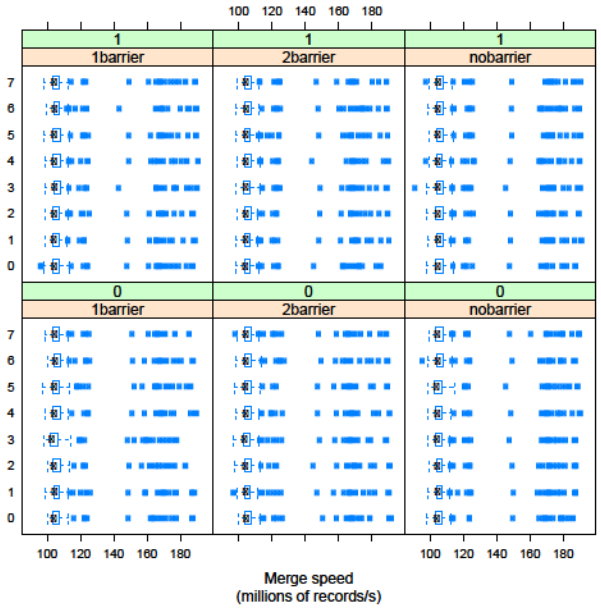

Figure 12: Distribution of merge speeds for each MPI rank, for 8-node runs on Cori I (Haswell nodes). The green ribbon shows the batch job number, and the tan ribbon shows the synchronization strategy. Each panel corresponds to a single batch job. Each line corresponds to a single program run, and shows the distribution of merge speeds for each of 256 ranks. Note the x-axis is logarithmic.

HEP data are easily stored in tabular format, including an overwhelming majority of data used in interactive analysis. It is worthwhile storing data in HDF5 groups in a way that can be easily represented in memory, with a form that makes it easier and more efficient to describe operations on these data. Such an organization allows reading and visualization to be simplified as well.

Data distribution: The number and size of files determine how to distribute data among MPI ranks. If there is one large file, it is straightforward to divide the data among all MPI ranks; if there are a large number of small files, each MPI rank can process one file. Our original data was organized into several files of different sizes. We consolidated data into one large file, where file size ranged from hundreds of MiBs to 100 GiB. The code to read one large file is simpler and efficient.

High Level API: Many operations commonly used in HEP analysis tasks can be directly described using the high-level APIs provided by numpy and pandas DataFrame API. The resulting code is easy to read and efficient as well.

Application tuning: The first version of the code we wrote, for both reading and analysis, had substantially worse performance than the version presented in this paper. Because the development environment and tools provided by Python were sufficiently easy to use, we were able to quickly identify performance bottlenecks. Such ease-of-development for non-experts is a critical advantage for our community.

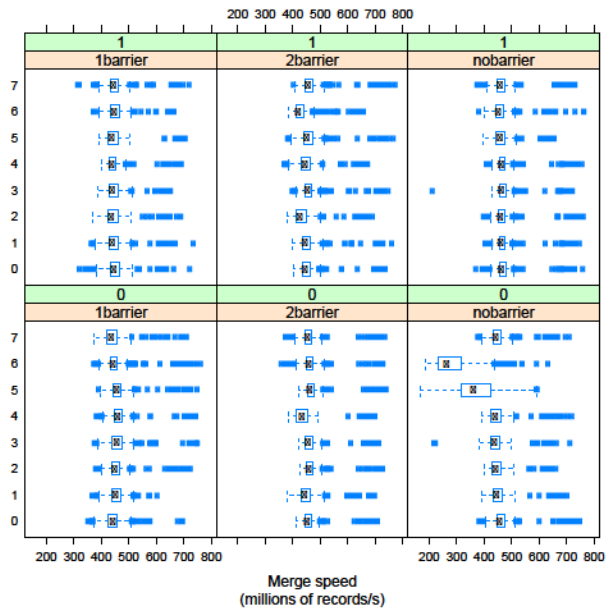

Figure 13: Distribution of merge speeds for each MPI rank, for 32-node runs on Cori I (Haswell nodes); see figure 12 for details. In this figure, there are 1024 ranks per program run.

Performance evaluations: Many HEP analysis tasks are very demanding of the I/O system. Performance evaluation of such processing is made difficult by the fact that the global filesystem is a shared resource, and is thus influenced by things other than the processing being measured. In order to understand the performance of such programs, access to performance data for the machine as a whole seems necessary.

6 CONCLUSION AND FUTURE WORK

The processing stages in the CMS Dark Matter analysis represent an important class of problems in experimental HEP. These stages include applying several selection and filtering criteria and plotting. HEP scientists use Python for several data processing tasks, providing them with data layouts to enable efficient analysis operations in Python is promising.

We will study scalability with a data set that is 100 times larger. and performance with the implementation of the same use case in Spark, an industry standard "big data" analysis system. We will also implement more use cases from HEP analyses. Read performance on the KNL nodes will also be investigated; it may not be best to read data on KNL nodes from all active ranks. A better organization may be to read from few ranks, or even one rank, per node. We will work on removing the explicit synchronization and communication aspects from the user code (as was visible in the reduction step of the histogramming example) by providing interfaces that abstract away the complexities of parallel code.

7 ACKNOWLEDGMENTS

We would like to thank our team at Fermilab: Oliver Gutsche, Matteo Cremonesi, Bo Jayatilaka, Cristina Mantilla, Alec Buchanan, and at Princeton University: Jim Pivarski, Alexey Svyatkovskiy for providing details on the science use case and computing model. This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics. 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] HDF5 for Python. http://www.h5py.org.
[2] HL-LHC: High Luminosity Large Hadron Collider. http://hilumilhc.web.cern.ch.
[3] MPI for Python. http://mpi4py.scipy.org.
[4] R. Brun and F. Rademakers. ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth.*, A389:81–86, 1997.
[5] 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.
[6] Jialin Liu, Quincey Koziol, Houjun Tang, Francois Tessier, Wahid Bhimji, Brandon Cook, Brian Austin, Suren Byna, Bhupender Thakur, Glenn Lockwood, et al. Understanding the io performance gap between cori knl and haswell. In *Cray User Group Meeting*, 2017.
[7] Thomas Mc Cauley. Higgs boson produced via vector boson fusion event recorded by CMS (Run 2, 13 TeV). CMS Collection., Aug 2016.
[8] Wes McKinney. pandas: a foundational python library for data analysis and statistics.
[9] Saba Sehrish, Jim Kowalkowski, and Marc F. Paterno. Spark and HPC for high energy physics data analyses. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017*, pages 1048–1057, 2017.
[10] The HDF Group. Hierarchical Data Format, version 5, 1997-2017.
[11] Wikipedia. Violin plot. https://en.wikipedia.org/wiki/Violin_plot.