Data-parallel Python for High Energy Physics Analyses

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

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

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

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

ABSTRACT

In this paper, we explore features available in Python which are useful for data reduction tasks in High Energy Physics (HEP). High-level abstractions in Python are convenient for implementing data reduction tasks. However, in order for such abstractions to be practical, the efficiency of their performance must also be high. Because the data sets we process are typically large, we care about both I/O performance and in-memory processing speed. In particular, we evaluate the use of data-parallel programming, using MPI and numpy, to process a large experimental data set (42 TiB) stored in an HDF5 file. We measure the speed of processing of the data, distinguishing between the time spent reading data and the time spent processing the data in memory, and demonstrate the scalability of both, using up to 1200 KNL nodes (76800 cores) on Cori at NERSC.

CCS CONCEPTS

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

KEYWORDS

HEP analysis, MPI, Python, HPC, HDF5, numpy, mpi4py, h5py

ACM Reference Format:
M. Paterno, C. Green, J. Kowalkowski, and S. Sehrish. 2018. Data-parallel Python for High Energy Physics Analyses. In *PyHPC'18: PyHPC'18: 8th Workshop on Python for High-Performance and Scientific Computing, November 12, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3149869.3149877

1 INTRODUCTION

The traditional programming model in High Energy Physics (HEP) is serial. Parallelism is usually introduced at the level of the workflow, rather than as part of the programming model. Recently, some computing in HEP has moved to using shared-memory task-parallel programming. A shared-memory task-based parallel program is, by nature, limited to running on a single node, and does not handle distribution across a large machine. Currently, a separate workload-management system controls many program executions to distribute the work across a large machine.

HEP has paid less attention to data parallelism. However, the same features of HEP data that make parallel workflows feasible also invite data-parallel programming. A data-parallel programming model allows a single program execution to span as much of a large machine as is convenient for the problem being solved. Data parallel programming also allows most or all of the parallelism in user code to be implicit.

In this paper, we explore the support for data-parallel programming in the Python language. We concentrate on a use case early in the HEP data processing workflow, a step in which the entire data set collected by the experiment is processed. Our data sets are typically large, *e.g.* DUNE [1] expects to collect 30 PiB of raw data per year. Refinements in analysis algorithms and improvements in understanding of the detector calibrations make it necessary to re-process these large data sets several times. Thus it is becoming increasingly important to harness the power of national HPC facilities for the timely processing of HEP data.

In section 2 we describe the scientific use case on which we based our work. In section 3 we describe the specific workflow we have implemented, and in section 4 we describe the theory behind the analysis technique implemented in this workflow. In section 5 we describe the design and its implementation, and in section 6 we describe and analyze the performance measurements we have made on our implementation. In the final sections we draw some conclusions, and discuss how we plan to continue this work.

2 SCIENCE USE CASE

Neutrinos are among the most abundant—and least understood—particles in nature. Neutrinos come in three types, called flavors, which are named for the particles with which they are associated: electron, muon, and tau. Neutrinos are known to transition from one type to another, a phenomenon called *oscillation.* Understanding neutrino oscillations may help scientists to understand why matter exists, and why the universe is filled with matter rather than radiation.

Neutrino oscillations have been observed in several environments, most recently by NOvA [2]. The NOvA experiment has the largest detector in its class, that of accelerator-based neutrino experiments. Such experiments work by directing an intense beam of neutrinos generated by a particle accelerator at a detector designed

PYHPC2018, Nov 2018, Dallas, TX USA

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

to observe the interactions of the generated neutrinos. Because neutrinos interact so rarely, intense neutrino beams and very large detectors are needed to observe them. In addition, the oscillation length of neutrinos can be large, so geographically-dispersed detectors are necessary to compare and contrast results. Improvement in the precision of the measurements requires more intense beams, ever-larger detectors, and increasingly accurate measurement and identification of particle interactions in the detector. The DUNE detector, for example, will be the largest neutrino detector ever built. It will consist of two separate detectors: a "near detector" at Fermilab, the source of the neutrino beam, and a much larger "far detector" about 800 miles away, underground in a mine near Lead, South Dakota. The DUNE far detector will contain 68 thousand tons of liquid argon. Most recently-designed neutrino detectors, including DUNE, are liquid argon time-projection chambers (LArTPCs). In a LArTPC, charged particles produced (for example, by neutrino interactions) leave a cloud of ionization, which is observed as voltage signals on wires in the detector. The wires are arranged in panels containing hundreds or thousands of closely-spaced wires. Wire panels are placed at different orientations, which allows the signals on the wires to provide two-dimensional information regarding the location of an ionization deposit (called a hit). Because the ionization clouds drift at a constant rate, sampling in time gives a third dimension to the hits, and thus measurement of particle trajectories and identification of the observed particles.

The LArIAT experiment [4] is a testbeam experiment which aims to improve the characterization of the behavior of LArTPCs by measuring the interactions of known particles in liquid argon. The LArIAT detector consists of several devices to identify and precisely measure the momenta of a variety of particle species, and a LArTPC with two wire planes. The LArIAT scientists will measure the response of the LArTPC detector to the testbeam particles, and thus improve our knowledge of this increasingly-important type of detector. In LArIAT, a four-second beam exposure (called a *spill*) is split into possibly many *events*, each of which corresponds to an identified particle (or particles) interacting with the LArTPC. The event is the atomic unit of processing for LArIAT.

3 ANALYSIS WORKFLOW IN LARIAT

The workflow for the LArIAT experiment starts with data collection from the detector. Voltage signals from ionization deposits on the wires are digitized by an analog-to-digital converter (ADC). These are the data read out from the LArTPC. Because the ADC waveform on a wire contains both ionization signal and noise, the first step in data processing is the use of digital signal processing to reduce the noise. The next step is the identification of two-dimensional hits. The 2D hits are then grouped into clusters, which represent an ionization deposition from a single particle. The clusters from multiple views (one for each wire plane) are then combined to from particle trajectories (called *tracks*). Track creation is followed by particle identification and classification of observed particle interactions. In this work, we have concentrated on the first step, which is noise reduction.

LArIAT uses LArSoft, a software toolkit for analysis of liquid argon experiment data, and *art*, an event-processing framework for HEP. *art* provides the modular facilities that allow LArIAT to compose many algorithms into workflows. LArSoft provides the implementations of these algorithms, and the often complex C++ classes that are used to represent the data (such as raw ADC data, hits, and tracks) used as input and output for these algorithms. For example, the class `raw::RawDigit` represents the ADC signal for a single wire. In addition to the ADC count data, it encodes the wire identity, the number of ADC samples for the wire, as well as summary information for the signal on that wire. In order to identify which wire plane a wire belongs to, use of a geometric information software subsystem is required.

Events recorded by LArIAT are independent, thus for all processing steps (not just the first, used in this work) they can be processed in isolation from each other. However, due to the way data is collected and organized into files (one file per spill of the beam), events are processed sequentially. A program is organized to loop over events, and loop over products within an event. This file-based serial workflow does not scale well on an HPC machine.

4 ANALYSIS TECHNIQUE USED IN NOISE REDUCTION

The signals collected by the electronics in the detector include many sources of noise. Noise sources include movement and impurities of the liquid argon, electromagnetic interference from power supplies and equipment within the facility, and vibrations from the environment. Correctly distinguishing between signal and noise is a vitally important task that is both difficult to assess and computationally expensive to apply. Fortunately some of the major sources of coherent noise can be removed by standard digital signal processing techniques [3]. Given that waveforms are composed of samples taken in fixed time internals, LArIAT uses the Fast Fourier Transfer (FFT) to eliminate unwanted noise frequencies. They conducted a detailed noise analysis to build a frequency mask, which is given as configuration data to the algorithms. For this project we emulate the actual algorithm as follows:

```
1. freq_sig = fft(waveform)
2. freq_mags = absolute(freq_sig)
3. thresholds = max(freq_mags) / 50
4. mask = 1 * (freq_mags > thresholds)
5. time_sig = inverse_fft(mask * freq_sig)
```

We use the real-value input FFT algorithm. In (2), the magnitudes of each complex-value bin of the transfer waveform is calculated. In step (3) we generate a mask for frequencies that are 50X smaller than the largest one. Step (4) calculates a mask for all frequencies that are below threshold. Step (5) uses the mask to zero entries below threshold and performs the inverse FFT to reconstruct the waveform in the time domain. Because most of the time involved in this calculation occurs within the FFT, this algorithm is a good representation of the calculations done within the LArIAT application. Our algorithm is slightly more expensive due to the per-waveform calculation of the mask; LArIAT's algorithm uses the pre-recomputed mask that is applied across many events.

5 DESIGN AND IMPLEMENTATION

As noted earlier, all events can be processed independently. This allows us to take maximum advantage of the available data parallelism for a any given number of computing resources. We have designed an MPI-based data-parallel program that processes streams of independent events within different ranks. We have organized the data to provide efficient access to arbitrary ranges of events. This allows us to scale our program to a nearly arbitrary number of MPI ranks. We note that for different workflow steps, a different level of parallelization could make sense. If the LArIAT data sample were much smaller (or the available number of cores were much higher), even this step of the workflow could be parallelized down to the level of the individual wires.

5.1 Organization of data in HDF5

We have chosen to use HDF5 [6] as our data storage solution, partly because of its ubiquity in HPC centers, and partly because implementations have been optimized for these environments. A major reason for choosing HDF5 is because of its integration with MPI-IO and parallel file systems. We use a tabular format for the data that is used in this project, and HDF5 provides flexible enough support to handle this data. A tabular format lends itself to data-parallel programming very easily, and also promotes efficient reading and processing in a variety of programming languages. We use an HDF5 *group* to represent a table, and an HDF5 *dataset* to represent a column (or attribute) within a table. We require all of the datasets in a given group to have the same number of entries to enforce the interpretation of the group as a table; each entry in a dataset thus represents part of a table row, and a given row of a table is specified by a single index, which can be used for each column in that dataset. See [5] for further details on this style of use of HDF5.

We organized the waveform data into a single HDF5 file containing three groups. Each group contains several datasets, and each dataset contains one entry per event. This allows a program to directly access the data for a given range of events by "slicing" each of the datasets using the same range of indices. One group carries a single dataset, containing an event identifier that consists of three `unsigned int` values. Each of the other groups carries the data from one of the two wire planes, as shown in the table 1. Each of these groups contains six datasets; the first one specifies the period of time in which the data were collected, two of these contain metadata that label each wire, two contain calibration data for that wire. For each of these four datasets, the recorded data is an array of length 240. The bulk of the data is the dataset that contains the time-sampled ADC counts on the wires. For each event, and for each wire frame, this is represented in a 240×3072 array of `short int` values. This data organization satisfied a design goal of allowing algorithms that process sets of wires to maximize the use vectorized operations and libraries.

The LArIAT raw dataset consists of approximately 15.7 million events, and the aggregate size of ADC waveform data is about 42 TiB. We have used the HDF5 "deflate" (gzip) compression algorithm on all datasets, retaining the default compression level of 6, resulting in an on-disk file size of approximately 4.2 TiB. About 99.8% of the in-memory data size, and 99.5% of the on-disk data size, corresponds to the waveform data. Note that HDF5 compresses datasets in chunks with a default of 128 dataset entries per chunk. We have not tuned this setting, so the our chunks for our wire plane datasets each contained data from 128 wire planes. We copy the file to the burst buffer, using *striped access mode*, before processing.

Table 1: Description of the data showing the name, type and size in bytes of each of the HDF5 datasets in the group `rawDigits_u` that represents the "u" plane. The HDF5 group corresponding to the "v" plane looks similar.

Name	Type	Size (bytes)
RunPeriod	char	11
adc	short	240×3072
channelID	unsigned short	240
pedestalMean	float	240
pedestalSigma	float	240
wireIndex	unsigned short	240

5.2 Implementation in Python/numpy

The Python package numpy provides the ability to write "vectorized" operations. For numpy, vectorized means that a loop is performed in compiled C code, rather than by the Python interpreter. It does not refer to use of vector instructions on the hardware. In our work, we are using a numpy implementation that is layered atop the Intel MKL library, and which therefore can also be making use of vectorized instructions on Intel hardware. The numpy style of vectorization both simplifies the code (when compared to explicit loops) and also provides high performance, because the loops in C are much faster than the equivalent loops in Python.

To obtain maximum processing efficiency, we want to make as few reads as possible. Memory limitations on the compute nodes determine how many events we can afford to read at a time. In the field of HEP, it is important for our programs to be able to run on both small and large machines; therefore we made the number of events to be read at one time a configuration parameter for the program.

Program execution consists of many iterations, with each iteration reading (and decompressing) the wire-plane data for several events, reshaping the data into an array of wires, performing the noise filtering on each wire, and then reshaping the data back into an array of (processed) wire-planes. In each case the reshaping operation does not copy data: only a few integers representing the shape of the data are changed. Listing 1 shows the relevant part of our code.

```
# first and last are calculated by library
# code, to tell this rank what part of the
# dataset it is to work on.
# adcdataset is the HDF5 dataset carrying
# the ADC data for either the 'u' or 'v'
# wire planes.

# read block of array
adc_data = adcdataset[first:last]
```

```
# convert short int to double
adc_floats = adc_data.astype(float)

# view data as an array of wires, rather
# than as events. Note that this does not
# copy any of the data.
adc_floats.shape =
  (nevts*WIRES_PER_PLANE, SAMPLES_PER_WIRE)

# do the noise reduction
waveforms = transform_wires(adc_floats)

# view the data as events again. This
# does not copy any data.
waveforms.shape(nevts,
                WIRES_PER_PLANE,
                SAMPLES_PER_WIRE)
```

Listing 1: The code executed for each iteration in the program.

To aid in understanding our performance measurements, we placed an MPI barrier near the beginning of the program, before any rank opens the input file. This assures all ranks start their real work at approximately the same time.

6 PERFORMANCE MEASUREMENTS

In order to understand the scaling, as well as the raw performance, of our code, we have instrumented the program to report the absolute time at which each rank encounters a given event: the beginning of an iteration, the start and end of reading the data for that iteration, and the end of the iteration. We also capture a timestamp before and after the MPI barrier noted above. We capture all timestamps using the MPI function `MPI_Wtime`. In the implementation we used, this function was not synchronized between nodes. However, for a typical experiment, the full spread of timestamp values taken after the start barrier is less than 50 milliseconds. For ranks on the same node (and thus sharing the same clock) the typical maximum spread is about 20 microseconds, and the largest is about 250 microseconds. Figure 1 shows the distribution of timestamps collected after the start barrier for each rank of a typical 1200-node program run. To evaluate the spread of timestamp values within each node, we first calculate the mean timestamp value of each node. We define the deviation of the timestamp for each rank as the difference between the timestamp for the rank and the mean of the timestamps for all ranks on the same node. The distribution of this deviation is shown in figure 2. In that figure, a single value at -230 microseconds has been suppressed, to more clearly show the distribution of the remaining values.

6.1 Experimental Setup

We ran multiple experiments at each number of nodes, as shown in table 2. The number of runs we could perform was limited by the size of our allocation; we performed most runs at the smallest and largest number of nodes in order to best evaluate the linearity of the scaling of our program. All program runs were done on KNL nodes of Cori at NERSC. Each KNL node has 68 cores, 96 GiB DDR4 2400 MHz memory, and 16 GiB of on-package, high-bandwidth multi-channel DRAM (MCDRAM). We have used only *cache mode*,

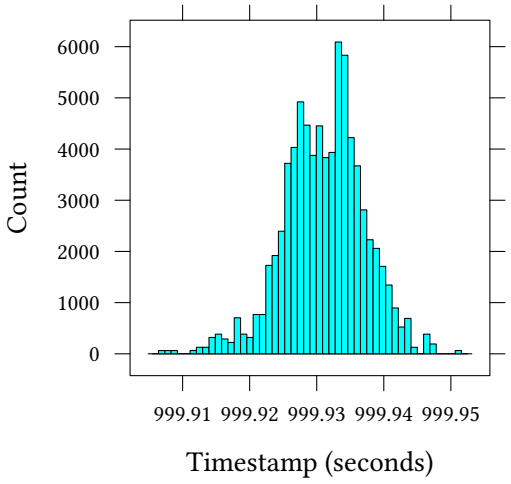

Figure 1: Distribution for timestamp values, collected after the start barrier, for each rank of one 1200-node program run.

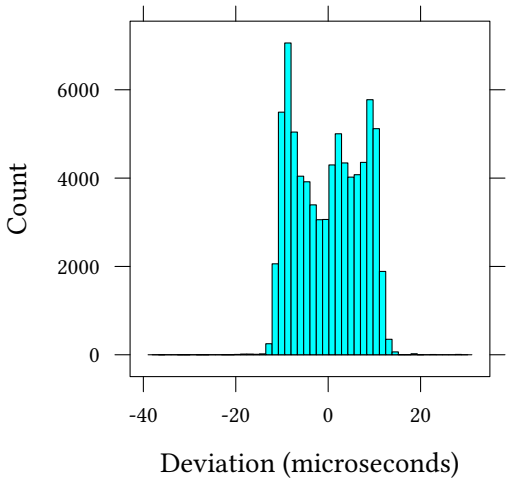

Figure 2: Distribution of the deviation from the mean for its node of the timestamps for each rank of one 1200-node program run. A single value of -230 microseconds has been suppressed from this histogram, to better show the remainder of the distribution.

which effectively uses the MCDRAM as an L3 cache. In each experiment, we used 64 MPI ranks per node. We use the term *program size* to refer to the number of nodes, and thus number of MPI ranks, used in the program run. In each experiment, we process the full 42 TiB data sample. The memory capacity of Cori KNL nodes limits us to reading 31 wire planes at a time for each rank.

Table 2: Configurations and number of runs of the various experiments, all run on KNL nodes of Cori. Each experiment used 64 ranks per node.

Program size (nodes)	Burst buffer allocation size (TiB)	Number of runs
200	5	6
200	10	1
600	5	3
600	10	1
1200	5	6
1200	10	1

6.2 Results

6.2.1 Overall program speed. Our first analysis was of the overall program speed, which is the feature most important to HEP scientists. Initially, we used only a 5 TiB burst buffer allocation in the experiments. To measure this speed for each experiment, we took the difference between the absolute timestamps for the last end-of-iteration event and the first begin-of-iteration event. While the clocks giving these timestamps were not synchronized across nodes, their spread (as discussed above) was negligible compared to the running time of the program. Figure 3 shows the processing speed, which is the number of events processed by the program divided by the running time as defined above, as a function of the program size. The line in the figure is a linear fit to the data. The reproducibility of the results is excellent, as is seen by the small spread in speed for the experiments at each program size. The scaling is nearly perfect. The 1200-nodes runs were, on average, 5.97 times faster than the 200 node runs.

6.2.2 Effect of burst buffer allocation size on overall program speed. It was suggested by the performance experts at NERSC that a larger allocation of burst buffer space might make processing faster. To investigate this, we repeated one experiment for each program size using a 10 TiB burst buffer allocation. In order to see what effect this had, as a function of the program size, we calculated a *normalized* processing speed by dividing each absolute processing speed by the mean of the processing speeds for that program size. As shown in figure 4, this made no significant difference. However, since our program spends about 10% of its time reading and decompressing data, for all program sizes, we also looked more carefully at that portion of the work.

6.2.3 Reading and decompression speed. Because we have recorded a timestamp before and after each array slice reading, we can calculate, for each iteration, and for each rank, the time taken to read and decompress the ADC data. However, we can not distinguish between the time taken to read the data from the file and the time taken to decompress that data. We calculate the *read+decompress speed* for the program as the number of bytes (after decompression) being read and decompressed by each rank divided by the median of the calculated reading times. We use the median because the last iteration of each program run reads fewer events than all others. Figure 5 shows how this speed varies with the program size, separately for the 5 and 10 TiB burst buffer allocations. At each program size, the 10 TiB allocation gives the best result.

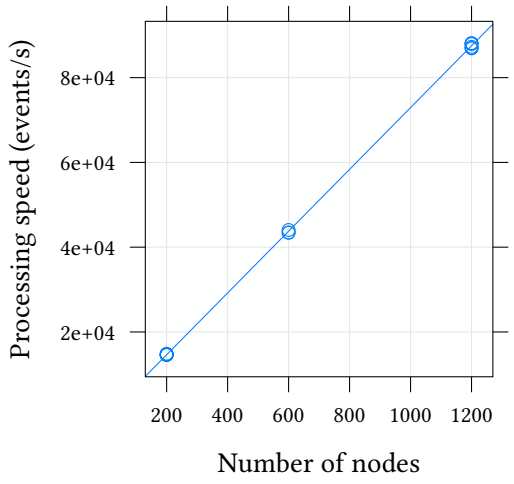

Figure 3: Scaling of speed of the processing of 42 TiB of data, for different program sizes. In all cases we used 64 MPI ranks per node, and a burst buffer allocation of 5 TiB. The line is a linear fit to the multiple measurements at each number of nodes, showing the nearly-perfect strong scaling.

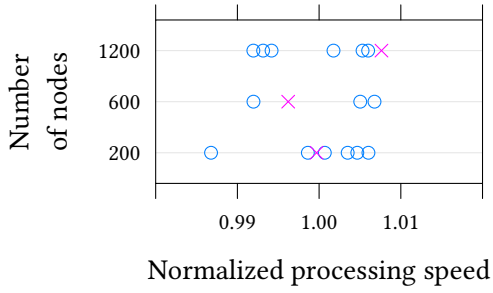

Figure 4: Variation of processing speed for different burst buffer allocation sizes, for each number of nodes. The blue circles correspond to experiments using a burst buffer allocation of 5 TiB, and the magenta crosses to an allocation of 10 TiB.

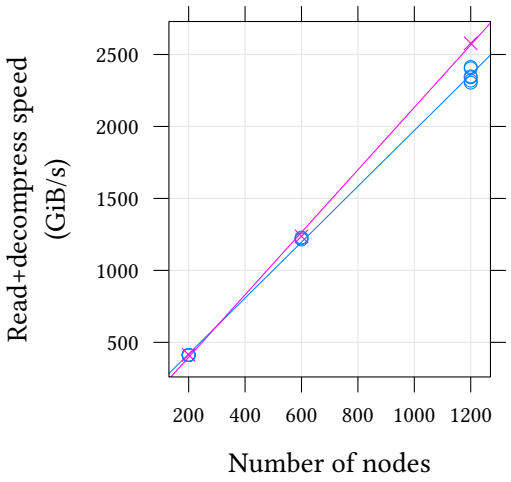

Figure 5: Scaling of reading+decompression speed of the processing of 42 TiB of data, for different program sizes. In all cases, 64 MPI ranks per node are employed. The line is a linear fit to the multiple measurements at each number of nodes, separately for each burst buffer allocation size. The blue circles correspond to experiments using a burst buffer allocation of 5 TiB, and the magenta crosses to an allocation of 10 TiB.

The nearly linear increase of reading+decompression speed of the program as we increase the program size hides, to some extent, the effect of varying the burst buffer allocation. To observe its effect more clearly, we can normalize the speed by dividing the speed for each experiment by the mean of all the experiments with the same program size. This result is shown in figure 6. The size of the burst buffer allocation seems to have some influenece: at each program size, the larger burst buffer allocation yielded the best performance. This best speed was better by 5.5, 2.0 and 5.0 standard deviations, for the 200, 600 and 1200 node experiments, respectively. A larger sample size would be necessary for an authoritative statement.

6.2.4 In-memory computation speed. The noise reduction step of the processing is entirely in-memory. We characterize this processing speed by calculating, for each experiment, the number of events processed by each rank for each iteration times the number of ranks used in the experiment, divided by the median time for the noise reduction step for each iteration. Again, the median is used because the final iteration processes fewer events than every other iteration. Figure 7 shows the in-memory processing speed as a function of the program size. We see nearly perfect scaling, and (as expected) that the size of the burst buffer allocation makes no difference to the speed of this in-memory calculation.

7 CONCLUSION

The work done in our program is only the first step of the full analysis workflow for an experiment like LArIAT. Our program

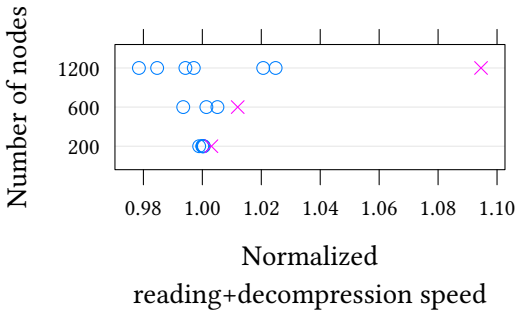

Figure 6: The effect of burst buffer allocation size on the normalized reading+decompression speed, for differing program sizes. The blue circles correspond to experiments using a burst buffer allocation of 5 TiB, and the magenta crosses to an allocation of 10 TiB.

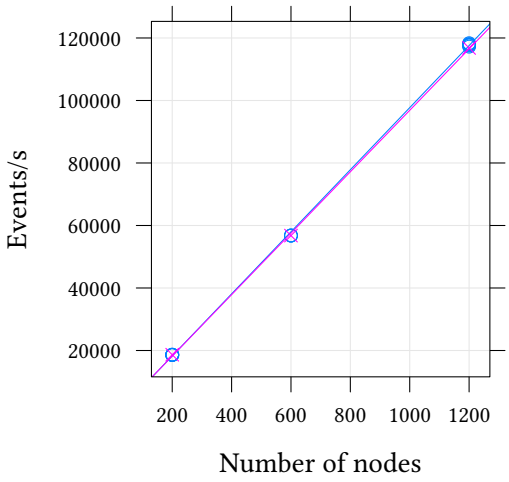

Figure 7: Scaling of noise reduction step speed, for different program sizes. In all cases, 64 MPI ranks per node are employed. The line is a linear fit to the multiple measurements at each program size, separately for each burst buffer allocation size. The blue circles correspond to experiments using a burst buffer allocation of 5 TiB, and the magenta crosses to an allocation of 10 TiB.

spent about 10% of its time reading data; a more complete program would do much more work on in-memory data, and would thus be even less sensitive to reading speed. We note that for a more substantial program, it may be that many different data types of many different sizes might be read; the IO for each might benefit from independent tuning. Nonetheless, even without tuning the chunking within the data file to match the amount of data read in

a single iteration, we find the input speed adequate and its scaling excellent.

We were surprised by the amount of memory used by the program, in comparison to the size of the raw data being read. Rough measurements of the program memory usage indicated that the program's working set size was approximately twenty times larger than the raw data size. While numpy copies data only when necessary, it is still not easy to identify all the places in which memory allocation is required.

Traditional HEP analysis programs are typically modular in design, with the main aspect of modularity being the runtime-plugin algorithms which are organized into a sequential workflow to be applied to each event. Events may be processed processes concurrently, using a task-based shared-memory parallel system. In this work, we use a different modularity. A given phase of processing (the noise reduction on a wire) was identified and the function written to implement that algorithm was applied to all the wires that could be read into a given program rank at one time. Different processing tasks in a more complete program could use different aggregations. For example, hit finding on a given wire might consider also the signal on physically adjacent wires, but within the same event, effectively yielding a stencil-type algorithm.

In a complete analysis program, we would have to save the results of at least some of the algorithms. We have not yet investigated the scalability of MPI-IO *output* for such a program. Because the sizes of the output of many algorithms is not known in advance—for example, each event may have a different number of hits and a different number of tracks—designing the program for good scalability in writing will be more challenging.

This work serves as a example to be followed in the design of solutions for future problems that use similar analysis techniques. Using new HPC hardware and non-traditional (for HEP) programming infrastructure, we reorganized the data and the corresponding code to efficiently utilize the resources. Using high-performance "vectorized" operations in numpy, with an easy-to-use Python interface familiar to our HEP community, we have shown that parallel processing efficiency can be achieved with readable and maintainable code. We demonstrate nearly-perfect scaling of processing speed of the program as a whole, and also for the reading (and decompressing) and computation time.

8 FUTURE WORK

There are several performance- and scaling-related tasks that we would like to carry out in order to better understand the behavior of our Python code on Cori. The traditional HEP implementation of this algorithm, which is designed to run on commodity "grid" nodes, is not straight-forward to compare with our implementation. Our plan is to run the traditional HEP code on one KNL node and to compare how many events per second such a node is capable of processing. Assuming that such a program can scale to a large number of nodes, we can then compare the speed of each approach. The ability to write parallel program using familiar and easy-to-use Python interface, will make our approach attractive to our community, if we find that the speed of the Python program is competitive with that of the traditional HEP solution.

We would like to understand whether our Python code fully benefits from the underlying machine architecture. We have just begun the process of profiling our code; this has turned out to be challenging, for a Python program running in a Shifter container. We will also implement the same use case in a compiled language (C++) and compare the performance of the two approaches. We will evaluate whether the design and implementation convenience from Python outweighs any performance hit.

We would like to extend our work to larger program sizes. However our first attempts to use 1800 nodes result in a failure in the initialization of mpi4py. We have not yet been able to investigate this failure. We would also like to understand why larger program sizes benefit more from the larger burst buffer allocation, and whether a still larger allocation would further improve performance.

Finally, we would like to extend this approach to different HEP problems, which may include selective reads, and selection or filtering type operations on the data. We are working on a problem for NOvA experiment to perform parallel event filtering as part of the SciDAC-4 project "HEP Data Analytics on HPC". In that project, the data are organized in an HDF5 file using a similar tabular design as described in this work. That design enables the application to support vectorized selection and filtering criteria.

9 ACKNOWLEDGMENTS

We would like to thank the members of the LArIAT collaboration for allowing us to use their data in this work.

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 material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program. 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. This work was supported in part by the Fermi National Accelerator Laboratory LDRD program, grant LDRD-2016-010.

REFERENCES

[1] B. Abi et al. The DUNE Far Detector Interim Design Report Volume 1: Physics, Technology and Strategies. 2018.

[2] M. A. Acero, P. Adamson, L. Aliaga, T. Alion, V. Allakhverdian, N. Anfimov, A. Antoshkin, E. Arrieta-Diaz, A. Aurisano, A. Back, C. Backhouse, M. Baird, N. Balashov, B. A. Bambah, K. Bays, B. Behera, S. Bending, R. Bernstein, V. Bhatnagar, B. Bhuyan, J. Bian, T. Blackburn, J. Blair, A. Bolshakova, P. Bour, C. Bromberg, J. Brown, N. Buchanan, A. Butkevich, V. Bychkov, M. Campbell, T. J. Carroll, E. Catano-Mur, A. Cedeno, S. Childress, B. C. Choudhary, B. Chowdhury, T. E. Coan, M. Colo, J. Cooper, L. Corwin, L. Cremonesi, D. Cronin-Hennessy, G. S. Davies, J. P. Davies, S. De Rijck, P. F. Derwent, R. Dharmapalan, P. Ding, Z. Djurcic, E. C. Dukes, P. Dung, H. Duyang, S. Edayath, R. Ehrlich, G. J. Feldman, M. J. Frank, H. R. Gallagher, R. Gandrajula, F. Gao, S. Germani, A. Giri, R. A. Gomes, M. C. Goodman, V. Grichine, M. Groh, R. Group, D. Grover, B. Guo, A. Habig, F. Hakl, J. Hartnell, R. Hatcher, A. Hatzikoutelis, K. Heller, A. Himmel, A. Holin, B. Howard, J. Huang, J. Hylen, F. Jediny, M. Judah, I. Kakorin, D. Kalra, D. M. Kaplan, R. Keloth, O. Klimov, L. W. Koerner, L. Kolupaeva, S. Kotelnikov, I. Kourbanis, A. Kreymer, Ch. Kulenberg, A. Kumar, C. Kuruppu, V. Kus, T. Lackey, K. Lang, S. Lin, M. Lokajicek, J. Lozier, S. Luchuk, K. Maan, S. Magill, W. A. Mann, M. L. Marshak, V. Matveev, D. P. Méndez, M. D. Messier, H. Meyer, T. Miao, W. H. Miller, S. R. Mishra, A. Mislivec, R. Mohanta, A. Moren, L. Mualem, M. Muether, S. Mufson, R. Murphy, J. Musser, D. Naples, N. Nayak, J. K. Nelson, R. Nichol, E. Niner,

A. Norman, T. Nosek, Y. Oksuzian, A. Olshevskiy, T. Olson, J. Paley, R. B. Patterson, G. Pawloski, D. Pershey, O. Petrova, R. Petti, S. Phan-Budd, R. K. Plunkett, B. Potukuchi, C. Principato, F. Psihas, A. Radovic, R. A. Rameika, B. Rebel, P. Rojas, V. Ryabov, K. Sachdev, O. Samoylov, M. C. Sanchez, J. Sepulveda-Quiroz, P. Shanahan, A. Sheshukov, P. Singh, V. Singh, E. Smith, J. Smolik, P. Snopok, N. Solomey, E. Song, A. Sousa, K. Soustruznik, M. Strait, L. Suter, R. L. Talaga, P. Tas, R. B. Thayyullathil, J. Thomas, E. Tiras, S. C. Tognini, D. Torbunov, J. Tripathi, A. Tsaris, Y. Torun, J. Urheim, P. Vahle, J. Vasel, L. Vinton, P. Vokac, A. Vold, T. Vrba, B. Wang, T. K. Warburton, M. Wetstein, D. Whittington, S. G. Wojcicki, J. Wolcott, S. Yang, S. Yu, J. Zalesak, B. Zamorano, and R. Zwaska. New constraints on oscillation parameters from ν_e appearance and ν_μ disappearance in the nova experiment. *Phys. Rev. D*, 98:032012, Aug 2018.

[3] Bruce Baller. Liquid argon TPC signal formation, signal processing and reconstruction techniques. *JINST*, 12(07):P07010, 2017.

[4] F. Cavanna, M. Kordosky, J. Raaf, and B. Rebel. LArIAT: Liquid Argon In A Testbeam. 2014.

[5] S. Sehrish, J. Kowalkowski, M. Paterno, and C. Green. Python and HPC for High Energy Physics Data Analyses. In *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing*, PyHPC'17, pages 8:1–8:8, New York, NY, USA, 2017. ACM.

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