HEPnOS: a Specialized Data Service for High Energy Physics Analysis

Sajid Ali[‡], Steven Calvez[†], Philip Carns[*], Matthieu Dorier[*], Pengfei Ding[‡], James Kowalkowski[‡], Robert Latham[*], Andrew Norman[‡], Marc Paterno[‡], Robert Ross[*], Saba Sehrish[‡], Shane Snyder[*], and Jerome Soumagne[§]

[*]*Argonne National Laboratory*, Lemont, IL, USA – {mdorier,carns,robl,rross,ssnyder}@mcs.anl.gov

[†]*Colorado State University*, Fort Collins, CO, USA – steven.calvez@colostate.edu

[‡]*Fermi National Laboratory*, Batavia, IL, USA – {sasyed,dingpf,anorman,jbk,paterno,ssehrish}@fnal.gov

[§]*Intel Corporation*, Santa Clara, CA, USA – jerome.soumagne@intel.com

***Abstract*—In this paper, we present HEPnOS, a distributed data service for managing data produced by high-energy physics (HEP) experiments. Using HEPnOS, HEP applications can use HPC resources more efficiently than traditional file-based applications. The file-based model leads to a rigid, chunk-based allocation of computational resources and limits the number of cores that can be used concurrently by an HEP application. The fundamental problem is that organizing domain-specific data into files inadvertently introduces a single, artificial, conflated tuning parameter that puts key optimization goals into conflict: larger file sizes reduce metadata overhead and thus improve I/O efficiency, but smaller file sizes provide more opportunity for workflow parallelism and load balancing. In this work, we introduce a domain-specific data service that decouples that constraint so that data can be accessed and processed in its natural granularity while still maintaining I/O efficiency. By removing the constraints introduced by file handling we are able to obtain better scaling and make efficient use of more cores for processing a fixed-sized data sample. We demonstrate the improved scalability by using an application developed in the file-based paradigm and comparing it to a version modified to use HEPnOS.**

***Index Terms*—HPC, Storage, Mochi**

I. Introduction

The design of most High Energy Physics (HEP) applications and workflows is influenced by the grid-based high-throughput computing and storage facilities that have traditionally been used in the field. The typical HEP workflow needed to complete a data processing campaign is broken into several distinct steps, each performed by the invocation of a different application.[1]

An HEP workflow running on grid compute nodes uses files both as the storage technique and to exchange data between successive processing steps. Because the file written as output by step n is read as input by step $n+1$, it is common for a step to "copy forward" data from its input file to its output file if those data are needed by later steps. This is the case even when those data are needed only by a possibly much later step and not by the step doing the copying forward. Important workflows typically generate thousands to hundreds of thousands of files. These files can range in size from a few megabytes to tens of gigabytes. The file size is chosen based on constraints imposed by the grid processing systems, including the maximum processing time allowable on grid nodes and the size requirements for archival storage, and the size and organization of the experimental data stream originating from the scientific detector systems. File handling features present within experiments' data acquisition systems, often impose stringent constraints on file size due to limited file buffering and near real-time operational requirements. These sizes and organizations often percolate through to the higher levels of grid processing even though the original real-time and hardware constraints are nolonger present.

In this manner, the file is the atomic unit of processing for the grid-oriented systems. This unit of dicretization is however in a sense artificial. It is not a reflection of any unit of processing inherent in the scientific (physics) content of the experiment data[2]. The natural atomic unit of data for the representation of subatomic interactions with nuclear fields in the experiments is denoted as the *event*. An event represents a single readout of a full detector covering a window of time that is of interest, typically identified by the experimental apparatus as a potential subatomic interaction. Events are discrete and atomic in the sense that each event is assumed to be causally disconnected from each other event and representative of an independent trial of the measurement or hypothesis. Under this formal assumption, each event can be processed independently and in any order with respect to each other event without inducing measurement bias. A traditional file can contain any number of events, from a few tens of events to tens of thousands of events, but typically contains events that were acquired over a macroscopic time scale of a few minutes to hours of experimental detector operations.

Large analysis tasks are run as batch jobs on distributed grid resources. Batch jobs typically consist of tens to thousands of concurrent processes, distributed over the grid resources. In the traditional design, each process works on a series of files; no two processes work on the same file in order to minimize redundancy in IO transfers between the storage system and the compute elements. To support parallelism between jobs, the files are delivered by a data handling system that allows the

[1]Often these applications are configurations of a common framework.

[2]Information regarding physical calibrations of the instruments is associated with the file structure but does not drive its organization

work to be pipelined. When a process is finished processing one file it requests the next file in the set of files to be processed by the job. Because the amount of work required to process each file is not the same, this pipelining allows more efficient resource usage than would be obtained by dividing the files equally between processes. Sometimes the processes in a batch job run several steps in the analysis chain. If there are n steps in the series, this produces n or more files. There is a limit to how many steps can be run in such a series, because of the grid resource constraints both on the allowable duration of the processes and disk space available on grid nodes. The complexity of each step can also be limited by the amount of memory available on the nodes. For a workflow with many steps which can not fit within the constraints of the grid nodes, a succession of batch jobs will be run serially. In this succession of jobs, each one will write out its results to files which are copied back to the storage system. The next stage in the workflow's chain is then run through the batch system and will retrieve and ingesting that previous stage's output, writing out in turn its results.

While this workflow model has been successful for grid-based processing for decades, in the move to HPC processing it is less suitable. We note here several issues. The need to "copy forward" data for use in later steps introduces superfluous IO traffic that could be avoided if later steps had access directly to the data from earlier processing steps.

The multiplicity of files introduced by the multi-step workflow creates significant file-handling and metadata-tracking work that would not be necessary if the data were in a single store. There is also the additional buffering space needed in the file system for transient data that is needed by downstream workflow stages. The serial execution of multiple batch jobs introduces inefficiency in load balancing, resource management, and overall latency till the final results are available. The artificially coarse granularity resulting from using the file as the atomic unit for work (rather than the event) results in a large scale idling of resources near the end of each stage, when the uneven distribution of work causes a few processes to be busy finishing their final round of files, while all other nodes in the batch ensemble are idled due to lack of files to process. Wide variation in the size of files, or the number or aggregate complexity of events in the files, exacerbates this imbalance.

To avoid these issues we have developed a distributed data service: the High Energy Physics new Object Store, HEPnOS. Using it, multiple processes can share a dataset with event-level, rather than file-level, granularity. A unique feature of the HEPnOS interface is that it follows concepts in High Energy Physics event processing. The current HEPnOS interface is a demonstration of this capability. The existing grid-based systems make it difficult to use more resources for steps of the analysis that are more computationally expensive. HEPnOS allows us to use more processes for the slower phases of the work without entailing the complications of file handling between the phases, and without arbitrary limits on the number of processes usable based on the number of files available at each stage.

A common scenario in many HEP analyses is the iterative refinement or tuning of the analysis process, based on the data available. This requires multiple passes through a given dataset. Having the data available in a distributed data service not only makes this more convenient, but also spreads the cost of loading the data over all iterations. On an HPC system, the distributed service can leverage the system's high-speed networking to deliver data faster than file-based IO can. If the service is able to keep the dataset in memory even better performance is possible.

In this work, we demonstrate the reading speed and scalability (both weak and strong) of HEPnOS. To do so we have chosen a use case from one HEP experiment and a limited dataset to which we were granted access. In section II we describe HEPnOS. We describe the application in section III. In section IV we report on our performance measurements. We compare the speed and scaling behavior of the existing grid-based application to one that uses HEPnOS to do the same work. Section V discusses related work. We draw some conclusions in section VI.

II. HEPnOS

This section presents the design of HEPnOS, its interface, and how it internally organizes data. HEPnOS was designed using the Mochi methodology [1], [2], which consists of relying on reusable, composable building blocks to develop storage systems that are highly tailored to their applications. Designing HEPnOS required discussing with HEP scientists to understand how their applications would interact with it, and what guarantees of performance, fault tolerance, and consistency it should provide.

A. Requirements and interface

The targeted HEP workflows were written in C++ and, they manipulate their data in the form of C++ objects, and persistent data is stored on files. The first set of requirements for HEPnOS was therefore (1) to be able to store and load C++ objects directly rather than going through files, and (2) to provide a way to ingest existing data from files.

The second set of requirements came from the way scientists organize HEP data into a hierarchy of *datasets*, *runs*, *subruns*, and *events*. Datasets are named containers (similar to folders in traditional file systems). They can contain other datasets as well as runs. Runs, subruns, and events are identified by numbers. Runs contain subruns, and subruns contain events. Finally runs, subruns, and events can contain zero or more *products* (C++ objects), each identified by a *label* and by a *type*.

The third set of requirements for HEPnOS came from the way HEP applications will interact with it. While traditional HEP applications do not use MPI, the applications that interact with HEPnOS will typically be embarrassingly-parallel MPI programs loading products from HEPnOS in an iterative manner, performing some analysis, and writing new products back into HEPnOS. Typical datasets will contain several millions of relatively small products (from tens of bytes to a few

```
#include <hepnos.hpp>

// example structure
struct Particle {
  float x, y, z;  // data members
  // serialization function for boost to use
  template <typename A>
  void serialize(A& a, unsigned long /* version */)
  {
    ar & x & y & z;
  }
};
// initialize a handle to the HEPnOS datastore
auto datastore =
    hepnos::DataStore::connect("config.json");
// access a nested dataset
hepnos::DataSet ds = datastore["path/to/dataset"];
// access run 43 in the dataset
hepnos::Run run = ds[43];
// create subrun 56 within this run
hepnos::SubRun subrun = run.createSubRun(56);
// create event 25 within this subrun
hepnos::Event ev = subrun.createEvent(25);
// store data (an std::vector of Particle)
st::vector<Particle> vp1 = ...;
ev.store(vp1);
// load data
std::vector<Particle> vp2;
sv.load(vp2);
// iterate over the subruns in a run
for(auto& subrun : run) {
  std::cout << subrun.number() << std::endl;
}
```

Listing 1. Example of client code using HEPnOS

megabytes). To perform well, HEP programs need a distributed storage system relying either on compute node memory or on node-local storage such as SSDs, if more persistence is needed.

All these requirements led us to design what amounts to a distributed key-value store with a C++ interface capable of serializing and deserializing C++ objects, and with a data organization in *datasets*, *runs*, *subruns*, and *events*.

From the scientist's point of view, HEPnOS is used as exemplified in Listing 1. This listing shows that navigating the HEPnOS hierarchy is very similar to accessing a C++ container such as an `std::map`. HEPnOS uses template metaprogramming in conjunction with Boost serialization to handle any C++ object that provides a `serialize` function (as well as any native datatype and C++ standard library container).

B. Design overview

HEPnOS is based on components of the Mochi projects. These components rely on the Mercury RPC library for communication [3], on Argobots [4] for threading and tasking, and on Margo to combine Argobots and Mercury into a simpler programming model.

HEPnOS's architecture, shown in Figure 1, is primarily based on the Yokan component.[3] Yokan is a remotely-accessible, single-node key-value storage component. It provides a number of persistent backends such as RocksDB, BerkeleyDB, LevelDB, *etc.*, as well as in-memory ones (based on C++ standard library containers such as `std::map`). The storage side of HEPnOS is a collection of compute nodes running individual Yokan instances (or "providers"[4]) backed up by local memory or local storage. Through Mercury, Yokan provides access to key-value pairs through RPC (for single small objects) and RDMA (for large objects or batches of multiple objects). Yokan also provides functions to iterate over the stored key-value pairs. The next section explains how these functions are used to provide the required data organization for HEPnOS.

The Bedrock component is used for bootstrapping and configuration. It takes a JSON configuration describing the service and spins up the components according to this configuration. This description notably contains Argobots information (*e.g.*, the number of execution streams and their schedulers and pools), Mercury configuration (*e.g.*, the type of network to use, the location of the progress loop thread, in which execution streams RPCs will be handled), and the list of other providers (*e.g.*, Yokan providers in our case) with their respective configuration (list of database instances) and mapping of providers to Argobots resources. This high degree of configurability is what allowed us to fine tune HEPnOS throughout its development and find configurations that work best for our use-cases, either via performance analysis and monitoring [5] or ML-based autotuning [6].

C. Data organization

Yokan stores its data in a flat key-value pairs namespace. HEPnOS requires data to be organized in a hierarchical manner. We enable this organization by carefully crafting the keys used to access containers (datasets, runs, subruns, and events) and data products, and by placing data on servers in such a way that we can iterate over them in a coherent manner.

1) Container keys: Since datasets can contain other datasets, a dataset can be identified by a string representing a full path, for instance `/fermilab/nova` for the "nova" dataset inside the "fermilab" dataset. These full paths are mapped to a UUID in a separate database. Runs are represented by a 64-bits number within their dataset. Hence they are uniquely identified by a key of the form `<dataset UUID><run number>` where the run number is converted to big-endian. The same strategy is applied for subruns and events. HEPnOS consists of a number of database instances that store either datasets, runs, subruns, or events. The number of databases for each type of container is independently configurable. In the key-value store, container keys do not have an associated value. The presence or absence of a key is enough to indicate whether the corresponding container exists.

[3]Our paper presenting the Mochi methodology [1] lists the Bake component, which has ultimately been removed from the design, and the SDSKV component, an older key-value storage component which was latter replaced with Yokan.

[4]The term "provider" is used in Mochi to represent an object capable of responding to a predefined set of RPCs to provide a specific functionality, such as storing key-value pairs in a database. A provider may manage multiple resources (*e.g.*, multiple databases), and be mapped to a specific Argobots pool in which RPCs are pushed. Effectively, providers are the mechanism by which the Argobots resources used to execute an RPC (*e.g.*, a CPU), are decoupled from the resources the RPC is acting on (*e.g.*, a database).

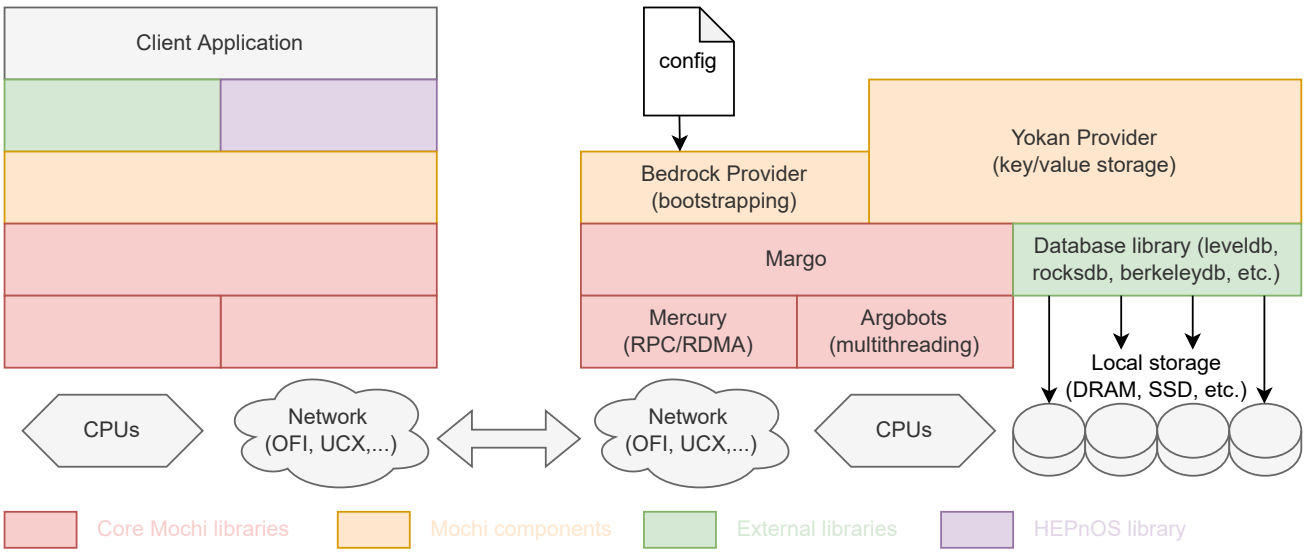

Fig. 1. Architecture of HEPnOS. HEPnOS is built upon the Mochi suite of building blocks including Margo, which provides Argobots-aware wrappers for Mercury RPC routines and Yokan, which is a key-value store that can use a variety of backends.

2) Product keys and values: To uniquely identify a product, its key is built by concatenating the key of its container with the label and the type of the data product, separated by a `#`. For example `<DatasetAUUID><0001><0001><0004>mylabel#Particle` is the key (shortened for space constraints) to a Particle object with label *mylabel* in DataSetA, run 1, subrun 1, event 4. The value associated with such a key is a serialized version of the C++ object.

3) Placement and iteration: One challenge in our design was to place container keys in such a way that we could iterate easily through datasets, runs, subruns, and events. By relying on consistent hashing of the full key for placement, listing the elements of a container would have required interrogating all the servers and merge their results. Instead, HEPnOS carefully places the keys on servers so that iterating over the elements of a container only involves using the iterator functionalities of one database. For this, we select the location of a particular container key by hashing its parent's key. For instance the location for the subrun key `</fermilab/nova/><33><42>` is selected through consistent hashing of the key `</fermilab/nova/><33>`. Because keys are sorted lexicographically inside a database and numbers are converted to big-endian, this strategy ensures that (1) all subcontainers directly inside a given container are located *in the same database instance*, and (2) they are *sorted alphabetically* for datasets and *in ascending order* for runs, subruns, and events.

HEPnOS does not enable iterating over products because that is not a relevant usage pattern for applications. The location of a product is determined by the consistent hashing of its parent container key. We chose this method as it allows reading products in batches when accessing multiple products from the same container.

Such a placement strategy is in line with the way HEP workflows operate: individual tasks of the workflow will generally process events in distinct subruns, hence distributing the load when iterating over events. The way products are distributed also ensures some load balancing both in terms of the amount of data stored, and in accesses.

D. Batching, asynchronous accesses, and load balancing

To improve performance when accessing many small data items, HEPnOS provides batching and asynchronous access capabilities. A `WriteBatch` object can be passed to functions that create containers or store objects. This object will accumulate the updates in a local buffer, group them by target database (since not all updates target the same database) and send batch updates upon destruction.

An `AsynchronousWriteBatch` object can be used to issue RPCs in the background and ensure that all the updates are completed when its destructor is called.

For reading, a `ParallelEventProcessor` object provides a high-level interface for a group of processes to iterate over the events in a given dataset in parallel and in a load-balanced manner. This interface takes a callable object (which can be provided by a lambda expression) that will be invoked on each event. The interface handles loading and distributing events across participants. It does so by designating a subset of processes as readers (typically as many readers as databases to read from). Readers load batches of events from HEPnOS in the background and place them in a distributed queue from which all processes pull. The `ParallelEventProcessor` object also takes care of prefetching products associated with an event if requested by the program.

III. Our Example Use Case

A. The NOvA Experiment

The NuMI Off-axis Electron Neutrino Appearance Experiment (NOvA), is the current running flagship of the United State's program in long baseline neutrino physics [7]. The NOvA experiment is hosted by Fermi National Accelerator Laboratory and is designed to measure the fundamental properties of the subatomic particles known as neutrinos. The experiment measures the worlds most intense beams of neutrinos and anti-neutrinos, which are produced by the Fermilab main injector beam facility [8] in Batavia, IL. This neutrino beam is measured by two separate particle physics detectors, one placed 103 m underground at a position 1 km from the beam source, and the second placed on the Earth's surface, 810 km from the beam source at a dedicated laboratory facility in Ash River, MN. The NOvA experiment measures the manner in which the neutrinos in the beam quantum mechanically change as they propagate along the 810 km flight path through the Earth's crust.

In particular, NOvA measures the rates at which muon type neutrinos and anti-neutrinos transform through the Pontecovo-Maki-Nakagawa-Sakata (PMNS) oscillation process into electron type neutrinos and anti-neutrinos [9]. To make these measurements, the data from the NOvA detectors are searched for topologically distinct patterns that are indicative of a neutrino interacting on a carbon or hydrogen nucleus. These interaction events, once selected, are then analyzed based on their measurable physical quantities such as electromagnetic shower energy depositions, particle trajectories, nuclear breakup activity, and other observable properties. The analysis of these quantities is used to determine the quantum mechanical identity and energy of the incident neutrinos in each interaction. These ensembles of interactions are then in turn to make measurements of the oscillation probability $P(\nu_\mu \rightarrow \nu_e)$ and survival probability $P(\nu_\mu \rightarrow \nu_\mu)$ for both neutrinos and anti-neutrinos.

The difficulties inherent in performing this selection process arise due to both the size and the complexity of the data. The NOvA far detector's real-time data acquisition systems have produced approximately 1.94 PB of raw data, spread over 16.8 million individual files, representing a collection of over 22.6 billion candidate interaction events that have been collected since its start of physics operations in 2013. Each of these events is split based on its spatial and temporal characteristics into regions of interest, colloquially referred to as "slices," which represent the candidate neutrino interactions. The candidate event data is processed and distilled into a collection of approximately 600 physics quantities which are derived from algorithms that attempt to reconstruct the individual particles, their characteristics, and trajectories. The derived physics quantities are hierarchically organized within any given slice based on the reconstructable subatomic particle content of the interaction. Since neutrino-nucleus interactions can proceed through many different nuclear interaction channels, the final state topologies of these interactions can differ greatly from interaction to interaction.

Historically these data have been analyzed and searched for the interactions of interest through a sequential scan of each event record in a file. The scans as originally implemented could achieve parallelism down to the level of an individual file. However, each data file in the NOvA beam dataset contains on average 9k-12k candidate slices that need to be examined. Data coming from the cosmic ray samples, which are recorded at a rate 12 times higher than the beam data, contain on average 108k-144k candidates in each file. This results in an extra 5 orders of magnitude that could be exploited if the artificial file-based organization was removed from the data analysis chain. Moreover, the neutrino selection algorithms for the long baseline analysis measurements perform a down-selection of the dataset with a rejection ratio of $\mathcal{O}(10^9)$. This sequential analysis approach has been used for the NOvA measurements of electron neutrino (ν_e) appearance and electron anti-neutrino ($\bar{\nu}_e$) appearance, as well as for establishing new constraints on neutrino mixing [10]–[12].

The NOvA experiment has performed measurements of electron neutrino and anti-neutrino appearance and muon neutrino and anti-neutrino disappearance. The most recent results from the NOvA collaboration [10] were a combined measurement of all four interaction modes which were combined to provide the world's leading measurement of neutrino oscillations and the first evidence for anti-electron neutrino appearance. These results were produced using data selection that was run on a collection of 172,029 files representing five years of data taken between 2013 and April 2018.

B. Our Example Application

We have constructed a data parallel selection application using the HEPnOS data service as a mechanism to eliminate file transfers between computing processes and allow for parallelism to be exploited down to the level of the individual events. We did not exploit the greater slice-level parallelism because we wanted to use the NOvA candidate selection code that was used for the published analysis with modification only to allow the parallelism. Since the operative code worked on one event at a time, we limited our parallelization to the level of the event.

For our demonstration of the HEPnOS data store we chose a set of 1929 of the files that were used for the 2018 analysis. These files represented approximately 4,359,414 triggered readouts of the detector, and 17,878,347 candidate particle interactions that would be examined in the analysis. This represents roughly 1.1% of the full dataset that was used in the NOvA 2018 analysis. We replicated these files 4 times in order to simulate a larger sample for our scaling studies.

IV. Evaluation

We have conducted a number of computational experiments to compare the throughput of the candidate selection workflow when using HEPnOS with the traditional file-based approach. We define *throughput* as the number of slices per second

processed by the whole program (for the HEPnOS workflow) or the whole set of processes (for the traditional file-based workflow). We determine the throughput based on the measured time between the start of data processing for the first rank (HEPnOS) or process (file-based) and the end of data processing for the last rank or process. We have focused on the speed with which data can be read from an already-prepared data service. We have studied the scalability with an increasing number of nodes and also an increasing dataset size.

Both applications, and the workflows in which they are used, have the same general structure. In each case, the data need to be prepared in the appropriate fashion to be available for reading by the application. The applications both use process-level parallelism to read the data from the already-prepared input. Each process reads a sequence of events; each event is read by only one process, and all events are read by some process. For each event, the relevant process iterates through all the slices in the event. Each slice is inspected to determine whether it is accepted or rejected based on whether or not it appears to contain a neutrino interaction (*i.e.* it is a *neutrino candidate*). The function used for this is part of the CAFAna [13] library from the NOvA experiment. The IDs of the accepted slices are accumulated so that we can assure that the two applications have obtained the same results.

In our experiments, we measured the time taken to run the applications. We did not measure the time needed to prepare the data for reading because many of the important uses of this sort of workflow involve reading the data multiple times; it is the speed and scalability of this processing that is of interest to the physicists. For the traditional workflow, we did not include the time taken to concatenate the returned results. This concatenation is not fully automated in the NOvA workflow, and its inclusion would yield somewhat artificially bad comparisons for the traditional workflow. For the HEPnOS-based workflow this step is fully automated, and the time taken is very small. In order to treat both applications similarly, we also ignore the concatenation time in the HEPnOS-based application.

A. Details of the traditional workflow

For the file-based workflow, we automated and parallelized the procedure a physicist would follow to perform candidate selection with the traditional code. The list of input files, and their corresponding storage locations, to be analyzed was specified as a list in a simple text file. This list is used as input to the CAFAna analysis framework that executes the routines that perform the actual selection of candidate events. The configuration of the CAFAna framework is capable of working upon a custom range of files through a starting and ending line number in the aforementioned text file. In this manner the full list of input files can then be decomposed into blocks of work, and then subsequently scheduled in parallel using the Python `multiprocessing` package. For each block of work the program spawns an independent CAFAna routine execution, which then operates sequentially on the analysis files in that subrange block. The Python program is implemented such that the decomposition of the analysis set into subranges, and the scheduling of the work across compute resources can be configured at runtime on a trial by trial basis. This configuration is based on the number of compute nodes, number of processes per node, number of files assigned to each process, and the total number of files. In this manner decomposition and execution of processes over the distributed resources is automated but still requires the user to specify the runtime topology. During execution, each independent routine performs the event selection and writes the list of neutrino interaction slices passing the selection to an independent text file. The processes also measure the time taken to run over all the specified files using `std::clock` calls and write the elapsed time to a separate text file.

B. Details of the HEPnOS based workflow

The HEPnOS-based application uses MPI. Each rank uses a `ParallelEventProcessor` to manage the work of fetching events from the HEPnOS service, and to pass the data to an event processing routine encapsulated by a C++ lambda expression. In this routine, the data are deserialized to recover the NOvA classes which represent the event data. The event is then given to the event-processing routine from CAFAna. The lambda expression then returns the IDs of the selected slices. An MPI reduction is then used to send those slice IDs to rank 0, which writes them to a file after the `ParallelEventProcessor` is executed. The timing data are collected by recording in memory and for each rank timestamps obtained from `MPI_Wtime`. After the event processing is finished, we write these timestamps to a separate file for each rank. The data are analyzed offline to determine the time take to run each step of the process.

Before running the workflow, data needs to be ingested into HEPnOS. For this work, we focused our effort on the HDF5 format because the data were available in this format. In these HDF5 files, data is represented as a hierarchy of groups. Leaf groups are named after the C++ class they store. Inside leaf groups, products are stored in a set of 1-dimensional tables of identical length. Three of these tables correspond to the run, subrun, and event numbers. The rest of the tables correspond to the values of individual member variables of the C++ class being stored. For example an HDF5 file containing instances of the `Particle` from Listing 1 would have six tables: *run*, *subrun*, *event*, *x*, *y*, and *z*.

To simplify ingesting such files, we developed a program, HDF2HEPnOS, which analyzes the structure of an HDF5 file, deduces the class name and its member variables, and generates the C++ code of the corresponding class along with functions to load and store instances to and from HDF5, and to and from HEPnOS. This `DataLoader` can then be compiled and run in parallel to ingest a number of files. It becomes the first step of an HEP workflow, and the only step whose scalability is constrained by the number of files. The next steps of an HEPnOS-based HEP workflow can work at event and product granularity, enjoying the parallelism and load-balancing capabilities of HEPnOS' high-level interface.

C. Software stack

We conducted our performance measurements on Theta at the Argonne Leadership Computing Facility (ALCF). Theta is a Cray XC40 machine that contains Intel Xeon Phi 7230 processors connected by a Cray Aries interconnect in a dragonfly topology [14], [15]. The software stack used by NOvA is based on Scientific Linux 6.[5]. This is a much older operating system version than is used on Theta; for this reason, we created a Docker image containing the NOvA code, which could be deployed using Singularity on Theta. Using the same container image that contains the traditional NOvA application as a base image, we used the Spack package manager [16] to install an environment containing HEPnOS and all of its dependencies. We did this to ensure that the HEPnOS-based application was built with a consistent set of libraries. We made a native build (not in our image) on Theta of the same source code version of HEPnOS. This version was used to run the servers outside of the container while the client processes were run inside the container. We used an installation of libfabric [17] with the user-space Generic Network Interface (uGNI) [18] fabric to harness the full potential of networking bandwidth afforded by the Cray Aries interconnect on Theta. We injected the directories containing the installations of HEPnOS and libfabric (along with other packages required by them) into the container to enable the client nodes to use the same transport protocols. Finally, we used protection domains (provided by the Cray ALPS job manager [15]) to ensure that the client and server were able to communicate.

All the container images used in this work are publicly available via the Dockerhub repository organization heponhpc[6]. The Docker image is converted into a Singularity image suitable for use at many HPC facilities.

D. Experimental setup

For both the traditional and HEPnOS-based workflows, we are interested in measuring how quickly a given dataset can be processed, using a given allocation of computing resources (nodes on Theta). We do this for a few different sizes of the dataset, and for a variety of sizes of allocated resources. For the traditional workflow, the different size datasets are reflected as different numbers of files being processed. For the HEPnOS workflow this is reflected by different numbers of events being loaded into the data service. We use the same datasets for both the traditional and HEPnOS-based experiments. The dataset sizes were 4,359,414, 8,718,828, and 17,437,656 events. The resource allocations we used varied from 8 nodes to 256 nodes.

For the traditional workflow, the datasets corresponded to 1929, 3858, and 7716 files. For each job, we were able to specify the number of nodes and cores per node to use via the Python `multiprocessing` script, itself executed via the `aprun` command. This approach could not take advantage of MPI or multithreading. In this workflow, all the nodes were used, but for the larger resource allocations, not all cores on each node could be used.

For the HEPnOS experiment, we allocated one of every 8 nodes to run the HEPnOS services, and the remaining nodes were used to run the client application processes. Both the HEPnOS service and the client application are MPI applications launched with the Cray `aprun` command. The flexibility resulting from the component design of HEPnOS allowed us to tune (partially automatically) the parameters that configure it.

When using the `std::map` backend, we used 1 MPI rank per node and each rank was given access to 64 cores. When using the the `RocksDB` backend, we used 16 MPI ranks per node, each rank was given access to 4 cores. Hyperthreading was disabled for all the runs. For each HEPnOS process, bootstrapped via their Bedrock component, we used 16 Argobots execution streams for handling RPCs (rpc-xstreams). Each HEPnOS process uses 16 Yokan providers, each mapped to its execution stream to avoid competing for access by multiple execution streams and to improve memory locality. These providers were serving data from 8 event databases and 8 product databases. In the clients, the `ParallelEventProcessor` application was configured so that events are loaded from HEPnOS by a subset of processes in batches of 16384 events (to produce fewer RPCs but with a large data transfer payload), then shared among processes in batches of 64 events (to enable fine-grain load-balancing once events are loaded into worker memory). For a single instance of the HEPnOS server, we use 16 Argobots user-level threads (hereafter ULTs) [4]. This configuration meant that the execution streams were oversubscribing the 4 cores assigned to the instance when using the `RocksDB` backend. These 16 Argbotos ULTs and execution streams are used to run 8 event and 8 product databases (together classified as "bedrock" providers). The loader MPI ranks fetch products in bulk from the HEPnOS servers and also send these products to the worker MPI ranks in bulk. Finally, we ran experiments using both in-memory storage and RocksDB writing to local SSD for Yokan.

All the experimental data was loaded using same number of client nodes used for the particular scaling run. This was necessary because the number of server nodes change in accordance with the number of client nodes (1 server node for every 7 client ndoes), thereby preventing us from reusing servers created for different scaling runs.

E. Results

We first consider the strong scaling behavior of the three workflows (traditional file-based, HEPnOS using RocksDB, and HEPnOS using in-memory storage). We used the largest (7716 file, 17,437,656 event) data sample. We varied the resource allocation from 16 nodes to 256 nodes, selecting configuration parameters for the workflow that obtained the best performance. We ran each experiment several times to obtain an estimate of the spread of running times. We have different numbers of runs because some attempted runs resulted

[5]https://scientificlinux.org/

[6]https://hub.docker.com/u/heponhpc

in crashes caused by failures apparently due to oversaturation of the injection bandwidth of the Aries NIC [15]. This prevented the re-use of servers between runs, thereby causing us to setup and shutdown a server instance for each run. [7] Figure 2 shows the throughput we measured as a function of the number of nodes used in the processing.

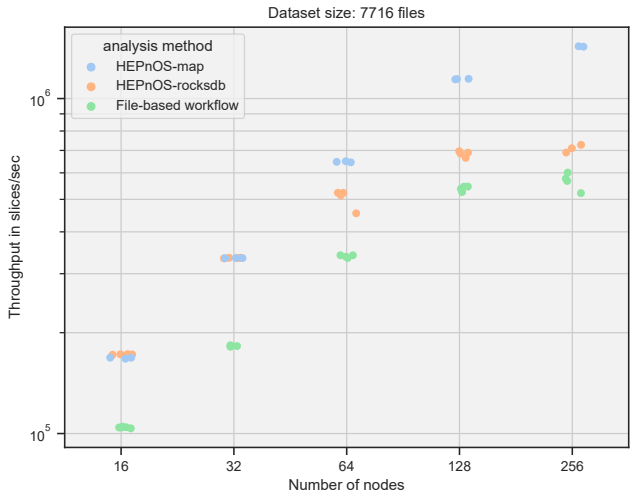

Fig. 2. Plot illustrating the throughput (in slices processed per second) as a function of the total number of nodes used for processing the data using the existing traditional workflow and the HEPnOS based workflows. The performance of the HEPnOS based workflow is superior across all the different number of nodes used. The dots have been jittered to reduce over-plotting.

The figure shows that HEPnOS, whether using the in-memory and RocksDB backend, performs better than does the file-based workflow. Comparing the in-memory and RocksDB versions of HEPnOS, we see that at the smaller node counts use of the RocksDB backend does not cause any inefficiency. However, as the node count increases beyond 32 nodes we see an increasing cost. At higher node counts the in-memory back-end achieves up to twice the throughput. With the in-memory backend the HEPnOS based workflow achieves 85% strong scaling efficiency at 128 nodes. For the kinds of real physics analysis for which this sort of system would be used, the amount of computing resources to be allocated would be determined by the calculations to be performed after the candidate selection was done. For realistic problems, it will almost always be the case that the dataset would fit in the memory available on those resources. We note that the size of the dataset used in these tests is less than 5% of the full dataset used in the related NOvA analysis. The dataset for this analysis is smaller than those used in many other NOvA analysis tasks. Comparing the HEPnOS in-memory performance with the traditional file-based performance, we observe that the file-based application is scaling poorly especially after 64 nodes at which point the number of cores outnumbers the number of file to process.

[7]In communication with the ALCF staff, we were told that this is a failure most often seen in the running of benchmark applications, but rarely seen in applications.

Finally, we consider the throughput as a function of the dataset size, for a fixed computing resource allocation size. This is shown in figure 3. We use the 128 node allocation because, due to techincal issues with the scripts handling the file-based workflow, we were unable to execute that workflow using 256 nodes on the 1929 file data sample. We see that, for the file-based workflow, performance is especially poor for the smaller datasets. This is because, for the smaller datasets, there are not enough files to keep all the cores on the allocated nodes busy. For the 1929 file sample, for example, only 24% of the cores are busy. It also produces imbalance: even when using as many processes as files, the duration of the application will be determined by the duration taken to process the largest file. This effect is greatly lessened in the HEPnOS workflow. While in the file-based workflow the size of a "batch" of events to be processed is determined by the file contents, in the HEPnOS workflow the size of the batch is a tunable parameter. The tuning that was done to optimize the throughput resulted in some residual load-balancing inefficiency.

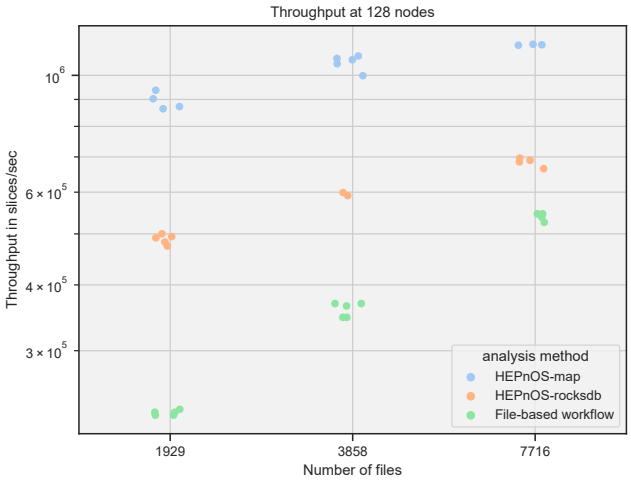

Fig. 3. Plot illustrating the throughput of the traditional workflow compared to the HEPnOS based workflow for varying sizes of datasets using 128 nodes. We see that constraints set by the performance of the parallel file system hamper the throughput achieved by the traditional based workflow for smaller data-sets. The dots have been jittered to reduce over-plotting.

V. Related Work

User-space data services tailored to specific applications have emerged in the past few years to replace or supplement parallel file systems. In situ and in transit analysis systems [19] are examples of such services which aim to provide data analysis and visualization capabilities for HPC simulations without involving the file system. Storage services aim to leverage storage capacity available on compute nodes (usually SSDs or memory) during the execution of an application [2]. Some of these storage services provide a familiar file system interface (e.g. POSIX) while relaxing some constraints on consistency that parallel file systems usually impose. UnifyFS [20], CHFS [21], and GekkoFS [22] are examples of such services. Other services

have seen their interface tailored to specific application use-cases. DataSpaces [23], for example, provides a N-dimentional data model for coupling parallel applications in workflows. Chumbuko [24] and SEER [25] respectively use a document-storage and a key/value-storage model to store performance data. Services like DAOS [26] provide a distributed file-system interface on top of storage-class memory, but also a lower-level object-store interface that applications can tailor to their particular needs.

Overall, HEPnOS adds to a long line of works that shows how moving from the traditional parallel file systems to user-space data services can be advantageous to HPC applications.

While the present paper is the first to introduce HEPnOS in use for its intended purpose, HEPnOS has been used throughout its development by other teams to study various aspects of data services, including work on monitoring and performance diagnostics [5] and AutoML-based autotuning [6]. The former helped diagnose performance problems in early development of HEPnOS and led to some of the optimization listed in this work (batching, parallel event processing). The latter helped us select and optimize relevant parameters (number of databases, batch sizes, etc.) in the present work. An early design of HEPnOS was used to evaluate the potential for storage rescaling [27], a technique that could further improve HEPnOS's potential by allowing users to add and remove storage resources to it while HEP applications are using it. We expect further performance improvements in HEPnOS to come out of all these collaborations.

VI. Conclusion

In this work we have demonstrated that a distributed data service, such as HEPnOS, can be built using the Mochi methodology and components. Such a service can be tuned to yield better performance on HPC resources than achieved by a traditional file-based HEP workflow. We have used a comparatively simple workflow from a current HEP experiment, NOvA, to do this. The improvement is largely due to better, and more tunable, load balancing of the resource usage. The demonstrated neutrino selection problem exhibits this behavior, but the imbalances it displays are small relative to other analysis problems, such as high resolution digital signal processing and particle trajectory fitting which are common throughout the HEP neutrino science space.

Current event-processing workflows used by all HEP experiments use the file-based data distribution paradigm. Most of the dominant workflows in terms of computational loads, that are run by experiments are vastly more complex than the candidate selection example presented here. These workflows, due to their complexity, multi-stage natures, and data interchange techniques usually have greater I/O loads and exhibit large (multi-order of magnitude) spreads in the time taken to process individual events. This results greater runtime load imbalances for these complex workflows than our sample workflow which has near uniform execution time per event, since the combination of non-uniform file sizes and content is compounded with highly non-uniform computational algorithms that the complex workflows are executing. The load balancing advantage available through a distributed data service will be even more valuable to these complex workflows and will permit the averaging out of outliers in the computational distributions.

Each HEP experiment uses a framework for constructing its complicated event simulation and event processing workflows. The designs of these frameworks interfaces to their I/O layers will need to change in many cases to take full advantage of a distributed data store and realize the full performance potential of the data store. In future work we will evaluate the advantages of the use of HEPnOS in a much more complete HEP workflow using one of these frameworks.

Acknowledgment

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, grant 1013935. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. 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.

References

[1] M. Dorier, P. Carns, K. Harms, R. Latham, R. Ross, S. Snyder, J. Wozniak, S. Gutierrez, B. Robey, B. Settlemyer, G. Shipman, J. Soumagne, J. Kowalkowski, M. Paterno, and S. Sehrish, "Methodology for the Rapid Development of Scalable HPC Data Services," in *Proceedings of the PDSW-DISC 2018 workshop (SC18)*, 2018, workshop. [Online]. Available: https://sc18.supercomputing.org/proceedings/workshops/workshop_pages/ws_pdsw106.html

[2] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham *et al.*, "Mochi: Composing data services for high-performance computing environments," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 121–144, 2020.

[3] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, 2013, pp. 1–8.

[4] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2018.

[5] S. Ramesh, R. Ross, M. Dorier, A. Malony, P. Carns, and K. Huck, "Symbiomon: A high-performance, composable monitoring service," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2021, pp. 332–342.

[6] M. Dorier, R. Egele, P. Balaprakash, J. Koo, S. Madireddy, S. Ramesh, A. D. Malony, and R. Ross, "Hpc storage service autotuning using variational- autoencoder -guided asynchronous bayesian optimization," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2022, pp. 381–393. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CLUSTER51413.2022.00049

[7] D. S. Ayres *et al.*, "The NOvA Technical Design Report," FERMILAB-DESIGN-2007-01, 10 2007.

[8] P. Adamson *et al.*, "The NuMI Neutrino Beam," *Nucl. Instrum. Meth. A*, vol. 806, pp. 279–306, 2016.

[9] R. L. Workman and Others, "Review of Particle Physics," *PTEP*, vol. 2022, p. 083C01, 2022.
[10] M. A. Acero *et al.*, "Improved measurement of neutrino oscillation parameters by the NOvA experiment," *Phys. Rev. D*, vol. 106, no. 3, p. 032004, 2022.
[11] ——, "New constraints on oscillation parameters from ν_e appearance and ν_μ disappearance in the NOvA experiment," *Phys. Rev. D*, vol. 98, p. 032012, 2018.
[12] ——, "First Measurement of Neutrino Oscillation Parameters using Neutrinos and Antineutrinos by NOvA," *Phys. Rev. Lett.*, vol. 123, no. 15, p. 151803, 2019.
[13] C. Backhouse, "The cafana framework for neutrino analysis," 2022. [Online]. Available: https://arxiv.org/abs/2203.13768
[14] S. Parker, V. Morozov, S. Chunduri, K. Harms, C. Knight, and K. Kumaran, "Early Evaluation of the Cray XC40 Xeon Phi System "Theta" at Argonne," 5 2017. [Online]. Available: https://www.osti.gov/biblio/1393541
[15] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray inc., white paper wp-aries01-1112," Technical report, Cray Inc, Tech. Rep., 2012.
[16] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack Package Manager: Bringing Order to HPC Software Chaos," ser. Supercomputing 2015 (SC'15), Austin, Texas, USA, November 15-20 2015, lLNL-CONF-669890. [Online]. Available: https://github.com/spack/spack
[17] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A brief introduction to the openfabrics interfaces - a new network api for maximizing high performance application efficiency," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 34–39.
[18] H. Pritchard, I. Gorodetsky, and D. Buntinas, "A ugni-based mpich2 nemesis network module for the cray xe," in *Recent Advances in the Message Passing Interface*, Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 110–119.
[19] H. Childs, S. D. Ahern, J. Ahrens, A. C. Bauer, J. Bennett, E. W. Bethel, P.-T. Bremer, E. Brugger, J. Cottam, M. Dorier *et al.*, "A terminology for in situ visualization and analysis systems," *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 676–691, 2020.
[20] A. Moody, D. Sikich, N. Bass, M. J. Brim, C. Stanavige, H. Sim, J. Moore, T. Hutter, S. Boehm, K. Mohror, D. Ivanov, T. Wang, C. P. Steffen, and U. N. N. S. Administration, "Unifyfs: A distributed burst buffer file system - 0.1.0," 10 2017. [Online]. Available: https://www.osti.gov/biblio/1408515
[21] O. Tatebe, K. Obata, K. Hiraga, and H. Ohtsuji, "Chfs: Parallel consistent hashing file system for node-local persistent memory," in *International Conference on High Performance Computing in Asia-Pacific Region*, 2022, pp. 115–124.
[22] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "Gekkofs-a temporary distributed file system for hpc applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 319–324.
[23] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci *et al.*, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–9.
[24] C. Kelly, S. Ha, K. Huck, H. Van Dam, L. Pouchard, G. Matyasfalvi, L. Tang, N. D'Imperio, W. Xu, S. Yoo *et al.*, "Chimbuko: A workflow-level scalable performance trace analysis tool," in *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2020, pp. 15–19.
[25] P. Grosset, J. Pulido, and J. Ahrens, "Personalized in situ steering for analysis and visualization," in *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2020, pp. 1–6.
[26] M. Hennecke, "DAOS: A scale-out high performance storage stack for storage class memory," *Supercomputing frontiers*, p. 40, 2020.
[27] N. Cheriere, M. Dorier, G. Antoniu, S. M. Wild, S. Leyffer, and R. Ross, "Pufferscale: Rescaling hpc data services for high energy physics applications," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 182–191.