Coupling Chapel-Powered HPC Workflows for Python

John Byrne
john.l.byrne@hpe.com
Hewlett Packard Enterprise

Harumi Kuno
harumi.kuno@hpe.com
Hewlett Packard Enterprise

Chinmay Ghosh
chinmay.ghosh@hpe.com
Hewlett Packard Enterprise

Porno Shome
shome@hpe.com
Hewlett Packard Enterprise

Amitha C
amitha.c@hpe.com
Hewlett Packard Enterprise

Sharad Singhal
sharad.singhal@hpe.com
Hewlett Packard Enterprise

Clarete Riana Crasta
clarete.riana@hpe.com
Hewlett Packard Enterprise

David Emberson
emberson@hpe.com
Hewlett Packard Enterprise

Abhishek Dwaraki
abhishek.dwaraki@hpe.com
Hewlett Packard Enterprise

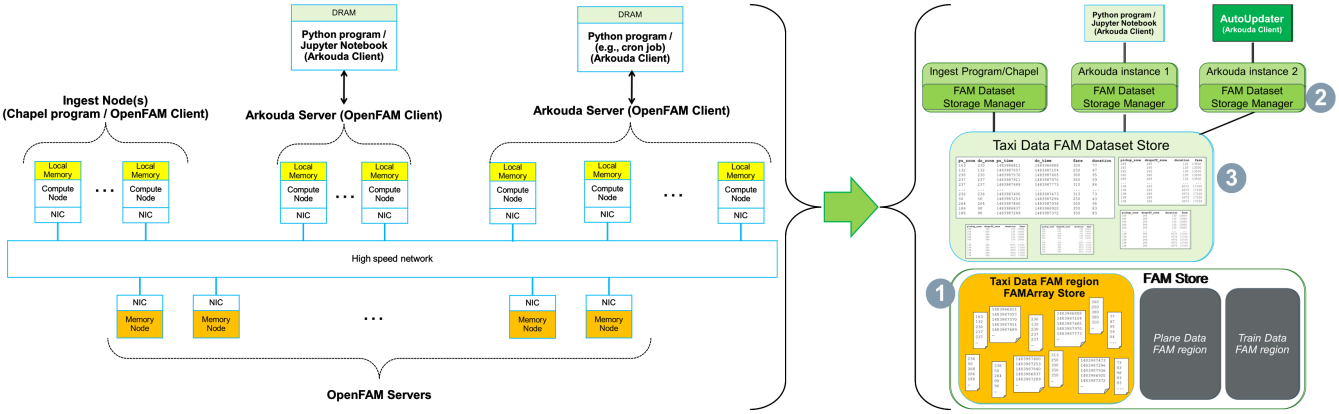

Figure 1: Powered by Chapel, Arkouda, and OpenFAM, the FAM Dataset Storage Manager enables Python programmers to fully utilize Fabric-Attached Memory to create incrementally-updated shared datasets that speed time to results.

ABSTRACT

Decades ago, when data analytics was known as data mining, there was an adage – "No data, no mining!" The pendulum has swung to the opposite extreme, as everything from hospitals to cars now produce massive quantities of data. We address the challenge of how to lower the barrier for efficiently processing massive quantities of data. We propose a solution that enables ordinary Python programmers to share results while working efficiently with datasets that can be too large to process using a single commodity machine. Our solution leverages Chapel, Arkouda, and OpenFAM to hide complexity, transparently enabling programmers to process large amounts of data on clusters of compute nodes while making it easy for them to share and incrementally maintain derived datasets.

ACM Reference Format:
John Byrne, Harumi Kuno, Chinmay Ghosh, Porno Shome, Amitha C, Sharad Singhal, Clarete Riana Crasta, David Emberson, and Abhishek Dwaraki. 2023. Coupling Chapel-Powered HPC Workflows for Python. In *CHIUW 2023*. ACM, New York, NY, USA, 7 pages.

CHIUW'23, June 2023, USA

1 INTRODUCTION

The goal of this work is to demonstrate an early proof of concept of how Python programmers can take full advantage of Fabric-Attached Memory (FAM) and solve problems that are too large to fit into the memory of a cluster of compute nodes. High Performance Computing (HPC) is characterized by computationally demanding workloads that discover or create intellectual products and that respond to scale, meaning that if more resources are applied, a more valuable product will be produced[6]. As the amount of available data has increased, the scope of HPC workloads has expanded to include AI and Data Analytics, and AI and Data Analytics workloads have expanded to take on an HPC-like response to scale[6]. It is thus important to lower the bar and enable HPC, AI, and Data Analytic communities to make effective use of available resources, including pools of FAM.

To this end, we bring together Arkouda, Chapel, and OpenFAM and propose a FAM dataset storage management system [4, 5, 7]. As sketched in Figure 1, the FAM Dataset Storage Manager is designed to enable programmers to increase dataset size and and to improve the throughput of their analytics. Ingested and derived data is processed and stored incrementally in a FAM Dataset Store, after which the programmer can define new derived columns and new derived datasets by applying operations such as filter, scatter, gather, and sort. To execute such operations in parallel, the FAM

Dataset Storage Manager uses Chapel, Arkouda, and OpenFAM to page batches of data from FAM into the memory of compute nodes for efficient local processing. In this way, the FAM Dataset Storage Manager supports FAM datasets that exceed the total DRAM capacity of compute nodes.

Once stored in FAM, these results can be shared with other programmers and used to speed time-to-results for further operations. Furthermore, the FAM Dataset Storage Manager tracks and stores the workflows used to create derived datasets and columns. Besides enabling programmers to update their derived data on-demand in interactive sessions, this also means that as new data is added to a FAM Dataset Store, derived datasets and columns can be maintained by an automatic update processor, sketched in the upper right-hand corner of the figure. Our approach opens the door for enabling suspend-and-resume and recovery-from-failure.

In the remainder of this paper, we describe our solution. We discuss our goals and general approach in Section 2. We have completed a preliminary implementation that demonstrates our approach running on an HPC cluster of HPE nodes connected using HPE's Slingshot hardware. We describe our extensions to Chapel and Arkouda that support this proof-of-concept in Section 3, then discuss in Section 4 how we used those extensions to implement a FAM Dataset Storage Manager as a Python application. We discuss related work in Section 5. Finally, we conclude in Section 6 by briefly summarizing the contributions of this work and lessons learned, and describing potential next steps.

2 APPROACH

The Arkouda Python package enables data scientists to interactively explore extremely large datasets extremely efficiently. At its heart, Arkouda makes it easy for Python programmers to work with parallel distributed arrays (pdarrays) as first-class Python objects. Each Arkouda pdarray object represents a collection of physical in-memory data arrays. The Arkouda server is written in Chapel, and thus each Arkouda pdarray object represents a collection of physical in-memory data arrays stored in the memory of compute nodes. The Python programmer can work with these arrays as they would work with a numpy array; they can sort, filter, and perform other operations upon those arrays, thereby producing results as new, derived, pdarray objects. The programmer simply invokes these operations; the Arkouda server then uses Chapel to distribute the operations across multiple compute nodes and execute them efficiently, in parallel.

The OpenFAM library for programming Fabric-Attached Memory enables the creation of pools of fabric-attached memory (FAM) hosted on conventional servers. Developers can write programs that, using the OpenFAM API, perform remote direct memory access (RMDA) operations on the disaggregated memory. Programmers can use the OpenFAM API to allocate shared memory to create *regions* of FAM from a given pool, as well as *data items* within regions. The OpenFAM API supports a host of data path operations on data items, including put, get, scatter, gather, copy, backup, and restore, as well as standard atomic operations such as fetch-and-add, compare-and-swap, etc. For example, Figure 2 shows an OpenFAM client program that has performed a gather operation to collect two values from an array of data stored in a FAM data item residing in a FAM region hosted by three memory nodes.

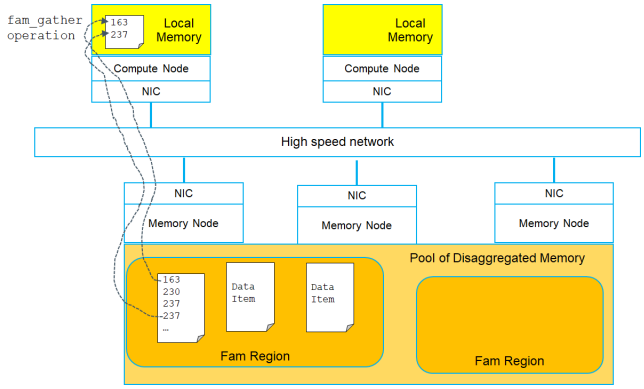

Figure 2: OpenFAM client programs can access data items residing in FAM regions hosted on OpenFAM servers.

OpenFAM API also includes map and unmap operations so as to support load/store operations on remote memory, as well as a variety of operations and constructs that facilitate the ordering of operations, including barriers, and support for user-specific context objects. For more information, readers are referred to the OpenFAM Reference Implementation [1].

One benefit of FAM is cost-effectiveness, in that multiple compute nodes can share a pool of high bandwidth, low-latency memory. However, although FAM offers much higher bandwidth and lower latency than other forms of storage, remote memory is by default lower bandwidth and higher latency than local memory. Thus, when considering how to enable Arkouda users to exploit FAM to solve problems that are too large to fit into the memory of the compute nodes, we need to consider non-uniform memory access times and are motivated to leverage local memory where possible.

A second, significant, benefit of FAM is that data stored in the memory pool can be shared between multiple users and processes. This means that results can be stashed in FAM and leveraged to speed time to future results.

The key ideas of our approach are illustrated in Figure 1. Sketched in the box labeled "1" at the bottom of the right-hand side of Figure 1, base data and metadata are stored in FAM as discrete batches. Batches are sized such that a single batch of data can be efficiently processed by the Arkouda Server within the memory of the compute nodes. The FAM Dataset Storage Manager, instances of which are shown in the boxes labeled "2" at the top of the right-hand side of Figure 1, manages metadata and data, organizing the discrete batches of data into logical FAM Datasets. Each FAM Dataset, sketched in the light green box labeled "3" in the middle of the right-hand side of Figure 1, represents a logical integrated view of a collection of related batches of data that have accummulated over time.

Similar to an Arkouda DataFrame, a FAM Dataset consists of an index, column names, and column data. However, one difference is that FAM index and column data are composed of multiple batches

of data. A second difference is that the FAM Dataset Storage Manager supports the creation of derived indexes and columns that are defined by applying operations to existing FAM indexes and columns. Operations that produce new indexes (e.g., filter, sort_runs, and sort) produce derived datasets. Operations that use existing orderings to produce new column data (e.g., gather, scatter, subtraction, addition) produce derived columns. The FAM Dataset Storage Manager lets programmers snapshot FAM data into Arkouda pdarrays or Arkouda DataFrame objects.

As shown in Figure 1, FAM Datasets that reside in the same FAM region are organized into a logical FAM Dataset Store. The figure shows three FAM regions in the bottom right-hand corner – one dedicated to "Taxi" data, one for "Plane" data, and one for "Train" data. The metadata records of the FAM Dataset Store indicate which batches of data have been published for all indexes and columns. This means that the Dataset Storage Manager that created them has made them available for use by other Dataset Storage Manager instances running in other Python processes. The FAM Dataset Store also records for each derived FAM Dataset index or column which batches of data have been consumed from its sources. Metadata itself is also shared on FAM in batches, so as to facilitate sharing and maintenance.

In general, data can reside in the memory of the machine where a Python program is executing, in the memory of the compute nodes where an Arkouda Server is running, or in pooled Fabric-Attached memory. Figure 3 sketches our approach, whereby shared data and metadata resides in the FAM, in batches. When a Python programmer operates upon a FAM Dataset, the FAM Dataset Storage Manager pages batches of data from FAM into Arkouda parallel distributed arrays (pdarrays) so that working data will reside in the local memory of compute nodes. Similarly for the sake of performance, each instance of the FAM Dataset Storage Manager maintains an in-memory working copy of the Dataset Store's metadata in Python.

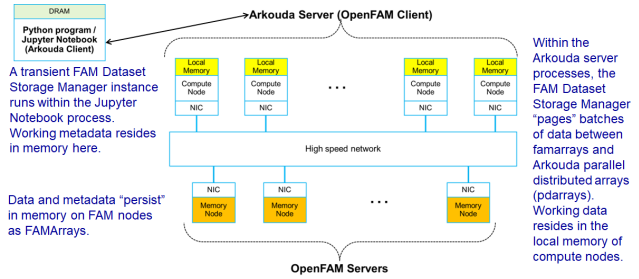

Figure 3: Data and metadata are stored in batches on FAM and processed in the memory of compute nodes.

3 EXTENSIONS TO CHAPEL AND ARKOUDA

We extended both Arkouda and Chapel in order to support FAM Dataset storage management. These extensions, shown in Figure 4, enable Python and Chapel programmers to create, access, and work with named single-dimension arrays of data that reside in FAM.

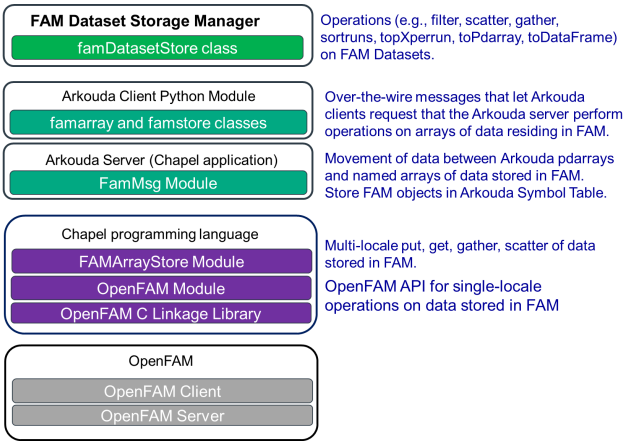

Figure 4: New modules add support for FAM Store and FAMArray to Arkouda and Chapel.

3.1 OpenFAM module and C linkage library for Chapel

The Arkouda server is implemented in HPE's Open Source Chapel programming language. The Chapel compiler parses Chapel source code and generates C code in the back end. The C code is in turn compiled using a standard C compiler into an executable and linked with the Chapel runtime library, which is written in C. Because OpenFAM is written in C++ and provides C++ APIs, we added a C linkage library that binds the OpenFAM C++ library to the Chapel runtime, as well as an OpenFAM Chapel module that allows Chapel, including the Arkouda server, to access the C linkage library.

3.2 FAMArrayStore module for Chapel

The OpenFAM module and underlying C linkage library simply enables Chapel programs to invoke OpenFAM operations; it is not responsible for managing Chapel tasks. As such, the OpenFAM module requires that all the virtual addresses involved in remote memory access (RMA) operations to the OpenFAM server must exist within the process address space of a single Chapel locale. Thus, a thread running in a given Chapel locale cannot instruct OpenFAM to read data from a different Chapel locale's process space into FAM. This requirement must be taken into account when performing IO between FAM and Chapel distributed arrays. For example, because the Chapel block distribution attempts to partition arrays evenly across locales, in proportion to each array, if an array with 100 elements were to be distributed across two locales, we should place 50 elements in each locale, whereas an array with 1000 elements would be distributed to place 500 elements in each locale.

The Chapel FAMArrayStore module handles such tasks, and more, for the programmer. There is one instance of the Chapel FAMArrayStore per OpenFAM region. Distinct arrays within a given FAMArrayStore instance must have distinct names. The FAMArrayStore supports an API for creating, looking up, and manipulating named arrays of data stored in FAM. The FAMArrayStore module internally converts these high-level array operations into FAM-specific accesses underneath. When an operation is invoked,

each locale is then assigned a portion of the FAM data for processing. Thus, parallel array operations are executed in parallel by the Chapel programmer.

An example of the end-to-end result of the Chapel and Arkouda extensions is sketched in Figure 5, which shows Python program running on a laptop using the FAMArray Store (the variable "fam_region" in the example) to manage arrays and scatter/gather data across FAM and compute nodes.

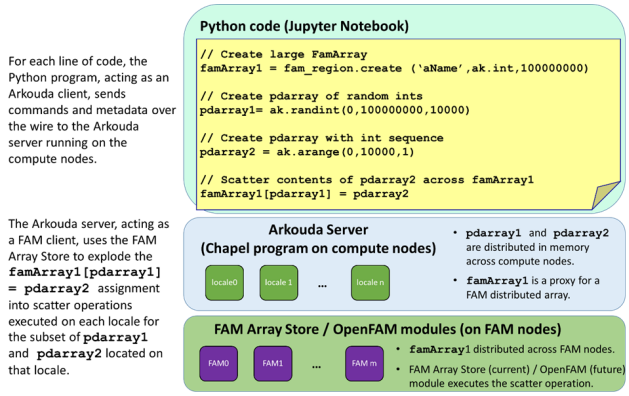

Figure 5: Example of how the new modules extend the existing Arkouda package to let a Python programmer scatter data across FAM and compute nodes.

Data and some metadata for each logical FAMArray is persisted using objects in FAM. Persisted metadata includes the type and number of items in the array. To support variable length strings, the offsets and sizes of individual strings are stored for FAMArrays containing PackedString data. Chapel FAMArrayStore instances maintain internal state at runtime to enable the distributed processing of FAM data by Chapel locales. For example, the Chapel FAMArrayStore associates each Chapel locale with its own FAM descriptors for accessing FAM objects and FAM regions.

The Chapel FAMArrayStore throws both FAM and FAMArray errors as enumerated exceptions (e.g., FAM_ERR_NOTFOUND). This enables programmers to reason about these exceptions in their code – for example to implement a find-or-create pattern for a named FAMArray.

At the current time, aggregation is implemented inside the Arkouda server because it allows certain assumptions to be made that simplify the logic. That said, the OpenFAM team has implemented an extended Chapel OpenFAM module that explicitly supports FAM-resident distributed arrays [3]. As Chapel evolves to support FAM data types, we anticipate that Chapel's support for auto-aggregation may allow a generalized data movement solution to be written concisely.

3.3 Extensions to the Arkouda Python module

Python programs using Arkouda act as clients of an Arkouda server, which runs on nodes in a computing cluster. To facilitate this, Arkouda provides the Python 'pdarray' class, each instance of which represents a distributed in-memory array on the Arkouda server. Through a combination of operator overloading plus additional methods, the pdarray class allows a Python programmer to remotely invoke operations on the Arkouda server by sending wire commands. Arkouda users can persist pdarray datasets by saving them to or reading them from files stored on the underlying file system using formats such as HDF5 and Parquet.

As sketched in Figure 5, we extended the Arkouda Python module to expose the FAMArrayStore. Two new classes enable programmers to create and attach to FAMArrayStore instances (one FAMArrayStore instance per FAM Region) and to manage the FAMArrays stored within each FAMArrayStore:

- famstore class – A famstore object is a reference to a named OpenFAM region opened by the Arkouda server that contains FAMArrays.
- famarray class – A famarray object contains a reference to a pdarray stored in FAM. The famarray class is cloned from the pdarray class and modified to add lock/unlock methods. Operations can be performed on it to load/store data to/from pdarrays in the server, and to logically lock/unlock the famarray objects. These follow the patterns set by pdarray. For example, __getitem__ and __setitem__ are used to move data in and out of FAM using the existing syntax for pdarrays.

3.4 Extensions to the Arkouda Server

Arkouda supports three styles of assignment/indexing operations on pdarrays – simple assignment operations like put and get, which involve a single index and a single value; slice operations, which access a single pdarray using optional parameters of start (inclusive), stop (exclusive), and stride; and scatter/gather operations, which use a pdarray as an index to another pdarray. A new Arkouda Server module, FamMsg, defines functions that extend the set of wire commands supported by the Arkouda Server with operations such as famStoreCreate, famStoreDelete, fam[int], fam[pdarray], fam[int]=val, fam[pdarray]=val, fam[pdarray]=pdarray. The logic for these operations is implemented using the Chapel FAMArrayStore module. FAM objects participate in the Arkouda server's symbol table.

3.5 HDF5

Although it is not directly used by the FAM Dataset Storage Manager at this moment, in addition to saving data directly as arrays in FAM, we also added an OpenFAM connector for HDF5. As sketched in Figure 6, the OpenFAM connector maps FAM storage into the HDF5 data model through the VOL layer [2]. This connector enables applications, regardless of programming language, to read/write HDF5 datasets to FAM using the popular HDF5 data format. HDF5 access facilitates ports of existing applications that already use HDF5. HDF5 access also eases the sharing of data between heterogeneous applications. For example, a data scientist may want to work with data received from an external source in HDF5 format.

4 FAM DATASET STORAGE MANAGEMENT WORKFLOW

The FAMArray extensions to the Arkouda and Chapel modules make it very easy for Python and Chapel programmers to create, access, and manipulate arrays of data stored in FAM. The heavy

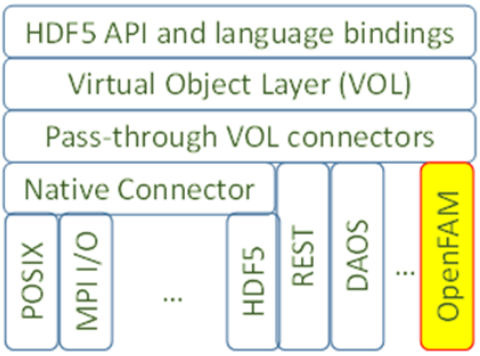

Figure 6: The FAM volume connector lets users create, read, and write HDF5 files on FAM. (This figure is based on a drawing from slide 31 of [2].)

lifting provided by these extensions enabled us to implement the FAM Dataset Storage Manager in user space as a Python application. Because the FAMArray Store insulates the Python user from the logistics of how to move data between the Arkouda servers and FAM, the logic implemented to realize the FAM Dataset Storage Manager largely focuses on metadata management, calling upon the underlying FAMArray Store to perform appropriate operations.

4.1 Data is ingested and transformed into batches

Ingested data is stored in FAM as single-dimension FAMArrays. Each batch is broken into a single FAMArray per data field. For example, Figure 7 sketches how a Chapel Ingest Application takes as input some NYC taxicab data and stores it on FAM as numbered batches using FAMArrays. We will use this data as a running example.

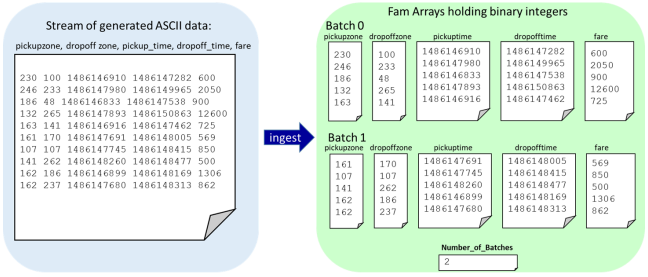

Figure 7: Data is ingested in ordered batches into discrete arrays of data on FAM

4.2 Metadata

Both the indexes and column data backing a FAM Dataset Store are stored in FAM using FAMArrays. In order to present this data in terms of integrated logical FAM Datasets, the FAM Dataset Storage Manager maintains and shares metadata about which FAMArrays hold data for which columns and indexes and their batch identifiers, as well as metadata that tracks structural and derivation relationships between columns and indexes/datasets. Other metadata associates unique (currently strictly increasing) identifiers with data items, and tracks which batches of data a derived dataset has processed from its parent data sources as well as which batches of data have been published for a given index or column. Like index and column data, shared metadata is stored on FAM in batches, using FAMArrays.

For example, the left side of Figure 8 sketches the two batches of data ingested in Figure 7. The figure shows that the FAM Dataset Storage Manager uses metadata to present a dataset consisting of five columns to Arkouda programmers: pickupzone, dropoffzone, pickuptime, dropofftime, and fare.

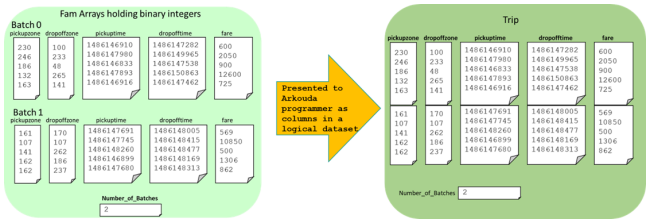

Figure 8: Presenting discrete arrays of data in FAM as an integrated FAM Dataset

For performance, when operating in Python, the FAM Dataset Storage Manager manages working metadata about each running instance in local memory. As noted in Section 3, the FAMArray Store can store variable-length strings in arrays on FAM. The FAM Dataset Storage Manager uses this functionality to implement an internal mechanism for storing and incrementally updating these Python dictionaries using append-only dictionaries as batches of data stored in FAM. The FAM Dataset Storage Manager uses this FAM dictionary mechanism to create, maintain, share, and incrementally update its working metadata.

4.3 Derived columns and derived datasets

The FAM Dataset Storage Manager enables Python programmers to create derived FAM columns and FAM Datasets using operations such as filter, gather, add, subtract, topXperrun, sortruns, and colsort. For example, Figure 9 shows a derived dataset that consists of the indices of items from the dataset in Figure 8 that correspond to trips with fares greater than 10,000 cents ($100). As sketched in the figure, at this point, this dataset consists of only an index – it has no column data that corresponds to this index. Programmers can invoke a "gather" operation to add columns to a derived dataset – using an index to gather data from a parent dataset into a column on a child dataset.

Derived columns represent values associated with indexed data items. These values can be gathered directly from a derived dataset's parent dataset. For example, Figure 10 sketches a "gather" operation that collects the "pickuptime" values for trips with fare greater than $100 USD and stores them in batches that together comprise a new derived column associated with the derived dataset from Figure 9.

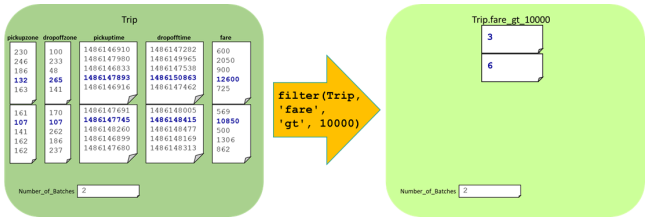

Figure 9: A derived dataset consists of index data.

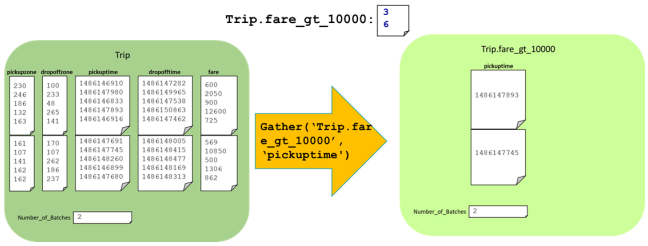

Figure 10: A gather operation materializes a column of interest, adding it to a derived dataset.

4.4 Order-Preserving vs. Order-Destroying Derivation Operations

Note that the FAM Dataset Storage Manager tracks the number of batches available from a derived data object's parent as well as the number of batches processed when driving the data object's content. Explicitly tracking this information enables the FAM Dataset Storage Manager to offer incremental processing when updating derived columns and datasets.

To support incremental processing in the face of updates to the underlying base data, the FAM Dataset Storage Manager internally distinguishes between order-preserving versus order-destroying derivation operations when creating derived datasets. Order-preserving derivation operations operate per batch – the addition of a new batch of data to the base dataset does not impact previously computed results. For example, it is simple to update the derived dataset index and its derived columns in response to the ingestion of an additional batch of base Taxi data because the new batch can be processed and simply appended to existing results. The same would not be true for a dataset derived using a full sort operation on, for example, the "fare" column of the base dataset, because the newly added "fare" data could be scattered throughout the results when sorted. Logical columns and datasets that are derived directly or indirectly from order-destroying operations must be fully updated when the data underlying the order-destroying operation changes. For example, a column derived using a subtract operation upon two columns gathered from an index defined by a full sort operation must also be recomputed when the underlying base table changes.

To facilitate the incremental update of sorted data, the FAM Dataset Storage Manager implements two order-preserving operations related to sort. The sort_runs operation produces a dataset whose index represents a partially-sorted ordering in which each batch's results are individually sorted. Similarly, the topXperRun operation extracts the top n elements from each batch, based on the values of a specified column. Datasets and columns derived using these operations can be incrementally updated.

4.5 Incremental Updates

Because the FAM Dataset storage manager tracks the derivation relationships between datasets and stores both base and derived Datasets in FAM, it can leverage prior results and update datasets incrementally in response to changes to base datasets. Incremental processing offers the added advantage that it enables Arkouda programmers to work with data sets that far exceed the combined memory of compute nodes.

It is important that all times during update, the metadata behind FAM Datasets must be kept in a consistent state, where consistent means that every batch of data that is published as available is actually available, and that the metadata dictionaries accurately reflect column name / dataset relationships. To incrementally update all derived FAM Datasets in a FAM Dataset Store, the FAM Dataset Storage Manager walks through all FAM Datasets from oldest (e.g., the base FAM Datasets and other ancestors of derived datasets) to the newest. For each FAM Dataset, if it is a derived dataset, it first updates the FAM Dataset's index data by applying the appropriate operations upon its data sources. It then walks through the FAM Dataset's columns, from oldest to newest and similarly updates the column data as appropriate.

The update methods that operate upon index data and column data have a parameter "Increment", which defaults to True so that by default the FAM Dataset Storage Manager will compare the published batch identifiers of the data source to the identifiers of batches that have been previously consumed, and not recompute results unnecessarily. If the "Increment" parameter is set to False, then the FAM Dataset Storage Manager will recompute all index and column data – we use this option to evaluate the impact of incremental updates.

4.6 Automatic Updates

In order to support automatic updates, the FAM Dataset Storage Manager exposes a method that users can call to register a derived FAM Dataset and all of its ancestors for automatic updates. This method basically adds these FAM Datasets to an internal metadata list of registered derived datasets that should be automatically updated.

The FAM Dataset Storage Manager's update method accepts an optional parameter – "autoupdate", which defaults to False. When "autoupdate" is set to True, instead of walking through all FAM Datasets from oldest (e.g., the base FAM Datasets) to the newest (the most recently derived), it will instead walk through the base FAM Datasets and just the derived FAM Datasets that have been registered for automatic update. Automatic updates can then be realized by running an Arkouda Server on one or more compute nodes and then running a simple Python script that invokes the update method with autoupdate=True. This Python update script can be added directly to the ingest process or invoked periodically (e.g., through a cron job, as sketched in Figure 1).

5 RELATED WORK

Our proof-of-concept demonstrates mechanisms that enable Python programmers to get full use from an HPC cluster equipped with a pool of Fabric-Attached Memory. Our extensions to Chapel and Arkouda shield Chapel and Python programmers from the complexity of reasoning about how to work with disaggregated memory when working with a compute cluster. Our FAM Dataset Storage Manager leverages these extensions to shield Python programmers from the complexity of reasoning about how to derive datasets through workflows and to maintain derived index and column data automatically and incrementally. Because our batch-centric approach is designed to process large datasets incrementally, we enable Python programmers to use FAM and process datasets that potentially exceed the memory of compute nodes by transparently paging batches data between FAM and compute-node memory.

Other researchers and practitioners focus on paging data at the granularity of memory pages. For example, Peng et al. provide UMap, a scalable and extensible userspace service for memory-mapping datastores that mapping datastores into the application process's virtual memory space to provide a unified "in-memory" interface [8]. In another example, Wahlgren et al. investigate the feasibility of using CXL type 3 devices to implement and use a CXL-based composable memory subsystems on future HPC systems by emulating how various workloads would perform if executed using a combination of local and pooled memory [9]. Unlike such approaches, our approach keeps the working dataset in the memory of compute nodes, when possible using bulk transfers to move data between local DRAM and FAM.

6 CONCLUSIONS AND NEXT STEPS

We have implemented a proof-of-concept that demonstrates how Python users can interactively use an HPC cluster of HPE nodes connected using HPE's Slingshot hardware to create, operate upon, and incrementally maintain logical datasets that represent batches of data stored in fabric attached memory (FAM). As sketched in Figure 1, we dedicate an Arkouda server to process updates, use compute nodes to run Chapel ingest programs that add new batches of data to FAM Datasets, and dedicate a partition of memory nodes to serve as a memory pool, via OpenFAM Servers.

As mentioned earlier, the FAM Dataset Storage Manager was implemented in Python. We find that from a Python programmer's perspective, the Chapel and Arkouda famarray modules successfully hid the complexity of moving data between the memory pool, the compute nodes, and the Python program.

We have also identified some opportunities for future improvements. For example, when working with a library that uses a communications library such as OFI or verbs, registering all the memory in one's heap up-front can be good for performance, and in fact Chapel does this internally for itself. We believe there may be a performance benefit if the FAMArray module could get access to the Chapel registered heap and register it for its own endpoints for use by OpenFAM. Doing this would let the Chapel FAMArray module use that memory when moving data between remote memory managed by OpenFAM. The challenge is how to enable the "user-level" Chapel FAMArray module to get a pointer to the Chapel heap. Chapel does have a non-advertised function for getting the Chapel heap, and we were able to use it (after turning off stack checking). However, one thing that was not addressed out-of-the box was that if the Chapel program is compiled to execute on a single node, then the Chapel compiler will believe that no inter-node communication is necessary and thus will not register a heap. This is an issue in our case because the FAMArray module would still like to register the Chapel heap in order to move data between Arkouda processes and the OpenFAM servers. We have filed a "wish list" issue for Chapel, requesting an environment variable that we could use to override single node case, forcing Chapel to, for example, build for an OFI communications library.

This project represents a work-in-progress. We anticipate that when the work described in [3] is merged into Chapel, then it will be straightforward to to port the FAMArray module to use the resulting extended Chapel. We believe that our approach for supporting automated incremental updates opens the door to explore useful functionality such as suspend-and-resume, operation-offloading, and recovery-from-failure.

We are also interested in the exploration of the potential benefits of applying our array/batch-oriented approach to Dataset Storage Management beyond OpenFAM to other mechanisms for sharing data – for example, using a filesystem or formats such as Parquet, Feather, Arrow, or HDF5.

ACKNOWLEDGMENTS

We thank HPE's OpenFAM and Chapel teams, as well as the Arkouda community for their gracious and valuable support. We thank the CHIUW reviewers for their thoughtful feedback and excellent suggestions for improvement.

REFERENCES

[1] [n. d.]. OpenFAM Reference Implementation. Retrieved May 25, 2023 from https://openfam.github.io/index.html

[2] M. Scot Breitenfeld, Elena Pourman, Suren Byna, and Quincey Koziol. 2020. Achieving High Performance I/O with HDF5. HDF5 Tutorial ECP Annual Meeting 20202. Retrieved April 21, 2023 from https://www.hdfgroup.org/wp-content/uploads/2020/02/20200206_ECPTutorial-final.pdf

[3] Amitha C, Brad Chamberlain, Sharad Singhal Sharad, and Clarete Crasta. 2022. Extending Chapel to Support Fabric Attached Memory. Retrieved April 21, 2023 from https://chapel-lang.org/CHIUW/2022/C.pdf

[4] Bradford L. Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael P. Ferguson, Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, and Greg Titus. 2018. Chapel Comes of Age : Making Scalable Programming Productive.

[5] Kimberly Keeton, Sharad Singhal, and Michael Raymond. 2019. The OpenFAM API: A Programming Model for Disaggregated Persistent Memory. In *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*, Swaroop Pophale, Neena Imam, Ferrol Aderholdt, and Manjunath Gorentla Venkata (Eds.). Springer International Publishing, Cham, 70–89.

[6] Bill Magro. 2018. Software Foundation for High-Performance Fabrics in the Cloud. Video. Retrieved April 21, 2023 from https://insidehpc.com/2018/04/intels-bill-magro-presents-software-foundation-high-performance-fabrics-cloud/

[7] Michael Merrill, William Reus, and Timothy Neumann. 2019. Arkouda: Interactive Data Exploration Backed by Chapel. In *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop* (Phoenix, AZ, USA) *(CHIUW 2019)*. Association for Computing Machinery, New York, NY, USA, 28. https://doi.org/10.1145/3329722.3330148

[8] Ivy Bo Peng, Maya B. Gokhale, Karim Youssef, Keita Iwabuchi, and Roger Pearce. 2022. Enabling Scalable and Extensible Memory-Mapped Datastores in Userspace. *IEEE Trans. Parallel Distributed Syst.* 33, 4 (2022), 866–877. https://doi.org/10.1109/TPDS.2021.3086302

[9] Jacob Wahlgren, Maya Gokhale, and Ivy B. Peng. 2022. Evaluating Emerging CXL-enabled Memory Pooling for HPC Systems. In *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE. https://doi.org/10.1109/mchpc56545.2022.00007