2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)

Methodology for the Rapid Development of Scalable HPC Data Services

Matthieu Dorier, Philip Carns, Kevin Harms, Robert Latham, Robert Ross, Shane Snyder, Justin Wozniak
Argonne National Laboratory, Lemont, IL, USA
{mdorier,carns,robl,rross}@anl.gov, harms@alcf.anl.gov, {ssnyder,wozniak}@mcs.anl.gov
Samuel K. Gutiérrez, Bob Robey, Brad Settlemyer, Galen Shipman
Los Alamos National Laboratory, Los Alamos, NM, USA
{samuel,brobey,bws,gshipman}@lanl.gov
Jerome Soumagne
The HDF Group, Champaign, IL, USA
jsoumagne@hdfgroup.org
James Kowalkowski, Marc Paterno, Saba Sehrish
Fermi National Laboratory, Batavia, IL, USA
{jbk,paterno,ssehrish}@fnal.gov

***Abstract*—Growing evidence in the scientific computing community indicates that parallel file systems are not sufficient for all HPC storage workloads. This realization has motivated extensive research in new storage system designs. The question of which design we should turn to implies that there could be a single answer satisfying a wide range of diverse applications. *We argue that such a generic solution does not exist.* Instead, *custom data services* should be designed and tailored to the needs of specific applications on specific hardware. Furthermore, custom data services should be designed in close collaboration with users. In this paper, we present a methodology for the rapid development of such data services. This methodology promotes the design of reusable building blocks that can be composed together efficiently through a runtime based on high-performance threading, tasking, and remote procedure calls. We illustrate the success of our methodology by showcasing three examples of data services designed from the same building blocks, yet targeting entirely different applications.**

***Index Terms*—HPC, Storage, Data, Services**

I. Introduction

Most scientific computing endeavors require more than just a single compute job or application to produce a meaningful result; they are instead manifested as campaigns that span instruments, computing facilities, and time. These scientific campaigns rely on a diverse array of data models and access methods, but the state of the practice for data management in this environment relies on an aging concept of files stored in parallel file systems. Such a data management method will become unsustainable as data models, applications, and resources become larger and more heterogeneous.

From a technology perspective, parallel file systems cannot absorb the growing amount of data generated by computation, and I/O becomes the bottleneck of parallel applications. This is evidenced by the increasing adoption of in situ analysis methods as alternatives to storing all the raw data. Parallel file systems are also limited by the interface they have to provide for genericity reasons and by the standards they need to follow [1].

From an application perspective, they require the use of several layers of library [2]–[4] and middleware [5] to bridge the semantic gap that exists between the in-memory data and the file representation [6].

These challenges, in conjunction with emerging NVRAM, solid state, and in-system storage topology, present an opportunity to reconsider our approach to data management in HPC. Replacing the ubiquitous parallel file system model is a daunting task, however. Any truly generic solution will inevitably be suboptimal for some applications and system platforms, while continuous feature expansion leads to untenable complexity and maintainability challenges.

We propose an alternative strategy: agile creation of new data services tailored to the needs of individual applications or problem domains.

This concept is obviously advantageous for applications that are not well served by generic storage solutions, but it presents a number of practical problems. If new data services are created from the ground up, they will lack the code maturity necessary for portability, maintainability, and subtle platform optimizations. This calls for a new methodology to enable rapid development of HPC data services that does not compromise practical deployment capability.

Over the past three years, our group of researchers from Argonne National Laboratory, Los Alamos National Laboratory, Carnegie Mellon University, and the HDF Group, gathered within the *Mochi project*, have worked on defining and implementing such a methodology, which this paper presents.

This methodology promotes the design of reusable building blocks that can be composed together efficiently through a runtime based on high-performance threading, tasking, and remote procedure calls. It enables the development of very different application-tailored data services with maximal code

978-1-7281-0192-7/18/$31.00 ©2018 IEEE
DOI 10.1109/PDSW-DISCS.2018.00011

reusability across services and minimal "glue." This glue code consists of (1) the client-side interface is a more productive alternative to generic, "one-size-fits-all" data model API (that is, applications interface directly with the service rather than going through multiple software layers) and (2) the composition code (or *daemon*), which defines how components are deployed, configured, and interact with one another.

We illustrate the success of our methodology by showcasing three examples of data services designed from the same building blocks, yet targeting entirely different applications. FlameStore provides a distributed cache for Python-based, deep learning workflows; HEPnOS is a distributed, hierarchical event-store for high energy physics codes written in modern C++; and SDSDKV is a distributed key/value store for parallel trajectory splicing in molecular dynamics applications.

Note that this paper *does not* aim to present performance evaluation of these system, which will be left for future papers presenting them individually and extensively. Rather, our paper focuses on the methodology and its effectiveness in terms of (1) development speed and (2) code reuse. Our previous work [7] provides performance evaluation of a simple component that is indicative of the capabilities of these services.

The rest of this paper is organized as follows. Section II presents our methodology for efficiently building HPC data services. We present an implementation of this methodology in Section III. Section IV illustrates this methodology with three examples of actual data services that we developed. Section V presents related work and Section VI summarizes our conclusions and briefly discusses future work.

II. Methodology

In this section we present our methodology for efficient development of HPC data services that are tailored to specific applications and problem domains. This methodology is based on composing a set of reusable building blocks in such a way that the resulting service satisfies user requirements. These building blocks cover features needed by most HPC data services (such as storing raw data on a local device or managing a key/value database). They are designed to be easily composable so that the code required to build a service fits in a few lines of code. Beyond this composition code, the only programming process required is building the client-side interface to the data service, which in this approach is tailored to the application, rather than mandating the use of a single API.

Our methodology is summarized in Table I. This section explains it step by step.

A. User requirements

The first step consists of gathering the requirements from users and applications that will access the services. These requirements come in the form of (1) the data model that is exposed by the application; (2) the applications' access patterns; and (3) guarantees that the service should meet, including consistency guarantees for the identified access patterns.

1) Data model: The data model is the data representation that the application works with. For example, a machine learning application may work with *NumPy arrays*. Scientific simulations and visualization applications usually work with *meshes* and *fields*. Scientific instruments such as particle accelerators generate *events*. Rather than flattening these data structures into files, the data service should interface directly to the application's data model. It should offer access, query, or indexing functionality that is relevant to the data model. It could even support receiving code segments from users to be executed on the stored data, avoiding costly data transfers. The data model also includes the namespace that the data service should expose, ranging from attributing objects a unique id, to organizing them in a hierarchy of named directories, or within a graph. It also includes how the data will be queried from the storage service.

2) Access pattern: Different applications exhibit different access patterns, ranging from randomly accessing many small objects to collectively accessing large ones. They may intensively access metadata or access them rarely. They may mostly read or mostly write. Knowing this access pattern, or at least the most frequent ones, ahead of service design is critical. It helps decide whether the service will need to rely on distributed metadata, object sharding, or other such data and metadata organization.

3) Guarantees: User requirements also include guarantees that the data service should satisfy. These guarantees are usually tied to the access pattern and the data model. They include at least the consistency model. For example, the application's access pattern may be write-once-read-many, indicating that no locking is needed.

B. Service requirements

Having established the user requirements, the second step in this methodology is to turn them into service requirements. These requirements describe how data will be concretely managed by the data service. These include the data organization, the metadata organization, and the service's interface. Note that there is no one-to-one matching between the users requirements and the service requirements. All the users requirements should be taken into account when defining the service requirements.

1) Data organization: Defining the data organization implies answering a number of questions. These questions include the following.

- **How should objects be distributed?** This distribution can be based on many factors, such as the object's metadata (e.g., using a hash of the object's name); its content (e.g., whether the object exhibits a certain property); the localization of its writer or its potential readers (e.g., to improve data locality); or even the availability (storage space or bandwidth) of particular storage servers at the time the object is written.
- **Should objects be sharded?** Large objects may be split into chunks and distributed across multiple storage targets, or kept entirely in one storage target. This decision

TABLE I: Our methodology matches user requirements (left) with generic, composable building blocks (right). Its goal is to maximize code reusability by enabling composition with minimal glue code, which consists of some small composition code and custom interfacing code tailored to the application.

User requirements	Service requirements		Building blocks (Mochi implementation)
• Data model – Arrays – Structured objects – Distributed structures – ... • Access pattern – Large vs. small accesses – Number of accesses – Concurrency – ... • Guarantees – Performance – Consistency – Fault tolerance – ...	• Data organization – Sharding – Distribution – Replication – ... • Metadata organization – Distribution – Indexing – Content – Namespace – ... • Interface – Language(s) – Programming interface – ...	**Composition and Interfacing**	• Runtime – Threading and tasking (Argobots) – Networking and RDMA (Mercury) – Serialization (Mercury, Boost, ...) • Providers – Local storage (Bake) – Local database (SDSKV) – Group membership (SSG) – Performance monitoring (MDCS) – Data indexing (Plasma) – Local processing (Poesie)

depends on typical access patterns from the application and on other aspects such as whether users can ship code to be executed on storage.

- **Should objects be replicated?** The answer to this question may depend on the required data availability, access performance, or fault tolerance concerns.

2) Metadata organization: Design decisions related to metadata organization are also driven by a number of questions, including the following.

- **Should metadata be distributed?** Applications that exhibit a metadata-intensive access patterns (e.g., creating/accessing many objects at the same time) likely require distributed metadata. Data services that manage a few large objects may not.
- **How should metadata be distributed?** If distributed, localizing metadata can be achieved in different ways. A hierarchical namespace may hash paths to map directory metadata to metadata servers. Other techniques can be used that rely on a distributed hash table, or on properties contained in the metadata.
- **What should the metadata contain?** Traditional file system metadata include the object's name, owner, permissions, and creation time. A custom data service may expose more information, such as a summary of the objects it describes (e.g., statistical information), or dependencies to other objects, or even access statistics for informed caching/prefetching.
- **How should the metadata be accessed?** Applications or users may need to query metadata based on something else than a path. For example, they may need to access the metadata related to all the objects written in a given timeframe or to all objects whose content exhibits a given property. The answer to this question drives decisions regarding the possible use of indexing techniques.

3) Service interface: Defining the interface to the service first consists of choosing the language this interface should be written in. This part of the design also includes defining how the interface interacts with the service's component. For example, a typical protocol to retrieve an object consists of (1) localizing the metadata server hosting that object's information; (2) querying this metadata server; (3) localizing the server hosting the object's data; and (4) querying this storage server. An alternative protocol consists of having the metadata act as a proxy, forwarding the request to the right storage server that performs server-directed remote direct memory access (RDMA) to the client's memory. User requirements drive the design of these protocols.

C. Building blocks

The third step in our methodology consists of matching the service requirements defined above with a *service design* based on building blocks.

In the right part of Table I we have identified a set of building blocks that are recurring in HPC data services. They include a high-performance communication library, a threading/tasking library, some component to control local storage space (SSD, disks, etc.), and some component to store and efficiently look up key/value pairs.

1) Terminology: When we say "building blocks" we are referring to the core libraries (threading, networking, serialization) as well as the components. The communication library, the threading/tasking library, and the serialization library play a particular role in the design of a distributed data service, since they are at its core and can be used by the rest of the building blocks. We call this group of libraries the "runtime." We call "component" a code relying on the runtime to provide remotely-accessible functionalities on a single node. Components export a "client" library and interface, and a "server" library and interface. The server library of a component contains methods to start up "providers." A single provider executing on a node forms a "microservice." A set of providers composed together across potentially multiple nodes form a "service." Providers must have a number of properties, described hereafter.

2) Limiting scope: A provider should be limited in terms of functionality that it provides. For example, if a provider

enables raw data storage on a device *and* also performs some processing on it (e.g., indexing), the developer must consider splitting such a provider into two separate ones, thus increasing the reusability and generality of each. Limiting the scope of functionality that a given provider offers is also a key to easing maintainability, since there is less functionality to test and implementation changes will not affect unrelated functionality.

3) Remote and local accessibility: The runtime used to develop service providers should transparently expose their functionality as remote procedure calls and as local procedure calls. Doing so enables clients (applications or other providers) to access the functionality of a provider without needing to know whether that provider runs on the same node or on a separate node. When deploying a service, this aspect enables users to test different mappings of providers to nodes and find the one that leads to the best performance on a given platform and for a given workload.

4) Decoupling: Providers must be as much as possible decoupled from one another. That is, they should be developed, maintained, and tested separately prior to being integrated and tested as a whole within a service. This decoupling increases the development productivity by limiting as much as possible the dependencies across teams working on different providers.

If a provider does depend on another one, it should be agnostic to the particular configuration of that other provider. For example, a user must be able to switch the particular implementation of a storage provider (e.g., from storing data in memory to storing it on an SSD or on a disk) without this change affecting other providers. This decoupling with respect to implementation is key to later adapting a service to a particular use case by simply configuring its providers independently.

5) Composability: Providers must be *composable*, that is, able to share the runtime without introducing dependencies among one another, without requiring providers to execute in distinct nodes, in distinct processes, or even in distinct threads.

Composability is the main reason the runtime is external to the providers. Execution resources (cores, threads, time slot in a scheduling policy) are managed by the runtime and assigned to providers based on a configuration that is external to all providers. For communications, providers should rely on a common communication runtime that will manage queuing and dispatching of requests to providers.

D. Composition and interfacing

The last step in our methodology is to compose providers on top of the runtime to make up a data service that satisfies the service requirements and ultimately the user requirements.

1) Ease of composition: The providers should be easy to compose. When reaching the development phase, this capability can be measured by how much code is required to compose providers. Well-designed building blocks should allow composing to be achieved with less than a couple of hundred lines of code. Ideally, this could even be made generic, with the composition being described in a YAML, JSON, or XML file, rather than programmatically.

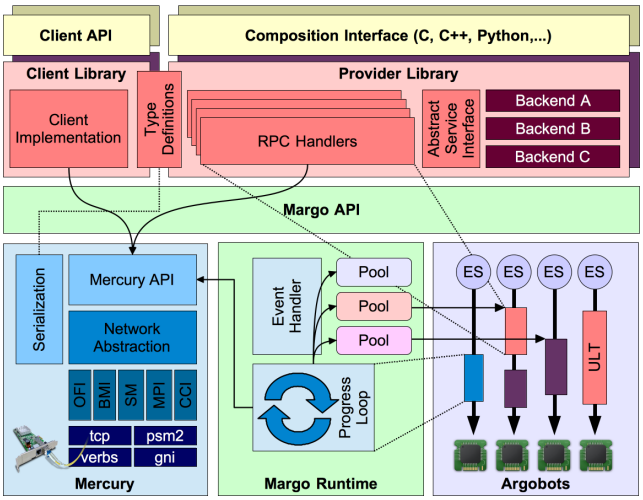


Fig. 1: Mochi component

2) Interface: The interface to the service (i.e., what the end user will see) requires the most development effort, because it is specific to each individual application. This interface should hide the composition in such a way that the application remains agnostic of how the service is implemented, how it is distributed across available resources (although in some cases this information may be exposed to optimize access locality), and how it is configured. This interface may be written in a language other than that of the providers.

III. Architecture of a Mochi Component

This section describes how we implemented the above methodology within the context of the Mochi project.

A. Anatomy of a Mochi component

Mochi components are built by using the Mercury and Argobots libraries, brought together within the Margo runtime. This section describes the different parts of such a component, whose architecture is shown in Figure 1.

1) The Margo runtime:

a) Mercury: Mercury[1] [8] is a C library for implementing remote procedure calls (RPC) and optimized for HPC systems. It features a number of network plugins, including libfabric [9], BMI [10], and CCI [11]. These plugins support various transport protocols ranging from TCP to HPC fabric transports such as Cray GNI [12]. Mercury enables RDMA of bulk data either by using the native capabilities of the underlying network plugin or by using shared memory if the target process is colocated within the same node.

Mercury abstracts the notions of client and server to favor instead origin and target semantics, which facilitates the development of services and their composition. It also provides a number of preprocessor macros to produce the C code necessary to serialize data structures. The API uses a callback-based model with explicit progress, which gives services

[1] http://mercury-hpc.github.io/

flexibility in making decisions on progress, execution of RPC callbacks, and the threading model used.

b) Argobots: Argobots[2] [13] (lower right in Figure 1) is a threading/tasking library designed for massively multicore processors. It directly leverages the lowest-level constructs in the hardware and operating system: lightweight notification mechanisms, data movement engines, memory mapping, and data placement strategies, in order to make the best use of the available resources.

Argobots decouples the notion of *execution streams* (ESs) mapped onto hardware threads and that of *user-level threads* (ULTs) and *tasks*, which are scheduled to be executed on execution streams. This programming model enables reducing contention and increasing concurrency by avoiding explicit locks (such as mutex) when possible and by relying instead on ULTs mapped to resources according to their dependencies and yielding to one another as needed.

c) Mercury + Argobots = Margo: Argobots gives us an opportunity to simplify the Mercury programming model, which is callback-driven with an explicit progress loop. To make Mercury and Argobots operate together, we developed Margo[3] (lower middle in Figure 1)). Margo hides Mercury's progress loop in an Argobots ULT, which either executes in a dedicated ES or shares an ES with other components.

Using Margo, Mercury callbacks are replaced with ULTs. This way of programming RPC handlers and Mercury client code leads to a much more natural control flow than does reasoning with callbacks. As an example, the following function in Mercury sends an RPC to a remote target.

```
hg_return_t HG_Forward(
    hg_handle_t h,        /* RPC handle */
    hg_cb_t cb,  /* completion callback */
    void *uarg, /* arguments for the cb */
    void *in);      /* input of the RPC */
```

This function immediately returns. The provided callback will be triggered within the Mercury progress loop once a response has been received. Meanwhile, the user must keep track of other work that can be completed while waiting for a response.

Margo, in contrast by doing implicit progress, provides the following function.

```
hg_return_t margo_forward(
    hg_handle_t h,       /* RPC handle */
    void *in);     /* input of the RPC */
```

This function sends the RPC, then yields back to the ES' scheduler, leaving the current ES free to be used by another ULT to perform useful work. Once the response is received, the ULT running the Mercury progress loop will yield back to the ULT that called `margo_forward` so that it can continue executing. The user no longer needs to keep track of what to do while waiting for a response. *This aspect is the key to microservice composition:* the Margo runtime is in charge of scheduling the work submitted by providers on the available resources. Margo can therefore transparently schedule ULTs from multiple providers without these providers being aware of one another.

2) Components: We design Mochi components on top of Margo. Each component consists of a client library and a server library. The client library defines a set of functions that issue an RPC to a target provider and wait for a response. The server library defines a *provider*, which encapsulates a set of functionality typically implemented in a number of backends. For example, the SDSKV provider, described later, exposes a put/get interface to a key/value store, and includes backends for BerkeleyDB and LevelDB. The Poesie provider enables one to remotely execute code written in languages such as Python or Lua. A common header is used by both client and server libraries to define the data types of RPC arguments and return values. These definitions are used by Mercury to generate serialization and deserialization code (although in some examples we bypassed this serialization to provide our own, based on either Boost or Python's pickle).

Multiple providers can execute on top of the same Margo runtime, in the same node. One provider can be a client of another provider. Clients and providers can reside in the same node or in different nodes. When in the same node, Mercury will optimize RPCs by using shared memory to transfer data.

3) Interface and bindings: Each component typically exposes a C interface for both its client-side and server-side libraries. The client-side interface is used to connect to a provider and to access its functionality. The server-side interface gives the means to spin up and set up a provider and compose it with other (local or remote) providers.

For convenience, most of the components we developed have a Python interface as well. *The main advantage of this Python interface is to enable writing the composition of providers in the form of a simple Python script that can then be deployed on the machine.*

Wrappers to the Margo runtime for Python (Py-Margo) and for C++ (Thallium) also enable the development of complete Mochi components in these languages.

B. Example Mochi components

The Mochi framework includes a set of common, predefined components that conform to the conventions described in the preceding section. Key examples are enumerated below.[4]

1) **Bake** provides remote access to persistent storage extents. Bake is optimized for RDMA networks and non-volatile memory. It uses Mercury and libpmemobj [14] to achieve low latency and high bandwidth on HPC platforms [7].
2) **SDSKV** provides remote access to key/value storage instances. It includes a modular backend with support for LevelDB [15], BerkeleyDB [16], and in-memory databases for flexible deployment without modifying the data service architecture.

[2]http://www.argobots.org/

[3]https://xgitlab.cels.anl.gov/sds/margo

[4]All these components, as well as Margo, Py-Margo, and Thallium, are available at https://xgitlab.cels.anl.gov/sds.

3) **Scalable Service Groups (SSG)** is a group membership system that can be used to assemble single-instance providers such as Bake and SDSKV into distributed services. SSG employs the SWIM gossip-based group membership protocol [17] to detect failures and evict dead members from the group.
4) **MDCS** is a lightweight component that tracks performance and usage metrics from other providers, for diagnostic purposes. For example, Bake can use MDCS to record statistics on bandwidth, access sizes, and number of operations, thereby making these statistics remotely available to the users or other service providers.
5) **Poesie** is used to embed programming language interpreters (currently Lua and Python) within a Mochi provider. Poesie clients can send code to Poesie providers to execute remotely on their behalf. Coupled with Python wrappers for building block components, Poesie enables on-the-fly recomposition and reconfiguration of Mochi services.
6) **ch-placement**[5] is a modular consistent hashing library that can be used by distributed services to map objects to storage servers in a reproducible and replication-friendly manner. It is not a Mochi provider (as it does not follow the client/server design pattern) but rather a utility library that generalizes a common data service capability.

IV. Examples of Custom HPC Services

The methodology presented above and its implementation within the Mochi project enabled us to quickly develop a number of storage services tailored to particular applications. This section showcases three of them: FlameStore, HEPnOS, and ParSplice. Note that the purpose of this paper is not to evaluate these services.It is to show how the methodology was used to design them. Future papers will focus on extensively presenting these services and evaluating their performance.

A. FlameStore

FlameStore[6] is a transient storage service tailored to deep learning workflows. It was developed to meet the needs of the CANDLE cancer research project.[7] These workflows train thousands of deep neural networks in parallel to build predictive models of drug response that can be used to optimize preclinical drug screening and drive precision-medicine-based treatments for cancer patients. Following discussions with users, FlameStore required only a few weeks of development to reach a first working version.

1) User requirements: Since CANDLE workflows train deep neural networks using the Keras framework [18] in Python, FlameStore needs to present a Python interface capable of storing Keras models (the workflow's *data model*). More generally, this can be achieved by enabling storing NumPy arrays along with JSON metadata.

[5]https://xgitlab.cels.anl.gov/codes/ch-placement/
[6]https://xgitlab.cels.anl.gov/sds/flame-store
[7]http://candle.cels.anl.gov/

The workflow's access pattern consists of writing potentially large NumPy arrays. Overall, users expect such models to range from a few hundreds megabytes to a few gigabytes. These arrays are written once and never modified.

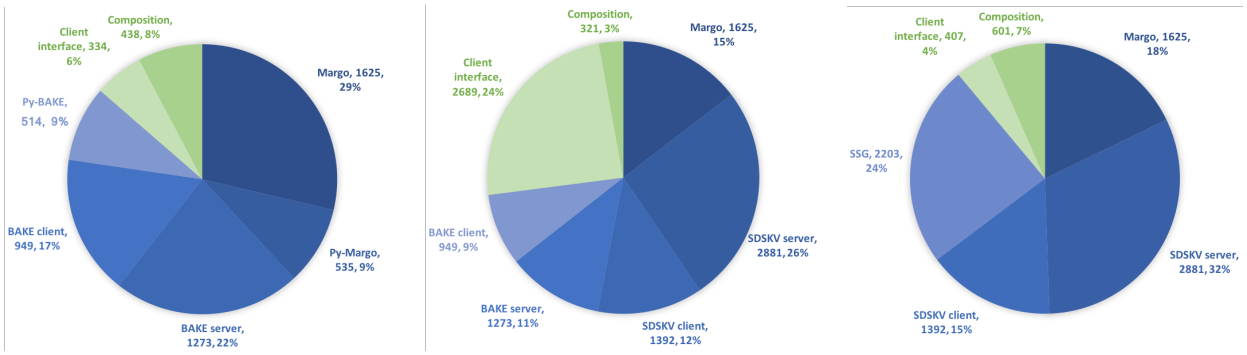
Users requested that FlameStore provide a flat namespace, that is, a simple mapping from a unique model name to a stored model. Trained models need to also be associated with a score indicating how well they perform on testing datasets. FlameStore needs to store such a score along with other user-provided metadata (including the hyperparameters used for training the model) that can be used for querying particular models. Users may also want to send Python code to nodes storing a model in order to perform local computation (e.g., evaluating some properties of the stored models in order to make decisions).

FlameStore needs to be a single-user service running for the duration of the workflow that accesses it. It needs to act like a semantic-aware distributed cache built on federated storage space (RAM, NVRAM, or disks) provided by compute nodes. It is backed up by a traditional parallel file system for persistence across multiple workflow executions.

2) Service requirements: Based on the user requirements, we expect FlameStore to store few (on the order of a thousand) large objects that need to be written atomically, read atomically, and accessed locally in a consistent manner. Hence we expect large data transfers to be the critical aspect of the service to optimize. We will need the storage space for these objects to be distributed. Because we need to be able to execute code within the data service to do some processing on single models, we need each model to be stored on a single node. This also aligns with the fact that workflow workers do no collectively work on the same model.

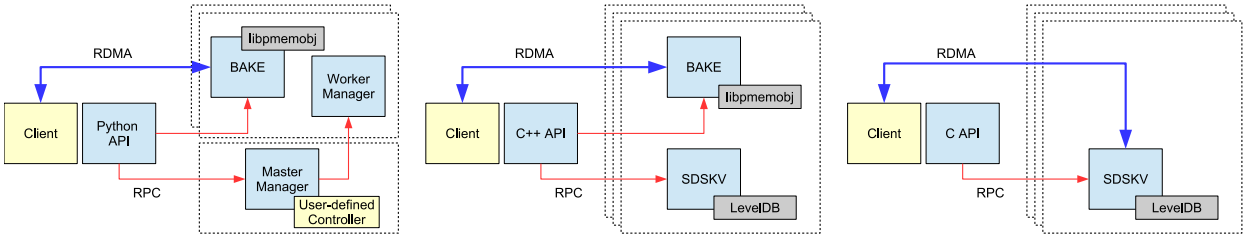
We do not expect metadata to be a bottleneck, and we can therefore use a single node to manage it. However, SDSKV is not sufficient to handle the type of queries expected from the workflow: FlameStore, indeed, needs not only to store the metadata but also to make decisions on *where* to store each model, based on colocality with the node that generates it, on available space in each storage node, and on the content (semantics) of the data.

3) Implementation with Mochi components: Figure 3a shows the organization of components used in FlameStore. Its implementation primarily relies on Bake for storage management. It uses PyMargo to implement a custom Python-based provider for semantic-aware metadata management and another custom provider for the management of storage nodes. PyBake is used to interface with Bake using Python. This Python interface also enables RDMA transfers of NumPy arrays to Bake providers. FlameStore's composition code is entirely written in Python. In the future, we plan to integrate the Poesie component to enable shipping Python code to the storage servers at run time. Figure 2a provides the number of lines of code used by FlameStore's components as well as the percentage this code represents: 86% of the code consists of reusable components, the remaining 14% comprising the client-side interface (6%) and the composition code and



(a) FlameStore (b) HEPnOS (c) SDSDKV

Fig. 2: Single Lines of Code (SLOC) of the three example services, broken down into common components and custom code.



(a) FlameStore (b) HEPnOS (c) SDSDKV

Fig. 3: Architecture of our three data services. The Margo runtime and some components such as MDCS and SSG have been omitted for simplicity.

custom providers (8%). Note that this figure does not include the lines of codes of Argobots (15,193) and Mercury (27,959) since these libraries existed before the Mochi project and could be replaced with alternatives in the implementation of our methodology. According to our git history, only 15 days were needed to finish a first version that users could start working with.

FlameStore enables users to plug in a *controller* module, written in Python, that implements smart data management policies. This controller makes decisions including persisting good models in HDF5 files; discarding models that have been outperformed by other models; migrating models to improve load balancing or data locality; or compressing models that are unlikely to be reused but still need to stay in cache.

FlameStore ensures that models are written only once and atomically. It does not allow updates and partial writes. It does not replicate data by default but enables the controller to duplicate models across multiple storage locations if they need to be reused by multiple workflow workers.

Metadata in FlameStore consist of (1) a model's name, (2) a JSON-formatted model architecture, (3) the location of weight matrices (i.e., network address of the storage node and indexing information), and (4) additional user-provided metadata (e.g., the hyperparameters used for training, the accuracy of the model, and the network address of the client that is writing it). User-provided metadata are used to drive the controller's decisions.

Clients write in FlameStore by first contacting the metadata provider with the model's metadata. The metadata provider responds with the identity of a Bake provider in which to write the model. At this point the metadata provider marks the model as "pending." It is not yet visible to other clients. The client contacts the selected Bake provider, which issues RDMA pull operations to transfer the NumPy arrays from the client's memory. Upon completion, the client contacts the metadata provider again to complete the model's metadata with the location of the stored NumPy arrays.

Clients read models by contacting the metadata provider with the model's name. The metadata provider returns the model's metadata, which include the information on how to retrieve NumPy arrays from Bake providers. The metadata is sometimes the only information clients need, since it encapsulates the entire model's architecture as well as user-provided metadata. If needed, the client can request the NumPy arrays from the corresponding Bake providers, which will transfer them using RDMA push operations.

4) Example client code: Listing 1 shows an example of client code that interacts with FlameStore. The client starts by initializing a `WorkspaceHandle`, which points to a directory in a parallel file system where FlameStore configuration files are located (with information on how to connect to the providers) and where models are stored when they are flushed to storage by FlameStore workers. The `store_model` and `load_model` are the main two functions for the client to use. The former extracts the metadata from the Keras model

and sends them to the Master Manager, which invokes the controller and responds with the identity of a Bake provider to contact for storing the layers' NumPy arrays through RDMA. The `load_model` contacts the Master Manager to get the metadata, then loads the NumPy arrays through RDMA back to the client and recomposes the model.

```
from flamestore.workspace import WorkspaceHandle
# open a FlameStore workspace
workspace = WorkspaceHandle("path/to/my/workspace")
model = ...     # define and train your Keras model
# store the model into FlameStore
workspace.store_model("mymodel", model)
# reload a model from FlameStore
model = workspace.load_model("mymodel")
# shutdown the FlameStore service
workspace.shutdown()
```

Listing 1: Example of client code using FlameStore

B. HEPnOS

HEPnOS[8] is a storage service targeting high energy physics experiments and simulations at Fermilab, and developed in the context of the SciDac4 "HEP on HPC" project.[9] It aims to eventually replace Fermilab's data storage system, which currently relies on ROOT files [19] stored in a parallel file system.

1) User requirements: Scientists at Fermilab currently use ROOT files to store the massive amount of events produced by their high energy physics experiments, and also by simulations and data-processing codes. Aiming to replace ROOT to achieve better performance, better use of new technologies, and more development simplicity, we started to develop HEPnOS to specifically address their needs.

HEPnOS needs to organize data objects in a hierarchy of datasets, runs, subruns, and events. These containers act in a way similar to directories but map better to the way high-energy physics experiments organize their data. Datasets are identified by a name and can contain runs as well as other datasets. Runs, subruns, and events are identified by an integer. Runs contain subruns; subruns contain events. The notions of "relative path" and "absolute path" make it possible to address a container relative to another or relative to the root of the storage system, respectively.

Events data consist of serialized C++ data objects. Hence, HEPnOS needs to present a C++ interface that resembles that of the C++ standard library's `std::map` class, allowing to navigate items within containers using iterators. The expected access pattern is, as in FlameStore, write-once-read-many, with only atomic accesses to single objects. However, users expect a much larger number of objects (several millions). These objects, after serialization, typically range in size from a few bytes to a few kilobytes.

2) Service requirements: Based on the user requirements, we defined the following service requirements. HEPnOS will need to distribute both the data and the metadata, given the large number of objects that it will store. Objects will not be sharded, but contrary to FlameStore the reason is their small size rather than because they need to be accessed locally.

Ultimately, Fermilab envisions running HEPnOS in production in a multiuser setting. In order to deal with fault-tolerance in this context, HEPnOS needs to enable both data and metadata replication. This also enables potentially better read performance.

Data and metadata will be queried based on the full path of the object; hence no particular indexing method is required.

Optimizations should also be implemented to enable bulk-loading and bulk-storing objects, in order to avoid the cumulated latency of many RPC round trips when storing or loading objects one at a time.

3) Implementation with Mochi components: Figure 3b shows the organization of components used in HEPnOS. HEPnOS uses Bake to store objects and SDSKV to store metadata. Typically, each service node hosts one Bake provider and one SDSKV provider, although we have not yet evaluated whether this setting is the best-performing one.

The SDSKV providers storing the information on a particular container (dataset, run, subrun, event) are selected based on the hash of the container's parent full path. Hence all the items within a given container are managed by the same set of nodes. Metadata related to serialized C++ objects, however, are managed by nodes chosen by hashing the full name of the object. This matches the expected sequential access to directory entries, versus parallel accesses to data objects.

HEPnOS also optimizes data accesses by storing small objects within their metadata, in a way similar to file systems storing data in their inodes when the data are small enough. Benchmarks should be executed on a given platform to establish the threshold below which embedding data inside metadata is advantageous.

HEPnOS bypasses Mercury's serialization mechanism and relies on Boost.Serialization instead, in order to enable serializing C++ objects with minimal changes to the user code.

Contrary to FlameStore, clients write in HEPnOS by first storing their object's data into multiple Bake providers in parallel. They then contact SDSKV providers (also selected by hashing the object's path) to store the corresponding metadata. Symmetrically, reading is done by contacting a relevant SDSKV provider, then a relevant Bake provider.

In terms of development effort, Figure 2b shows that reusable components make up 63% of HEPnOS' code. The larger portion of HEPnOS's custom code is its client-side interface, which provides extensive functionalities to navigate the data store using C++ iterator patterns. The code that actually calls the Mochi components fits in a 276-line file. Our git repositories indicate that less than two months were needed between the creation of the project and the release of a first version that Fermilab could start using. While the server-side composition was ready within two weeks, most the remaining time was spent iterating on new client-side functionalities.

4) Example client code: Listing 2 shows an example of client code that interacts with HEPnOS. The client initializes a `hepnos::DataStore` handle with a configuration file

[8]https://xgitlab.cels.anl.gov/sds/HEPnOS
[9]http://computing.fnal.gov/hep-on-hpc/

indicating how to connect to the HEPnOS providers. The code then exemplifies how to navigate the hierarchy of datasets, runs, subruns, and events, and to store/load custom data objects (here a vector of `Particle` instances) to/from HEPnOS. The last line shows that one can use C++ range-based loops to navigate easily within a run. The same can be done within datasets and subruns.

```
#include <hepnos.hpp>

// example structure
struct Particle {
  float x, y, z; // member variables
  // 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
hepnos::DataStore datastore("config.yaml");
// access a nested dataset
hepnos::DataSet ds = datastore["path/to/dataset"];
// access run 43 in the dataset
hepnos::Run run = ds[43];
// access subrun 56
hepnos::SubRun subrun = run[56];
// access event 25
hepnos::Event ev = subrun[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
// using a C++ range-based for
for(auto& subrun : run) { ... }
```

Listing 2: Example of client code using HEPnOS

C. ParSplice

The Parallel Trajectory Splicing (ParSplice) [20] application uses a novel time-parallelization strategy for accelerated molecular dynamics. The ParSplice technique (and associated application) enables long-time scale molecular dynamics (MD) simulations of complex molecular systems by employing a Markovian chaining approach allowing many independent MD simulations to run concurrently to identify short trajectories called "segments" that are then spliced together to create a trajectory that spans long time scales. A master/worker approach is used to generate segments starting from a set of initial coordinates stored in a key/value database. From these initial coordinates the workers use traditional MD simulation to generate a new segment and upon completion stores the final coordinate of the segment in a distributed key/value database.

During the course of a ParSplice simulation, the key/value database continues to grow to include all the states necessary for workers to generate new trajectories from a prior state. Since workers are distributed across many individual compute nodes and are stateless, the key/value store must provide scalable concurrent access (read/insert). Exascale simulations using ParSplice could span tens of thousands of compute nodes with thousands of database clients accessing the key/value store concurrently. To support this level of concurrency, and to minimize the memory footprint required on any one worker node, we have developed a distributed key/value service, SDSDKV, built on Mochi microservices, as described in Section III.

1) User requirements: The introduction of the SDSDKV service is motivated principally by the potential reduction of code complexity in ParSplice via componentization. Moreover, this organizational strategy allows for easier runtime customization of key/value service behavior (e.g., selecting an appropriate communication protocol, database back-end, or key distribution methodology [21]), thereby improving program performance portability.

The SDSDKV service needs to store values of a few thousand bytes that represent the MD state including positions, velocities, charges, and other particle characteristics. The number of key/value pairs ranges from tens of thousands at current scales to several millions expected at exascale. These key/value pairs are written once and never overwritten, and are accessed atomically (i.e., no partial access to a value is required). The current key/value store does not erase entries, but future expansion of the service may need to remove keys.

2) Service requirements: The service requirements are driven by large runs that will need to distribute the key/value store across multiple nodes to balance out memory use, access latency and bandwidth, and keep the fan-out size from a master worker within a scalable size. The objects will be distributed by their hash keys, obviating the need for metadata. Replication is an option for improving response times. This service can be asynchronous without any guarantees of determinism or handling of race conditions. The service interface needs to be implemented in C with a simple API of create/destroy for service control and put/get/delete for data handling.

3) Implementation with Mochi components: Figure 3c shows the organization of components used in SDSDKV. SDSDKV (∼1,000 SLOC) is based on the SDSKV and SSG components and on ch-placement for consistent hashing. It exposes a small, straightforward C interface providing runtime service configurability through user-supplied input parameters as shown in Listing 3. SDSDKV's use centers on opaque context handles that encapsulate service-maintained state. With this design, multiple, independent SDSDKV instances may exist within a single application, each with potentially different configurations such as membership makeup, database backend type, and communication protocol used. At `sdsdkv_open()`, all members of the initializing communicator (supplied during `sdsdkv_create()`) collectively participate in service startup, initializing the individual components composing SDSDKV. From this point until context destruction `sdsdkv_put()` and `sdsdkv_get()` operations may be performed—routed by ch-placement and ultimately serviced by the appropriate SDSKV provider.

Figure 2c shows the fraction of code that is reused and the fraction that is custom. Custom code include the composition code (7%) and the client interface (4%). The client interface provides a simple, minimalistic put/get interface dispatching

the operations to particular SDSKV providers based on a hash of the keys. The composition code is written in C++ and spins up multiple SDSKV providers, grouped by using SSG and distributed based on the node placement of server processes.

4) Example client code: Listing 3 illustrates the use of the SDSDKV API, which essentially is a simple put/get interface accessing raw data.

```
// Determine personality type from global MPI ID.
sdsdkv_config_personality p = (
  (rank % 2 == 0) ? SDSDKV_PERSONALITY_SERVER
                  : SDSDKV_PERSONALITY_CLIENT
);
// Define an SDSDKV instance configuration.
sdsdkv_config dkv_config = {
  MPI_COMM_WORLD, // Initializing MPI communicator
  p, // Process personality (client or server)
  rpc_thread_count, // RPC threading factor
  SDSDKV_HASHING_CH_PLACEMENT, // Hashing back-end
  SDSDKV_DB_LEVELDB,  // Database back-end type
  SDSDKV_COMPARE_DEFAULT, // K/V compare function
  "groupname", // Group identifier
  db_name, // Base path to database backing stores
  "ofi+tcp", // Communication protocol
};
// Create an SDSDKV instance named dkvc.
sdsdkv_create(&dkvc, &dkv_config);
// Collectively open the dkvc instance.
sdsdkv_open(dkvc);
// Client processes interact with key/value service
// while server processes field put/get requests.
sdsdkv_put(dkvc, &k, sizeof(k), &v, sizeof(v));
...
sdsdkv_get(dkvc, &k, sizeof(k), &v, &v_size);
...
// Collectively destroy SDSDKV instance.
sdsdkv_destroy(dkvc);
```

Listing 3: Example pseudocode using the SDSDKV API.

V. Related Work

This section puts our methodology in the perspective of other works related to parallel file systems, data services, networking libraries, and alternate programming models.

A. Parallel file systems

File system developers have long found the classic POSIX interface and semantics limiting. Researchers have implemented extensions to meet various parallel I/O workloads. PVFS [22] removed POSIX semantics, implementing something closer to MPI-IO semantics. PVFS also provided a user-level library with features such as datatype I/O [23]. Few if any applications used these interfaces directly. These projects, however, have demonstrated the need to move beyond the POSIX interface. Our service-oriented approach means we can quickly implement novel interfaces tailored to specific applications. We no longer need to think about an interface that both extends POSIX and has sufficient application acceptance to be viable.

Object storage has emerged as an alternative interface for storage devices. Some file systems built on top of object stores expose that interface directly to users (e.g., Ceph [24] with RADOS [25]). We, too, have implemented an object storage interface, although its backend implementation may or may not be an object store.

Some approaches have narrowed their focus to one type of access pattern (e.g., checkpoints in PLFS [26], that is, write frequently, maybe read sometimes). Others have explored the use of active storage offload capability to extend file system functionality [27], [28]. The benefits of such special-purpose interfaces directly motivate our service model: separating interface from implementation will allow us to deploy many specialized interfaces all of which utilize a shared core.

B. Specialized data services

Specialized data services are already widespread in scientific computing as a means to augment parallel file system functionality. Examples include services for multidimensional scientific data [29], tuple spaces [30], key/value pairs [31], checkpointing [32], in situ indexing [33], and shared-library management [34]. These may be deployed persistently to meet the workload needs of a particular platform or deployed on demand to meet the needs of a particular application. Such services are often built from the ground up to meet the needs of particular use cases. More recently, Sevilla et al. [35] have explored how to decompose existing parallel file systems into components that can then be reused within programmable data services expressed using Lua. This approach offers rapid access to robust software components from existing production services. Those components also inherit limitations of the original service, however, such as limited support for in-system HPC hardware. Other specialized services such as DataWarp [36] retain a traditional file system data model, but implement it as a transient service that provisions in-system resources on demand to meet the requirements of a specific application.

C. Networking

Communication and message exchange for data services is key to efficient composition. In the context of HPC, the first communication and messaging interface that comes to mind is MPI [37], which can also take advantage of RDMA [38]. As detailed in [39], however, while MPI is most likely the best communication interface for scientific applications, it is not suitable for the development of data service middleware where resiliency and dynamic resource scheduling are critical.

Other solutions such as Protobuf [40] or ZeroMQ [41] are widely used solutions for building distributed services, but they are not adapted to HPC environments and do not support native network fabrics, nor do they provide native RMA semantics to transfer large data.

Existing HPC frameworks have also been developing and integrating similar solutions such as DART [42] or Nessie's NNTI [43] [44] in the context of building either data staging mechanisms or distributed services. In either case, Mercury provides an API that is more adapted to the development of distributed services: by using origin and target semantics; providing a dedicated interface for handling bulk data; and letting high-level components be in charge of dedicated progress and multithreading models.

D. Programming models

Programming models for concurrent distributed data services must strike a balance between runtime performance and developer productivity. Traditional options include multithreading as exemplified by the Apache web server [45], event-driven architectures as exemplified by memcached [46], and state machines as exemplified by the PVFS/OrangeFS file system [47]. Recent works have also explored alternatives to traditional concurrent programming models. Examples include the Seastar futures and promises model [48], the Boost.ASIO proactor model [49], [50], Grand Central Dispatch queues [51], Aesop C concurrent language extensions [52], and Intel TBB tasking templates [53]. Each of these has introduced new primitives that aid in the expression of concurrent code paths, but they also present a nontrivial barrier to entry for rapid development of community services. Conventional multithreading models remain the most widely understood among POSIX developers, can be readily expressed in any language, and are comparatively simple to understand and debug.

User-space threads offer the potential to express concurrent code paths in using straightforward multithreading conventions while retaining the advantages of low latency and high concurrency offered by explicitly asynchronous programming models [54]. A number of lightweight threading or tasking frameworks have been developed and deployed in the HPC community as a result [55]. Notable examples include QThreads [56], MassiveThreads [57], and Converse Threads [58]. Among available user-space threading packages, Argobots is most readily applicable to Mochi because of a combination of three key features: portability across HPC architectures, customizable schedulers that aid in services composition, and low resource for data services colocated with applications [13].

Closer to our work is Faodel [59], which provides a set of libraries to build workflow-oriented data services. Ulmer et al. also propose to interface directly to applications through a high-performance communication library, RDMA, and a key/blob storage service. They demonstrate the effectiveness of their set of libraries in a particle-in-cell simulation workflow with VTK-based analysis. Contrary to our paper, however, they do not focus on the software engineering methodology around the use of their libraries.

VI. Conclusion

We have introduced a methodology for rapidly developing HPC data services tailored to particular applications. By relying on componentization, our methodology proved to enable high code reuse and minimal extra code. This methodology allowed us to quickly develop three data services for very different use cases: deep learning workflows for cancer research, high-energy physics experiments, and molecular dynamics simulations. Contrary to parallel file systems, which are meant to be generic and require layering middleware and libraries to be usable by applications, our data services interface directly with the applications and are modular enough to adapt to the particular applications' requirements.

A number of interesting aspects have not been discussed in this paper. One is authentication/authorization, which is important for data services that are shared between multiple users or remotely accessible from outside the supercomputer where they are deployed. We plan to further investigate how to integrate auth/auth with our methodology and how to concretely implement it within the Mochi framework.

Data ingestion is another important aspect of data services when existing data is stored in a parallel file system or when the data service should still store data to a backend parallel file system (e.g., for persistence across multiple jobs that rely on a transient service). From our experience, however, the data ingestion problem is specific to the use case, and we can hardly generalize practices into a methodology.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. This manuscript is approved by LANL for unlimited release LA-UR-18-29348.

References

[1] E. Zadok, D. Hildebrand, G. Kuenning, and K. A. Smith, "POSIX is dead! long live... errr... what exactly?" in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, 2017.

[2] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. ACM, 2011, pp. 36–47.

[3] R. Rew and G. Davis, "NetCDF: An interface for scientific data access," *IEEE Computer Graphics and Applications*, vol. 10, no. 4, pp. 76–82, 1990.

[4] D. Boyuka, S. Lakshminarasimham, X. Zou, Z. Gong, J. Jenkins, E. Schendel, N. Podhorszki, Q. Liu, S. Klasky, and N. Samatova, "Transparent in situ data transformations in ADIOS," in *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2014, pp. 256–266.

[5] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, and L. Orf, "Damaris: Addressing performance variability in data management for post-petascale simulations," *ACM Transactions on Parallel Computing (TOPC)*, vol. 3, no. 3, p. 15, 2016.

[6] M. Dorier, M. Dreher, T. Peterka, and R. Ross, "CoSS: Proposing a contract-based storage system for HPC," in *Proceedings of the PDSW-DISC 2017 workshop (SC17)*, 2017, workshop. [Online]. Available: https://dl.acm.org/citation.cfm?id=3149396

[7] P. Carns, J. Jenkins, S. Seo, S. Snyder, R. B. Ross, C. D. Cranor, S. Atchley, and T. Hoefler, "Enabling NVM for data-intensive scientific services," in *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*, 2016.

[8] 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 *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1–8.

[9] O. Alliance, "Libfabric OpenFabrics," https://ofiwg.github.io/libfabric/.

[10] P. Carns, W. Ligon, R. Ross, and P. Wyckoff, "BMI: A network abstraction layer for parallel I/O," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE, 2005.

[11] S. Atchley, D. Dillow, G. Shipman, P. Geoffray, J. M. Squyres, G. Bosilca, and R. Minnich, "The Common Communication Interface (CCI)," in *2011 IEEE 19th Annual Symposium on High Performance Interconnects*, Aug. 2011, pp. 51–60.

[12] H. Pritchard, E. Harvey, S.-E. Choi, J. Swaro, and Z. Tiffany, "The GNI provider layer for OFI libfabric," in *Proceedings of Cray User Group Meeting, CUG*, 2016.

[13] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault *et al.*, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2018.

[14] A. Rudoff, "Persistent memory programming," *Login: The Usenix Magazine*, vol. 42, pp. 34–40, 2017.

[15] J. Dean and S. Ghemawat, "Leveldb," http://leveldb.org/.

[16] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley DB," in *USENIX Annual Technical Conference, FREENIX Track*, 1999, pp. 183–191.

[17] A. Das, I. Gupta, and A. Motivala, "Swim: Scalable weakly-consistent infection-style process group membership protocol," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 303–312.

[18] F. Chollet *et al.*, "Keras: Deep learning library for Theano and Tensorflow," *https://keras.io*, vol. 7, no. 8, 2015.

[19] R. Brun and F. Rademakers, "ROOT – An object oriented data analysis framework," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 389, no. 1-2, pp. 81–86, 1997.

[20] D. Perez, E. D. Cubuk, A. Waterland, E. Kaxiras, and A. F. Voter, "Long-time dynamics through parallel trajectory splicing," *Journal of Chemical Theory and Computation*, vol. 12, no. 1, pp. 18–28, 2015.

[21] M. A. Sevilla, C. Maltzahn, P. Alvaro, R. Nasirigerdeh, B. W. Settlemyer, D. Perez, D. Rich, and G. M. Shipman, "Programmable caches with a data management language and policy engine," in *Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGrid'18)*, 2018.

[22] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proceedings of the 4th annual Linux Showcase and Conference*, 2000.

[23] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp, "Efficient structured data access in parallel file systems," in *Proceedings of Cluster 2003*, Hong Kong, Nov. 2003.

[24] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating Systems Design and Implementation*. USENIX Association, 2006, pp. 307–320.

[25] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "RADOS: A scalable, reliable storage service for petabyte-scale storage clusters," in *Proceedings of the 2nd international workshop on Petascale Data Storage: held in conjunction with Supercomputing'07*. ACM, 2007, pp. 35–44.

[26] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*. IEEE, 2009, pp. 1–12.

[27] E. J. Felix, K. Fox, K. Regimbal, and J. Nieplocha, "Active storage processing in a parallel file system," in *Proceedings of the 6th LCI International Conference on Linux Clusters: The HPC Revolution*, 2005.

[28] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W.-K. Liao, and A. Choudhary, "Enabling active storage on parallel I/O software stacks," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–12.

[29] P. Cudré-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla *et al.*, "A demonstration of SciDB: A science-oriented DBMS," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1534–1537, 2009.

[30] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An interaction and coordination framework for coupled simulation workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.

[31] H. Greenberg, J. Bent, and G. Grider, "MDHIM: A parallel key/value framework for HPC." in *HotStorage*, 2015.

[32] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2011, pp. 1–12.

[33] Q. Zheng, G. Amvrosiadis, S. Kadekodi, G. A. Gibson, C. D. Cranor, B. W. Settlemyer, G. Grider, and F. Guo, "Software-defined storage for fast trajectory queries using a deltafs indexed massive directory," in *Proceedings of the 2Nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, ser. PDSW-DISCS '17. New York, NY, USA: ACM, 2017, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/3149393.3149398

[34] W. Frings, D. H. Ahn, M. LeGendre, T. Gamblin, B. R. de Supinski, and F. Wolf, "Massively parallel loading," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 389–398.

[35] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn, "Malacology: A programmable storage system," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 175–190.

[36] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, "Architecture and design of Cray DataWarp," *Cray User Group CUG*, 2016.

[37] T. M. Forum, *MPI: A Message-Passing Interface Standard Version 3.0*, 2012.

[38] W. Gropp and R. Thakur, "Revealing the performance of MPI RMA implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, F. Cappello, T. Herault, and J. Dongarra, Eds. Springer, 2007, vol. 4757, pp. 272–280.

[39] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi, "Using MPI in high-performance computing services," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: ACM, 2013, pp. 43–48. [Online]. Available: http://doi.acm.org/10.1145/2488551.2488556

[40] Google Inc, "Protocol Buffers," 2012. [Online]. Available: https://developers.google.com/protocol-buffers

[41] H. Pieter, "ZeroMQ: messaging for many applications," *O'Reilly Media*, p. 484, 2013.

[42] C. Docan, M. Parashar, and S. Klasky, "Enabling high-speed asynchronous data extraction and transfer using DART," *Concurrency and Computation : Practice and Experience*, vol. 22, no. 9, pp. 1181–1204, 2010.

[43] J. Lofstead, R. Oldfield, T. Kordenbrock, and C. Reiss, "Extending scalability of collective IO through Nessie and staging," in *Proceedings of the Sixth Workshop on Parallel Data Storage*. New York, NY, USA: ACM, 2011, pp. 7–12.

[44] R. A. Oldfield, T. Kordenbrock, and J. Lofstead, "Developing integrated data services for Cray systems with a Gemini interconnect," in *Cray User Group Meeting*, April 2012. [Online]. Available: https://cug.org/proceedings/attendee_program_cug2012/includes/files/pap186.pdf

[45] "Apache MPM worker documentation." [Online]. Available: http://httpd.apache.org/docs/2.0/mod/worker.html

[46] "Multithreading support in memcached." [Online]. Available: http://code.sixapart.com/svn/memcached/trunk/server/doc/threads.txt

[47] "The Parallel Virtual File System." [Online]. Available: http://www.pvfs.org

[48] ScyllaDB, "Seastar." [Online]. Available: https://seastar.io/

[49] C. Kohlhoff, "Asio C++ library." [Online]. Available: https://think-async.com/Asio

[50] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2013, vol. 2.

[51] Apple Inc., "Grand Central Dispatch." [Online]. Available: http://developer.apple.com/technologies/mac/snowleopard/gcd.html

[52] D. Kimpe, P. Carns, K. Harms, J. M. Wozniak, S. Lang, and R. Ross, "AESOP: Expressing concurrency in high-performance system software," in *Proceedings of 7th IEEE International Conference on Networking, Architecture, and Storage (NAS 2012)*, 2012.

[53] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[54] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *HotOS*, 2003.

[55] A. Castelló, A. J. Pena, S. Seo, R. Mayo, P. Balaji, and E. S. Quintana-Ortí, "A review of lightweight thread approaches for high performance computing," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2016, pp. 471–480.

[56] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," 2008.

[57] J. Nakashima and K. Taura, "Massivethreads: A thread library for high productivity languages," in *Concurrent Objects and Beyond*. Springer, 2014, pp. 222–238.

[58] L. V. Kalé, M. Bhandarkar, R. Brunner, and J. Yelon, "Multiparadigm, multilingual interoperability: Experience with converse," in *International Parallel Processing Symposium*. Springer, 1998, pp. 111–122.

[59] C. Ulmer, S. Mukherjee, G. Templet, S. Levy, J. Lofstead, P. Widener, T. Kordenbrock, and M. Lawson, "Faodel: Data management for next-generation application workflows," in *Proceedings of the 9th Workshop on Scientific Cloud Computing*, ser. ScienceCloud'18. New York, NY, USA: ACM, 2018, pp. 8:1–8:6. [Online]. Available: http://doi.acm.org/10.1145/3217880.3217888