A Middleware Supporting Data Movement in Complex and Software-Defined Storage and Memory Architectures

Christopher Haine[1], Utz-Uwe Haus[1], Maxime Martinasso[2], Dirk Pleiter[3,4], François Tessier[6], Domokos Sarmany[5], Simon Smart[5], Tiago Quintino[5], and Adrian Tate[7]

[1] HPE HPC/AI Research Lab, Basel, Switzerland
[2] CSCS, Swiss National Supercomputing Centre, 6900 Lugano, Switzerland
[3] Forschungszentrum Jülich, 52425 Jülich, Germany
[4] KTH, 100 44 Stockholm, Sweden
[5] European Centre for Medium-Range Weather Forecasts (ECMWF), Reading RG2 9AX, United Kingdom
[6] Inria Rennes Bretagne-Atlantique, 35042, Rennes, France
[7] NAG, Oxford, United Kingdom

Abstract. Among the broad variety of challenges that arise from workloads in a converged HPC and Cloud infrastructure, data movement is of paramount importance, especially oncoming exascale systems featuring multiple tiers of memory and storage. While the focus has, for years, been primarily on optimizing computations, the importance of improving data handling on such architectures is now well understood. As optimization techniques can be applied at different stages (operating system, run-time system, programming environment, and so on), a middleware providing a uniform and consistent data awareness becomes necessary. In this paper, we introduce a novel memory- and data-aware middleware called Maestro, designed for data orchestration.

Keywords: HPC and Cloud infrastructures · Software-defined infrastructures · Workflows.

1 Introduction

The multiplicity of emerging memory and storage technologies as well as the evolution towards converged HPC and Cloud architectures requires that we rethink the way data is managed. We have seen for years the development of new types of memory and storage layers on large-scale systems to overcome the data movement bottleneck. Node-local storage, burst buffer nodes [8] or storage-class memory [6], to name a few, are now becoming widely available. Those new tiers come with their own characteristics, whether it is in terms of performance (capacity, bandwidth, latency) or in terms of access (byte or block addressable) [18].

On the other hand, applications and workflows are becoming dominated by data movement [13]. The ever-increasing resolution from scientific simulations as

well as the diversity of workloads (such as Big Data analytics or AI for instance) tend to generate a growing amount of data that has to be properly handled to minimize its impact on performance. For example, the operational weather forecasting workflow at the European Centre for Medium-Range Weather Forecasts (ECMWF) currently generates around 30 TiB per time-critical one-hour forecast four times per day and estimates that this volume of data produced will continue increasing by 40% per year [19]. Data-locality requirements are expected to be a major driver for creating converged infrastructures based on HPC and Cloud technologies, which have to be highly flexible to support different workflows with a variety of requirements, in particular with respect to data handling, on top of a complex hardware architecture (see, e.g., the case of numerical weather prediction [4]).

Therefore, it is useful to develop data movement optimization techniques that can base their decisions on both the way data is accessed (pattern, I/O method) and the underlying hardware (memory and storage hierarchy). The variety in the former poses problems for systems software, and results in an inability to make use of the semantics of data-movement across the entire software stack. This constraint could be addressed with a unified data model. On the hardware side, while it is clear that user-software should not be concerned with non-portable hardware details, higher-level software making data movement decisions cannot do so without some form of locality information being available. An approach is to keep the locality information visible from the user through an abstracted, hierarchical model providing hardware information to any level of the software stack consistently.

In order to address the aforementioned challenges, we introduce Maestro, a memory- and data-aware middleware for data movement orchestration within workflows. Its central element is a pool of resources that each workflow component contributes to. The data, encapsulated in objects along with metadata, is submitted or requested to/from the pool while the data movements are handled and optimized by Maestro. In this document in particular, we present the core data model providing common and consistent access to multiple software layers regardless of the current location in the memory system.

The key contributions of our work are as follows:

- The design of a memory- and data-aware middleware
- A light-weight annotation- and object-based data model for manipulating user data
- An API for easily handling objects within and across applications and workflows

The outline of this paper is the following. In Section 6, we introduce the state of the art memory- and data-aware abstractions for optimized data movement. Then, we introduce in Section 2 the architecture of the Maestro middleware while we detail both our data abstraction and our data management API in Section 3. Section 4 is dedicated to an evaluation of our model based on our implementation of the Maestro middleware. Section 5 puts Maestro in a workflow perspective. We present early results obtained with a component of a major weather forecasting

workflow. Finally, we conclude this paper while putting the emphasis on the limitations of our approach and our future work to resolve them.

2 Architecture

The Maestro middleware is built around the idea that applications should be empowered to delegate the access and movement of the data they provide and/or require to a smart middleware. This middleware should then reason about the system characteristics, data-movement cost, and workflow-level scheduling of data placement. At the same time, data should not be required to be allocated in Maestro-defined data structures. Instead, low-overhead annotation of existing data should be sufficient to inform the middleware and permit it to handle such application-defined and -managed data.

The Maestro middleware can be understood to provide its features at three different levels:

- As a data management layer for all memory tiers of a system inside a single process, e.g. across multiple threads, to use the abstraction of *core data objects* (CDOs, detailed in Section 3) to better structure data exchanges between program parts, across devices (e.g., GPUs), and to use convenient data transformations provided by the library.
- As a data management layer for all memory tiers of a system across multiple execution domains (compute nodes, processes, workload manager jobs, and/or allocations), including the coupling of applications by their CDO dependencies.
- As an enabler of a workflow-management solution, which enables the workflow manager to observe and influence data availability, demand, locality, and transfer without the applications knowing about the workflow they are embedded within.

To support these features, Maestro is based on an architecture centred around a CDO pool to (from) which CDOs can be offered (requested). CDOs are managed by the pool on basis of a system model that allows assessing the availability of storage resources or the costs of data movement as well as data transformations. The Maestro APIs are defined such that seamless access to various memory layers is supported.

This architecture supports the following usage patterns:

- Data objects (CDOs) are declared using a workflow-level unique name.
- Attributes (Maestro-specific or user-specific) can be added to each CDO (e.g. concerning lifetime).
- Data objects are offered to other participants of a workflow or requested from other participants via a conceptual pool.
- Participants eventually withdraw the object they contributed to the pool
- During the time an object is pooled, and only in this phase, Maestro takes full control over it; it may move, re-layout, redistribute, or copy the data as it sees fit across the entire union of resources available to the workflow, including the resources contributed by participants in the form of CDOs.

Maestro being a middleware, its use will be triggered by different means. The design is catering to these in a consistent way, so that the different usage scenarios will be able to interact seamlessly.

Application-level usage of Maestro to simplify data management – In this case a user application, typically a scientific application, will directly use `libmaestro` as a library, from C/C++/FORTRAN or a scripting language to take advantage of the memory management facilities and the feature to offer application data to unknown consumers, or request data from unknown sources. Other applications of this use case are compilation tools, compilers implementing advanced data layout or movement transformations, the implementation of tasking frameworks, or programming environments like OpenMP, MPI, or UPC.

Workflow-controlled usage of Maestro as a coupling tool – In this case a workflow description language is used to coordinate the execution of multiple applications, including resource provisioning. Translation to an execution schedule may occur statically, or dynamically with the workflow-manager or dedicated watcher components observing data-object creation and requests. Using object attributes, it is possible for the workflow manager to steer the behaviour of the Maestro middleware; using telemetry information, it is possible to implement feedback profiling or online re-scheduling or re-resourcing.
This is the usage scenario we believe will be most prevalent for end users: minimal application changes to annotate the core data objects required for coupling applications will be combined with a powerful workflow description, and execution strategies will be controlled and tuned at the workflow manager level, with the middleware in the position to minimize data transfer cost transparently.

On the core middleware side, automatic multi-application rendezvous is implemented with the help of a dedicated pool manager component using `libfabric`[8], and using a `protobuf`-based protocol[9], avoiding user selection of network interfaces, or administrative permissions/daemons.

The pool client component, which is implemented as an in-process Maestro core instance, is the interface between the application and the pool manager, which handles API calls that translate into pool protocol, that includes the CDO management API, and the subsequent network communication towards the pool manager. A four-step protocol, detailed in Section 3, specifies the way that an application and Maestro share ownership of data objects. Broadly speaking, "Give-Take" semantics describe applications giving data objects to Maestro pool and taking data objects back from Maestro pool. In case of "Take" on a CDO, be it in a multithreaded context or not, the first check the Maestro middleware performs is to look-up the presence of the "Taken" CDO in the local pool – the set of resources tied to the process. If it is indeed present, the "Take" can be satisfied promptly without wasting pool manager time. This also allows for a single application Maestro usage, without the need to launch a dedicated pool manager application.

[8] `http://libfabric.org/`
[9] `https://developers.google.com/protocol-buffers`

CDO transport is decoupled from the pool operations, which means CDO "Take" resolution and layout metadata transfer are decoupled from the actual CDO content transfer. It makes the pool operations fast, and independent of CDO storage resource handling, transport methods, and layout transformations.

3 Core Middleware API

A core data object (CDO) combines all available information from both the hardware/storage side and the software/semantic side. As the most complete understanding that the middleware can obtain about a particular data object, the CDO is how applications communicate intentions with Maestro. The CDO typically represents real data and their physical location (if known).

CDOs possess two binary states that dictate how Maestro can interact with the CDO

1. `ACCESSIBLE` defines whether the contents of a CDO can be accessed via Maestro accessor functions (e.g. elemental/tile set/get).
2. `POOLED` defines whether a CDO has been given to the Maestro management pool or not.

These states are used to indicate whether it is the application or Maestro that is in control of the object. When a CDO is `POOLED`, Maestro may move, copy or transform the CDO. When a CDO is `ACCESSIBLE`, then its content and structure can be queried by an application, and a set of accessor methods allows setting and retrieval of the data. If an application creates a `POOLED` CDO, the application is relinquishing control of the CDO to the middleware.

CDOs encapsulate additional CDO information or metadata – otherwise referred to as attributes in Maestro core language. We will refer to the attributes proposed by default by Maestro core as *core attributes*, as opposed to *user-defined attributes*, which Maestro core supports via schemata, and typically correspond to domain-specific key-value metadata.

A CDO contains optional metadata related to its usage context, such as access relations and relations to other CDOs. Layout attributes are part of the core attributes as well, and allow users to add data semantics to Maestro – either manually or via a static-analysis tool that can infer these attributes and automatically inject them into the code – in order to take advantage of its automatic transformations.

An extra API is needed to allow users to add their specific metadata. In order to incorporate user-defined attributes within CDOs, Maestro core expects the user to provide a YAML schema that user-defined attribute operations will have to be compliant with, essentially consisting of a key-value list of possibly optional keys.

A four-step protocol specifies the way that an application and Maestro share ownership of data objects. Broadly speaking, "Give-Take" semantics describe applications giving data objects to Maestro pool and taking data objects back from Maestro pool. To accommodate the necessary differentiation between data

producer and data consumer use cases, "give" and "take" have bilateral counterparts. The overall give-take API is composed of `DECLARE`, `OFFER/REQUIRE`, `WITHDRAW/DEMAND/RETRACT`, and `DISPOSE`. They represent four distinct steps in the lifetime of a CDO as detailed below, namely:

1. CDO declaration,
2. CDO pool injection,
3. CDO pool retraction,
4. CDO disposal.

In all of the operations, CDOs are referred to on the application side by an application-defined name at `DECLARE` time, and then referenced by a *handle* that corresponds to a (Maestro-internal) object identifier (UUID). Declaration is done through `DECLARE` and allows an application to describe a CDO to Maestro. The application then obtains a handle that will be needed for any future communication about this object. Maestro can, depending on the object attributes used in the declaration, allocate resources, prepare and plan for necessary transfers, schedule other workflow components, etc.

Pool injection is done through `OFFER` as a producer and `REQUIRE` as a consumer, the two GIVE variants that take CDO definitions from Maestro to the CDO Management Pool. After this operation the CDO content may not be accessed by the application. It is entirely up to Maestro where the CDO content resides or even whether its content is consistent with any previous or future state. Maestro can, depending on the object attributes used in the declaration, allocate resources, prepare and plan for necessary transfers, schedule other workflow components, etc. It can even use the CDO's storage allocation for other purposes.

Pool retraction is conveyed by `WITHDRAW` on the producer side, and `DEMAND/RETRACT` on the consumer side. "Take" is the moment at which CDO ownership is transferred (back) to the application. This operation is blocking: Maestro may choose to delay all operations until this point (permitting lazy CDO handling semantics).

CDO disposal is done through `DISPOSE`, which is the inverse of the `DECLARE` operation. It indicates the end of the CDO's lifetime. If resources were provided by Maestro for the CDO they may be deallocated.

As part of Maestro's metadata support, it is also possible for a client to inspect the state and properties of a CDO, such as its attributes or whether it has been `DECLARED`ed or `OFFER`ed to the pool. It also allows for user applications to retrieve CDOs based on their attributes.

4 Performance Evaluation

One of many applications that may greatly benefit from the Maestro middleware is numerical weather forecasting. The transfer between the data producers of a time-critical forecast run and the post-processing consumers is the bottleneck in many global numerical weather forecasting workloads, including the one run by ECMWF. Its global numerical model, the Integrated Forecast System (IFS),

outputs its forecast data via a domain-specific I/O library, called `multio`[17], which is responsible for routing the data to multiple datasinks, such as the domain-specific object store FDB [19].

We have created a backend to the `multio` library that supports data output to the Maestro core middleware. To simulate the output process without having to run the actual forecast model, we have also built a tool, called `multio-hammer` on top of the `multio` library. It takes sample data as input and permutes it through the range of metadata required for a forecast run, thus outputting dummy data with various sets of metadata. This data is output through the full I/O stack used in normal operations. In this way it simulates the forecast model from the I/O perspective.

We aim to evaluate the metadata performance of the Maestro management API for producer-consumer applications and its support for domain-specific attributes. For this purpose, we have set up a test experiment that communicates all necessary metadata without initiating any data transfer. The experiment consists of: *a)* a user-defined YAML schema of weather-forecasting-specific attributes; *b)* a pool manager running on one node; *c)* a single consumer application running on another node, polling for `OFFER`ed CDOs; and *d)* various numbers of `multio-hammer` instances, each running on a separate node and executing three forecast steps.

Each forecast step creates 22236 CDOs with unique metadata. We measure, for each step, the time it takes to create and inject the 22236 CDOs and the time it takes for the consumer application to inspect all CDOs for possible retrieval.

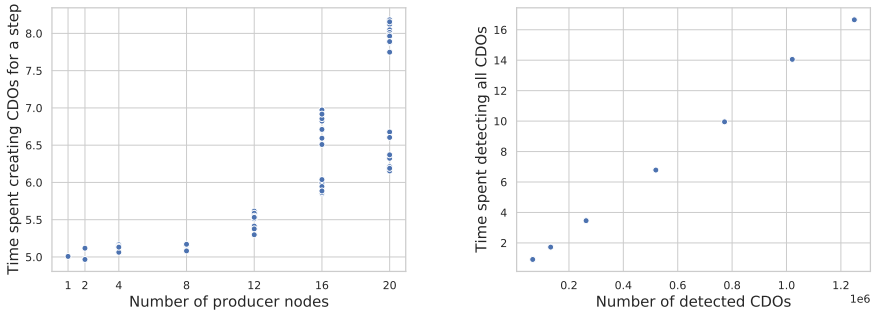
We have run these tests on a 60-node Atos Linux cluster with Infiniband interconnects and Slurm submission system. The machine is the prototype and test machine for the next set of HPC systems that will be used at ECMWF for both operations and research.

This setup primarily aims to verify part of the Maestro management API that interfaces with the data producers, although it also uses Maestro's metadata introspection to verify that communication between the producer and consumer based on metadata can be established via the pool manager. It measures the overhead of metadata operations and the interaction with the Maestro middleware API. For every CDO, a producer makes a CDO declaration, sets the core attributes, sets all the user-defined attributes, seals the CDO declaration, offers the CDO to the management pool and, finally, disposes of the CDO.

Figure 1a shows for a single time step the time in seconds to create all 22236 CDOs. The steps are grouped according to the number of different nodes on which an instance of the producer task `multio-hammer` is running.

In terms of the scaling of metadata operations, the time for the CDO creation per step goes from around 5s on single node to around 5-8s on 20 nodes. In other words, it takes at most 60% longer for the pool manager to process 20 times more data. These numbers roughly corresponds to an overhead of a few percent on top of the generation of mock data, which in turn is much faster than the generation of actual data.

Figure 1b shows the amounts of times it takes to detect (inspect) all CDOs produced (OFFERed) by the different number of producers. The measurements include the acknowledgements sent back to the pool manager that the event has been handled. The plot indicates linear relation in the range of 1–20 nodes, as expected. The results from the 20-node setup translate to creating more than 50000 CDOs per second.



(a) (b)

Fig. 1: Time in seconds to complete a single time step, creating 22236 CDOs per node (left), and to detect (inspect) all CDOs produced in three time steps (right)

5 Dynamic Provisioning and Workflow Support

To facilitate on-demand availability of storage resources to Maestro-enabled workflows, one tool has been developed for software-defined provisioning of storage resources. The concept of on-demand storage resources enables a workflow to deploy a specific data manager or parallel file system into the targeted infrastructure. To provide higher degree of performance, the selected data manager offers a specific feature , such as caching, that increases I/O throughput of the workflow. Moreover, the data manager is deployed on near-compute storage resources.

In a first proof-of-concept, it has been demonstrated how a parallel file system, e.g. BeeGFS, can be deployed on top of near-compute storage nodes, e.g. HPE/Cray DataWarp nodes, connected on the same network layer as the compute nodes [21]. Later, other data managers have been added to demonstrate the capability of the dynamic provisioning tool. Support of object stores using MinIO and databases using Cassandra have been added. The design of the dynamic provisioning enables an easy integration of other types of data managers by the usage of containers.

In the next proof-of-concept, we integrated the use of Maestro middleware and dynamically provided storage resources for workflows based on Pegasus. The

on-demand data manager is deployed as a job by the batch scheduler installed on the infrastructure. As Pegasus is using a batch scheduler to execute workflow tasks, the dynamic provisioning is simply integrated as a new task in a Pegasus workflow. To create such task, a tool has been developed to semi-automatically augment workflow descriptions in Pegasus format with information and tasks that take care of dynamic resource provisioning and start-up of the Maestro middleware.

6 Related work

Data access relies on two overlapping research topics: data model and API.

New data models have been proposed over the years as architectures have evolved [23, 15]. HDF5 [5], for instance, provides namespace-like characteristics to parts of user data, but with an assumption that the data will be stored in a (potentially virtual) filesystem. With a similar idea but focusing on in-memory storage, Conduit [3] provides a hierarchical scheme for relating program data structures to the contents of DRAM, with no means to extend the hierarchy into the fileystem or to split between, for example, flash and DRAM. Another related project is ADIOS [12]. ADIOS is an abstraction layer that allows the user to annotate their I/O operations using an API, but through which a multitude of real I/O libraries, servers and transport layers can be activated. Finally, NetCDF [9] proposes an abstract data type in the form of arrays and an API to manipulate the data-structure. Maestro differs from those approaches in that Maestro is not only focused on I/O but data movement at all levels, I/O in the application, I/O in the workflow and data movement in the memory system.

Another area relevant to data-models is data stagers. One example is the DataStager [2] framework, which comprises a client library which provides the basic methods required for creating data objects and transporting them. At the crossroads of data models and data stagers, we find research works such as LABIOS [11]. LABIOS is a distributed I/O system enabling transparent asynchronous I/O on heterogeneous and elastic storage resources. Data is abstracted in a *label*, a structure made of a pointer to the data, pointers to functions for data transformation and metadata. Very similar concepts are realised in Maestro. However, our approach is more general and not limited to asynchronous data objects in the context of I/O. Another interesting work in that domain, by Tang et al. [20], proposes to encapsulate scientific data in an object-oriented manner.

ADIOS2 [7] and XIOS [14] also share some goals with Maestro. ADIOS2, the Adaptable Input Output System, focuses on asynchronous coupling of applications by user-defined variables, while XIOS is primarily used as an (XML) I/O-server build on top of the existing NetCDF data model for weather and climate simulations. Maestro, by contrast, puts major significance on extensive user-defined attribute handling, programmatic observability of availability of data objects, and awareness and handling of the memory hierarchy. Applications do not need to know how to communicate and transport to other applications in

Maestro, and synchronization can be achieved by data objects. This permits easier composition of Maestro-enabled applications and workflows, including dynamically changing numbers of producers and consumers. Implementing an ADIOS2 Engine based on Maestro will be the topic of a future study. Similarly, it is conceptually possible to implement data transport in XIOS using Maestro instead of MPI.

Unity [10] also shares philosophical intentions with Maestro, being an attempt to unify the memory and storage spaces. Its focus is on data-intensive processing and mixed workloads and the close integration of HPC with data analytics. At a lower level, Perarneau et al. [16] and Unat et al. [22] have been working on low-level abstraction of data layout in memory. Work on SharP also brings a low-level abstraction layer for data management [24].

The Maestro design uses similar concepts to some of these projects – it uses HDF5-like namespacing, it abstracts I/O like ADIOS, it provides a unified memory/storage scheme like Unity and it also provides common data models to different components. However, while many of the mentioned projects are an explicit API, Maestro differs from each of the data models in that it is a middleware layer designed to be used in a variety of environments. It provides an API but can be used in other ways and users can avoid using any explicit calls to the Maestro library if preferred. Maestro therefore has the potential to provide better legacy support than most of these libraries, but also serve as a powerful foundation for new software and frameworks.

7 Conclusion

Taking into account the movements of data on current and future architectures is crucial. The increasing amount of data generated by scientific applications and workflows is concomitant with a relative decline in I/O performance on supercomputers. However, the HPC software stack was not designed with this in mind.

In this paper, we present the data model and API at the base of the Maestro middleware whose goal is to orchestrate data movement. In particular, we detail a model based on the encapsulation of data and metadata into objects as well as its data-access semantics. The preliminary results are encouraging and confirm our approach. A test experiment based on the production of weather-forecast data has shown that the Maestro middleware is able to handle the injection of more than 50000 CDOs/s (together with their domain-specific metadata) on a 20-node setup. This represents an overhead of just a few percent on top of the generation of mock data, which we deem on track to meet the future objective of scaling to an operational configuration.

For the metadata communication presented in this paper, libfabric high-speed interconnect has already been used. Work is ongoing to implement a Maestro I/O (MIO) interface as an abstraction layer to different object store technologies. It initially focuses on Cortx [1], later it will be extended to Ceph using the RADOS [25] service.

References

1. CORTX object store. https://github.com/Seagate/cortx
2. Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., Zheng, F.: Datastager: Scalable data staging services for petascale applications. Cluster Computing **13**, 277–290 (06 2009). https://doi.org/10.1007/s10586-010-0135-6
3. Aspesi, G., Bai, J., Deese, R., Shin, L.: Havery mudd 2014-2015 computer science conduit clinic final report (5 2015). https://doi.org/10.2172/1184132, https://www.osti.gov/biblio/1184132-havery-mudd-computer-science-conduit-clinic-final-report
4. Bauer, P., Dueben, P.D., Hoefler, T., Quintino, T., Schulthess, T.C., Wedi, N.P.: The digital revolution of earth-system science. Nature Computational Science **1**(2), 104–113 (Feb 2021). https://doi.org/10.1038/s43588-021-00023-0, https://doi.org/10.1038/s43588-021-00023-0
5. Folk, M., Heber, G., Koziol, Q., Pourmal, E., Robinson, D.: An overview of the hdf5 technology suite and its applications. In: Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases. p. 36–47. AD '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1966895.1966900, https://doi.org/10.1145/1966895.1966900
6. Freitas, R.F., Wilcke, W.W.: Storage-class memory: The next storage system technology. IBM Journal of Research and Development **52**(4.5), 439–447 (2008)
7. Godoy, W.F., Podhorszki, N., Wang, R., Atkins, C., Eisenhauer, G., Gu, J., Davis, P., Choi, J., Germaschewski, K., Huck, K., Huebl, A., Kim, M., Kress, J., Kurc, T., Liu, Q., Logan, J., Mehta, K., Ostrouchov, G., Parashar, M., Poeschel, F., Pugmire, D., Suchyta, E., Takahashi, K., Thompson, N., Tsutsumi, S., Wan, L., Wolf, M., Wu, K., Klasky, S.: Adios 2: The adaptable input output system. a framework for high-performance data management. SoftwareX **12**, 100561 (2020). https://doi.org/https://doi.org/10.1016/j.softx.2020.100561, http://www.sciencedirect.com/science/article/pii/S2352711019302560
8. Henseler, D., Landsteiner, B., Petesch, D., Wright, C., Wright, N.J.: Architecture and design of Cray DataWarp. In: Proceedings of 2016 Cray User Group (CUG) Meeting (2016)
9. Jianwei Li, Wei-keng Liao, Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., Zingale, M.: Parallel netCDF: A High-Performance Scientific I/O Interface. In: SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing. pp. 39–39 (2003)
10. Jones, T., Brim, M.J., Vallee, G., Mayer, B., Welch, A., Li, T., Lang, M., Ionkov, L., Otstott, D., Gavrilovska, A., et al.: Unity: unified memory and file space. In: Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017. pp. 1–8 (2017)
11. Kougkas, A., Devarajan, H., Lofstead, J., Sun, X.H.: LABIOS: A Distributed Label-Based I/O System. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing. p. 13–24. HPDC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3307681.3325405, https://doi.org/10.1145/3307681.3325405
12. Liu, Q., Logan, J., Tian, Y., Abbasi, H., Podhorszki, N., Choi, J.Y., Klasky, S., Tchoua, R., Lofstead, J., Oldfield, R., Parashar, M., Samatova, N., Schwan, K., Shoshani, A., Wolf, M., Wu, K., Yu, W.: Hello ADIOS: The Challenges and Lessons of Developing Leadership Class I/O Framework. Concur-

rency and Computation: Practice and Experience **26**(7), 1453–1473 (May 2014). https://doi.org/10.1002/cpe.3125, https://doi.org/10.1002/cpe.3125
13. Luu, H., Winslett, M., Gropp, W., Ross, R., Carns, P., Harms, K., Prabhat, M., Byna, S., Yao, Y.: A Multiplatform Study of I/O Behavior on Petascale Supercomputers. In: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. p. 33–44. HPDC '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2749246.2749269, https://doi.org/10.1145/2749246.2749269
14. Meurdesoif, Y.: XIOS current developments and roadmap. https://forge.ipsl.jussieu.fr/ioserver/raw-attachment/wiki/WikiStart/XIOS-ROADMAP-15102020.pdf (2020)
15. Otstott, D., Zhao, M., Williams, S., Ionkov, L., Lang, M.: A foundation for automated placement of data. In: 2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW). pp. 50–59 (2019)
16. Perarnau, S., Videau, B., Denoyelle, N., Monna, F., Iskra, K., Beckman, P.: Explicit data layout management for autotuning exploration on complex memory topologies. In: 2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC). pp. 58–63 (2019)
17. Quintino, T., Smart, S., Sarmany, D.: MultIO – a multiplexing I/O library. https://github.com/ecmwf/multio
18. Ross, R., Ward, L., Carns, P., Grider, G., Klasky, S., Koziol, Q., Lockwood, G.K., Mohror, K., Settlemyer, B., Wolf, M.: Storage Systems and I/O: Organizing, Storing, and Accessing Data for Scientific Discovery. Report for the DOE ASCR Workshop on Storage Systems and I/O (9 2018). https://doi.org/10.2172/1491994
19. Smart, S., Quintino, T., Raoult, B.: A high-performance distributed object-store for exascale numerical weather prediction and climate. In: Proceedings of the Platform for Advanced Scientific Computing Conference. pp. 1–11 (2019)
20. Tang, H., Byna, S., Tessier, F., Wang, T., Dong, B., Mu, J., Koziol, Q., Soumagne, J., Vishwanath, V., Liu, J., Warren, R.: Toward scalable and asynchronous object-centric data management for hpc. In: 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). pp. 113–122 (2018)
21. Tessier, F., Martinasso, M., Chesi, M., Klein, M., Gila, M.: Dynamic provisioning of storage resources: A case study with burst buffers. In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 1027–1035 (2020). https://doi.org/10.1109/IPDPSW50202.2020.00173
22. Unat, D., Nguyen, T., Zhang, W., Farooqi, M.N., Bastem, B., Michelogiannakis, G., Almgren, A., Shalf, J.: Tida: High-level programming abstractions for data locality management. In: Kunkel, J.M., Balaji, P., Dongarra, J. (eds.) High Performance Computing. pp. 116–135. Springer International Publishing, Cham (2016)
23. Unat, D., Shalf, J., Hoefler, T., Schulthess, T., (Editors), A.D., Besta, M., , et al.: Programming Abstractions for Data Locality. Tech. rep. (04 2014)
24. Venkata, M.G., Aderholdt, F., Parchman, Z.: Sharp: Towards programming extreme-scale systems with hierarchical heterogeneous memory. In: 2017 46th International Conference on Parallel Processing Workshops (ICPPW). pp. 145–154 (2017)
25. Weil, S.A., Leung, A.W., Brandt, S.A., Maltzahn, C.: 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. pp. 35–44 (2007)