Object-Centric Data Management in HPC Workflows - A Case Study

Chen Wang
Lawrence Livermore National Laboratory
Livermore, USA
0000-0001-9297-0415

Houjun Tang
Lawrence Berkeley National Laboratory
Berkeley, USA
0000-0001-7038-8360

Jean Luca Bez
Lawrence Berkeley National Laboratory
Berkeley, USA
0000-0002-3915-1135

Suren Byna
The Ohio State University
Columbus, USA
0000-0003-3048-3448

***Abstract*—HPC workflows consist of multiple phases and components executed collaboratively to reach the same goal. They perform necessary computations and exchange data, often through system-wide POSIX-compliant parallel file systems. However, POSIX file systems pose challenges in performance and scalability, prompting the development of alternative storage systems like object stores. Despite their potential, object stores face adoption barriers in HPC workflows due to their lack of workflow awareness and the structured nature of HPC data. This work presents a case study using the Proactive Data Containers (PDC), a framework focusing on object-centric runtime data management, to support a real-world astronomy workflow that runs on HPC systems, called Montage. Due to its user-space deployment feature, PDC is flexible to be adopted transparently with existing I/O libraries. This study explores the use of PDC with Montage's existing FITS-based I/O methods and discusses workflow-oriented optimizations such as caching, prefetching, and write aggregation, and provides insights and lessons learned throughout the porting process.**

***Index Terms*—Parallel I/O, Object Stores, HPC Workflow**

I. Introduction

High-performance computing (HPC) workflows are composed of multiple phases executed in coordination to achieve a common objective. These phases may execute sequentially or in parallel, depending on control and data dependencies. Within each phase, multiple tasks may concurrently execute to perform necessary computations. When running on an HPC system, these tasks frequently exchange data within and across phases, typically utilizing parallel file systems. Most HPC workflows need a POSIX-compliant [1] file system for data exchange. Tasks within the workflow can either execute direct POSIX calls or use high-level I/O libraries such as HDF5 [2] and ADIOS [3].

POSIX file systems have long been criticized for hindering HPC I/O performance and scalability [4]–[6]. Over the years, several alternative storage systems have emerged [7]–[9]. Among these, object stores have become increasingly popular due to their scalability and flexibility in managing large datasets. These characteristics present significant opportunities for improving the performance of I/O-intensive workflows. However, general-purpose object stores may prove unsatisfactory for HPC workflows as they lack awareness of workflow structures. They tend to treat different phases of a workflow as unrelated applications, hindering I/O optimizations such as caching and prefetching. Consequently, it is challenging for users to convey information about their workflow to the underlying system. Additionally, running a workflow across multiple sites, each with a different storage system, complicates data transfer. For instance, a user might execute computation-intensive phases on an HPC system while performing analysis-oriented phases on a cloud system that offers better AI support. In such cases, users often must manually transfer large volumes of data between the HPC system and the cloud, placing a significant burden on them and consuming valuable time.

To address these challenges, we have proposed and developed PDC (Proactive Data Container) [8], an object-centric runtime data management system. In contrast to object-based file systems such as DAOS [9] and storage layer-focused file systems such as UnifyFS [7], PDC is designed to work with different types of underlying hardware and file systems, leveraging new storage techniques while abstracting away the increasingly complex storage hierarchies (storage-class memory, NVRAM, traditional disks, and remote storage). Nevertheless, the structured nature of data in HPC workflows and the potentially substantial effort required to utilize data stores often make users hesitant to switch. Specifically, porting existing POSIX-dependent applications and workflows to a new non-POSIX object-based system is challenging. In this work, we examine this issue by exploring the use of PDC to support a real-world astronomy workflow, Montage [10], which uses CFITSIO [11] for storing and reading data.

In the rest of the paper, we briefly describe the Montage workflow, discuss PDC's object-centric non-POSIX interface, and detail the efforts required to run Montage using PDC. Additionally, we explore potential optimizations for workflow-aware systems. Finally, we share insights and lessons learned from the process of porting Montage to PDC.

II. Montage Workflow

Montage [10] is a versatile toolkit designed for the creation of custom science-grade astronomical image mosaics by assembling FITS images. Users specify the desired mosaic through a set of parameters encompassing dataset selection, wavelength designation, sky location, and size, coordinate

system, projection method, and spatial sampling rate. A typical execution of a Montage workflow comprises multiple phases, including image projection, background correction, and image coaddition. These phases involve significant data exchange, generating and consuming a considerable number of image files. The high I/O demands within each phase, coupled with extensive and small amounts of data exchange across phases, make the Montage workflow an ideal as well as challenging subject for our case study.

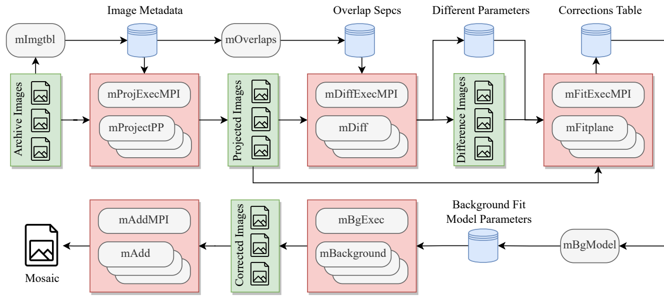

Fig. 1: An example Montage workflow. Rounded rectangles represent components in Montage. The arrows indicate I/O between application memory and file systems.

An example Montage workflow is illustrated in Fig. 1. Rounded rectangles, all beginning with an "`m`", represent components within Montage. Components labeled with an "MPI" suffix (e.g., `mProjExecMPI`) are MPI programs intended for execution in a distributed environment. Other components are serial and executable using a single process on a single node. Unlike traditional HPC applications, the MPI components in Montage do not directly engage in computation. Instead, they operate in an embarrassingly parallel manner: each MPI process spawns a child process responsible for executing the actual serialized computation component. For instance, `mProjExecMPI` may execute N instances of `mProjectPP`, where N denotes the number of MPI processes. The parent MPI program solely manages the collection of results from child processes and their subsequent reporting.

The discussion of the detailed functionality of each component is beyond the scope of this paper. We refer readers to papers [10], [12] for more details. Here, we mainly focus on their I/O behaviors. In Fig. 1, each arrow denotes a directional data flow involving either the reading or writing of single or batches of files. When executed on a file system, Montage generates and consumes two primary types of files: image files and metadata files. Metadata files, depicted in database shapes, encapsulate metadata information pertaining to the image files. These metadata files are small in size and require minimal time for reading or writing. In contrast, the bulk of the I/O operations are dedicated to accessing image files, as indicated by the green boxes in Fig. 1. With only a few exceptions, all image files involved in the execution of the Montage workflow are in the Flexible Image Transport System (FITS) format. FITS is an open standard archival data format for astronomical data sets and it is currently the most commonly used digital file format in astronomy. The majority of these FITS files are intermediate files that can be safely deleted after the execution of the workflow. Although the number of files produced may vary from phase to phase, it typically ranges from one to five times the number of input files. In most scenarios, the number of files significantly exceeds the number of processes, ensuring each process has ample work to perform.

Montage workflows can be executed phase by phase using job schedulers or dedicated workflow managers such as Pegasus [13]. In this study, we focus on scheduling Montage workflows using the system-wide job scheduler. Incorporating PDC in workflow managers is a promising direction, which we leave to our future work.

III. Proactive Data Containers (PDC)

Proactive Data Containers (PDC) is an object-centric metadata and data management system designed for transparent, asynchronous, and autonomous data movement, taking advantage of the memory/storage hierarchy (i.e., main memory, NVRAM, disks). Moving away from file-oriented, PDC relies on three abstractions: *containers*, *objects*, and *regions*. A container is a meta-object that stores a collection of objects and provides a convenient way to access a group of data and metadata objects. An *object*, which describes any byte stream of information, can belong to multiple containers.

In scientific applications, a data object stores vast amounts of information, typically in multi-dimensional arrays (e.g., a 2D image, variables in a 3D grid, or animations). A metadata object stores data associated with the data object, including the object's name, time of data generation, ownership, and relations to other objects. Metadata objects can have a large set of attributes used for identifying or enhancing data object information. Further, objects that are multi-dimensional arrays can be partitioned into smaller *regions* to enable parallel processing and flexible data placement. When applications map their data objects to PDC objects using PDC's object-focused interface and initiate transfers, PDC runtime moves them to storage layers asynchronously and transparently.

A. Object-focused Interface

Applications express their data access intent using PDC's object-focused APIs, and the PDC runtime system performs scalable metadata operations to locate data objects and asynchronously move the data across the memory and storage hierarchy.

To be specific, HPC applications interact with PDC using the following APIs. `PDCinit` initializes the client-side PDC library and connects to the PDC servers. `PDCprop_create` creates container and object properties, `PDCprop_set_*` set the property values such as object dimensions and data type.

`PDCcont_create` and `PDCobj_create` create containers and objects, respectively. Key-value tags can be added, retrieved, queried, and deleted with `PDCobj_put_tag`, `PDCobj_get_tag`, `PDCquery_tag`, and `PDCdel_tag`.

Regarding actual I/O functionalities, PDC uses the *region* abstraction for data management. For each object data

transfer, applications need to define a *local* and a *global* region using `PDCregion_create`. Both regions include information such as offsets and sizes of a multi-dimensional array (with units depending on the object data type). The local region represents the local data buffer (client-side), while the global region represents the global object space (server-side). The data transfer process is broken down into three calls to achieve asynchronous I/O and provide more flexibility. First, `PDCregion_transfer_create` takes the local and global regions as input and creates a *transfer request*. Then, the data transfer is started asynchronously using `PDCregion_transfer_start`, which takes the previously created transfer request as input. This call allows applications to perform other tasks, such as computation and communication, while the data is being transferred to the PDC servers. The PDC servers may decide to cache the data in the memory or flush them to the storage system. Lastly, the application can explicitly wait for the transfer to complete using `PDCregion_transfer_wait`. This call guarantees that the update associated with the transfer request becomes visible to all subsequent transfers.

IV. Montage with PDC

Here, we conduct a case study on running Montage using PDC. This case study offers insights into how a storage system can be tailored to accommodate workflow requirements, the benefits of proactive data management compared to general-purpose storage systems, and the available optimizations for I/O-intensive workflows.

As previously noted, a significant portion of Montage's I/O time is dedicated to accessing FITS files. Therefore, our study primarily focuses on using PDC to facilitate the I/O access of these FITS files. In a Montage workflow execution, the I/O operations on these FITS files are executed using the CFITSIO library [11]. CFITSIO is a library of C and Fortran subroutines designed for reading and writing data files in FITS format. CFITSIO operates as a serial library, meaning there are no concurrent accesses to a single FITS file within a process. The MPI components in Montage work in an embarrassingly parallel manner, with each rank handling their assigned files.

A. CFITSIO PDC Driver

CFITSIO provides multiple drivers for accessing FITS files in various locations. For instance, FITS files can be stored on remote servers and accessed using NFS or FTP protocols. Additionally, CFITSIO offers a mechanism for implementing custom drivers to support different storage backends. Leveraging this feature, we developed a driver for storing and accessing FITS files on PDC servers. Since nearly all accesses to FITS files in Montage are routed through CFITSIO, integrating the CFITSIO PDC driver enables seamless execution of Montage on PDC without any modification to its I/O logic. Furthermore, as CFITSIO is widely used in astronomy applications requiring access to FITS files, the PDC driver is reusable for supporting these applications as well.

The implementation of the PDC driver involves defining a set of CFITSIO interfaces. The complete set can be found in [11]. Here, we discuss the most essential APIs, as listed below. In CFITSIO, each FITS file is treated as a stream of bytes similar to a POSIX file, and its APIs also resemble that of POSIX.

```
fits_open(char* filename, int mode, int* handle)
fits_create(char* filename, int* handle)
fits_seek(int handle, long offset)
fits_remove(char* filename)
fits_read(int handle, void* buffer, long nbytes)
fits_write(int handle, void* buffer, long nbytes)
fits_flush(int handle)
fits_close(int handle)
```

1) open and create: We treat each FITS file as a PDC object. Since each opened/created file is assigned a unique CFITSIO handle, we associate this handle with the object ID of the corresponding PDC object. This association is managed by a hash table established at open/create time and released at close time.

Each PDC object includes properties such as dimensions and data types. Our current implementation designates all objects as one-dimensional and sets `PDC_CHAR` as their data type. These decisions are based on observations that Montage predominantly accesses FITS files sequentially and rarely performs sub-image selection. As there is no need for PDC servers to merge requests from multiple regions or dimensions, the 1D layout offers optimal performance.

Finally, in PDC, objects are always opened with both read and write permissions, so `mode` is simply ignored.

2) seek and remove: CFITSIO maintains a file pointer for each opened file. In our implementation, the file pointer corresponds to the global region offset, which the seek call modifies. This offset is also incremented after each successful read or write. The remove call is implemented using the PDC's `PDCobj_delete` API.

3) read and write: Read and write operations are executed using PDC's transfer calls. A naive approach involves immediately following `PDCregion_transfer_start` with `PDCregion_transfer_wait` at each read and write time. However, the wait call is expensive as it involves communication with PDC servers and data transfer from clients to servers. Unlike POSIX systems, which offer sequential consistency, the PDC's interface enables flexible control over when to make updates visible to others. This means that the PDC wait calls do not necessarily need to occur at the same time as the write call—they can be delayed until the close or flush time.

Additionally, knowing the I/O characteristics of Montage and CFITSIO, specific optimizations can be enabled automatically, such as aggregating writes and prefetching reads. These optimizations will be discussed in the following subsection.

4) flush and close: The flush call is implemented using the `PDCregion_transfer_wait` call, which ensures that modifications become available to all subsequent reads. Before performing any network communications, the wait call first checks whether there are any ongoing transfers.

For the close call, we release any temporary data structures and resources associated with the corresponding object. The close call also implies a transparent flush.

B. Workflow-oriented Optimizations

Traditional HPC workflows like Montage rely on parallel file systems (PFSs) such as Lustre and IBM Spectrum Scale for data exchange across phases. As discussed earlier, these PFSs are not designed to be workflow-aware. They are deployed system-wide and shared by all users, potentially missing significant opportunities for I/O optimizations specific to a particular workflow execution.

In contrast, PDC is a proactive data management system running at the user level and has the same lifespan as the targeted workflow. PDC acts as a private storage solution dedicated to supporting the I/O demands of a single workflow execution, thereby enabling more workflow-specific I/O optimizations. Here, we outline a few observations of the Montage workflow and the optimizations we implemented based on these observations.

- **Server-side Caching:** In Montage, some phases (e.g., projection) produce a large number of small files, which will be read by subsequent phases. These intermediate image files can be cached on the server side for fast retrieval.
- **Write Aggregation:** Our study of Montage I/O patterns using Recorder [14] revealed that Montage components never perform read-back operations within a single file open-close session. In other words, reads only occur once the file has been updated and closed. Therefore, we can aggregate and cache writes locally, performing a single transfer at flush or close time.
- **Read Prefetching:** Knowing that CFITSIO often performs small and contiguous reads, prefetching can be performed on the client side to reduce future network transfers.

V. Evaluation

This section presents the preliminary performance results of running Montage on PDC. All experiments were conducted on Perlmutter, a heterogeneous system at the National Energy Research Scientific Computing Center (NERSC), which includes 3,072 CPU-only nodes and 1,792 GPU-accelerated nodes. Our experiments utilized only the CPU nodes. Each node is equipped with 2 AMD EPYC 7763 (Milan) 64-core CPUs, 512 GB of DDR4 memory, a PCIe 4.0 NIC-CPU connection with a 25 GB/s bandwidth, and an HPE Slingshot 11 NIC.

The Montage workflow we evaluated is identical to the one shown in Fig 1. We ported all components within the workflow and ran it on 4 nodes, with 32 processes per node. The input dataset consisted of 91 FITS images, and the final output was a single mosaic image. During the workflow's execution, hundreds of intermediate FITS images were produced and consumed. Figure 2 reports the timing results of four components: mProjecExecMPI, mDiffExecMPI, mFitExecMPI, and mBgExec. These components exhibit diverse I/O access patterns. The first three are MPI programs that run on all allocated processes, while the last is a serial program running on a single process.

We enabled server-side caching for all runs, as it always leads to better performance, provided sufficient caching space is available. Our focus was on examining the effects of the other two optimizations—aggregation and prefetching—on different components. Aggregation consistently improved performance by reducing the number of writes and network transfers. For this specific workflow, aggregation is always feasible as conflicting reads only occur after the modified files have been closed and reopened. In other words, there is no need for strict POSIX consistency. Consequently, the aggregation optimization can leverage PDC's `transfer_all` and `wait_all` APIs to merge and delay writes effectively. Furthermore, components such as mBgExec and mDiffExecMPI benefit from both aggregation and prefetching due to their extensive contiguous write and read operations. Take mBgExec as an example, enabling both optimizations reduced the average execution time by a factor of 6 compared to the version with no optimizations.

Conversely, performing prefetching blindly can negatively impact performance. In scenarios where a process reads only a small portion of the object (e.g., mFitExecMPI and mProjExecMPI), prefetching may load unnecessary data, potentially reducing bandwidth. In these cases, an online tunable prefetching chunk size or an autonomous learning system during execution would yield better read performance.

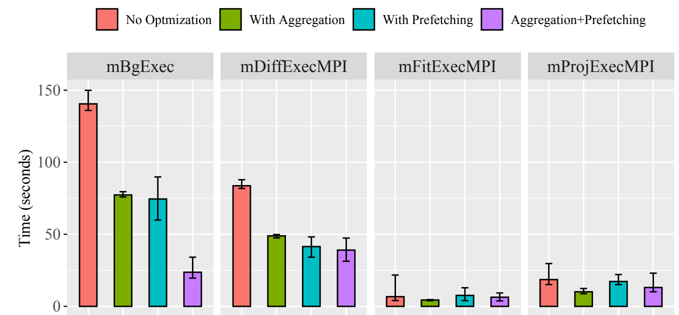

Fig. 2: Montage on PDC. Different components benefit from different optimizations, decided by their access patterns.

VI. Lessons Learned and Future Directions

Throughout the process of porting Montage to PDC, we have encountered and addressed several challenges. Here are some key lessons we have learned:

- **Object Creation:** The dimensions and size of an object are often unknown at the time of creation. The object creation interface and implementation should account for this uncertainty.
- **Global Synchronizations:** The client side of the object store should handle global synchronizations carefully. In workflows, tasks may follow different execution paths,

potentially causing global synchronizations to be blocked indefinitely.

- **Workflow I/O Patterns:** Workflow I/O patterns are often simple. For example, as seen in Montage, reads may not occur until the modified file has been closed, which means strict POSIX consistency is not required. Optimizations such as write aggregation can be performed to improve performance. These optimizations relax the provided consistency semantics without breaking the workflow.
- **Intermediate Files:** Intermediate files can be deleted safely after execution. They can be cached on the server side for faster retrieval during future read phases.
- **POSIX Support:** In Montage, most I/O is handled by CFITSIO, but some operations are performed directly using POSIX calls. These include checking file existence and using files to communicate with spawned child processes. A non-POSIX system needs to manage POSIX operations (which may require workflow modifications) to ensure execution correctness.

Our future tasks include: (1) testing with larger datasets on more nodes, (2) identifying and enabling additional optimizations for Montage, and (3) conducting a comprehensive comparison with existing parallel file systems such as IBM Spectrum Scale and Lustre.

ACKNOWLEDGMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 23-ERD-053. LLNL-CONF-866305. This manuscript has been co-authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy.

REFERENCES

[1] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7," *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951, 2018.

[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*, ser. AD '11, 2011.

[3] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck *et al.*, "ADIOS 2: The Adaptable Input Output System. A Framework for High-Performance Data Management," *SoftwareX*, vol. 12, p. 100561, 2020.

[4] C. Wang, K. Mohror, and M. Snir, "File System Semantics Requirements of HPC Applications," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2021, pp. 19–30.

[5] M. Vilayannur, S. Lang, R. Ross, R. Klundt, L. Ward *et al.*, "Extending the POSIX I/O Interface: A Parallel File System Perspective," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2008.

[6] C. Wang, K. Mohror, and M. Snir, "Formal Definitions and Performance Comparison of Consistency Models for Parallel File Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 6, pp. 937–951, 2024.

[7] M. J. Brim, A. T. Moody, S.-H. Lim, R. Miller, S. Boehm, C. Stanavige, K. M. Mohror, and S. Oral, "UnifyFS: A user-level shared file system for unified access to distributed local storage," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 290–300.

[8] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, J. Mu, Q. Koziol, J. Soumagne, V. Vishwanath, J. Liu *et al.*, "Toward scalable and asynchronous object-centric data management for hpc," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 113–122.

[9] M. Hennecke, "Daos: A scale-out high performance storage stack for storage class memory," *Supercomputing frontiers*, p. 40, 2020.

[10] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. Prince *et al.*, "Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking," *International Journal of Computational Science and Engineering*, vol. 4, no. 2, pp. 73–87, 2009.

[11] W. D. Pence, "CFITSIO: a FITS file subroutine library," *Astrophysics Source Code Library*, pp. ascl–1010, 2010.

[12] D. S. Katz, J. C. Jacob, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, G. Berriman, J. Good, A. Laity, and T. A. Prince, "A Comparison of Two Methods for Building Astronomical Image Mosaics on a Grid," in *2005 International Conference on Parallel Processing Workshops (ICPPW'05)*. IEEE, 2005, pp. 85–94.

[13] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. Da Silva, M. Livny *et al.*, "Pegasus, a Workflow Management System for Science Automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[14] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient parallel I/O tracing and analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 1–8.