A Transparent Server-managed Object Storage System for HPC

Short paper

Jingqing Mu*, Jerome Soumagne*, Houjun Tang†, Suren Byna†, Quincey Koziol†, Richard Warren*
*The HDF Group, Champaign, IL
†Lawrence Berkeley National Laboratory, Berkeley, CA
Email: *{kmu, jsoumagne, richard.warren}@hdfgroup.org, †{htang4, sbyna, koziol}@lbl.gov

***Abstract*—On the road to exascale, the high-performance computing (HPC) community is seeing the emergence of multi-tier storage systems. However, existing data management solutions for HPC applications are no longer suitable for handling the increased level of storage complexity and currently delegate that task back to the user.**

We describe a novel object-based data abstraction that takes advantage of deep memory hierarchies by providing a simplified programming interface that enables autonomous, asynchronous, and transparent data movement with a server-driven architecture. Users can define a mapping between the application memory and abstract storage objects, creating a linkage between either all or part of an object's content without data copy or transfer, avoiding explicit management of complex data movement across multiple storage hierarchies. We evaluate our system by storing plasma physics simulation data with different storage layouts.

***Keywords*—asynchronous transfer; data handling; large scale systems; memory management; object-centric models;**

I. Introduction

Scientific applications on upcoming HPC systems are facing challenges from three directions: extreme parallelism, a deepening heterogeneous memory hierarchy, and data that is massively increasing in volume and complexity. In particular, one of the challenges to address the diverse performance characteristics of deep memory hierarchies expected in exascale systems is the capability and efficiency of data movement across storage layers. Existing HPC data management and movement solutions, which were designed for simpler systems are no longer able to handle that level of complexity; similarly, scientific data models, which have been designed for 2-tiered storage hierarchies, need to be revised to embrace a more gradated storage hierarchy.

Moving toward an end-to-end, object-centric data abstraction and storage mechanism that takes advantage of deep storage hierarchies and enables proactive automated performance tuning, we have been investigating *Proactive Data Containers* (PDC) [1]. A PDC is a container that may reside in a single storage location (i.e., memory, NVRAM, disk, etc.) or span across multiple levels, and stores data in an object-oriented manner. The PDC system provides an interface for creating, updating, retrieving, and deleting data objects and for managing metadata on those objects.

Our previous work [1] demonstrated I/O operations initiated explicitly from a client application. In this paper, we focus on mapping objects in different levels of the storage hierarchy, as well as efficient strategies implemented for moving data asynchronously between storage hierarchies using PDC. We present new APIs to facilitate *server-managed* I/O that allows applications to provide the intent of persistence of objects while PDC servers handle the actual data movement and persistence, removing the burden of data movement decisions from application developers. Overall, this paper has the following contributions:

1) Application intent-based object persistence mechanism with transparent data management in hierarchical storage using the PDC system;
2) Exploration of PDC data management server placement options in shared and dedicated modes;
3) Implementation of data movement strategies using TCP and Cray GNI;
4) Evaluation and demonstration of an object-centric HPC storage system for scientific use cases.

The paper is organized as follows: we first discuss related work in Section II, and then introduce in Section III our PDC system architecture enabling object storage and data movement model. In Section IV, we outline APIs that enable data transfer without explicit read and write semantics and provide experimental results, evaluating new methodology in the PDC system using I/O patterns representative of science applications on HPC systems.

II. Related Work

As part of the POSIX standard defined in the late 1980s, POSIX-IO [2], describes the file access API, data model, and data consistency semantics. In POSIX-IO, data is viewed as a stream of bytes. Parallel file systems, such as PVFS [3], [4], Lustre [5], GPFS [6], and NFS [7] were all designed to comply with the POSIX-IO standard. However, compliance to POSIX-IO comes with a cost, as its original design was not intended for highly concurrent programming models, which are now common in HPC systems [8]. As we are moving toward an increasing system complexity with an increasing number of memory/storage layers, the I/O bottleneck is becoming increasingly severe and significantly hinders the overall application performance [9], [10]. To alleviate the I/O bottleneck, research efforts have been made to relax the POSIX semantics across the parallel I/O software stack: from high-level libraries (e.g., HDF5 [11], netCDF [12], ADIOS [13]), I/O middleware (e.g., MPI-IO [14], TAPIOCA [15]), to I/O forwarding layers [16]. HDF5, netCDF, and ADIOS provide

2168-9253/18/$31.00 ©2018 IEEE
DOI 10.1109/CLUSTER.2018.00063

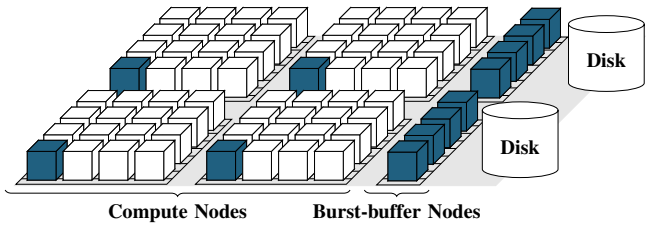

Fig. 1. Simplified representation of a pre-exascale system with multi-tier storage. PDC running services are highlighted in blue. PDC services are either co-located within the same node or distributed over remote burst buffer nodes.

an array-based data model to organize the data and define data access semantics.

New data models have also been proposed, such as *object-based* storage [17] [18] [19] [20], which describes an abstract data container that consists of many byte-streams (or *objects*). [21] provides a detailed analysis that discusses whether the object model is the right abstraction level in HPC and on big data platform. Ceph [22] and DAOS [23] are emerging as possible replacements for parallel file systems, with objects being defined as first-class citizens. However, their scalability is still under evaluation and their features are still in development [24]. Object storage in cloud environments, such as Amazon S3 [25] and OpenStack Swift [26], provide object-centric interfaces and data management. However, their applicability in HPC systems is limited by the tightly coupled storage hierarchy and stringent performance requirements of science applications. Tyr [27] proposes a new object storage system, which features built-in high-performance support for multiple object transactions and guarantees objects transaction with low overhead. Furthermore, research efforts to implement object-based storage have been attempted on single levels separately, but none has integrated those efforts across the entire memory hierarchies. UNITY [28] proposes a distributed runtime on each node and manages local data objects placement by mapping data from application to local resources upon availability. However, TCASM [29] used by UNITY to share memory from application requires kernel-level modification.

III. Architecture

In this section, we introduce our data management interface in the *Proactive Data Containers* (PDC) system that targets enabling efficient and scalable data management for the upcoming exascale storage systems. Figure 1 shows a simplified representation of a PDC enabled system and where the application's data may reside. A PDC runtime system, consisting of servers, allows efficient data movement in critical areas of the exascale data management software stack to take place and enables *in-transit* and *in-situ* analysis operations.

The main abstractions of PDC are its data constructs and operations performed on the data constructs. As shown in Figure 2, the data constructs include *Containers*, *Objects*, and *Regions*, all of which can have different *Properties*. An *Object* is a generic term to describe byte streams in an abstract manner, which can represent either a data variable or an application object depending on a user's definition. In PDC,

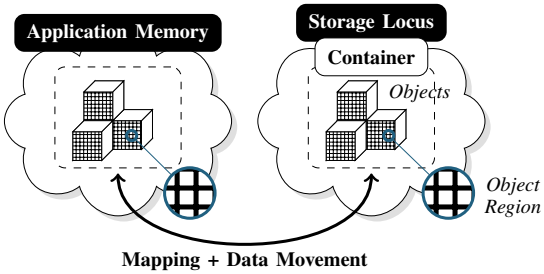

Fig. 2. A mapping operation established between an application's memory region and a PDC object region.

data objects can be arrays or key-value pairs. Objects in PDC are globally visible, independently of their storage hierarchy.

A *Container* is a collection of *Objects* that share similar user-defined attributes, such as all data variables produced by a simulation or an experiment. As shown in Figure 2, PDC contains a set of objects, where the objects are managed by PDC services and are placed at any level of the storage hierarchy (e.g., NVRAM, burst-buffer, disk, etc.). We abstract the containers consisting of data within the entire storage stack. This approach spreads the data, objects, and containers over different types of storage media, defined as *storage locus*, which can be thought of as levels of cache. PDC also uses the concept of spatial *regions*, which partition the problem domain into smaller sub-regions. Each region contains the actual data and associated metadata, and is the basic unit for data movement operations in PDC. All of the previously mentioned entities include *Properties*, regarded as metadata. A user also may add rich metadata as *tags* [30].

Common I/O libraries have been providing so far *explicit* data movement operations through read and write functions, including our previous work [1]. In this paper, we introduce the concept of *object mapping* to make those calls *implicit* to the user by defining a *map* operation.

1) Object Mapping Mechanism: As highlighted in Figure 2, the object mapping primitive allows a user to define a mapping between a region within an application's memory and a region within a global PDC object. Mapping operations are defined on a per-region basis and can be thought of as a publish and subscribe mechanism, in the sense that once a mapping is established and a region is published, data movement can occur to keep updates globally visible. When defining a mapping, the application provides property information about the mapped region, which is essential for the PDC system to keep track of the mappings that are established and prevent potential overlaps. More complex mappings can be built upon this primitive such as object to object mapping, though in this paper we focus on the application memory to PDC object mapping exclusively.

2) Consistency and Locking Mechanism: To keep data consistency between the application's memory and the PDC object, we propose locking semantics for PDC objects (at the region granularity so that multiple regions of an object can be concurrently updated) and distinguish *read locks* from *write locks*. Assuming the mapping from memory to object has already been established, when a user has the intent to modify

the application's memory region, the object region *write* lock must be acquired before any memory write access can occur (or that access would be considered as undefined). After the write lock is acquired, further changes to the object data region is not allowed globally until the lock is released. Once the lock is released per user's request, it effectively notifies the PDC system that it is now safe to move data between storage locations, and data movement will occur asynchronously if the memory region has been modified. Similarly in the case of read locks, the application expresses the intent of accessing the data and effectively prevents the PDC system from making any implicit update to the memory mapped buffers while they are being read.

IV. Implementation and Evaluation

Our implementation of PDC adopts a client-server approach to monitor and manage I/O operations on objects. Along with enabling multithreading capabilities to the server, PDC is able to asynchronously handle I/O operations in the background, while the client application proceeds with computation without having to wait for persistent I/O to be done, exploiting the compute and storage resource strengths of each storage locus. Several approaches can be considered when designing a client-server middleware for HPC, however the trend has now been to design HPC system software components in user-space exclusively: first, no additional kernel module or kernel code modification is necessary; second, it eliminates the extra cost of entering the kernel. In our architecture, PDC servers manage both metadata and data. PDC separates data from metadata, which is also used to locate the data spread across data servers. Both data and metadata are managed in a *flat namespace*.

We have developed two server placement strategies, where the server may be either co-located on the same nodes with the application (i.e., *shared* mode) or on separate nodes (i.e., *dedicated* mode). In both cases, we have relied on the Mercury [31] package, an HPC-optimized C library for Remote Procedure Calls (RPCs), as the communication mechanism between client and server and between servers.

1) Experimental Setup: We use the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC), which is a Cray XC40 supercomputer with 1630 Intel Xeon Haswell nodes. Each node consists of 32 cores and 128GB memory. The shared file system Lustre is HDD-based. There is also an SSD-based "Burst Buffer", located between compute nodes and storage systems on Cori.

With the *shared* mode, we have one PDC server on each node, which occupies one core leaving the remaining 31 cores for user application execution. In the *dedicated* mode, PDC servers and user's application are on separate nodes. We configure Mercury with two communication protocols using the libfabric plugin [32] over TCP and over Cray GNI [33]. In the latter case, the PDC server was configured to use Cray DRC to allow the user's application and PDC server to share credentials and communicate together.

We used VPIC-IO to evaluate the PDC system's performance, which is extracted from VPIC [34], a code developed

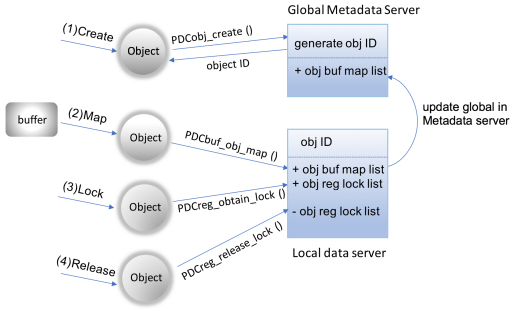

Fig. 3. Four major steps to enable data movement by PDC without making any explicit data copy or transfer call.

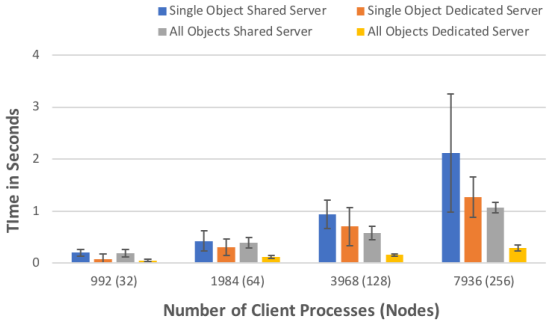

Fig. 4. Time for one VPIC-IO property to establish mapping between memory buffer and a PDC object.

for simulating several plasma physics phenomenon. In VPIC-IO, each MPI process writes a region of 8M (8×2^{20}) particles and each particle has 8 properties. Each reported time includes client and server communication time, data and metadata server communication time, metadata maintain time and data movement time, if any of these is involved. We only report the time by multithreading server execution.

2) Mapping Memory to a PDC Object: As the first step in Figure 3, the client initiates a call to create an object by calling `PDCobj_create()` before the mapping procedure, and the metadata server receives the RPC call, creates a unique metadata on one metadata server, and generates a global object ID. As the second step in Figure 3, the client establishes a map relationship between the application's data in memory and an object, created globally at any level of the memory hierarchy, by calling `PDCbuf_obj_map()`.

Figure 4 shows the time to map between memory and an object for both shared and dedicated modes. The "single object" timing is measured by adding synchronization before and after each map function. For "all objects" case, the time is measured by adding a barrier before the first and after the last map function. That time is divided by 8 (the number of particle properties in VPIC-IO) for average and compared to "single object" measurement. That is the same for the following experiments. The x-axis is the number of client processes and PDC servers (in brackets). We observe the mapping process with dedicated mode is faster than with shared mode. This is because the servers in dedicated mode have more compute resources due to configuration. Overall, the mapping overhead is small and is a one-time effort.

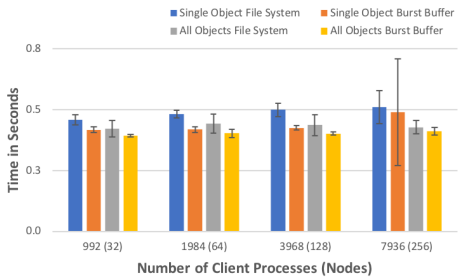

Fig. 5. Time to release the lock for one object with shared server.

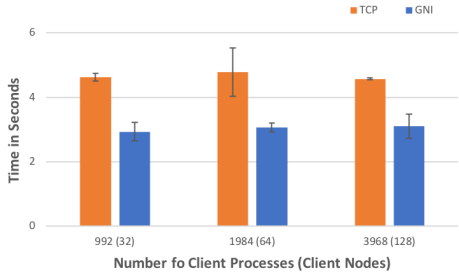

Fig. 6. Time to release the lock for all objects with dedicated server.

3) Object Lock: Typically, before an application modifies data in memory, a `PDCreg_obtain_lock()` call is required for the mapped regions as shown in Figure 3 (step 3). The object region lock list, which is stored only in the local data servers, is updated accordingly. Unlike the mapping requests, there is no communication between data and metadata servers. Similar to the mapping overhead, the locking overhead is less than 0.15s, which is negligible.

4) Object Lock Release: Once a client initiates a lock release operation by calling `PDCreg_release_lock()` as shown in Figure 3 (step 4), the data server takes over the task and determines if data transfer is needed by going through the object region lock list. If so, the data server initiates a one-sided data transfer using remote memory access (RMA) by Mercury, which makes use of cross memory attach (CMA) for zero copy transfers when using shared-memory in shared mode, or makes use of native RDMA exposed by the system (e.g., uGNI over libfabric) in dedicated mode. The data server responds to the application lock request once it receives data from the client, and in background asynchronously writes to lower storage tiers (i.e., burst buffer, file system). With two copies of results, data in data server is able to be quickly shared to other analysis or post processing tasks, such as in situ processing, while the copy in permanent storage allows the data to be recovered even if the application sees a fault for any reason.

With the extra threads dealing with asynchronously transfer, lock release time is around 0.5s for the shared mode (Figures 5) and 0.6s for the dedicated mode (not presented) by hiding the actual data transfer time to lower storage, whereas the time is ≈2s to 2.5s without asynchronous I/O.

5) Object Lock Release with GNI: To enable faster communication and data transfer, we compare the lock release time using Cray GNI instead of TCP used in previous experiments,

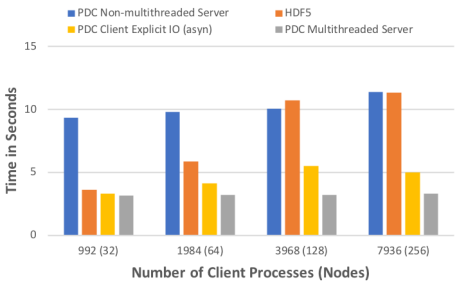

Fig. 7. Time to move data from application memory to burst buffer by the user's point of view.

since libfabric with GNI is targeting multithreaded applications requiring concurrent access to Aries high speed network (Cori network), with minimal contention between threads. In Figure 6 we show that the performance of lock release for dedicated mode using GNI is ≈3s compared to 4.5s using TCP protocol to release all objects.

6) PDC Server Controlled Data Movement Performance Comparison: We compare the I/O performance of the new PDC mechanisms against HDF5 (with multiple optimizations [35]) for VPIC-IO and against our previous work of explicit I/O in PDC [1], by moving data from memory to a lower storage tier through Lustre (not presented) and burst buffer (Figure 7) from the view of application users. Each process writes 256*MB* of data. We use PDC servers in shared mode since that was the configuration in our previous work. When using HDF5, data is written to the storage directly as a single file. From the users' perspective, the performance of the new PDC server-managed method is improved by 65% and 49% on average compared to HDF5 writing directly to file system and burst buffer respectively, with asynchronous data movement. It also outperforms our previous client-explicit asynchronous I/O by 10% and 25% on average with respect to writing to file system and burst buffer.

V. Conclusion

Considering the challenges presented by current I/O and inefficiency of data management and movement across deep memory hierarchies, we propose and investigate Proactive Data Containers (PDC), which may reside in a single storage location or span across multi-tier storage. The PDC object-centric system provides a novel mechanism of map, lock, and lock release to enable autonomous and asynchronous data movement, eliminating the burden of data movement decisions by application scientists. Our future work includes enabling data transformation and analysis framework in PDC as well as making communication between servers topology aware.

Acknowledgment

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 and DE-SC0016454 (Program manager: Dr. Lucy Nowell). This research used resources of the National Energy Research Scientific Computing Center (NERSC).

REFERENCES

[1] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, J. Mu, Q. Koziol, J. Soumagne, V. Vishwanath, J. Liu, and R. Warren, "Toward Scalable and Asynchronous Object-centric Data Management for HPC," in *CCGRID*, 2018.

[2] S. R. Walli, "The POSIX Family of Standards," *StandardView*, vol. 3, no. 1, pp. 11–17, Mar. 1995.

[3] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel Virtual File System for Linux Clusters," *Linux J.*, 2000.

[4] M. Moore *et al.*, "OrangeFS: Advancing PVFS," *FAST poster session*, 2011.

[5] P. J. Braam *et al.*, "The Lustre storage architecture," 2004.

[6] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, 2002, pp. 231–244.

[7] S. Microsystems, "NFS: Network File System Protocol Specification," 1989.

[8] K. J. Barker *et al.*, "Entering the Petaflop Era: The Architecture and Performance of Roadrunner," in *Supercomputing*, 2008.

[9] M. Jung, W. Choi, J. Shalf, and M. T. Kandemir, "Triple-A: A Non-SSD Based Autonomic All-flash Array for High Performance Storage Systems," *SIGPLAN Not.*, vol. 49, pp. 441–454, 2014.

[10] F. Schürmann and et.al., "Rebasing I/O for Scientific Computing: Leveraging Storage Class Memory in an IBM BlueGene/Q Supercomputer," in *ISC*, 2014, pp. 331–347.

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

[12] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A High-Performance Scientific I/O Interface," in *Supercomputing, 2003 ACM/IEEE Conference*, Nov 2003, pp. 39–39.

[13] Q. Liu, J. Logan, Y. Tian *et al.*, "Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[14] R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance," in *ICPADS*, 1999, pp. 23–32.

[15] F. Tessier, V. Vishwanath, and E. Jeannot, "TAPIOCA: An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers," in *CLUSTER*, 2017, pp. 70–80.

[16] B. Welton, D. Kimpe, J. Cope, C. M. Patrick, K. Iskra, and R. Ross, "Improving I/O Forwarding Throughput with Data Compression," in *CLUSTER*, 2011, pp. 438–445.

[17] A. L. Brown and R. Morrison, "A Generic Persistent Object Store," *Software Engineering Journal*, vol. 7, no. 2, pp. 161–168, March 1992.

[18] J. E. B. Moss, "Design of the mneme persistent object store," *ACM Trans. Inf. Syst.*, vol. 8, no. 2, pp. 103–139, Apr. 1990. [Online]. Available: http://doi.acm.org/10.1145/96105.96109

[19] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison, "Persistent object management system," *Software: Practice and Experience*, vol. 14, no. 1, pp. 49–71, 1984. [Online]. Available: http://dx.doi.org/10.1002/spe.4380140106

[20] G. A. Gibson *et al.*, "A Cost-effective, High-bandwidth Storage Architecture," *SIGPLAN Not.*, vol. 33, pp. 92–103.

[21] P. Matri, A. Costan, G. Antoniu, J. Montes, and M. Prez, "Could Blobs Fuel Storage-Based Convergence Between HPC and Big Data?" in *CLUSTER*. IEEE, 2017.

[22] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-performance Distributed File System," in *OSDI*, 2006, pp. 307–320.

[23] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and Friends: A Proposal for an Exascale Storage System," in *Supercomputing*, 2016, pp. 50:1–50:12.

[24] M. S. Breitenfeld, N. Fortner, J. Henderson, J. Soumagne, M. Chaarawi, J. Lombardi, and Q. Koziol, "DAOS for Extreme-scale Systems in Scientific Applications," *CoRR*, vol. abs/1712.00423, 2017. [Online]. Available: http://arxiv.org/abs/1712.00423

[25] Amazon. Amazon Web Services. Http://s3.amazonaws.com.

[26] J. Arnold, *OpenStack Swift: Using, administering, and developing for swift object storage*. O'Reilly Media, Inc., 2014.

[27] P. Matri, Y. Alforov, A. Brandon, M. Kuhn, P. Carns, and L. T., "Tr: Blob Storage Meets Built-In Transactions," in *SC*. IEEE, 2016.

[28] T. Jones, M. J. Brim, G. Vallee, B. Mayer, A. Welch, T. Li, M. Lang, L. Ionkov, D. Otstott, A. Gavrilovska, G. Eisenhauer, T. Doudali, and P. Fernando, "UNITY: Unified Memory and File Space," in *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*, ser. ROSS '17. New York, NY, USA: ACM, 2017, pp. 6:1–6:8. [Online]. Available: http://doi.acm.org/10.1145/3095770.3095776

[29] D. Otstott, N. Evans, and L. Ionkov, "Enabling composite applications through an asynchronous shared memory interface." Big Data (Big Data), 2014 IEEE International Conference on, 2014, pp. 219–224.

[30] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol, "SoMeta: Scalable Object-Centric Metadata Management for High Performance Computing," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017, pp. 359–369.

[31] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling Remote Procedure Call for High-Performance Computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2013, pp. 1–8.

[32] Libfabric. [Online]. Available: https://ofiwg.github.io/libfabric/

[33] 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*, 2016.

[34] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulationa)," *Physics of Plasmas*, vol. 15, no. 5, 2008.

[35] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, "Taming Parallel I/O Complexity with Auto-tuning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, 2013, pp. 68:1–68:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503278