Scalable Data Management for Object-based Storage Systems

Bharti Wadhwa

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Application

Ali R. Butt, Chair
Eli Tilevich
Francisco Servant
Na Meng
Feiyi Wang

July 16, 2020
Blacksburg, Virginia

Keywords: Lustre, Ceph, High Performance Computing, Parallel File Systems, Parallel I/O Optimization, Load Imbalance, Resource Contention

Scalable Data Management for Object-based Storage Systems

Bharti Wadhwa

(ABSTRACT)

Parallel I/O performance is crucial to sustain scientific applications on large-scale High-Performance Computing (HPC) systems. Large scale distributed storage systems, in particular the object-based storage systems, face severe challenges for managing the data efficiently. Inefficient data management leads to poor I/O and storage performance in HPC applications and scientific workflows. Some of the main challenges for efficient data management arise from poor resource allocation, load imbalance in object storage targets, and inflexible data sharing between applications in a workflow. In addition, parallel I/O makes it challenging to shoehorn new interfaces, such as taking advantage of multiple layers of storage and support for analysis in the data path. Solving these challenges to improve performance and efficiency of object-based storage systems is crucial, especially for upcoming era of exascale systems.

This dissertation is focused on solving these major challenges in object-based storage systems by providing scalable data management strategies. In the first part of the dissertation (Chapter 3), we present a resource contention aware load balancing tool (`iez`) for large scale distributed object-based storage systems. In Chapter 4, we extend `iez` to support Progressive File Layout for object-based storage system: Lustre. In the second part (Chapter 5), we present a technique to facilitate data sharing in scientific workflows using object-based storage, with our proposed tool *Workflow Data Communicator*. In the last part of this dissertation, we present a solution for transparent data management in multi-layer storage hierarchy of present and next-generation HPC systems.

This dissertation shows that by intelligently employing scalable data management techniques, scientific applications' and workflows' flexibility and performance in object-based storage systems can be enhanced manyfold. Our proposed data management strategies can guide next-generation HPC storage systems' software design to efficiently support data for scientific applications and workflows.

Scalable Data Management for Object-based Storage Systems

Bharti Wadhwa

(GENERAL AUDIENCE ABSTRACT)

Large scale object-based storage systems face severe challenges to manage the data efficiently for HPC applications and workflows. These storage systems often manage and share data inflexibly, without considering the load imbalance and resource contention in the underlying multi-layer storage hierarchy. This dissertation first studies how resource contention and inflexible data sharing mechanisms impact HPC applications' storage and I/O performance; and then presents a series of efficient techniques, tools and algorithms to provide efficient and scalable data management for current and next-generation HPC storage systems.

To my mother, for always believing in me
To the memory of my father, who never gave up

Acknowledgements

This Ph.D. would not have been possible without the support and encouragement from many people; I would like to express my deepest gratitude to the following individuals.

First, I would like to thank my advisor Ali R. Butt for his constant support and guidance. Ali gave me the great opportunity to work at Distributed Systems and Storage Laboratory and guided me throughout my journey as a graduate student and a researcher. Ali has taught me everything that was needed to succeed in graduate school: how to read/write a research paper, how to deliver a good presentation, how to communicate with peers and establish a good and productive collaboration relationship, and more. Most importantly, he believed in me and always kept me motivated. Without Ali, I would have never achieved this goal.

I have been fortunate to work with many other researchers and faculty members at Virginia Tech and other institutions. I am grateful to Dr. Suren Byna (Lawrence Berkeley National Lab) for providing valuable insights into HPC research in the first years of my Ph.D. I am thankful to Dr. Feiyi Wang, Dr. Sarp Oral, and Dr. Sarah Neuwirth for their help with the `iez` project. I am grateful to Dr. Lars Schneidenbach (IBM Research T. J. Watson) for his guidance on the *Workflow Data Communicator* project. Last but not least, I am grateful to Arnab K. Paul for his help with the `iez` and *Workflow Data Communicator* projects.

I would also like to thank my Ph.D. committee members: Professors Eli Tilevich, Francisco Servant, Na Meng and Dr. Feiyi Wang, for their insightful comments and feedback on my research at various stages of my Ph.D.

My work as a Ph.D. student was made possible, and constantly enjoyable, by the collaboration with my labmate, and close friend, Arnab. He has been a constant source of inspiration for me through out my stay at Virginia Tech. His passion for research, work ethic, and willingness to discuss and share ideas were a great asset while working on the three papers we co-author together. I will always remember our brain-storming sessions regarding research problems and our lengthy discussions about various aspects of life.

I would also like to thank my friend and DSSL's alumnus Ali Anwar for mentoring me, for generously answering my many questions and for helping me throughout my Ph.D. Ali has been an inspiration for me in this journey and I will always be grateful to him for his constant support and encouragement. I am fortunate to have worked with a great group of labmates at DSSL: Safdar, Yue, Nannan, Luna, Hyogi, Hadeel, Jingoo, Subil, Redwan and Debasmita. Our social interactions have enriched my grad school life. I would also like to thank all my close friends in Blacksburg and my long-distance friends, Shaina,

Sherry, Kuldeep, Raghav and Vivek for supporting me in myriad different ways.

Finally, I am immensely grateful to my family without whom this Ph.D. would have been lightweight and meaningless. My late father, who always believed in my dreams more than his own, and prepared me to be a strong and independent person. My mother, who taught me patience, positive attitude and hard work. Lastly, I am grateful to my brother, Navish's support and encouragement. He is my inspiration for daring to do a Ph.D. and coming to the United States. He has been my first mentor and has always inspired me to rise higher. I dedicate this dissertation to my family: Ma, Papa, Navish and Julia.

Funding Acknowledgement: My research was supported by the National Science Foundation under the grants: CNS-1405697, CNS-1615411, and CNS-1565314/1838271, CCF-1919113, CS at Virginia Tech, US Department of Energy, and IBM Research Watson Lab. Many thanks to all for making this dissertation possible.

Contents

List of Figures xi

List of Tables xv

1 Introduction 1

1.1 Motivation 1

1.1.1 Resource Contention Aware Load Balancing 2

1.1.2 Efficient Data Sharing in Scientific Workflows 2

1.1.3 Data Management in Multi-layer Storage Hierarchy 3

1.2 Research Contribution 3

1.3 Dissertation Organization 4

2 Background 6

2.1 Object-based Storage Systems 6

2.2 Load Management in Distributed Storage 8

2.3 Data Sharing in Scientific Workflows 9

2.4 Data Management in Multi-layer Storage Hierarchy 10

3 Resource Contention Aware Load Balancing 12

3.1 Introduction . . . 12
3.2 Background and Motivation . . . 13
3.2.1 Lustre Architecture . . . 13
3.2.2 Approaches for Load Balancing in Lustre . . . 14
3.2.3 Use Cases and Benchmarks . . . 15
3.2.4 Load Imbalance in a Default Lustre Deployment . . . 15
3.3 System Design . . . 17
3.4 Evaluation . . . 26
3.4.1 Chapter Summary . . . 32

4 Load Balancing for Progressive File Layout **33**

4.1 Introduction . . . 33
4.1.1 Progressive File Layout . . . 33
4.1.2 Motivation . . . 34
4.2 Design . . . 35
4.2.1 Parallel File Access Modes for Varying Striping Layouts . . . 37
4.2.2 Configuration Manager . . . 38
4.2.3 Interaction Database . . . 38
4.2.4 Placement Library . . . 39
4.3 Evaluation . . . 41

4.3.1 Load Balance for FPP Access IOR 42
4.3.2 Load Balance for FPP Access HACC-I/O 43
4.3.3 I/O Performance 44
4.3.4 Utilization of OSSs 45
4.3.5 Load Balance for Single Shared File Access 46
4.3.6 Load Balance for Concurrent Applications 47
4.4 Chapter Summary 47

5 Facilitating Data Sharing in Scientific Workflows 49

5.1 Introduction 49
5.2 Background 51
5.2.1 Workflow 51
5.2.2 Data Broker 53
5.2.3 Object Based Storage 55
5.2.4 Ceph Object Store 55
5.3 System Design 58
5.3.1 *Workflow Data Communicator* Architecture 58
5.3.2 *Workflow Data Communicator* Components 59
5.3.3 Functions in *Workflow Data Communicator* 62
5.4 Evaluation 63

5.4.1 Benchmarking . . . 64
5.4.2 Data Access Patterns . . . 64
5.4.3 Workflows . . . 67
5.5 Chapter Summary . . . 70

6 Data Management in Multi-layer Storage Hierarchy **72**
6.1 Introduction . . . 72
6.2 Background . . . 74
6.2.1 Scientific Applications Use Cases . . . 74
6.3 Design . . . 75
6.3.1 Object Definition . . . 75
6.3.2 Object-based Storage Model for Multi-layer HPC Storage . . . 76
6.3.3 Object Creation and Data Mapping Process . . . 80
6.4 Evaluation . . . 81
6.4.1 VPIC-I/O . . . 81
6.4.2 HACC-I/O . . . 84
6.4.3 Chapter Summary . . . 85

7 Conclusion **87**

Bibliography **89**

List of Figures

3.1 Overview of Lustre architecture. 14

3.2 OST Utilization under RR for IOR. 16

3.3 OST Utilization under RR for HACC-I/O. 16

3.4 Overview of the proposed `iez` architecture. 17

3.5 Publisher-Subscriber model for statistics collection. 22

3.6 Graph used in OST Allocation Algorithm. 26

3.7 OST Storage Utilization for IOR in FPP mode and Stripe Count 8. 27

3.8 OST Storage Utilization for IOR in FPP mode and Stripe Count 4. 28

3.9 OST Storage Utilization for HACC-I/O in FPP mode. 28

3.10 Read and write performance of IOR and HACC-I/O for FPP and SSF accesses for different Stripe Counts. 29

3.11 OSS Storage Utilization for IOR. 30

3.12 OST Storage Utilization for IOR in SSF mode for 16 Processes. 31

3.13 OST Storage Utilization for IOR in SSF mode for 32 Processes. 31

3.14 OST Storage Utilization for a simultaneous IOR and HACC execution in FPP mode with Stripe Count of 4 and 8, respectively. 32

4.1 An example PFL layout. 34

4.2 OST utilization under default load balancer in Lustre (RR) for IOR and HACC-IO for Non-PFL setup (stripe count 8). 34

4.3 OST utilization under default load balancer in Lustre (RR) for IOR and HACC-IO PFL setup (Configuration 1) 35

4.4 Overview of the updated `iez` architecture. 36

4.5 Comparison between FPP and SSF approaches in non-PFL and PFL layouts. 37

4.6 OST storage utilization for IOR in FPP mode and non-PFL layout (stripe count 8). 42

4.7 OST storage utilization for IOR in FPP mode and PFL Configuration 1. . 43

4.8 OST storage utilization for IOR in FPP mode and PFL Configuration 2. . 43

4.9 OST COST for HACC-IO in FPP mode and non-PFL layout (stripe count 8). 44

4.10 OST COST for HACC-IO in FPP mode and PFL layout. 44

4.11 Read and write performance of IOR and HACC-I/O for FPP and SSF accesses for PFL and non-PFL layouts. 45

4.12 OSS Storage Utilization for IOR in FPP mode for PFL Configuration 1. . 46

4.13 OST Cost for IOR in SSF mode in non-PFL layout (stripe count 8). 46

4.14 OST Cost for IOR in SSF mode in PFL Configuration 1. 47

4.15 OST Storage Utilization for a simultaneous IOR and HACC execution in FPP mode in PFL Configuration 1 and PFL Configuration 2, respectively. 48

5.1 An example of scientific workflow. 52

5.2 An example of an APEX scientific simulation workflow [6]. 52

5.3 Overview of Data Broker. 53

5.4 Block-based vs Object-based storage model. 56

5.5 Ceph storage cluster. 57

5.6 Overview of system architecture. 58

5.7 Mapping namespaces to object pools. 61

5.8 Sequence diagram of *Workflow Data Communicator*. 62

5.9 Data access patterns. 64

5.10 Processing time (ms) for data access patterns. 66

5.11 Average commit latency (ms) for data access patterns. 66

5.12 Average apply latency (ms) for data access patterns. 66

5.13 Processing time (sec) on small scales for broadcast data access pattern. . . 67

5.14 Processing time (sec) on large scales for broadcast data access pattern. . . 67

5.15 Benchmark workflows. 68

5.16 Processing time (ms) to run workflows. 69

5.17 Brain Atlas workflow. 70

5.18 Montage workflow. 71

6.1 Exascale system storage hardware stack. 73

6.2 Properties of a particle in a VPIC simulation. 75

6.3 The proposed object structure. 76

6.4 VPIC data represented as Basic Objects. 77

6.5 VPIC data represented as Composite Object. 77

6.6 Data Storage for VPIC-I/O: HDF5 Vs. Object Interface. 78

6.7 High-level architecture of the proposed object storage system. 78

6.8 Write performance for VPIC-I/O Basic Objects. 82

6.9 Write performance for VPIC-I/O Composite Objects. 83

6.10 Write performance for VPIC-I/O using Verical Sharding. 83

6.11 Read performance for VPIC-I/O using Verical Sharding. 84

6.12 Write performance for HACC-I/O Composite Objects. 85

6.13 Read performance for HACC-I/O Composite Objects. 86

List of Tables

3.1 Interaction Database Snapshot showing write requests for HACC-I/O in FPP mode. 18

3.2 Interaction Database Snapshot showing OST Allocation for HACC-I/O in FPP mode. 26

4.1 Interaction database snapshot for IOR in FPP mode in PFL layout. 38

4.2 Interaction database snapshot for IOR in FPP mode in non-PFL layout. . 39

4.3 Interaction database snapshot showing OST allocation for IOR in FPP mode in non-PFL layout. 41

4.4 Interaction database snapshot showing OST allocation for IOR in FPP mode in non-PFL layout. 42

5.1 Example set of namespace and data access functions. 54

5.2 Benchmark results of *Workflow Data Communicator*. 64

6.1 Object APIs for creating containers/objects and writing to the storage system. 79

6.2 Specifications of various Object-Sharding Strategies for VPIC-I/O. 82

Chapter 1

Introduction

Scalable computing systems such as HPC deployments are the backbone of modern computing-based research and discovery. A large number of scientific applications from domains such as Physics, Geology, and Cosmology use HPC systems for storage and computation of scientific data. With the upcoming exascale era, these scalable HPC systems are required to perform optimally, more than ever before. The performance of HPC systems is greatly dependent on efficient I/O and storage mechanism, which is a core components of these distributed systems. In recent years, object-based storage systems, such as Lustre [8] and Ceph [3] have been widely popular to support large scale HPC applications. These object-based storage systems combine key advantages like high scalability and reliability and an improved data throughput by separating data and metadata management.

1.1 Motivation

HPC systems are known for their massive computation power but they are highly dependent on their storage and I/O systems to transfer input data and output results. HPC applications perform data storage and I/O on parallel backend storage systems which face many challenges for managing data efficiently between their several components. Inefficient data management that arise from poor resource allocation, inflexible data sharing between applications in a workflow, and multiple layers of storage hierarchy in the object-based storage systems, leads to poor I/O and storage performance in the HPC applications and scientific workflows.

To address the above issues, this dissertation proposes, designs, and implements a series of novel techniques, algorithms, and frameworks, for scalable data management in object-based storage systems. This dissertation is focused on applying simple and effective data management strategies for existing and upcoming exascale object-based storage systems for HPC to bridge the gap between modern data-intensive applications and scientific workflows. In the following sections we briefly describe the research problems, proposed

research methodologies, and evaluation results of each work included in this dissertation.

1.1.1 Resource Contention Aware Load Balancing

Load imbalance and poor resource allocation have been identified as major causes of performance degradation in many HPC storage systems, including Lustre [90], the widely-used object-based storage system for scientific computing. A typical Lustre deployment consists of a large number of distributed Object Storage Targets (OSTs) that are managed by Object Storage Servers (OSSs). Application data is stored on Lustre OSTs using the underlying block storage strategy. When application data is written to a file, an associated OSS maps the file into multiple objects that are striped into different OSTs. The striping is done in a way to yield efficient parallel access. Extant practice is to use round-robin scheduling to place objects on the OSTs. However, this often leads to a load imbalance across the OSTs given typical non-uniform data accesses and contention for OST resources containing "hot" objects, consequently affecting the I/O performance and thus the overall application performance. There is a clear need of *contention-aware placement of application data to the various resources to yield improved load balancing and high performance* for large-scale parallel file systems in HPC. In this work, we develop a novel end-to-end control plane, that combines the application-centric strengths of client-side approaches with the system-centric strengths of server-side approaches.

1.1.2 Efficient Data Sharing in Scientific Workflows

Modern scientific and enterprise computing employ complex workflows comprising a set of related simultaneously-running applications that solve different parts of a target problem. Workflows are popular due to benefits such as reusability and extensibility. However, lack of optimal data sharing and communication among the applications poses major challenges to achieve high performance when executing a workflow. Prior work has explored the use of in-memory data stores to facilitate such data sharing. However, in-memory data stores introduce their own challenges such as operational overhead of restarting during a crash and inefficient backup data management, which precludes their use when data to be shared is more than available memory capacity or if data needs to be persisted. We tackle the above problems by presenting *Workflow Data Communicator*, a novel approach that employs a key-value based object data storage system to facilitate persistent and reliable data sharing between applications in a workflow.

1.1.3 Data Management in Multi-layer Storage Hierarchy

Upcoming exascale high performance computing (HPC) systems are expected to comprise multi-tier storage hierarchy, and thus will necessitate innovative storage and I/O mechanisms. Traditional disk and block-based interfaces and file systems face severe challenges in utilizing capabilities of storage hierarchies due to the lack of hierarchy support and semantic interfaces. Object-based and semantically-rich data abstractions for scientific data management on large scale systems offer a sustainable solution to these challenges. Such data abstractions can also simplify users' involvement in data movement. In this work, we take the first steps of realizing such an object abstraction and explore storage mechanisms for these objects to enhance I/O performance, especially for scientific applications. We explore how an object-based interface can facilitate next generation scalable computing systems. Our proposed storage model stores data objects in different physical organizations to support data movement across layers of memory/storage hierarchy.

1.2 Research Contribution

From the above aspects, we demonstrate that we can improve the storage and I/O mechanism in large scale object-based storage systems by providing scalable and efficient data management techniques.

Resource Contention Aware Load Balancing First, we present quantitative study of the load imbalance in large scale object-based systems like Lustre, using two common use-cases: a synthetic I/O benchmark IOR [107] as well as a real-world scientific I/O application kernel HACC-I/O [80]. Our observations stress the need of a resource contention aware load balancing approach for large scale distributed storage systems. Second, we propose a novel end-to-end control plane, `iez`, that combines the application-centric strengths of client-side approaches with the system-centric strengths of server-side approaches. It gathers real time information from clients and servers about the applications' storage requirements as well as the load on storage servers and maps the present and future job requests on OSTs in a balanced manner to provide efficient utilization across a set of servers. The system considers per-client job requests in a dynamic client-wide prediction model to synchronize holistic job placement and resource allocation. Our data placement strategy supports two widely-used classes of application file access, i.e., File-Per-Process (FPP), and Single-Shared-File (SSF) per job request. Third, we extend our load balancing approach `iez` to support multiple striping patterns in the files. Striping in parallel file system enables users to obtain high I/O performance throughput [95]. Progressive file layout (PFL) [111] is a feature in Lustre where a file can have multiple

striping layouts. We evaluated `iez` on a real Lustre testbed on two different scales with IOR and HACC-I/O, for both PFL and non-PFL layout, with multiple stripe counts of files as well as SSF and FPP accesses.

Efficient Data Sharing in Scientific Workflows In this work, we propose a novel approach, *Workflow Data Communicator*, for using object-based storage to facilitate data sharing in scientific workflows. *Workflow Data Communicator* provides data persistence, reduces the operational overhead of using in-memory data store, and facilitates the storage and retrieval of large amount of shared data in an efficient way. *Workflow Data Communicator* enables a persistent back-end storage for Data Broker [136] (a programming model used for data transformations among applications in a scientific workflow) by using Ceph Object Store. It manages the shareable application data in the form of objects and pools (separate logical partitions), thus providing a high level view of namespaces and tuples to the applications, to facilitate data sharing, without any modification to the application or workflows. It overcomes the limitations of in-memory data stores by providing a stable, highly-reliable and scalable persistent object storage. We evaluate *Workflow Data Communicator* on a 8 TB Ceph Object Store cluster, and demonstrate the applicability of *Workflow Data Communicator* on different data access patterns and real-world HPC workflows.

Data Management in Multi-layer Storage Hierarchy We presented a novel object based storage interface to facilitate storage and I/O for HPC applications in exascale systems that will include deep memory and storage hierarchy. Our interface maps scientific applications' data into objects that can store both simple as well as complex data structures along with their properties to preserve semantic information in various layers of memory and storage. Our approach offloads the task of managing MPI-IO [143] or HDF5 [74] calls for data storage from the users to our system. To achieve high performance as well as storage efficiency, we store objects partially into burst buffers or node local persistent storage as well as on disk-based parallel file system (e.g. Lustre) based on application requirements. Our storage model stores the objects to multiple subfiles (instead of single shared file for all processes), which reduces contention. We implement our proposed model with two scientific use cases, VPIC-I/O [52] and HACC-I/O [80] and evaluate it for the applications for a scale of up to $16K$ processes.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter 2 we introduce the background technologies and state-of-the-art related work that lay the foundation of the research conducted in this dissertation. In Chapter 3 we present our solution for resource

contention aware load balancing in large scale storage systems. In Chapter 4, we present the solution for load balancing in Lustre while providing support for the PFL layout. In Chapter 5, we introduce a flexible data sharing approach in scientific workflows by using object-based storage. In Chapter 6, we introduce a transparent data management technique for multi-layer storage hierarchy. Chapter 7 concludes the dissertation.

Chapter 2

Background

This dissertation is focused on applying simple and effective data management strategies for existing HPC object-based storage systems. This chapter summarizes the state-of-the-art research that is closely related to the major themes described above. I also compare them with our work by emphasizing the effectiveness, novelty, and benefits of techniques and algorithms proposed in this dissertation.

2.1 Object-based Storage Systems

We studied a wide range of distributed storage systems and found that object-based storage recently become the most popular. So, we did some literature review for current and past object-based storage.

Object-based storage [28, 29, 77, 142] is a generic term used to describe an abstract data container that consists of multiple byte-streams (or *objects*), each with related attributes [25]. Object-based storage has been implemented for disks, NVRAM, and in memory [31, 173]. However, existing efforts do not integrate objects across the entire memory hierarchy.

We studied the design and implementation of some of the popular object storage systems, and in the following present a brief discussion [151] to highlight the breadth of features and use cases that object-based stores offer.

Lustre: Lustre File System [90] is an object-based, parallel distributed file system. It is mainly used for large-scale cluster computing. It provides high data availability, independence to physical data location, and scalability. These features makes it a suitable choice for general-purpose back-end storage file systems and also to support various scientific applications. It decouples computation and storage and stores data on object-based storage disks (OSTs) and uses metadata target (MDT) to store the metadata. The three

main components of a Lustre file system are: Object Storage Client (OSC), Object Storage Server (OSS), and Metadata Server (MDS). All these components are connected via Lustre Network Layer (LNET).

Ceph: Ceph [159] is an object-oriented distributed file system with a highly scalable design that makes it suitable for scientific applications. As in Lustre, Ceph also separates metadata from data and distributes it over multiple nodes running a software component called `OSD`. The metadata servers in Ceph are clustered and hence the filesystem directory tree is partitioned across the cluster. OSD enables nodes to self-manage object replication, linear scaling in both capacity and performance, cluster expansion, recovery and fault tolerance.

OpenStack Swift: Swift [14] is another widely-used and popular object storage system. Swift is designed to store cloud data such as large binary blobs, image and video files, backups, analytics data, and other unstructured data at large scale in the form of objects, with high availability and durability.

DAOS: Distributed Application Object Storage (DAOS) [88] is the storage system for ongoing Fast Forward [96] project to meet the requirements of exascale computing systems. It makes the bottom layer of fast forward I/O stack. The top layer being HDF5 I/O, and Input/Output dispatcher at the middle layer. DAOS is an extremely simple object model that encapsulates entire exascale data and metadata. It replaces POSIX with transactional, shared-nothing, distributed object store known as DAOS container.

Apache Spark: Spark [1] is an in-memory data processing and cluster-computing framework mainly used for enterprise big data analytics [60]. Spark provides in-memory computations using Resilient Distributed Dataset (RDD) [170] abstraction, which is an immutable distributed collection of objects, for increased speed and data processing over MapReduce[64].

MarFS: MarFS [79] is another object-based file system that presents cloud-style object-storage as scalable near POSIX filesystem. It is a recently developed system and is becoming popular for providing high scalability to POSIX interface in cloud-style object storage, making it quite efficient for HPC applications.

Docker Registry: Docker container images are stored in an online store called Docker registry [5]. Container image itself is a set of layers stored in object storage [35, 36, 105]. Slimmer [174, 175] and DupHunter [172] are recent work that focus on providing a flexible and scalable object storage for Docker containers. Similarly, Infinicache [154] exploits serverless functions to build a cost-effective cache using containers.

These storage systems provide various benefits to manage and store data such as high performance, security, and scalability. But most of these systems are mainly used for cloud computing applications and are optimized for storage of immutable data, and thus may not be directly applicable to many HPC use cases. Lustre and Ceph have been successfully supporting HPC application data at scale, and are most popular and commonly used large scale storage systems in the HPC environment.

2.2 Load Management in Distributed Storage

Recent studies [29, 70, 125, 127, 155] have shown that load imbalance in HPC systems creates resource contention and degrades overall performance. Load management has been incorporated into a number of modern distributed storage system designs [34, 98, 99, 120]. GlusterFS [50] uses elastic hashing algorithm that completely eliminates location metadata to reduce the risk of data loss, data corruption, and data unavailability. However, no load balancing is supported across the storage targets. Ceph [112, 160] uses dynamic load balancing based on CRUSH [161], a pseudo-random placement function. It also adds limited support for read shedding, where clients belonging to a read flash crowd are redirected to replicas of the primary copy of the data.

Previous works [23, 41, 42] present auto-tuning approaches for MPI-IO and Lustre to learn and predict the I/O parameters for improving read and write performance of HPC applications. These approaches use a range for Lustre stripe-counts, I/O buffer sizes and I/O bandwidth of previous runs. In [66], dynamic data migration is proposed to balance the load under various constraints. Machine learning and data mining techniques have also been used for the more general problem of resource allocation that also includes some load balancing. Schaerf et. al. [134] explore the problem space of adaptive load balancing by exploring reinforcement learning techniques. Martinez et. al. [109] explore basic learning techniques for improving scheduling in hardware systems. Game Theory is also used for resource allocation [126]. Federated Learning [45, 69, 94, 139, 141, 144, 168] allows training a model without sharing the data. TiFL [55, 56] is a recent work on mitigating the effect of resource and data heterogeneity to overcome load balancing problems within federated learning. Finally, Google has recently explored the use of machine learning to optimize various system-level metrics [110].

While such techniques show the promise of machine learning for optimizing system parameters, they are orthogonal to our target problem of load balancing in HPC storage systems, and not directly applicable to our use case.

2.3 Data Sharing in Scientific Workflows

Data Broker [136] is a programming model that helps applications share data among themselves using tuples, based on the concept of namespaces. It uses key-value store to facilitate data communication. There is a wide variety of key-value stores available [17, 18, 19, 71, 87, 91, 102]. Most of them provide convenient APIs for applications. However, since they focus on the storage and management of the actual data, there's only minor flexibility in the storage properties. The concept of Data Broker is to leverage what these key-value stores offer and add the flexibility of selecting the right back-end for the desired storage requirements of the data with the goal to cover the spectrum from fast access to small amounts of transient data to highly available persistence of vast data pools [30, 32]. However in-memory key-value stores face challenges in data persistence which our approach *Workflow Data Communicator* aims to mitigate.

Direct intra- and inter-application data sharing is a common method in distributed applications. For example, Sockets, Pipes, or RDMA to directly connect two peer applications or PGAS [44] or MPI [11] to exchange data in a more collective way with more than two processes. These approaches require all participating processes to be active at the time of the data exchange and there's no immediate concept of data persistence or resilience that would be required for a dynamic, complex, and fault-tolerant workflow.

Vairavanathan et al. [147] first came up with the idea of workflow-aware storage systems. AnalyzeThis [137] builds upon this idea to create an analysis workflow-aware storage system by leveraging an array of Active Flash devices [137]. This helps expedite workflow execution and minimizes data movement. However, applications with complex workflows and heavy dependencies cannot be executed using such an approach.

Linda Programming Model [54] provides a simplified, high-level coordination language to help applications in the distributed system scenario to communicate with each other. Our approach uses the Linda Programming Model to simplify communication among applications in scientific workflows on object storage.

Common Component Architecture [40] is an effort to standardize and formulate the communication among heterogeneous applications in HPC environment. It however does not specify any implementation details. Our method uses the standards specified in Chapter 5 to formulate a novel method of data communication in scientific workflows.

Workflow management systems for HPC include Swift/T [167], cylc [4], Fireworks [89], Pegasus [65], TaskFarmer [12], Tigres [7], Ophidea [117], and Kepler [16]. These aid in visualizing workflows but do not optimize I/O.

Object storage is a common type of storage in modern cloud environments. Systems such as Lustre [8, 90], IBM Spectrum Scale [135], and Ceph [3] support both file system and object-based access to storage. While they provide some support for efficient data communication, these are not workflow-aware and need high-level programming paradigms for complex scientific workflows to make efficient use of such large scale parallel file systems and object stores.

2.4 Data Management in Multi-layer Storage Hierarchy

Recently, many other object-based storage systems have been implemented. The T10 standards [72, 113] were proposed to store data and attributes as objects. Based on T10 standards, Seagate, IBM (ObjectStore [67]), and Panasas (PanFS [164]) implemented prototypes [113] and demonstrated the capability of object-based storage systems. HDF5 [74] data model offers three types of objects-groups, datasets, and links between objects and stores them into file-based storage systems. Network Attached Secure Disk (NASD) [76, 77, 86] stores variable-length, logical byte streams as objects. In addition to the data, each block contains attributes for *access control*, *clustering*, *cloning*, *size*, *access time*, etc. PVFS [81] is similar to Lustre, and provides object-based distributed files onto multiple servers. Slab allocation [46, 47] and Memcached [73] use the concept of objects in main memory. NVRAM and various implementations of FLASH devices have also been proposed as solutions to alleviate I/O performance issues of HPC systems. Rajimwale et al. [132] revisited the traditional data model of HDD on SSD devices and found that the object-based storage is more suitable for these new devices. Kang et al. [92] proposed object-based models to support hardware devices with different configurations. Application related information is allowed to be exchanged through the object interface and this provides for performance benefits and object-level reliability. Lee et al. [101] implemented a SSD based object storage system, where the attributes and I/O usage information is stored as metadata. Muninn is an object-based versioning key-value store [93] to enable transparent versioning on file systems. MOANA [27, 53] studies I/O variability in parallel system experimental design.

Despite numerous studies of object-based storage solutions, there is no uniform object management across all the memory and storage layers that will be common in exascale systems. Generally, existing research focuses on an individual level without considering the presence of other layers. When data moves through the hierarchies, semantic information embedded in objects is lost, resulting in poor performance. In addition, object-oriented mechanisms are unexplored for expressing data structures that can transcend through

all the layers of the hierarchy. Our approach addresses this short coming, and offers an integrated solution for deep hierarchy storage for emerging exascale systems.

Parallel I/O We also review some other state-of-art parallel I/O systems which are not object-based. PLFS [43], a parallel log-structured file system, transforms small non-contiguous $N-1$ writes into sequential $N-N$ contiguous files. For providing global view of a shared files, PLFS combines metadata from each individual process. ADIOS [106] is a high-level I/O library that provides adaptability and helps improving parallel I/O performance with I/O re-routing, that can be achieved by changing entries in an XML file. Gao et al. presents a subfiling interface [75] to store multidimensional global arrays into smaller subarrays. To reconstruct the global array, this interface is incorporated in parallel netCDF library [103]. To improve performance for hybrid parallel file systems, HARL [83] divides a file into regions and replicates each region with appropriate file stripe sizes on servers in a cost effective manner. Several other studies have explored the problem of parallel data placement [21, 63, 118, 122, 123, 138], data monitoring [62, 119, 121] data partitioning [97, 128] and Virtual Machine distribution [124, 126, 148, 149, 150] in data centres. Wang et al. presents a distributed file system for management of node local burst buffers, BurstFS [156], that has the same lifetime as the batch job, and can be shared by multiple applications at the same time. LACIO [58] is a collective I/O strategy that integrates the underlying physical data layout information to match the partitioning of file domains. PDLA [169] is another layout aware scheme for parallel I/O that replicates data access patterns with appropriate data layouts.

These systems aim to improve parallel I/O similar to our approach, however, existing efforts do not consider multiple storage layers and I/O path that is an important factor to meet exascale I/O requirements. Our API provides a unified storage approach to handle multiple layers of storage transparently

Chapter 3

Resource Contention Aware Load Balancing

3.1 Introduction

Load imbalance and poor resource allocation have been identified as major causes of performance degradation in many HPC storage systems, including Lustre [90], the widely-used parallel file system for scientific computing. As discussed in Chapter 1, a typical Lustre deployment consists of a large number of distributed Object Storage Targets (OSTs) that are managed by Object Storage Servers (OSSs). When application data is written to a file, an associated OSS maps the file into multiple objects that are striped into different OSTs. The striping is done in a way to yield efficient parallel access. Extant practice is to use round-robin scheduling to place objects on the OSTs. However, this often leads to a load imbalance across the OSTs given typical non-uniform data accesses and contention for OST resources containing "hot" objects, consequently affecting the I/O performance and thus the overall application performance. What is needed is a *contention-aware placement of application data to the various resources to yield improved load balancing and high performance* for large-scale parallel file systems in HPC.

A number of recent works [70, 116, 120, 155] have targeted load imbalance in Lustre. Some aim to simultaneously perform resource allocation for all concurrently running applications on the system (i.e., server-side approaches [70, 120]). These include the default approach adopted in Lustre's request ordering system, the Network Resource Scheduler (NRS) [131], which reorders incoming I/O requests on the server-side. Other techniques attempt to minimize resource contention on per-application basis (i.e., client-side approaches [116, 155]). Unfortunately, while client- and server-side approaches in isolation work well for some applications, the diversity of I/O workloads (e.g., large sequential writes from many clients versus scientific data processing on a large data set for a single application) leads to situations where both isolated approaches suffer reduced performance.

We propose a novel end-to-end control plane, `iez`, that combines the application-centric strengths of client-side approaches with the system-centric strengths of server-side approaches.

1. We introduce an end-to-end control plane, `iez`, to optimize I/O resource allocation in existing as well as next generation HPC systems that use Lustre for their storage platform.

2. We present the detailed design of `iez` for the Lustre file system, which incorporates both server side and client side functionality to optimize I/O data placement and job placement.

3. We implement and evaluate the effectiveness of `iez` on a cluster with two commonly used use cases: synthetic I/O benchmark IOR [107] and a scientific application I/O kernel HACC-I/O [80]. Results show that `iez` improves I/O performance by up to 34% compared to the extant round-robin based load balancing adopted in Lustre.

3.2 Background and Motivation

Recent studies [29, 70, 125, 127, 155] have shown that load imbalance in HPC systems creates resource contention and degrades overall performance. The complex path of an application I/O request—comprising myriad components such as I/O libraries, network resources, and backend storage—is a significant bottleneck. In today's HPC deployments, there is no global I/O coordinator to handle the overall resource contention problem. Thus, existing parallel file and storage systems can only partially optimize some portions of the I/O path, but not the entire end-to-end path. For example, *Network Request Scheduler* (NRS) [131] balances load in the distributed network of Lustre file system, i.e., on the server side; whereas *Balanced Placement I/O* (BPIO) [155] performs load balancing on per-application basis, i.e., on the client side.

3.2.1 Lustre Architecture

We have implemented `iez` atop Lustre, the parallel file system deployed most widely in the world's top supercomputing systems [20]. Lustre is a scalable storage platform that is based on distributed object-based storage. Figure 3.1 shows a high-level overview of the Lustre architecture and its key components. Lustre clients provide an interface between

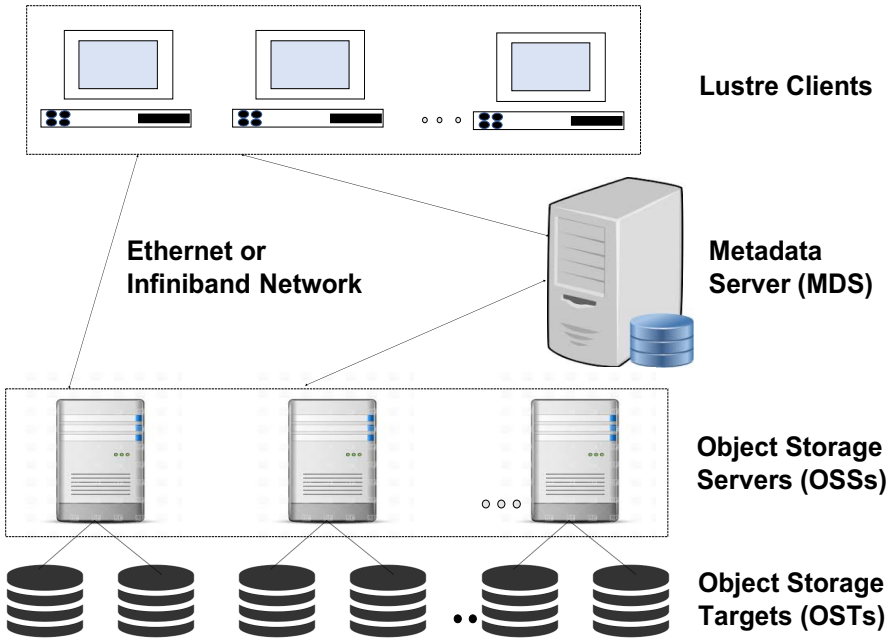


Figure 3.1: Overview of Lustre architecture.

applications and the storage servers. The application data is managed by two types of servers, MDS and OSS. MDS manages all name space operations and stores the name space metadata on one or more storage servers called Metadata Targets (MDT). The bulk storage of contents of application data files is provided by OSSs. Each OSS manages a number of OSTs and stores the data on one or more OSTs. OSTs are a kind of direct-attached storage. Each data file can be striped across multiple OSTs, with the stripe count specified by the user. The distributed components are typically connected via a high-speed data network protocol, LNet [114], which supports a host of networks, e.g., Ethernet, Infiniband [130], etc.

3.2.2 Approaches for Load Balancing in Lustre

Given typical non-uniform data accesses patterns, the striping of application data across OSTs give rise to load imbalance in most cases. Currently, the default OST load-balancing approach adopted in Lustre is round-robin (RR), also known as RAID 0. The main limitation is that, RR aims to balance the load of OSTs only, i.e., without any consideration to the load on other components, e.g., MDS and OSS. Moreover, our earlier work on quantitatively studying HPC I/O behavior [120] has shown that RR can take a long time to balance a system, and it is unable to capture the complex application behavior. Consequently, the default policy falls short of providing the desired I/O balanced system.

I/O load balance in scalable parallel file systems is being studied extensively [25]. One approach is to address the problem from the client side on per job basis [30, 31, 151, 173]. The applications' I/O calls can be intercepted from client side during runtime and the OSTs assignment can be managed accordingly to mitigate resource contention [84, 104, 145, 176]. An example of such an approach is TAPP-I/O [116]. TAPP-I/O intercepts file I/O calls (metadata operations) during runtime, supports both statically and dynamically linked applications, and provides an automatic placement strategy for both FPP and SSF I/O modes. However, the main limitation is that these approaches do not consider the requirements of other applications running simultaneously on the system due to lack of a global view of the storage servers.

Conversely, another approach is to have a global view of storage servers and consider the load balance across all applications instead of per-application basis. For this, the load balancing problem can be handled from server side [70, 120, 140, 171]. The main limitations of such approaches are that they require modification of application source code, and do not consider the different file I/O layouts (SSF or FPP).

The above approaches improve the I/O performance of Lustre by effectively reducing the resource contention and improving load balance, but fail to exploit the opportunity for end-to-end I/O path optimization. In contrast, `iez` aims to provide an end-to-end load balancing solution, which globally coordinates between clients and servers of parallel file and storage systems.

3.2.3 Use Cases and Benchmarks

We use a representative synthetic I/O benchmark *InterleavedOrRandom* (IOR) [107] as well as a real-world scientific I/O application kernel *Hardware Accelerated Cosmology Code* (HACC-I/O) [80] for presenting the design and implementation of `iez`. IOR performs reads and writes to and from files on parallel file systems like Lustre, and provides the throughput rates. HACC-I/O utilizes the MPI-I/O interface and differentiates between FPP and SSF parallel I/O modes.

3.2.4 Load Imbalance in a Default Lustre Deployment

In order to highlight the load imbalance in a default Lustre deployment, we conducted a quantitative study that uses RR to distribute I/O load on OSTs. We deployed a testbed of 6-node Lustre cluster, with 1 MDS, 3 OSSs and 2 Clients. Each OSS in our cluster

manages 5 OSTs with a capacity of 10 GB each. Hence, the cluster has 15 OSTs in total with a total capacity of 150 GB. We ran IOR benchmark with 8, 16, and 64 processes that produce 16, 32, and 64 GB of data, respectively, to be stored on the OSTs in the FPP access mode. We also ran HACC-I/O application for 8, 16, and 32 processes for 10 M particles.

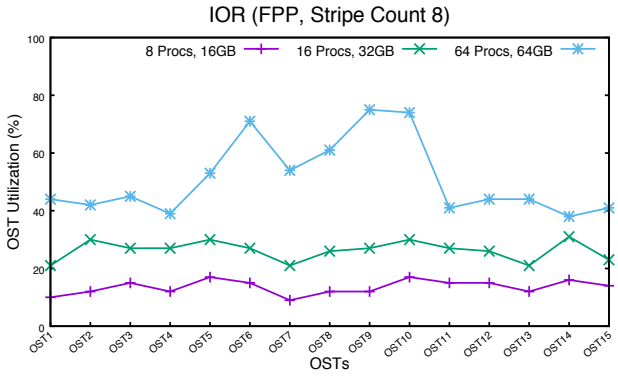


Figure 3.2: OST Utilization under RR for IOR.

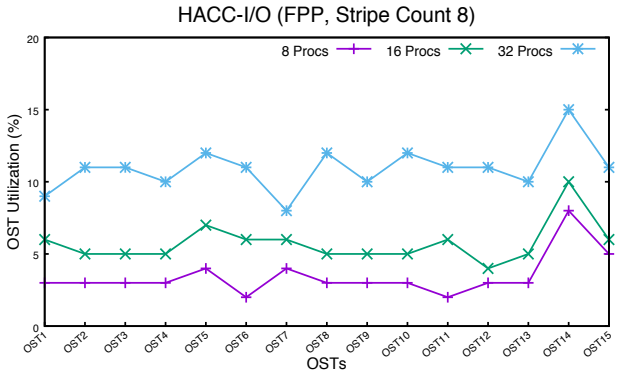


Figure 3.3: OST Utilization under RR for HACC-I/O.

Figure 3.2 shows the storage utilization in each OST, for different number of processes and a stripe count of 8 for IOR in FPP mode. In a balanced load setting, these graphs would be straight lines (representing ideal load balance), but in the studied scenario, the load is observed to be imbalanced with some OSTs getting a lot more load than others. A similar pattern can be seen in Figure 3.3 for HACC-I/O for FPP mode with stripe count of 8.

These results show that with the default Lustre deployment that uses RR scheduling to allocate OSTs for each job, there can be a significant load imbalance at the server level. The load imbalance persists at different scales and different stripe-count and thus can lead to imbalanced resource usage and resource contention.

3.3 System Design

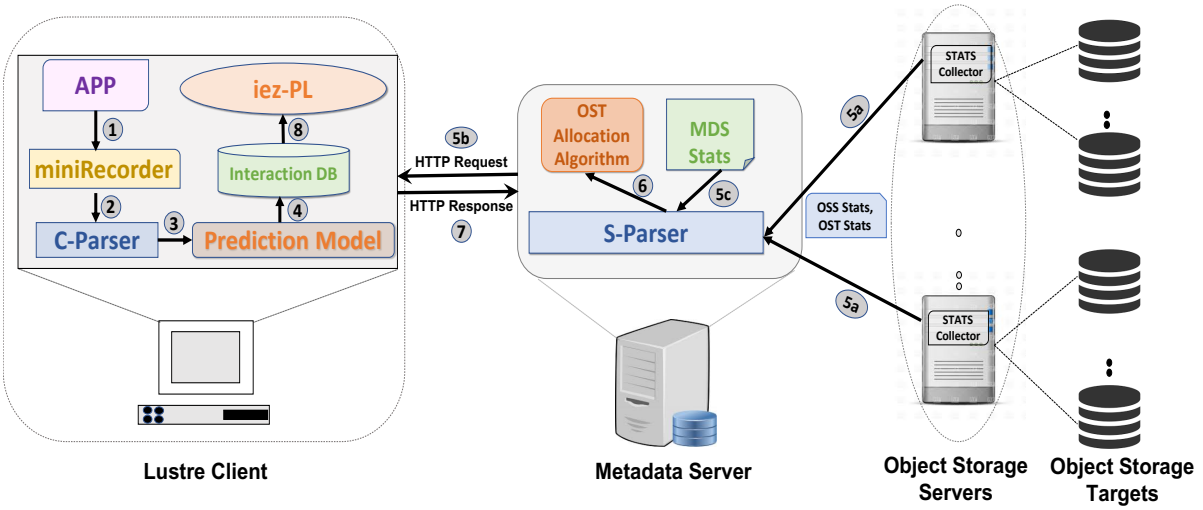


Figure 3.4: Overview of the proposed `iez` architecture.

We have implemented `iez` for the widely-used Lustre file system. However, our design can be extended for use in other HPC distributed storage and I/O systems that employ a similar hierarchical structure.

Figure 3.4 shows an overview of the `iez` architecture. It presents an "end-to-end" control plane for managing I/O with components both on the client side and the server side. When running applications for the first time, the client side makes use of a customized tracing tool, `miniRecorder`. This tool collects information about the I/O accesses, such as the number of bytes written, file name, number of stripes, and MPI rank and communicator for each file. `MiniRecorder` needs to collect traces only for the first run of an application. `iez` identifies an application's I/O behavior, which does not change across multiple runs of an application. The collected traces are fed into the Client (C)-Parser that then uses the information to drive the prediction algorithm. Our predictions are based on ARIMA time series modeling [51]. The output of the time series prediction provides estimates of future application requests, which are stored in an "interaction database" for later use by `iez`. We refer to the database as "interaction database" because it offers a point of interaction between our server-side and client-side libraries.

On the server side, OSSs collect the CPU and memory usage information, associated OSTs capacity (*kbytestotal*) and the number of bytes available on the OSTs (*kbytesavail*). These statistics are sent to the MDS. The collected information on the MDS is parsed using Server (S)-Parser and fed to the OST allocation algorithm. The input to the OST allocation algorithm is the predicted set of requests received from the clients via HTTP-based interactions, and the output is the OSTs to be allocated for every request, which

Table 3.1: Interaction Database Snapshot showing write requests for HACC-I/O in FPP mode.

File Name	Write Bytes	Stripe Count	MPI Rank
/lustre/scratch/hacc-io/FPP1-Part00000000-of-00000008.data	803405824	8	0
/lustre/scratch/hacc-io/FPP1-Part00000001-of-00000008.data	803405824	8	1
/lustre/scratch/hacc-io/FPP1-Part00000002-of-00000008.data	803405824	8	2
/lustre/scratch/hacc-io/FPP1-Part00000003-of-00000008.data	803405824	8	3
/lustre/scratch/hacc-io/FPP1-Part00000004-of-00000008.data	803405824	8	4
/lustre/scratch/hacc-io/FPP1-Part00000005-of-00000008.data	803405824	8	5
/lustre/scratch/hacc-io/FPP1-Part00000006-of-00000008.data	803405824	8	6
/lustre/scratch/hacc-io/FPP1-Part00000007-of-00000008.data	803405824	8	7

will yield a load-balanced distribution over the involved OSSs and OSTs. The allocated OSTs are stored in the interaction database along with the predicted requests. Next, the placement library, `iez-PL`, intercepts the I/O requests from the applications, consults the interaction database, and routes the application requests to appropriate resources.

1. Client-Side `iez` Components

In the following, we describe the `iez` components that run on the clients.

a) Tracing Tool

We implement a simple I/O tracing library, `miniRecorder`, based on Recorder [108]. Recorder is a multi-level I/O tracing framework, which can capture I/O function calls at multiple levels of the I/O stack, including HDF5, MPI-IO, and POSIX I/O. For our end-to-end system, we limit the range of intercepted function calls to file creation and write calls, and record the bytes written, file name, stripe count, and MPI rank and communicator for each file. We focus on writes more than reads because caching mechanisms and burst buffers that are typical in modern HPC deployments absorb most of the read requests once the file has been written. Therefore, the load imbalance is mainly due to write requests [108, 120]. The traced data is processed by the *C-Parser* and converted into a readable (comma separated) *.csv* format file. This file is then sent to the prediction library (AIPA- ARIMA Inspired Prediction Algorithm discussed in the next section). The tracing tool is lightweight, and our tests show that it adds a negligible memory and CPU overhead of $\sim$0.3% and $<$1%, respectively, during application execution.

b) ARIMA-Inspired Prediction Algorithm (AIPA)

HPC applications have been known to show distinct I/O patterns [120]. Based on our interactions with HPC practitioners, this predictability is expected for emerging applications as well. We leverage this observation to model three key properties of HPC I/O: write bytes, stripe count, MPI rank. We collect these parameters using the tracing tool

on the first run of an application to train our model. We use AutoRegressive Integrated Moving Average (ARIMA) model to fit our multivariate time series data and predict future values. Our choice of ARIMA is dictated by its performance and the time-series nature of the write bytes and stripe count of the requests. We experimented with several alternatives, such as the Markov Chain Model [133], to model the data. However, ARIMA yielded better accuracy with lower memory and computing overhead. For example, we observed a 99.1% accuracy in IOR data using ARIMA, while Markov chain model yielded an accuracy of 95.5%. The CPU overhead for ARIMA was less than 1.2% compared to 4.5% in Markov Chain, while the memory usage for ARIMA is 10 MB in comparison to 90 MB usage in Markov chain.

We implement our prediction model on the client side, where the calls would be intercepted, rather than on MDS. This has two advantages: i) since each application has its own client, the model can be applied at scale without overwhelming the centralized MDS; and (ii) the approach makes the MDS application-agnostic, where the server can focus on write requests and types in a global fashion and not be concerned with individual applications. The approach also provides for a much more efficient solution when multiple applications run on (multiple) clients simultaneously.

A time series is defined as a sequential set of data points, measured typically over successive times. It is represented as a set of vectors $x(t)$, $t = 0, 1, 2...$, where t is the elapsed time [49]. The term `ARIMA` involves three parts, `AR` denotes that the variable is regressed on its prior values, `I` stands for 'integrated', which means that the data values are replaced with the difference between the present and previous values, and `MA` represents the fact that the regression error is a linear combination of error terms occurring in the past. There are three parameters used for every ARIMA model. The parameter `'p'` is the number of lag observations (lag order), `'d'` denotes the number of times raw observations are differenced (degree of differencing), and `'q'` represents the size of moving average window (order of moving average).

The first step is to select the values of parameters `(p, d, q)`. To this end, we run the model on all combinations (skipping the ones that fail to converge) of the parameters over our dataset, which we get from the tracing tool, and select the combination with the least Root Mean Square Error (RMSE). We vary the values of p, d, and q from 0 to 5. We go from 0 to 5 for all the values because going beyond 5 would be computationally expensive. For *HACC-IO*, the least RMSE was found for `(5, 1, 2)` and *IOR* gave the minimum RMSE for `(2, 1, 1)`. The next step is to fit the ARIMA`(p, q, d)` model by exact maximum likelihood via Kalman filters [82]. This fitted model is then used to predict the write bytes, stripe count, and MPI rank values for future application I/Os. We use `statsmodels.tsa.arima_model` package in Python for our ARIMA implementation. Our results show a 98.3% accuracy in HACC-I/O data and 99.1% accuracy in IOR data.

Algorithm 1: File layout and Lustre inode creation.

```
Input: File Name file, Access mode flags
Output: Call real metadata operation, e.g., open()
1  begin
2      if fileExits(file) == TRUE then
3          return realMetadataOperation(file, flags)
4      r = queryInteractionDatabase(file)
5      layout→scount = row → stripe_count
6      layout→stripe_size = row → stripe_size
7      layout→ost_list = row → ost_indices
8      createInode(layout, flags, mode)
9      return realMetadataOperation(file, flags)
10 Function createInode
       Input: File Layout layout, Access mode flags
11     mode = 0644
12     flags = flags | O_LOV_DELAY_CREATE
13     request = LL_IOC_LOV_SETSTRIPE
14     fd = __real_open(file, flags, mode)
15     ioctl(fd, request, placement)
16     close(fd)
```

c) Interaction Database

The interaction database is an SQL database located on the Lustre clients. It serves as the medium through which the MDS and clients interact with one another. First, the values predicted by AIPA are stored in the database. Table 3.1 shows an example snapshot of the interaction database for HACC-I/O in FPP mode. We store the file names, the number of bytes written, number of stripes associated with every file, and the MPI Rank. The MDS uses the HTTP protocol to access the interaction database. The process starts with the MDS sending a HTTP Request to the client to retrieve the required database contents. For our implementation, we use MySQL 8.0.12 Community Server Edition. Our results show that writing and retrieving data from the interaction database is very efficient, using <0.3% and <0.4% of CPU and memory, respectively.

d) Placement Library The placement library, `iez-PL`, complements the prediction model by providing a lightweight and user-friendly mechanism to place an application's I/O workload in accordance with the predicted set of OSTs. `iez-PL` utilizes function interposition to prioritize itself over standard function calls, and the profiling interface to MPI (PMPI) for MPI and MPI-IO routines. `iez-PL` is implemented as a shared, dynamic user library, and can be used by specifying it as the preloading library via the environment variable `LD_PRELOAD`. The POSIX I/O, MPI-IO, and HDF5 metadata operations (e.g., `open()`), which are issued by the application, are intercepted and re-routed to `iez-PL` for processing purposes. For every I/O cycle, the library queries the interaction database with the file name passed by the function call and fetches the corresponding row with the predicted placement information. This information includes the stripe count, stripe size, and a list of OST indices. Next, `iez-PL` creates a Lustre inode on the MDS and places the stripes on the OSTs as returned by the prediction model. For this pur-

pose, Lustre provides the user library *llapi*, which allows the user to describe a specific file layout, i.e., the striping pattern. However, `iez-PL` cannot directly take advantage of the user library, since *llapi* calls `open()` internally, which means that the preloading mechanism would cause an endless loop. Therefore, `iez-PL` mimics the behavior of *llapi* and communicates directly with the Lustre Logical Object Volume (LOV) client software layer to create the Lustre inode. Algorithm 3 presents a simplified overview of the preloading library, which is run on every client. If the file does not exist yet, it queries the prediction database to receive the predicted layout information. The next step is to create the Lustre inode by issuing the `ioctl()` system call with the Lustre-specific request code `LL_IOC_LOV_SETSTRIPE`, which allocates a Lustre file descriptor and applies the striping pattern to the file-to-be-created. In the last step, `iez-PL` forwards the call to the original metadata operation (e.g., `open()` or `MPI_File_open()`). Currently, `iez-PL` supports the following I/O function calls: `open[64]()`, `creat[64]()`, `MPI_File_open()`, and `H5Fcreate()`. The key advantage of this transparent approach is that user applications can directly benefit from the prediction model without modifying the source code.

2. Server-Side `iez` Components

In the following, we describe the `iez` components that run on the servers (OSSs and MDS) and how they interact with each other.

a) Statistics Collection

Statistics collection is done for every OSS. The list of all the OSTs for a particular OSS is saved in a configuration file that is provided as input to the collector for that specific OSS along with the OSS id. For every OST, we collect the total and available capacity, found in the files $/proc/fs/lustre/obdfilter/ost\_name/kbytestotal$ and $/proc/fs/lustre/obdfilter/ost\_name/kbytesavail$, respectively. We also collect the CPU and memory utilization of the OSS by reading data from the files $/proc/meminfo$ and $/proc/loadavg$. We save all the statistics in a space separated string *statistics*. The statistics collection algorithm runs every 60 seconds on all the OSSs. We choose 60 seconds as our interval for statistics collection so that we get updated statistics on the MDS without over-loading the OSSs. These statistics are sent to the MDS.

The load monitoring (statistics collection) solution needs to be scalable [26, 33, 37, 38, 39]. Therefore, we use a publisher-subscriber model [121] for the statistics collection framework. This is shown in Figure 3.5. OSSs act as publishers and MDS as the subscriber. Statistics collected in the OSSs are sent to the MDS via a message queue. We use ZeroMQ (ϕMQ) [85] as our message queue because it is lightweight and has been shown to be very efficient at large scale. We also collect the CPU and memory utilization of the MDS every 60 seconds, in the same way as it is collected in the OSSs. Our tests with the

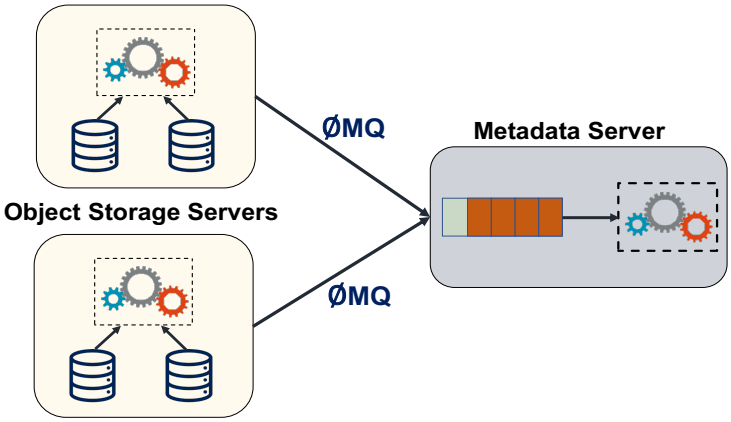


Figure 3.5: Publisher-Subscriber model for statistics collection.

implementation show that the statistics collection framework on average has negligible CPU and 0.1% memory utilization on the OSSs, and 0.6% CPU and 0.1% memory on the MDS.

b) Statistics Parser

The *S-Parser* in the MDS is responsible for handling the following:

- Statistics collected from the MDS.
- Statistics collected in the OSSs and published to the MDS via ZeroMQ.
- Current and predicted write requests retrieved from the clients via HTTP.

The CPU and memory utilization collected in the MDS is important to determine when the OST allocation algorithm will run. We allow the allocation algorithm to run only if the CPU utilization is lower than 70% and memory utilization is lower than 50%. This is done so that the load balancing algorithm does not disrupt the normal functionalities of Lustre's MDS. The parsed statistics of the OSS and the write requests from the clients are sent as input to the OST allocation algorithm (explained next). Parsing is done whenever unparsed data arrives at the MDS, therefore should be done very efficiently. Our results show that the parser has a CPU utilization of 0.1% and negligible memory utilization.

c) OST Allocation Algorithm

The OST allocation algorithm runs only when the CPU utilization goes below 70% to ensure no interruption to Lustre's default activities. The input to the algorithm is the

parsed OSS and OST statistics, and the write requests (file name, write bytes and stripe count) sent by the clients. Before we present the details of our algorithm, we discuss a key practical constraint imposed by Lustre.

Lustre's 64k alignment constraint for stripe size Stripe size is an important parameter for load balancing in a distributed file system. Every stripe needs to be assigned to an OST. Intuitively, in order to calculate stripe size, we can divide the total file size by the stripe count. Both of these parameters are given as input to the OST allocation algorithm. But calculating stripe size is not that simple because Lustre imposes the constraint that in order to place stripes into the allocated OSTs, stripe size should be even multiples of 64k. We term 64k or 65536 bytes as Alignment Parameter (AP). This constraint becomes a problem for files which are not AP aligned, for example in Table 3.1, $803,405,824$ is not an even multiple of AP. We consider two ways to overcome this constraint.

Method-I: This method assumes that we can allocate equal number of even multiples of AP into the first *(stripeCount - 1)* number of OSTs, and the remaining even multiple of AP goes into the last OST. Equation 3.1 gives the total allocation such that the stripes are AP aligned. The first part of the equation is the placement on first *(stripeCount - 1)* number of OSTs and the second part is the number of bytes written on the last OST.

$$\begin{aligned} writeBytes = (AP * 2 * N * (stripeCount - 1)) + \\ (AP * 2 * X) \end{aligned} \tag{3.1}$$

where,

$$N = \left\lfloor \frac{writeBytes}{AP * 2 * (stripeCount - 1)} \right\rfloor \tag{3.2}$$

The remaining number of bytes to be written on the last OST is then given by:

$$\begin{aligned} remainingBytes = writeBytes - \\ (AP * 2 * (stripeCount - 1)) \end{aligned} \tag{3.3}$$

Therefore,

$$X = \left\lceil \frac{remainingBytes}{AP * 2} \right\rceil \tag{3.4}$$

Note that we round down the allocation in the first *(stripeCount - 1)* number of OSTs and round up the allocation in the last OST. Multiplication with 2 ensures that the stripe size is an even multiple of AP. Stripe size for the last OST is $(AP * 2 * X)$, and for each of the remaining OSTs is $(AP * 2 * N)$. However, a major drawback here is that this method does not place the load evenly among all the OSTs. The number of bytes written on the

last OST will always be smaller compared to the bytes written on the other OSTs for a particular file.

Method-II: This method overcomes the drawback of the previous method by allocating even multiple of AP in all the OSTs.

$$writeBytes = AP * 2 * N * stripeCount \tag{3.5}$$

where,

$$N = \left\lceil \frac{writeBytes}{AP * 2 * stripeCount} \right\rceil \tag{3.6}$$

Stripe size for all the OSTs is given by $(AP*2*N)$. Therefore, both methods ensure an even multiple of 64k alignment of stripe size for all the stripes allocated in the *stripeCount* number of OSTs, by allocating a little bigger file than was requested by the client. The second method places all the stripes equally on all the OSTs but needs a bigger file to be allocated in comparison to the first method. Thus, there is a trade-off between how big the file allocation we can allow versus balancing all the stripes among the OSTs. Our results show that for a 766.175 MB file size, we allocated 833 KB (0.1%) bigger file using the second method and 63 KB (0.008%) more in the first method. We proceed with the second method because in spite of allocating a little more than was requested by the client, this approach ensures allocating equal stripes on all the OSTs. This would lead to similar load accesses from all OSTs and OSSs, therefore approaching towards a load-balanced setup.

Algorithm 2 shows the the OST allocation algorithm, which employs a minimum-cost maximum-flow approach [24]. The flow graph that is used to solve the problem is shown in Figure 3.6. We calculate the *stripeSize* based on Method-II described above, cost to reach an OST (which is the load of the OSS), cost of an OST (ratio of bytes already used in the OST to the total size of the OST), and capacity of an OST (the number of stripes that can be handled by the OST, given by the ratio of available space in the OST to the stripe size). In order to derive the flow graph, we need to identify the *source* and *sink* nodes. The total demand for the *source* node is the total number of stripes requested, and the total demand for the *sink* node is the negative amount of the total number of stripes requested. We solve the minimum-cost maximum-flow using the Ford-Fulkerson algorithm [146]. This outputs a list of OSTs (*OSTAllocationList*) using which will yield a balanced load over both OSS and OSTs. For our implementation, we use the `networkx` library in Python. Our results show that the algorithm on average uses 1.58% CPU and 0.1% memory on the MDS.

Algorithm 2: Obtaining list of OSTs for each request.

```
Input: OSS statistics cpu & mem, OST statistics totalKbytes & kbytesAvail, Write Requests writeBytes &
       stripeCount
Output: OSTAllocationList
1  begin
2      for request r in WriteRequests do
3          stripeSize = calculateStripeSize(writeBytes, stripeCount)
4      for OSS oss in OSSList do
5          ossLoad = (cpuweight * cpu) + (memweight * mem)
6          for OST ost in OSTList do
7              ostCostToReach = OSSLoad
8              ostCost = (totalKbytes − kbytesAvail)/totalKbytes
9              ostCapacity = kbytesAvail/stripeSize
10     flowGraph = buildGraph(Requests, OSS, OST)
11     OSTAllocationList = minCostMaxFlow(flowGraph)
12     return (OSTAllocationList)
13 Function calculateStripeSize
       Input: writeBytes w, StripeCount sc
       Output: stripeSize
14     stripeSize = ceil(w/(AP * 2 * sc))
15     return (stripeSize)
16 Function buildGraph
       Input: Requests req, StripeCount sc, OSTCostToReach ossLoad, ostCost ostLoad, OSTCapacity ostCap
       Output: FlowGraph G
17     totalDemand = sum of stripeCount for all Requests
18     G.addNode('source', totalDemand)
19     G.addNode('sink', -totalDemand)
20     for request r in reg do
21         G.addEdge('source', r, cost = 0, capacity = sc)
22         for OST ost in ostList do
23             G.addEdge(r, ost, cost = ossLoad, capacity = 1)
24     for OST ost in ostList do
25         G.addEdge(ost, 'sink', cost = ostLoad, capacity = ostCap)
26     return (G)
```

The list of OSTs obtained from the OST allocation algorithm, along with the *stripeSize*, are then sent to the respective clients where they are stored in the interaction database. An example entry for the database with the complete allocation for a HACC-I/O application in FPP mode is shown in Table 3.2. We replace the *Write Bytes* in the database with the *stripeSize* and add a new column *OST List*. The *OST List* is a space separated load-balanced list of OSTs for every write request. This example is for a setup with 3 OSSs and 15 OSTs (5 OSTs associated with every OSS) – therefore, OST ids range from 1 to 15. As described earlier, the placement library (`iez-PL`) then uses this information to place the requests, thus completing the load-balanced allocation of resources. If for any run of the application, `iez-PL` is unable to find more than 50% of files in the interaction database, `miniRecorder`, AIPA and OST Allocation Algorithm will be executed again to update the interaction database.

Summary: `iez` components run both on the client and server side of a Lustre deployment. `MiniRecorder` along with AIPA runs on the clients during the first execution of an application (or if the system cannot find prediction information for most of the accessed

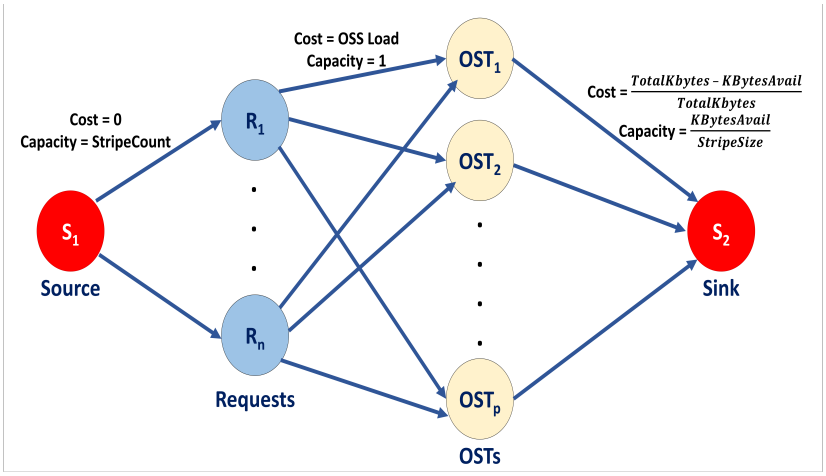


Figure 3.6: Graph used in OST Allocation Algorithm.

files) to capture its I/O characteristics. The client and server libraries of `iez` interact using the Interaction Database. Statistics collection on the OSS and MDS happens periodically. Whenever, the interaction database is updated, it sends the new values to the MDS, which (if not overloaded) proceeds to parsing the most recent statistics via the Statistics Parser. The OST Allocation Algorithm is then executed and the results are sent back to the interaction database. For subsequent application runs, the placement library intercepts the application write calls, reads from the interaction database and writes the request to a load-balanced set of OSTs.

Table 3.2: Interaction Database Snapshot showing OST Allocation for HACC-I/O in FPP mode.

File Name	Write Bytes	Stripe Count	MPI Rank	OST List
/lustre/scratch/hacc-io/FPP1-Part00000000-of-00000008.data	100532224	8	0	8 13 12 9 7 4 3 5
/lustre/scratch/hacc-io/FPP1-Part00000001-of-00000008.data	100532224	8	1	11 2 15 10 1 6 8 13
/lustre/scratch/hacc-io/FPP1-Part00000002-of-00000008.data	100532224	8	2	12 9 7 4 3 11 2 5
/lustre/scratch/hacc-io/FPP1-Part00000003-of-00000008.data	100532224	8	3	1 6 15 10 8 13 12 9
/lustre/scratch/hacc-io/FPP1-Part00000004-of-00000008.data	100532224	8	4	7 4 3 11 2 1 6 5
/lustre/scratch/hacc-io/FPP1-Part00000005-of-00000008.data	100532224	8	5	15 10 8 13 12 9 7 4
/lustre/scratch/hacc-io/FPP1-Part00000006-of-00000008.data	100532224	8	6	3 11 2 1 6 5 15 10
/lustre/scratch/hacc-io/FPP1-Part00000007-of-00000008.data	100532224	8	7	14 8 13 12 9 7 4 3

3.4 Evaluation

We evaluate `iez` using a real Lustre deployment testbed. We use a Lustre cluster of 6 nodes with 1 MDS, 3 OSSs and 2 clients. All of the nodes run CentOS 7 atop a machine with 8 cores, 3.2 GHz processor, and 16 GB memory. Furthermore, each OSS has 5 OSTs, each supporting 10 GB of attached storage. Our tests use HACC-I/O kernel and IOR benchmark in FPP and SSF access modes.

To the best of our knowledge, `iez` is the first work to consider a global view of all the system resources in deciding application request stripe placement. Existing approaches (BPIO [155], TAPP-I/O [116] etc.) balance load among I/O servers on per-application basis, i.e., on the client side, and do not consider the global view, i.e., the requirements of the other applications as well as the OSS and OST states. Moreover, unlike `iez`, such client-side techniques cannot handle multiple simultaneous applications. Thus, these are not directly comparable to `iez`. For these reasons, we choose to use the default RR approach of Lustre as the basis for our comparison.

To capture the degree of load balancing across participating OSTs for a particular test run, we define a metric, *OSTCost*, as the ratio of the maximum utilization of any OST to the mean utilization of all the OSTs. Therefore,

$$OSTCost = \frac{MaxOSTUtil}{MeanOSTUtil} \tag{3.7}$$

An ideal load balanced system has the *OSTCost* of 1. We also define *OST Utilization* of an OST as the storage used by the client application on the OST as a fraction of the total storage available on the OST.

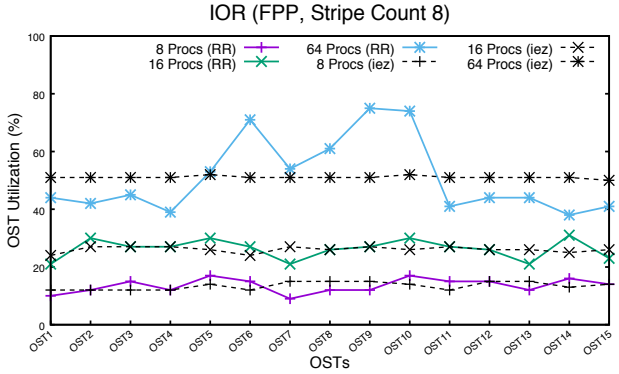


Figure 3.7: OST Storage Utilization for IOR in FPP mode and Stripe Count 8.

Load Balance for FPP Access IOR Figure 3.7 shows the comparison of load under `iez` and the default RR data allocation on 15 OSTs in the Lustre cluster represented as *OST Utilization* for the IOR benchmark with varying stripe count, processes, and data sizes. We see that `iez` balances the load on all OSTs in a near-optimal manner. For example, for 64 processes, the maximum load observed with RR approach is on OST-9: the OST utilization is 75%, while the mean utilization is 51.06%, resulting in the *OSTCost* of 1.47. In contrast, the maximum load observed under `iez` is 52% on OST-5 and OST-10 with the corresponding *OSTCost* of 1.01, i.e., near optimal. The almost horizontal line

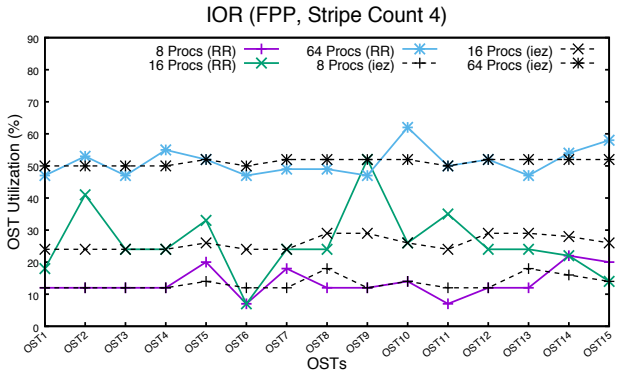


Figure 3.8: OST Storage Utilization for IOR in FPP mode and Stripe Count 4.

for *OST Utilization* for `iez` underscores its effectiveness. Overall, `iez` was able to reduce the *OSTCost* by 31.3% compared to the default RR approach.

We observed similar results while running IOR with stripe count of 4. As shown in Figure 3.8, `iez` is able to distribute data on all 15 OSTs in a balanced way for other studied cases as well. In this case, we observe an *OSTCost* of 1.22 and 1.01 under RR and `iez` respectively. Hence, `iez` provides 17% better *OSTCost* than the default RR approach.

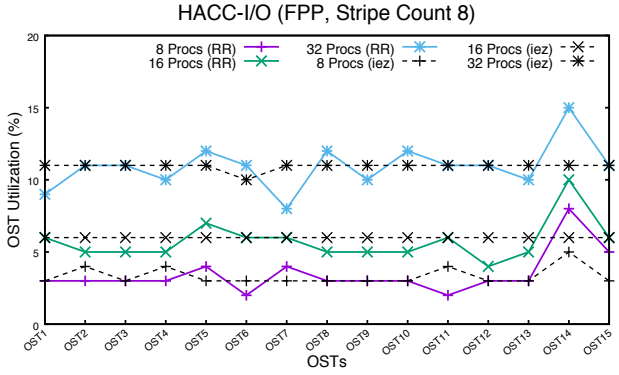


Figure 3.9: OST Storage Utilization for HACC-I/O in FPP mode.

Load Balance for FPP Access HACC-I/O

For HACC-I/O, we evaluated `iez` for particle data of 10 M for 8, 16 and 32 processes. Each process creates one data file that is stored in the Lustre OSTs with a stripe count of 8. Total data stored in the files is approximately 4, 7.7, and 13.5 GB for 8, 16, and 32 processes, respectively. As for IOR, we observe a significant improvement in load balancing for HACC-I/O as well (shown in Figure 3.9) compared to the default approach. *OSTCost* for 32 processes with default load balancing approach and `iez` is observed to

be 1.37 and 1.00, respectively, with `iez` reducing the OSTCost by 27 %. Similarly, for 16 processes, the *OSTCost* is observed to be almost 1.00 under `iez`.

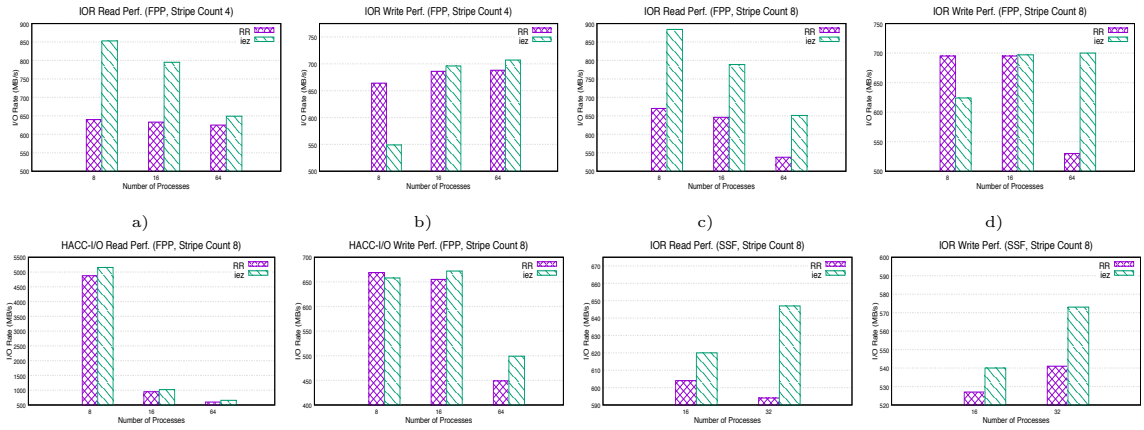


Figure 3.10: Read and write performance of IOR and HACC-I/O for FPP and SSF accesses for different Stripe Counts.

I/O Performance Next, we compared the read and write performance for the studied cases. We measured the 'I/O rate' for storing the data to and reading it from OSTs. Figures 3.10a and 3.10b show the read and write performance, respectively, for the IOR benchmark with stripe count 4 and FPP access mode. We observe that even though for smaller scale, there is a small decline in write performance, compared to the default RR approach, `iez`'s I/O rate is 34% higher for reading the data from OSTs. This improvement is achieved due to a balanced load over OSTs and OSSs, which help in mitigating the resource contention and hence improve the parallelism in the data access. Similar results were observed for the IOR benchmark with stripe count 8, as well as for HACC-I/O with stripe count 8. We observe an improvement of up to 32% in the I/O rate while reading the data for IOR with stripe count 8 as shown in Figure 3.10c. Moreover, in this case, there is improvement of up to 32% in the write performance (Figure 3.10d).

For HACC-I/O, we observe an improvement of up to 8% for reading (Figure 3.10e) and 11% for writing (Figure 3.10f). Here we also observe that the improvement gains vary for different observation points. This is because, OST load depends on a number of factors namely, write-bytes of a request, number of stripes, and MPI-Rank for MPI jobs. Different configurations of a job will yield different jobs requests, and hence different performance improvement, mainly due to varying load and stripe count used in the evaluation.

Utilization of OSSs In our next experiment, we measured the storage utilization of each OSS under the default approach and `iez`. This is because we want to balance the load on both OSSs and OSTs for an overall load-balanced setup. To this end, we aggregated the load (storage utilization) on each OST and calculated the ratio of storage being used

with respect to the total storage in each OSS. In a balanced scenario, each OSS should be utilized equally by hosting an equal share of application data. We observe that with the default approach the OSSs are imbalanced, while `iez` distributes the application data in a balanced manner across the OSSs. Figure 4.12 shows the comparison of *OST Utilization* of all 3 OSSs of our testbed under `iez` as compared to the default approach. Here we use the IOR benchmark with 64 processes, FPP access, and stripe count 8 storing a total application data of 64 GB. With the default approach, OSS-2 becomes highly loaded—31% more than mean utilization—as compared to OSS-1 and OSS-3. In contrast, with `iez`, all the three OSSs are utilized in a balanced manner.

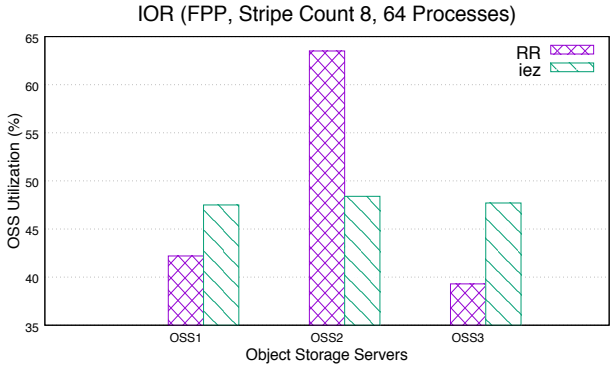


Figure 3.11: OSS Storage Utilization for IOR.

Load Balance for Single Shared File Access For SSF mode, all the processes write into and read from one shared file. We observe that since there is only one file created and the whole data is striped into almost equal stripes, under the default approach, there is only a little imbalance, as compared to the FPP access. But `iez` is able to eliminate even that imbalance. We ran both IOR and HACC-I/O with different stripe counts to observe the load distribution for SSF access. We found almost same pattern in all cases. Due to space limitation, we present two representative scenarios.

Figure 3.12 shows the *OST Utilization* of all OSTs for IOR benchmark with 16 processes, creating a file of 32 GB with a stripe count 8. Figure 3.13 shows the *OST Utilization* of all OSTs for IOR benchmark with 32 processes creating a file of 64 GB with a stripe count 8. For a single shared file access, for both HACC-I/O as well as IOR, all the processes create one single file that is striped across the eight OSTs, since the Lustre stripe count for these two data files is 8. Hence only eight OSTs are allocated to store each of the files, and the unallocated OSTs are not used. For example, as shown in Figure 3.12, with the default RR allocation scheme, for IOR, the created data file is striped across, OSTs 1, 2, 5, 7, 8, 13, 14, and 15. Out of the 8 allocated OSTs, the OST utilization for OSTs 5, 14 and 15 is greater than 50%, but for the rest of the allocated OSTs is less than 50%. In

contrast, while using `iez`, the data file is striped across OSTs 3, 4, 7, 8, 9, 11, 12, and 13. For all of these allocated OSTs, the utilization is the same (47%). Hence, mitigating the imbalance with RR. Figure 3.13 shows a similar behavior. We also observed that in SSF access mode, `iez` performs better for reading and writing the data with SSF as compared to default approach. Figures 3.10g and 3.10h show the read and write performance of `iez` compared to the default approach for 16 and 32 processes, respectively. `iez` improves read and write performance by up to 8% and 6%, respectively, compared to the default approach.

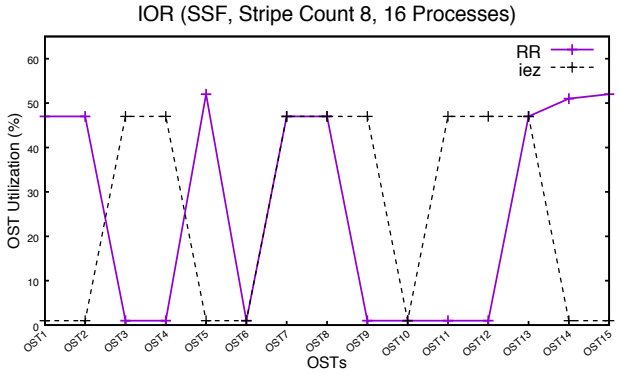


Figure 3.12: OST Storage Utilization for IOR in SSF mode for 16 Processes.

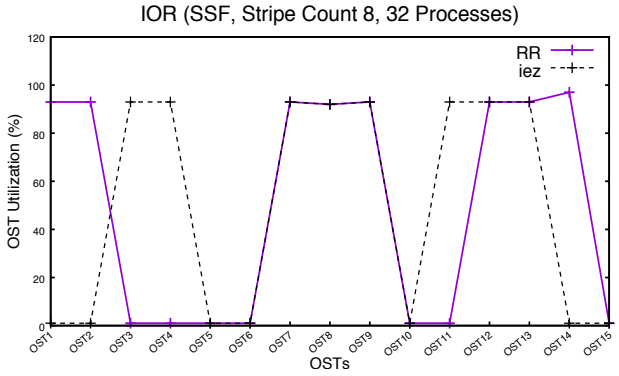


Figure 3.13: OST Storage Utilization for IOR in SSF mode for 32 Processes.

Load Balance for Concurrent Applications In our next test, we evaluated `iez` simultaneously running HACC-I/O and IOR with different job configurations from two different Lustre Clients with 16 processes on each. Each process creates one file that is stored on Lustre OSTs with a stripe count of 8 for HACC-I/O and 4 for IOR. The total amount of data stored for each file for HACC-I/O and IOR is 7.7 GB and 32 GB, respectively. As with the single application tests, we observe a significant improvement in load balancing for concurrent applications compared to the default RR approach. Figure 4.15 shows the comparison of load under both approaches on 15 OSTs in the Lustre cluster. We see that `iez` balances the load on all the OSTs. The maximum load observed with

RR is on OST1 with a utilization of 35%, while the mean utilization is 30%, resulting in the *OSTCost* of 1.17. In contrast, under `iez`, the OST load observed on all of the 15 OSTs is 30%, and hence has the *OSTCost* of 1.0. Moreover, the CPU utilization and memory usage on MDS while using `iez` for load balancing in concurrent application runs is observed to be about 1.55% and 0.12%, respectively.

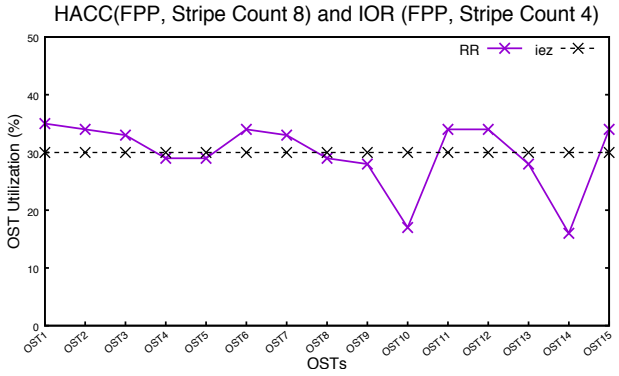


Figure 3.14: OST Storage Utilization for a simultaneous IOR and HACC execution in FPP mode with Stripe Count of 4 and 8, respectively.

3.4.1 Chapter Summary

We presented the design of an "end-to-end control plane" to optimize parallel and distributed HPC I/O systems, such as Lustre, by providing efficient load balancing across storage servers. Our proposed system, `iez`, provides global view of the system, enables coordination between the clients and servers, and handles the performance degradation due to resource contention by considering operations on both clients as well as servers. Our implementation of `iez` provides a balanced distribution of load over OSTs and OSSs in the Lustre file system. We evaluated `iez` on a real Lustre testbed using two representative benchmarks—IOR and HACC-I/O—with multiple stripe counts of files as well as SSF and FPP accesses. Compared to the default Lustre RR policy, `iez` provides up to 31.3% improvement in balancing the load. Moreover, we also observed an I/O performance improvement of up to 34% for reads and 32% for writes. Finally, the transparent design of `iez` makes it attractive for adoption in real-world deployments where access to application source code, needed for existing approaches, may not always be possible.

Chapter 4

Load Balancing for Progressive File Layout

4.1 Introduction

In Chapter 3, we presented our resource contention aware load balancing tool, `iez` [152]. For an efficient load balancing, it must also be able to serve multiple striping patterns within a single file. Striping in parallel file system enables users to obtain high I/O performance throughput [95]. Progressive file layout (PFL) [111] is a feature in Lustre where a file can have multiple striping layouts. We thus extended `iez` to support Progressive File Layout in Lustre file system, and we evaluated it on a larger scale for both PFL and non-PFL layout.

In this chapter, I describe the motivation behind our extended research, new additional features supported by `iez`, the updated architecture of `iez`, and the evaluation of the updated version of `iez` on a real Lustre deployement testbed.

4.1.1 Progressive File Layout

In Lustre file system, data is divided into stripes according to a striping pattern and the striped data is stored across OSTs in a round-robin fashion. Progressive file layout (PFL) [111] is a recent feature in Lustre where a file can have a series of flexible striping layouts. Using PFL, a file can be created with several non-overlapping extents, with each extent having different striping parameters.

An example of a 4-component PFL layout is shown in Figure 4.1. The first component with extents ranging from zero to 128 MB has only one stripe. As the file size increases beyond 128 MB till 512 MB, the file will be divided into three stripes, from 512 MB to

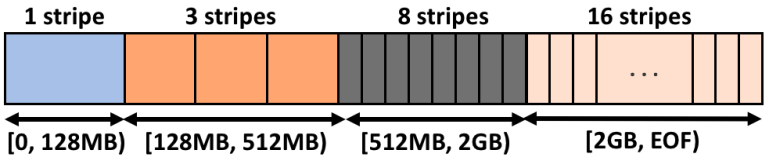


Figure 4.1: An example PFL layout.

2 GB, the number of stripes is eight, and beyond 2 GB till the end of file, the file will have 16 stripes. The PFL feature is implemented using composite file layouts for regular files. The number of sub-layouts in each file and the number of stripes in each sub-layout can be specified by users using the `lfs setstripe` command. An example command for the PFL layout in Figure 4.1 is given below, where parameters `E` and `c` specify the extent and stripe count respectively.

```
lfs setstripe -E 128M -c 1 -E 512M -c 3 -E 2G -c 8 -E -1 -c 16 <filename>
```

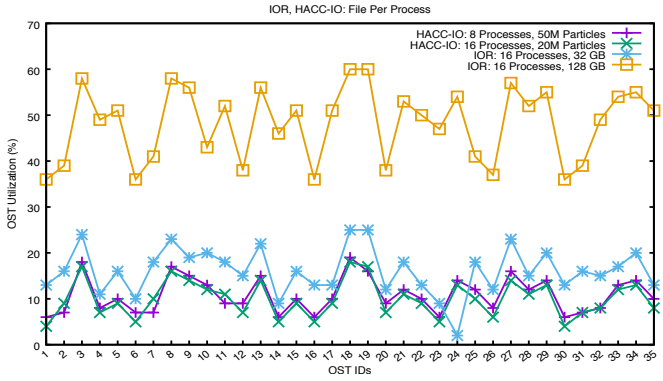


Figure 4.2: OST utilization under default load balancer in Lustre (RR) for IOR and HACC-IO for Non-PFL setup (stripe count 8).

4.1.2 Motivation

In order to highlight the load imbalance in a default Lustre deployment, we conducted a quantitative study that used RR to distribute I/O load on OSTs. We deployed a testbed of 10-node Lustre cluster, with 1 MDS, 7 OSSs and 2 Clients. Each OSS in our cluster manages 5 OSTs with a capacity of 10 GB each. Hence, the cluster has 35 OSTs in total with a total capacity of 350 GB. We ran IOR benchmark with 16 processes that produce 32 GB and 128 GB of data to be stored on the OSTs in the FPP access mode. We also ran HACC-I/O application for 8 and 16 processes for 50 *million* and 20 *million* particles, generating 14.3 *GB* and 11.7 *GB* data respectively. All experiments were run for both PFL and non-PFL setups. For the PFL setup, we use the same configuration as shown

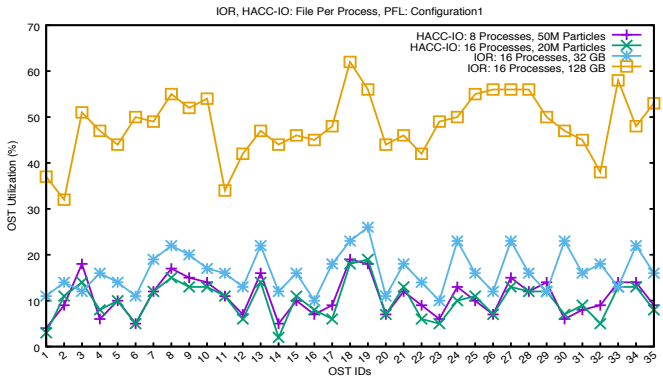


Figure 4.3: OST utilization under default load balancer in Lustre (RR) for IOR and HACC-IO PFL setup (Configuration 1)

in Figure 4.1, referring to it as *Configuration 1*. For non-PFL setup, the stripe count was taken as 8.

Figure 4.2 shows the storage utilization in each OST, for different runs of IOR and HACC-IO in FPP mode in the non-PFL setup. In a balanced load setting, these graphs would be straight lines (representing ideal load balance), but in the studied scenario, the load is observed to be imbalanced with some OSTs getting a lot more load than others. A similar pattern can be seen in Figure 4.3 for IOR and HACC-I/O for FPP mode in PFL setup. These results show that with the default Lustre deployment that uses RR scheduling to allocate OSTs for each job, there can be a significant load imbalance at the server level. The load imbalance persists at different scales and different striping layouts (PFL and non-PFL) and thus can lead to imbalanced resource usage and resource contention.

4.2 Design

Figure 4.4 shows an overview of `iez` architecture to support both PFL and non-PFL layouts. When running applications for the first time, the client side makes use of a customized tracing tool, `miniRecorder`. This tool collects information about the I/O accesses, such as the number of bytes written, file name, number of stripes, and MPI rank and communicator for each file. `MiniRecorder` needs to collect traces only for the first run of an application. `iez` identifies an application's I/O behavior, which does not change across multiple runs of an application. The collected traces are fed into the `parser` that then uses the information to drive the `prediction model`. Our predictions are based on ARIMA time series modeling [51]. The output of the time series prediction provides

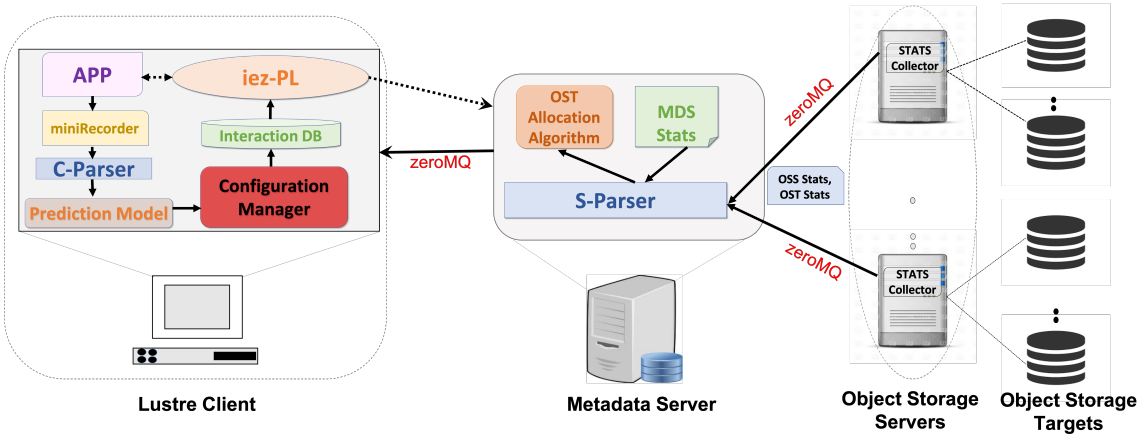


Figure 4.4: Overview of the updated `iez` architecture.

estimates of future application requests, which are sent to the `configuration manager`. The configuration manager is responsible for determining if the layout for an application is PFL. `PFL Config` stores all the PFL configurations added by a user and is used by the configuration manager. The output is then stored in an `interaction database` for later use by `iez`. We refer to the database as "interaction database" because it offers a point of interaction between our server-side and client-side libraries.

On the server side, OSSs collect the CPU and memory usage information, associated OSTs capacity (*kbytestotal*) and the number of bytes available on the OSTs (*kbytesavail*). These statistics are sent to the MDS using the `statistics collector` module on the OSSs. The collected information on the MDS is parsed in the `statistics collector` placed on the MDS to generate a file containing updated statistics for the MDS, OSS and associated OSTs. This file is fed to the `OST allocation algorithm`. The input to the OST allocation algorithm is the predicted set of requests received from the clients from the `interaction database` via ZeroMQ message queues [85], and the output is the list of OSTs to be allocated for every request, which will yield a load-balanced distribution over the involved OSSs and OSTs. The allocated OSTs are stored in the interaction database along with the predicted requests. Next, the `placement library`, `iez-PL`, intercepts the I/O requests from the applications, consults the interaction database, and routes the application requests to appropriate resources by creating a given file's metadata on the MDS.

In the following sections, we explain the new features of `iez` in detail.

4.2.1 Parallel File Access Modes for Varying Striping Layouts

Before explaining the updated components of `iez`, we show how different file access modes - FPP and SSF are represented in PFL and non-PFL layouts. We show an overview of the representation in Figure 4.5. The PFL layout used in the example is *Configuration 1* discussed in Section 4.1.1.

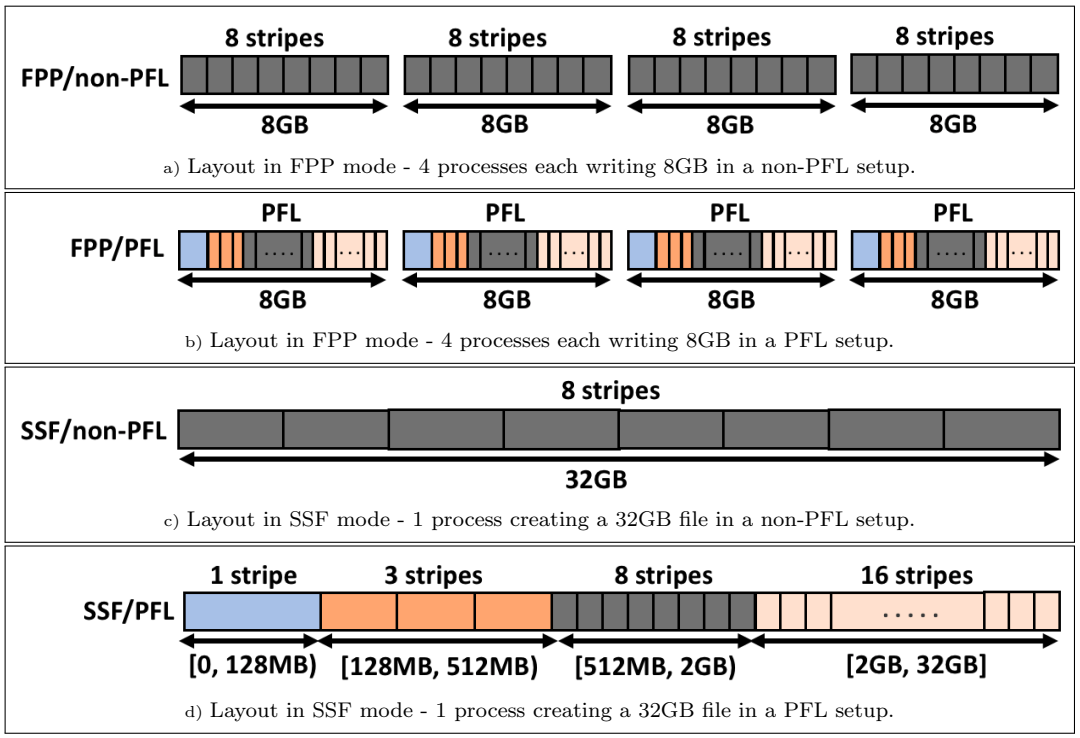


Figure 4.5: Comparison between FPP and SSF approaches in non-PFL and PFL layouts.

For FPP mode, we have four processes each writing 8 GB to the parallel file system. In the non-PFL layout, each 8 GB file will be striped into a pre-defined number of stripes (8 in the figure), where as, in the PFL layout each 8 GB file will be striped according to the PFL layout set by the user for those files or directories (*Configuration 1*). In SSF mode, a single process creates a single file where all other processes perform I/O operations. This file will be striped into a pre-determined number of stripes according to a non-PFL or a PFL layout. It is evident that different stripes will be of different sizes based on the file access modes as well the striping layouts. The challenge lies in selecting OSTs to place these varying size stripes such that all OSTs are load balanced and have less resource contention, which improves the I/O throughput.

4.2.2 Configuration Manager

The `configuration manager` determines if an application is performing I/O in PFL or non-PFL layout. It takes as input the predicted set of requests, containing the file name, stripe count, write bytes, and MPI rank, which is given by the `prediction model`, and the file containing all PFL layouts for a client. If the file names or the application directory in the predicted requests set match those in the PFL configuration file, the stripe size and stripe counts for the file are set based on the PFL configuration. If there are no matches found for the file name or the directory in the PFL configuration, then the layout is non-PFL. The output of the configuration manager is the set of all predicted requests combined with the corresponding stripe size, stripe count, and MPI rank of the files. This entire set is sent to the `interaction database`. The stripe size for both layouts is calculated for all files in the configuration manager based on the 64k-alignment constraint by Lustre (Chapter 3, Section 3.3).

Table 4.1: Interaction database snapshot for IOR in FPP mode in PFL layout.

File Name	File Size	E-ID	Extent Start	Extent End	Stripe Size	SC	Rank
/mnt/lustre/ior/test.0	8589934592	1	0	134217728	134217728	1	0
/mnt/lustre/ior/test.0	8589934592	2	134217728	536870912	134217728	3	0
/mnt/lustre/ior/test.0	8589934592	3	536870912	2147483648	201326592	8	0
/mnt/lustre/ior/test.0	8589934592	4	2147483648	8589934592	402653184	16	0
/mnt/lustre/ior/test.1	8589934592	1	0	134217728	134217728	1	1
/mnt/lustre/ior/test.1	8589934592	2	134217728	536870912	134217728	3	1
/mnt/lustre/ior/test.1	8589934592	3	536870912	2147483648	201326592	8	1
/mnt/lustre/ior/test.1	8589934592	4	2147483648	8589934592	402653184	16	1

4.2.3 Interaction Database

The `interaction database` is a SQL database located on the Lustre clients. It serves as the medium through which the MDS and clients interact with one another. First, the output set from the `configuration manager` is stored in the database. Different tables are used to store PFL and non-PFL layout files. Tables 4.1 and 4.2 show an example snapshot of the interaction database for IOR in FPP mode with 2 processes each writing 8 GB in PFL and non-PFL layout respectively. For PFL layout, we use the example PFL configuration (*Configuration 1*), discussed in Section 4.1.1. As seen in Table 4.1, every file is associated with all the extents specified in the PFL configuration. For each file, we store the file name, the file size in bytes, the extent ID (E-ID) from the PFL configuration file, the corresponding start and end range for the extent, stripe size, stripe count (SC) and the MPI rank. For non-PFL layout, shown in Table 4.2, we store the file names, stripe size of the files, number of stripes associated with every file, and the MPI Rank. The

stripe size of the files is calculated using the 64k-alignment parameter which is discussed in Chapter 3, Section 3.3.

Table 4.2: Interaction database snapshot for IOR in FPP mode in non-PFL layout.

File Name	Stripe Size	Stripe Count	MPI Rank
/mnt/lustre/ior/test.0	1073741824	8	0
/mnt/lustre/ior/test.1	1073741824	8	1

The MDS uses a scalable publisher-subscriber model via Zero message queue [85] to retrieve the required contents from the interaction database. The pub-sub model helps in scaling `iez` to a large number of clients [121]. The specific steps are discussed in OST Allocation Algorithm described in (Chapter 3, Section 3.3). For our implementation, we use MySQL 8.0.12 Community Server Edition. Our results show that writing and retrieving data from the interaction database is very efficient, using <0.3% and <0.4% of CPU and memory, respectively.

The list of OSTs obtained from the OST allocation algorithm are then sent to the respective clients using the publisher-subscriber model via ZeroMQ. The complete set of requests are stored in the interaction database. Example entries for the database with the complete allocation for an IOR application in FPP mode with 2 processes each writing an 8 GB file in both PFL (*Configuration 1*) and non-PFL layouts are shown in Table 4.3. We add a new column *OST List* in the database. The *OST List* is a space separated load-balanced list of OSTs for every write request. This example is for a setup with 7 OSSs and 35 OSTs (5 OSTs associated with every OSS) – therefore, OST ids range from 1 to 35. As described earlier, the placement library (`iez-PL`) then uses this information to place the requests, thus completing the load-balanced allocation of resources. If for any run of the application, `iez-PL` is unable to find more than 50% of files in the interaction database, `miniRecorder`, AIPA and OST Allocation Algorithm will be executed again to update the interaction database.

4.2.4 Placement Library

Following the example layouts shown in Figure 4.5, `iez-PL` supports both non-PFL and PFL layouts for FPP and SSF.

For every I/O cycle, `iez-PL` queries the interaction database via the MySQL C API with the file name passed by the original metadata operation, fetches the matching rows, and applies the predicted striping pattern to the file, if the file does not exist yet. To facilitate this, Lustre provides a user library called *llapi*, which allows the user to describe a specific

striping pattern. However, `iez-PL` cannot use *llapi* directly, since it internally triggers `open()` calls, which would result in a continuous, recursive loop due to nature of the preloading mechanism. Therefore, `iez-PL` mimics the behavior of *llapi* and communicates directly with the Lustre Logical Object Volume (LOV) client to create the file metadata on the MDS, similarly to [115, 116, 152].

Algorithm 3: File layout creation on the MDS.

```
Input: File Name file, Access mode flags
Output: Call to real metadata operation (e.g., open())
1  begin
2      if fileExits(file) == TRUE then                          // File exits; return.
3          return realMetadataOperation(file, flags)
4      flags = flags | O_LOV_DELAY_CREATE
5      result = queryInteractionDatabase(file)
6      if numMySQLrows(result) == 1 then                        // Non-PFL layout.
7          row = fetchMySQLrow(result)
8          layoutEA = allocLayoutEA(row.stripeCount, row.stripeSize, row → OSTs)
9          createLayoutEAonMDS(file, flags, 0644, layoutEA)
10     else if numMySQLrows(result) > 1 then                    // PFL layout.
11         while row = fetchMySQLrow(result) do
12             allocExtentPFL(layoutPFL, row.extentEnd, row.stripeCount, row.stripeSize, row → OSTs)
13         createCompositeLayoutMDS(file, flags, 0644, layoutPFL)
14     return realMetadataOperation(file, flags)
```

If the result returned by the MySQL query contains only one row, the non-PFL layout is applied by allocating a Lustre file identifier and the corresponding *Layout Extended Attributes* (Layout EA) on the MDS. For both FPP and SSF, the predicted striping pattern is applied by initializing the Layout EA with the stripe count, stripe size, and list of OSTs retrieved from the interaction database.

If the MySQL query returns more than one row, the PFL layout is utilized. For both FPP and SSF, every row represents one non-overlapping extent for a given file. `iez-PL` iterates over all rows, allocates an array of sub-layout components (one for each file extent), and applies the predicted striping pattern to the file by storing it in a composite layout on the MDS. Composite layouts, unlike Layout EA, allow the specification of different specific striping patterns for different ranges (i.e., extents) in the same file.

Algorithm 3 presents a simplified overview of `iez-PL`, which is run on every client. `iez-PL` currently supports POSIX I/O, MPI-IO, and HDF5 and the following I/O calls: `open[64]()`, `creat[64]()`, `MPI_File_open()`, and `H5Fcreate()`. The key advantage of this transparent approach is that applications can directly benefit from the prediction model without modifying the source code.

Table 4.3: Interaction database snapshot showing OST allocation for IOR in FPP mode in non-PFL layout.

File Name	Stripe Size	Stripe Count	MPI Rank	OST List
/mnt/lustre/ior/test.0	1073741824	8	0	30 20 5 1 22 35 14 23
/mnt/lustre/ior/test.1	1073741824	8	1	20 19 1 9 29 2 33 18

4.3 Evaluation

We evaluate `iez` using a real Lustre deployment testbed. We use a Lustre cluster of 10 nodes with 1 MDS, 7 OSSs and 2 Clients. All of the nodes run CentOS 7 atop a machine with 8 cores, 3.2 GHz AMD FX-8320E processor, and 16 GB memory. Furthermore, each OSS has 5 OSTs, each supporting 10 GB of attached storage, resulting in a 350 GB Lustre store. Our tests use HACC-I/O kernel [80] and IOR benchmark [107] in FPP and SSF access modes.

To the best of our knowledge, `iez` is the first work to consider a global view of all the system resources in deciding application request stripe placement. Existing approaches (BPIO [155], TAPP-I/O [116] etc.) balance load among I/O servers on per-application basis, i.e., on the client side, and do not consider the global view, i.e., the requirements of the other applications as well as the OSS and OST stats. Moreover, unlike `iez`, such client-side techniques cannot handle multiple simultaneous applications. Thus, these are not directly comparable to `iez`. For these reasons, we choose to use the default RR approach of Lustre as the basis for our comparison.

To capture the degree of load balancing across participating OSTs for a particular test run, we define a metric, *OSTCost*, as the ratio of the maximum utilization of any OST to the mean utilization of all the OSTs. Therefore,

$$OSTCost = \frac{MaxOSTUtil}{MeanOSTUtil} \tag{4.1}$$

An ideal load balanced system has the *OSTCost* of 1. We also define *OST Utilization* of an OST as the storage used by the client application on the OST as a fraction of the total storage available on the OST.

To evaluate `iez` on PFL, we use two PFL configurations as shown in Table 4.4. PFL Configuration 1 is described in the Section 4.1.1 and is used in the motivation graphs in Section 4.1.2. PFL Configuration 2 is a 3-component PFL layout. The first component with extents ranging from zero to 128 MB has only one stripe. As the file size increases beyond 128 MB till 2 GB, the file will be divided into 12 stripes, from 2 GB till the end of file, the number of stripes is 32.

Table 4.4: Interaction database snapshot showing OST allocation for IOR in FPP mode in non-PFL layout.

PFL Configuration 1		**PFL Configuration 2**	
Extent Range	***Stripe Count***	***Extent Range***	***Stripe Count***
[0, 128 MB)	1	[0, 128 MB)	1
[128 MB, 512 MB)	3	[128 MB, 2 GB)	12
[512 MB, 2 GB)	8	[2 GB, EOF)	32
[2 GB, EOF)	16		

4.3.1 Load Balance for FPP Access IOR

Figure 4.6 shows the comparison of load under `iez` and the default RR data allocation on 35 OSTs in the Lustre cluster represented as *OST Utilization* for the IOR benchmark with varying data sizes. We see that `iez` balances the load on all OSTs in a near-optimal manner. For example, for 16 processes, 32 GB data, the maximum load observed with RR approach is on OST-18: the OST utilization is 25%, while the mean utilization is 16.06%, resulting in the *OSTCost* of 1.56. In contrast, the maximum load observed under `iez` is 18% with the corresponding *OSTCost* of 1.08, i.e., near optimal. The almost horizontal line for *OST Utilization* for `iez` underscores its effectiveness. Overall, `iez` was able to reduce the *OSTCost* by 30.8% compared to the default RR approach. We observed similar results while running IOR with 16 processes and 128 GB. `iez` is able to distribute data on all 35 OSTs in a balanced way for other studied cases as well. In this case, we observe an *OSTCost* of 1.25 and 1.1 under RR and `iez` respectively. Hence, `iez` provides 12% better *OSTCost* than the default RR approach.

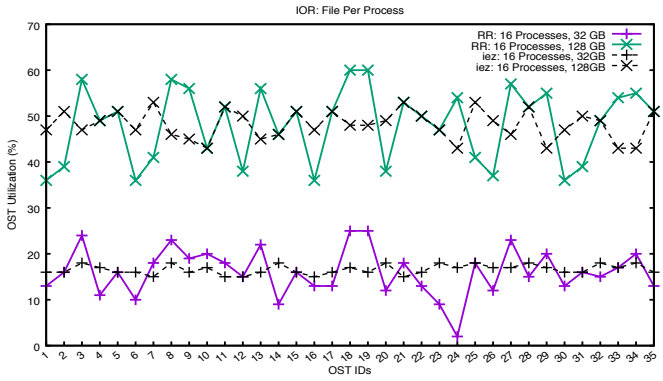


Figure 4.6: OST storage utilization for IOR in FPP mode and non-PFL layout (stripe count 8).

`iez` is able to balance the load for IOR in FPP access modes for PFL layouts as well. The results for PFL Configuration 1 and 2 are shown in Figures 4.7 and 4.8. For 16 processes, 32 GB data, we get an OST cost of 1.58 and 1.08 under RR and `iez` respectively for PFL Configuration 1, and an OST cost of 1.5 and 1.02 under RR and `iez` respectively for PFL Configuration 2, thereby providing a 31.6% and 32% improvement in load balance. Similar results are seen in IOR 16 processes 128 GB run in FPP mode.

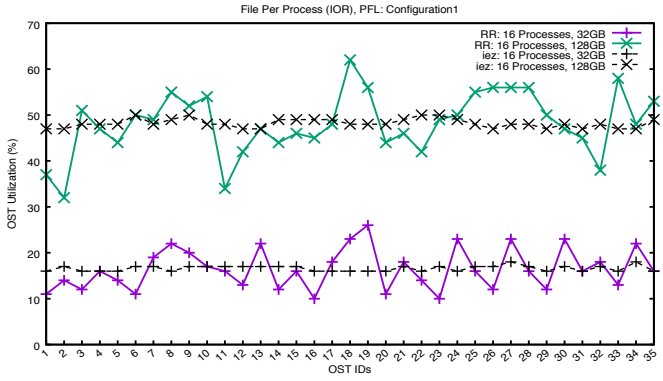


Figure 4.7: OST storage utilization for IOR in FPP mode and PFL Configuration 1.

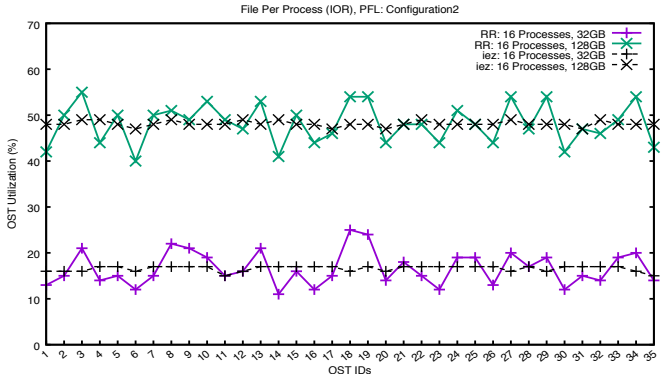


Figure 4.8: OST storage utilization for IOR in FPP mode and PFL Configuration 2.

4.3.2 Load Balance for FPP Access HACC-I/O

For HACC-I/O, we evaluate `iez` for 8 processes with 50 *million* particle data, and 16 processes with 20 *million* particle data. Each process creates one data file that is stored in the Lustre OSTs in a non-PFL layout with a stripe count of 8, and using two PFL configurations. Total data stored in the files is approximately 14.3 *GB*, and 11.7 *GB* for 8, and 16 processes, respectively. Similar to IOR, we observe a significant improvement in load balancing for HACC-I/O as well, shown in Figure 4.9 for non-PFL layout, and Figure 4.10 for PFL layout, compared to the default RR approach. *OSTCost* for 16 processes with default approach and `iez` for non-PFL layout is observed to be 1.8 and 1.2, respectively, with `iez` reducing the OSTCost by 33 %. Similar behavior is observed for PFL layout, where the *OSTCost* is observed to be significantly lower than default and closer to 1.00 under `iez`.

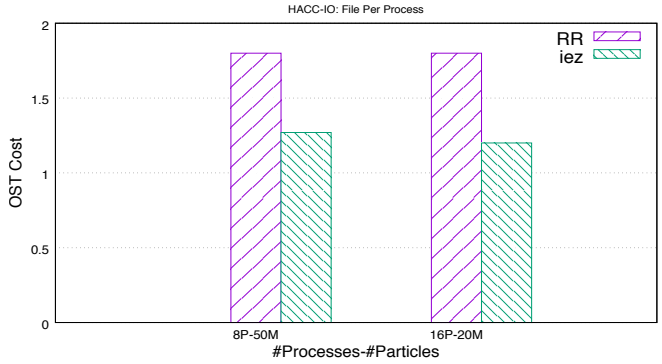


Figure 4.9: OST COST for HACC-IO in FPP mode and non-PFL layout (stripe count 8).

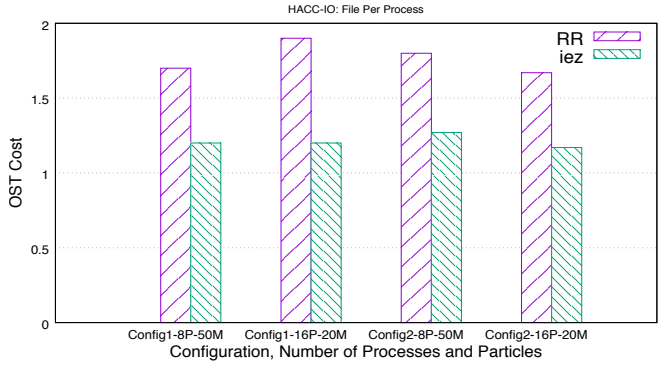


Figure 4.10: OST COST for HACC-IO in FPP mode and PFL layout.

4.3.3 I/O Performance

Next, we compare the read and write performance for the studied cases. We measured the 'I/O rate' for storing the data to and reading it from OSTs. Read performance is shown for HACC-IO and IOR in FPP and SSF access modes for both PFL and non-PFL layouts in Figures 4.11a, 4.11b, 4.11c, 4.11d, 4.11e, and 4.11e. We see an improvement of upto 43% in read performance for `iez` compared to default RR approach. This improvement is achieved due to a balanced load over OSTs and OSSs, which help in mitigating the resource contention and hence improve the parallelism in the data access.

Write performance results for HACC-IO and IOR is shown in Figures 4.11g, 4.11h, 4.11i, 4.11j, 4.11k, and 4.11k. Write performance of HACC-IO and IOR behaves slightly better or at par for allocation via `iez` when compared to default RR allocation. This means that `iez`'s interception of create calls and allocating appropriate OSTs to the files incurs a negligible overhead on the file system.

We observe that the improvement gains vary for different observation points. This is because, OST load depends on a number of factors namely, write-bytes of a request, number of stripes, and MPI-Rank for MPI jobs. Different configurations of a job will yield different jobs requests, and hence different performance improvement, mainly due to varying load and stripe count used in the evaluation.

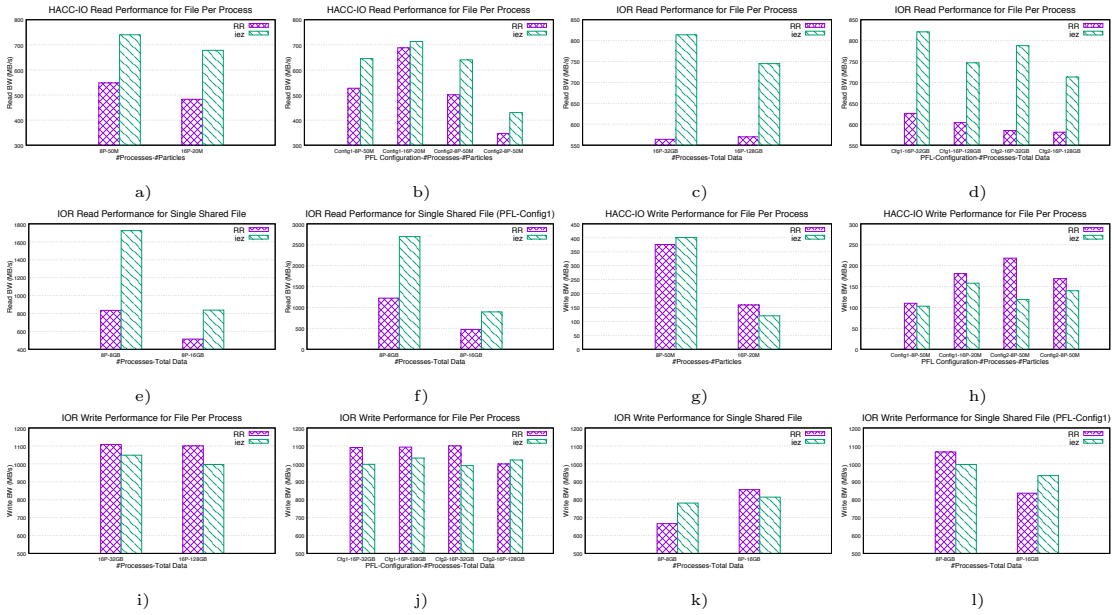


Figure 4.11: Read and write performance of IOR and HACC-I/O for FPP and SSF accesses for PFL and non-PFL layouts.

4.3.4 Utilization of OSSs

We want to achieve an end-to-end load balance in the file system. Therefore, `iez` needs to balance the load on both OSSs and OSTs for an overall load-balanced setup. In our next experiment, we measure the storage utilization of each OSS under the default approach and `iez`. To this end, we aggregate the load (storage utilization) on each OST and calculated the ratio of storage being used with respect to the total storage in each OSS. In a balanced scenario, each OSS should be utilized equally by hosting an equal share of application data. Figure 4.12 shows the comparison of *OSS Utilization* of all seven OSSs of our testbed under `iez` as compared to the default approach. We only show the result of running IOR benchmark with 16 processes in FPP access mode, storing a total application data of 32 GB and 128 GB under PFL Configuration 1. We observe that with the default approach the OSSs are slightly imbalanced, while `iez` removes that slight imbalance by distributing the application data in a balanced manner across the OSSs.

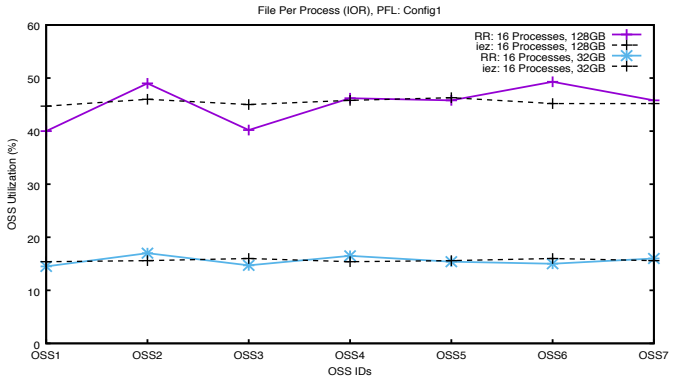


Figure 4.12: OSS Storage Utilization for IOR in FPP mode for PFL Configuration 1.

4.3.5 Load Balance for Single Shared File Access

For SSF mode, all the processes write into and read from one shared file. We run IOR in SSF access mode in both non-PFL and PFL configuration 1 layout, and the results for OST cost are shown in Figures 4.13 and 4.14 respectively. We run IOR for 8 processes generating 8 GB and 16 GB files. We observe that `iez` is able to reduce the OST cost for all scenarios when compared to the default RR approach. We get a reduction in OST cost by 38% and 37.8% in non-PFL and PFL layouts respectively. Additionally, as discussed in the previous section, we observe that in SSF access mode, `iez` provides better read and write performance when compared to the default approach.

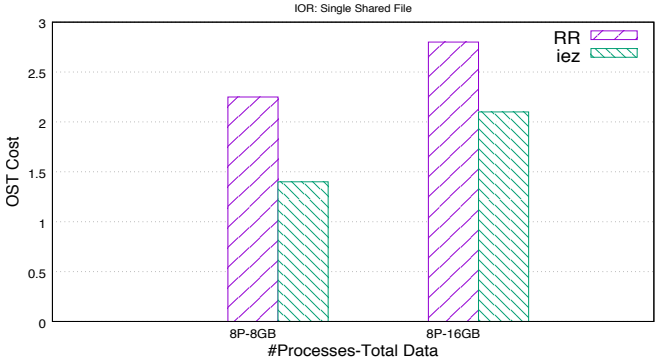


Figure 4.13: OST Cost for IOR in SSF mode in non-PFL layout (stripe count 8).

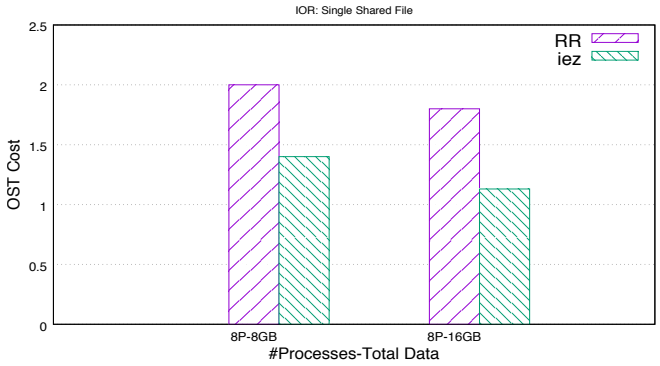


Figure 4.14: OST Cost for IOR in SSF mode in PFL Configuration 1.

4.3.6 Load Balance for Concurrent Applications

We also evaluate `iez` for concurrent applications by simultaneously running IOR and HACC-I/O with different job configurations from two different Lustre Clients with 8 processes on each. Each process creates one file that is stored on Lustre OSTs with PFL Configuration 1 for IOR and PFL Configuration 2 for HACC-I/O. The total number of particle data stored for each file for HACC-I/O is 50 *million* and the total data size for IOR is 64 *GB*. As with the single application tests, we observe a significant improvement in load balancing for concurrent applications compared to the default RR approach. Figure 4.15 shows the comparison of load under both approaches on 35 OSTs in the Lustre cluster. We see that `iez` balances the load on all the OSTs. The maximum load observed with RR is on OST9 with a utilization of 51%, while the mean utilization is 35%, resulting in the *OSTCost* of 1.46. In contrast, under `iez`, the OSTCost observed on all of the 35 OSTs is 1.08, thereby improving the load balance by 26%. Moreover, the CPU utilization and memory usage on the MDS while using `iez` for load balancing in concurrent application runs is observed to be about 1.55% and 0.12%, respectively. We observe similar improvements in load balance for concurrent application in non-PFL layout as well.

4.4 Chapter Summary

In this chapter, we presented the extended version of `iez` that supports the Progressive File Layout in Lustre. `iez` provides global view of the system, enables coordination between the clients and servers, and handles the performance degradation due to resource contention by considering operations on both clients as well as servers. Our implementation of `iez` provides a balanced distribution of load over OSTs and OSSs in the Lustre file system, and is able to handle both PFL and non-PFL layouts for files. We evaluated

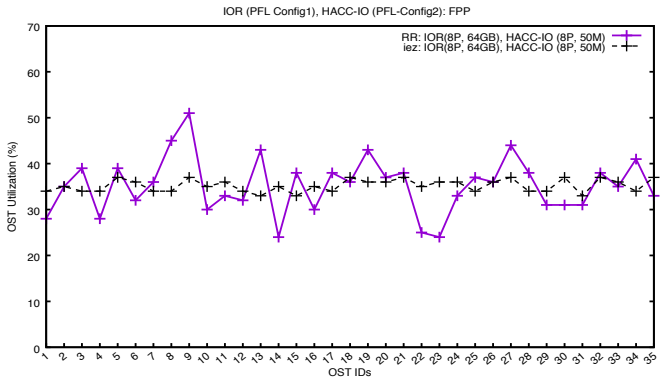


Figure 4.15: OST Storage Utilization for a simultaneous IOR and HACC execution in FPP mode in PFL Configuration 1 and PFL Configuration 2, respectively.

`iez` on a real Lustre testbed using two representative benchmarks—IOR and HACC-I/O—with multiple stripe counts of files as well as SSF and FPP accesses. Compared to the default Lustre RR policy, `iez` provides up to 33% improvement in balancing the load. Moreover, we also observed an I/O performance improvement of up to 43% for reads without affecting the performance for writes. Finally, the transparent design of `iez` makes it attractive for adoption in real-world deployments.

Chapter 5

Facilitating Data Sharing in Scientific Workflows

5.1 Introduction

High Performance Computing (HPC) systems provide a myriad of compute and storage resources to sustain modern complex scientific applications, and offer unprecedented capabilities with the dawn of the exascale computing era. At the same time, modern Big Data driven science relies on sophisticated workflows [100] that encompass distributed storage locations and involve dataflows from instruments to processing resources and archival storage. The workflows from a multitude of inter-disciplinary scientific domains typically comprise multiple applications solving different parts of the science problem simultaneously. More precisely [157], a scientific workflow consists of a set of control- or data-dependent jobs that form a directed acyclic graph (DAG) to carry out a complex computational process [9, 59, 158]. Control-flow dependency specifies that a job must be completed before other jobs further down the DAG can start. In contrast, dataflow dependency specifies that a job cannot start until all its input data (typically created by previously completed jobs and stored in a file system) is available. Therefore, one of the major challenges for efficient and high-performance workflow execution is optimal data sharing and communication among the various applications in a workflow.

A number of prior works have explored how to mitigate the workflow-aware data communication challenges. For example, existing approaches include establishing point-to-point connectivity by direct socket connections among applications, leveraging POSIX-IO file systems for applications to access data using standard POSIX APIs [61, 137, 158], and leveraging pre-defined workflow-aware data models (Swift [166], Pegasus [65]). One promising solution, Data Broker [136], provides software based data isolation and utilizes a programming model atop key-value stores for data transformations among applications. However, Data Broker currently only comes with a Redis [15] back-end to store data in memory, which poses the following limitations when applied to support scientific workflows at scale.

- ***Lack of Data Persistence***. Redis is an in-memory data store, and does not have an efficient data persistence mechanism for data sharing among multiple applications. Data persistence in Redis works well on single-node Redis instances but does not scale efficiently. Therefore, issues arise when workflow applications require multiple nodes along with data persistence. The lack of long-term data storage entails that applications either need to re-run the computation or retrieve data from slow disks, often multiple times, as data from one application cannot be effectively shared with another. Thus, a performance bottleneck is created.
- ***Deployment Challenges.*** Every time a workflow starts, or restarts after a failure, the associated instance of Redis needs to be restarted. Data needs to be read from disks into the in-memory database of a large number of servers, and parts of data which do not fit in-memory would still be spilled onto disks. This poses deployment issues and slowdowns, thus making Data Broker atop Redis a poor choice for persistent data. The problem is exasperated in scalable systems where failures are the norm and not an exception.
- ***Memory Constraints.*** I/O-intensive applications in large scale workflows need large chunks of data to be shared. This further makes the above two challenges problematic as applications are now more likely to run out of available memory and entail use of slower disks and face the issues associated with such accesses.

While Data Broker is desirable, the above challenges introduce huge functional overhead for running large scale scientific workflows that are data-dependent and need iterative runs of applications over the same data. Currently, with the absence of an efficient persistent storage, the workflow applications would have to be limited to using data size that can be exchanged through in-memory store, to get acceptable performance. Thus, a performance versus application scale trade-off is created, which is not desirable. Therefore, there is an increasing need for efficient and high-performance persistent storage for the application data that exceeds memory limits.

Parallel file systems, such as Lustre [8], and IBM Spectrum Scale [135] provide efficient storage capabilities in HPC environment and can provide data persistence. However, these suffer from scalability limitations due to unmitigated metadata load distribution. In parallel file systems, the metadata server (MDS) is responsible for the data layout and all application requests have to be translated by the MDS into logical block addresses. Object Storage Devices (OSDs) on the other hand, combine CPU, network interface, and local cache with an underlying disk, and therefore can replace the traditional block-level interface. Ceph object storage [3] acting on top of OSDs, utilizes a highly adaptive distributed metadata cluster architecture that improves the scalability of metadata access. Therefore, large-scale object stores provide a reliable storage solution for large-scale HPC workflows.

To address the above issues, we propose a novel approach, *Workflow Data Communicator*, for using Object Based Storage to facilitate data sharing in scientific workflows. *Workflow Data Communicator* provides data persistence, reduces the operational overhead of using in-memory data store, and facilitates the storage and retrieval of large amount of shared data in an efficient way. *Workflow Data Communicator* enables a persistent back-end storage for Data Broker by using Ceph Object Store. It manages the shareable application data in the form of objects and pools (separate logical partitions), thus providing a high level view of namespaces and tuples to the applications, to facilitate data sharing, without any modification to the application or workflows. It overcomes the limitations of in-memory data stores by providing a stable, highly-reliable and scalable persistent object storage. We evaluate *Workflow Data Communicator* on a 8 TB Ceph Object Store cluster, and show the applicability of *Workflow Data Communicator* on different data access patterns and real-world HPC workflows.

5.2 Background

This section introduces the background information on scientific workflows, and gives an overview of Data Broker [136]. We also describe Object based storage and introduce Ceph object store [3]. Our enhancements to Data Broker design are built atop these technologies.

5.2.1 Workflow

A set of applications that work on different parts of the same problem by sharing the results with each other can be represented together as a Workflow, as depicted in Figure 5.1. For instance, a scientific problem may consist of multiple simulations, machine learning based analysis, and visualization applications. Many of these components can be run simultaneously to take advantage of high computation and network bandwidth, thus speeding up the problem solution. Advantages of using a workflow for problem-solving include ease of maintenance of a set of smaller applications, reusing applications across workflows, and extending existing workflows to add additional functionalities by adding new applications.

The formal definition of HPC workflows stems from the APEX Workflows whitepaper [6] published in collaboration by a number of U.S. Department of Energy (DOE) computing facilities. A workflow governs the dependencies of tasks (logical entity) and data, and

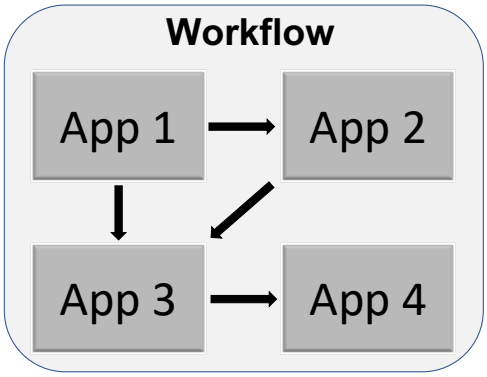


Figure 5.1: An example of scientific workflow.

is often represented by using a directed acyclic graph (DAG) for simple workflows, and cyclic graphs for complex workflows.

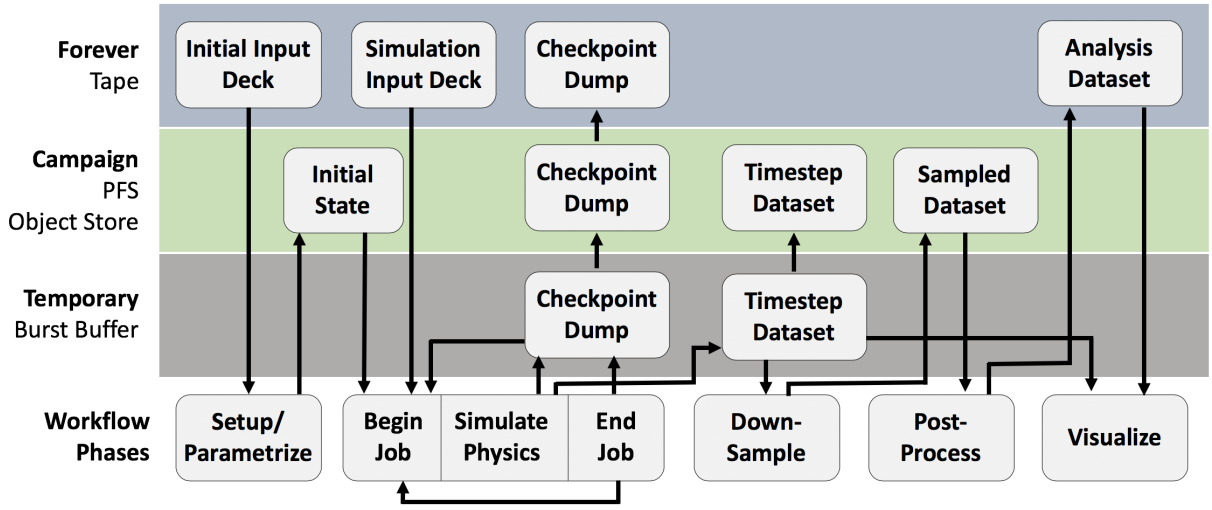


Figure 5.2: An example of an APEX scientific simulation workflow [6].

An illustration of an APEX simulation workflow is shown in Figure 5.2. A workflow typically consists of three to four major phases. First, the pre-processing phase transforms the initial input data to an initial state for the simulation. Next, during the simulation/data generation phase, applications write snapshots and timestep data for fault tolerance, post-processing applications, and potential archiving. The post-processing and visualization phases read data from the sampled-and-analysis-datasets generated by the simulation phase. This simulation workflow might be a part of the pipeline of other more data-intensive workflows. Therefore, it is clear that different applications or different phases in the same application may need to read data from campaign storage (Parallel File System (PFS) or Object Store). *Workflow Data Communicator* aims to facilitate such data sharing.

5.2.2 Data Broker

Data Broker [136] provides a library for applications to exchange or store data in the form of named tuples. Each tuple is associated to a namespace for data isolation. The architecture of Data Broker is shown in Figure 5.3 and consists of two major components: a client library and a system library, with a specified API between the two. The client library provides the application-facing API and the system library interacts with the storage back-end. The design allows for different system libraries to provide different types of storage classes and types such as in-memory or disk. The currently available implementation supports Redis [15] for in-memory storage with limited persistence. There is also a forwarding system library that supports building tree connectivity for extreme scale environments where the workflow would reach the limits of permitted file descriptors (or connections).

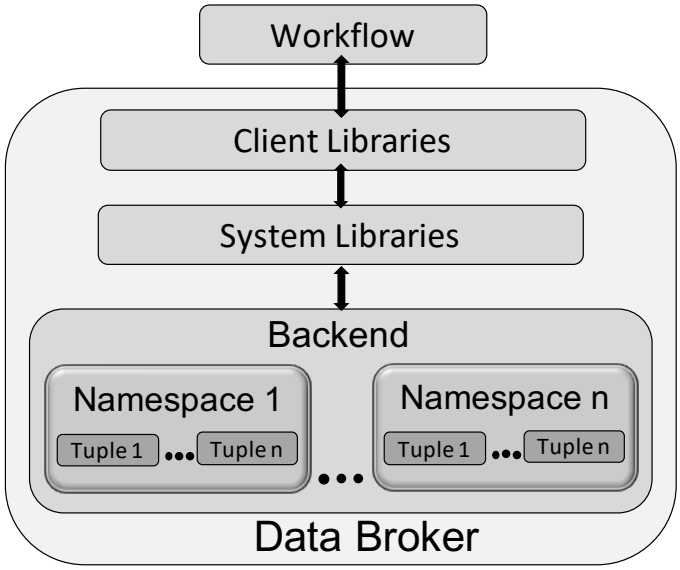


Figure 5.3: Overview of Data Broker.

Namespaces These provide isolation of data between users or applications and can group data with similar properties or requirements, for example, the type of storage class or the level of fault tolerance. The storage class determines whether data should be made persistent or can be kept in memory. The fault tolerance requirement lets the user make the trade off between faster access and more resilience against failures of storage components. While not implemented yet, namespaces can also be used to define access permissions. This could reduce the amount of meta-data required for individual tuples.

Tuples These are immutable data containers that are addressed by a name or key and inherit the properties of their namespace. The semantics of access are inspired by the Linda Tuple Language [165]. When a tuple is created, it is added to an existing namespace. Writing another tuple to the same key will not overwrite or modify the existing value. Instead, the write creates a queue of tuples, which is accessed in first-in-first-out (FIFO) order.

Client Library This is a thin layer that extends the variety of available functions for the user while hiding some of the complexity of the system library API, especially handling of non-blocking requests and out-of-order completions. Table 5.1 shows some examples of the client API to manage namespaces and access data. Note there are two types of APIs to retrieve data: a destructive *dbrGet* and a non-destructive *dbrRead*.

dbrCreate	Create a new namespace
dbrAttach	Connect to an existing namespace
dbrDetach	Disconnect from a namespace
dbrDelete	Delete namespace
dbrPut	Store a named tuple
dbrRead	Non-destructive retrieve of first tuple
dbrGet	Destructive retrieve of first tuple
dbrPutA	Non-blocking version of dbrPut
dbrTest	Test for completion of non-blocking request

Table 5.1: Example set of namespace and data access functions.

There are also a number of API extensions for put, read, and get, which provide scatter-gather lists for direct access to non-contiguous data in the application memory. This provides a powerful tool set for data transformation such as transposing from row-major to column-major representations.

System Library It is also called the back-end library, and acts as the translation layer from the non-blocking system library API to a particular data store. The primary interaction between system and client libraries happens via request and completion queues in conjunction with non-blocking API calls to post requests and test for completions. The system library maintains any required connections to the storage and needs to implement the semantics of the Data Broker requests in cases where the storage does not support it directly. For example, Redis has a blocking API, therefore the system library has to make sure that client API calls are not blocked.

Redis Back-End The existing version of Data Broker exclusively uses the Redis key/-value storage. While Redis provides ways to keep data persistent, it still limits the data size of a workflow to the amount of available memory. Especially if the Redis servers and the applications are co-located, the in-memory storage can become a bottleneck. There is also operational overhead for data recovery to make sure a restarted Redis cluster can serve the previously stored data. This includes the start of the exact amount of Redis nodes as before and the exact assignment of key ranges to each server before reloading the data into the servers. If the number of Redis nodes changes, the cluster needs to be resharded—an expensive operation. These challenges motivate the use of Ceph [3] as another storage back-end.

5.2.3 Object Based Storage

The earliest versions of distributed file systems, such as NFS [129] faced issues with scalability, load imbalance, and network hot-spots due to the co-location of data and metadata. In contrast, parallel file systems such as Lustre [8] and IBM Spectrum Scale [135] deliver improved data throughput by separating data and metadata management. In such parallel file systems, distributed storage servers offer high-performance storage with file systems semantics, at the expense of introducing another layer in the data path. With object based storage, this intermediary layer can be dispensed of, by offloading many aspects of metadata management to object storage devices [68].

Figure 5.4 shows the difference how Object-based Storage Devices (OSDs) transform storage management compared to traditional block-based storage. In the block-based model, the host kernel is responsible for the data layout and all application requests have to be translated by the kernel into logical block addresses. In the object-based model, the job of organizing the data on the storage medium is moved to the OSD. This eliminates the tiered file structure, places everything in flat address space, and thus provides a flexible access to data in the storage nodes. Next, we provide an overview of the Ceph Object Store.

5.2.4 Ceph Object Store

Ceph [3] provides a distributed object storage system and has become one of the most popular storage systems. Clients can use both the object interface and the legacy file systems simultaneously, making it suitable for both Cloud as well as HPC environments. Ceph's distributed architecture (Figure 5.5) provides high scalability, great performance,

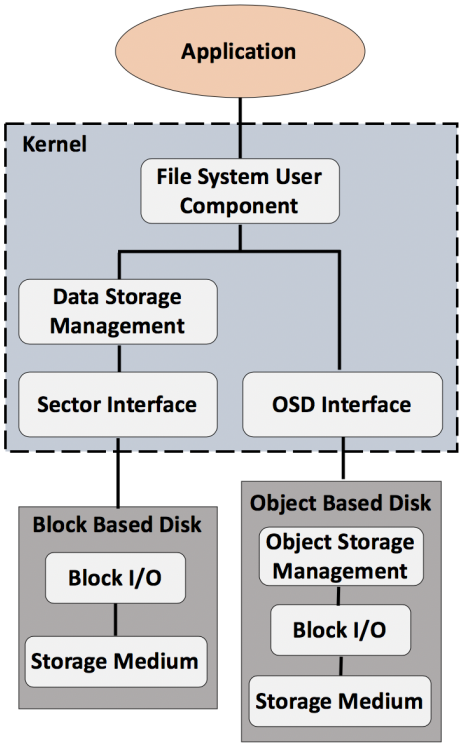


Figure 5.4: Block-based vs Object-based storage model.

and high reliability. Ceph object storage separates data from metadata and uses key/value access mechanism to distribute data as objects over multiple storage nodes. All these features make Ceph a suitable persistent storage option for Data Broker.

Ceph Storage Cluster

In Ceph, Object-Based Storage Device capability is handled by Object Storage Daemons (OSDs). Ceph Storage Cluster is infinitely scalable and is based upon RADOS (a Reliable, Autonomic Distributed Object Store) [163]. RADOS offers the data storage service that Ceph OSDs use to maintain high scalability and reliablity.

Data is stored as separate objects. Objects are members of the placement groups and the placement groups are distributed across OSDs using CRUSH (Controlled, Scalable, Decentralized Placement of Replicated Data) algorithm. CRUSH uses hierarchical cluster map and placement rules to map a placement group on the OSDs.

For distributed object storage, a Ceph storage cluster mainly consists of Ceph Monitors, Ceph Managers and Ceph OSDs. A *Ceph Monitor (ceph-mon)* maintains maps of the

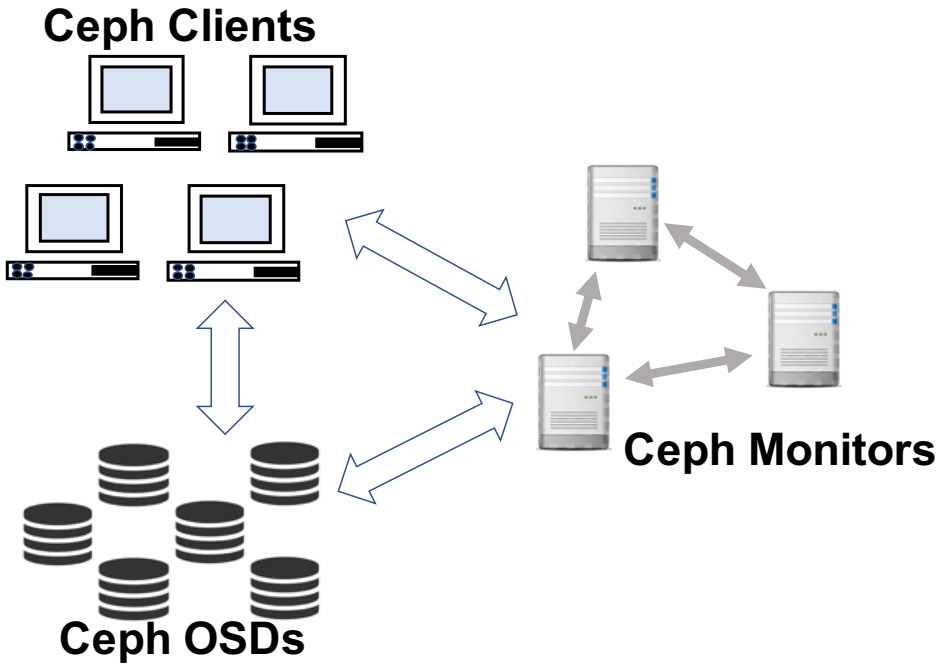


Figure 5.5: Ceph storage cluster.

cluster state, including the monitor map, manager map, the OSD map, the MDS map, and the CRUSH map. The maps are used by Ceph to maintain the cluster state by co-ordinating all daemons with each other. A *Ceph Manager* daemon keeps track of runtime metrics such as storage utilization, current performance metrics, and system load. A *Ceph OSD (ceph-osd)* stores data and provides some monitoring information to Ceph Monitors and Managers. It also handles recovery, load rebalancing, and data replication. In case of a node failure, all Ceph daemons, OSDs and monitors coordinate to balance the cluster, hence making it highly reliable and fault-tolerant. *Ceph storage cluster clients* retrieve a copy of the cluster map from the Ceph monitor to access the cluster and store and retrieve the data objects in the cluster.

Ceph Pools

The Ceph storage cluster stores data objects in logical partitions called *pools* to separate the data from different users or applications. Ceph pools define data durability methods, placement group counts and crush rules during the creation. For storing an object, a Ceph Client first creates an object and places it into a pool. Next, CRUSH algorithm assigns the object into a placement group and then maps the placement group to a single OSD or a set of OSDs.

5.3 System Design

We have implemented *Workflow Data Communicator* atop Ceph object-based storage cluster (Section 5.2). The similarity in the logical view of Ceph's pools and Data Broker's namespaces enable *Workflow Data Communicator* to efficiently map the data tuples to objects, hence providing reliable and persistent data sharing techniques. Although we focus in our design of *Workflow Data Communicator* to use Ceph object store, the approach can be extended for use in other object stores such as GlusterFS [78]. In the following, we discuss the system architecture and components of *Workflow Data Communicator*.

5.3.1 *Workflow Data Communicator* Architecture

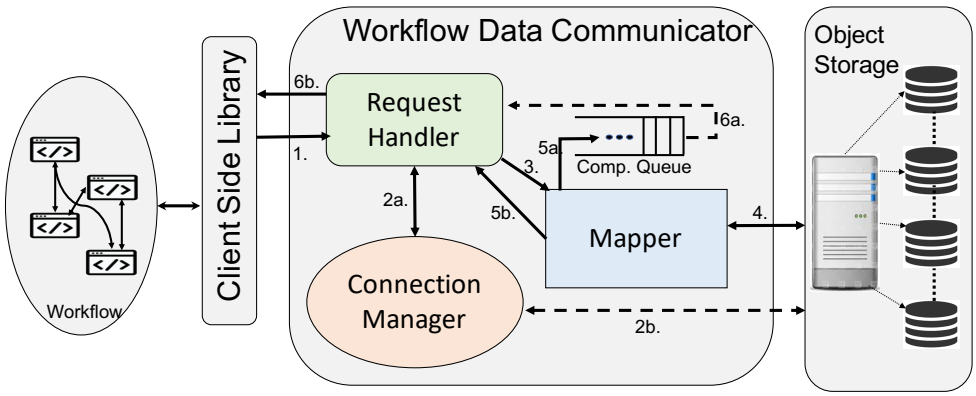


Figure 5.6: Overview of system architecture.

The overall architecture of *Workflow Data Communicator* is shown in Figure 5.6. Applications in a scientific workflow use client-side library to access the system. The Data Broker client library provides an API for the users to create, manage and destroy a namespace and add immutable data in the form of tuples. *Workflow Data Communicator* interprets the calls made to the client library, and maps them to the object-based storage cluster. The tuple data is stored and retrieved as immutable objects of data. *Workflow Data Communicator* provides namespace management and data access functionalities by mainly using three components: *Request Handler*, *Connection Manager*, and *Mapper*.

When an application in a scientific workflow submits a request to the client library, the request handler first collects the request from the client library. It then sends the request to Connection Manager to establish a connection with the storage cluster. The connection manager after successfully establishing a connection to the storage cluster, notifies the request handler. The request handler then forwards all client requests to the

mapper along with an initialized back-end handle to facilitate communication with the storage cluster. The mapper takes as input the client requests' namespaces and tuples, and creates, manages, and provides access to respective pools and objects stored in the back-end storage cluster. After the completion of each request, the mapper sends the request to the completion queue along with the updated status and any associated data for subsequent requests, such as the pool-handle, which is the context to access the pools. Mapper also returns the updated request handle to the request handler that constitutes any requested data from the storage. After that, the request handler examines the requests in the request completion queue. It checks the status of the requests for success/errors, and any associated data related to current or future requests. Finally, the request handler notifies the client workflow about the status of the request and the associated output data, such as keys to access pools and objects, and contents of the objects.

5.3.2 *Workflow Data Communicator* Components

Request Handler

Workflow Data Communicator uses the request handler to interact with the client library. It handles the client requests and provides the response for each request to the client. The main components of a client request (based on parameters shown in Table 5.1) are: *name of the namespace*, *type of request*, *name of the tuple*, and *contents of the tuple*.

When the client library sends a request to *Workflow Data Communicator*, it also provides a send or receive buffer in the request handle for any data to be written or read from the object storage. The request handler performs the following functions upon receiving a request:

- *Issue a command to the connection manager* for establishing a connection with the storage cluster.
- *Pass the request handle to the mapper.*
- *Acknowledge the response from mapper* after completion of the request.
- *Check the completion queue* after each request acknowledgement.
- *Send the request completion status* along with any requested data back to the client.

Connection Manager

The main tasks of the connection manager is to establish, maintain, and remove connectivity to the back-end storage cluster based upon an application's request. To access OSDs, Ceph provides librados API [2] that is internally used by the connection manager. The two main tasks of connection manager are as follows:

- *Initialization of back-end connection:* When the request handler issues a command to the connection manager to create a connection, the connection manager first creates a storage cluster handle and sends a request to the cluster for connection. After a successful connection, the connection manager notifies the request handler that the cluster is ready for the subsequent requests, and passes the cluster handle within a back-end context handle to the request handler.
- *Removal of back-end connection:* When an application sends a request to the request handler to end a connection, the connection manager is notified, which then destroys the cluster handle and terminates the connection to the back-end storage cluster.

Mapper

Mapper uses librados [2] internally to map client requests to object-based storage cluster of Ceph. It first computes an object's primary OSD from object and pool name. The location of an object in an OSD is calculated from the object-keys using the CRUSH algorithm [162]. The mapper then stores or retrieves data in the form of objects from the identified location in the back-end storage.

1. *Mapping Namespaces to Pools*

   Figure 5.7 represents the one-to-one mapping of user namespaces to object pools. The application views the data as namespaces and immutable tuples. *Workflow Data Communicator* maps each namespace to a Ceph object pool while providing the same high-level view to applications. Accordingly, it maps the tuples associated with namespace to objects in the mapped Ceph object pool. *Workflow Data Communicator* gives each object in the object pool a unique name (key), similar to a tuple-key. After creation, the keys to the object pools and objects are returned to the applications for subsequent requests.

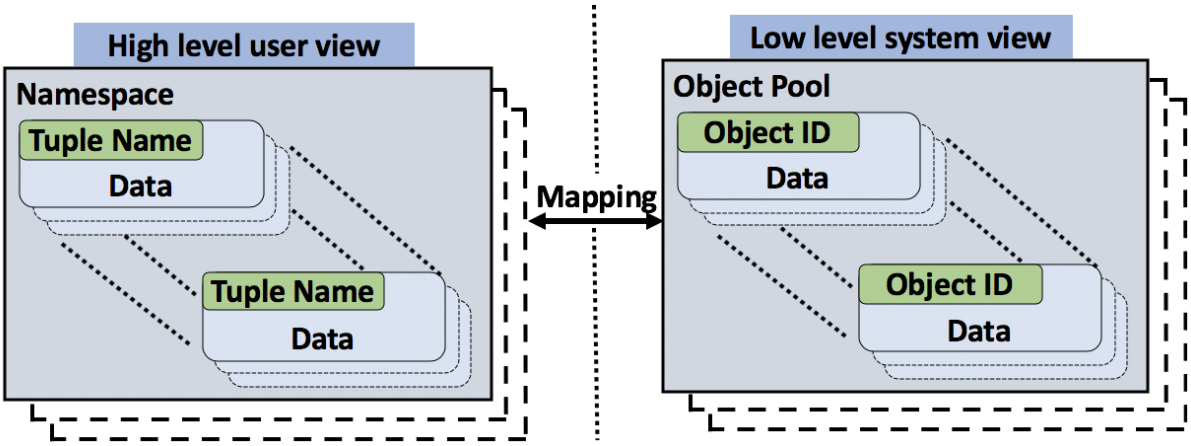


Figure 5.7: Mapping namespaces to object pools.

2. *Mapper Algorithm*

Algorithm 4: Processing Client Requests *(Create, Put, Read)* in Mapper.

Input: Request Pointer *req*, Initialized Backend Handle *BE*
Output: *RequestHandle*

```
1  switch (req.opcode):
2      case create:
3          poolname = req.key
4          Create new Object Pool with poolname
5          createCompletion(poolHandle, status)
6      case put:
7          poolname = req.poolhandle
8          objectID = req.key
9          Write Tuple Data into Object
10         createCompletion(countOfInsetedElements, status)
11     case read:
12         poolname = req.poolhandle
13         objectID = req.key
14         Retrieve Object Data
15         createCompletion(sizeOfRetrievedData, status)
16     return RequestHandle
```

Algorithm 4 shows the steps taken by the mapper for processing three representative request types (Create, Put, and Read) out of the nine client request types supported by *Workflow Data Communicator* and shown in Table 5.1. The input to the mapper algorithm is a request pointer (*req*) to the structure of the request containing the request definition and an initialized back-end handle (*BE*), which is given to the request handler by the connection manager. The output is an updated request handle which the mapper generates after completion of each request.

- If the request type is `create`, the mapper generates an object pool name that is mapped to the namespace key given in the request definition, and returns

the pool handle to the request handler through the completion queue. The client uses the pool handle for subsequent requests.

- In case of a `put` request, the mapper writes the tuple data in the objects that are created with an object identifier that is the same as the tuple-key. The data to be written is received by the mapper from the request pointer. The mapper reads the data from the sender buffer associated with the request structure.
- For a `read` request, the mapper reads the workflows data from the object specified by the object pool key in the request definition, and returns the data in the send buffer.

After completion of each request, *createCompletion()* sends the request to the completion queue. The request handler then checks the associated status (success or error) of each request.

5.3.3 Functions in *Workflow Data Communicator*

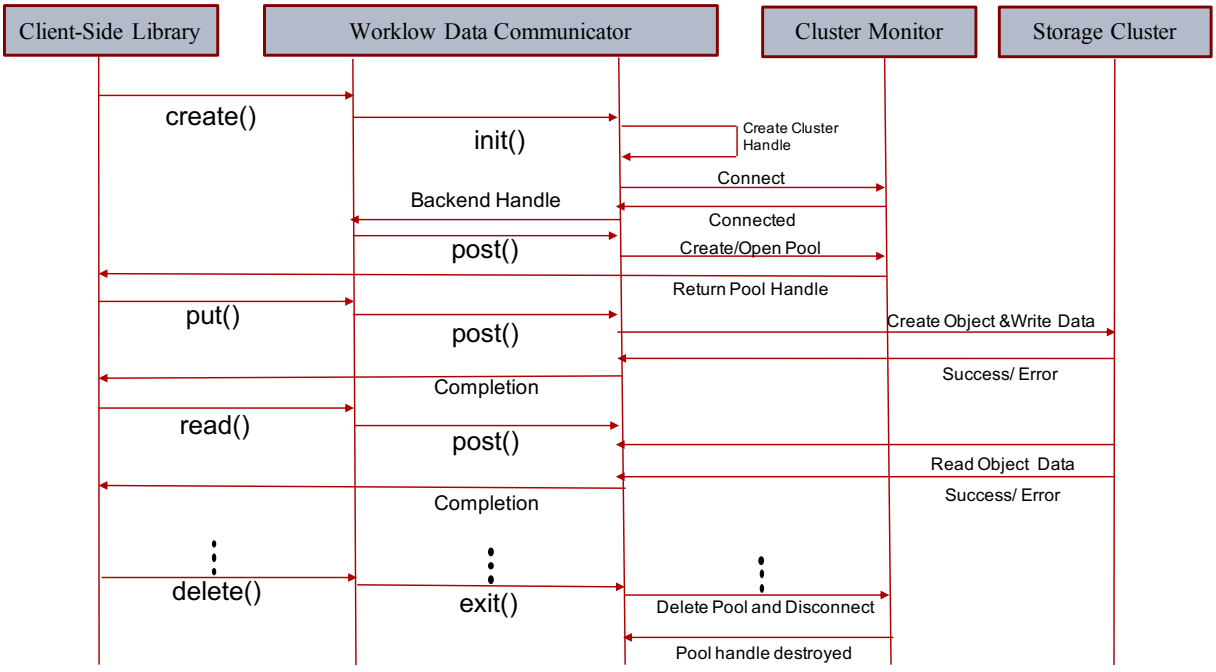


Figure 5.8: Sequence diagram of *Workflow Data Communicator*.

A sequence diagram for client requests in *Workflow Data Communicator* is shown in Figure 5.8. *Workflow Data Communicator* implements each client request as non-blocking functions. The basic functions used by *Workflow Data Communicator* are described below.

- *init():* This performs any initialization necessary to support other functions, such as creation of a storage back-end handle to connect to the storage cluster. It is also responsible for creating a *completion queue* where completed requests are placed by the mapper and the notification of request completion is sent to the client by the request handler.

- *post():* The upper layer (client library) uses this function to post a new request to the back-end. *Workflow Data Communicator* processes the user requests in a non-blocking operation. Some examples of requests include create/remove an object, get/read object data, create/delete a namespace, and attach/detach to an existing namespace as shown in Table 5.1.

- *fetch():* Request handler uses this function to fetch a completed request from the completion queue and returns a pointer to the request.

- *exit():* This is invoked at the end to destroy all connections with the back-end storage cluster.

Summary: *Workflow Data Communicator* runs atop object storage cluster and provides seamless data sharing and communication for workflow applications. All three components of *Workflow Data Communicator*: request handler, connection manager and mapper, interact with each other to handle client requests by mapping them to the underlying persistent storage while hiding lower level storage details. Client library interacts with *Workflow Data Communicator* through request handler to send requests and receive responses. Connection manager take care of the connection with the back-end storage cluster, and mapper stores and retrieves data to and from the storage cluster. *Workflow Data Communicator* provides data persistence, reduces the operational overhead of using in-memory data store, and facilitates the storage and retrieval of large amount of shared data in an efficient way.

5.4 Evaluation

We evaluate the performance of *Workflow Data Communicator* on a 8 TB Ceph Object Store running Ceph version 10.2.11. The cluster has one client, one monitor, two OSDs of 4 TB each and one metadata server. All nodes run CentOS 7.7 and have 8 cores, 3.4 GHz processor, and 16 GB memory. Each experiment is run five times and the mean results are reported in the following.

5.4.1 Benchmarking

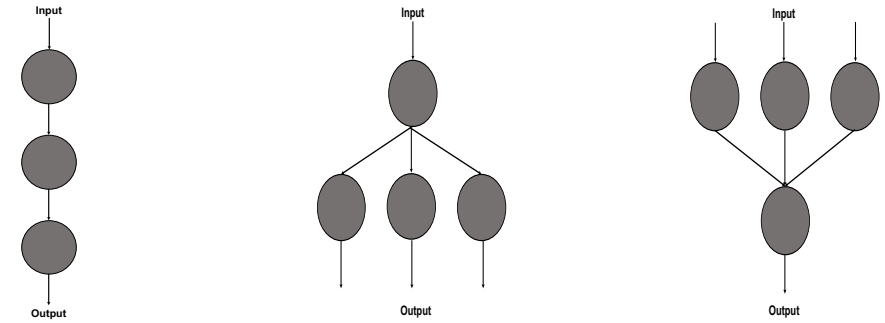
We benchmark *Workflow Data Communicator* and compare the performance of reading and writing 1 MB objects with Ceph's default client library, *librados* [2]. Table 5.2 shows the result. It is observed that *Workflow Data Communicator*, in spite of managing all of the extra functionalities of Data Broker, incurs minimal overhead and performs similar to *librados*. *Workflow Data Communicator* manages 223 *put* requests (creating and writing in objects in storage) per second as compared to 238 put requests per second in *librados*. For *get* requests (reading objects from storage), *Workflow Data Communicator* achieves 218 requests per second, whereas *librados* manages 222 get requests per second. The observed I/O throughput for the two cases is also similar. *Workflow Data Communicator* manages to perform at a similar rate as *librados* with a CPU and memory utilization of less than 1% and 20%, respectively.

	Default (librados)	***Workflow Data Communicator***
#PutReqests/second	238	223
#GetRequests/second	222	218
I/O throughput (MB/sec)	229.88	219.78

Table 5.2: Benchmark results of *Workflow Data Communicator*.

5.4.2 Data Access Patterns

We evaluate *Workflow Data Communicator* with three different data access patterns observed in real-world application workflows [147].



a) Pipeline. b) Broadcast. c) Reduce.

Figure 5.9: Data access patterns.

Data Access Patterns

We employ the following access patterns.

1. **Pipeline pattern:** Figure 5.9a shows this pattern that represents a set of tasks, chained together in a sequence such that the output of one task becomes the input of second task. We realize this data access pattern by creating one object using a *put* request, reading the contents of the same object using *read* request, and then writing the contents in a new object using a *put* request.

2. **Broadcast pattern:** Broadcast access pattern as shown in Figure 5.9b is the one where the output of one task becomes the input of multiple similar tasks. We realize the broadcast data access pattern by *getting* one object and the output is *put* to multiple objects.

3. **Reduce pattern:** As shown in Figure 5.9c, reduce access pattern is where a single task gets its input from the output of multiple similar previous tasks. We evaluate *Workflow Data Communicator* with the reduce pattern by *reading* and concatenating multiple objects, and then *putting* the result into one object.

Performance Results

We evaluate the performance of *Workflow Data Communicator* and compare it to Ceph's default client library. In the following graphs, *WDC* represents *Workflow Data Communicator* and *Default* represents *librados*.

1. **Storage Throughput**

   In case of *pipeline* data access, we create an object of 10 MB. A subsequent *read* request reads the contents and *put* creates a new object taking input from previous *read*. Ten objects, each of 10 MB is created resulting in a total storage of 100 MB. The total processing time *Workflow Data Communicator* takes is similar to *librados* as shown in Figure 5.10.

   A similar result is also seen for *broadcast* data access pattern to *put* ten objects of 10 MB each, with a total of 100 MB storage.

   For *reduce* data access pattern, ten *put* requests create 1 MB object each, which then invokes ten *read* requests, and then a final *put* request concatenates the contents of ten objects into one object of 10 MB. So overall, 20 MB of objects are stored in the storage in one data access workflow. Figure 5.10 shows that *Workflow Data Communicator* is able to achieve similar performance as *librados*, without adding any substantial overhead.

2. **Latency**

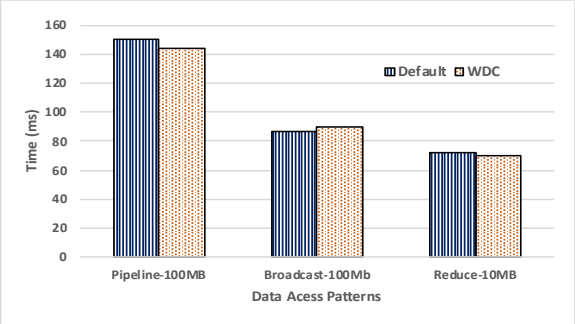


Figure 5.10: Processing time (ms) for data access patterns.

We measured both commit and apply latencies of the storage cluster while running the three data access patterns using both *Workflow Data Communicator* and *librados*. Commit latency is measured by the total time it takes for an operation to be applied to disk—similar to the time taken to write a metadata journal entry. Apply latency is the time taken for an operation to get applied to the file system (which can be throttled by various things to prevent us getting arbitrarily large amounts of dirty data).

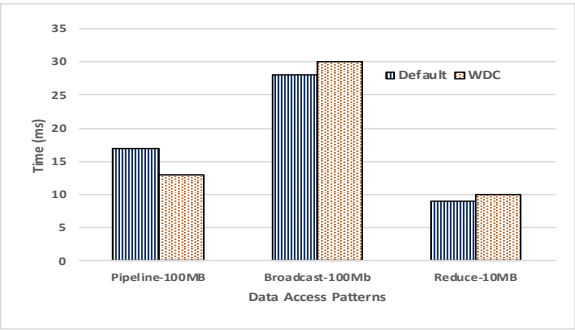


Figure 5.11: Average commit latency (ms) for data access patterns.

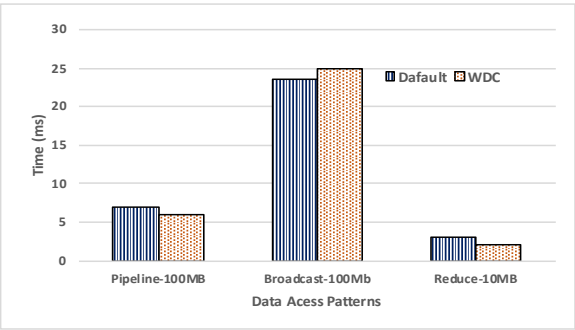


Figure 5.12: Average apply latency (ms) for data access patterns.

Figures 5.11 and 5.12 show the commit and apply latencies, respectively, for each data access pattern for *Workflow Data Communicator* and compares it with *librados*.

We observe that *Workflow Data Communicator* performs similar to *librados* for all three data access patterns.

Scalability

To evaluate the scalability of *Workflow Data Communicator*, we ran a workflow with *broadcast* data access pattern for different object sizes ranging from 10 bytes to 1 MB, for a total data ranging from 10 KB to 1 TB. Figure 5.13 shows the sprocessing time of *Workflow Data Communicator* to process total data ranging from 10 KB to 1 GB and compares it with *librados*. Figure 5.14 shows the processing time for total data ranging from 10 GB to 1 TB. We found that the processing times were similar for both *Workflow Data Communicator* and *librados*. Similar results were exhibited for both *pipeline* and *reduce* data access patterns.

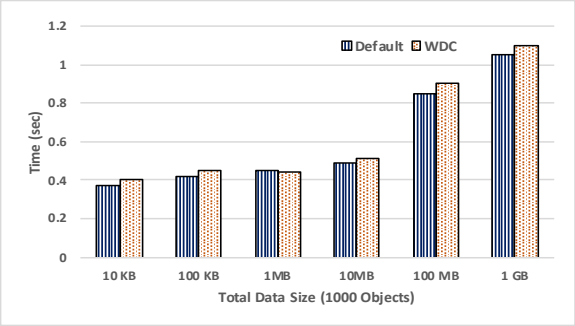


Figure 5.13: Processing time (sec) on small scales for broadcast data access pattern.

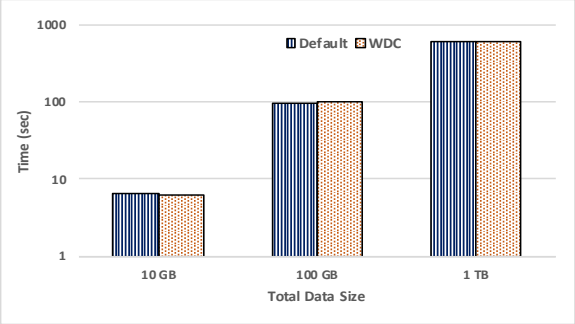


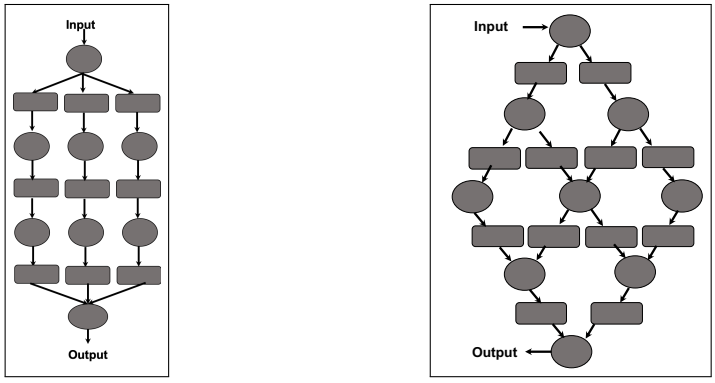
Figure 5.14: Processing time (sec) on large scales for broadcast data access pattern.

5.4.3 Workflows

We evaluate *Workflow Data Communicator* with two types of workflows: Benchmark Workflows [158], and Real World Workflows [137].

Benchmark Workflows

We use two benchmark workflows to evaluate *Workflow Data Communicator*: Fork&Join, and Lattice.



a) Fork&Join. b) Lattice.

Figure 5.15: Benchmark workflows.

1. **Fork&Join:** As shown in Figure 5.15a, Fork&Join is a combination of broadcast, pipeline and reduce data access patterns. In intermediate states, it creates files to pipeline the data. The Fork&Join structure is characterized by the number of stages and fan-out factors. We implement this benchmark workflow by *broadcasting* the output of one object to three different files. The intermediate jobs create and read objects in a *pipeline* manner. Finally, the output from the last intermediate stage is *reduced* to one object.

2. **Lattice:** Figure 5.15b shows the Lattice benchmark workflow. The input and output of the workflow are obtained by doing a *put* and *get* operation on the objects. For all intermediate states, the data gets *broadcasted* and *reduced* into files and objects, and finally *reducing* the whole workflow data to one object. The Lattice structure is characterized by its width and height. Figure 5.16 shows the processing time for both Fork&Join and Lattice benchmark workflows.

   Workflow Data Communicator manages to achieve better performance than *librados* in case of Fork&Join, and similar performance as *librados* for Lattice workflow.

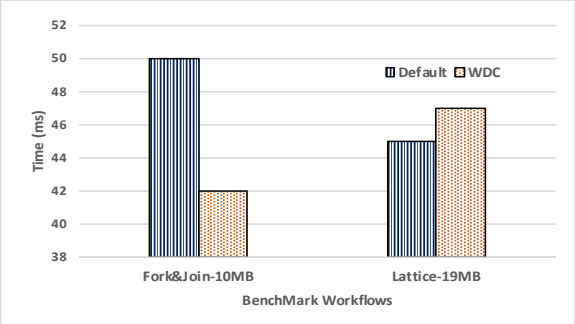


Figure 5.16: Processing time (ms) to run workflows.

Real World Workflows

We test *Workflow Data Communicator* with two real world workflows - Montage [10] and Brain Atlas [57].

1. **Montage:** This workflow creates a mosaic with ten astronomy images by using eight analysis kernels, and is composed of 36 tasks. To run a representative Montage workflow on *Workflow Data Communicator*, we create ten input objects of 5 KB each. When the workflow completes, Montage creates two output objects (represented as 'J' in Figure 5.18), of 4.5 MB each. Overall, the whole workflow stores and reads 93.7 MB of data and creates a total of 94 objects. The overall processing time for *Workflow Data Communicator* to run Montage workflow without the analysis kernels is 230 ms, achieving an I/O throughput of 407 MB/sec.

2. **Brain Atlas:** The Brain Atlas workflow [57] creates population-based brain atlases from the fMRI Data Center's archive of high resolution anatomical data [13]. To represent Brain Atlas in *Workflow Data Communicator*, we create two types of input objects - 'i' (image) and 'h' (header), as shown in Figure 5.17 of sizes 1 MB and 0.25 MB respectively. The workflow takes 209 ms on average to complete with *Workflow Data Communicator* and generates three output objects 'g' (graphic) of size 10 MB each. Overall, the whole workflow creates thirty objects generating a total data of 96.25 MB, and achieving an I/O throughput of 461 MB/sec.

In summary, *Workflow Data Communicator* efficiently combines the functionalities of both Data Broker and Ceph object store and facilitates efficient data communication and sharing for scientific workflows. *Workflow Data Communicator* mitigates the challenges of using in-memory key-value store for Data Broker, and enables Ceph to be workflow-aware. The design provides persistent data storage facility to Data Broker at a minimal overhead with performance matching that of the default Ceph *librados* client library that does not offer workflow support.

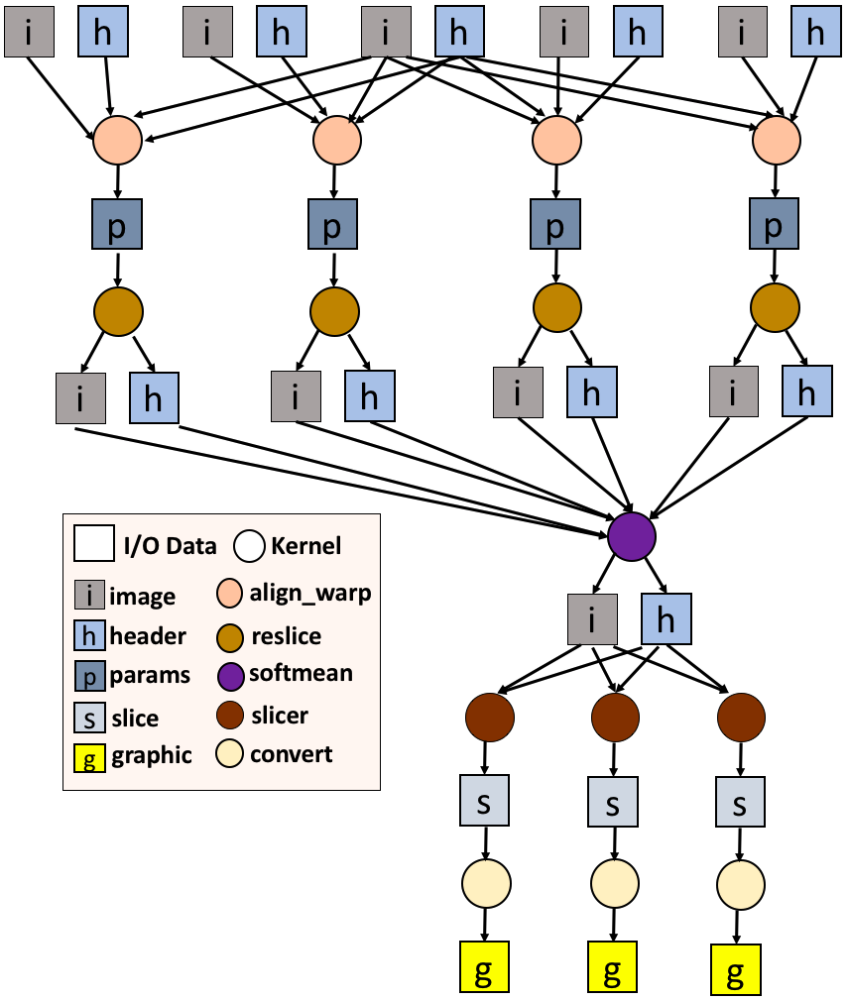


Figure 5.17: Brain Atlas workflow.

5.5 Chapter Summary

We have presented *Workflow Data Communicator*, a tool to facilitate efficient data communication among applications in complex scientific workflows. *Workflow Data Communicator* uses key-value based object storage to enable persistent and reliable data sharing. It employs a simple yet efficient design that allows Data Broker to select from a broader set of back-ends depending on the desired data performance, persistence, and fault tolerance.

We evaluate *Workflow Data Communicator* on a 8 TB Ceph object store and show that our approach gives comparable performance to Ceph's default client API—*librados*—in terms of latency and I/O throughput for three different data access patterns and benchmark workflows. *Workflow Data Communicator* is scalable and also works efficiently

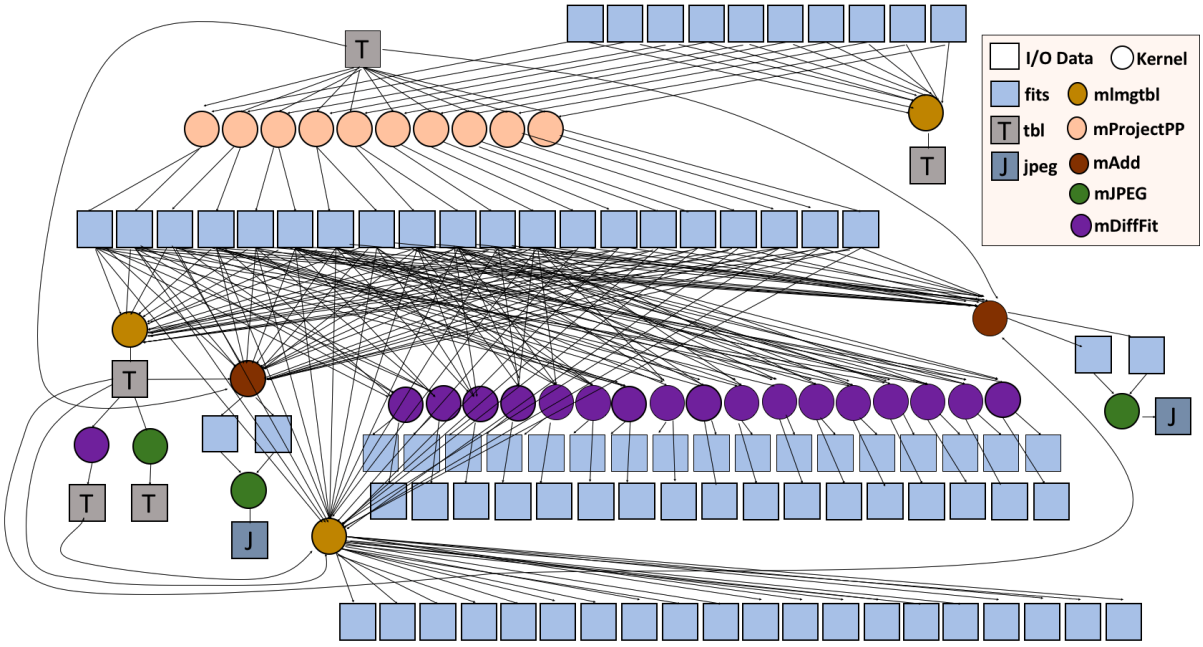


Figure 5.18: Montage workflow.

for real world workflows such as Montage and Brain Atlas. *Workflow Data Communicator* is able to provide efficient data persistence while overcoming deployment and memory constraint challenges faced by in-memory stores. Our future work involves extending *Workflow Data Communicator* for other back-end storage systems such as Lustre and GlusterFS.

Chapter 6

Data Management in Multi-layer Storage Hierarchy

6.1 Introduction

Parallel I/O for scalable computing systems such as HPC deployments need a transformative upgrade in the era of exascale computing to support emerging systems with deep memory and storage hierarchies. Realizing efficient storage and I/O mechanisms entails designing specialized software that takes full advantage of components introduced in these hierarchies.

As shown in Figure 6.1, upcoming exascale storage system architectures consist of main memory, node-local persistent memory, compute-edge persistent memory/storage (a.k.a. burst buffers), a disk-based parallel file system, and tape archives.

Traditional file and block based storage systems face severe challenges to meet the requirements of scientific applications. The dependence of parallel file systems on the POSIX-IO [153] enforces strict, costly data consistency requirements and lacks semantic data abstractions. However, these features are crucial for data movement while preserving semantics, storage and retrieval of scientific applications' data, especially in deep memory and storage hierarchies. In addition, parallel I/O makes it challenging to shoehorn new interfaces, such as taking advantage of multiple layers of storage and support for analysis in the data path. In the cloud computing environments, object-based storage systems, such as Openstack Swift [14], have become quite popular. However, most of these systems have been developed to store immutable cloud data and do not consider storage of large scale HPC application data. Other object-based storage systems such as Lustre [90] and Ceph [159] do support HPC applications. High-level I/O interfaces, such as HDF5 [74], manage data arrays and attributes as objects. However, the existing file systems and high-level library solutions do not support data storage in multiple layers of storage hierarchy, and just store the data in to one layer (disk- or SSD-based storage) of the stack. Hence, there is a crucial need for an efficient data storage and I/O system for HPC, which supports

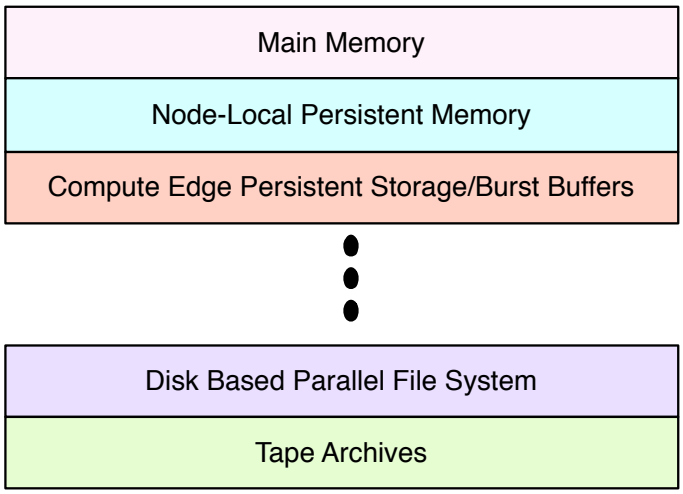


Figure 6.1: Exascale system storage hardware stack.

the complex memory/storage hierarchy and facilitate transparent data movements among the multiple layers of exascale storage stack.

We propose an object-based storage interface to transparently store data in upcoming HPC storage hierarchy, and to manage semantics of data. This approach provides the flexibility of storing data into multiple layers of storage hierarchy. It provides the application users the options to store data partially into different layers of storage stack, thus enhancing performance by storing part of data closer to analysis process as needed. To achieve this flexibility, our proposed storage model targets storing scientific data using an object abstraction that remains uniform across the different layers of storage. The uniformity ensures that data can be moved between layers without unnecessary translation that can loose semantic information. It also supports optimizing storage of the data and the metadata inside the object and container abstractions in the storage hierarchy. These objects can move between the layers of exascale storage stack efficiently, and provide enhanced flexibility and data consistency. The object-based abstractions can store complex data structures such as multidimensional arrays and key-value stores in a consistent way in all layers of the storage stack, enabling high scalability and efficient management.

We implement the object-based data mapping atop the MPI-IO [143] interface and provide multiple options to store the applications' data into different layers of storage system (such as SSD-based burst buffers and/or disk-based Lustre file system), according to the user requirements. We illustrate the utility of our approach by applying it to two science simulation use cases: VPIC-I/O [52] and HACC-I/O [80]

6.2 Background

We studied a number of object-based systems like Lustre, Ceph, MarFS, DAOS, Apache Spark, OpenStack Swift etc. and found that these storage systems provide various benefits to manage and store data such as high performance, security, and scalability. But some of these systems are mainly used for cloud computing and are optimized for storage of immutable data, and thus may not be applciable to the HPC use cases.

Lustre and Ceph have been successfully supporting HPC application data at scale, but these parallel file systems for HPC manage data as streams of bytes via POSIX I/O and susceptible to limitations at exascale due to lack of inherent support for object based abstractions. This is especially problematic as HPC applications face highly concurrent writing and reading on the same file during the course of an application. Also these existing object-based systems do not support any abstractions or interfaces to expose the internal layers of a deep memory and storage hierarchy. Hence, current object-based storage systems are suboptimal when it comes to supporting HPC and upcoming deep memory storage hierarchy in exascale systems. They lack a uniform and consistent data interface for each layer of the storage stack, which can facilitate storage and input/output of scientific application data, and support seamless data movement across hierarchies. Simple and efficient methods for data management and storage through this hierarchy are critical for sustaining storage systems for scientific applications at exascale.

6.2.1 Scientific Applications Use Cases

We have used two scientific applications, VPIC-I/O [52] and HACC-I/O [80], for presenting the design and implementation of our proposed object-based data abstractions.

VPIC-I/O

VPIC [48] is a particle physics simulation developed by Los Alamos National Lab. It is designed to simulate field data and particle data and minimize data motion. It is a highly scalable and optimized plasma physics simulation. The field data includes information such as electric and magnetic field strength, and the particle data include information about the field's position, momentum, and energy. VPIC-I/O [52], is a kernel extracted from VPIC's particle I/O pattern. Currently, it uses H5Part API [22] for issuing HDF5 calls to create a file and write particle data to multiple HDF5 datasets. We use VPIC

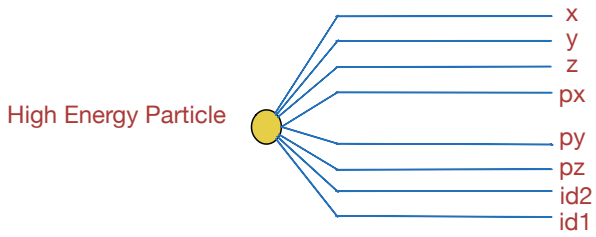


Figure 6.2: Properties of a particle in a VPIC simulation.

particle data to show mapping of the scientific data into our object-based abstractions. Each particle of VPIC simulation consists of eight variables (shown in figure 6.2) representing some attributes of the application. Each variable is stored as a one dimensional array. In [52], the VPIC simulation contained 1 trillion particles. We map each particle of data into two types of objects: basic objects and composite objects. (More details are presented in Section 6.3.1).

HACC-I/O

The Hardware Accelerated Cosmology Code (HACC)[80] application uses N-body techniques to simulate the formation of structure in collisionless fluids under the influence of gravity in an expanding universe. The HACC-I/O benchmark captures the I/O patterns and evaluates the performance for the HACC simulation code. Similar to VPIC-I/O, we also map the application data for HACC-I/O to our proposed object-based data abstractions and use the proposed object storage design model for I/O.

6.3 Design

6.3.1 Object Definition

The typical data structures used by scientific applications to represent data are multidimensional arrays. We define an object comprising a multi-dimensional array that contains either one property or multiple properties of a physical scientific object. An object can also contain an associative array (key-value store). Each object contains metadata (object ID, name, dimension, datatype, etc.), payload (actual data of a multi-dimensional array), extended metadata (attributes or properties of the payload, locations of products of data, etc.), and provenance (time of creation, owner of an object, time and user of accesses,

etc.). A high-level view of an object and its components is shown in Figure 6.3. Multiple objects can be packed into a container to manage them as a collection. One can expand this definition to add more metadata and to support more data structures as payload.

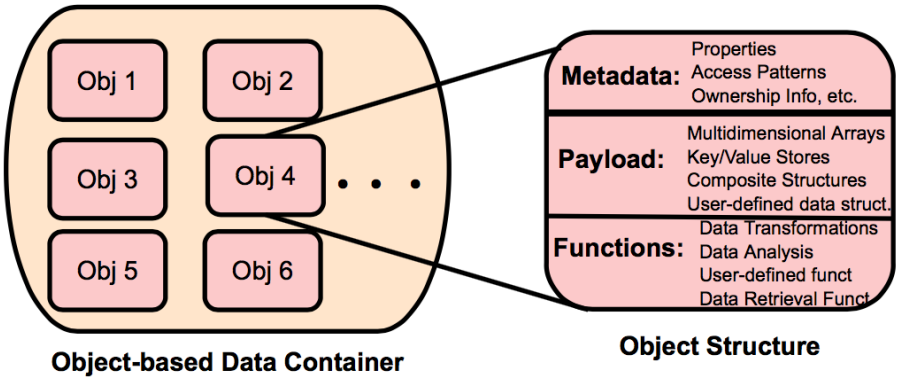


Figure 6.3: The proposed object structure.

In parallel applications, objects are created in parallel on all participating processes, but have a global view of a single object. Each process creates a set of individual local sub-objects (one object per process), which are considered collectively as a subset of the global object (or object container). Once objects and containers are created, application data is mapped into them using object API, and *container-id*'s and *object-id*'s are returned to the client for future access.

An object can either represent a single property of a set of physical objects or a data structure containing multiple properties of the objects. We name the former as *basic objects*, and the latter as *composite objects*. Depending upon the application requirement, either basic or composite objects can be created on the compute nodes.

Figure 6.4 and Figure 6.5 show the basic and composite object for VPIC particle data, for `n` number of particles respectively. For composite object, R_i is the composite record for particle `i`.

6.3.2 Object-based Storage Model for Multi-layer HPC Storage

In this section, we first present the object APIs that we provide to applications, then we discuss our system architecture, followed by the object storage design model for efficient objects storage.

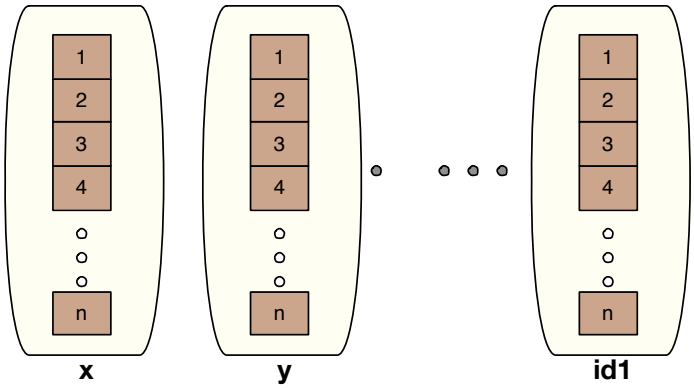


Figure 6.4: VPIC data represented as Basic Objects.

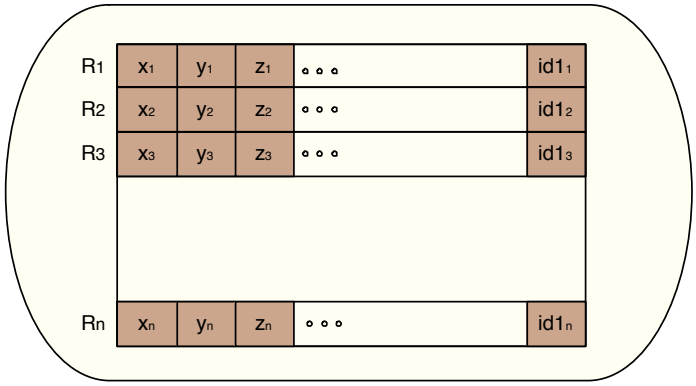


Figure 6.5: VPIC data represented as Composite Object.

Object APIs We provide a number of object APIs to the application as listed in Table 6.1. The APIs support object management functions such as creating containers and objects, and storing the objects into different layers of the storage system. Figure 6.6 shows a comparison of our new object APIs with existing HDF5 calls for VPIC-I/O. Our design model offloads from the users to our system the task of managing MPI-IO or HDF5 calls to the separate layer. A key advantage of uniform object APIs is that data can be moved between layers without modification, which preserves semantics and enables seamless migration of data between storage hierarchy layers as needed.

System Architecture Figure 6.7 presents the overall system architecture of our approach. P_{nk} is the k^{th} process on n^{th} compute node N_n. O_{nk} is the object created by process P_{nk}, S_n is n^{th} object-shard created by group of processes on N_n. Burst Buffers are SSD-based storage system. The parallel file system is HDD based, e.g., Lustre. The application interacts with the system at the top layer that exports the object APIs to the user. In this layer, our object model creates data containers and associated objects

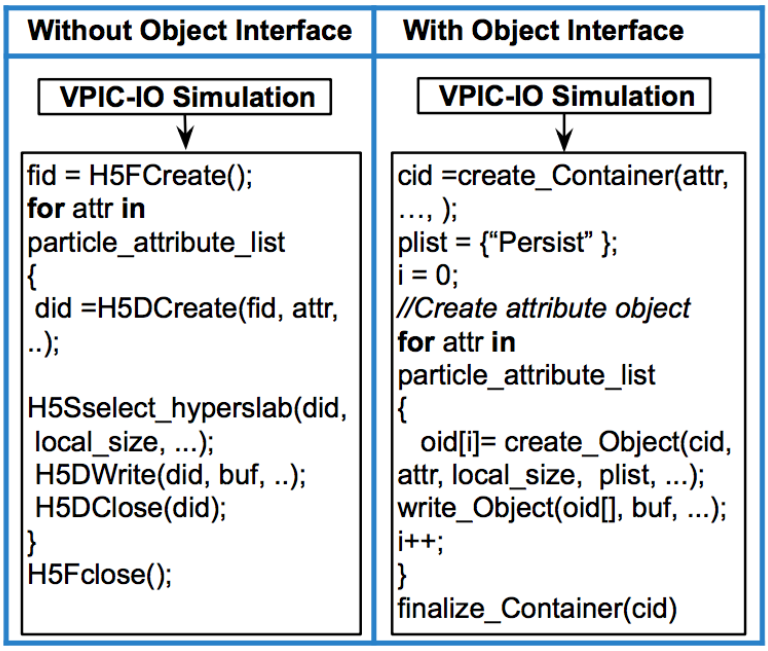


Figure 6.6: Data Storage for VPIC-I/O: HDF5 Vs. Object Interface.

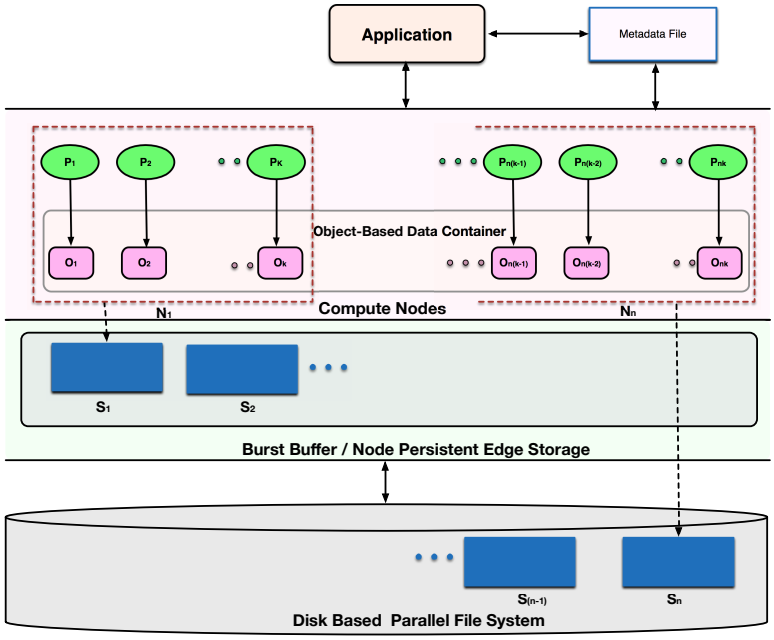


Figure 6.7: High-level architecture of the proposed object storage system.

Object API (provided to the application users)	
`cid=create_Container(attr, ..)`	Creates the data container with a global view which holds all the objects. Returns the container-id cid to user
`oid[i]=create_Object(cid, attr, local_size, plist, ..)`	Creates the objects for each process, with application data, semantic information and other attributes attr and properties list plist, Returns object-id oid to use
`write_Object(oid[], buf,... )`	Creates a memory buffer buf for in-memory computations, and writes objects to storage system
`finalize_container(c_id)`	Deletes the container and its component objects

Table 6.1: Object APIs for creating containers/objects and writing to the storage system.

on the compute nodes and returns the *container-id* and *object-id*s to the user. There are n compute nodes, and each of them runs k processes, P_1 to P_k, to map application data into objects, O_1 to O_k. Each process creates one object locally in the memory of the compute node on which it executes. Another component of this layer is the object-based data container, which provides a global view of all the objects to the application.

The next component of the system is SSD-based burst buffers/nodes that offer persistent edge storage. This layer provides a closer (to compute) and faster storage system for meeting high performance needs. Objects created on the compute nodes are pushed onto this layer for short-term storage. To store the objects, our object storage model creates multiple object-shards (S_1, S_2, ..) in this layer.

The third and the last layer of the system comprise a disk based parallel file system, e.g., Lustre. This layer provides long-term storage for objects (in the form of object-shards) that are in compute nodes or burst buffers.

Object-based Storage Model Our storage model uses MPI-IO interface for mapping the data objects into files. The objects can be stored in the burst buffers, in the disk-based file system, or both according to the application requirements.

6.3.3 Object Creation and Data Mapping Process

First, as described earlier, our proposed approach creates an object-based data container that contains all the objects created on the compute nodes and set its properties such as its type, lifetime, and state. A unique *container-id* is assigned to this container, which is returned to the user after its creation and setup. Once a data container is created, each process creates a data object into this container to map the application data using the object-creation APIs listed in Table 6.1 . Based on the application requirements, either basic or composite objects are created. Next, application data is mapped into these objects using the object-mapping APIs and the *object-id*s are returned to the user. Objects sharing the same node-local memory are called sub-objects.

Once the objects are created in memory and application data is mapped into these objects, the next step is to examine the application requirements and determine if these objects need to be kept in memory, or stored into the storage layers. The storage model assigns each application a storage quota, based on which some sub-objects are stored into burst buffers and the rest are stored in Lustre. For example, if the storage quota is assigned as (25:75), 25% of all the sub-objects from the applicaiton will be pushed to burst buffer and the rest of the objects to the lower storage layer, i.e., Lustre. Our object interface contributes to seamless semantic-preserving data migration between these storage layers.

The model maps the data objects into object-shards as follows. As shown in Figure 6.7, in the top layer, the model divides all the processes into groups based on the nodes on which they run. Therefore, if there are n compute nodes and k processes on each node, we get n groups of processes where each group has k members. The grouping of the processes is based on the nodes so that all the local sub-objects can be stored collectively in physical proximity. The storage model can classify the sub-objects into one group per node or one group for a collection of two or more nodes.

Once the data objects are grouped together, our storage model maps them into object-shards and pushes them to the burst buffers or the disk-based parallel file system, depending on the storage quota assigned to the application data.

We experimented with 5 different object-sharding strategies using a representative application (VPIC-I/O [52]) as shown in Table 6.2. Each strategy forms different number of object-shards per node for our use cases. The experimental details of these strategies are presented in the next section. We observed that Strategy-3, which creates $n/2$ object-shards for composite objects and $4n$ object-shards for basic objects performs the best for storing the objects in our studied application. These shards reside partially into the burst

buffers and Lustre file system. If required, application can access the object-shards from burst buffers, bringing them back into memory, or send them to the lower storage layer.

Our storage model preserves the information of each object-shard (such as its name and location) into the metadata portion of the object as well as a separate Metadata file. This approach helps us manage the mapping of objects to object-shards without using a separate metadata server.

In summary, our design enables a flexible object-based store with uniform APIs, and allows for seamless migration of data between different hierarchies of the HPC storage stack.

6.4 Evaluation

In this section, we present the implementation details of our object-based approach and its evaluation. Our implementation targets I/O of two scientific applications: VPIC-I/O [52], a particle physics simulation that simulates field data and particle data; and HACC-I/O [80], a benchmark that captures the I/O patterns and evaluates the performance for the HACC simulation code.

6.4.1 VPIC-I/O

In VPIC-I/O, original data structure contains 8 variables for each particle and these variables are stored as 1-D arrays using HDF5 datasets. Using our object interface, we implemented both types of objects – Basic and Composite. We mapped the particle data into objects, where each process creates one sub-object in the local memory of node to which it belongs. Then we used our object-based storage model to map these sub-objects into object-shards mainly using five object-sharding strategies (specified in Table 6.2) based on number of shards formed on n compute nodes, to which the sub-objects are mapped.

We ran all our experiments on the Cray XC40 system 'Cori' installed at the National Energy Research Scientific Computing Center (NERSC) for a scale of up to $16,384$ cores/MPI processes. We measured the 'I/O rate' obtained for storing these object-shards in a file system using our storage model and compared the performance with that of writing the same amount of application data using HDF5 datasets. The 'I/O rate' is the ratio of the

Strategy	No. of Object-Shards for n Nodes		Size of each Object-Shard (GB)	
	Basic	Composite	Basic	Composite
1	8	1	32-525	256-4100
2	8n	n	1	8
3	4n	n/2	2	16
4	2n	n/4	4	32
5	n	n/8	8	64

Table 6.2: Specifications of various Object-Sharding Strategies for VPIC-I/O.

amount of total data written to the time taken for storing the data in the file system. We store the object-shards in 2 ways: (1) All object-shards into Lustre (**Horizontal Sharding**); (2) A fraction of total number of object-shards into burst buffers and rest into Lustre (**Vertical Sharding**). For the VPIC-I/O experiments, each MPI process writes 8 M particles. We use a Lustre OST count of 248 and stripe size of 32 M for all of our experiments.

Horizontal Sharding

In our tests, we have created and stored object-shards for both basic and composite objects.

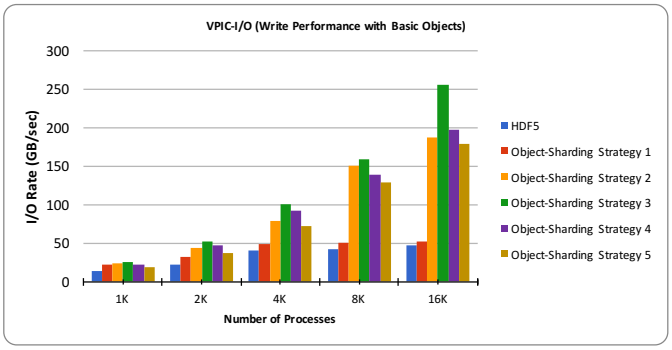


Figure 6.8: Write performance for VPIC-I/O Basic Objects.

Basic Objects In the case of storing basic objects, each process creates 8 sub-objects (one for each particle property or variable). Each of these sub-objects are grouped together based on the compute nodes, and then mapped into object-shards. Figure 6.8 shows the I/O rate obtained for storing the basic objects for five object-sharding strategies compared to that of HDF5 [74] dataset storage. We observed a performance improvement of up to 5× for storing all the objects under our approach (object-sharding strategy 3 giving the maximum performance). The performance difference rises significantly, especially on a scale higher than $4K$ (i.e., 4 x 1024 = 4096) processes. At higher scales, the larger

number of object-shards facilitates parallel storage operations, which significantly reduces the contention among processes.

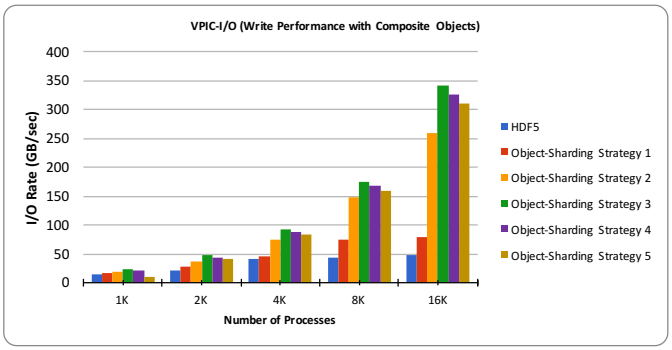


Figure 6.9: Write performance for VPIC-I/O Composite Objects.

Composite Objects Figure 6.9 shows the I/O rate obtained to store all object-shards for composite objects into Lustre. As shown in Figure 6.9, we observe that, similar to basic objects, the performance for object-sharding strategy 3 is much better for composite objects too. This storage performance is mainly achieved due to efficient grouping of processes based on nodes, as all the processes belonging to one node are topologically nearer to each other.

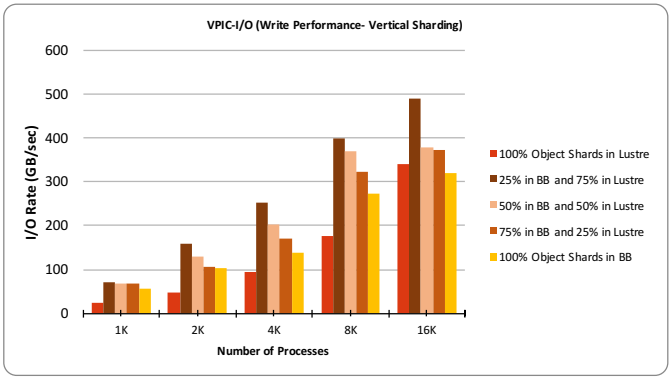


Figure 6.10: Write performance for VPIC-I/O using Verical Sharding.

Vertical Sharding

As described earlier (Section 6.3.2), one of the main contributions of our object storage design model is that, given its uniform storage APIs, it can store object-shards on multiple layers of storage. In this study, we have stored data partially on both SSD-based burst buffers and disk-based Lustre by assigning a storage quota for each application based on performance requirements. Some applications need to keep hot objects closer, which can be achieved by storing them into burst buffers.

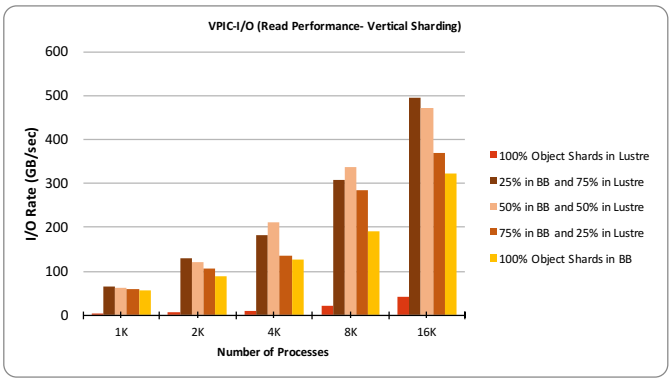


Figure 6.11: Read performance for VPIC-I/O using Verical Sharding.

In Figure 6.10, we show the performance of storing all object-shards for VPIC composite objects into Lustre compared to storing 25%, 50%, 75%, or 100% of them into buffers and rest into Lustre, using object-sharding strategy 3. We have statically chosen these distributions between the two layers to demonstrate the concept. Users may choose better distribution based on the knowledge of application requirements. We show in the figure a comparison of vertical sharding with horizontal sharding, where all object-shards are stored in Lustre. Storing all object-shards in burst buffers may not be feasible due to capacity limitations and by moving just 25% of the object-shards from Lustre to burst buffers, it provides up to 2.4× improvement compared to the case where all of them are stored in Lustre and up to 8× compared to HDF5 (Figure 6.9). Thus, our uniform API approach provides a flexible mechanism to bring objects that are being used frequently (hot data) closer to the computation in a seamless manner as needed.

We also measured the I/O rate for reading partial object-shards stored in the burst buffers and compared the performance with reading all of the shards from Lustre. In Figure 6.11, we show the performance of reading the object-shards for VPIC composite objects sharded vertically between Lustre and burst buffer using object-sharding strategy 3. As burst buffer layer is closer to computation, reading the object-shards partially from burst buffer improves the performance by up to 8× compared to reading all of them from the Lustre.

6.4.2 HACC-I/O

For HACC-I/O, we implemented the objects as composite objects. Each particle of HACC simulation consists of nine variables representing some attributes of the application. For all the experiments, we stored data for 8 M particles per process. All the experiments are done on the NERSC's Cori supercomputers for scale of up to 16384 cores. We used object-sharding strategies 1 to 4 (specified in Table 6.2) for horizontal sharding of HACC-I/O

application data using the composite objects. The baseline implementation of HACC-I/O uses MPI-IO [143] library.

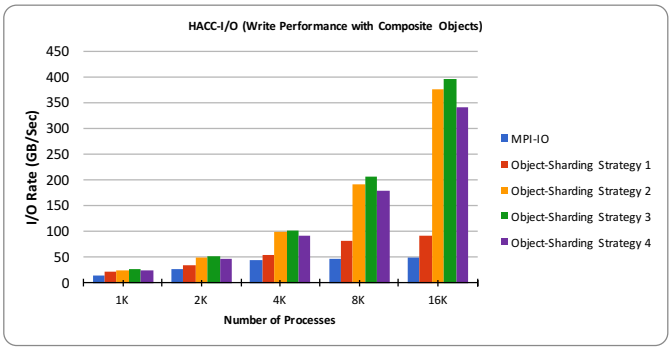


Figure 6.12: Write performance for HACC-I/O Composite Objects.

Storage Performance using Horizontal Sharding

We show a comparison of I/O rate for storing all the object-shards under the four object-sharding strategies with storing application data using MPI-IO into one single file in Figure 6.12. We observe that as in the case of VPIC-I/O, object-sharding strategy 3 (which creates $n/2$ object-shards on n nodes) performs significantly better. We observe a performance gain of up to 6× for storing the object-shards with strategy 3 as compared to MPI-IO. It can also be seen in the figure that performance gain increases manifold at higher scales. This performance gain is achieved by significant contention reduction among processes for storing multiple object-shards in contrast to single shared file architecture.

Read Performance using Horizontal Sharding We also measured the performance for reading objects from object-shards and compared the same with the MPI-IO single-file model. In Figure 6.13, we show that the I/O rate obtained to read all the objects shards using the three object-sharding strategies as well as using MPI-IO. We observed a performance gain of up to 30% using sharding strategy 3.

6.4.3 Chapter Summary

We have presented an initial design of a novel object based storage interface to facilitate storage and I/O for HPC applications in exascale systems that will comprise deep memory and storage hierarchy. Our interface maps scientific applications' data into objects that can store both simple as well as complex data structures along with their properties to

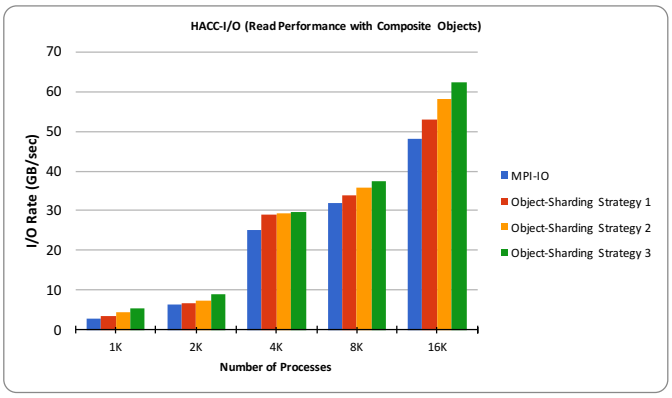


Figure 6.13: Read performance for HACC-I/O Composite Objects.

preserve semantic information in various layers of memory and storage. Our approach can offload the task of managing MPI-IO or HDF5 calls for data storage from the users to our system. To achieve high performance as well as storage efficiency, we store objects partially into SSD-based burst buffers or node-local persistent storage as well as on disk-based parallel file system (e.g. Lustre) based on application requirements. We have implemented our proposed model with two scientific use cases, VPIC-I/O and HACC-I/O, and evaluated it for a scale of up to $16K$ processes. Experimental results show that compared to the state of the art, our object-based storage model can improve the I/O performance by up to 7×. Based on this initial success, we are developing a runtime system to optimize sharding strategies, to move data asynchronously and pro-actively, and to support data processing while the data is in transit among different storage layers.

Chapter 7

Conclusion

Inefficient data management leads to poor I/O and storage performance in the HPC applications and scientific workflows. Some of the major challenges for efficient data management arise from poor resource allocation, load imbalance in object storage targets, inflexible data sharing between applications in a workflow and inefficient data management of data in multiple layers of storage This dissertation proposes, designs, and implements a series of novel techniques, algorithms, and frameworks, for scalable data management in object-based storage systems.

In the first part of the dissertation (Chapter 3), we presented our resource contention aware load balancing tool (`iez`) for large scale distributed object-based storage systems. We presented the design of an "end-to-end control plane" to optimize parallel and distributed HPC I/O systems, such as Lustre, by providing efficient load balancing across storage servers. Our proposed system, `iez`, provides global view of the system, enables coordination between the clients and servers, and handles the performance degradation due to resource contention by considering operations on both clients as well as servers. Our implementation of `iez` provides a balanced distribution of load over OSTs and OSSs in the Lustre file system. We evaluated `iez` on a real Lustre testbed using two representative benchmarks—IOR and HACC-I/O—with multiple stripe counts of files as well as SSF and FPP accesses. Compared to the default Lustre RR policy, `iez` provides up to 31.3% improvement in balancing the load. Furthermore, we observed an I/O performance improvement of up to 34% for reads and 32% for writes. Finally, the transparent design of `iez` makes it attractive for adoption in real-world deployments where access to application source code, needed for existing approaches, may not always be possible.

In Chapter 4 , we extended `iez` to support Progressive File Layout for object-based storage system: Luster. Compared to the default Lustre RR policy, `iez` provides up to 33% improvement in balancing the load. We also observed an I/O performance improvement of up to 43% for reads without affecting the performance for writes, for —IOR and HACC-I/O—with multiple stripe counts of files as well as SSF and FPP accesses.

In the second part (Chapter 5), we presented a technique to facilitate data sharing in

scientific workflows using object-based storage, with our proposed tool *Workflow Data Communicator*. *Workflow Data Communicator* uses key-value based object storage to enable persistent and reliable data sharing. It employs a simple yet efficient design that allows Data Broker to select from a broader set of back-ends depending on the desired data performance, persistence, and fault tolerance.

We evaluated *Workflow Data Communicator* on a 8 TB Ceph object store and show that our approach gives comparable performance to Ceph's default client API—*librados*—in terms of latency and I/O throughput for three different data access patterns and benchmark workflows. *Workflow Data Communicator* is scalable and also works efficiently for real world workflows such as Montage and Brain Atlas. *Workflow Data Communicator* is able to provide efficient data persistence while overcoming deployment and memory constraints faced by in-memory stores. Our future work involves extending *Workflow Data Communicator* for other back-end storage systems such as Lustre and GlusterFS.

In the last part of this dissertation (Chapter 6), we presented a solution for transparent data management in multi-layer storage hierarchy of present and next-generation HPC systems. Our initial design of a novel object based storage interface facilitates storage and I/O for HPC applications in exascale systems that will comprise deep memory and storage hierarchy. Our interface maps scientific applications' data into objects that can store both simple and complex data structures along with their properties to preserve semantic information in various layers of memory and storage. Our approach can offload the task of managing MPI-IO or HDF5 calls for data storage from the users to our system. To achieve high performance as well as storage efficiency, we store objects partially into SSD-based burst buffers or node-local persistent storage as well as on disk-based parallel file system (e.g. Lustre) based on application requirements. We have implemented our model with two scientific use cases, VPIC-I/O and HACC-I/O, and evaluated it for a scale of up to $16K$ processes. Experimental results show that compared to the state of the art, our object-based storage model can improve the I/O performance by up to $7\times$.

This dissertation proposes intelligent and scalable data management techniques that enhance efficiency, performance and flexibility of scientific applications and workflows. It shows that by employing our proposed scalable data management techniques, scientific applications' and workflows' flexibility and performance in object-based storage systems can be enhanced significantly. Our proposed data management strategies can guide next-generation HPC storage systems' software design to efficiently support data for scientific applications and workflows.

Bibliography

[1] Apache spark. https://wiki.openstack.org/wiki/Swift. Accessed: March 20 2017.

[2] Ceph, Librados. https://docs.ceph.com/docs/giant/rados/api/librados-intro/. Accessed: May 22 2020.

[3] Ceph Object Storage. https://ceph.io/. Accessed: May 15 2020.

[4] CYLC - A Workflow Engine. https://cylc.github.io/. Accessed: May 22 2020.

[5] Docker-Registry. https://github.com/docker/docker-registry.

[6] LANL, NERSC, and SNL, "APEX Workflows". https://www.nersc.gov/assets/apex-workflows-v2.pdf. Accessed: May 18 2020.

[7] LBL - Tigres. http://tigres.lbl.gov/home. Accessed: May 18 2020.

[8] Lustre File System. http://lustre.org/. Accessed: May 15 2020.

[9] modFTDock: Protein-RNA Docking. https://sites.google.com/site/swiftguide/swift-applications/modftdock. Accessed: May 15 2020.

[10] Montage - An Astronomical Image Mosaic Engine. http://montage.ipac.caltech.edu/docs/m101tutorial.html. Accessed: May 24 2020.

[11] Mpi forum. https://www.mpi-forum.org/. Accessed: May 20, 2020.

[12] NERSC - TaskFarmer. https://docs.nersc.gov/jobs/workflow/taskfarmer/. Accessed: May 18 2020.

[13] OpenfMRI Dataset. https://openfmri.org/dataset/. Accessed: May 24 2020.

[14] Openstack object storage. https://wiki.openstack.org/wiki/Swift. Accessed: Jul 22 2016.

[15] Redis. https://redis.io/. Accessed: May 15 2020.

[16] The Kepler Project. https://kepler-project.org/.

[17] Foundationdb, 2019. https://www.foundationdb.org/.

[18] Memcached, 2019. https://memcached.org/.

[19] Microsoft azure cosmo db, 2019. https://docs.microsoft.com/en-us/azure/cosmos-db/.

[20] Top 500. Top 500 supercomputers. Accessed: July 12 2018.

[21] Subil Abraham, Arnab K Paul, Redwan Ibne Seraj Khan, and Ali R Butt. On the use of containers in high performance computing environments. In *2020 IEEE International Conference on Cloud Computing (CLOUD)*, pages 1–10. IEEE, 2020.

[22] A. Adelmann, R. D. Ryne, J. M. Shalf, and C. Siegerist. H5part: A portable high performance parallel data interface for particle simulations. In *Proceedings of the 2005 Particle Accelerator Conference*, pages 4129–4131, May 2005.

[23] Megha Agarwal, Divyansh Singhvi, Preeti Malakar, and Suren Byna. Active learning-based automatic tuning and prediction of parallel i/o performance. In *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, pages 20–29. IEEE, 2019.

[24] Ravindra K Ahuja. *Network flows: theory, algorithms, and applications*. Pearson Education, 2017.

[25] Ali Anwar. *Towards Efficient and Flexible Object Storage Using Resource and Functional Partitioning*. PhD thesis, Virginia Tech, 2018.

[26] Ali Anwar, Salman A Baset, Andrzej P Kochut, Hui Lei, Anca Sailer, and Alla Segal. Scalable metering for cloud service management based on cost-awareness, March 31 2016. US Patent App. 14/871,443.

[27] Ali Anwar, Yue Cheng, and Ali R Butt. Towards managing variability in the cloud. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1081–1084. IEEE, 2016.

[28] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R Butt. Taming the cloud object storage with mos. In *Proc. PDSW-DISCS*. ACM, 2015.

[29] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R Butt. Mos: Workload-aware elasticity for cloud object stores. In *Proc. HPDC*. ACM, 2017.

[30] Ali Anwar, Yue Cheng, Hai Huang, and Ali Raza Butt. Clusteron: Building highly configurable and reusable clustered data services using simple data nodes. In *Proc. HotStorage*. USENIX, 2016.

[31] Ali Anwar, Yue Cheng, Hai Huang, Jingoo Han, Hyogi Sim, Dongyoon Lee, Fred Douglis, and Ali R Butt. bespo kv: application tailored scale-out key-value stores. In *Proc. SC*. IEEE, 2018.

[32] Ali Anwar, Yue Cheng, Hai Huang, Jingoo Han, Hyogi Sim, Dongyoon Lee, Fred Douglis, and Ali R Butt. Customizable scale-out key-value stores. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2081–2096, 2020.

[33] Ali Anwar, Andrzej Kochut, Anca Sailer, Charles O Schulz, and Alla Segal. Dynamic metering adjustment for service management of computing platform, March 31 2016. US Patent App. 14/926,384.

[34] Ali Anwar, KR Krish, and Ali R Butt. On the use of microservers in supporting hadoop applications. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 66–74. IEEE, 2014.

[35] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S Warke, Heiko Ludwig, and Ali. R. Butt. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies*, page 265, 2018.

[36] Ali Anwar, Lukas Rupprecht, Dimitris Skourtis, and Vasily Tarasov. Challenges in storing docker images. *login Usenix Mag.*, 44(3), 2019.

[37] Ali Anwar, Anca Sailer, Andrzej Kochut, and Ali R Butt. Anatomy of cloud monitoring and metering: A case study and open problems. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 6. ACM, 2015.

[38] Ali Anwar, Anca Sailer, Andrzej Kochut, Charles O Schulz, Alla Segal, and Ali R Butt. Cost-aware cloud metering with scalable service management infrastructure. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 285–292. IEEE, 2015.

[39] Ali Anwar, Anca Sailer, Andrzej Kochut, Charles O Schulz, Alla Segal, and Ali R Butt. Scalable metering for an affordable it cloud service management. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 207–212. IEEE, 2015.

[40] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No. 99TH8469)*, pages 115–124. IEEE, 1999.

[41] Ayse Bagbaba. Improving collective i/o performance with machine learning supported auto-tuning. In *The Fifteenth International Workshop on Automatic Performance Tuning*, 2020.

[42] Babak Behzad, Surendra Byna, and Marc Snir. Optimizing i/o performance of hpc applications with autotuning. *ACM Transactions on Parallel Computing (TOPC)*, 5(4):1–27, 2019.

[43] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.

[44] Dan Bonachea, Paul H. Hargrove, Michael Welcome, and Katherine Yelick. Porting gasnet to portals: Partitioned global address space (pgas) language support for the cray xt. 5 2009.

[45] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[46] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

[47] Jeff Bonwick and Jonathan Adams. Magazines and vmem: Extending the slab allocator to many cpus and arbitrary resources. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 15–33, Berkeley, CA, USA, 2001. USENIX Association.

[48] 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*, 15(5), 2008.

[49] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[50] Eric B Boyer, Matthew C Broomfield, and Terrell A Perrotti. Glusterfs one storage server to rule them all. Technical report, Los Alamos National Laboratory (LANL), 2012.

[51] Peter J Brockwell, Richard A Davis, and Matthew V Calder. *Introduction to time series and forecasting*, volume 2. Springer, 2002.

[52] Surendra Byna, Jerry Chou, Oliver Rübel, Prabhat, Homa Karimabadi, William S. Daughton, Vadim Roytershteyn, E. Wes Bethel, Mark Howison, Ke-Jou Hsu, Kuan-Wu Lin, Arie Shoshani, Andrew Uselton, and Kesheng Wu. Parallel i/o, analysis,

and visualization of a trillion particle simulation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 59:1–59:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[53] Kirk W Cameron, Ali Anwar, Yue Cheng, Li Xu, Bo Li, Uday Ananth, Thomas Lux, Yili Hong, Layne T Watson, and Ali R Butt. Moana: Modeling and analyzing i/o variability in parallel system experimental design. 2018.

[54] Nicholas J Carriero, David Gelernter, Timothy G Mattson, and Andrew H Sherman. The linda alternative to message-passing systems. *Parallel computing*, 20(4):633–655, 1994.

[55] Zheng Chai, Ahsan Ali, Syed Zawad, Stacey Truex, Ali Anwar, Nathalie Baracaldo, Yi Zhou, Heiko Ludwig, Feng Yan, and Yue Cheng. Tifl: A tier-based federated learning system. *To appear in ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2020.

[56] Zheng Chai, Hannan Fayyaz, Zeshan Fayyaz, Ali Anwar, Yi Zhou, Nathalie Baracaldo, Heiko Ludwig, and Yue Cheng. Towards taming the resource and data heterogeneity in federated learning. In *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*, pages 19–21, 2019.

[57] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[58] Y. Chen, X. H. Sun, R. Thakur, P. C. Roth, and W. D. Gropp. Lacio: A new collective i/o strategy for parallel i/o systems. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 794–804, May 2011.

[59] Yue Chen, Wei Chen, Melanie H Cobb, and Yingming Zhao. Ptmap—a sequence alignment software for unrestricted, accurate, and full-spectrum identification of post-translational modification sites. *Proceedings of the National Academy of Sciences*, 106(3):761–766, 2009.

[60] Yue Cheng, Zheng Chai, and Ali Anwar. Characterizing co-located datacenter workloads: An alibaba case study. *arXiv preprint arXiv:1808.02919*, 2018.

[61] Lauro Beltrão Costa, Hao Yang, Emalayan Vairavanathan, Abmar Barros, Ketan Maheshwari, Gilles Fedak, D Katz, Michael Wilde, Matei Ripeanu, and Samer Al-Kiswany. The case for workflow-aware storage: An opportunity study. *Journal of Grid Computing*, 13(1):95–113, 2015.

[62] Breno Dantas Cruz, Arnab K Paul, and Eli Tilevich. Stargazer: A deep learning approach for estimating the performance of edge-based clustering applications. In

2020 IEEE International Conference on Smart Data Services (SMDS), pages 1–10. IEEE, 2020.

[63] Arjun Datta and Arnab Kumar Paul. Online compiler as a cloud service. In *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*, pages 1783–1786. IEEE, 2014.

[64] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[65] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[66] Shyam C Deshmukh and Sudarshan S Deshmukh. Improved load balancing for distributed file system using self acting and adaptive loading data migration process. In *4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions), 2015*, pages 1–6. IEEE, 2015.

[67] Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, Nawab Ali, and P. Sadayappan. Integrating parallel file systems with object-based storage devices. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 27:1–27:10, New York, NY, USA, 2007. ACM.

[68] Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, Nawab Ali, and P Sadayappan. Integrating parallel file systems with object-based storage devices. In *SC'07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10. IEEE, 2007.

[69] Tim d'Hondt, Anna Wilbik, Paul Grefen, Heiko Ludwig, Natalie Baracaldo, and Ali Anwar. Using bpm technology to deploy and manage distributed analytics in collaborative iot-driven business scenarios. In *Proceedings of the 9th International Conference on the Internet of Things*, pages 1–8, 2019.

[70] Bin Dong, Xiuqiao Li, Qimeng Wu, Limin Xiao, and Li Ruan. A dynamic and adaptive load balancing strategy for parallel file system with large-scale i/o servers. *JPDC*, 72(10):1254–1268, 2012.

[71] Stefan Eilemann, Fabien Delalondre, Jon Bernard, Judit Planas, Felix Schuermann, John Biddiscombe, Costas Bekas, Alessandro Curioni, Bernard Metzler, Peter Kaltstein, Peter Morjan, Joachim Fenkes, Ralph Bellofatto, Lars Schneidenbach, Thomas Ward, and Blake Fitch. Key/value-enabled flash memory for complex scientific workflows with on-line analysis and visualization. 05 2016.

[72] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: the future building block for storage systems. In *Local to Global Data Interoperability - Challenges and Technologies, 2005*, pages 119–123, June 2005.

[73] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[74] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proc. EDBT/ICDT 2011*, AD '11, New York, NY, USA. ACM.

[75] K. Gao, W. k. Liao, A. Nisar, A. Choudhary, R. Ross, and R. Latham. Using subfiling to improve programming flexibility and performance of parallel shared-file i/o. In *2009 International Conference on Parallel Processing*, pages 470–477, Sept 2009.

[76] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Howard Gobioff, Erik Riedel, David Rochberg, and Jim Zelenka. Filesystems for Network-Attached Secure Disks. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1997.

[77] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGPLAN Not.*, 33(11):92–103, October 1998.

[78] Gluster. Gluster-cloud storage for the modern data center, 2017. http://moo.nac.uci.edu/ hjm/fs/AnIntroductionToGlusterArchitectureV7110708.pdf.

[79] Gary Grider. Marfs: A scalable near-posix file system over cloud objects background, use, and technical overview, 2016.

[80] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann. Hacc: Extreme scaling and performance across diverse architectures. In *Proc. SC*, Nov 2013.

[81] Ibrahim F. Haddad. Pvfs: A parallel virtual file system for linux clusters. *Linux J.*, 2000(80es), November 2000.

[82] Andrew C Harvey. *Forecasting, structural time series models and the Kalman filter*. Cambridge university press, 1990.

[83] S. He, Y. Wang, X. H. Sun, and C. Xu. Harl: Optimizing parallel file systems with heterogeneity-aware region-level data layout. *IEEE Transactions on Computers*, 66(6):1048–1060, June 2017.

[84] Shuibing He, Xian-He Sun, and Adnan Haider. Has: Heterogeneity-aware selective data layout scheme for parallel file systems on hybrid servers. In *Proc. IPDPS*. IEEE, 2015.

[85] Pieter Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly, 2013.

[86] Howard Gobioff and Garth Gibson and Doug Tygar. Security for Network Attached Storage Devices. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1997.

[87] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[88] Intel. DAOS Lustre Restructuring and Protocol Changes Design : FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O, 2014. https://goo.gl/oCKLso.

[89] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, et al. Fireworks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 27(17):5037–5059, 2015.

[90] P. J.Braam. The lustre storage architecture (tech. rep.). Technical report, Available: http://wiki.lustre.org/., 2004.

[91] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[92] Y. Kang, Jingpei Yang, and E.L. Miller. Object-based scm: An efficient interface for storage class memories. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–12, May 2011.

[93] Yangwook Kang. *High-Performance, Reliable Object-Based NVRAM Devices*. PhD thesis, Storage Systems Research Center, University of California, Santa Cruz, 9 2014.

[94] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[95] Donghun Koo, Jik-Soo Kim, Soonwook Hwang, Hyeonsang Eom, and Jaehwan Lee. Utilizing progressive file layout leveraging ssds in hpc cloud environments. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 90–95. IEEE, 2016.

[96] Quincey Koziol. Design and implementation of fastforward features in hdf5 for extreme-scale computing research and development (fast forward) storage and i/o. Technical report, The HDF Group, 2014.

[97] K. R. Krish, B. Wadhwa, M. S. Iqbal, M. M. Rafique, and A. R. Butt. On efficient hierarchical storage for big data processing. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 403–408, 2016.

[98] KR Krish, Ali Anwar, and Ali R Butt. hats: A heterogeneity-aware tiered storage for hadoop. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2014.

[99] KR Krish, Ali Anwar, and Ali R Butt. [phi] sched: A heterogeneity-aware hadoop workflow scheduler. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 255–264. IEEE, 2014.

[100] Anastasia Clower Laity, Nate Anagnostou, G Bruce Berriman, John C Good, Joseph C Jacob, Daniel S Katz, and Thomas A Prince. Montage: an astronomical image mosaic service for the nvo. 2005.

[101] Young-Sik Lee, Sang-Hoon Kim, Jin-Soo Kim, Jaesoo Lee, Chanik Park, and Seungryoul Maeng. Ossd: A case for object-based solid state drives. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–13, May 2013.

[102] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[103] Jianwei Li, Wei 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*, pages 39–39, Nov 2003.

[104] Xiuqiao Li, Limin Xiao, Meikang Qiu, Bin Dong, and Li Ruan. Enabling dynamic file i/o path selection at runtime for parallel file system. *The Journal of Supercomputing*, 68(2):996–1021, 2014.

[105] Michael Littley, Ali Anwar, Hannan Fayyaz, Zeshan Fayyaz, Vasily Tarasov, Lukas Rupprecht, Dimitrios Skourtis, Mohamed Mohamed, Heiko Ludwig, Yue Cheng, and Ali R Butt. Bolt: Towards a scalable docker registry via hyperconvergence. In *IEEE International Conference on Cloud Computing*, 2019.

[106] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, Manish Parashar, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Matthew Wolf, Kesheng Wu, and Weikuan Yu. Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[107] LLNL. Ior benchmark. Accessed: March 12 2018.

[108] Huong Luu, Babak Behzad, Ruth Aydt, and Marianne Winslett. A Multi-Level Approach for Understanding I/O Activity in HPC Applications. In *Proc. CLUSTER*. IEEE, September 2013.

[109] J. F. Martinez and E. Ipek. Dynamic multicore resource management: A machine learning approach. *IEEE Micro*, 29(5):8–17, 2009.

[110] Rich Miller. Google using machine learning to boost data center efficiency — data center knowledge, 2014.

[111] Rick Mohr, Michael Brim, Sarp Oral, and Andreas Dilger. Evaluating progressive file layouts for lustre. In *Cray User Group Conference (CUG 2016)*, 2016.

[112] Esteban Molina-Estolano, Carlos Maltzahn, and Scott Brandt. Dynamic load balancing in ceph. 2008.

[113] David Nagle, Michael Factor, Sami Iren, Dalit Naor, Erik Riedel, Ohad Rodeh, and Julian Satran. The ANSI T10 object-based storage standard and current implementations. *IBM Journal of Research and Development*, 52(4-5):401–412, 2008.

[114] Lustre Networking. High-performance features and flexible support for a wide array of networks, 2008.

[115] Sarah Neuwirth. *Accelerating Network Communication and I/O in Scientific High Performance Computing Environments*. PhD thesis, Heidelberg University, Germany, December 2018.

[116] Sarah Neuwirth, Feiyi Wang, Sarp Oral, and Ulrich Bruening. Automatic and transparent resource contention mitigation for improving large-scale parallel file system performance. In *Proc. ICPADS*. IEEE, 2017.

[117] Cosimo Palazzo, Andrea Mariello, Sandro Fiore, Alessandro D'Anca, Donatello Elia, Dean N Williams, and Giovanni Aloisio. A workflow-enabled big data analytics software stack for escience. In *2015 International Conference on High Performance Computing & Simulation (HPCS)*, pages 545–552. IEEE, 2015.

[118] Arnab K Paul, Olaf Faaland, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Ali R Butt. Improving i/o performance of hpc applications using intra-job scheduling. In *2019 Work-In-Progress Proceedings of the Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, pages 1 – 1, 2019.

[119] Arnab K Paul, Olaf Faaland, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Ali R Butt. Understanding hpc application i/o behavior using system level statistics, 2019.

[120] Arnab K Paul, Arpit Goyal, Feiyi Wang, Sarp Oral, Ali R Butt, Michael J Brim, and Sangeetha B Srinivasa. I/o load balancing for big data hpc applications. In *Proc. Big Data 2017*. IEEE, 2017.

[121] Arnab K Paul, Steven Tuecke, Ryan Chard, Ali R Butt, Kyle Chard, and Ian Foster. Toward scalable monitoring on large-scale storage for software defined cyberinfrastructure. In *Proc. PDSW-DISCS*. ACM, 2017.

[122] Arnab K Paul, Brian Wang, Nathan Rutman, Cory Spitz, and Ali R Butt. Efficient metadata indexing for hpc storage systems. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 162–171. IEEE, 2020.

[123] Arnab Kumar Paul. *Dynamic virtual machine placement in cloud computing*. PhD thesis, 2015.

[124] Arnab Kumar Paul, Sourav Kanti Addya, Bibhudatta Sahoo, and Ashok Kumar Turuk. Application of greedy algorithms to virtual machine distribution across data centers. In *Annual IEEE India Conference (INDICON)*, 2014.

[125] Arnab Kumar Paul and Bibhudatta Sahoo. Dynamic virtual machine placement in cloud computing. In *Resource Management and Efficiency in Cloud Computing Environments*. IGI Global, 2017.

[126] Arnab Kumar Paul and Bibhudatta Sahoo. Dynamic virtual machine placement in cloud computing. In *Resource Management and Efficiency in Cloud Computing Environments*, pages 136–167. IGI Global, 2017.

[127] Arnab Kumar Paul, Wenjie Zhuang, Luna Xu, Min Li, M Mustafa Rafique, and Ali R Butt. Chopper: Optimizing data partitioning for in-memory data analytics frameworks. In *Proc. CLUSTER*. IEEE, 2016.

[128] Arnab Kumar Paul, Wenjie Zhuang, Luna Xu, Min Li, M Mustafa Rafique, and Ali R Butt. Chopper: Optimizing data partitioning for in-memory data analytics frameworks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 110–119. IEEE, 2016.

[129] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. Nfs version 3: Design and implementation. In *USENIX Summer Technical Conference*, pages 137–152. Boston, MA, 1994.

[130] Gregory F Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[131] Yingjin Qian, Eric Barton, Tom Wang, Nirant Puntambekar, and Andreas Dilger. A novel network request scheduler for a large scale storage system. *Computer Science-Research and Development*, 23(3-4):143–148, 2009.

[132] Abhishek Rajimwale, Vijayan Prabhakaran, and John D. Davis. Block management in solid-state devices. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 21–21, Berkeley, CA, USA, 2009. USENIX Association.

[133] Ramesh R Sarukkai. Link prediction and path analysis using markov chains1. *Computer Networks*, 33(1-6):377–386, 2000.

[134] Andrea Schaerf, Yoav Shoham, and Moshe Tennenholtz. Adaptive load balancing: A study in multi-agent learning. *Journal of Artificial Intelligence Research*, 2:475–500, 1995.

[135] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proc. 1st USENIX FAST*, 2002.

[136] Lars Schneidenbach, Bruce D'Amora, Claudia Misale, Carlos HA Costa, Sara Kokkila Schumacher, and Thomas Ward. Data broker: a case for workflow enablement using a key/value approach. In *Proceedings of the International Symposium on Memory Systems*, pages 250–260, 2019.

[137] Hyogi Sim, Youngjae Kim, Sudharshan S Vazhkudai, Devesh Tiwari, Ali Anwar, Ali R Butt, and Lavanya Ramakrishnan. Analyzethis: an analysis workflow-aware storage system. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[138] Hyogi Sim, Arnab K Paul, Eli Tilevich, Ali R Butt, and Muhammad Shahzad. Cslim: automated extraction of iot functionalities from legacy c codebases. In *Proceedings of the 20th International Conference on Distributed Computing and Networking*, pages 421–426, 2019.

[139] Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, and Ameet S Talwalkar. Federated multi-task learning. In *Advances in Neural Information Processing Systems*, pages 4424–4434, 2017.

[140] Huaiming Song, Yanlong Yin, Xian-He Sun, Rajeev Thakur, and Samuel Lang. A segment-level adaptive data layout scheme for improved load balance in parallel file systems. In *Proc. CCGRID 2011*, pages 414–423. IEEE, 2011.

[141] Toyotaro Suzumura, Yi Zhou, Natahalie Barcardo, Guangnan Ye, Keith Houck, Ryo Kawahara, Ali Anwar, Lucia Larise Stavarache, Daniel Klyashtorny, Heiko Ludwig, et al. Towards federated graph learning for collaborative financial crimes detection. *arXiv preprint arXiv:1909.12946*, 2019.

[142] 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 *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 113–122, 2018.

[143] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *Proc. IOPADS*, NY, USA, 1999. ACM.

[144] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. A hybrid approach to privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pages 1–11, 2019.

[145] Yuichi Tsujita, Tatsuhiko Yoshizaki, Keiji Yamamoto, Fumichika Sueyasu, Ryoji Miyazaki, and Atsuya Uno. Alleviating i/o interference through workload-aware striping and load-balancing on parallel file systems. In *ISC*. Springer, 2017.

[146] Alan Tucker. A note on convergence of the ford-fulkerson flow algorithm. *Mathematics of Operations Research*, 2(2):143–144, 1977.

[147] Emalayan Vairavanathan, Samer Al-Kiswany, Lauro Beltrão Costa, Zhao Zhang, Daniel S Katz, Michael Wilde, and Matei Ripeanu. A workflow-aware storage system: An opportunity study. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 326–334. IEEE, 2012.

[148] B. Wadhwa and A. Verma. Carbon efficient vm placement and migration technique for green federated cloud datacenters. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 2297–2302, 2014.

[149] B. Wadhwa and A. Verma. Energy and carbon efficient vm placement and migration technique for green cloud datacenters. In *2014 Seventh International Conference on Contemporary Computing (IC3)*, pages 189–193, 2014.

[150] B. Wadhwa and A. Verma. Energy saving approaches for green cloud computing: A review. In *2014 Recent Advances in Engineering and Computational Sciences (RAECS)*, pages 1–6, 2014.

[151] Bharti Wadhwa, Suren Byna, and Ali R Butt. Toward transparent data management in multi-layer storage hierarchy of hpc systems. In *Proc. IC2E*. IEEE, 2018.

[152] Bharti Wadhwa, Arnab K Paul, Sarah Neuwirth, Feiyi Wang, Sarp Oral, Ali R Butt, Jon Bernard, and Kirk W Cameron. iez: Resource contention aware load balancing for large-scale parallel file systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 610–620. IEEE, 2019.

[153] Stephen R. Walli. The POSIX Family of Standards. *StandardView*, 3(1):11–17, March 1995.

[154] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, 2020.

[155] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan S Vazhkudai. Improving large-scale storage system performance via topology-aware and balanced data placement. In *Proc. ICPADS 2014*, pages 656–663. IEEE, 2014.

[156] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An ephemeral burst-buffer file system for scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 69:1–69:12, Piscataway, NJ, USA, 2016. IEEE Press.

[157] Y. Wang, P. Lu, and K. B. Kent. Wafs: A workflow-aware file system for effective storage utilization in the cloud. *IEEE Transactions on Computers*, 64(9):2716–2729, 2015.

[158] Yang Wang, Paul Lu, and Kenneth B Kent. Wafs: a workflow-aware file system for effective storage utilization in the cloud. *IEEE Transactions on Computers*, 64(9):2716–2729, 2014.

[159] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[160] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. USENIX OSDI 2006*, pages 307–320. USENIX Association, 2006.

[161] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of ACM/IEEE SC*, 2006.

[162] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 31–31. IEEE, 2006.

[163] Sage A Weil, Andrew W Leung, Scott A Brandt, and Carlos 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*, pages 35–44, 2007.

[164] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.

[165] George Wells. Coordination languages: Back to the future with linda. In *Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT05)*, pages 87–98, 2005.

[166] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[167] Justin M Wozniak, Timothy G Armstrong, Michael Wilde, Daniel S Katz, Ewing Lusk, and Ian T Foster. Swift/t: Scalable data flow programming for many-task applications. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 309–310, 2013.

[168] Runhua Xu, Nathalie Baracaldo, Yi Zhou, Ali Anwar, and Heiko Ludwig. Hybridalpha: An efficient approach for privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pages 13–23, 2019.

[169] Y. Yin, J. Li, J. He, X. H. Sun, and R. Thakur. Pattern-direct and layout-aware replication scheme for parallel i/o systems. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 345–356, May 2013.

[170] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[171] Zeng Zeng and Bharadwaj Veeravalli. On the design of distributed object placement and load balancing strategies in large-scale networked multimedia storage systems. *IEEE TKDE*, 20(3):369–382, 2008.

[172] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R Butt. Duphunter: Flexible high-performance deduplication for docker registries. *To appear in USENIX Annual Technical Conference (ATC 20)*, 2020.

[173] Nannan Zhao, Ali Anware, Yue Cheng, Mohammed Salman, Daping Li, Jiguang Wan, Changsheng Xie, Xubin He, Feiyi Wang, and Ali Butt. Chameleon: An adaptive wear balancer for flash clusters. In *Proc. IPDPS*. IEEE, 2018.

[174] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10. IEEE, 2019.

[175] Nannan Zhao, Vasily Tarasov, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit Warke, Mohamed Mohamed, and Ali Butt. Slimmer: Weight loss secrets for docker registries. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 517–519. IEEE, 2019.

[176] Mingfa Zhu, Guoying Li, Li Ruan, Ke Xie, and Limin Xiao. Hysf: A striped file assignment strategy for parallel file system with hybrid storage. In *Proc. HPCC_EUC*. IEEE, 2013.