2017 IEEE 23rd International Conference on Parallel and Distributed Systems

Automatic and Transparent Resource Contention Mitigation for Improving Large-scale Parallel File System Performance

Sarah Neuwirth*, Feiyi Wang†, Sarp Oral† and Ulrich Bruening*
*Institute of Computer Engineering, University of Heidelberg, Germany
{sarah.neuwirth, ulrich.bruening}@ziti.uni-heidelberg.de
†National Center for Computational Sciences, Oak Ridge National Laboratory, USA
{fwang2, oralhs}@ornl.gov

***Abstract*—Proportional to the scale increases in HPC systems, many scientific applications are becoming increasingly data intensive, and parallel I/O has become one of the dominant factors impacting the large-scale HPC application performance. On a typical large-scale HPC system, we have observed that the lack of a global workload coordination coupled with the shared nature of storage systems cause load imbalance and resource contention over the end-to-end I/O paths resulting in severe performance degradation. I/O load imbalance on HPC systems is generally a self-inflicted wound and mostly occurs between the I/O paths and resources consumed by each individual job.**

In this paper, we introduce TAPP-IO, a dynamic, shared load balancing framework for mitigating resource contention. TAPP-IO extends our previous work and solves two major limitations: First, it transparently intercepts file creation calls during runtime to balance the workload over all available storage targets. The usage of TAPP-IO requires no application source code modifications and is independent from any I/O middleware. The framework can be applied to almost any HPC platform and is suitable for systems that lack a centralized file system resource manager. Second, the framework proposes a new placement strategy to support not only file-per-process I/O, but also single shared file I/O. This opens the door to a new class of scientific applications that can leverage the placement library for improved performance. We demonstrate the effectiveness of our integration on the Titan system at the Oak Ridge National Laboratory. Our experiments with a synthetic benchmark and real-world HPC workload show that, even in a noisy production environment, TAPP-IO can improve large-scale application performance significantly.

I. Introduction

Top-tier scientific HPC systems are constantly increasing in scale to improve application resolution and reduce the time to solution. Such a scale increase also intensifies the system complexity and decreases the overall system reliability (i.e., mean time to failure). To cope with system failures, most scientific applications periodically write out memory states. This defensive, bursty I/O (i.e., checkpointing) is the main source of I/O activity on large-scale HPC systems. The rule of thumb for checkpointing is that it should not take more than 10% of the application run time in an hour. As the HPC systems grow in scale, the cumulative memory size also grows, resulting in larger amounts of data to be written out during each checkpoint window. This pushes the limits and capabilities of parallel file and storage systems serving HPC systems, since the growth trajectories for these two are not always the same. Most scientific HPC applications make no difference between defensive I/O and scientific output; they do I/O operations at regular pre-defined intervals and expect a minimal amount of time is spent in I/O subroutines.

Parallel I/O systems are inherently complex, particularly in the context of end-to-end I/O paths. For example at the starting point of a typical I/O path, an application can use a high-level library, such as HDF5 [1], for various reasons including portability, improved data management, and enhanced metadata capabilities. HDF5 is implemented on top of MPI-IO [2] which, in turn, performs POSIX I/O calls against a parallel file system such as Lustre [3]. Furthermore, before an I/O request reaches its eventual storage target, it may have to traverse through the compute fabric (e.g., a 3D torus or Dragonfly, or a plain fat-tree), and a large-scale storage network fabric (e.g., InfiniBand). Finally, the request reaches the backend storage that provides a block interface, though the parallel file system (PFS) sits on top and across these backend storage devices. In large-scale HPC deployments, I/O subsystems are typically shared among multiple applications running concurrently with different usage patterns. Due to the shared nature and inherently complex design, the observed I/O performance at the application level can be much lower than the theoretical peak bandwidth of the underlying storage system. At the Oak Ridge Leadership Computing Facility (OLCF), we observe firsthand that such a complex I/O subsystem severely suffers from contention and significant load imbalances among the different storage system components. This effect is even more pronounced when the system is under stress (i.e., bursty I/O activity periods). Also, it is observed that at large scales the contention is mostly due to self-interference (for a given scientific application) rather than by contention from competing workloads [4].

Efficient use of extreme-scale computing resources often requires extensive application tuning and code modification. There is a steep learning curve for scientific application developers to understand the complex I/O subsystem and to address the I/O load imbalance and contention issues. Therefore, it is a major hurdle for applications to adopt and take advantage of any underlying improvement. To ease this transition in the most transparent way, we propose TAPP-IO (*Transparent Automatic Placement of Parallel I/O*), a dynamic, shared load balancing framework that balances the I/O workload evenly among all storage system components. Similarly to the *Balanced Placement I/O* (BPIO) library [5], the framework addresses the resource contention problem by providing a topology-aware, balanced placement strategy that is based on a tunable, weighted cost function of available system components. This paper makes the following contributions:

- We design and implement TAPP-IO, an I/O load balancing framework, that transparently intercepts file I/O calls (metadata operations) during runtime to distribute the workload evenly over all available storage targets.
- TAPP-IO supports three major, widely used I/O interfaces for broad application compatibility: HDF5, MPI-IO, and POSIX I/O. The framework requires no application source code modifications and supports both statically and dynamically linked applications.
- We propose a new placement strategy supporting both file-per-process and single shared file I/O. The single shared file I/O model support enables a new class of scientific applications to leverage the placement library (transparently) for improved I/O performance.
- We demonstrate the effectiveness of TAPP-IO by repeating small-, medium- and large-scale runs over an extended period of time and compare the results to our previous work (i.e, BPIO and Aequilibro [6], [7]). We evaluate the performance for POSIX I/O, MPI-IO, and HDF5, by utilizing the Interleaved Or Random (IOR) synthetic benchmark and a real-world HPC workload. E.g., for the file-per-process I/O model with 4,096 nodes and a file size of 4 GB per writing process, TAPP-IO provides performance improvements of about 54% for POSIX I/O, 51% for MPI-IO, and 32% for HDF5 (see also Figure 6).

The remainder of this paper is organized as follows. Section 2 provides the background and related work. Section 3 describes the balancing framework. In Section 4, we discuss the analysis methodology and evaluate the effectiveness of our framework. Finally, we summarize our current efforts in Section 5, discuss open issues, and outline future work.

II. Background and Related Work

The experiments in this paper were conducted on the Titan system at the Oak Ridge National Laboratory. Titan is a Cray XK7 system with 18,688 compute nodes. Its parallel file system is Spider II [8], which is based on the Lustre technology [3] and one of the world's fastest and largest POSIX-compliant parallel file systems. It is configured and deployed as two independent and non-overlapping file systems, each with 144 Lustre Object Storage Servers (OSSs) and 1,008 Lustre Object Storage Targets (OSTs). The OSSes currently run the Lustre parallel file system version 2.8.0.

Resource contention has a negative impact on the performance and scalability in high-performance storage systems. As emphasized by Wang et al. [5], a significant variation in usage across system resources has been identified on a large-scale file and storage system. This variation and the resulting contention is due to the lack of a system-wide I/O load organizer on modern distributed large-scale HPC systems. Parallel file and storage systems supporting these large-scale systems only have a partial view of the overall I/O activity and can only try to optimize the resource usage at one end of the I/O path. Also, Xie et al. [4] observed that most I/O load imbalances are the result of improper resource allocation within a given job (i.e., scientific application), which makes this work particularly relevant.

The end-to-end resource contention problem can be addressed by different approaches. One possibility would be the improvement of the Lustre OST allocation scheme. But, this only addresses one end of the problem (ignoring the rest of the resources on the end-to-end I/O paths) and therefore, is disregarded. Another way to achieve balanced resource utilization is the deployment of a centralized, system-wide I/O coordination and control mechanism. For example, Fastpass [9] is a network framework that aims for high utilization with zero queuing for data centers. For large-scale scientific HPC systems, this approach is not feasible. Multiple applications are running concurrently, with a variety of different I/O patterns and workloads. A system-wide I/O organizer requires to coordinate between different subsystems (often designed and provided by different vendors), as well as scientific application code changes to participate in system-wide coordinated I/O. Therefore, even though it is possible to design a system-wide I/O coordinator, it is practically prohibited to be deployed on a large-scale HPC system. The system-wide, balanced I/O request coordination would lead to a tremendous communication overhead, and therefore, it likely would lead to a sub-optimal utilization of the available computational resources.

The third approach is to balance the I/O workload on an end-to-end and per job basis. This technique is adopted by the *Balanced Placement I/O* (BPIO) library [5]. BPIO intelligently allocates I/O paths for a parallel file system. The library employs a placement strategy that provides a binding between an I/O client (compute node or MPI rank ID) and a storage target while aiming at an evenly I/O traffic distribution across resource components to avoid points of contention. The library utilizes the *Fine-Grained Routing* (FGR) congestion avoidance method [10]. FGR organizes I/O paths to minimize end-to-end hop counts and congestion. This is done by pairing clients with their closest possible, and in the case of Titan, optimal LNET router. The BPIO placement algorithm uses a placement cost function that takes a weighted average of how frequently different file system resources have been used by previous I/O requests issued by the same application. The most general case is defined by

$$C = w_1 R_1 + w_2 R_2 + ... + w_n R_n \tag{1}$$

where C is the cost of an I/O path, R_i is a resource component, and w_i is the weight factor with $\sum_{i=1}^{n} w_i = 1$. For Lustre, possible resource components are logical I/O routers (i.e., LNET), or actual file system and networking resources (i.e., Lustre I/O routers, OSSs, and OSTs). The algorithm loops over all reachable storage targets to choose one with the lowest placement cost per compute node. This is repeated for all I/O clients allotted to the application once before it enters the I/O write phase. An initial performance evaluation of the library was performed on the Titan supercomputer [11] at OLCF.

Aequilibro [6], an ADIOS-based middleware, attempts to resolve the load imbalance by utilizing the BPIO library. ADIOS [12] is a flexible middleware that provides a simple I/O application programming interface (API) with portable, fast, scalable, metadata-rich output. An initial performance evaluation has been carried out on the Titan system. Drawbacks are that the usage is limited to specific ADIOS transport methods, namely `POSIX` and `MPI_AGGREGATE`, and the overhead added by ADIOS limits the performance improvement compared to a direct BPIO integration into an application.

As we move towards the exascale era, resource contention and performance variability in HPC systems remain a major challenge. I/O workload imbalance paired with the increasing gap between compute capabilities of HPC systems and the underlying storage system are known issues in the HPC domain [13], [14]. Several research studies have addressed these problems to provide better I/O techniques. For example, Gainaru et al. [15] introduce a global scheduler that minimizes congestion caused by I/O interference by considering the application's past behaviors when scheduling I/O requests. Herbein et al. [16] present a job scheduling technique that reduces contention by integrating I/O-awareness into scheduling policies. As shown by Yildiz et al. [14], scheduler-level solutions not always lead to improved performance even though it helps to control the level of interferences.

Some research efforts consider network contention as the major contributor to I/O load imbalance. Luo et al. [17] introduce a preemptive, core stateless optimization approach based on open loop end-point throttling. Jiang et al. [18] introduce new endpoint congestion-control protocols to address the differentiation between network and endpoint congestion more properly. Li et al. [19] present *ASCAR*, a storage traffic management system for improving the bandwidth utilization and fairness of resource allocation. This is not feasible for large-scale HPC systems like Titan. There are too many applications and the I/O patterns change drastically depending upon the run. Another area takes a file and storage-system centric view. Zhu et al. [20] present CEFT-PVFS, a modification to the PVFS file system [21] to achieve a better I/O load balancing. Luu et al. [22] analyze the problem of low I/O performance on leading HPC systems. They use Darshan [23] logs of over a million jobs representing a combined total of six years of I/O. Liu et al. [24] present a set of dynamic and proactive ADIOS transport methods that are able to shift workloads dynamically to lightly used areas of the storage system while applying a throttling technique that limits how much data can be re-routed during writing.

This paper complements previous work by bridging the gap between the interconnect level, and file and storage-system centric view. It provides users with a transparent auto-tuning framework that takes full advantage of the optimizations done at the interconnect level and the load balancing done at the file system level. We argue that by designing a pre-loadable BPIO-inspired framework, we translate the benefits of BPIO transparently into an application without any source code modification.

Resource allocation has also been researched in the context of commercial data centers, such as *Pulsar* [25] and *Baraat* [26]. Many of the assumptions made for commercial data center performance optimizations are not applicable to large-scale scientific simulation systems. For example, scientific HPC systems do not have a global view on all available system resources and allocations. Commercial and scientific applications have different requirements [27]. While commercial codes can be classified as high-throughput computing, scientific workloads are categorized as latency-sensitive, large-scale, and tightly coupled computations. They assume the presence of a high-bandwidth, low-latency interconnect, a shared parallel file system between compute nodes, and a head node that can submit MPI jobs to all worker nodes. Therefore, TAPP-IO was developed for a scientific HPC environment and tested on a system optimized for large-scale simulations.

III. TAPP-IO Framework

TAPP-IO is designed to work with parallel file systems and does not require any modifications to application or I/O library source code. It is implemented as a user space library, based on an improved placement algorithm (as detailed in section 3.1) for intelligently allocating resources along I/O paths. The TAPP-IO framework works with both dynamically and statically linked applications and supports both file-per-process and single-shared-file I/O modes. It also supports three major I/O interfaces: POSIX I/O, MPI-IO, and HDF5 I/O. As pointed out by a recent study of petascale supercomputers [22], between 50% and 95% of HPC applications use the POSIX I/O library. The remaining jobs use MPI-IO directly or libraries built atop MPI-IO, such as HDF5. This ensures that we have a broad application compatibility support.

A. Parallel I/O Support

The TAPP-IO framework proposes a file placement strategy that supports both file-per-process and single-shared-file I/O access patterns. File-per-process scales the single writer I/O pattern to encompass all processes instead of just one process. Each process performs I/O operations on their own separate file. Thus, for an application run of N processes, N or more files are created. In a single-shared-file application I/O pattern, multiple processes perform I/O operations either independently or concurrently to the same file. The possible HPC application I/O patterns can roughly be classified as follows (the ratio can be read as *writer count* : *file count*):

1) $N : N$, stripe count $= 1$
2) $N : N$, stripe count > 1
3) $N : M$, stripe count $\geqslant 1$, $M < N$
4) $N : 1$, stripe count > 1

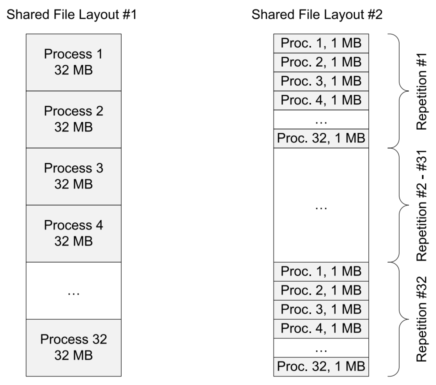

Fig. 1. Two possible shared file layouts.

Case 1) and 2) describe file-per-process I/O patterns, case 3) presents a strategy where the writing is aggregated in M shared files, and case 4) is the single-shared-file strategy where multiple clients write to multiple ranges within the same file. With a *file-per-process* I/O pattern, it is best to use no striping (stripe count of 1). This limits the storage target contention when dealing with a large number of files / processes. Therefore, case 2) is disregarded for the TAPP-IO framework. Case 3) is a special case of case 4) where multiple writers are aggregated in multiple shared files. TAPP-IO currently supports cases 1), 3), and 4) with the limitation that the balancing algorithm needs the expected file size for the shared files. For *single shared files*, TAPP-IO tries to minimize both the overhead associated with splitting an operation between storage targets and contention between writing processes over a single storage target. For example, Figure 1 displays two possible shared file layouts for 32 writing processes. Layout #1 keeps the data from a process in a contiguous block, while Layout #2 strides the data throughout the file. When accessing a single shared file from many processes, the stripe count should equal the number of processes, if possible. The size and location of I/O operations from the processes should be carefully managed to allow as much stripe alignment as possible resulting in each writing process accessing only a single storage target. Analogous to file-per-process, the algorithm follows the placement strategy implied by Layout #1, i.e., *stripe count* = *number of writers* and *stripe size* = (*file size* / *stripe count*). With these parameters, the algorithm tries to achieve high levels of performance while mitigating storage targets contention at large process counts. The placement algorithm is invoked only once for every I/O write phase which adds minimal overhead. The optimal set of storage targets is determined similarly to the Balanced Placement procedure [5]. For each stripe of a shared file, the optimal stripe to storage target assignment is calculated with the help of the BPIO placement cost function. Algorithm 1 displays a simplified version of the TAPP-IO balancing algorithm. TAPP-IO extends prior work [5], [6] by introducing a support for transparent function interpositioning, HDF5 I/O, and single shared files.

Algorithm 1 TAPP-IO Balancing Algorithm (simplified)

```
1: /* I/O call, e.g. open(), triggers balancing */
2: /* Update NID/OST binding with BPIO cost function */
3: osts ← Balanced Placement (NIDs, OSTs)
4: /* Determine placement parameters */
5: if (File-per-process) then
6:     start_ost ← osts[my_rank]
7:     stripe size ← 1 MB
8:     stripe count ← 1
9: else if (Single shared File) && (my_rank == 0) then
10:    stripe count ← #writing processes
11:    stripe size ← file size / stripe count
12:    ost_list ← osts
13: end if
14: /* Initialize Lustre file descriptor via llapi */
15: if (File-per-process) then
16:    llapi_file_create(...);
17: else if (Single shared File) && (my_rank == 0) then
18:    llapi_file_open_param(...);
19: end if
```

The implementation of the TAPP-IO balancing framework has been deployed and tested on Titan's Spider II file system. In order to specify the striping information for the file-per-process strategy, it is sufficient to set a file descriptor's Lustre striping information via the *llapi* library before opening the file via the corresponding I/O interface (`MPI_File_open()` or `H5Fcreate()`). `llapi_file_create()` allows us to specify the stripe size, stripe count, and OST offset of a file via the logical object volumn (LOV) manager. When MPI-IO or HDF5 try to create a file, the I/O layers transparently realize that the Lustre file descriptor was already created. Therefore, the existing descriptor is used to open the corresponding file. Historically, Spider II was based on Lustre 2.4 which lacked the ability to provide fine-grained control of object placement. Spider II is now running Lustre version 2.8. With Lustre 2.7 [28], a new feature was introduced that provides the user with the ability to explicitly specify the striping pattern via an ordered list of OSTs. We utilize the Lustre *llapi* to specify the Lustre striping parameter struct *llapi_stripe_param* where a list of OSTs can be passed to the LOV manager. Unlike the file-per-process strategy, `llapi_file_open_param()` is called by MPI rank 0 to create the Lustre file descriptor. The TAPP-IO library returns a list of MPI rank ID to OST assignments which is used to specify the striping pattern. Currently, the balancing algorithm for single shared files needs the expected file size from the application in order to match the stripes with the writing processes. Via `MPI_Info_set()`, the application can forward the file size by specifying a value pair `(fileSize,value)`. To utilize TAPP-IO for multiple-shared files, the framework also needs the number of writing tasks per file and the corresponding MPI communicator. TAPP-IO extracts the information from the info object and calculates the stripe size. The stripe size is matched to keep data from a process in a contiguous block. Processes can

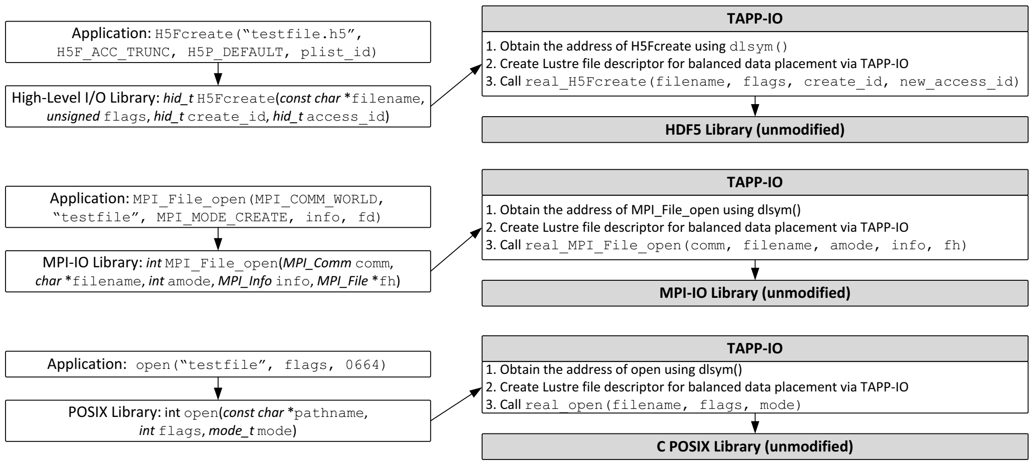
Application: H5Fcreate("testfile.h5", H5F_ACC_TRUNC, H5P_DEFAULT, plist_id)
High-Level I/O Library: hid_t H5Fcreate(const char *filename, unsigned flags, hid_t create_id, hid_t access_id)
TAPP-IO
1. Obtain the address of H5Fcreate using dlsym()
2. Create Lustre file descriptor for balanced data placement via TAPP-IO
3. Call real_H5Fcreate(filename, flags, create_id, new_access_id)
HDF5 Library (unmodified)
Application: MPI_File_open(MPI_COMM_WORLD, "testfile", MPI_MODE_CREATE, info, fd)
MPI-IO Library: int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)
TAPP-IO
1. Obtain the address of MPI_File_open using dlsym()
2. Create Lustre file descriptor for balanced data placement via TAPP-IO
3. Call real_MPI_File_open(comm, filename, amode, info, fh)
MPI-IO Library (unmodified)
Application: open("testfile", flags, 0664)
POSIX Library: int open(const char *pathname, int flags, mode_t mode)
TAPP-IO
1. Obtain the address of open using dlsym()
2. Create Lustre file descriptor for balanced data placement via TAPP-IO
3. Call real_open(filename, flags, mode)
C POSIX Library (unmodified)

Fig. 2. Dynamic interception of I/O functions at runtime.

concurrently access a single shared file. Still, this feature lacks the flexibility to dynamically re-size a file. For the time being, dynamic re-striping or re-sizing of a file comes with an enormous overhead. The basic idea would be to re-create the file with the new striping pattern. But, this involves the copying (i.e, reading into memory and then writing out of the memory) of the file to the client and back to the parallel file system. This procedure is resource consuming and therefore, not feasible. It is expected that the introduction of the progressive file layouts [29] which is based on composite layouts with Lustre 2.10 will provide the means to efficiently enhance the balancing algorithm for single shared files.

B. Function Interpositioning

TAPP-IO uses function interpositioning, similar to Recorder [30] and Darshan [23], to prioritize itself over standard functions. For dynamically linked applications, the framework is built as a shared, dynamic library. Once TAPP-IO is specified as the preloading library via `LD_PRELOAD`, it intercepts POSIX I/O, MPI-IO, and HDF5 file creation calls issued by the application and reroutes them to the balancing framework. For statically linked applications, the library requires no source code modifications, but has to be added transparently during the link phase of MPI compiler scripts such as `mpicc` or `mpif90`. This approach is a compromise in that existing binaries must be recompiled (or relinked) in order to use

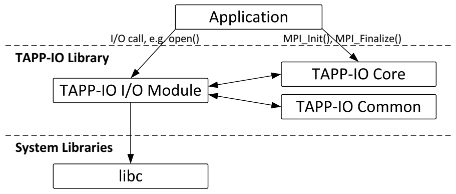

Fig. 3. TAPP-IO runtime environment.

TAPP-IO. POSIX routines are intercepted by inserting wrapper functions via the GNU linker's `--wrap` argument. After rerouting the function calls to the TAPP-IO framework, the library evenly places the data on the available storage targets. This balancing approach is transparent to the user because alterations are made without changes to application and library source code.

For both dynamically and statically linked applications, TAPP-IO intercepts MPI-IO routines using the profiling (PMPI) interface to MPI. Figure 3 illustrates the TAPP-IO runtime environment. The framework consists of three main components: *TAPP-IO Core*, *TAPP-IO Common*, and *TAPP-IO I/O Modules*. The core of the framework handles the initialization and clean up of the library. Before any I/O call can be rerouted to TAPP-IO, the internal data structures need to be initialized. This happens during `MPI_Init()`. The common module hosts the balancing algorithm and helper functions to maintain module specific I/O characterization data. In addition, there is an I/O module for every supported I/O interface. The I/O modules implement the wrapper functions. Figure 2 displays the dynamic interception of I/O routines at runtime. The following sequence illustrates the mode of operation of TAPP-IO for HDF5:

1) TAPP-IO intercepts and reroutes `H5Fcreate()` to the corresponding I/O module.
2) TAPP-IO Common provides a list of NID/OST bindings.
3) A Lustre file descriptor is allocated with the balancing information.
4) The function returns by calling `real_H5Fcreate()`.

The mechanism is the same for the MPI-IO and POSIX I/O. It offers per job and end-to-end I/O performance improvement in the most transparent way. Currently, the framework supports the following I/O calls: `open[64]()`, `creat[64]()`, `MPI_File_open()`, and `H5Fcreate[64]()`. These mechanisms have been tested with the MPICH MPI implementation for both GNU and Cray

TABLE I
IOR BENCHMARK VARIANTS FOR THE FILE-PER-PROCESS STRATEGY.

Index	Variant	Description
(I)	Default	The original IOR benchmark.
(II)	BPIO	A modified version of IOR that utilizes the BPIO library for balanced data placement.
(III)	ADIOS	An IOR benchmark where all I/O calls are replaced with the ADIOS API for I/O handling.
(IV)	Aequilibro	Same code base as IOR ADIOS, but utilizes the BPIO library for balanced data placement.
(V)	TAPP-IO	Unmodified IOR benchmark utilizing TAPP-IO via `LD_PRELOAD`.

C, C++, and Fortran compilers. It also works correctly for both static and dynamic compilation, requires no additional supporting infrastructure for instrumentation, and is compatible with other MPI implementations and compilers.

IV. Evaluation

In this section, we evaluate and analyze the effectiveness of our load balancing framework using a synthetic benchmark tool and a real-world HPC workload running on Titan.

A. Methodology

The I/O evaluation methodology is based on two benchmarks, the Interleaved Or Random (IOR) benchmark and Genarray, a benchmark that emulates I/O workload similar to S3D [31]. In addition, we describe benchmark parameters that need to be specified to model HPC workload behavior.

1) IOR Benchmark: IOR [32] provides a flexible way of measuring I/O performance with different parameter configurations, including I/O interfaces ranging from traditional POSIX to advanced parallel I/O interfaces like MPI-IO and differentiates parallel I/O strategies between file-per-process and single-shared-file. Shan et al. [33] demonstrated that IOR can be used to characterize and predict the I/O performance on HPC systems at scale. Table I displays the IOR benchmark variants used with the *file-per-process* strategy. The evaluation is divided into three different I/O performance comparisons. First, the original version of BPIO is directly used for the data placement by modifying the IOR source code, referred to as *IOR BPIO*. Before creating a file with Lustre's *llapi*, the BPIO library is used to determine the compute node (NID) to OST assignment. The results are compared to the unmodified IOR benchmark *IOR Default*. Second, all I/O interface calls are replaced by the ADIOS API. We use IOR as a workload generator to drive the ADIOS framework, denoted as *IOR ADIOS*. Using ADIOS with IOR provides an easy way to stress the file system while handling file I/O with the ADIOS API. A side benefit is that Aequilibro can be tested without any additional code modification. The third part of the evaluation provides the comparison of IOR Default and *IOR TAPP-IO*.

For the *single shared file* (SSF) I/O strategy, we use three different variants of the IOR benchmark setups: (I) *IOR Default SSF*, (II) *IOR Optimized SSF*, and (III) *IOR TAPP-IO SSF*. Variant (I) uses the Lustre default striping (stripe count = 4, stripe size = 1MB). Variant (II) uses optimized a striping information (stripe count = numberOfWriters, stripe size = fileSize / numberOfWritingTask), but the Lustre default OST placement. Variant (III) uses the same stripe count and size as (II), but utilizes the BPIO balancing algorithm to obtain MPI process ID to OST binding. This list is used to set the specific striping information. The metrics of interest include the overall execution time and the end-to-end I/O *performance improvement* gained by using either BPIO, Aequilibro or TAPP-IO. It is provided in percentage and calculated with the following equation:

$$\text{Performance Improvement} = 100 * \left(\frac{\text{BW}_{balanced}}{\text{BW}_{default}} - 1 \right) \quad (2)$$

2) HPC Workload: S3D is a combustion code simulation that is widely used on HPC systems. It generates a large amount of I/O requests. Verifying the I/O performance improvement of S3D with TAPP-IO provides us with a good indicator of the impact on other large-scale applications. Genarray is an S3D workload simulator provided by ADIOS. We utilize the pre-loadable version of TAPP-IO to demonstrate its effectiveness. This requires a small modification in the ADIOS source code. By default, the `MPI_AGGREGATE` transport method sets the striping information for files. In order to run ADIOS with TAPP-IO, we remove the part that specifies the striping information. In Genarray, three dimensions of a global array are partitioned among MPI processes along X-Y-Z dimensions in the same block-block-block fashion. Each process writes an N^3 partition. The size of each data element is 4 bytes, leading to the total data size of $N^3 * P * 4$ bytes, where P is the number of processes. One key difference between the IOR benchmark tool and Genarray is that by default Genarray utilizes all cores present on a compute node. This improves the computational efficiency of the simulation. On the other hand, Genarray generates pressure on single storage targets, because each compute node hosts its own operating system with a single mount point per file system.

3) Benchmarking Parameters: In order to accurately model an HPC workload behavior, the benchmark parameters need to be aligned with the desired workload. The IOR benchmark provides a wide range of parameters including *API*, *FilePerProc*, *WriteFile*, *NumTasks*, *BlockSize*, and *TransferSize*. On Titan, the memory size per node is 32 GB with 2 GB per processor. We run the IOR with different blocksizes to evaluate the impact of caching effects and a *TransferSize* of 1 MB. For POSIX I/O, the *fsync* and *useO_DIRECT* options are set. O_DIRECT bypasses I/O and file system buffers. For MPI-IO, the same effect can be achieved by enabling the *direct_io* MPI-IO optimization hint. For Lustre-specific settings, each file is created with a stripe size of 1 MB and in the case of file-per-process mode, a *StripeCount* of 1. The stripe size should be aligned with the *TransferSize* in order to get the best performance. *StripeCount* specifies the number of OSTs where the data is striped across while *StripeSize* defines the size of one stripe. The default Lustre stripe count is 4.

4) Experimental Setup: All tests were performed on the Titan supercomputer. In order to get representative results,

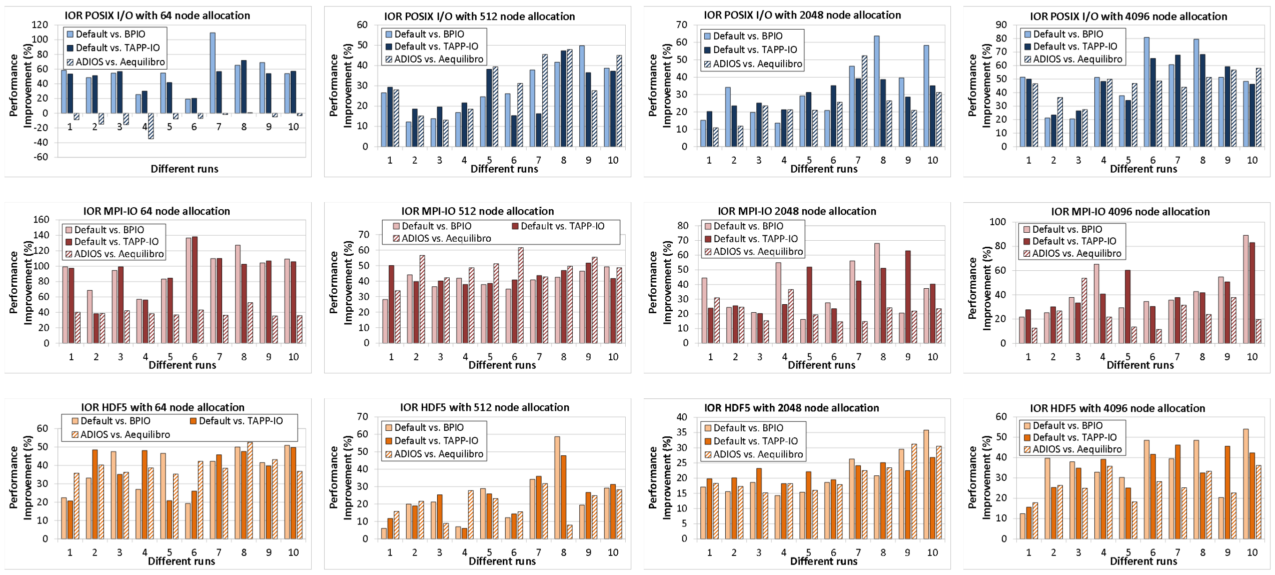

Fig. 4. Performance improvements for IOR large-scale runs.

two major issues are addressed. First, all experiments are conducted in a busy production environment. No tests are run during the quiet maintenance mode. The results show that performance gains can be achieved in an active production environment. Second, a broad set of compute nodes are used instead of just a certain subset of nodes. This demonstrates that independently from any specific node set on Titan, an application can readily benefit from the presented balancing framework. The *application level placement scheduler* (ALPS) on Titan returns a node allocation list where nodes tend to be logically close to each other. There are two attempts to get a higher node coverage. The first one is to submit scaling tests one after another independently, in the hope that a different set of compute nodes is covered with every run. The second attempt is to submit scaling runs in parallel to occupy a larger set of nodes. Both approaches are used to get a broader coverage. All of our experiments are conducted in a noisy, active production environment. Therefore, performance numbers may not always be conclusive. To cope with this issue and to draw consistent observation, multiple tests are performed with at least three repetitions per run. Different runs are allocated on different sets of nodes, enabling us to cover a broad set of compute nodes on Titan.

B. Synthetic Benchmark Results

The results of the scaling runs with a 4 GB file size per writing process and file-per-process strategy are summarized in Figure 4 for 64, 512, 2,048, and 4,096 nodes. Over a period of four months, more than 30 scaled runs per node allocation size were obtained. Each sub-figure represents a particular node allocation. The X-axis represents the enumeration of runs with the same count of node allocation, but for *different* sets of nodes. We compare the bandwidth performance of IOR Default and IOR BPIO (denoted as *Default vs. BPIO*), IOR Default and IOR TAPP-IO (denoted as *Default vs. TAPP-IO*), and IOR ADIOS and IOR Aequilibro (denoted as ADIOS vs. Aequilibro) utilizing Equation 2. In all cases, it can be seen that the balancing provides significant performance improvements for small-, medium-, and large-scale runs. An exception is the performance for Aequilibro at smaller scales for POSIX I/O. IOR BPIO and IOR TAPP-IO show similar performance improvement trends. For POSIX I/O, TAPP-IO provides about 40% of performance improvement for 2,048 nodes and about 50% for 4,096 nodes. Similar trends can be observed for HDF5 and MPI-IO at large-scale. For 4,096 nodes, TAPP-IO provides up to 89% of performance improvement for MPI-IO and 54% for HDF5. It is noteworthy that the performance improvement achieved by Aequilibro is inferior to BPIO and TAPP-IO. The additional overhead introduced by the I/O middleware framework lowers the overall I/O performance. While there are variations across different runs, it can be observed that the trend remains the same. There are consistent performance gains across multiple runs and iterations. Optimizing the overall I/O cost leads to a reduced application execution time (especially for large-scale runs) and therefore, to a reduced operational cost per executed application.

Figure 5 summarizes the IOR bandwidth results for different file sizes scaling from 8 to 4,096 nodes for file-per-process. The results illustrate the average bandwidth per second from over more than 40 scaled runs with at least three repetitions per IOR variant (refer to Table I) per node allocation within one run. The results were collected over the period of four months. From the throughput results, we make the following observations. First, starting from small-scale runs with at least 16 nodes, our load balancing framework TAPP-

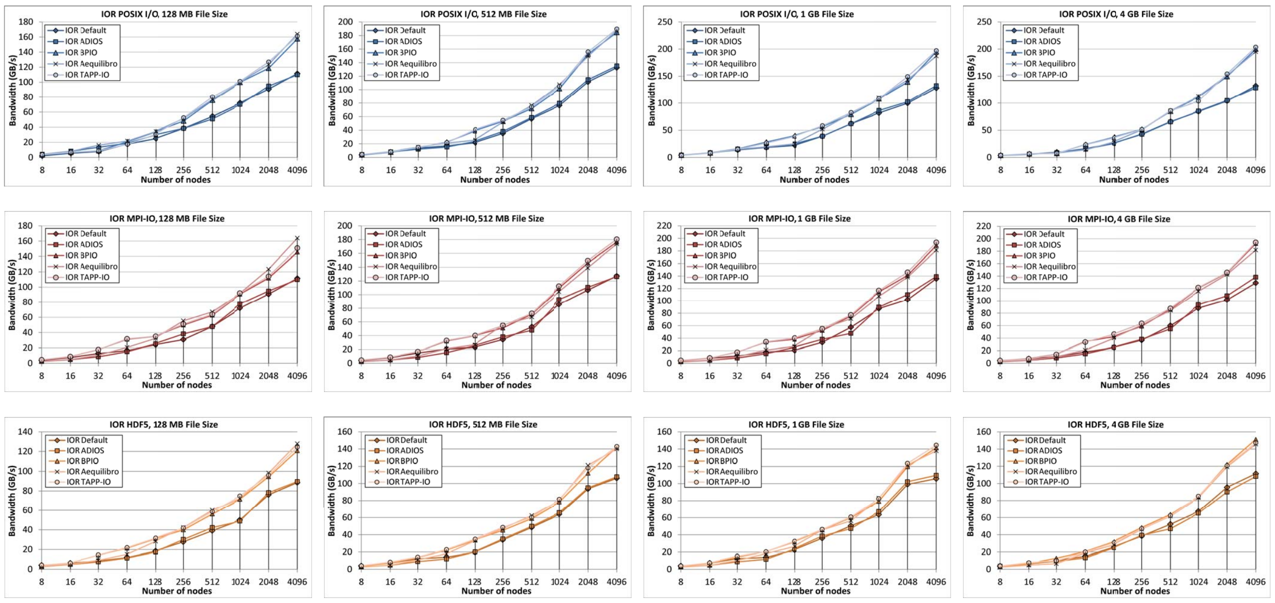

Fig. 5. IOR bandwidth performance for IOR (I) to (V) for different file sizes.

IO and the BPIO library both provide consistent bandwidth improvements. Aequilibro does not provide any significant improvement for less than 128 nodes. Second, as we scale up in terms of I/O processes and allocated computing nodes, POSIX, MPI-IO, and HDF5 benefit from utilizing TAPP-IO with IOR. For example, IOR TAPP-IO with POSIX I/O achieves up to 202.7 GB/s on average for a 4,096 node allocation and a 4 GB file size per writing process. This can be translated to 54% performance improvement compared to the default data placement with IOR Default. For smaller file sizes, the maximum bandwidth is less, but the average performance improvement trend remains the same compared with the default data placement. This consistent improvement can be observed for MPI-IO and HDF5 as well. Another noteworthy aspect is that the expected caching effects for smaller file sizes were non-existent. In summary, it can be said that all IOR variants utilizing a data balancing algorithm are able to provide similar performance results. But, TAPP-IO makes the application independent from the need to actively adopt the BPIO mechanism by integrating it into the application or using an I/O framework such as Aequilibro. Note, the performance varieties that could be observed in previous work [6], [34] are leveled for POSIX I/O and MPI-IO. We think that this is the effect of only creating the Lustre file descriptor for MPI-IO instead of pre-creating a file via the POSIX interface. Figure 6 shows the performance improvement averaged over all completed runs for file-per-process I/O and a 4 GB file size per writing process (see also Figure 4). It confirms that TAPP-IO consistently provides a higher throughput with the balanced placement algorithm. The only exceptions are MPI-IO for 8 nodes and HDF5 for 32 nodes.

Figure 7 presents the average application execution time of IOR Default and IOR TAPP-IO for MPI-IO with file-per-process. The percentage on top of the bars describes the time improvement. A similar trend can be observed for POSIX I/O and HDF5. The results are not displayed due to the brevity of the paper. From the results, we conclude that resolving resource contention at the storage system level directly impacts

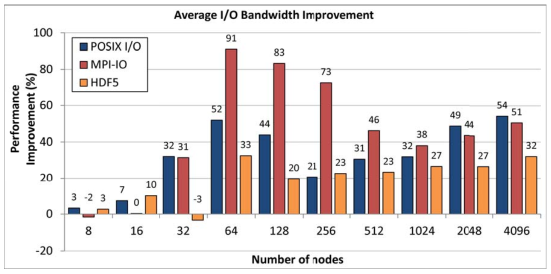

Fig. 6. Average performance gains (in %): TAPP-IO vs. Default placement.

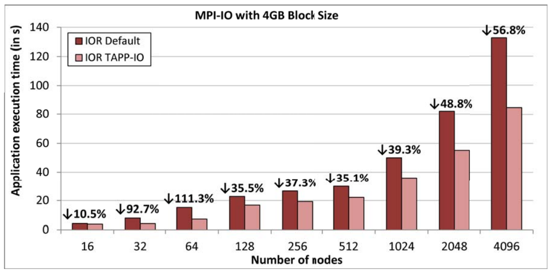

Fig. 7. Average application execution time for MPI-IO with 4 GB block size.

the overall execution time of the application. With the prospect of big data and the increasing amount of defensive I/O in mind, we expect that the balancing mechanism will have a tremendous effect on an application's performance.

The IOR benchmark output provides the standard deviation and mean calculated from the performance results of the executed repetitions per run. While evaluating the collected benchmarks results, we constantly were able to make the following observation for all IOR variants utilizing a balancing algorithm independently from the I/O interface. For TAPP-IO, the standard deviation is tremendously lower for all tested file sizes. The standard deviation is a measure to quantify the amount of variation of a set of data values. In other words, when utilizing the BPIO, the achieved bandwidth of each repetition is relatively close while using the Lustre default data placement leads to a huge variation among repetitions.

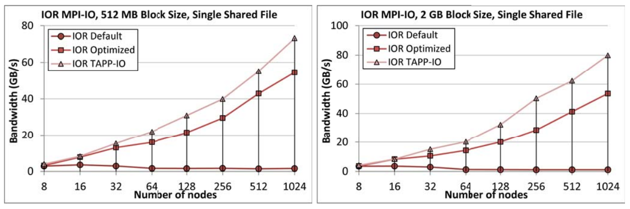

Fig. 8. IOR bandwidth performance for MPI-IO and single shared file.

Figure 8 displays initial performance results of TAPP-IO with the single shared file balancing algorithm for IOR with MPI-IO for different block sizes. The result present the average bandwidth out of 30 scaled runs with node allocations ranging from 8 to 1,024 nodes with one writing process per allocated node. Similar to the file-per-process mode, TAPP-IO provides significant performance improvement starting with a node allocation as minimal as 32 nodes. The default Lustre striping pattern throttles the throughput tremendously. The default striping pattern distributes the file over 4 OSTs with a striping size of 1. Multiple writing processes try to access the same OST at the same time. The optimized IOR version provides an increasing bandwidth compared to the default variant, but still utilizes the Lustre default OST placement like the file-per-process results. The observed performance gain by distributing stripes of the same file evenly among available storage targets is consistent with the observations made for the file-per-process I/O pattern. For example, TAPP-IO provides a performance improvement of about 75.8% compared to the optimized placement for 256 nodes. Another consistent observation that should be noted is that the standard deviation results obtained from different iterations within the same run was relatively small for TAPP-IO compared to the results obtained with the Lustre default data placement.

C. HPC Application Results

We perform scaled runs with 128, 256, 512, 1,024, 2,048, and 4,096 nodes which correspond to 2,048, 4,096, 8,192, 16,384, 32,768, and 65,536 MPI processes, respectively. We

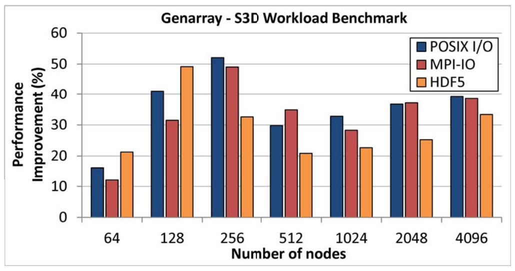

Fig. 9. Average I/O bandwidth improvement for S3D workload.

use weak scaling of the problem size grid such that each process generates an 8 MB output/checkpoint file periodically (10 checkpoints in each run). The I/O bandwidth measurement is performed for default (ADIOS) and balanced data placement (ADIOS with TAPP-IO) by running three Genarray simulations within the same allocation. Figure 9 displays the summary of the I/O bandwidth improvements observed for S3D-IO. The improvements are averaged over ten runs for each configuration. It can be observed that even for small node count runs the performance can be improved. For large-scale runs, we observe that TAPP-IO significantly improves the I/O bandwidth. This is consistent with the IOR synthetic benchmark performance results. For large node/processor counts, applications can directly benefit from TAPP-IO without any additional code changes.

V. Conclusions

This work attempts to resolve I/O contention in busy HPC environments, by introducing TAPP-IO, a dynamic, shared data placement framework that mitigates resource contention and load imbalance at the lowest level, thereby improving the application-level performance. TAPP-IO introduces a balancing algorithm for the file-per-process and single-shared-file I/O patterns and supports HDF5, MPI-IO, and POSIX I/O. It does not require any source code modifications and acts as a transparent auto-tuning layer for parallel I/O performance.

The effectiveness of the TAPP-IO framework is evaluated in comparison to our past work for POSIX I/O, MPI-IO, and HDF5. We utilize IOR, a synthetic benchmark, and a real-world HPC workload. Our results show that TAPP-IO translates the benefits of BPIO transparently into an application while providing consistent performance improvements for different node allocations. For example, POSIX I/O MPI-IO can be improved by up to 50% on per job basis while HDF5 shows performance improvements of up to 32%. The simplicity of the integration shows that TAPP-IO is a viable solution for improving the overall I/O performance.

Future work will include the performance evaluation with scientific HPC workloads, especially for the single-shared-file strategy. Although our evaluation is centered around Titan and Spider II, load imbalance and resource contention are a common problem in large-scale HPC systems. We believe that TAPP-IO and our proposed techniques can be applied to HPC platforms that lack a centralized resource manager.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

References

[1] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An Overview of the HDF5 Technology Suite and its Applications," in *EDBT/ICDT 2011 Workshop on Array Databases (AD '11)*, 2011, pp. 36–47.

[2] R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance," in *6th Workshop on I/O in Parallel and Distributed Systems*, 1999, pp. 23–32.

[3] P. J. Braam *et al.*, "The Lustre Storage Architecture," 2004.

[4] B. Xie, Y. Huang, J. S. Chase, J. Y. Choi, S. Klasky, J. Lofstead, and S. Oral, "Predicting Output Performance of a Petascale Supercomputer," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2017, pp. 181–192.

[5] F. Wang, S. Oral, S. Gupta, D. Tiwari, and S. Vazhkudai, "Improving Large-scale Storage System Performance via Topology-aware and Balanced Data Placement," in *20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014, pp. 656–663.

[6] S. Neuwirth, F. Wang, S. Oral, S. Vazhkudai, J. Rogers, and U. Bruening, "Using Balanced Data Placement to Address I/O Contention in Production Environments," in *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2016, pp. 9–17.

[7] S. Neuwirth, S. Oral, F. Wang, and U. Bruening, "An I/O Load Balancing Framework for Large-scale Applications (BPIO 2.0)," Poster at *2016 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC16)*, 2016.

[8] S. Oral, D. A. Dillow, D. Fuller, J. Hill, D. Leverman, S. S. Vazhkudai, F. Wang, Y. Kim, J. Rogers, J. Simmons *et al.*, "OLCF's 1 TB/s, Next-generation Lustre File System," in *Cray User Group Conference (CUG 2013)*, 2013.

[9] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A Centralized "Zero-queue" Datacenter Network," in *2014 ACM Conference on SIGCOMM (SIGCOMM '14)*, 2014, pp. 307–318.

[10] M. Ezell, D. Dillow, S. Oral, F. Wang, D. Tiwari, D. E. Maxwell, D. Leverman, and J. Hill, "I/O Router Placement and Fine-Grained Routing on Titan to Support Spider II," in *Cray User Group Conference (CUG 2014)*, 2014.

[11] A. S. Bland, J. C. Wells, O. E. Messer, O. R. Hernandez, and J. H. Rogers, "Titan: Early Experience with the Cray XK6 at Oak Ridge National Laboratory," in *Cray User Group Conference (CUG)*, 2012.

[12] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and Integration for Scientific Codes Through The Adaptable IO System (ADIOS)," in *6th International Workshop on Challenges of Large Applications in Distributed Environments*, 2008, pp. 15–24.

[13] S. Ahern *et al.*, "Scientific Discovery at the Exascale: Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis and Visualization," http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Exascale-ASCR-Analysis.pdf, 2011.

[14] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 750–759.

[15] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC Applications Under Congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015, pp. 1013–1022.

[16] S. Herbein, D. H. Ahn, D. Lipari, T. R. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer, "Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 69–80.

[17] M. Luo, D. K. Panda, K. Z. Ibrahim, and C. Iancu, "Congestion Avoidance on Manycore High Performance Computing Systems," in *26th ACM International Conference on Supercomputing (ICS '12)*, 2012, pp. 121–132.

[18] N. Jiang, L. Dennison, and W. J. Dally, "Network Endpoint Congestion Control for Fine-grained Communication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15)*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 35:1–35:12.

[19] Y. Li, X. Lu, E. L. Miller, and D. D. Long, "ASCAR: Automating contention management for high-performance storage systems," in *31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–16.

[20] Y. Zhu, H. Jiang, X. Qin, D. Feng, and D. R. Swanson, "Improved Read Performance in a Cost-effective, Fault-tolerant Parallel Virtual File System (CEFT-PVFS)," in *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2003, pp. 730–735.

[21] R. B. Ross, R. Thakur *et al.*, "PVFS: A Parallel File System for Linux Clusters," in *4th Annual Linux Showcase and Conference*, 2000, pp. 391–430.

[22] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A Multiplatform Study of I/O Behavior on Petascale Supercomputers," in *24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*, 2015, pp. 33–44.

[23] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access through Continuous Characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, pp. 8:1–8:26, October 2011.

[24] Q. Liu, N. Podhorszki, J. Logan, and S. Klasky, "Runtime I/O Re-Routing + Throttling on HPC Storage," in *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, 2013.

[25] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska, "End-to-end Performance Isolation Through Virtual Datacenters," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, 2014, pp. 233–248.

[26] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized Task-aware Scheduling for Data Center Networks," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014, pp. 431–442.

[27] K. Yelick *et al.*, "The Magellan Report on Cloud Computing for Science," http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Magellan_Final_Report.pdf, 2011.

[28] OpenSFS, "Lustre 2.7.0 Released," http://lustre.org/lustre-2-7-0-released/, March 2015.

[29] R. Mohr, M. J. Brim, S. Oral, and A. Dilger, "Evaluating Progressive File Layouts For Lustre," in *Cray User Group Conference (CUG 2016)*, 2016.

[30] H. Luu, B. Behzad, R. Aydt, and M. Winslett, "A multi-level approach for understanding I/O activity in HPC applications," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2013, pp. 1–5.

[31] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. Hawkes, S. Klasky, W. Liao, K. Ma, J. Mellor-Crummey, N. Podhorszki *et al.*, "Terascale Direct Numerical Simulations of Turbulent Combustion using S3D," *Computational Science & Discovery*, vol. 2, no. 1, p. 015001, January 2009.

[32] LLNL, "The Interleaved Or Random (IOR) Benchmark," https://github.com/LLNL/ior, May 2017.

[33] H. Shan, K. Antypas, and J. Shalf, "Characterizing and Predicting the I/O Performance of HPC Applications using a Parameterized Synthetic Benchmark," in *2008 ACM/IEEE Conference on Supercomputing (SC08)*, 2008, pp. 42:1–42:12.

[34] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing Variability in the IO Performance of Petascale Storage Systems," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*, 2010, pp. 1–12.