2018 IEEE International Parallel and Distributed Processing Symposium

# A Migratory Heterogeneity-Aware Data Layout Scheme for Parallel File Systems

Shuibing He*, Xian-He Sun†, Yang Wang§, and Chengzhong Xu§
*School of Computer Science, Wuhan University, Wuhan, Hubei 430072, China
†Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, USA
§Shenzhen Institute of Advanced Technology, Chinese Academy of Science, Shenzhen, China
heshuibing@whu.edu.cn, sun@iit.edu, {yang.wang1, cz.xu}@siat.ac.cn

***Abstract*—Parallel file systems (PFSs) are widely deployed to speed up the performance of high-performance computing (HPC) applications. In recent years, hybrid PFSs that consist of HDD-SSD servers, have attracted much attention in HPC community. However, existing data layout schemes do not well consider the characteristics of heterogeneous servers and heterogeneous access patterns, thus may experience considerable inefficiencies. In this study, we propose MHA, a migratory heterogeneity-aware data layout scheme to improve the data distribution of hybrid PFS. More specifically, to accommodate heterogeneous access patterns, MHA first migrates file data into several regions, each with similar access patterns. Then, by leveraging a data access cost model, MHA determines the appropriate stripe sizes on heterogeneous servers to get the best performance on each region. We have implemented MHA under MPI-IO library on top of OrangeFS file system. Experimental results show that MHA can significantly improve the hybrid PFS I/O system performance compared to existing data layout schemes.**

***Index Terms*—Parallel I/O System; Parallel File System; Solid State Drive; Data Layout**

## I. Introduction

Scientific applications are getting increasingly data-intensive. This makes I/O the major performance bottleneck of many high-performance computing (HPC) applications. To address this I/O bottleneck issue, parallel file systems (PFS), such as OrangeFS [1], Lustre [2], and GPFS [3], are widely deployed in HPC centers. By utilizing the parallelism of multiple storage servers, PFSs can greatly improve the system I/O bandwidth and storage capacity. Although PFSs deliver decent peak I/O performance, they still fail to perform well for certain common-case access patterns in reality [4], [5].

Flash-based solid state drives (SSDs) provide an alternative solution for I/O system designs. Comparing with traditional hard disk drives (HDDs), SSDs have an order of magnitude higher performance, thereby becoming an ideal medium to build high-performance I/O systems [6]. Nevertheless, because of the high costs, it might be impractical for SSDs to completely replace HDDs in large-scale PFSs. Therefore, hybrid PFS, which consists of both HDD-based servers and SSD-based servers, has attracted much attention in recent HPC systems [7], [8]. For the rest of this paper, we refer to the server equipped with an HDD as HServer and the server with an SSD as SServer.

Although promising, the efficiency of hybrid PFS is closely related to its data layout scheme, which defines the way how a file distributes its data on underlying servers. Typical layout schemes often distribute file data across multiple servers using a fixed-size stripe [5], [9]. The benefits of these designs are even data distribution on servers and decent I/O bandwidth in certain cases. Although widely used in modern PFSs, these schemes are only amenable to homogeneous access patterns and file servers. With the emergence of hybrid PFSs and heterogeneous access patterns, existing layout schemes suffer from the following drawbacks.

First, the heterogeneous access patterns may compromise the efficiencies of parallel I/O systems. Previous studies [10]–[12] have pointed out that many HPC applications have heterogeneous patterns. For example, request size can be large at one file chunk but small at another; request type can be read operation in one I/O phase but write in another; request concurrency is high in one file location but low in another. However, existing schemes adopt a fixed striping method for the entire file, regardless of the access pattern changes at different parts of the file. As a specific layout policy is often efficient for a certain type of access patterns [13], these one-size-fits-all schemes will entail I/O system inefficiencies for applications with complicated access patterns.

Second, the heterogeneous servers can also offset I/O performance. In hybrid PFS, as SServer is much faster than HServer, it takes less I/O time to process the same amount of data than the HServer. Nevertheless, current layout schemes do not pay much attention to the performance disparity between heterogeneous servers and still assign fixed-size file stripes to them. This usually makes each server process the same amount of data, leading to severe load imbalance among different types of servers. As we have shown in our previous work [8], the load difference among HServers and SServers can be 3.5X, yielding considerable I/O system inefficiencies.

A number of approaches have been proposed to address these issues in the data layout for parallel I/O systems. These efforts, for example, include optimizing file stripe size [10], [14]–[16], refining the number of servers to distribute file data [13], [17], optimizing the data scope to apply a specific layout [10], and their combinations [5], [8], [18]. We note that all these techniques optimize file data layout directly based on the *inherent* data access orders in parallel files, without grouping I/O requests with the same type of patterns. This is insufficient to address the workload and server heterogeneity

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00122

in hybrid PFS, meaning that the potential of the storage system is not fully utilized.

In this paper, we propose MHA, a migratory heterogeneity-aware data layout scheme to optimize the data placement of hybrid PFS. To accommodate complex access patterns, MHA first divides file requests into different groups, each with similar access patterns, and then stores the data of each group into a separate region. Since a specific data layout is often efficient for a certain type of access patterns, such data grouping (reordering) can choose a specific optimized data layout, which is better to fit the file requests in that reordered region. Then, by leveraging a proposed data access cost model, MHA determines the variable stripe sizes on the heterogeneous servers to achieve the best performance on each region. In this way, all groups of reordered data are configured with their individually optimized data layouts, rather than sharing a global but inefficient layout.

To the best of our knowledge, this is the first attempt to leverage data reordering and variable file stripes to enhance the data layout for parallel I/O systems. MHA is designed as an I/O middleware scheme, thus it is transparent to applications and requires no changes of applications. Moreover, MHA is independent of the file system layer, thus can be applied to many kinds of PFSs, such as OrangeFS and Lustre.

In summary, this study makes the following contributions.

- We propose a data reordering method, which analyzes file access patterns and migrates file data into different regions, each with similar access patterns.
- We present a varied-size file striping scheme, which determines the appropriate stripe sizes on heterogeneous servers for each region based on their performance.
- We implement the MHA scheme in MPICH2 [19] library on top of OrangeFS file system. Extensive experimental results show that MHA can significantly improve the I/O throughput of hybrid PFSs.

In practice, MHA is not necessary to apply to the entire file system, but rather to critical data sets and data sections. It is an effective tool for I/O performance optimization.

The remainder of this paper is organized as follows. Section II presents the background and motivation of this work. The design and implementation of MHA are described in Section III and Section IV, respectively. Section V presents the extensive evaluation of MHA. Section VI surveys related work. Finally, the conclusions are summarized in Section VII.

## II. Background and Motivation

### A. Existing File Striping Schemes

Typical layout schemes often distribute file data across different servers using a fixed-size stripe in a round-robin fashion, as shown in Fig. 1. In this example, a file is placed on two HServers and two SServers using a fixed-size stripe, e.g., 64KB. Although this policy leads to even data distribution and decent I/O performance for homogeneous servers, it is inefficient for hybrid PFS. For example, assuming that each process issues requests at a size of 256KB, then a file request

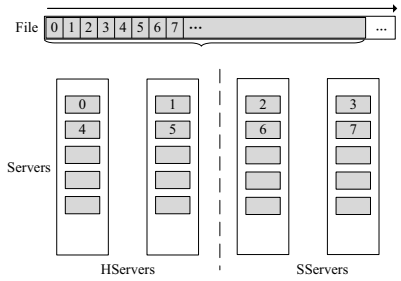



Fig. 1: Traditional fixed-size file striping scheme.

is divided into four sub-requests, each serviced by one server at a size of 64KB. Since the SServers have relatively high speed, they will complete the sub-requests much faster than the HServers. Consequently, the SServers make no contribution to the overall system performance because the I/O time of a file request depends on the slowest sub-requests which are typically on the HServers.

One solution to dealing with the server heterogeneity is to distribute data across the servers using varied-size stripes [16]. By assigning the SServers with larger stripe sizes and the HServers with smaller ones, all the servers can finish their sub-requests nearly at the same time, thereby alleviating the load imbalance among the servers. However, this approach is only suitable for uniform access patterns. As the data access patterns can be largely heterogeneous, such one-size-fits-all policy is still inefficient for heterogeneous data accesses. Using Fig. 1 as an example, a stripe pair of <32KB, 96KB> [1] may be efficient for a request size of 256KB, but it fails to perform well for a request size of 16KB in a different location because of the under-utilized I/O parallelism on the servers.

### B. Motivation of MHA

To address the above issues, we have proposed an heterogeneity-aware region-level (HARL) data layout scheme for hybrid parallel I/O systems. The term "region" refers to a logical scope of a file. As shown in Fig. 2, HARL first divides a file into several small regions, and then adjusts the stripe sizes on the servers for each region based on the server performance. More details can be found in our previous work [8]. Although HARL often helps boost the I/O performance, it only tries to optimize the data layout for all inherent requests in each region, instead of an individual request. When the access pattern of each request differs considerably, it is still hard for HARL to find an efficient stripe pair for all the requests.

While the heterogeneous access patterns may exist in a parallel file [10]–[12], we find that similar access patterns may exist in the heterogeneous accesses from a global file view. To illustrate this observation, we analyze the I/O trace of LANL anonymous application [20]. For each loop in the application, there are three I/O operations, one small request with 16 bytes, and followed by two large requests with 128K-16 bytes and

[1] A stripe pair <h, s> means the stripe sizes on the HServers and the SServers are h and s, respectively.

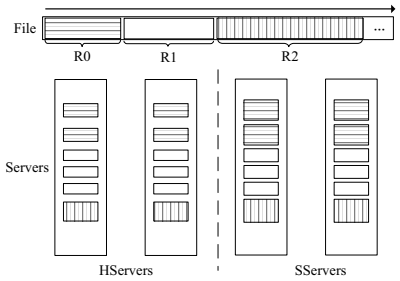



Fig. 2: Heterogeneity-aware region-level data layout scheme.

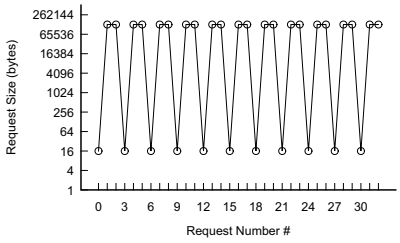



Fig. 3: Data access sequence in a loop of LANL application.

128 KB, respectively. Fig. 3 shows the size of each request in the access sequence. We can see that the I/O requests with the same size do not exist in a successive sequence but they exist in the request sequence across the file. In addition to LANL application, we can obtain the same observation in two other applications (LU [21] and Cholesky [22] in Section V).

As a given layout is often efficient for a certain type of access patterns, we can choose a specific optimized layout better fitting the requests in a reordered region if we group the requests with similar patterns (e.g., similar or same request size) into the region. Since each region is configured with its own efficient data layout, the overall file system performance can be largely improved. This motivates us to propose the MHA layout scheme.

## III. Design

In this section, we first introduce the basic idea of MHA, and then describe its architecture, followed by the details of each key component in MHA.

### A. Idea of MHA

MHA aims to optimize the hybrid PFS layout by leveraging data reordering. More specifically, to adapt to heterogeneous access patterns, MHA first migrates file data with similar access patterns into different regions before making a placement policy, and then determines the appropriate stripe sizes on the servers based on their performance to accommodate heterogeneous servers for each region.

Fig. 4 illustrates the idea of MHA scheme. In this example, there are six requests (A0-2 and B0-2) that access a parallel file. Requests labeled as Ax or Bx have similar access patterns. Before determining the optimized data layout, MHA migrates A0-2 and B0-2, respectively. In other words, it constructs

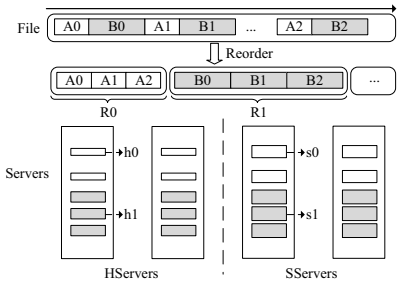



Fig. 4: Migratory heterogeneity-aware data layout scheme.

two new regions: R0 and R1. Then, for each region, MHA determines the stripe sizes for the HServers and the SServers based on the access patterns and the server performance. For example, the optimized stripe sizes for R0 and R1 are <h0, s0> and <h1, s1>, respectively. Compared to existing approaches that directly optimize the file data layout based on the inherent accesses in the original file, MHA can choose a specific optimized data layout, which is better to fit the requests in each reordered region, thus can improve the overall file system performance.

The proposed scheme needs a prior knowledge of data access patterns. Fortunately, many HPC applications have predictable access patterns [17], [23], [24]. This is because the data accesses of HPC applications are mostly determined by their inherited numerical methods, not input data. For example, the BTIO application [25] that solves block-tridiagonal matrices has this feature. Once the parameters, such as the size of array, the number of time steps, etc., are given, the I/O behaviors of BTIO can be accurately predicted. Because HPC applications often run multiple times to process different datasets, this feature facilitates MHA to achieve the optimized data layout based on I/O trace analysis. Similar approaches have also been used numerous times in data partition, data replication, and data prefetching to successfully improve I/O system performance [17], [23], [26].

### B. System Overview

Fig. 5 shows the system overview of MHA scheme. It resides at the I/O middleware layer. Application processes call the MPI-IO library to access the file data. MHA profiles the

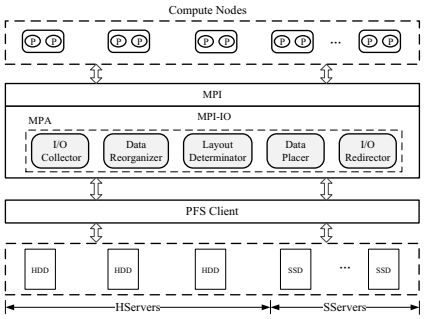



Fig. 5: System Overview of MHA.

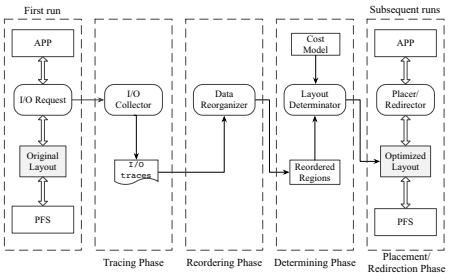



Fig. 6: The workflow of MHA scheme.

application on-line during the application's first run, and makes data placement optimization off-line based on I/O access patterns and server performance analysis. Once the optimized layout policy is obtained, MHA will place file data accordingly for the subsequent runs of the application, which will greatly improve the system performance.

MHA follows a five-phase process, as shown in Fig. 6. The *I/O Collector* in the *tracing phase* profiles application's I/O accesses. In the *reordering phase*, the *Data Reorganizer* analyzes I/O traces and groups file data into several regions, each with similar access patterns. In the *determination phase*, the *Layout Determinator* determines the stripe sizes for each server based on a data access cost model. In the *placement phase*, each region is placed on heterogeneous servers with the optimized stripe sizes by the *Placer*. Finally, in the *redirection phase*, the I/O requests are sent to the proper locations by the *Redirector* at runtime in the subsequent runs of the application.

### C. I/O Access Collection

Although there are some techniques and tools that can be used for this purpose, we choose IOSIG [27] to collect I/O accesses because it can capture the required information of MHA and yields acceptable overheads. IOSIG is developed as a pluggable library at MPI-IO layer, which supports MPI-IO and standard POSIX IO interfaces for portable deployment. IOSIG profiles all file operations of a parallel application and records this information in several trace files.

The collected information includes process ID, MPI rank, file descriptor, request type, file offset, request size, and time stamp information. To facilitate the following phases during the layout optimization procedure, file operation records are sorted in an ascending order in terms of their offsets.

### D. Similar Access Detection

A key issue in our design is to identify two requests that bear similar access patterns. A typical solution is to compare the features abstracted from each access, where a pair of similar requests will share many features. While it is possible to enumerate the closest matches by comparing all features, we use request size and request concurrency to characterize each request because they give a good indication of how the parallel file is accessed, which has been discussed in [13]. In our context, the request concurrency refers to the number of requests that are simultaneously issued to the file. Each

**Algorithm 1** Iterative Request Grouping

```
1: procedure RG(R[1,i])
2:     if (i ≤ k) then
3:         for (∀i ∈ [1, k]) do
4:             S_{g_i} ← randomly selected R[t]
5:         end for
6:     else
7:         count ← 0
8:         while (S_{g_i} is changed||count ≤ 3) do
9:             G_i ← arg min_{|G_j|} {||S_{s_j} − S_{g_i}||}
10:            S_{g_i} ← (1/|G_i|) Σ_{S_{s_j} ∈ G_i} S_{s_j}
11:            count++
12:        end while
13:    end if
14: end procedure
```

request $r_i$ can be represented by a point $(x_i, y_i)$ in a two-dimensional $Euclidean\ Space$, where its $x$ and $y$ axises denote respectively the request size and the request concurrency. The distance between any pair of points $S_i(x_i, y_i)$ and $S_j(x_j, y_j)$ can be expressed as

$$||S_i - S_j|| = \sqrt{(\frac{x_i - x_j}{\max\limits_k\{x_k\} - \min\limits_k\{x_k\}})^2 + (\frac{y_i - y_j}{\max\limits_k\{y_k\} - \min\limits_k\{y_k\}})^2} \quad (1)$$

where $\max\limits_k\{x_k/y_k\} - \min\limits_k\{x_k/y_k\}$ represent the maximal distance among the points projected to either $x$ axis or $y$ axis. The normalization is required to enable different dimensions to have a uniform compared space.

We classify all file requests into $k$ groups, each with similar access patterns. Inspired by the data clustering approach in statistics domain [28], we try to find the centers of these groups with an iterative refinement method as shown in Algorithm 1. If the number of requests is less than or equal to $k$, the algorithm randomly selects a request point as the initial value of $S_{g_i}$, which refers to the center of the $i$-th group. Otherwise, each request point is assigned to group $G_i$, whose center is closest to the request point. After all the request points have been processed, the algorithm updates the center for each group. This procedure is repeated until $S_{g_i}$ is no longer changed or three times at most.

One potential issue is that this algorithm may generate too many groups, leading to substantial meta-data management overhead. To overcome this, we limit the value of $k$ to an upper bound. This tuning can guarantee that the number of the groups is bounded by the number of the fixed-size region division method [8]. Another concern is that the computational overhead of the algorithm increases in proportion to the number of requests. However, it runs off-line and only runs once, so the overhead is acceptable in a modern HPC system.

TABLE I: Parameters in the data cost model.

| Symbol | Meaning |
|---|---|
| $o$ | Offset of the file request |
| $l$ | Size of the file request |
| $op$ | Type of the file request (read or write) |
| $M$ | Number of HServers |
| $N$ | Number of SServers |
| $t$ | Unit data network transfer time |
| $\alpha_h$ | Average storage startup time on HServer |
| $\beta_h$ | Unit data transfer time on HServer |
| $\alpha_{sr}$ | Average read startup time on SServer |
| $\beta_{sr}$ | Unit data read transfer time on SServer |
| $\alpha_{sw}$ | Average write startup time on SServer |
| $\beta_{sw}$ | Unit data write transfer time on SServer |
| $h$ | Stripe size on HServer |
| $s$ | Stripe size on SServer |

### *E. Data Reordering*

Once the request grouping process is completed, the *Data Reorganizer* reorders the data in each group into a separate region. The final data placement for each region is carried out by the *Data Placer* in the subsequent runs of the application. In our current design, each region is implemented by a physical file in the same file system, but with an optimized layout. For each region, requests identified to be similar are located together, ordered by their offsets within the original file. That is, a later data block is moved to be adjacent to the first data block it is similar to.

To locate data for an application, the *Data Reorganizer* outputs a *Data Reordering Table* (DRT) to track the data location relationships between the original file and the reordered regions. Each entry in DRT includes five important variables. `O_file` and `O_offset` are the file name and the offset of the data in the original file, `R_file` and `R_offset` are the file name and the offset of the data in the reordered region. `Length` is the size of the data. DRT is updated each time a data location has been changed. By maintaining DRT, the *Redirector* can continuously track the most up-to-date location of the data, which ensures data consistency between the original files and the reordered regions. Once the reordered regions and DRT are obtained, we choose the specific data layout optimization for each region as the following subsection describes.

### *F. Stripe Size Optimization via Cost Model*

For each reordered region, MHA determines the optimized stripe sizes for each server via an access cost model described by the related parameters are listed in Table I. In this model, the cost is defined as the I/O time of a file request in hybrid PFS, and, $h$ and $s$ denote a specific data layout. For a read request $r$, the access cost $T_R(r,h,s)$ with different layouts is defined as follows.

$$T_R(r,h,s) = \max\{p_i \times \alpha_h + s_i \times (t+\beta_h), \\ p_j \times \alpha_{sr} + s_j \times (t+\beta_{sr}) | \forall i \in \mathcal{H}, j \in \mathcal{S}\} \quad (2)$$

where $\mathcal{H}$ and $\mathcal{S}$ are the sets of the HServers and the SServers, and $p_i$, $p_j$, $s_i$, and $s_j$ denote the involved number of processes on HServer $i$ and SServer $j$ and the accumulated sub-request size on HServer $i$ and SServer $j$, respectively. We can calculate $s_i$, $s_j$, $p_i$, and $p_j$ on each server based on the request and the layout information. Eq. 2 only displays the cost for read requests; write cost $T_W(r,h,s)$ is similar except the startup and unit data transfer time for SServers will change. This model assumes all servers offer the same network bandwidth. However, heterogeneous servers provide different I/O performance. Furthermore, the SServers are assumed to offer disparate read and write performance due to their inherent storage characteristics. This model is inspired by our previous work [8], but we extend it by considering I/O concurrency for better cost estimation.

With the proposed model, MHA uses an iterative approach (Algorithm 2) to find the optimized stripe sizes for each type of servers on per-region basis. Starting from $h$ equaling 0, the loop iterates $h$ in 'step' increments while $h$ is less than the upper bound $B_h$. The extreme configuration is where $h$ is 0, which means dispatching the data only on SServer is allowed as long as this leads to enhanced performance. In the second loop, $s$ starts from a size larger than $h$ to avoid load imbalance among heterogeneous servers. For each pair of stripe sizes, the loop iterates to calculate the total access cost of all requests in that region. Note that the operation type is considered here because SServers have disparate read/write performance. Finally, the stripe pair <H, S> with smallest region access cost ($Reg\_cost$) is selected.

Unlike the previous work [8], which uses the average request size as the upper bounds for the potential stripe sizes, this scheme uses an adaptive policy to determine the optimized layout. If the maximal request size ($r_{max}$ ) is relatively small, $r_{max}$ is selected as the bounds. As such, it not only increases the chance to find the optimal layout by traversing more candidates, but also limits the search space. Otherwise, we choose $r_{max}/M$ and $r_{max}/N$ as the bounds. This increases the chance for all the servers to work together, which helps improve the I/O performance for those large requests [8] and avoids unnecessary searching. The 'step' value is 4KB, which can be configured by the user. Generally finer 'step' values result in more precise stripe pairs, but with increased calculation overhead. However, this overhead is usually acceptable as the algorithm is refined and the calculations are simple arithmetic operations that run off-line.

### *G. Data Placement and I/O Redirection*

In the *placing phase*, the reordered regions will be dispatched on respective servers using the pairs of optimized stripe sizes for the subsequent runs of the application. To guide the data placement, such stripe pairs of all the regions are stored into a global *Region Stripe Table* (RST), which is managed by a *Meta-Data Server* (MDS). Upon receiving a file request, a client first contacts the MDS to get the file's meta-data, then it interacts with servers directly. To perform optimized data placement, the MDS looks up the RST according to the request's offset and length, and then returns this information to the client. After that, the client writes the

**Algorithm 2** Region Stripe Size Determination (RSSD)

```
1: procedure RSSD(r[1,k], r_max)
2:     step ← 4KB, opt_cost ← ∞
3:     if R_max < (M + N) * 64KB then
4:         B_h ← r_max, B_s ← r_max
5:     else
6:         B_h ← r_max/M, B_s ← r_max/N
7:     end if
8:     for h ← 0; h ≤ B_h; h ← h + step do
9:         for s ← h + step; s ≤ B_s; s ← s + step do
10:            Reg_cost ← 0
11:            for i ← 1; i ≤ k; i ← i + 1 do
12:                if op(r_i) = read then
13:                    T_i ← T_R(r_i, h, s)          ▷ Call Eq. (2)
14:                else
15:                    T_i ← T_W(r_i, h, s)          ▷ For writes
16:                end if
17:                Reg_cost ← Reg_cost + T_i
18:            end for
19:            if Reg_cost < opt_cost then
20:                opt_cost ← Reg_cost
21:                H ← h, S ← s                      ▷ The opt. layout
22:            end if
23:        end for
24:    end for
25: end procedure
```

file data on the underling servers with the optimized stripe sizes from the RST.

The *I/O Redirector* is responsible for redirecting user's I/O requests to the proper locations. Once receiving a file request, the *I/O Redirector* will first determines the requested regions based on the request offset and request size. Then it examines the DRT with these parameters to find the target regions containing the request. Finally, the read/write operations will be forwarded to the target regions on the underlying HServers or SServers.

## IV. Implementation

We implement MHA within MPICH2 [19] on top of OrangeFS [1]. In the following, we discuss the primary challenges in the implementation.

### A. Key Data Structures

The DRT table saves the data reordering information between original files and the reordered regions. As it is frequently accessed by the *Redirector* and shared by multiple processes, the effectiveness of the DRT is a significant challenge. Similar to the previous work [17], we use *Berkeley DB* [29] to implement the DRT as a database file stored in the same directory as the MPI program. The *Berkeley DB* is configured as a hash table and each record is a key-value pair. We encode the original data access information as the key and use the value to contain the data locations in reordered regions. The RST table is another key structure to record the optimized stripe pairs for all the regions. We also use *Berkeley DB* to implement this structure.

By leveraging the advantage of the light-weighted database, the access contention and meta-data operations are performed in an efficient way. To reduce the size of the in-memory reordering table for efficient lookup, we use a list to maintain frequently accessed reordering entries. Changes to the reordering entries in memory are synchronously written to the storage in order to survive power failures.

### B. I/O Redirection

We modify the MPI library so that the reordering table *DRT* is loaded with `MPI_Init()` and unloaded with `MPI_Finalize()`. To keep track of the location of each original logical file region, *DRT* is stored in a file in the same directory of the MPI program. The reordering table entries are also hashed in memory for efficient table lookup. Changes made to the reordering entries in memory are synchronously written to the storage to survive power failures. We also modify the `MPI_File_read/write()` (and other variants of read/write), so that the user requests can be atomically forwarded to the alternative file servers. In more detail, if the requested regions are found in *DRT*, the logical file regions will be translated to the target regions. Then, the following read/write operations will be forwarded to the target regions on underlying servers.

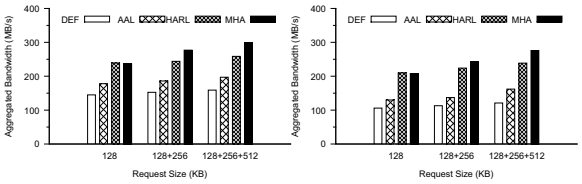
## V. Evaluation

### A. Experimental Setup

We conduct experiments on a SUN Fire Linux cluster, where each node was configured with two Opteron quad-core processors, 8GB memory and a 250GB SATA-II disk. All nodes are equipped with Gigabit Ethernet interconnection, and eight nodes are equipped with additional PCI-E X4 100GB SSD. The operating system is Ubuntu 13.04, the MPI library is MPICH2-1.4.1p1, and the parallel file system is OrangeFS 2.8.6. By default, we employ six nodes as HServers, two as SServers, and the other eight nodes as computing nodes. The file is striped over the file servers in a round-robin fashion.

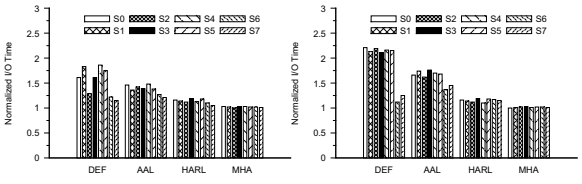
We compare MHA with three other schemes: the default layout (DEF), the application-aware layout (AAL) [10], and the heterogeneity-aware region-level layout (HARL) [8]. For DEF, the data are placed on servers with the default stripe size of 64KB. For AAL, it distributes file data on servers with varied-sized stripes by considering application's access patterns, but it ignores server heterogeneity. HARL takes both access pattern and server heterogeneity into account but without data grouping and migration.

We first evaluate MHA with two micro-benchmarks (i.e., IOR and HPIO) and one macro-benchmark BTIO, and then show the efficiency of MHA using three real application traces (i.e., LANL, LU, and Cholesky).



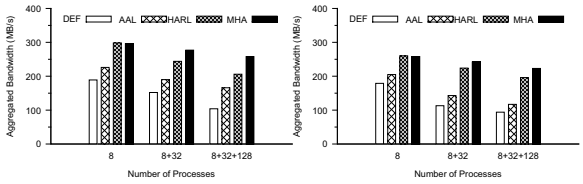

(a) Read (b) Write

Fig. 7: Bandwidths of IOR with mixed request sizes. "$x$+$y$" denotes the mixed request sizes are $x$KB and $y$KB.



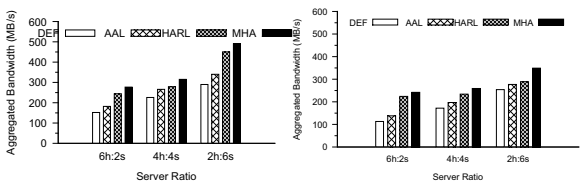

(a) Read (b) Write

Fig. 8: I/O time of each server under different data layout schemes. S0-5 are HServers and S6-7 are SServers.




(a) Read (b) Write

Fig. 9: Bandwidths of IOR with mixed process numbers. "$a$+$b$" refers to the mixed process numbers are $a$ and $b$.




(a) Read (b) Write

Fig. 10: Bandwidths of IOR with various server ratios. "$x$h:$y$s" denotes the system has $x$ HServers and $y$ SServers.

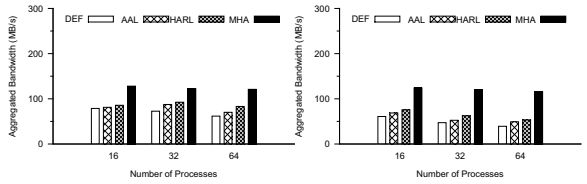
### *B. Micro-Benchmark Results*

**IOR Benchmark:** IOR is a parallel file system benchmark developed at Lawrence Livermore National Laboratory [30]. It provides three APIs: MPI-IO, POSIX, and HDF5, we only use MPI-IO in the tests. During these tests, IOR by default runs with 16 processes, each performing I/O operations on a shared file with request size of 64KB.

Fig. 7 shows the I/O performance of IOR with mixed request size configurations. We modify IOR to run it with various request sizes to simulate heterogeneous patterns. The process number is fixed to 32 and each process issues random requests at multiple sizes to access a 16GB file. The label "128+256" means the mixed request sizes are 128KB and 256KB. Other labels have the similar meanings. We observe that MHA and HARL are always better than DEF and AAL because they are heterogeneity-aware in terms of application and server while DEF and AAL are not. For I/O requests with a single size of 16KB, MHA is comparable to HARL, because it degrades to HARL for uniform access patterns. However, MHA outperforms DEF, AAL, and HARL for all mixed access cases. By using migratory heterogeneity-aware data layout optimization, MHA improves read performance from 63.4% to 88.1%, and write performance from 96.2% to 128.1% respectively, in comparison with DEF. Compared with HARL, MHA improves read performance up to 14.5%, and write performance up to 15.6%. As the request size increases, IOR's bandwidth becomes higher because the increasingly amortized disk seek time reduces the penalty of unbalanced disk accesses on heterogeneous servers.

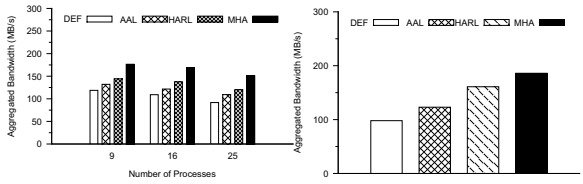
To give a detailed explanation for MHA's performance improvement, Fig. 8 plots the I/O time of each server when the request sizes are mixed by 128KB and 256KB. The result is normalized to the minimum of all servers under the MHA layout. We observe that the I/O loads are largely skewed across the servers under DEF and AAL. This is because these two schemes ignore the performance disparity between the HServers and the SServers. In contrast, HARL and MHA have nearly even loads because they assign variable sized stripes on the heterogeneous servers to achieve load balance. However, MHA has less I/O times on servers. The reason is that it uses more appropriate stripe sizes to further reduce I/O waiting times among the heterogeneous servers.

Fig. 9 shows the I/O bandwidths of IOR with mixed numbers of processes. The request size is fixed to 256KB, and IOR is modified to issue requests with different process configurations. For example, a configuration of "8+32" means IOR sends requests at different parts of the file with 8 and 32 processes respectively. MHA shows comparable performance to HARL when IOR runs with a single process number of 8, because it degrades to HARL for uniform requests. However, for mixed process numbers, MHA improves I/O performance over the three other layout schemes. Compared to DEF, the I/O bandwidth is increased up to 148.5% and write improvements are up to 137.3%. In terms of AAL, MHA improves read performance up to 55.6%, and write performance up to 90.5%. In contrast to HARL, the read performance achieves a 17.6% and 25.4% improvement, and write achieves a 11.0% and 12.1% improvement. As the number of processes increases, the I/O bandwidth gets lower because each server needs to serve more processes' requests and the contention among processes becomes more severe. However the performance degradation of MHA is not as substantial as those of other schemes. The reason is that MHA makes each server handle more even requests. This indicates that MHA a high scalability in terms




(a) Read (b) Write

Fig. 11: Bandwidths of HPIO with various process numbers.




(a) BTIO (b) LANL

Fig. 12: Performance of BTIO and LANL applications.

of the number of processes.

Fig. 10 depicts the I/O performance of IOR with various server ratios. The process number is fixed to 32, and the mixed request sizes are 128KB and 256KB. MHA improves I/O performance for both data reads and writes: read performance increases from 39.3% to 81.6%, while write performance improves from 33.2% to 114.1% in comparison with DEF. Compared to AAL and HARL, MHA achieves an improvement up to 13.6% for reads and 20.8% for writes. In the experiments, read and write performance improve as the number of SServers increases. This is because the requests can be distributed on more SServers, which have high-performance I/O performance. We also note that, with larger number of SServers, the performance improvement of MHA over HARL is more significant. This is because MHA can better utilize the high-performance SServers by applying a migratory data layout scheme.

**HPIO Benchmark:** HPIO is a program designed by Northwestern University and Sandia National Laboratories to systematically evaluate parallel I/O system performance [31]. This benchmark includes three parameters: region count, region spacing, and region size. We modify HPIO to issue mixed request sizes (region sizes) to generate heterogeneous I/O patterns. In our experiment, the region count is 4096, the region space is 0, the region sizes are 16KB, 32KB, and 64KB, and we vary the number of processes from 16 to 64.

Fig. 11 shows the I/O bandwidth of HPIO. We can see that MHA have obvious performance advantages over the other three layout schemes. In contrast to the best of the three counterparts (HARL), MHA can still increase the I/O bandwidth up to 49.3%, 31.8%, and 45.4% respectively for various numbers of processes. It means that MHA is effective with respect to HPIO benchmark. We also note that, as the number of processes increases, the system performance degradation ratio gets more obvious compared that in the IOR test. This is because for small I/O requests disk seek time derived from multi-process contention rather than load imbalance are the major factor that impacts I/O system efficiency. However, MHA is always better than DEF, AAL, and HARL.

### C. Macro-Benchmark Results

BTIO represents a typical scientific application with interleaved intensive computation and read/write I/O phases [25]. We consider the Class *B* and *C* workloads with *simple* subtype in the experiments. To emulate the heterogeneous access pattern situation, we modify BTIO to access a new file with the total size of 1.69GB+6.8GB. Furthermore, each process issues file requests at the sizes of those in Class B and C in an interleaved fashion. We use 9, 16, and 25 processes since BTIO requires a square number of processes. The output file is striped across six HServers and two SServers.

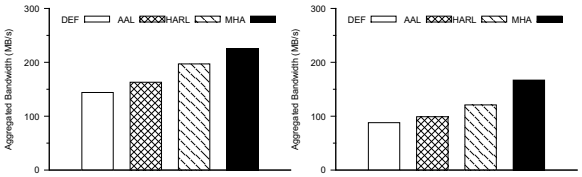
Fig. 12a plots the aggregate I/O bandwidths of BTIO benchmark. Compared with the original I/O system (DEF), MHA achieves 48.6%, 54.3%, and 64.7% improvement with 9, 16, 25 processes, respectively. In contrast to AAL and HARL, MHA also demonstrates performance advantages.

### D. Trace-Driven Results

**LANL Application:** We first evaluate MHA with a real application's I/O trace from LANL [20]. This application has mixed access patterns that can show the adaptivity of our scheme. During the execution of the application, the processes issue requests in a non-uniform way at different locations of a shared file. More specifically, each process issues three types of requests iteratively. The detailed data access patterns of each process are illustrated in Fig. 3. We replay the data accesses of the application according to the I/O trace. In the experiment, we employ eight nodes as computing nodes, six nodes as HServers, and two as SServers. Fig. 12b shows that MHA obtains 89.7%, 51.2%, and 15.6% performance improvement over DEF, AAL, and HARL respectively. The results indicate that MHA is effective for applications with complicated I/O access patterns.

**LU Decomposition:** We replay the data accesses of the application according to the I/O trace [21]. This application computes the dense LU decomposition of an out-of-core matrix [12]. It uses synchronous reads/writes to performs I/O operations, and is driven by an 8192*8192 double precision matrix with a slab size of 64 columns. The dataset is stored in 8 files, one per process. During the run of the application, each process of the parallel program issues read and write requests. The write request size is fixed to 524544 bytes. However, the read request size ranges from 6272 bytes to 524544 bytes. In the experiment, we employ eight nodes as computing nodes, six nodes as HServers, and two as SServers. Fig. 13a shows that MHA obtains 56.2%, 8.1%, and 14.2% performance improvement over DEF, AAL, and HARL respectively. The results indicate that MHA is effective for the LU application.

**Sparse Cholesky:** This application computes Cholesky decomposition for sparse, symmetric positive definite matrices




(a) LU (b) Cholesky

Fig. 13: Performance of LU and Cholesky applications.

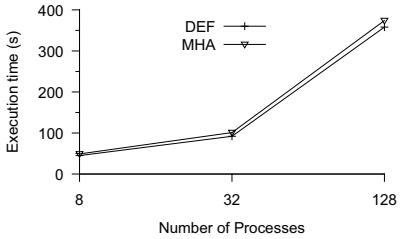



Fig. 14: MHA performance overhead.

[12]. It stores the matrix as panels rather than blocks and conducts synchronous I/O accesses. During the execution of the application, each process sends varied-sized I/O requests at different parts of the file. The read request size ranges from 2 bytes to 4206976 bytes, and write size ranges from 131556 bytes to 4206976 bytes. We replay the data accesses of this application according to the I/O trace [22], by simulating the same application scenario: 8 clients, and same I/O requests for each client. We employ the same server configuration as the previous LU tests. The dataset was stored in 8 files, each for one process. Fig. 13b shows that MHA obtains 78.4%, 58.6%, and 29.6% performance improvement over DEF, AAL, and HARL respectively. We note the I/O bandwidths of Cholesky are smaller than those of LANL and LU though it has the largest request size. This is because the request size of Cholesky varies more considerably and only has a small number of large requests in terms of the two others. However, MHA still brings performance improvements over the other layout schemes. The results indicate that the proposed migratory heterogeneity-aware layout is an effective optimization approach for applications with heterogeneous requests.

### E. Overhead Analysis

*1) Performance Overhead:* In the "tracing phase", the collector incurs overhead during the application's first run. As shown in previous work [27], the on-line profiling overhead is low (2-6% based on our measurement). In the following two phases, since the data reorganizer and layout determinator are carried out off-line and only once, the CPU and memory overhead is also acceptable for most HPC computing systems.

In the "redirection phase", the redirector needs to determine where to send the requests. It incurs additional overhead in file open and file read/write operations. To show the introduced redirection overhead, we run IOR with mixed request sizes of 4KB and 64KB. We intentionally do not make data reordering so that I/O requests are redirect to the original I/O system. The number of process is set to 8, 32, and 128. Fig. 14 shows that with various processes the redirection overhead is acceptable.

*2) Meta-data Space Overhead:* To maintain data consistency, MHA stores the data reordering information in the DRT in the same parallel file system, incurring additional storage space. The system has a maximal space overhead when all the requests are of 4KB. Assuming that the available storage space is $S$ GB and that each entry in the DRT in our implementation occupies $6 * 4$ B, the maximal number of records in the *DRT* then is $S/4 * 10^6$. Therefore, the maximal meta-data space overhead is 0.6% of the replica space, an acceptable requirement for a storage cluster.

## VI. Related work

Modern PFSs often support three layout policies: 1-DH, 1-DV, and 2-D [13], each being suitable for a certain type of access patterns. For more complex requests, sophisticated optimization methods are developed, such as data partition [23] and data replication [17], [32]. Facing data query workloads, PARLO deploys multiple data layout optimizations to speed up I/O performance [33]. Tantisiriroj *et al.* [34] use HDFS-specific layout rearrangement to enhance PVFS performance. PLFS [35] redirects parallel write requests to a set of reorganized log-formatted files. Although PLFS reorders file requests like our work, it does not further optimize the data layout of the reordered files. The most related work is using adaptive file stripe sizes to boost I/O performance [10], [14]. However, all these studies are designed for homogeneous HDD-based I/O systems. In contrast, this work aims to improve the data layout of a heterogeneous I/O system.

Due to attractive features, SSDs are widely deployed into parallel I/O systems, either as a cache tier or as a storage tier. Unlike these single-sever layout optimizations, our study focuses on the multi-server data layout in a parallel I/O system. S4D-Cache [4] investigates the data placement in a HDD-based parallel file system with a temporary cache on a small number of SSD servers. CARL [36] uses both HDD servers and SSD servers as persistent storage, and it places file regions with high access costs only on SSD servers. However, this may compromise I/O performance because I/O parallelism on all servers may not be fully utilized. Our current work, MHA, can do this because of its adaptive data distribution.

In a hybrid PFS that treats SSD servers as ordinary storage devices, PADP [16] and PSA [15] employ variable-sized file stripe to improve I/O performance. Similarly, HAS [5] selects a best-fitting data layout from three typical candidates to reduce I/O access costs. These three approaches target heterogeneous servers and homogeneous access patterns, however MHA targets for both heterogeneous servers and heterogeneous access patterns. Perhaps the closest related work is our previous work on HARL [8]. While HARL addresses the same problem as MHA, it is limited to optimize data layout based on the

original data order in a parallel file. In contrast, MHA targets data layout optimization via data grouping and data reordering.

## VII. Conclusions

In this study, we have proposed MHA, a migratory heterogeneity-aware data layout scheme to improve the performance of hybrid parallel I/O system. MHA first migrates file data with similar access patterns together, and then optimizes the appropriate stripe sizes for heterogeneous servers based on their respective performance. We have implemented MHA under MPI-IO library on top of OrangeFS file system. Experimental results show that compared to existing approaches, the pre-processed data reordering and the adaptive file stripe sizes can significantly improve I/O system performance.

As future work, we plan to evaluate MHA in a much larger cluster, which is not currently available to us. We also intend to develop dynamic approaches to further improve the performance of those applications with unpredictable patterns.

## Acknowledgments

The authors are grateful to the anonymous reviewers for their valuable comments and suggestions. This work was supported by the National Science Foundation of China under Grant No. 61572377 and 61672513.

## References

[1] "Orange File System," http://www.orangefs.org/.

[2] S. Microsystems, "Lustre File System: High-performance Storage Architecture and Scalable Cluster File System," Tech. Rep. Lustre File System White Paper, 2007.

[3] F. Schmuck and R. Haskin, "GPFS: A shared-disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002, pp. 231–244.

[4] S. He, X.-H. Sun, and B. Feng, "S4D-Cache: Smart Selective SSD Cache for Parallel I/O Systems," in *Proceedings of the International Conference on Distributed Computing Systems*, 2014, pp. 514 – 523.

[5] S. He, X.-H. Sun, and A. Haider, "HAS: Heterogeneity-Aware Selective Data Layout Scheme for Parallel File Systems on Hybrid Servers," in *Proceedings of 29th IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 613–622.

[6] A. Caulfield, L. Grupp, and S. Swanson, "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications," in *Proceedings of the Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[7] M. Zhu, G. Li, L. Ruan, K. Xie, and L. Xiao, "HySF: A Striped File Assignment Strategy for Parallel File System with Hybrid Storage," in *Proceedings of the IEEE International Conference on Embedded and Ubiquitous Computing*, 2013, pp. 511–517.

[8] S. He, X.-H. Sun, Y. Wang, A. Kougkas, and A. Haider, "A Heterogeneity-Aware Region-Level Data Layout Scheme for Hybrid Parallel File Systems," in *Proceedings of the 44th International Conference on Parallel Processing*, 2015.

[9] P. H. Carns, I. Walter B. Ligon, R. B. Ross, and R. Thakur, "PVFS: A Parallel Virtual File System for Linux Clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000, pp. 317–327.

[10] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "A Segment-Level Adaptive Data Layout Scheme for Improved Load Balance in Parallel File Systems," in *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2011, pp. 414–423.

[11] H. Tang, S. Byna, S. Harenberg, X. Zou, W. Zhang, K. Wu, B. Dong, O. Rubel, K. Bouchard, S. Klasky, and N. F. Samatova, "Usage pattern-driven dynamic data layout reorganization," in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 16-19 May 2016 2016, pp. 356–365.

[12] M. Uysal, A. Acharya, and J. Saltz, "Requirements of I/O Systems for Parallel Machines: An Application-Driven Study," College Park, MD, USA, Tech. Rep., 1997.

[13] H. Song, Y. Yin, Y. Chen, and X.-H. Sun, "A Cost-Intelligent Application-Specific Data Layout Scheme for Parallel File Systems," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, 2011, pp. 37–48.

[14] H. Song, H. Jin, J. He, X.-H. Sun, and R. Thakur, "A Server-Level Adaptive Data Layout Strategy for Parallel File Systems," in *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops and PhD Forum*, 2012, pp. 2095–2103.

[15] S. He, Y. Liu, and X.-H. Sun, "A Performance and Space-Aware Data Layout Scheme for Hybrid Parallel File Systems," in *Proceedings of the Data Intensive Scalable Computing Systems Workshop*, 2014, pp. 41–48.

[16] S. He, X.-H. Sun, B. Feng, and F. Kun, "Performance-aware data placement in hybrid parallel file systems," in *Proceedings of the 14th International Conference on Algorithms and Architectures for Parallel Processing*, 2014, pp. 563–576.

[17] 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 *Proceedings of 27th IEEE International Parallel and Distributed Processing Symposium*, 2013, pp. 345–356.

[18] S. He, Y. Wang, and X.-H. Sun, "Boosting Parallel File System Performance via Heterogeneity-Aware Selective Data Layout," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2492–2505, 2016.

[19] A. N. Lab, "MPICH2 : A High Performance and Widely Portable Implementation of MPI." [Online]. Available: http://www.mcs.anl.gov/research/project-detail.php?id=2

[20] "Application I/O Traces: Anonymous LANL App2," http://institutes.lanl.gov/plfs/maps/, 2014.

[21] "LU Decomposition," 2018. [Online]. Available: http://www.cs.umd.edu/projects/hpsl/mambo/lu.html

[22] "Sparse Cholesky Factorization," 2018. [Online]. Available: http://www.cs.umd.edu/projects/hpsl/mambo/cholesky.html

[23] Y. Wang and D. Kaeli, "Profile-Guided I/O Partitioning," in *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003, pp. 252–260.

[24] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces," in *Proceedings of the 12th USENIX conference on File and Storage Technologies*, 2014, pp. 213–228.

[25] "The NAS parallel benchmarks," www.nas.nasa.gov/publications/npb.html, 2014.

[26] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O Prefetching Using MPI File Caching and I/O Signatures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2008, pp. 1–12.

[27] Y. Yin, S. Byna, H. Song, X.-H. Sun, and R. Thakur, "Boosting Application-Specific Parallel I/O Optimization Using IOSIG," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012, pp. 196–203.

[28] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A K-Means Clustering Algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[29] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley DB," in *Proceedings of the USENIX Annual Technical Conference*, 1999, pp. 183–191.

[30] "Interleaved Or Random (IOR) Benchmarks." [Online]. Available: http://sourceforge.net/projects/ior-sio/

[31] A. Ching, A. Choudhary, W.-k. Liao, L. Ward, and N. Pundit, "Evaluating I/O Characteristics and Methods for Storing Structured Scientific Data," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, 2006.

[32] J. Jenkins, X. Zou, H. Tang, D. Kimpe, R. Ross, and N. F. Samatova, "RADAR: Runtime Asymmetric Data-Access Driven Scientific Data Replication," in *Proceedings of the International Supercomputing Conference*. Springer, 2014, pp. 296–313.

[33] Z. Gong, D. A. B. II, X. Zou, Q. Liu, N. Podhorszki, S. Klasky, X. Ma, and N. F. Samatova, "PARLO: PArallel Run-time Layout Optimization for Scientific Data Explorations with Heterogeneous Access Patterns," in *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013.

[34] W. Tantisiriroj, S. Patil, G. Gibson, S. Seung Woo, S. J. Lang, and R. B. Ross, "On the Duality of Data-Intensive File System Design: Reconciling HDFS and PVFS," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[35] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A Checkpoint Filesystem for Parallel Applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–12.

[36] S. He, X.-H. Sun, B. Feng, X. Huang, and K. Feng, "A Cost-Aware Region-Level Data Placement Scheme for Hybrid Parallel I/O Systems," in *Proceedings of the IEEE International Conference on Cluster Computing*, 2013, pp. 1 – 8.