Pattern-Direct and Layout-Aware Replication Scheme for Parallel I/O Systems

Yanlong Yin*, Jibing Li*, Jun He*, Xian-He Sun*, and Rajeev Thakur†
*Computer Science Department
Illinois Institute of Technology, Chicago, Illinois 60616
Email: {yyin2, jli33, jhe24, sun}@iit.edu
†Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, Illinois 60439
Email: thakur@mcs.anl.gov

***Abstract*—The performance gap between computing power and the I/O system is ever increasing, and in the meantime more and more High Performance Computing (HPC) applications are becoming data intensive. This study describes an I/O data replication scheme, named Pattern-Direct and Layout-Aware (PDLA) data replication scheme, to alleviate this performance gap. The basic idea of PDLA is replicating identified data access pattern, and saving these reorganized replications with optimized data layouts based on access cost analysis. A runtime system is designed and developed to integrate the PDLA replication scheme and existing parallel I/O system; a prototype of PDLA is implemented under the MPICH2 and PVFS2 environments. Experimental results show that PDLA is effective in improving data access performance of parallel I/O systems.**

***Keywords*-Parallel I/O; I/O optimization; data replication; data reorganization; data access pattern**

I. Introduction

During the last several decades, the rapid development of semiconductor technology allowed the processor speed to increase exponentially. Supercomputers are moving from petascale towards exascale in the coming decade. However, the developments of the data input/output (I/O) system and storage devices do not keep pace with that of the computing power. As believed by many, the trend of the biased technology advance will continue in the near future. This unbalanced technology advance leads to the so-called I/O-wall problem.

In the meantime, large-scale scientific applications grow continuously in terms of data access intensity, imposing greater workload on the I/O and storage subsystems. This trend of applications puts even more pressure on already saturated I/O systems. For instance, in astronomy, giant radio telescopes capture observation images continuously, and then the captured data are saved into storage systems. The data analysis applications, such as Montage [1] developed by NASA, then read the data out of storage systems and analyze them. The telescopes may generate data at a rate of many gigabytes to even petabytes per second and the data analysis is both computational intensive and data intensive [2].

Relatively slow storage devices compounded with data intensive applications make I/O system the primary performance bottleneck in many HPC systems. *This drawback motivates this study, which aims to alleviate the I/O bottleneck, especially for data intensive applications.*

I/O is a hot issue in recent years. Many I/O optimization techniques have been developed, such as data sieving [3], List I/O [4], DataType I/O [5], and Collective I/O [3] [6]. Some systems may also integrate new layers/middleware into the parallel I/O software stack. All these layers and optimization techniques make the parallel I/O system exceedingly complex. How to optimize I/O performance is elusive, and the optimization is a complex, error-prone, and time-consuming task, especially for applications with complex I/O behaviors. For example, Zhang's work [7] shows that Collective I/O works well in some cases but not in others. Song's work [8] shows that finding the optimal data layout configuration in PVFS2 can be a daunting task. Their works further confirm our belief that I/O performance is application dependent, and a general I/O system need to be adjustable to different applications [9]. *This raised a must have property of our solution: the I/O optimization should bring the application and system's characteristics into consideration and be adaptive for different applications.*

To achieve the goal of alleviating I/O bottleneck and to satisfy the requirement of the I/O optimization's adaptability, we design and implement the Pattern-Direct and Layout-Aware (PDLA) replication scheme for parallel I/O systems. We design PDLA based on the following facts.

1) Contiguous data access is preferable.

The performance of contiguous data access is higher than that of noncontiguous data access. This stays true for both hard disk drives (HDD) and solid state disks (SSD) [10].

2) Data layout matters.

Data layout in parallel file systems can largely influence the I/O performance. Modern parallel file systems support multiple data layout policies. Users can choose to distribute some data on one single storage node, on a set of nodes, or on all available nodes. The previous work [8] shows that, for applications with different data access patterns, the optimal data layouts are different. The optimal data layout yields the

1530-2075/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPS.2013.114

lowest data access cost, hence the optimal I/O performance.

3) It is valuable and feasible to make use of application's characteristic information for I/O optimizations.

In HPC area, data intensive applications have a large amount of data reads or writes, and they read or write in certain ways. In other words, their I/O behaviors exhibit patterns. For example, due to the iterative loop structures of program codes, some individual or group of data accesses may repeat for many times in one execution [11]. It is feasible to collect and utilize this information, as we did in our previous work [9] [10] [12]. Nevertheless, among different runs, the same patterns can be identified under the same execution environment and configuration.

We shaped the design of PDLA based on the above three facts. 1) PDLA transforms noncontiguous data accesses into contiguous ones. 2) It takes advantage of the application's data access pattern to rearrange data. 3) It distributes replication data with awareness of the physical data layout in parallel file systems.

The PDLA replication scheme includes two major optimizations. In *the "Pattern-Direct" (PD) replication scheme*, the system makes a reorganized data replication for each identified data access patterns of the application. As a result, the logical data in the replication are organized in order with how they are accessed. After determining the reorganization, in *the "Layout-Aware" (LA) replication placement*, the system stores the generated replications in their optimal data layouts in parallel file systems, based on the results of quantitative analysis on data access cost. Once the replications are ready, the I/O system is able to serve future data accesses with the replications.

This study makes the following contributions. (1) We design a Pattern-Direct data replication scheme to reorganize data according to access patterns. (2) To make the storage of replication files layout-aware, we construct a data access cost model for parallel I/O systems to identify the optimal data layout for each replication. (3) To integrate PDLA scheme into existing parallel I/O systems, we design a runtime system that discovers access patterns, creates the replications, and redirects application's requests. We implement a prototype of this runtime system within MPICH2 [13] and PVFS2 [14]. Experimental results show that the PDLA replication scheme is effective in exploiting the full potential of parallel I/O systems.

The rest of this paper is organized as follows. Section II and III describe the key designs of the Pattern-Direct replication scheme and Layout-Aware replication placement respectively. Section IV describes the runtime system design and its implementation. Section V presents the evaluation results. Section VI discusses some issues related to write optimization and presents some related experimental results. Section VII reviews the related work in data replication and data organization. Section VIII concludes this paper.

II. Pattern-Direct Replication Scheme

The success of the Pattern-Direct replication scheme relies on solving two technical issues: obtaining data access pattern information and optimizing data replications using pattern information.

A. Data access patterns

We describe a data access pattern from five key aspects: 1) spatial locality, 2) size of accesses, 3) temporal information, 4) iterative behavior, and 5) I/O operations. The spatial locality can be contiguous, noncontiguous, and the combinations of contiguous and noncontiguous patterns. The noncontiguous patterns are further divided based on byte order distance between successive accesses. Some applications access data just once, whereas some access the same data in the same order multiple times. This can be described with repetitive behavior, which occurs often in iterative loop codes. Request size is crucial and plays a significant role in striping factor, stripe size, and the number of requests. We classify temporal behavior based on intervals between accesses, which can be fixed or random. I/O operations are divided into read only, write only, and read and write.

The procedure of obtaining access patterns includes two steps. The first step is to trace the I/O operations of the underlying application during its execution into trace files. The second step is to perform the offline analysis on trace files and obtain the results, namely, data access patterns. The related implementation details are described in Section IV.

We define two types of data access patterns, local and global ones. A local data access pattern represents the information of a single process's access patterns. By co-analyzing the local I/O access patterns of the underlying application, we are able to acquire some global data access patterns that represent the I/O behavior of the entire application. For example, when an application conducts a series of collective I/O operations in which all the processes participate, the operations are recorded in every single process's trace file. In the global view, these operations are no longer separated behaviors of each process; instead, they are collectively providing the global behaviors of the application. In many situations, local patterns cannot provide the true story of the application, and a global view is necessary to optimize I/O performance.

For more details about the definition and representation of access patterns, please refer to our previous work [9] [12].

B. Pattern-Direct replication policy

1) Replication creation policy: Each replication contains "a data object" instead of an entire original file in the file system. More specifically, it contains the data accessed in one data access patterns. Also, data in a replication are reorganized in the access order according to a corresponding access pattern. Each replication is stored as a new file in the same file system.

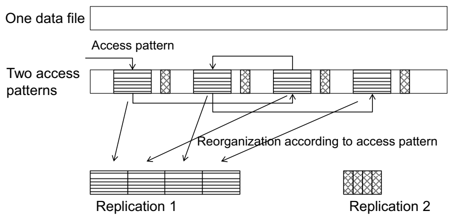

Figure 1. Two access patterns on one data file. PDLA replication scheme creates one replication for each access pattern.

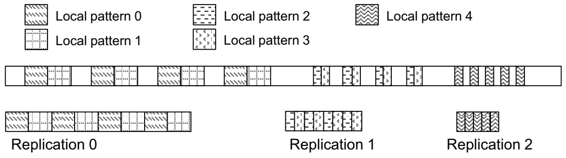

Figure 2. Local patterns are combined into global patterns.

As shown in Figure 1, Pattern-Direct replication scheme only replicates the accessed data. The data that are not accessed or do not fall into any data access pattern, will not be replicated, and only exist in the original data file. The Pattern-Direct replication scheme, comparing with the trivial data replication, yields more efficient uses of storage space.

In PDLA replication scheme, we set the number of replications to be one. The parameter "number of replications" plays a pivotal role in modern distributed file systems. For example, in HDFS [15], this parameter is 3 by default for high data availability and flexible data locality. PDLA's one-replication policy relies on several considerations. First, keeping one replication for each data object, is as good as keeping $n (n > 1)$ replications in terms of I/O performance. Because, in the HPC area, it is rare that multiple processes read the same data at the same time. Even when that happens, the application usually adopts collective I/O, which means only few processes read the required data from storage, and then exchange data among all relative processes on the client side. Second, the one-replication policy is more efficient in terms of storage space consumption. Third, this policy makes it simple to maintain the data consistency between the replications and the original data.

As mentioned in Section II-A, the system may obtain various data access patterns for a given application, including both local and global patterns. The system first makes replications for all the global data access patterns, and then makes replications for those local patterns that do not belong to any global pattern. In this way, we reduce the number of data replications and retain the flexibility of data layout optimization. As illustrated in Figure 2, local patterns 0 and 1 are combined, thus their data are in the same file – Replication 0. Local patterns 2 and 3 are also combined. Local pattern 4 does not belong to any global pattern, and its related data form an independent replication.

2) When to create replications: By default, the scheme creates the replications offline, which means the creation does not occur simultaneously with the application running and accessing the original data. During the first execution of the application, data access patterns are identified and added into the I/O system's pattern database. After that, the system starts to make replications based on the queue of newly added patterns. This procedure may work only when there are unused computing resources and I/O bandwidth, to make replications without affliction to the execution of normal tasks. In our future work, the system may allow users to submit pattern items to the queue by manipulating the pattern database, in case that the future runs of the application will be accessing other data with the same patterns.

3) Where to store replications: Replication files lie on the same parallel file system used by the supercomputer where the applications run. PDLA replication scheme works automatically and hides all the details; replication files are only visible to the I/O middleware that redirects the requests from original files to replications. All the replications are placed in some specially named directories, and any naming rule would work as long as the system's metadata keeps the replications' file paths. Placing these replication files in separated directories also brings convenience for the Layout-Aware replication placement, because modern parallel file systems allow users to control the data layout by setting the attributes of the directories.

Admittedly, the pattern-direct data replication strategy consumes some amount of storage space, like almost all other replication schemes. This is a trade-off between data access performance and storage capacity. But, for many applications this is a good trade-off from energy saving point-of-view. Reducing data access time will reduce energy consumption. In addition, since replications are the small portion active data of the original data, the original data then could be stored on slow spin disks or even on tapes. For this kind of applications, the trade-off of space becomes blurry and the gains in I/O performance and energy consumption become obvious. We will not explore energy saving in this study, but focus on I/O optimization.

III. Layout-Aware Replication Placement

A. Data layout in parallel file systems

We category three most popularly adopted data layout methods as: one-dimensional horizontal (1-DH), one-dimensional vertical (1-DV), and two-dimensional (2-D) data layouts. As shown in Figure 3, 1-DH data layout is the simple striping method where each process distributes its data across all available storage nodes. 1-DV data layout refers to the policy that data to be accessed by each I/O client

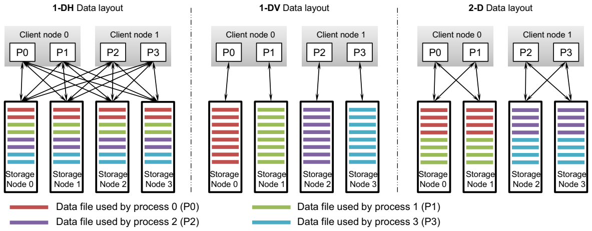

Figure 3. Three data layouts in parallel file systems.

process are stored on one storage node. 2-D data layout refers to the policy in which data to be accessed by each process are stored on a subset (storage group) of storage nodes.

Among these three layout policies, 1-DH data layout is the most widely used one, as it can provide acceptable I/O performance for many situations. In PVFS2, it is the default data layout method namely "simple striping." However, in some cases, it yields poor performance. For example, when the number of processes is much larger than the number of storage nodes, each storage node has to serve requests from all processes, and these requests compete for shared disks. As a result, the disks work in an interleaving way and each request can finish only when all the sub-requests on all nodes finish [16]. Consequentially, each request suffers a large latency hence a high access cost. In fact, for this example case, 1-DV data layout yields higher I/O performance than 1-DH does. This example shows that the number of storage nodes is not the only parameter affecting I/O performance. The number of processes, the request size, and other parameters also play critical roles. Therefore, finding the optimal data layout configuration in PVFS2 can be a daunting task.

Replications created by Pattern-Direct replication scheme are logical files before getting stored. While storing them into parallel file systems, the Layout-Aware data storage optimization first needs to identify the optimal data layouts for them.

B. Optimal data layout selection based on access cost analysis

To identify the optimal data layout, we built a mathematical model of data access cost with consideration of all the critical parameters in the computing environment. The cost model is listed in Table I and the parameters it uses as input are listed in Table II. More details about constructing the cost model can be found in our previous research [8].

The Layout-Aware replication placement works as follows. Given a request and its associated runtime information, the cost model is able to estimate the time cost of fulfilling this request under each of the three data layouts. Given a data access pattern, the cost model estimates the access cost for each request included in the pattern. Then it adds all the requests' costs together to get the access cost for the entire pattern. Also, for each data layout, the model generates a cost result. Naturally, the data layout that produces the lowest cost is the optimal selection, and the scheme will adopt this optimal layout in the parallel file system for the corresponding replication.

Table I
COST MODEL FORMULAS FOR THREE DATA LAYOUT POLICIES.

Data layout	Access cost
1-DV	$max(m, \lceil \frac{p}{n} \rceil) * (e + sv) + \lceil \frac{p}{n} \rceil * (a + sb)$
1-DH	$max(p, mn) * e + max(\frac{p}{n}, m) * sv + pa + \frac{p}{n} * sb$
2-D	$max(m\lceil \frac{p}{n} \rceil, \lceil \frac{p}{g} \rceil) * e + max(m, \frac{\lceil \frac{p}{g} \rceil}{\lceil \frac{n}{g} \rceil}) * sv + a\lceil \frac{p}{g} \rceil + \frac{\lceil \frac{p}{g} \rceil}{\lceil \frac{n}{g} \rceil} * sb$

Table II
PARAMETERS IN THE COST ANALYSIS MODEL.

Parameters	Description
p	Number of I/O client processes.
n	Number of storage nodes.
m	Number of processes on one I/O client node.
s	Data size of one access.
e	Cost of single network connection establishing.
v	Network transmission rate.
a	Startup time of one disk I/O operation.
b	Cost of reading/writing one unit of data.
g	Number of storage groups in 2-D layout.

Some brief guidelines can be derived from the model. 1) When the number of processes p is much smaller than the number of storage nodes n, the cost of 1-DH layout policy is the lowest among all three policies. 2) When $p \approx n$, 2-D layout policy produces higher bandwidth than the other two. 3) When $p > n$, 1-DV layout policy would be the best choice.

IV. THE RUNTIME SYSTEM AND ITS IMPLEMENTATION

Figure 4 shows the system design of the PDLA replication scheme, which consists three phases in chronicle order. In the first phase, during the application's execution, the pattern recognition module identifies and saves the data access patterns. In the second phase, the system creates replications directly according to the recognized data access patterns. Compared with the original data files, the replication files represent the data in the order of the data are accessed. In the third phase, the system automatically forwards I/O requests in the later runs of the same application to the replication files for better performance.

We implement a prototype of the PDLA replication scheme and its runtime system under MPICH2 [13] and PVFS2 [14]. The implementation adds some components into the default parallel I/O system, as illustrated in Figure

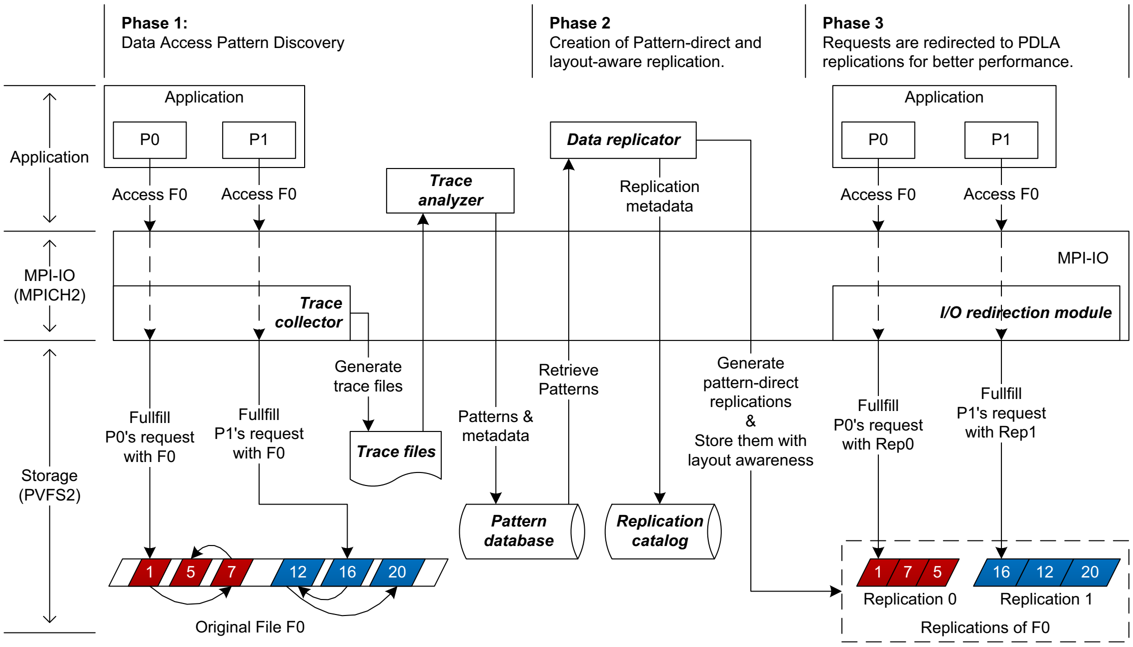

Figure 5. The architecture of the PDLA data replication scheme.

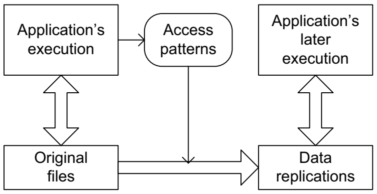

Figure 4. System design overview.

5. Implementation details of each component are explained hereinafter.

A. Trace collector

The trace collector simply traces MPI-IO calls defined in the MPI standard. It gathers information of all standard MPI-IO file operations, such as file open/close, file read/write, and seek. The MPI-IO file operations can be blocking or non-blocking, and collective or non-collective. The trace collector captures the I/O operation parameters by using the Profiling MPI interface (PMPI). The Profiling MPI interface reroutes MPI calls to user-defined instrumentations wrapped around the actual MPI calls. The trace collector is implemented as a library, which can be linked to any application we want to trace. Other than this linking step, there is no need for programmer intervention.

During execution, each process of the application linked to the trace collection library generates one trace file that contains all its I/O operations. For each file operation, the trace collector gathers the following information: 1) MPI rank and process ID; 2) a unique file ID; 3) file offset and request size; 4) name of the I/O operations, such as MPI_File_read_at; 5) the starting time of the operation; and 6) the operation's ending time. The trace collector also records the mapping between the unique file ID and the file path.

B. Trace analyzer

Trace analyzer performs offline trace analysis and utilizes the "template matching" approach to recognize data access patterns from trace files. To some extent, one trace file is a list of file operation records, and each record contains an operation's data access information. The analyzer uses a cursor to mark its analysis progress in the trace. It starts from the first record and moves the cursor forward to scan all records until reaching the end. During scanning, the analyzer always picks a predefined access pattern as a template, to check whether it matches the records around the cursor. Once a match is found, the cursor moves forward along with the same pattern in the trace, until the match does not hold. If there is no match for the first template, the analyzer switches to other templates and tries again. If the analyzer fails to find a match for all templates, it skips the current record, moves the cursor forward, and starts over the matching at the new position. The analyzer produces all the local patterns by analyzing each trace file. After that, it examines all the local patterns and combines the relative ones into global patterns. Trace analyzer inserts obtained patterns into Pattern Database.

C. Pattern database

Both the data replicator and the request redirection module in MPI-IO need to retrieve an application's data access patterns. We keep these metadata in "Pattern Database". It saves the mapping relation between applications and their data access patterns, including: 1) which application a pattern belongs to; 2) which file a pattern depends on; 3) the rank of a process that a pattern belongs to; and 4) which local access pattern is included in a global access pattern. Pattern Database also saves the metadata on the runtime environment of the owner application, including mainly the parameters of the system that will be used to determine the optimal data layout for the generated replication files.

We use Berkeley DB to implement the pattern database. The Berkeley DB is configured as a hash table, and each record is a key-value pair. We generate "patternID" by encoding the following information together: application's execution command, number of process, rank of the process, and the original file name. Each record in the Berkeley DB hash table is a key-value pair; the key is the patternID and the value contains the data access pattern and the runtime information. The value's presentation in the code is a structure definition (in C language) that includes several fixed member variables and a union (also in C language) of various type of data access patterns.

D. Data replicator

The data replicator is a lightweight daemon program that runs in the background. It monitors a queue that contains all the data access patterns that need to be replicated on. When the trace analyzer inserts a new access pattern in to the pattern database it also en-queues the same pattern into this global queue.

When the queue is not empty, the replicator de-queues data access patterns and starts to make replications according to them, one at a time. In the meantime, the data replicator uses the runtime information and the cost-based data layout model to find the optimal data layout configuration. Then it just reads data from the original file and writes them into the replication file placed in the PFS with the optimal data layout. We implement such a queue using Berkeley DB's built-in queue access mode. To configure the data layout the data replicator just sets up a directory with the optimal data layout configuration in PVFS2, and then stores the replication into that directory. PVFS2 provides a tool *pvfs2-xattr* for configuring a directory's data layout policy.

The data replicator also works as the replication scanner. It periodically scans the pattern database, and whenever it finds that a pattern's original file is missing, it removes the corresponding data replication and related metadata.

E. Replication catalog

The replication catalog is used to store metadata for replications. It manages metadata about the relationships among data replications, original files, and the data access patterns, including: 1) which original file a replication's data comes from, and 2) based on which access pattern a replication is created. Its implementation also uses Berkeley DB configured as a hash table; the key is the patternID (the same key in Pattern Database) and the value is the path to the replication file based on the corresponding data access pattern.

F. I/O redirection module in MPI-IO

I/O redirection module redirects data accesses on the original files to the replications. Usually an application issues a data request with three parameters: the identifier of the original file, the data offset, and the request size. After locating the replication file according to these three I/O parameters, runtime information, and the metadata in replication catalog, the redirection module translates the filename and offset between original file and the replication and fulfills the request using the replication.

We have made the following modifications to MPI-IO standard functions to implement the translation.

MPI_File_open: While opening a file, instead of opening the original file, the method tries to open the corresponding replication file.

MPI_File_read/MPI_File_write (and other formats of read/write, such as MPI_File_read_at, etc.): For each I/O read or write, this method uses the file handle of the replication file and checks whether the access pattern has changed or whether the opened file contains the requested content. If the application is still following the same pattern, the module calculates the correct data offset, and issues the data request using the new offset and the input file handle. If the pattern has changed, the module finds new patterns, opens new replication files, and issues request to them.

MPI_File_close: It closes the opened replication file.

MPI_File_seek: It calculates the offset and conducts the seek operation in the replication if necessary.

When the requested data do not belong to any data access pattern and do not have replications, this system will act as the same as the default MPI-IO implementation.

V. Evaluation

The experiments were conducted on a 65-node Sun Fire Linux based cluster, including one head node and 64 computing nodes. The head node was Sun Fire X4240, equipped with dual 2.7 GHz Opteron quad-core processors, 8GB memory, and 12 500GB 7200RPM SATA-II drives configured as RAID5 disk array. Each computing node has two Opteron quad-core processors, 8GB memory and a 250GB 7200RPM SATA-II disk (HDD). We employ 8 nodes as storage nodes managed by PVFS2 and all the

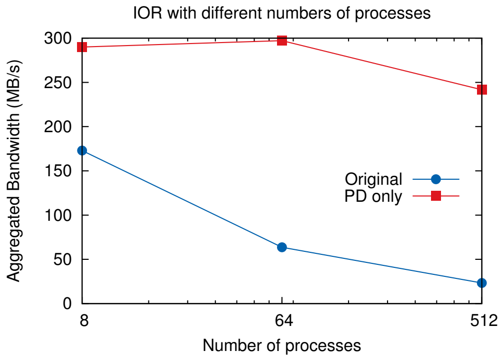

Figure 6. IOR performance improvements with various numbers of processes after enabling PD replication scheme.

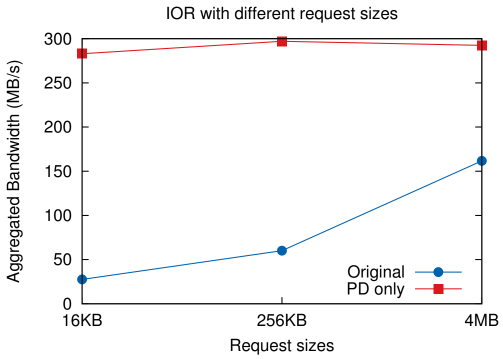

Figure 7. IOR performance improvements with various request sizes after enabling PD replication scheme.

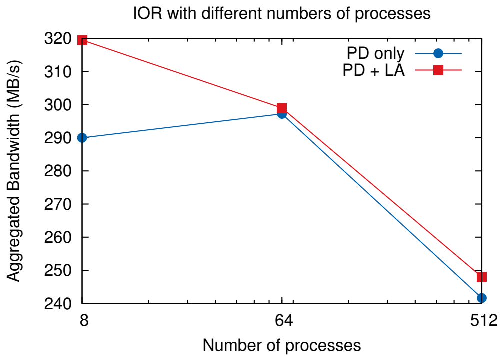

Figure 8. IOR performance improvements with various numbers of processes after further enabling LA replication placement.

other nodes work as client nodes. The head node is used for management, and there was no overlap between file servers and I/O client nodes, so that all the data accesses between the application and the file system are remote accesses.

A. In-depth evaluation with IOR benchmark

1) Evaluation on Pattern-Direct replication scheme: accessing replications versus accessing original data: The evaluation in this subsection is to show the effectiveness of the Pattern-Direct data replication scheme in improving I/O performance. To ensure that the improvements only come from applying the Pattern-Direct replication, for our testing, we disabled the Layout-Aware placement in these tests. Therefore, the replication and original files are using the same data layout. The layout is 1-DH data layout (simple-stripe distribution in PVFS2) with stripe size of 64KB.

First we vary the number of processes. We run IOR benchmark with 8, 64, and 512 processes. Each process accesses 100MB of data in a fixed-stride data access pattern, and the request size is 256KB. Different processes access different regions of the original file so that no process's data co-locate with any other's data. Figure 6 shows the results of this test. We can see that, the overall bandwidth is improved by 67% to 935%. With the number of processes getting larger, IOR's bandwidth gets lower because each storage node needs to serve more processes' request and the competition among processes gets more severe. Figure 6 also shows another improvement brought by PD replication: when the process number increase, the rate of bandwidth degrading is much lower with PD than that with the original case. In other words, the I/O system's scalability on serving more concurrent requests has been significantly improved.

We also vary the request size of IOR. We run IOR with request size of 16KB, 256KB, and 4MB. The number of processes is fixed to 64. Figure 7 shows the results of this test. Similar to the previous test, the bandwidth is improved by 80% to 926%. With the request size getting smaller, IOR's bandwidth gets lower because each storage node needs to handle larger number of small non-contiguous data requests thus the disk seekings get more frequent. Figure 7 also reveals another improvement by applying PD replication: when the request size decreases, the rate of bandwidth degrading is much lower with PD than that with the original case. In other words, the I/O system's ability to handle large number of small requests has been significantly improved.

2) Evaluation on Layout-Aware replication placement: storing replications with layout awareness versus without: We conduct experiments to show that the data layout can further improve performance, which verifies the need to optimize data layouts of replications. We store the replications in two different ways, one set of them are stored in PVFS2's default data layout, and the other set of replications are stored in the optimal data layout calculated by data replication using the cost model presented in Section III-B. These two sets of replications are identical with each other, so all the performance differences are the result from the differences of their data layouts.

In this test, we use IOR benchmark with the data access pattern of fixed-stride data access pattern. We vary the number of processes. We run IOR benchmark with 8, 64,

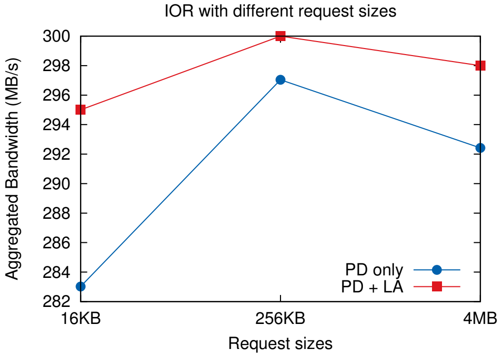

Figure 9. IOR performance improvements with various request sizes after further enabling LA replication placement.

and 512 processes. The corresponding results are shown in Figure 8. We also vary the request size of IOR. We run IOR with request size of 16KB, 256KB, and 4MB. The corresponding results are shown in Figure 9. The results shows that the LA replication placement produces extra 0.5% to 10% performance improvements based on the performance that is already significantly improved by PD replication scheme. The overall performance is improved by 84% to 970% with applying both PD and LA.

B. Evaluation on overall performance improvement with PIO-Bench and MPI-Tile-IO Benchmark

The above in-depth evaluation with IOR already demonstrates the effectiveness of PDLA in improving I/O performance. The overall performance improvement with IOR is 84% to 970%. To convince the IOR testing is representative, we have extended the evaluation to PIO-Bench and MPI-Tile-IO benchmarks.

We run PIO-Bench with a nested-stride access pattern and MPI-Tile-IO with its default access pattern. MPI-Tile-IO treated the entire data file as a 2-D matrix and divides it into $n \times n$ tiles (n rows by n columns). Given n^2 processes, each process accesses the data in one tile, with fixed-stride access pattern. The data of n tiles in the same row are nested together. Therefore, MPI-Tile-IO's data access pattern is also nested-stride.

In this test, we run both benchmarks with 64 processes and various request sizes. The request sizes are 1KB, 4KB, 16KB, 64KB, 256KB, and 1MB. The data layout for the original data files is 1-DH with the default 64KB stripe size. We record each program's execution and use it to divide the total data access size to get the aggregated bandwidth. Figure 10 shows the performance improvements that PDLA brings to MPI-Tile-IO. The aggregated bandwidth increases by 36% to 115%. Figure 11 shows that, for PIO-Bench, the I/O performance improvement is 10% to 98%.

As mentioned above, the data access patterns of both PIO-Bench and MPI-Tile-IO are nested-stride. This means, each process has a fixed-stride access pattern. But multiple local access patterns are nested with each other and can be combined into global access patterns. Therefore, the nested-stride pattern yields better data locality then does the fixed-stride data access pattern that we used for IOR's tests. As a result, the performance improvements of these two benchmarks are not as large as that of IOR, but are still significant. This further confirms the adaptability of PDLA; when the application's data accesses have a poorer performance (due to the poorer data locality among consecutive accesses), it gains more benefit from PDLA.

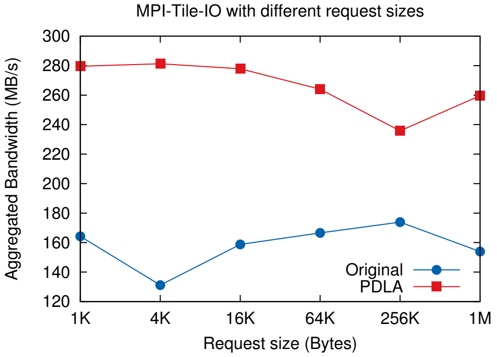

Figure 10. Overall performance improvements with MPI-Tile-IO.

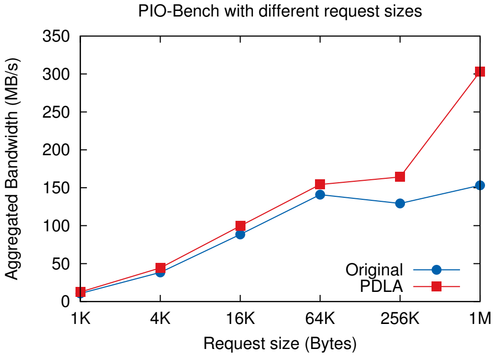

Figure 11. Overall performance improvements with PIO-Bench.

C. System overhead

As showed in Figure 5, we integrate some components into the default parallel I/O system to make the PDLA scheme works automatically. For some application with recognized data access patterns and PDLA replication files, the system is able to improve its overall I/O performance. However, some applications do not have regular data access patterns thus will not benefit from the access to the PDLA replication files. In this case, overhead may exist, which may degrade performance if it is noticeable in volume.

Access pattern recognition and access cost analysis are conducted offline in the background, thus do not affect the

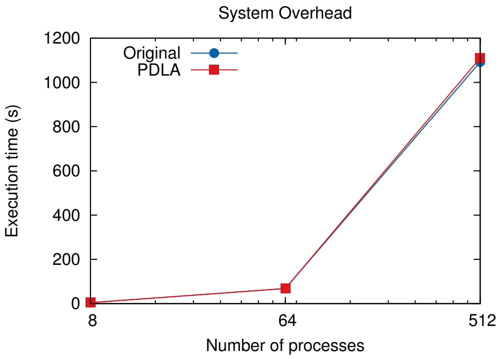

Figure 12. System overhead test results.

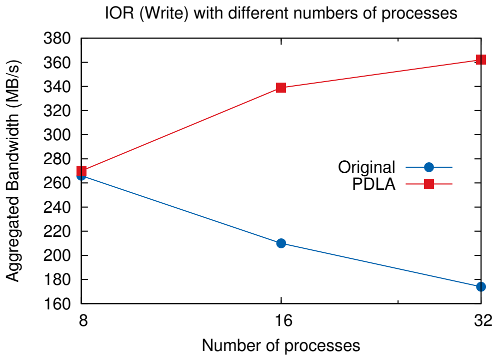

Figure 13. Write performances with various numbers of processes.

execution of user applications. The trace collection only happens once, in the first execution of an application; and the overhead brought by this is illegible as shown in our previous evaluation [9]. Therefore, in this section we only evaluate the following two possible sources of overheads in the runtime system.

1) During "file open", the I/O redirection module needs to look up the data access pattern in the Pattern Database.

2) During "file read/write", the module needs to check whether the opened file is a replication, thus to decide whether to do the offset calculation.

The overhead is very small. To show the overhead is negligible, we run IOR with contiguous data access pattern, and put no related pattern in the database and make no data replication. So the system just accesses the original files. We run IOR with 8, 64, and 512 processes, and each of them 100MB data with the request size of 256KB. Figure 12 shows the results. As expected, the overhead is almost not observable.

VI. Discussion on Write Optimization

At this time, we have fully studied the performance optimization of read operations. Performance optimization for writes is often complex due to data consistence issues. In the meantime, due to the effect of write-buffer, optimized write is not as effective as its read counterpart. For the sake of page limitation, we only present the design and partial evaluation results of write optimization mechanism of PDLA herein.

For one write access pattern, the data replicator first check whether there is a data dependency (data hazard) between this pattern and others. If there is a write after read (WAR) or write after write (WAW) data dependency, no action is taken. If no WAR or WAW data hazard exist, the Data Replicator uses the access pattern to create an empty replication file, and while the application issues the write request, the request redirection model directly calculates the data mapping with the access pattern, and then writes the data into the empty file. The data replicator running in the background then syncs all the data from the replication file back to the original data file.

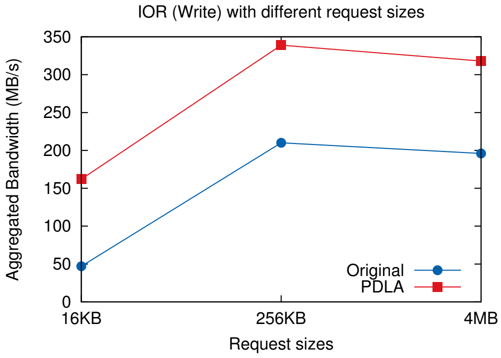

Figure 14. Write performances with various request sizes.

Some experiments are conducted to show that the PDLA scheme is also able to improve the write performance. The experiment setup is similar to the read performance tests. The only difference is to change the I/O operation from read to write. We also use IOR and vary both the number of processes and the data request size.

The results are shown in Figure 13 and Figure 14. The performance improvement is up to 3 times, which is smaller than the improvements in the read tests. The data writes usually will be cached by the storage node's memory first and then flushed back to physical devices. The original performance already benefits from the cached writes. Thus the performance difference between the original system and the PDLA enabled system is not large.

In some cases, the write bandwidth is even larger than the read bandwidth. This is also because of the cached writes. In reads test, we flush the cache of all nodes before each tests, thus no reads can benefit from the cache on either client side or server side.

VII. Related Work

Existing works related to this study fall into two categories: data replication and data organization. Numerous researchers have spent much research efforts on I/O optimizations with these techniques.

A. Data replication

Many servers may simultaneously access different parts of one segment of data in an interleaving way. InterferenceRemoval [17] identifies segments of files that could be involved in this kind of interfering accesses and replicates them to their respectively designated I/O nodes, so some I/O requests can be re-directed to the replicas on other I/O nodes. InterferenceRemoval reduces the degree of interference on each node. PDLA replication scheme creates replications based on data access patterns thus the rule of selecting data to be replicated is straightforward. PDLA also reorganizes data according to the access order in the pattern, so it transforms noncontiguous accesses to contiguous ones.

There are also many I/O optimizations based on data replications in "non-uniform data access" environments. In these "non-uniform access" storage systems, such as HDFS [15] and GPFS-SNC [18], accessing data from different locations yields different costs. Multiple copies of each data file are placed in multiple locations, and the systems [15] [18] [19] [20] always select the "best" (closest on location in terms of access cost) replica for accesses based on storage and network performance predictions provided by an information service inside a cluster or grid. Our PDLA replication scheme focuses on using data replication with awareness of access pattern and layout to optimized I/O performance of a uniform access storage system, such as PVFS2 [14] and Lustre [21].

Some modern parallel file systems, such as Ceph [22], Lustre [21], and GPFS [23], provide built-in data replication functionalities, but these features are mainly designed for the purpose of enhancing the system's fault tolerance or used as data backup service. Similar redundant data placement approaches [24] [25] are designed so that one or few storage devices entering or leaving the storage system does not affect the whole system's data integrity and availability. Instead of being for the purpose of improving the storage system's fault tolerance, the PDLA replication scheme is intended for I/O performance optimization based on data replications, an area that has attracted attention only recently [8].

B. Data organization

AILS [26], FS2 [27], and BORG [28] automatically reorganize selected disk blocks based on the dynamic reference stream to increase effective storage performance by reducing the disk seek distance between requests thus reducing the seeking overhead of each request. These techniques are efficient for single disk and disk arrays but require complex implementation in the disk device driver and local file systems. With a simple implementation in I/O middleware, PDLA replication scheme suits today's large-scale HPC systems well and has better pattern recognition ability.

SOGP [29] is a technique that stores a copy of data that is often accessed in a more efficient organization to improve read performance. It helps PVFS2 [14] use the local storage more efficiently, which bridges the gap between PVFS2 and local storage. PDLA focuses bridging the gap between application and logical data and the gap between logical and physical data to make an integrated optimization crossing these different layers. He et al. proposed a file reorganization method according to access pattern to increase the contiguousness by remapping files in MPI-IO layer [30] [31]. Compared with that, replications generated by PDLA scheme cost less storage resource and are more flexible for further data layout optimization. More importantly, PDLA is a combined system approach with replication, reorganization, and optimized data layout working collectively for best performance.

For write optimization, PLFS [32] [33] stores the writes in a set of efficiently reorganized log-formatted files; the write performance can be dramatically improved, but the performance of read back on those files may not be good due to the inevitable data restructuring. In PDLA replication scheme, read and write access patterns are handled separately to achieve optimal performance for both of them.

To make the data sets generated by HPC applications more accessible to MapReduce-based data analysis applications, MRAP [34] reorganizes the data sets according to data access patterns, during the procedure of copying them from HPC storage to MapReduce system's storage. The PDLA data replication scheme focuses on I/O optimization in general purpose parallel file systems, such as PVFS2.

VIII. Conclusion

We have introduced the Pattern-Direct Layout-Aware (PDLA) replication scheme for I/O optimization based on application specific I/O characteristics. We have refined and combined our previous work in data access pattern identification and cost analysis in designing PDLA and uniquely proposed a system solution for pattern-based replication optimization.

PDLA consists of two key components: Pattern-Direct (PD) replication scheme and Layout-Aware (LA) replication placement. With the "Pattern-Direct replication scheme," the I/O system creates a reorganized data replication each for each identified access pattern of the application. One advantage of PD is that the replication will be accessed in a contiguous way, yielding high I/O performance; another is that it only replicates the accessed data patterns, thus has efficient uses of storage resources. With "Layout-Aware replication placement," the system stores the generated replications in their optimal data layouts based on access cost analysis.

Other contributions of this study are that a runtime system is also designed to integrate PDLA into existing parallel I/O path and that a prototype under the MPICH2 and PVFS2 environment is implemented to evaluate the design. The design consists of six components. They are: 1) trace collector, 2) trace analyzer, 3) pattern database, 4) data replicator, 5) replication catalog, and 6) I/O redirection module.

Experimental results show that the PDLA replication scheme is feasible and effective in improving I/O performance. Given an application with regular data access patterns, the scheme improves the read performance by 10% to 970% of the original performance and improves write performance up to 3 times. The introduced overhead is negligible, even in the worst case where applications have no regular access patterns.

A general assumption of data replication is that replication will lead to space and energy cost. Since PDLA only replicates a small portion active data based on data access pattern, the space and energy trade-off may not be a subject of concern. In fact, with replications of hot data, the faster access on the replications can save some energy, and offloading cold data to slower disk may save more, in terms of energy and space.

In the future work, we plan to continue exploring other ways of utilizing application's I/O characteristics to make the storage system more intelligent and more efficient, and to test more applications from both performance and energy consumption point-of-view.

Acknowledgment

The authors are thankful to and Dr. Robert Ross and Dr. Dries Kimpe of Argonne National Laboratory and Dr. Huaiming Song of Dawning Information Industry for their constructive and thoughtful suggestions toward this work. The authors are also grateful to anonymous reviewers for their valuable comments and suggestions. This research was supported in part by National Science Foundation under NSF grant CCF-0937877 and CNS-1162540, and in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] NASA and the California Institute of Technology, "Montage - Image Mosaic Software for Astronomers," http://montage.ipac.caltech.edu.

[2] Cebit.com.au, "Square Kilometre Array (SKA) Telescope Will Generate Big Data," http://tinyurl.com/8wlzl9o, 9 August 2012.

[3] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[4] A. Ching, A. Choudhary, K. Coloma, W. Liao, R. Ross, and W. Gropp, "Noncontiguous I/O Accesses through MPI-IO," in *Proceedings of IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2003.

[5] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp, "Efficient Structured Data Access in Parallel File Systems," in *Proceedings of IEEE International Conference on Cluster Computing*, 2003.

[6] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, "Server-Directed Collective I/O in Panda," in *Proceedings of IEEE/ACM Supercomputing '95*, 1995.

[7] X. Zhang, S. Jiang, and K. Davis, "Making Resonance a Common Case: A High-performance Implementation of Collective I/O on Parallel File Systems," in *Proceedings of International Parallel and Distributed Processing Symposium*, 2009.

[8] 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.

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

[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 IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2011.

[11] T. Madhyastha and D. Reed, "Exploiting Global Input/Output Access Pattern Classification," in *Proceedings of IEEE/ACM Supercomputing '97*, 1997.

[12] 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 IEEE/ACM Supercomputing '08*, 2008.

[13] Argonne National Laboratory, "MPICH2: High-Performance and Widely Portable MPI," http://www.mpich.org/.

[14] R. Ross, R. Thakur *et al.*, "PVFS: A Parallel File System for Linux Clusters," in *In Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[15] D. Borthakur, "HDFS Architecture Guide," *Hadoop Apache Project*, 2008.

[16] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-Side I/O Coordination for Parallel File Systems," in *Proceedings of IEEE/ACM Supercomputing '11*, 2011.

[17] X. Zhang and S. Jiang, "InterferenceRemoval: Removing Interference of Disk Access for MPI Programs through Data Replication," in *Proceedings of IEEE/ACM Supercomputing '10*, 2010.

[18] K. Gupta, R. Jain, I. Koltsidas, H. Pucha, P. Sarkar, M. Seaman, and D. Subhraveti, "GPFS-SNC: An Enterprise Storage Framework for Virtual-Machine Clouds," *IBM Journal of Research and Development*, 2011.

[19] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, and I. Foster, "Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing," in *Proceedings of Mass Storage Conference*, 2001.

[20] A. Chakrabarti and S. Sengupta, "Scalable and Distributed Mechanisms for Integrated Scheduling and Replication in Data Grids," *Distributed Computing and Networking*, pp. 227–238, 2008.

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

[22] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[23] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the First USENIX Conference on File and Storage Technologies*, 2002.

[24] A. Brinkmann, S. Effert, C. Scheideler *et al.*, "Dynamic and Redundant Data Placement," in *Proceedings of International Conference on Distributed Computing Systems*, 2007.

[25] A. Brinkmann and S. Effert, "Redundant Data Placement Strategies for Cluster Storage Environments," *Principles of Distributed Systems*, pp. 551–554, 2008.

[26] W. Hsu, A. Smith, and H. Young, "The automatic improvement of locality in storage systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 4, pp. 424–473, 2005.

[27] H. Huang, W. Hung, and K. Shin, "FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 263–276, 2005.

[28] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reORGanization for Self-Optimizing Storage Systems," in *Proccedings of the 7th USENIX Conference on File and Storage Technologies*, 2009.

[29] P. Gu, J. Wang, and R. Ross, "Bridging the Gap between Parallel File Systems and Local File Systems: A Case Study with PVFS," in *Processings of International Conference on Parallel Processing*, 2008.

[30] J. He, H. Song, X.-H. Sun, Y. Yin, and R. Thakur, "Pattern-Aware File Reorganization in MPI-IO," in *Proceedings of Parallel Data Storage Workshop*, 2011.

[31] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT press, 1999, vol. 1.

[32] 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 IEEE/ACM Supercomputing '09*, 2009.

[33] M. Polte, J. Lofstead, J. Bent, G. Gibson, S. Klasky *et al.*, "...And eat it too: High read performance in write-optimized HPC I/O middleware file formats," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, 2009.

[34] S. Sehrish, G. Mackey, J. Wang, and J. Bent, "MRAP: A Novel MapReduce-based Framework to Support HPC Analytics Applications with Access Patterns," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010.