Boosting Parallel File System Performance via Heterogeneity-Aware Selective Data Layout

Shuibing He, Yang Wang, and Xian-He Sun, *Fellow, IEEE*

Abstract—Hybrid parallel file systems (PFS) that combine HDD servers with SSD servers provide a promising solution for data intensive applications. The efficiency of a hybrid PFS relies on the data layout schemes. However, most current layout strategies are designed for homogeneous servers, which neither address the heterogeneity of servers nor the varying access patterns of applications. In this paper, we propose HAS, a novel heterogeneity-aware selective data layout scheme for hybrid PFSs. HAS alleviates inter-server load imbalance through skewing data distribution on heterogeneous servers based on their storage performance. Furthermore, to obtain the optimal performance for a specific access pattern, HAS selects one static data layout policy with lowest access cost from three typical layout candidates as the final file data layout method. To adapt to the mixed access patterns within an application, HAS uses a dynamic data layout scheme, which stores file with multiple copies, each using a different data layout policy, and then selects the copy with the lowest access cost to serve file requests. We have implemented HAS within MPICH2 and OrangeFS. Experimental results show that HAS can significantly increase the I/O throughput of hybrid PFSs, compared to existing data layout optimization methods.

Index Terms—Parallel I/O system, parallel file system, data layout, solid state drive

1 Introduction

Parallel file systems (PFS) have been widely used in high-performance computing (HPC) systems during the past few decades. A PFS, such as OrangeFS [1], Lustre [2] and GPFS [3], can achieve superior I/O bandwidth and large storage capacity by accessing multiple file servers simultaneously. However because of the existing performance gap between file servers and CPU, the so called I/O wall, current PFSs cannot fully meet the growing data access requirements of many HPC applications [4], especially for data intensive HPC applications.

NAND flash based solid state disks (SSD) are attracting attention in HPC domains [5]. An SSD is a purely electronic device without mechanical components, thus can provides low access latency, high data bandwidth, lower power consumption, lack of noise, and shock resistance. However, due to the high cost of SSDs and the inherent merits of HDDs (high capacity and decent peak bandwidth for sequential requests), building a large file system solely based on SSDs may be unfeasible for most systems. Therefore, a hybrid PFS, which is comprised of both HDD servers (HServer) and SSD servers (SServer), provides a promising solution for data-intensive applications [6], [7].

- *S. He is with the State Key Laboratory of Software Engineering, Computer School, Wuhan University, Luojiashan, Wuhan 430072, Hubei, China, and the State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha 410073, Hunan, China. E-mail: heshuibing@whu.edu.cn.*
- *Y. Wang is with the Shenzhen Institute of Advanced Technology, Chinese Academy of Science, Xueyuan Avenue 1068, Shenzhen University Town, Shenzhen 518055, China. E-mail: yang.wang1@siat.ac.cn.*
- *X.-H. Sun is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616. E-mail: sun@iit.edu.*

Manuscript received 17 May 2015; revised 11 Nov. 2015; accepted 23 Nov. 2015. Date of publication 3 Dec. 2015; date of current version 10 Aug. 2016. Recommended for acceptance by Y. Lu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2504969

A high-performance hybrid PFS must rely on an efficient data layout, which is an algorithm defining a file's data distribution across available servers. To achieve even data placement, most traditional layout methods utilize a fixed-size stripe to dispatch data across multiple servers. There are three typical such schemes: the one-dimensional horizontal layout, one-dimensional vertical layout, and two-dimensional layout [8], as shown in Fig. 1. To further improve the storage performance, numerous efforts are devoted to the file data layout optimizations, such as data stripe resizing [9], data replication [10], and data reorganization [11]. However, most current schemes are designed and optimized for homogeneous servers. When applied to hybrid PFSs, such schemes have the following three limitations.

First, the heterogeneity of file servers may significantly decrease the overall system performance. Traditional layout schemes usually distribute the same number of file stripes on each server. However, due to their intrinsic properties, SServers almost always outperform HServers [12]. In this case, SServers are easily left idling while HServers continue to process their requests when they concurrently serve a large file request. This inter-server load imbalance leads to underutilization of system hardware resources, which can significantly slows down a request as shown in Section 2.2.

Second, due to the changes of access patterns across different applications, current layout schemes, designed for a specific set of access patterns, are no longer efficient. For example, the commonly used one-dimensional horizontal layout in OrangeFS [1], is only suitable for large parallel file requests, but performs poorly for small requests with a high degree of access concurrency [8]. As access patterns of different applications may vary, in terms of request size, access type (read or write), and access concurrency, a data layout strategy optimized for an application's access pattern is not efficient for other applications.

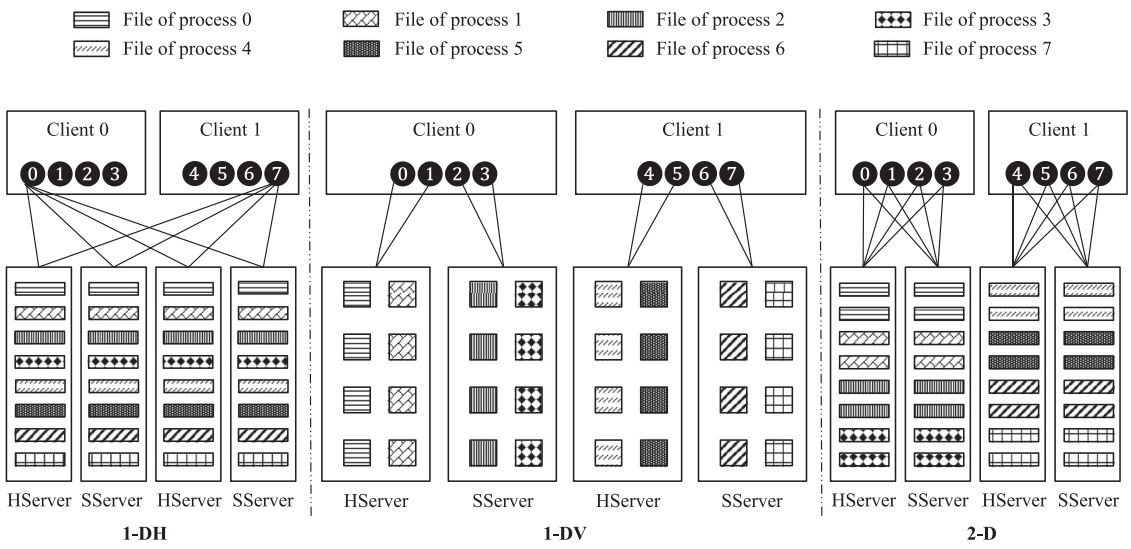


Fig. 1. Three typical data layout policies in PFSs. For 1-DH and 2-D, HServer and SServer are assigned with fixed-size file stripes. For 1-DV, HServer and SServer are distributed with identical number of files.

Third, as applications become more complex, the access patterns within an application can vary considerably, rendering conventional static layout approaches incapable of adapting to the frequently changed access patterns. For example, the application may have a small number of concurrent I/O requests at one moment, but a burst of I/O requests at another, and the requests in each phase have different sizes. In such a case, a static layout approach can be sub-optimal, or even ineffective if an application does not have a dominant pattern. An efficient and comprehensive data layout scheme should depend on the changes of application's access patterns.

In this paper, we propose a heterogeneity-aware selective (HAS) data layout scheme for hybrid PFSs to address the above challenges. HAS eliminates load-imbalance by assigning varied-size file stripes and distributing different number of files to heterogeneous servers based on their performance. In addition, to obtain the optimal performance for a specific access pattern, HAS uses a cost model to select one static data layout policy with lowest access cost from three typical candidates existed in PFSs as the final data layout method. Furthermore, to accommodate the various access patterns within an application, HAS uses a dynamic data layout scheme, which stores file with multiple copies, each using a different data layout policy, and then selects the copy with the lowest cost to serve file requests. Compared with the static layout scheme in our conference version [13], such dynamic replication-based data layout strategy can adaptively serve various access patterns of the application for the best performance.

Specifically, we make the following contributions.

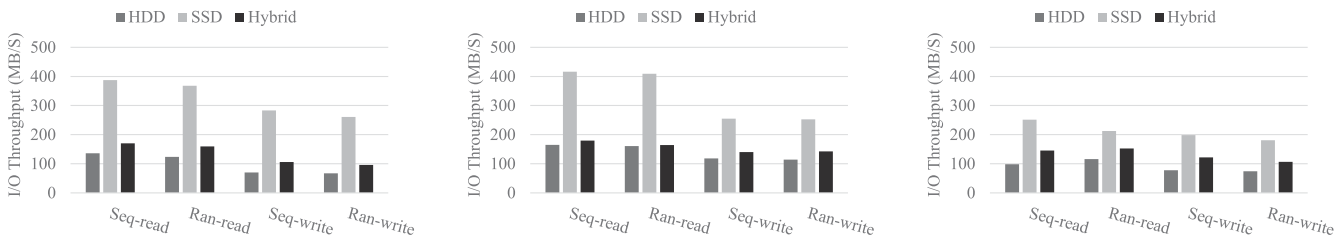
- We introduce a cost model, which is a function of I/O access patterns, file data layout policies, and system configurations, to evaluate the I/O completion time of each file request in a hybrid PFS.
- We propose a selective static data layout scheme for specific access patterns, which distributes file data with the least expensive layout policy determined by the cost analysis. The distribution is implemented either by varying the file stripe sizes or varying the number of files on different servers.
- We present a selective dynamic data layout scheme for various access patterns. This strategy stores file data with multiple copies, each using a different layout policy, and then selects the proper copy with the lowest access cost to serve file requests. This replication-based dynamic strategy can adapt to changed access patterns of a complex application for the best performance.
- We implement the prototype of the HAS scheme under MPICH2 and OrangeFS, and have conducted extensive tests to verify the benefits of the HAS scheme. Experiment results illustrate that HAS can adapt to various access patterns, static and dynamic, and significantly improves I/O performance.

The rest of this paper is organized as follows. The background and motivation are given in Section 2. We describe the design and implementation of HAS in Section 3. Performance evaluations of HAS are presented in Section 4. We introduce the related work in Section 5. Section 6 discusses the applicable spheres of HAS, and Section 7 concludes the paper.

2 Background and Motivation

2.1 Typical Data Layout Policies in PFSs

To obtain optimal performance for different I/O access patterns, PFSs, such as OrangeFS [1], Lustre [2] and GPFS [3], support three typical data layout policies—one-dimensional horizontal (*1–DH*), one-dimensional vertical (*1–DV*), and two-dimensional (*2–D*) layout [8]. As shown in Fig. 1, *1–DH* distributes a process's file across all available servers in a round-robin fashion; *1–DV* performs no striping at all, and instead places the file data on one server; *2–D* is a hybrid method, it distributes the file on a subset of servers. *1–DH* is the most widely used data layout. For example, it is the default layout policy called *"simple striping"* in OrangeFS [1]. All three layout policies utilize fixed-size file stripes to distribute file data, and each of them work well for a particular



(a) 1-DH layout (b) 1-DV layout (c) 2-D layout

Fig. 2. Throughputs of IOR with three typical data layout schemes. The system is tested with homogeneous and heterogeneous server configurations, the file stripe is 64 KB, and the group size is two in 2–D layout.

kind of I/O access patterns [8], [14]. However, these schemes are designed for homogeneous PFSs on identical file servers, and could perform poorly for a hybrid PFS.

2.2 Motivation Example

To illustrate the impact of server heterogeneity on I/O system performance, we ran IOR [15] in a parallel file system OrangeFS under three server configurations: four HServers (denoted by HDD), four SServers (denoted by SSD), and four HServers and four SServers (denoted by Hybrid). IOR ran with 16 processes, each accessing an individual file. The request size was 512 KB, and the access patterns were sequential and random reads and writes. The file stripe size was the default 64 KB.

For each server configuration, we tested IOR performance under three typical layout policies. For *1–DH*, each process's data were distributed on all servers and one server served 16 processes; for *1–DV*, each process's data were distributed on one server and each server served $16/N$ processes' data, where N is the number of servers in the file system; for *2–D*, the group size was 2. Fig. 2 shows the throughputs, which were measured as the aggregated data amount divided by the application's I/O time. We observe that for all policies, the hybrid cluster with eight servers slightly outperforms the homogeneous cluster with four low-speed HServers. Even worse, the hybrid system performs worse than the homogeneous system with four SServers. In other words, more heterogeneous hardware resources actually degrade I/O performance. This result illustrates that traditional layout schemes are highly inefficient for hybrid PFSs.

2.3 Reasons for Poor Performance of Hybrid PFSs

2.3.1 Inter-Server Load Imbalance

The main reason for the poor hybrid PFS performance is that traditional layout schemes may lead to load imbalance among hybrid servers. Generally, a large file request will be divided into multiple sub-requests, concurrently served by the underlying servers. With fixed-size file striping, HServers and SServers possibly handle same-size sub-requests. As SServers are faster storage devices, they will finish the I/O operations quickly and waste much time on waiting for HServers. To verify this, we tested the I/O times of all servers in the sequential read test for Hybrid configuration in Fig. 2a, we find that HServers take roughly 3.5X time to complete I/O operations compared with SServers (Other layouts have the similar imbalance). Because the I/O time of the application depends on the slowest server, this load imbalance will largely offset the overall system performance.

2.3.2 Single Server I/O Inefficiency

Even the inter-server load balance can be maintained, existing layout polices may incur severe I/O inefficiency on a single server, affecting the overall system performance. For example, for requests with a large number of processes, *1–DH* produces much higher I/O concurrency than *1–DV* on each server. In this case, the performance of a single server under *1–DH* can be largely degraded due to more I/O contention from multiple processes. Therefore, a layout policy not considering application access patterns will offset the overall system performance even the inter-server load balance is obtained.

3 Design and Implementation

In this section, we first describe the basic idea of the heterogeneity-aware selective data layout scheme. Then we introduce a cost model to evaluate the data access time in a hybrid PFS. Based on this model, we present the selective data layout schemes for applications with specific access patterns and mixed access patterns respectively. Finally, we give the detailed implementation.

3.1 Basic Idea

Since traditional layout schemes lead to severe performance degradation, the proposed data layout scheme, HAS, aims to optimize the performance of hybrid PFSs through skewing data distribution on heterogeneous servers. Fig. 3 shows the optimized data layout polices of HAS. As opposed to traditional layout policies which assign each server with fixed-size file stripes or fixed-number of files, HAS distributes file data on heterogeneous servers with varied-size file stripes or various number of files based on the server performance. This can alleviate inter-server load imbalance. Further more, HAS selectively chooses the layout scheme with lowest access cost from three typical layout candidates as the final layout for an application with a specific access pattern to improve I/O efficiency. Finally, to adapt to complex applications with mixed access patterns, HAS uses a dynamic data layout strategy, which stores file data with multiple copies, each using a different layout policy, to selectively serve requests with the best fit copy incurring lowest cost.

One might expect to determine the proper skewing data layout for an application's access pattern to be simple. In reality, it is a complex issue for several reasons. First, the server performance can be significantly impacted by request

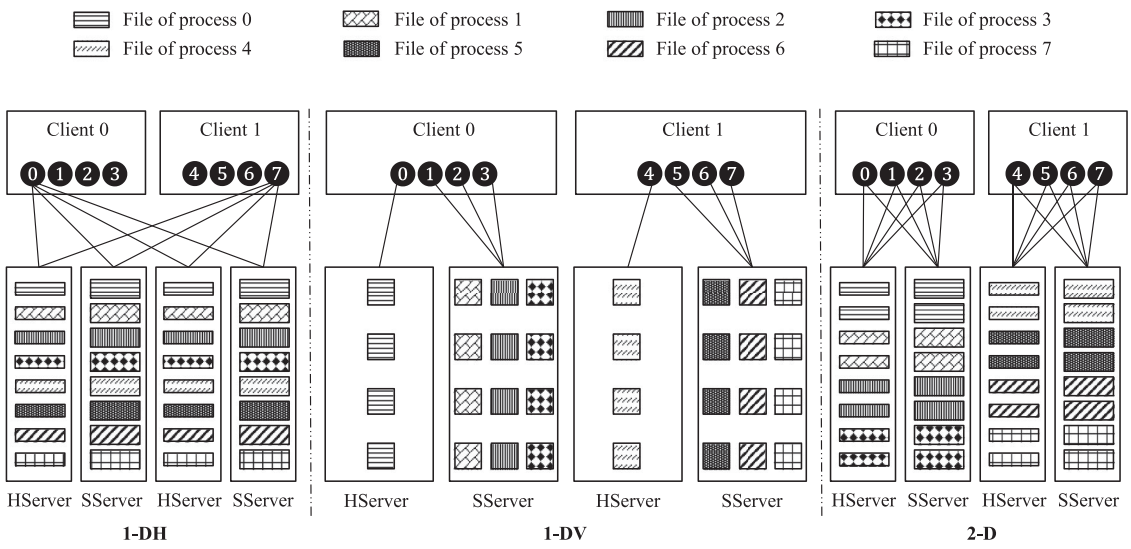


Fig. 3. The three data layout policies in hybrid PFSs after optimization. For 1–DH and 2–D, HServer and SServer are assigned with varied-size file stripes. For 1–DV, HServer and SServer are distributed different number of files.

access patterns, such as request size, access type (read or write), access concurrency (number of processes), etc. Second, the server performance is also related with the storage media. Even under the same pattern, HServer and SServer exhibit different performance behaviors. Finally, besides the storage cost, the network cost, affected by application access patterns, is also an integral part of the overall data access cost. To overcome these issues, we built a cost model accounting for the above factors application, network, and storage characteristics to evaluate the data access time of a request in a heterogeneous I/O system.

3.2 Data Access Model in a Hybrid PFS

3.2.1 Assumptions and Definitions

The overall I/O time is a function of various parameters, which are listed in Table 1. The *system* and *application pattern* parameters are used as known inputs, and *data layout* parameters are optimized depending on the inputs.

Note that the storage related parameters show distinct characteristics on heterogeneous servers. First, the start up time of SServer is much smaller than HServer's. Second, the data transfer time of SServer is several times smaller than that of HServer's. Finally, while HServer may have identical read and write performance, SServer usually has a faster read performance than writes because write operations lead to many background activities due to garbage collection and wear leveling [16].

We only consider *1–DH*, *1–DV*, and *2–D* layout policies due to their popularities [1], [2], [3]. In each policy, the files are distributed on the underlying servers as in Fig. 3. We assume each process accesses one file and only that file. For *1–DH* and *2–D*, stripe sizes are varied depending on the type of server. Note that, for *1–DV*, since all file data is distributed on one server, varying stripe sizes will not affect the request cost and improve the I/O performance. Hence, we vary the number of files on heterogeneous servers instead. Due to symmetry, we assume perfect load balance of data access within HServers and SServers but not between different types of servers. In addition, we have the following assumptions for each policy:

- For *1–DH*, we assume each file request is served by all the $m + n$ servers, so that each storage node can contribute to the aggregate I/O performance. Each sub-request on the server has the same size with the stripe on that server. Thus we have the following constraint:

$$m \times s_h + n \times s_s = r. \tag{1}$$

  Usually s_s is larger than s_h to achieve load balance. In an extreme case, s_h can be zero (which means file data are only distributed on SServers) if there is a possibility to improve performance.
- For *1–DV*, the number of processes on each server equals the number of files. We assume all files are distributed on the $m + n$ servers, thus

TABLE 1
Parameters in Cost Analysis Model

System Parameters	
m	Number of HServers
n	Number of SServers
c	Number of process on one client node
e	Average network establishing time per connection
t	Unit data network transmission time
α_h	Average startup time of one operation on HServer
β_h	Unit data transfer time on HServer
α_{sr}	Average startup time for read on SServer
β_{sr}	Unit data transfer time for read on SServer
α_{sw}	Average startup time for write on SServer
β_{sw}	Unit data transfer time for write on SServer
Application Pattern Parameters	
p	Number of client processes
r	Size of the file request
o	Type of the file request (read or write)
Data Layout Parameters	
s_h	Stripe size on HServer in 1–DH and 2–D layout
s_s	Stripe size on SServer in 1–DH and 2–D layout
p_h	Number of process on HServer in 1–DV layout
p_s	Number of process on SServer in 1–DV layout
g	Number of storage groups in 2–D layout

TABLE 2
Data Access Cost for Read Requests on Hybrid HServers and SServers

Cost	Condition		Network cost T_{NET}		Storage cost T_{STO}
			Establish T_E	Transfer T_X	max{Start T_{SH}+I/O T_{TH}, Start T_{SS}+I/O T_{TS}}
T_{1-DH}	$p \le c(m+n)$		$c(m+n)e$	$\max\{crt, ps_st\}$	$p * \max\{\alpha_h + s_h\beta_h, \alpha_{sr} + s_s\beta_{sr}\}$
	$p > c(m+n)$		pe	$\max\{crt, ps_st\}$	$p * \max\{\alpha_h + s_h\beta_h, \alpha_{sr} + s_s\beta_{sr}\}$
T_{1-DV}	$c \le p_s$		p_se	p_srt	$\max\{p_h(\alpha_h + r\beta_h), p_s(\alpha_{sr} + r\beta_{sr})\}$
	$c > p_s$		ce	crt	$\max\{p_h(\alpha_h + r\beta_h), p_s(\alpha_{sr} + r\beta_{sr})\}$
T_{2-D}	$p \le g$		$c\lceil\frac{p}{g}\rceil e$	crt	$\lceil\frac{p}{g}\rceil * \max\{\alpha_h + s_h\beta_h, \alpha_{sr} + s_s\beta_{sr}\}$
	$p > g$	$\lceil\frac{p}{g}\rceil \le c\lceil\frac{n}{g}\rceil$	$c\lceil\frac{n}{g}\rceil e$	$\max\{crt, \lceil\frac{p}{g}\rceil s_st\}$	$\lceil\frac{p}{g}\rceil * \max\{\alpha_h + s_h\beta_h, \alpha_{sr} + s_s\beta_{sr}\}$
		$\lceil\frac{p}{g}\rceil > c\lceil\frac{n}{g}\rceil$	$\lceil\frac{p}{g}\rceil e$	$\max\{crt, \lceil\frac{p}{g}\rceil s_st\}$	$\lceil\frac{p}{g}\rceil * \max\{\alpha_h + s_h\beta_h, \alpha_{sr} + s_s\beta_{sr}\}$

$$m \times p_h + n \times p_s = p. \quad (2)$$

Similarly, p_s is larger than p_h, and p_h can be zero.

- For *2–D*, we assume each group includes m/g HServers and n/g SServers, and a file request is distributed on all the $(m+n)/g$ servers in that group as in *1–DH* policy, thus

$$m \times s_h + n \times s_s = g \times r. \quad (3)$$

3.2.2 Access Cost Analysis

Our cost model divides the overall I/O time of a parallel data access into four parts. T_E is the network establishing time, T_X is the network transferring time, T_S is the storage startup time, and T_T is the storage transfer time. The former two parts are network related access costs (T_{NET}), and the latter two are storage related access costs (T_{STO}). The cost model is derived from our earlier work [8] and it considers storage heterogeneity. Taking their sum gives the overall I/O time (T):

$$T = T_E + T_X + T_S + T_T. \quad (4)$$

Establishing cost. T_E depends on the number of establishing operations for the parallel data accesses. Since a network establishing operation is related with both the client and the server, T_E is determined by the higher cost of the two. Take the *1–DH* layout as an example, each client needs to establish network connections with all servers serially, thus $T_E = c(m+n)e$. From a server's point of view, it is accessed by p processes, thus $T_E = pe$. Then, the final establishing time $T_E = max\{c(m+n)e, pe\}$.

Transfer cost. T_X is related with the network data transfer size and the network data transfer rate. Similarly, it is determined by the maximal cost of a client and a server. We still use the *1-DH* layout as an example. For a client, $T_X = crt$; for HServer, $T_X = ps_ht$; for SServer, $T_X = ps_st$. Since s_h will always be at most s_s, $T_X = max\{crt, ps_st\}$.

Startup cost. T_S is relatively straightforward and determined by the number of I/O operations on one server, namely the number of client processes assigned on that server. For a parallel request, T_S is determined by the maximal cost among all involved file servers.

Read/write cost. T_T is the time spent on actual data read/write operations. It can be calculated by the ratio of the request size over the data transfer rate of storage devices. Similarly, T_T is determined by the maximal value of all file servers for a parallel request.

We refer to the data access cost in the three layout polices as $T_{1\text{-}DH}$, $T_{1\text{-}DV}$ and $T_{2\text{-}D}$ respectively, which is calculated as in Table 2. $T_{1\text{-}DH}$ is derived from our previous work [17], which expresses the cost as a function of s_h and s_s. For the new proposed formulas, $T_{1\text{-}DV}$ describes the cost as a function of p_h and p_s, and $T_{2\text{-}D}$ utilizes s_h, s_s and g for the same goal. Table 2 only displays the access cost for read requests; writes will be similar except the startup time and the unit data transfer time for SServers will change. These three policies imply more pattern-aware and effective layout optimization methods for a hybrid PFS.

By examining the formulas, we can capture the following implications for data layout optimizations.

- With fixed-size stripes on HServers and SServers, *1–DH* and *2–D* lead to severely server load imbalance, which can significant degrade the overall I/O system performance. With uniform file distribution on HServers and SServers, *1–DV* also incurs poor I/O performance.
- For *1–DH* and 2–D, the storage read/write cost T_{TH} and T_{TS} on the two types of servers can be balanced by increasing the stripe size of SServer (s_s) and decreasing that of HServer (s_h), but doing so may increase the network transfer time T_X on SServer, possibly delaying the overall completion time (T).
- For *1–DV*, the storage read/write cost T_{TH} and T_{TS} can be balanced by increasing the number of files on SServer (p_s) and decreasing that on HServer (p_h). Similarly, this may increase the network transfer time T_X, offsetting the reduction of T.

3.3 Selective Static Data Layout Scheme for Specific Accesses

For a specific access pattern, different layout policies (and the related layout parameters) lead to different access cost. We note that the model consists of linear equalities and inequalities of unknown variables (max can be expressed as

multiple linear inequalities). Therefore, the model can be solved exactly to minimize the total I/O cost under three layout policies, subject to the above constraints. We first show the *local* data layout optimization for each policy, then describe the selective *global* optimization for applications with specific access patterns which do not change in the run-time environment during the application's execution.

3.3.1 1–DH Layout Optimization

The system and pattern related parameters can be regarded as constants, and $T_{1\text{-}DH}$ is a function of two unknowns—s_h and s_s. The final problem is to choose the values of s_h and s_s to minimize $T_{1\text{-}DH}$. For the case where $p \leq c(m+n)$, according to the member values in the corresponding maximum expressions in the formulas in Table 2, we translate this optimization problem into four linear programming (LP) problems shown below

$$\text{Case 1: Minimize } T^1_{1\text{-}DH} = c(m+n)e + crt + p(\alpha_h + s_h\beta_h) \tag{5}$$

$$\text{subject to} \begin{cases} ms_h + ns_s & = r \\ ps_s & \leq cr \\ \alpha_{sr} + s\beta_{sr} & \leq \alpha_h + h\beta_h \\ 0 & \leq s_h \leq r/m \\ 0 & \leq s_s \leq r/n. \end{cases} \tag{6}$$

$$\text{Case 2: Minimize } T^2_{1\text{-}DH} = c(m+n)e + p(s_s t + \alpha_h + s_h\beta_h) \tag{7}$$

$$\text{subject to} \begin{cases} ms_h + ns_s & = r \\ cr & \leq ps_s \\ \alpha_{sr} + s\beta_{sr} & \leq \alpha_h + h\beta_h \\ 0 & \leq s_h \leq r/m \\ 0 & \leq s_s \leq r/n. \end{cases} \tag{8}$$

$$\text{Case 3: Minimize } T^3_{1\text{-}DH} = c(m+n)e + crt + p(\alpha_{sr} + s_s\beta_{sr}) \tag{9}$$

$$\text{subject to} \begin{cases} ms_h + ns_s & = r \\ ps_s & \leq cr \\ \alpha_h + h\beta_h & \leq \alpha_{sr} + s\beta_{sr} \\ 0 & \leq s_h \leq r/m \\ 0 & \leq s_s \leq r/n. \end{cases} \tag{10}$$

$$\text{Case 4: Minimize } T^4_{1\text{-}DH} = c(m+n)e + ps_s(t + \alpha_h + s_h\beta_h) \tag{11}$$

$$\text{subject to} \begin{cases} ms_h + ns_s & = r \\ cr & \leq ps_s \\ \alpha_h + h\beta_h & \leq \alpha_{sr} + s\beta_{sr} \\ 0 & \leq s_h \leq r/m \\ 0 & \leq s_s \leq r/n. \end{cases} \tag{12}$$

Then

$$T_{1\text{-}DH} = min\{T^1_{1\text{-}DH}, T^2_{1\text{-}DH}, T^3_{1\text{-}DH}, T^4_{1\text{-}DH}\}. \tag{13}$$

The first constraint of Equations (6), (8), (10), and (12) is Equation (1). The second constraint accounts for the two possible values of T_X in Table 2. The third shows the possible values of T_{STO}. The fourth and fifth constraints are directly derived from Equation (1). For example, the max of s_h is achieved by letting $p_s = 0$ in Equation (1), as goes for calculating the max of s_s. The final cost for *1DV* determined by Equation (13) is the minimum of the four cases.

The cases for the alternate condition will be similar to the four above, except some values will be interchanged according to the formulas in Table 2.

3.3.2 1–DV Layout Optimization

For $T_{1\text{-}DV}$, it is a function of two unknowns—p_h and p_s. The final problem is to choose the values of p_h and p_s to minimize $T_{1\text{-}DV}$. For condition $c \leq p_s$, according to the formulas in Table 2, we similarly translate the layout optimization problem as as following:

$$\text{Case 1: Minimize } T^1_{1\text{-}DV} = p_s(e + rt) + p_h(\alpha_h + r\beta_h) \tag{14}$$

$$\text{subject to} \begin{cases} mp_h + np_s & = p \\ p_s(\alpha_{sr} + r\beta_{sr}) & \leq p_h(\alpha_h + r\beta_h) \\ 0 & \leq p_h \leq p/m \\ c & \leq p_s \leq p/n. \end{cases} \tag{15}$$

$$\text{Case 2: Minimize } T^2_{1\text{-}DV} = p_s(e + rt) + p_s(\alpha_{sr} + r\beta_{sr}) \tag{16}$$

$$\text{subject to} \begin{cases} mp_h + np_s & = p \\ p_h(\alpha_h + r\beta_h) & \leq p_s(\alpha_{sr} + r\beta_{sr}) \\ 0 & \leq p_h \leq p/m \\ c & \leq p_s \leq p/n. \end{cases} \tag{17}$$

Then

$$T_{1\text{-}DV} = \min\{T^1_{1\text{-}DV}, T^2_{1\text{-}DV}\}. \tag{18}$$

The first constraint of Equations (15) and (17) is Equation (2). The second constraint is the only difference between case 1 and case 2; they account for the two possible values of T_{STO} in Table 2. The third and fourth constraints for *1–DV* are directly derived from Equation (2). For example, the max of p_h is achieved by letting $p_s = 0$ in Equation (2), as goes for calculating the max of p_s. The final cost for *1–DV* determined by Equation (18) is the minimum of the two cases.

The cases for the alternate condition will be similar to the two above, except some values will be interchanged according to the formulas in Table 2.

3.3.3 2–D Layout Optimization

T_{2-D} is a function of three unknown parameters s_h, s_s, and g. Similarly, based on the member values in the maximum expressions in Table 2, we translate the optimization problem into two linear programming problems for the condition $p \leq g$

$$\text{Case 1: Minimize } T^1_{2\text{-}D} = c((p/g)e + rt) + (p/g)(\alpha_{sr} + s_s\beta_{sr}) \tag{19}$$

$$\text{subject to} \begin{cases} ms_h + ns_s & = gr \\ 1 < & g < (m+n) \\ \alpha_h + s_h\beta_h & \leq \alpha_{sr} + s_s\beta_{sr} \\ 0 & \leq s_h \\ 0 & \leq s_s. \end{cases} \tag{20}$$

$$\text{Case 2: Minimize } T^2_{2\text{-}D} = c((p/g)e + rt) + (p/g)(\alpha_h + s_h\beta_h) \tag{21}$$

$$\text{subject to} \begin{cases} ms_h + ns_s & = gr \\ 1 < & g < (m+n) \\ \alpha_{sr} + s_s\beta_{sr} & \leq \alpha_h + s_h\beta_h \\ 0 & \leq s_h \\ 0 & \leq s_s. \end{cases} \tag{22}$$

Then

$$T_{2\text{-}D} = \min\{T^1_{2\text{-}D}, T^2_{2\text{-}D}\}. \tag{23}$$

The first constraint is Equation (3). The second constraint ensures the achieved layout remains *2D*. Similar to the second constraint of Equation (15), the third constraint accounts for the possible values of T_{STO}. The last two constraints for Equations (19) and (21) simply ensure the stripe sizes remain positive. The cases for other conditions will be similar to the two above.

The above optimizations for *1–DH*, *1–DV*, and *2–D* use the storage values for reads, the optimizations for writes can be done similarly. Note that because the above linear program is expressed with only two or three unknown variables, the search space is very small and solving the program requires acceptable time overhead.

3.3.4 Selective Global Data Layout Scheme

After the above *local* linear optimizations, there will be three potential layout methods (*1–DH*, *1–DV*, and *2–D*) for a given access patten, each with their own sub-optimal cost $T_{1\text{-}DH}$, $T_{1\text{-}DV}$, and $T_{2\text{-}D}$. Then, it is nature to select the minimum of three candidates as the most optimal data layout distribution for the given access pattern, which fully accounts for application, storage, and network characteristics. Namely, the global optimal data access cost

$$T_{opt} = \min\{T_{1\text{-}DH}, T_{1\text{-}DV}, T_{2\text{-}D}\}. \tag{24}$$

If we have a prior knowledge of data accesses of an application, we can use it to determine the global optimal data layout from the three candidates. Fortunately, most data-intensive HPC applications access their files with predictable access patterns and they often run multiple times [18], [19], thus I/O behavior can be learned from previous runs and it provides an opportunity to achieve the proposed data layout scheme. By comparing the three access costs, HAS takes one layout manner which requires the lowest cost as the optimal data layout method for that application. The implementation is described in Section 3.5

3.4 Selective Dynamic Data Layout Strategy for Mixed Accesses

In the previous section, we described three typical data layout policies to adapt to diverse data access patterns in an efficient way. However, each of these layouts is designed for a specific data access pattern, and as such, it might not be applied to the case where an application exhibits different I/O behaviors during its execution. For example, the application may have a small number of concurrent I/O requests at one moment, but a burst of I/O requests at another, and the requests in each I/O phase have different sizes. As a result, it is impossible for any static data layout policy to effectively serve all data access patterns.

To address this issue, we propose a dynamic data layout strategy, which leverages the data replication technology [8], to facilitate the I/O accesses with different patterns at the runtime. Each file has several copies with different data layout policies in the parallel file system. For each data access, the strategy dynamically chooses one copy with the minimal access cost to serve the request. Since each file request is assigned to the best fit copy, this replication-based dynamic data layout strategy can serve various kinds of I/O workloads with high performance.

In terms of data replication, the first question is how many replicas (denoted by k) should be created for the application with various access patterns. Ideally, we can minimize the overall I/O cost of the application if we create a corresponding replica for each access pattern with a perfect data layout policy. However, it results in unfeasible space cost. Hence, we account for both storage performance and space when determining a practical k value. For the sake of simplicity, we assume that there are three replicas for each file throughout this paper. Of course, users can choose different value of k depends on their performance and cost trade-offs.

The second issue is how to determine the data layout policy for each replica. Since an application may have many access patterns during it's execution, we can't minimize the overall I/O cost if we create k replicas with layout policies based on randomly chosen access patterns. To address this challenge, we propose an data grouping scheme that classifies file requests of the application into k groups based on I/O trace analysis, and then create one replica with an optimal layout policy for requests with closed access patterns in that group.

The effectiveness of data grouping depends on the grouping criteria. In fact, improper criteria may co-locate requests with different access patterns into the same group, thus deteriorating the effectiveness of grouping. To correctly reflect the access pattern distribution of file requests, we propose an *iterative request grouping* algorithm to determine the grouping criteria. Inspired by the data clustering approach in statistics [20], we divides file requests into k groups and tries to find the centers of these groups with an iterative refinement method.

The detailed description of the algorithm is as Algorithm 1, where each request is characterized by two-element tuple, denoted by $(pro, size)$, pro refers to the number of processes while $size$ is the request size. As such, all requests can be represented by a set of points in a two-dimensional $Euclidean\ Space$. For any point $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$, their distance can be defined as

$$||P_1 - P_2|| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \tag{25}$$

As shown in Algorithm 1, if the number of requests is less than or equal to k, a randomly selected request point is assigned to P_{g_i} as a center of the ith 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 re-compute the new center for each group. This procedure is repeated until P_{g_i} is no longer changed or three times at most.

Algorithm 1. Iterative Request Grouping

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

For the sake of simplicity, we assume that there are three replicas for each file in Algorithm 1 throughout this paper. Although the computational overhead of the algorithm increases in proportion to the number of requests, the request grouping is an off-line method and it only runs once based on I/O trace analysis, so the computation overhead in a practical HPC system is acceptable.

Once the grouping finishes, the dynamic layout scheme will create k replicas for each file. The optimal data layout policy for each replica is determined by the access pattern of one center of the groups, according to the static data layout scheme in previous Section 3.3.4. Since file requests in each group have closed access patterns, they will have high likelihoods to benefit from the given replica for that group. During the later run of the application, the dynamic data layout scheme will estimate the request access cost if it were redirected to the created replicas, and assign it to the corresponding replica with lowest access cost.

Despite the fact that data read in the later runs of the application is quite simple, data write is more complicated since it involves several replicas. There are a lot of possible alternatives to process data write operations. In our design and implementation, we use a lazy synchronization mechanism [14] for data writes. First, we write data to the selected replica. Then, we apply lazy updates to synchronize data from the first replica to other replicas. Hence, we only consider data access cost on the chosen replica for data writes, and ignore the background data synchronization cost.

3.5 Implementation

We implemented HAS in the MPI-IO library MPICH2 and parallel file system OrangeFS. We choose OrangeFS because it is a popular parallel file system and directly provides the varied-size striping method for file servers.

3.5.1 Selective Static Data Layout

The procedure of the static data layout of HAS scheme includes the following three phases, as shown in Fig. 4.

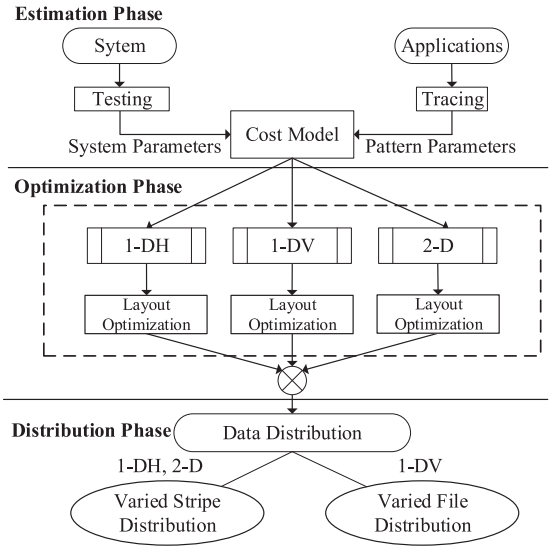


Fig. 4. The static data layout optimization procedure.

The *estimation* phase consists of two parts, system testing and application tracing. For system testing, the network parameters, e and t, the storage parameters, $\alpha_h, \beta_h, \alpha_{sr/w}$, $\beta_{sr/w}$, and the system parameters, such as m and n can be regarded as constants. We use all file servers in the parallel file system to test the storage parameters for HServers and SServers with sequential/random and read/write patterns and then we calculate the average for HServers and SServers. We use many pairs of clients and file servers to estimate the network parameters. Again the tests are conducted thousands of times for the purpose of accuracy, and we use the average value for the network parameters. For application tracing, we use a trace collector, IOSIG [19], to obtain the run-time statistics of data accesses during the application's first execution. Based on the I/O trace, we obtain the application's I/O pattern related parameters, such as p, r, and o.

In the *optimization* phase, using the parameters obtained in the *estimation* phase, we apply the cost model and linear programming optimization methods in Section 3.3 to determine the optimal file data distribution on HServers and SServers for each of the three layout policies. Since each policy may only give sub-optimal performance because of unique characteristics of applications, HAS compares their performance and chooses the one with lowest cost as the final data layout policy.

In the *distribution* phase, we distribute the file data with the optimal data layout policy and the corresponding layout parameters for later runs of the applications. For *1–DH* and *2–D*, we utilize the APIs supported by OrangeFS to implement the specific variable stripe distribution and group distribution. In OrangeFS, a file can either be accessed by the PVFS2 or the POSIX interface. For PVFS2 interface, we utilize the "pvfs2-xattr" command to set the data distribution policy and the related layout parameters for directories where the application files are located. For POSIX interface, we use the "setfattr" command to reach the similar data layout optimization goal. For *1–DV* policy, we create different numbers of process files on HServers and SServers.

3.5.2 Selective Dynamic Data Layout

The selection of the replica for each file request is based on the cost analysis with the proposed model. We made a prototype of the cost estimation and dynamic data replica selection by modifying the standard MPI-IO functions.

MPI_File_open. While opening a file, the dynamic strategy opens all the corresponding replica files, each are distributed on the underlying servers with a different data layout.

MPI_File_read. For each I/O read, the dynamic strategy first evaluate the data access costs for all replicas based on the proposed model, and then chooses the replica with the lowest cost to handle the I/O request. When data access is finished, the offsets of all replicas are synchronized.

MPI_File_Write. For each I/O write, the strategy synchronize related data blocks and then perform I/O operation on one replica with the lowest cost. Then the strategy insert the write requests of other replicas into a lazy synchronization queue. When data access is finished, the offsets of all replicas are synchronized.

MPI_File_seek. It calculates the offset and conducts the seek operation in the opened replica files.

MPI_File_close. It synchronizes data for all replicas and closes all the opened replica files.

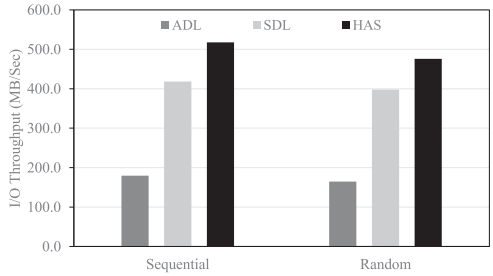
For data writes, all write requests issued to other replicas are insert into a lazy request queue right after writing to the selected replica. In order to avoid interfering with the normal I/O operations, a dedicated data synchronization thread is implemented to conduct these lazy write requests in the queue. Since the data synchronization is a background operation, each write request can return immediately after inserting the lazy write requests into the queue.

4 Performance Evaluation

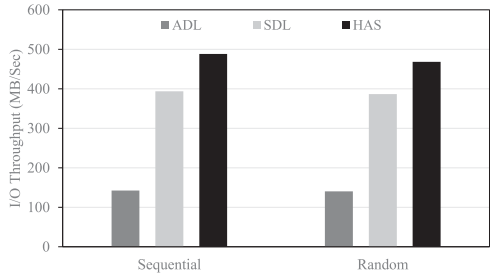
4.1 Experimental Setup

We conducted the experiments on a 65-node SUN Fire Linux cluster, where each node has two AMD Opteron(tm) processors, 8 GB memory and a 250 GB HDD. Sixteen nodes are equipped with additional OCZ-REVODRIVE 100 GB SSD. All nodes are equipped with Gigabit Ethernet interconnection. The operating system is Ubuntu 9.04, the MPI-IO library is MPICH2-1.4.1p1, and the parallel file system is OrangeFS 2.8.6. Among the available nodes, we select eight as client computing nodes, eight as HServers, and eight as SServers. In the experiments, the hybrid OrangeFS file system is built on four HServers and four SServers unless otherwise specified.

We compare HAS with two other data layout schemes: the application-aware scheme (ADL) [8] and the storage-aware scheme (SDL) [17]. In ADL, file data is placed across the hybrid file servers with one of the three policies according to the application's access pattern, but each server is assigned a fixed-size file stripe. In SDL, the file stripe sizes for the hybrid servers are determined by the server performance but only *1–DH* policy is chosen, without fully considering application access patterns. Since ADL and SDL have shown considerable performance improvements over the default data layout scheme namely the fixed-size striping in previous work [8], [17], we do not compare HAS with the default scheme in this paper.



(a) Read throughput



(b) Write throughput

Fig. 5. Throughputs of IOR under different layout schemes with different I/O modes.

We use the popular benchmark IOR [15], BTIO [21], HPIO [22] and a real application [23] to test the performance. We first show the efficiency of the selective static data layout scheme for a specific access pattern, then we show the efficiency of the dynamic data layout for mixed access patterns.

4.2 Selective Static Data Layout

4.2.1 IOR Benchmark

We provide two sets of experiments, varying application characteristics and varying storage characteristics. Unless otherwise specified, IOR runs with 32 processes, each of which performs I/O operations on an individual 256 MB parallel file with request size of 512 KB. We illustrate the importance of considering both application and storage characteristics for an efficient layout scheme, by comparing to schemes which only consider one type of characteristics.

Varying application characteristics. We vary the following application related traits: I/O operation type, number of processes, and request size.

First we ran IOR with sequential and random read and write I/O operations. Fig. 5 shows the throughput of IOR. We observe that HAS outperforms ADL and SDL. By using the optimal data distribution for HServers and SServers, HAS improves read performance up to 189.7 percent over ADL with all I/O access patterns, and write performance up to 242.7 percent. Compared with SDL, HAS improves the performance up to 23.8 percent for reads and 21.1 percent for writes. Although ADL accounts for I/O operation type variation, HAS has superior performance than ADL because it considers file server performance differences. HAS provides optimal performance for read and write operations, but SDL degrades in performance because its lack of application awareness.

To give a detailed explanation for HAS's performance, Fig. 6 plots the I/O time of each file server when IOR issued sequential read operations under the three layout schemes.

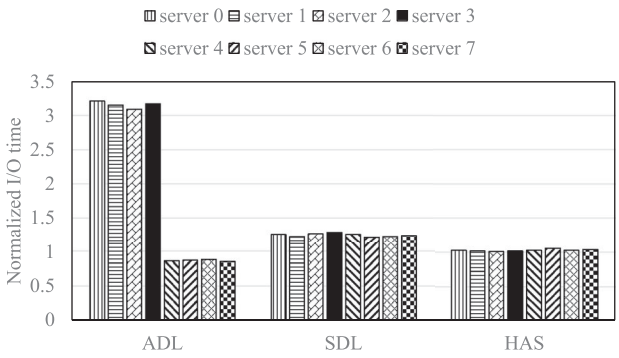
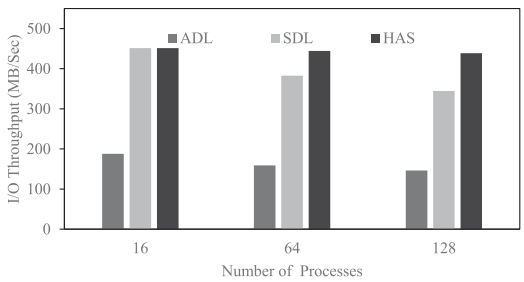


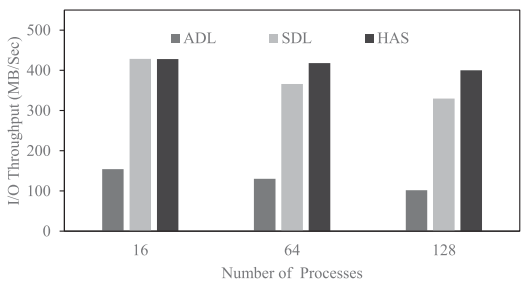
Fig. 6. I/O time on each server under different data layout schemes.

The I/O time refers to the aggregated data access completion time of all sub-requests assigned to a server. We use PVFS2 utilities such as "pvfs2-set-eventmark" and "pvfs2-set-debugmark" to trace I/O access information in the pvfs2-server.log on each file server, and then analyze the log file to get the I/O time. The I/O time is normalized to that of the minimal I/O time of all file servers under the HAS layout. Among the eight file servers, server 0 to 3 are HServers, and the rest are SServers. We observe that the I/O loads of HServers and SServers are severely skewed under ADL since it uses a fixed-size stripe. In contrast, SDL and HAS have nearly even I/O loads. However, HAS leads to less I/O time on each server than SDL because it selects the data layout policy (*1–DV*) with the lowest cost while SDL uses *1–DH*. Thus, HAS improves the file system performance.

Then we evaluated the layout schemes with different number of processes. The IOR benchmark was executed under the random access mode with 16, 64 and 128 processes. As displayed in Fig. 7, the result is similar to the previous test. HAS has the best performance among the three schemes. Compared with ADL, HAS improves the read performance by 140.3, 179.7, and 200.2 percent respectively with 16, 64 and 128 processes, and write performance by 174.3, 200.7, and 292.7 percent. Compared with SDL, HAS

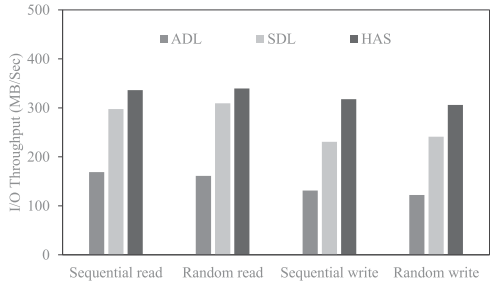


(a) Read throughput

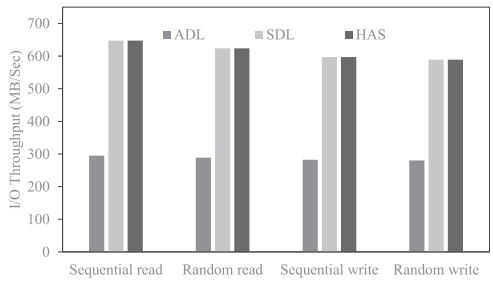


(b) Write throughput

Fig. 7. Throughputs of IOR with varied number of processes.



(a) Request size is 128KB



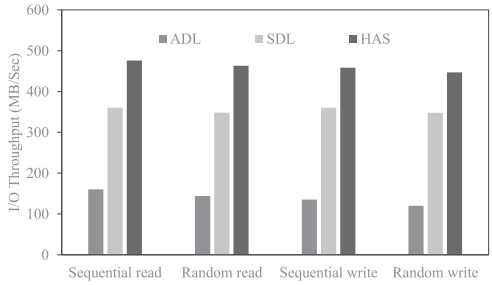
(b) Request size is 4096KB

Fig. 8. Throughputs of IOR with varied request sizes.

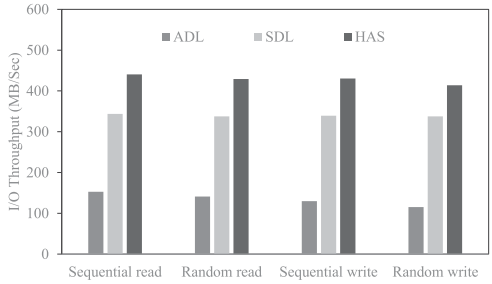
achieves similar performance for 16 processes. For 64 and 128, HAS improves read performance by 16.1 and 27.3 percent respectively, and write performance by 14.2 and 21.3 percent. When the number of processes is large, the *1-DV* policy, implemented in HAS, provides better performance than the *1–DH* layout used by SDL. As the number of processes increase, the performance of the hybrid PFS decreases because more processes lead to server I/O contention in HServers and SServers. These results show that HAS scales excellently with the number of I/O processes.

Finally, the I/O performance was examined with different request sizes. We set the request size to 128 and 4,096 KB, and the number of processes to 32. From Fig. 8a, we can observe that HAS can improve the read performance up to 110.3 percent, and write up to 151.6 percent in comparison with ADL. Compared with SDL, HAS also has better performance: the read performance is increased up to 13.4 percent, and write performance is increased up to 37.7 percent. As the request size increases, *1–DH* tends to be the best layout policy. For example when the request size is 4,096 KB, HAS selects the same data layout policy as SDL, *1–DH*. These results validate that HAS can choose appropriate data distribution for HServers and SServers when the request size varies.

Varying system characteristics. We examined the I/O performance with different server configurations. We varied the numbers of HServers and SServers with the ratios of 5:3 and 6:2. Fig. 9 shows the bandwidth of IOR with different file server configurations. Based on the results, HAS can improve I/O throughput for both read and write operations. When the ratio is 5:3, HAS improves the read and write performance by up to 171.6 and 232.4 percent respectively, when compared to ADL. Compared with SDL, HAS increases the read performance by 21.9 percent, and write performance by 17.1 percent. When the ratio is 6:2, the performance gap is decreased because the server configuration is more homogeneous. In the experiments, the read and write performance disparity between HAS and ADL



(a) 5HServers : 3SServers



(b) 6HServers : 2SServers

Fig. 9. Throughputs of IOR with varied file server configurations.

enlarges as the number of SServers increase because HAS is storage device conscious. By varying the system characteristics, we prove that the consideration of system traits is essential to optimal data distribution.

4.2.2 BTIO Benchmark

We also use the BTIO benchmark [21] to evaluate the proposed scheme. BTIO represents a typical scientific application with interleaved intensive computation and I/O phases.

We consider the Class *B* and *epio* subtype BTIO workload. That is, we write and read a total size of 1.69 GB data. We use 16, 36, and 64 compute processes since BTIO requires a square number of processes. Each process accesses its own independent file. Output files are distributed across six HServers and two SServers on the hybrid OrangeFS file system.

As shown in Fig. 10, compared to ADL and SDL, HAS achieves better throughput and scalability. Compared to ADL, HAS improves the performance by 153.1, 157.6, and 175.2 percent with 16, 36, 64 processes, respectively. For SDL, HAS achieves the improvement by up to 48.2 percent.

4.3 Selective Dynamic Data Layout

To verify the efficiency of the dynamic replication-based data layout strategy, we first ran several IOR or HPIO

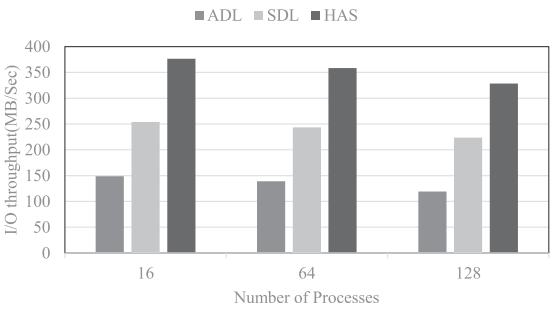


Fig. 10. Throughputs of BTIO under different data placement schemes.

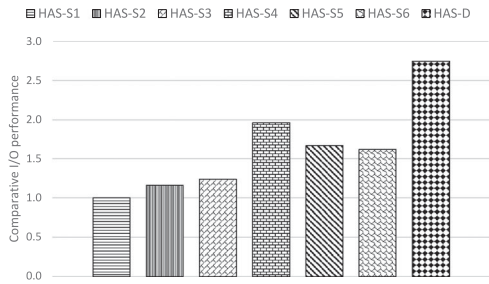


Fig. 11. Performance of IOR with mixed access patterns.

instances one by one with various parameters to simulate mixed access patterns at different moments, then we used a real application to evaluate the performance.

4.3.1 IOR Benchmark

We varied the request size in each instance of IOR. The number of process was 32, the requests were random read operations, and the request sizes were 32, 64, 128, 512 KB, 1, and 2 MB. We measured the performance of all IOR instances under six static layout policies (HAS-S1 to HAS-S6), each optimized for one kind of request size respectively. For example, in HAS-4, the stripe size pairs on HServer and SServer was $<28, 100\ KB>$, optimized for random reads with size of 512 KB. We set these layout policies to different directories in OrangeFS system. As for the dynamic layout policy (HAS-D), it first chose one directory to access based on the cost estimation, and then wrote/read files in that directory.

Fig. 11 shows the results of the mixed IOR workloads, where y-axis represents the comparative performance with HAS-S1 layout policy. Here the performance is represented by the average I/O bandwidth, which is calculated by the total data size divided by the total running time. As shown in Fig. 11, the proposed replication-based dynamic data layout policy can get the best performance. The performance improvement is up to 174.5 percent compared with the other data layout policies.

4.3.2 HPIO Benchmark

We also varied the number of processes in each HPIO instance. HPIO can generate variousdata access patterns by changing three parameters: region count, region spacing, and region size. We set the region count to 1,024, and keep the region spacing to 0. The region size is fixed to 512 KB, and the numbers of processes were 8, 64, 256, and 512. We changed HPIO code to make each process access one file. We measured the performance of all IOR instances under four static layout policies (HAS-S1 to HAS-S4), each optimized for requests with one kind of process number respectively, and then compared them with the dynamic data layout policy. For instance, if the process number is 8, the static layout policy (HAS-S1) was 1–DH and the stripe size pair on HServer and SServer was $<24, 104\ KB>$; if the process number is 256, the layout policy (HAS-S3) was 1–DV and the process number pair on HServer and SServer was $<16, 48>$. Fig. 12 shows the results, in which y-axis represents the comparative performance normalized to HAS-S1 data layout policy. The performance improvement of the dynamic replication-based strategy (HAS-D) is around 29.8-130.6 percent compared with other static layout policies.

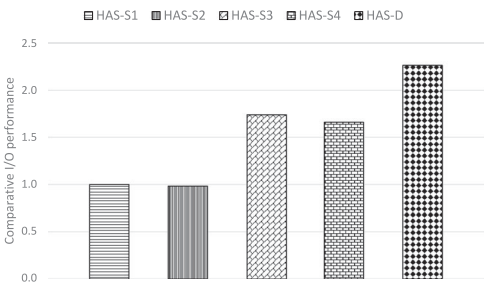


Fig. 12. Performance of HPIO with mixed access patterns.

Fig. 13. Performance of LANL App2 with different layouts.

4.3.3 Real Application

Finally, we used a real application, 'Anonymous LANL App 2' [23], to evaluate the proposed layout scheme. In this application, each process issues I/O requests in a non-uniform way at different parts of a shared file. The request size is varied at different moments during the application's execution. For example, in one part of the file, the request size of each process is relative small, which is less than 8 KB. In another part of the file, each process issues relative large requests of 131,072 and 131,056 bytes iteratively. The data accesses of this application were replayed according to the I/O trace, but we manually enforce each process to access an individual file. We measured the performance of the application with three static layout policies (HAS-S1 to HAS-S3), optimized for the top-three frequently accessed request sizes, and under the dynamic layout policy (HAS-D). Similar to the previous tests, Fig. 13 indicates that the dynamic data layout scheme can achieve 26.8-63.1 percent performance improvement compared to other static data layout policies.

From the above experiments, we can find that for IOR, HPIO and the real application, the selective dynamic layout strategy is superior to any static data layout policy. Hence, such dynamic layout strategy based on data replication is effective for mixed I/O workloads, which implies a great potential of trading unused storage space for better I/O performance.

5 Related Work

5.1 I/O Request Stream Optimization

A great deal of research has focused on reorganizing I/O request streams to minimize access time spent on I/O device and network. Generally, such optimizations are implemented at the I/O middleware layer. For example, instead of accessing multiple small, noncontiguous requests, data sieving [24] applies the strategy of accessing a contiguous chunk created by gathering the noncontiguous requests. Datatype I/O [25] and List I/O techniques [26] allow noncontiguous I/O requests to be converted into a single I/O request, thereby limiting the number of total requests. Collective I/O [24] also rearranges I/O accesses into a larger contiguous request, but considers the multi-process level instead of a single process.

5.2 Data Layout in Homogeneous File Systems

Parallel file systems have different data layout strategies, which allow for numerous data layout optimization methods [8]. Several techniques, including data partition [27], [28], data migration [29], and data replication [8], [10], [14], are applied to optimize data layouts depending on I/O workloads. Segment-level layout scheme logically divides a file to several parts and appoints an optimal stripe size for each part [30]. Another methodology, server-level adaptive layout strategy, selects different stripe sizes depending upon the type of the file server [9]. PARLO is designed for accelerating queries on scientific datasets by applying user specified optimizations [11]. Tantisiriroj et al. [31] use HDFS-specific layout optimizations [32] to improve the performance of PVFS. However, all these studies are designed for homogeneous HDD-based file systems, and can't be applied to heterogeneous environments.

5.3 Data Layout in Heterogeneous File Systems

SSDs are commonly integrated into parallel file systems due to their performance benefits. For now, most SSDs are used as a cache [33] or as a hybrid storage device [12], [34]. However, the vast majority of research is focused on a single file server. In contrast, AdaptRaid confronts load imbalance in heterogeneous disk arrays [35], which cannot be implemented in PFSs. Liu et al. use SSD-based nodes as buffers to handle burst requests [36]. CARL [37] situates data regions with high access costs onto SSD-based file servers. Welch and Noer place small files and file metadata onto SSDs, and large file extents onto HDDs [38]. However, these schemes cannot simultaneously utilize HDDs and SSDs. The PADP [17] employs stripe size variation to improve the performance of hybrid PFSs, yet the schemes are only optimized for the one-horizontal *(1DH)* layout policy. Our previous work [13] adaptively selects the optimal data layout for heterogeneous parallel file systems. However, it is only suitable for applications with specific access patterns. This work can be further applied to applications with mixed access patterns.

6 Discussion

Admittedly, the proposed replication-based dynamic data layout strategy requires more storage space for both HServers and SServers, which might be an unwanted feature by users. This is a trade-off between data access performance and storage capacity, like almost all other replication-based strategies. Since the capacities of current HDDs and SSDs are increasing quickly, and HAS can be used to only replicate a small portion active data based on data access pattern for performance-critical data, the space trade-off may not be a subject of concern. With replications of performance-critical data, the proposed data layout scheme provides a good alternative to existing approaches for data-intensive applications.

7 Conclusions

In this study, we propose a heterogeneity-aware selective data layout scheme for parallel file systems with both HDD and SSD-based servers. We alleviates the inter-server load imbalance by varying the file stripe sizes or the number of files on different servers based on their storage performance. In addition, we selects the optimal static data layout from three types of candidates for applications with specific access patterns. Moreover, to adapt to dynamic changes of access behaviors in some complex applications, we also developed a dynamic data layout strategy that stores file with multiple copies, each using a different layout policy, and selects the proper copy with the lowest cost to serve I/O requests. Generally, a large number of copies for the dynamic layout policy would lead to a better performance, but also come at a higher cost. In principle, HAS improves hybrid parallel file system performance by matching data layout with both application characteristics and storage capabilities. We have developed and presented the proposed layout optimization scheme under MPICH2 and OrangeFS. Experimental results show that HAS improves the I/O performance by up to 292.7 percent over the existing file data layout schemes.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback and comments, which substantially improved the quality of this paper. We would also like to thank Ning Liu, who helped to further improve this paper. This research is supported in part by the China National Basic Research Program (973 Program, No. 2015CB352400), NSFC under grant U1401258, No. 61572377, No. 61572370, and No. 61373040, the PhD Programs Foundation of Ministry of Education of China under Grant No. 20120141110073, the Natural Science Foundation of Hubei Province of China under Grant No. 2014CFB239, the Open Fund from HPCL under Grant No. 201512-02, the Open Fund from SKLSE under Grant No. 2015-A-06, and the US National Science Foundation under Grant CNS-1162540.

References

[1] Orange File System. (2015) [Online]. Available: 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 *Proc. 1st USENIX Conf. File Storage Technol.*, 2002, pp. 231–244.

[4] R. Latham, R. Ross, B. Welch, and K. Antypas, "Parallel I/O in practice," Tech. Rep. Tutorial of the Int. Conf. High Perform. Comput., Netw., Storage Anal., 2013.

[5] A. Caulfield, L. Grupp, and S. Swanson, "Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications," in *Proc. 14th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2009, pp. 217–228.

[6] 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 *Proc. IEEE Int. Conf. Embedded Ubiquitous Comput.*, 2013, pp. 511–517.

[7] S. He, X.-H. Sun, and B. Feng, "S4D-cache: Smart selective SSD cache for parallel I/O systems," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 514–523.

[8] H. Song, Y. Yin, Y. Chen, and X.-H. Sun, "A cost-intelligent application-specific data layout scheme for parallel file systems," in *Proc. 20th Int. Symp. High Perform. Distrib. Comput.*, 2011, pp. 37–48.

[9] H. Song, H. Jin, J. He, X.-H. Sun, and R. Thakur, "A server-level adaptive data layout strategy for parallel file systems," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, 2012, pp. 2095–2103.

[10] J. Jenkins, X. Zou, H. Tang, D. Kimpe, R. Ross, and N. F. Samatova, "RADAR: Runtime asymmetric data-access driven scientific data replication," in *Proc. Int. Supercomput. Conf.*, 2014, pp. 296–313.

[11] 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 *Proc. 13th IEEE/ACM Int. Symp. Cluster, Cloud, Grid Comput.*, 2013, pp. 343–351.

[12] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the best use of solid state drives in high performance storage systems," in *Proc. Int. Conf. Supercomput.*, 2011, pp. 22–32.

[13] S. He, X.-H. Sun, and A. Haider, "HAS: Heterogeneity-aware selective data layout scheme for parallel file systems on hybrid servers," in *Proc. 29th IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 613–622.

[14] 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 *Proc. 27th IEEE Int. Parallel Distrib. Process. Symp.*, 2013, pp. 345–356.

[15] Interleaved Or Random (IOR) Benchmarks. (2014) [Online]. Available: http://sourceforge.net/projects/ior-sio/

[16] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. 11th Int. Joint Conf. Meas. Model. Comput. Syst.*, 2009, pp. 181–192.

[17] S. He, X.-H. Sun, B. Feng, and F. Kun, "Performance-aware data placement in hybrid parallel file systems," in *Proc. 14th Int. Conf. Algorithms Archit. Parallel Process.*, 2014, pp. 563–576.

[18] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Automatic identification of application I/O signatures from noisy Server-side traces," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 213–228.

[19] 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 *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2008, pp. 1–12.

[20] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning, "Conquest: Better performance through a disk/persistent-RAM hybrid file system," in *Proc. USENIX Annu. Techn. Conf.*, 2002, pp. 15–28.

[21] The NAS parallel benchmarks. (2014) [Online]. Available: www.nas.nasa.gov/publications/npb.html

[22] A. Ching, A. Choudhary, W.-K. Liao, L. Ward, and N. Pundit, "Evaluating I/O characteristics and methods for storing structured scientific data," in *Proc. 20th Int. Parallel Distrib. Process. Symp.*, 2006.

[23] Application I/O Traces: Anonymous LANL App2. (2014) [Online]. Available: http://institutes.lanl.gov/plfs/maps/

[24] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Proc. 7th Symp. Frontiers Massively Parallel Comput.*, 1999, pp. 182–189.

[25] A. Ching, A. Choudhary, W.-k. Liao, R. Ross, and W. Gropp, "Efficient structured data access in parallel file systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2003, pp. 326–335.

[26] A. Ching, A. Choudhary, K. Coloma, L. Wei-keng, R. Ross, and W. Gropp, "Noncontiguous I/O accesses through MPI-IO," in *Proc. 3rd IEEE/ACM Int. Symp. Cluster Comput. Grid*, 2003, pp. 104–111.

[27] Y. Wang and D. Kaeli, "Profile-guided I/O partitioning," in *Proc. 17th Annu. Int. Conf. Supercomputing*, 2003, pp. 252–260.

[28] S. Rubin, R. Bodik, and T. Chilimbi, "An efficient Profile-analysis framework for Data-layout optimizations," *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 140–153, 2002.

[29] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "Borg: Block-reorganization for Self-optimizing storage systems," in *Proc. 7th Conf. File Storage Technol.*, San Francisco, CA, USA, 2009, pp. 183–196.

[30] 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 *Proc. 11th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2011, pp. 414–423.
[31] 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 *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–12.
[32] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[33] T. Pritchett and M. Thottethodi, "SieveStore: A highly-selective, ensemble-level disk cache for cost-performance," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 163–174.
[34] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, "Cost effective storage using extent based dynamic tiering," in *Proc. 9th Conf. File Storage Technol.*, 2011, pp. 273–286.
[35] T. Cortes and J. Labarta, "Taking advantage of heterogeneity in disk arrays," *J. Parallel Distrib. Comput.*, vol. 63, no. 4, pp. 448–464, 2003.
[36] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in Leadership-class storage systems," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–11.
[37] 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 *Proc. IEEE Int. Conf. Cluster Comput.*, 2013, pp. 1–8.
[38] B. Welch and G. Noer, "Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions," in *Proc. IEEE 29th Symp. Mass Storage Syst. Technol.*, 2013, pp. 1–12.

Shuibing He received the PhD degree in computer science and technology from Huazhong University of Science and Technology, China, in 2009. He is currently an assistant professor at the Computer School, Wuhan University, China. His current research areas include parallel I/O systems, file and storage systems, high-performance computing, and distributed computing.

Yang Wang received the BSc degree in applied mathematics from Ocean University of China, 1989, and the MS and PhD degrees in computer science from Carleton University (2001) and the University of Alberta, Canada (2008), respectively. He is currently with Shenzhen Institute of Advanced Technology, Chinese Academy of Science, as a professor. His research interests include cloud computing, big data analytics, and Java Virtual Machine on multicores.

Xian-He Sun received the BS degree in 1982 in mathematics from Beijing Normal University, China, and the MS and PhD degrees in 1987 and 1989, respectively, in computer science from Michigan State University. He is a distinguished professor in the Department of Computer Science, the Illinois Institute of Technology (IIT), Chicago, and the director in the Scalable Computing Software laboratory, IIT. He is a guest faculty in the Mathematics and Computer Science Division at the Argonne National Laboratory. His research interests include parallel and distributed processing, memory and I/O systems, software systems, and performance evaluation and optimization. He is a fellow of the IEEE.