Heterogeneity-Aware Collective I/O for Parallel I/O Systems with Hybrid HDD/SSD Servers

Shuibing He, Yang Wang, Xian-He Sun, *Fellow, IEEE*, Chuanhe Huang, and Chengzhong Xu, *Fellow, IEEE*

Abstract—Collective I/O is a widely used middleware technique that exploits I/O access correlation among multiple processes to improve I/O system performance. However, most existing implementations of collective I/O strategies are designed and optimized for homogeneous I/O systems. In practice, the homogeneity assumptions do not hold in heterogeneous parallel I/O systems, which consist of multiple HDD and SSD-based servers and become increasingly promising. In this paper, we propose a heterogeneity-aware collective-I/O (HACIO) strategy to enhance the performance of conventional collective I/O operations. HACIO reorganizes the order of I/O requests for each aggregator with awareness of the storage performance of heterogeneous servers, so that the hardware of the systems can be better utilized. We have implemented HACIO in ROMIO, a widely used MPI-IO library. Experimental results show that HACIO can significantly increase the I/O throughputs of heterogeneous I/O systems.

Index Terms—Parallel I/O system, I/O middleware, collective I/O, solid state drive

1 Introduction

DATA access is one of the critical performance bottlenecks of modern computer systems. Most of today's high performance computing (HPC) applications in scientific computing and engineering simulation fields become increasingly data-intensive [1]. To satisfy the enormous I/O requirements, HPC clusters rely on parallel I/O systems to provide efficient data accesses. However, when facing noncontiguous small requests, parallel I/O systems still perform poorly even adding more servers into the system [2]. Hence both new hardware technologies and software strategies are required to boost I/O system performance.

On the hardware side, HPC systems have begun to use solid state drives (SSD) to provide data storage service [3]. Compared with HDDs, SSDs have much higher data transfer rate and lower access latency. However, due to the high price of SSD, it may not be cost-effective to build a large I/O system completely based on SSDs. A hybrid parallel I/O system comprised of both HDD servers (HServer) and SSD servers (SServer), becomes increasingly attractive [4], [5], [6].

There are two typical strategies to exploit SSDs in a hybrid I/O system. One strategy is to leverage SServers as a cache layer of HServers [2]. While intuitive, this strategy is only suitable for I/O systems where SServers can provide much better aggregated I/O performance than HServers, by either using high-end SSDs or deploying a large number of SServers. However this is not always available in reality. In contrast, the other strategy puts SServers at the same level of HServers [4], [7]. This strategy caters for the requirements where only a small number of entry-level SSDs are deployed. In this study, we focus on the hybrid I/O system adapting the later strategy.

On the software side, collective I/O [8] is the most popular middleware method to optimize parallel I/O performance. As shown in Fig. 1, it shuffles data among multiple processes and merges data requests into large contiguous ones before sending them to underlying parallel file systems (PFS). However, although there are various methods devoted to optimize the collective I/O operations [9], [10], [11], [12], [13], most approaches are originally designed for homogeneous servers. With the emergence of the hybrid I/O systems, existing collective-IO strategies may largely compromise the system performance, as we illustrate in Section 2.2.

In this paper, we propose HACIO, a new Heterogeneity-Aware Collective I/O strategy to optimize the performance of hybrid parallel I/O systems. HACIO enhances existing strategies by reorganizing the request order of each aggregator with consideration of the performance disparity between heterogeneous servers. By coordinating the request orders of multiple aggregators, HACIO leads to a better load balance among heterogeneous servers and hence makes a more efficient utilization of system hardware. Furthermore, HACIO reorders requests only within each file domain, thus does not introduce additional overhead in the data exchange phase while other collective I/O optimization schemes might do.

To the best of our knowledge, this study is the first effort to integrate storage heterogeneity into the data access optimization at parallel I/O middleware layer. Specifically, we make the following contributions.

- By analyzing the collective I/O operation process, we find the traditional collective I/O strategies cannot fully utilize the system hardware resource in a hybrid parallel I/O system.
- We propose a heterogeneity-aware collective I/O strategy, which reorganizes request orders of file domains by scheduling more requests to the same type of servers in one circle, to mitigate the load imbalance among heterogeneous servers.
- We implement the prototype of HACIO in MPI-IO library, and have conducted extensive tests to verify the benefits of the HACIO scheme. Experiment results illustrate that HACIO can significantly improve I/O system performance.

The rest of this paper is organized as follows. We introduce the background and motivation in Section 2. We describe the idea, design and implementation of HACIO in Section 3. Performance evaluations of HACIO are presented in Section 4. We introduce the important related work in Section 5. Finally, we conclude the paper in Section 6.

2 Background and Motivation

2.1 Collective I/O and Implementation

Instead of issuing I/O requests independently for each process, collective I/O merges noncontiguous requests from multiple processes and performs large contiguous I/O operations on a subset of processes called aggregators. The benefits of collective I/O are three-fold. First, it can filter redundant requests from multiple processes. Second, it can produce large contiguous accesses to a file region, leading to better disk efficiency. Third, it can lessen the I/O overheads because of the reduced number of file system calls.

The most popular implementation of collective I/O is two-phase I/O in ROMIO [14], which is a high-performance implementation of Message Passing Interface (MPI)-IO library. It consists of a communication phase and an I/O phase. The number of processes participating in the I/O phase (aggregators) can be specified by

- *S. He and C. Huang are with the State Key Laboratory of Software Engineering, Computer School, Wuhan University, Luojiashan, Wuhan, Hubei 430072, China, and the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Xueyuan Blvd. 1068, Shenzhen 518055, China. E-mail: {heshuibing, huangch}@whu.edu.cn.*
- *Y. Wang and C. Xu are with the Shenzhen Institute of Advanced Technology, Chinese Academy of Science, Xueyuan Blvd. 1068, Shenzhen 518055, China. E-mail: {yang.wang1, cz.xu}@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 29 June 2016; revised 13 Nov. 2016; accepted 5 Dec. 2016. Date of publication 7 Dec. 2016; date of current version 16 May 2017.
Recommended for acceptance by J. Cao.
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/TC.2016.2637353

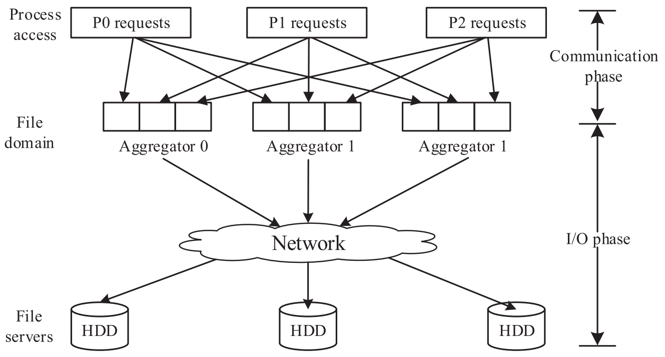

Fig. 1. Collective I/O and the two-phase implementation.

users. Fig. 1 shows an example of a two-phase collective I/O write operation for three processes. We assume three processes participate in the I/O phase and each aggregator has a sufficient memory buffer. In the communication phase, each process communicates to each other thus each process knows the aggregated span of I/O requests from all processes. Then the aggregated span of I/O requests is partitioned into multiple file regions (called file domains), each of which is assigned to one aggregator. After that, each aggregator sends data to requesting aggregators, which write the data on behalf of it, and receives desired data from other aggregators. In the I/O phase, each aggregator is responsible for carrying out I/O requests belonging to its own file domain.

There are two main user configurable parameters in the ROMIO collective I/O: the number of aggregators and the temporary buffer size. By default, ROMIO picks one process as an aggregator at each computing node and sets the buffer size to 4 MB for each aggregator. These parameters can be changed by users through the MPI-IO's hint mechanism. Generally, if the I/O size is too large to fit in a single buffer, ROMIO will perform the two-phase I/O in several cycles to complete the whole I/O operation [12]. Each aggregator first calculates the length of its file domain, and then divides the length by the allowed maximum buffer size to get the total number of cycles needed.

2.2 A Motivating Example

As most existing collective I/O strategies reorganize requests in a way that is oblivious to the heterogeneity of file servers, they can largely degrade I/O system performance.

Fig. 2 demonstrates a representative example of the traditional collective I/O strategy and the data layout on heterogeneous file servers. We assume that the parallel program has four processes (P0-3) and two of them act as the aggregators (A0-1). Each process writes two logical blocks to the file system. For example, P0 issues write requests to block 0 and 4, while P1 writes block 1 and 5. We assume the system consists of two HServers and two SServers, and

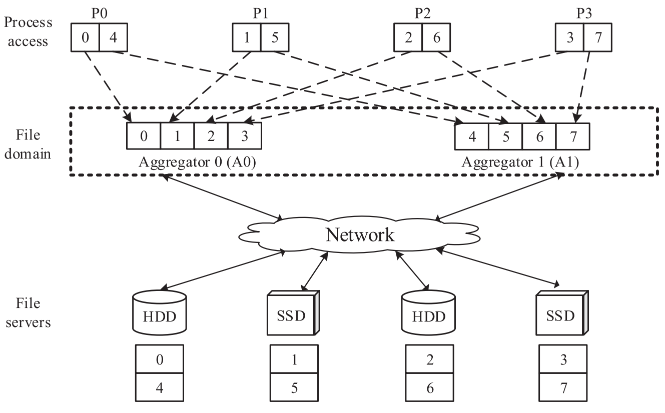

Fig. 2. An example of conventional collective I/O strategy. The I/O request order in each file domain is based on the request's logical address.

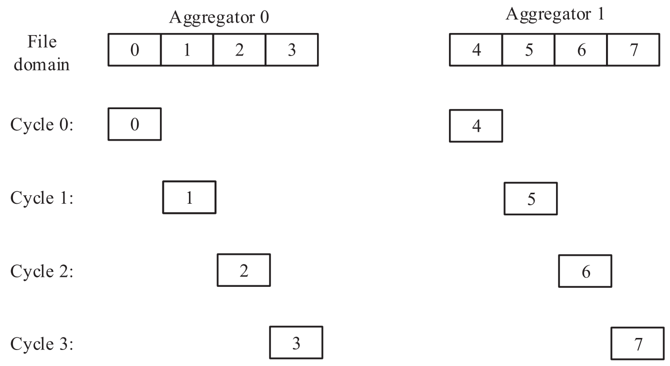

Fig. 3. The request order for each file domain with the conventional collective I/O strategy. The load imbalance issue among each I/O server occurs.

file blocks are distributed on servers with the popular round-robin data layout scheme.

By the default collective I/O strategy, the required file region is evenly assigned to two file domains since there are two aggregators. Aggregator A0 has a file domain containing block 0-3, while A1 has a file domain containing block 4-7, as shown in the dash rectangle of Fig. 2. Then each aggregator takes multiple cycles to issue requests in its file domain depending on the buffer size. We assume the buffer size is one block then each aggregator needs four cycles to finish the collective I/O operation. Traditional strategy naively selects the requests *according to the request logical address*. For example, A0 first issues request for block 0, then for block 1 and 2, and finally for block 3 in the four cycles. The detailed request order of each aggregator is shown in Fig. 3. During the communication phase in each cycle, as all aggregators need to be synchronized to make sure the buffer can be safely used [10], the execution time of each cycle is determined by the maximal time of all aggregators. Since SServers are much faster than HServers, each cycle's execution time is determined by the execution time of the requests accessing HServers. In specific, we assume each request on HServer and SServer requires 5T and 1T time, Table 1 shows the I/O time of each cycle. In Table 1, the I/O time of certain server is zero because there is no data access on the server in the cycle. As we can see that, the current strategy will lead to severe load imbalance among servers, which would result in suboptimal I/O performance.

There is an alternative approach to alleviate the load imbalance problem by taking the data layout information into consideration when deciding the I/O request orders of file domains. As shown in Fig. 4, by reorganizing the request orders of file domains, each aggregator takes responsibility for the data that resides in a different server in each cycle, which avoids the access contention from multiple aggregators and increase the I/O concurrency among servers. We call this strategy as concurrency-aware collective I/O (CACIO) in the context of this paper. The detailed I/O request order of CACIO for each aggregator is shown in Fig. 5. However, this strategy still suffers from the load imbalance issue among servers due to the heterogeneity between HServers and SServers. As shown in Fig. 5, although aggregator A1 only needs 1T to finish its I/O request in the first cycle, it has to wait A0 which needs 5T to finish the I/O request, because A0 and A1 are synchronized to

TABLE 1
Execution Time of Traditional Collective I/O Strategy

Cycle #	HServer0	SServer0	HServer1	SServer1	Cost (T)
0	10	0	0	0	10
1	0	2	0	0	2
2	0	0	10	0	10
3	0	0	0	2	2
		Sum			24

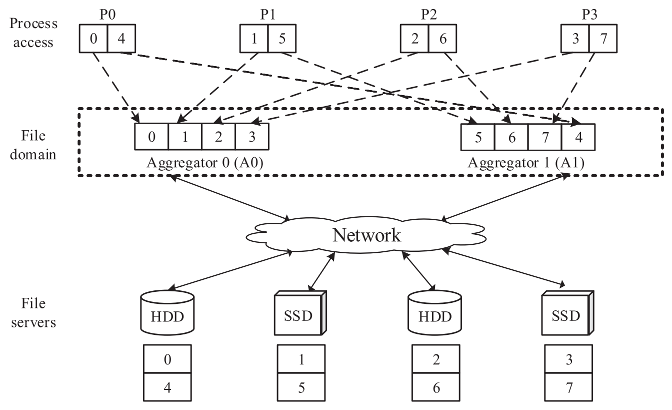

Fig. 4. Concurrency-aware collective I/O (CACIO). In each cycle each aggregator issues requests to a different server to increase I/O concurrency. For example, aggregator A1 sends requests in the order of “5,6,7,4” instead of “4,5,6,7” as in Fig. 2. However, it still suffers from the load imbalance issue among heterogeneous servers.

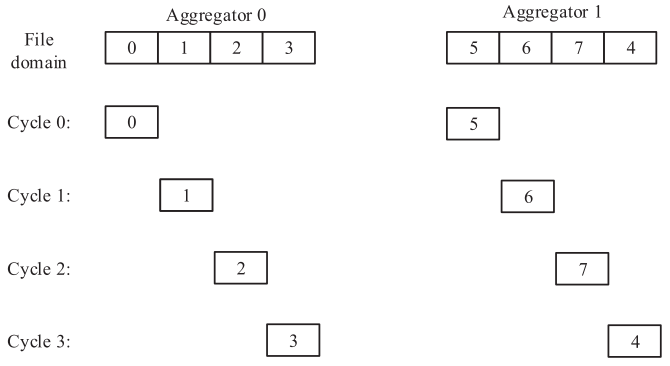

Fig. 5. The request order for each file domain with the alternative CACIO strategy. The load imbalance issue among heterogeneous file servers exists.

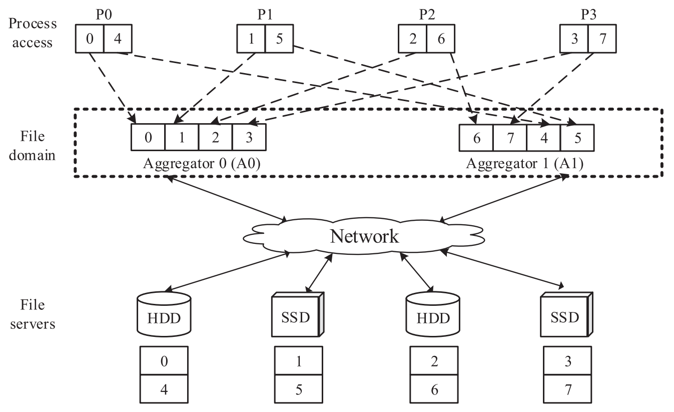

Fig. 6. Heterogeneity-Aware Collective I/O (HACIO). It reorganizes the request order for each file domain considering the heterogeneity of file servers and the physical data layout of I/O requests.

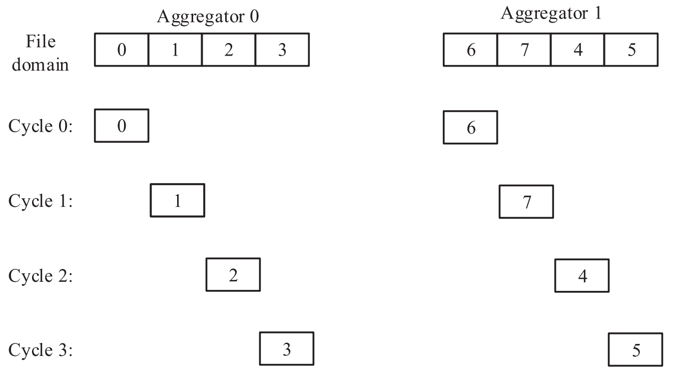

Fig. 7. The request order after reordering requests within the file domain by HACIO. The requests on the same type of servers are simultaneously carried out on different servers to avoid load imbalance, thus reducing the overall execution time of the collection I/O operation.

TABLE 2
Execution Time of the Alternative Collective I/O Strategy

Cycle #	HServer0	SServer0	HServer1	SServer1	Cost (T)
0	5	1	0	0	5
1	0	1	5	0	5
2	0	0	5	1	5
3	5	0	0	1	5
		Sum			20

each other in each cycle. A similar situation happens for A0 and A1 in terms of Cycle 1 to Cycle 3. Table 2 shows the detailed I/O time of each cycle. We can see that while CACIO can reduce the overall I/O time of the collective I/O operation from 24T to 20T, it still offsets the benefits of the parallel I/O system due to the ignorance of the heterogeneity among different types of file servers.

3 Heterogeneity-Aware Collective I/O Strategy

3.1 HACIO Design

As explained in the previous section, a limitation of the current collective I/O designs is that they are unaware of the performance heterogeneity among file servers, which can lead to low utilization of system hardware. Our strategy takes server performance features into consideration when determining collective I/O operations, together with file data layout and file system information. As one collective I/O operation may include multiple cycles and each cycle's performance is determined by the slowest aggregator, HACIO coordinately reorders the requests of all aggregators in each cycle. By scheduling more requests from the same type of servers in one cycle, HACIO alleviates the cost of each cycle and hence increases the efficiency of the entire collective I/O call.

We continue to use the previous example to show our design. As displayed in Fig. 6, the request sequence in the file domain of each aggregator is reordered depending on the request's server type. By doing so, all requests within a cycle attempt to access one type of server, as shown in Fig. 7. Thus, less I/O time is spent on waiting for the completion of requests on the slow HServers in the collective I/O operation. For example, in HACIO all aggregators issue requests on the same type of servers in Cycle 0, 1, 2 and 3. In the conventional implementation, all aggregators need to access requests on few servers within each cycle, leading to severe access contention. In the proposed alternative CACIO strategy, all aggregators need to carry out requests on two types of servers. While increasing I/O concurrency than the conventional strategy, it renders serious load imbalance (I/O time) on heterogeneous servers. As previously noted, the cycle where requests are issued to both HServers and SServers will finish at the same time as the cycle where requests only issued to HServers, since the cycle execution time is determined by the longest request, which are the HServer requests for both. The requests of Cycle 1 and Cycle 3 of HACIO are all physically located on SServers, thus the quick access time of SServers can be used to diminish the total collective I/O execution time, because these cycles can execute considerably faster than all cycles of the conventional collective I/O implementation and the mentioned alternative implementation. Thus, HACIO utilizes minor reordering of the data requests to create more cycles including requests on homogeneous servers, which can exploit the merits of high performing SServers. Table 3 shows the I/O execution time of each cycle in HACIO strategy for this example. We can see that

TABLE 3
Execution Time of Heterogeneity-Aware Collective I/O Strategy

Cycle #	HServer0	SServer0	HServer1	SServer1	Cost (T)
0	5	0	5	0	5
1	0	1	0	1	1
2	5	0	5	0	5
3	0	1	0	1	1
		Sum			12

HACIO improves the I/O performance by 50 and 40 percent respectively over the conventional and alternative collective I/O implementation.

Notice that our strategy does not change the file domain partitioning phase in the traditional strategy, but only reorders the requests within each file domain. Hence it does not result in additional communication overhead because processes do not need to exchange requests. The additional cost of reorganizing the data within each aggregator's file domain is minuscule as this is a simple computation as shown in Section 3.2. Also each aggregator must determine what set of requests it must send, but again this is a slight communication overhead compared to the performance benefits of storage performance awareness.

3.2 Implementation

We implement the proposed HACIO strategy in the current MPI-IO library ROMIO [14]. To enable the proposed strategy, we first need to obtain the data layout information in the MPI process in the computing nodes. Fortunately, the file layout information, such as stripe size, stripe factor, and distribution policy, can be easily accessed using the underlying parallel file system's APIs. For example, this information is readily available in many commonly used parallel file systems, including PVFS2 [15], Lustre [16], and GPFS [17].

After obtaining the layout knowledge, all aggregators partition file domains and rearrange requests follow the HACIO design in each collective I/O operation. The request order of each aggregator is the core difference from the conventional collective I/O implementation. In the current collective I/O implementation in the MPI-IO library, the aggregator contains an array indexed by process rank. The aggregator sequentially walks through this array adding the offset and length pairs of I/O requests to the buffer. HACIO proposes a heterogeneity-aware request reordering algorithm that hops through this same array in a manner which all requests stored on a certain server are fulfilled before fulfilling the other server's requests.

Algorithm 1 shows the heterogeneity-aware request determination process of each aggregator. The goal of this algorithm is to reorder the I/O requests of each aggregator so that requests from all aggregators have a high priority to access the same type of servers in each cycle. To this end, the algorithm groups original requests of each aggregator according to the server type into two sets: HServer requests and SServer requests. It first inserts HServer requests into the temporal sequence S (lines 3 to 13) then inserts SServer requests. In this way, the load imbalance issue among different types of servers can be alleviated. Furthermore, the algorithm adjusts the request order for the same type of requests based on the server number where the request resides and the current aggregator rank (lines 22 to 25). By using a modulo operation, the algorithm tries to make aggregators issue requests to more of the same type of servers to increase I/O concurrency. The main addition our algorithm provides to the conventional implementation is lines 5 and 6. We pass the offset and length pair to functions called $Server\_type()$ and $Server\_number()$, which use parallel file system and hardware knowledge to return the server type (0 for HServer and 1 for SServer) and server number (0 to $hn-1$ for HServer and 0 to $sn-1$ for SServer) respectively. In a way, the two functions are the connection between the parallel file system and the storage layer in the parallel I/O stack. Thus, depending on the server type and server number, the algorithm will either append this request to the optimal request order or wait until the $Type$ and $number$ value changes.

Algorithm 1. Heterogeneity-Aware Request Reordering

```
Input: Processes number: $p$; The current aggregator rank: $a$;
       HServer number: $hn$; SServer number: $sn$; Request
       arrays in this aggregator's file domain indexed by pro-
       cess rank: $req[0], req[1], \dots, req[p-1]$; Offsets of a
       request indexed by request number: $of[0], of[1], \dots,$
       $of[req.count-1]$;
Output: Request sequence for this aggregator: $S$
1  /*Adjust the request order according to the server type to
   reduce load imbalance, type==0 indicates the server is an
   HServer*/
2  for ($type = 0; type < 2; type++$) do
3    for ($i = 0; i < p; i++$) do
4      for ($j = 0; j < req[i].count; j++$) do
5        if ($type == Server\_type(req[i].of[j])$) then
6          $number \leftarrow Server\_number(req[i].of[j])$
            /*Add original requests to the temporal sequence
           T */ ;
7          if ($req[i].of[j]$ not in $T[type][number]$)
           then
8            append $req[i].of[j]$ to $T[type][number]$ ;
9          end
10       end
11     end
12   end
13 end
14 /*Adjust the request order according to the server number
    to increase I/O concurrency*/ ;
15 for ($type = 0; type < 2; type++$)do
16   if ($type == 0$) then
17     $count \leftarrow hn$
18   end
19   else
20     $count \leftarrow sn$
21   end
22   for ($i = 0; i < count; i++$)do
23     $number \leftarrow (a+i)\%count$ ;
24     append $T[type][number]$ to $S$;
25   end
26 end
```

This decision process will allow all of the requests located on the same type of storage servers to be grouped together and requests located on the different servers to be issued concurrently. When conducting collective I/O operations, the aggregator can carry out I/O requests by accessing S sequentially. In this way, the aggregator will perform I/O operations on all servers of the same type before moving to a different type of server. Compared with conventional method, this solution can reduce the load imbalance and increase I/O concurrency among servers and thus likely improves the system performance.

In addition to the easily obtained inputs, our algorithm is extremely efficient, scaling linearly with the number of requests. Granted the original ordering strategy is a simple $O(1)$ computation, our method provides additional, yet minimal, computation overhead to drastically reduce execution time. For each aggregator, our algorithm creates the order of requests by iterating through each original request and positioning, based on its physical data location, the request into its appropriate position in the new request order.

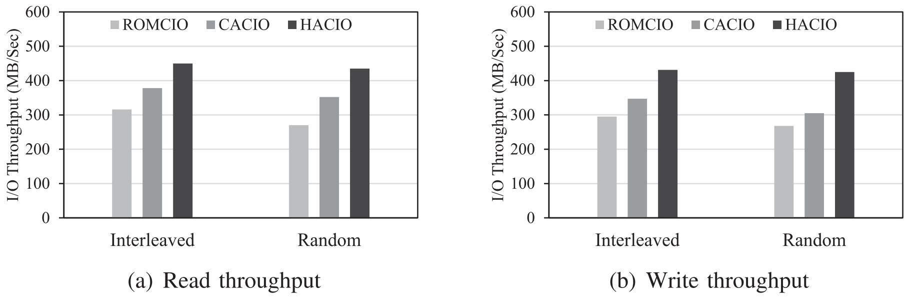

Fig. 8. Throughputs of IOR with different type of I/O operations.

4 Performance Evaluation

We implemented the HACIO strategy and the alternative CACIO strategy in ROMIO [14], a widely used MPI-IO library. In this section we compare HACIO with the existing ROMIO's collective I/O implementation (ROMCIO hereafter) and the alternative CACIO strategy. We first briefly describe the experimental environment and then present the experimental results.

4.1 Experimental Setup

We conduct the experiments on a 33-node SUN Fire Linux cluster. This cluster is composed of one Sun Fire X4240 head node, with dual 2.7 GHz Opteron quadcore processors and 8 GB memory, and 32 Sun Fire X2200 compute nodes with dual 2.3 GHz Opteron quad-core processors and 8 GB memory. Among the computing nodes, 16 nodes are equipped with OCZ-REVODRIVE 100 GB SSD, and each of the rest computing nodes has a 250 GB SATA hard drive. All 33 nodes are connected with Gigabit Ethernet. The operating system is Ubuntu 13.04, the MPI library is MPICH2-1.4.1p1, and the parallel file system is OrangeFS 2.8.6. Among the available nodes, we select 16 as client computing nodes, eight as HServers, and eight as SServers. By default, the heterogeneous OrangeFS file system is built on six HServers and two SServers. We run each test five times and the average is used as the performance result.

4.2 IOR Benchmark

IOR is a popular parallel file system benchmark developed at Lawrence Livermore National Laboratory [18]. It can mimic many I/O access patterns of real applications by changing various parameters. We use the MPI-IO interface to test the collective I/O operations. Unless otherwise specified, IOR runs with 32 processes, each of which performs I/O operations on a shared 16 GB parallel file with request size of 256 KB. We vary application characteristics and server configurations to verify the effectiveness of the proposed strategy.

4.2.1 Different Type of I/O Operations

First we compare HACIO against ROMCIO and CACIO under different types of I/O operations. Fig. 8 shows the throughputs of IOR under interleaved and random read and write requests. We observe that HACIO outperforms ROMCIO and CACIO for both read and write operations. Compared to ROMCIO, HACIO improves read performance up to 58.5 percent, and write performance up to 61.1 percent. In comparison with CACIO, HACIO has a read improvement up to 23.6 percent, and write improvement up to 39.3 percent. CACIO exceeds ROMCIO because it considers the physical data layout to increases concurrency while ROMCIO determines the request order only based on logical data information. However, HACIO has superior performance than CACIO because it further considers the performance disparity among heterogeneous servers.

4.2.2 Different Request Sizes

We also examine the I/O performance of HACIO, ROMCIO and CACIO with different request sizes. In these tests, the request size of IOR is varied from 32 to 1,024 KB. As Fig. 9 shows, HACIO can achieve better I/O performance compared to ROMCIO and CACIO with the increasing size of file requests. Compare to ROMCIO and CACIO, HACIO improves the read and write performance up to 197.7 and 260.6 percent with request size of 32 KB, and up to 46.8 and 56.9 percent with request size of 1,024 KB. As the request size increases, HACIO shows better I/O performance. This is because large requests causes the file domain size to also increase, thus there are more requests in each file domain to be reorganized to better utilize the hardware resources. These results validate that HACIO can choose appropriate request order to improve the heterogeneous I/O system performance as the request size varies.

4.2.3 Different Number of Processes

Next we measure and compare the performance of the three collective I/O strategies with respect to different number of processes. The IOR benchmark is executed under the interleaved access patterns with 16, 64 and 128 processes. As displayed in Fig. 10, HACIO improves the read performance by 12.5, 26.1, and 39.4 percent respectively with 16, 64 and 128 processes, and write performance by 13.1, 24.4, and 47.8 percent, compared to ROMCIO and CACIO. When the process number is small, the system has a higher

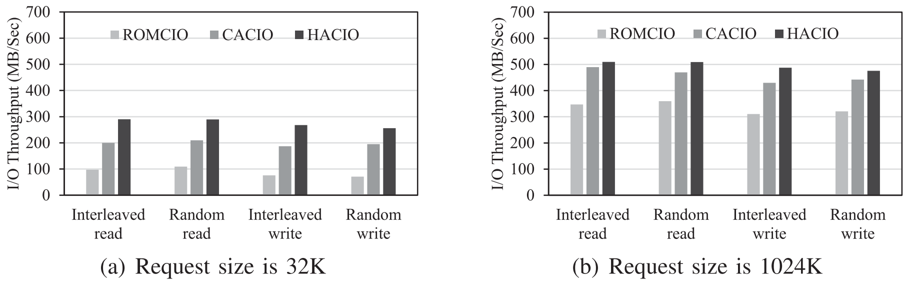

Fig. 9. Throughputs of IOR with different request sizes.

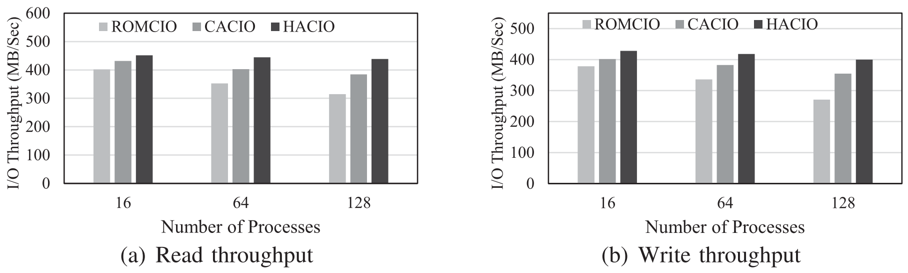

Fig. 10. Throughputs of IOR with different numbers of processes.

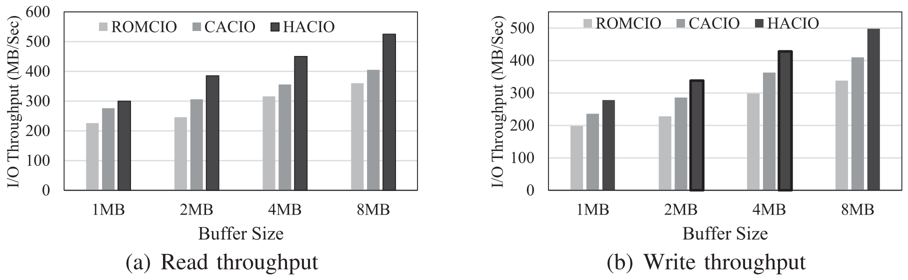

Fig. 11. Throughputs of IOR with different buffer sizes.

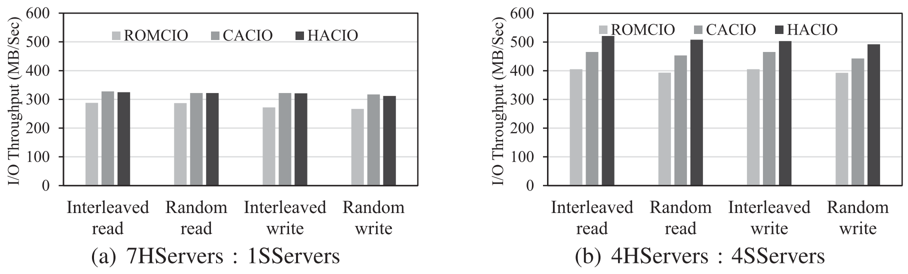

Fig. 12. Throughputs of IOR with different server configurations.

I/O bandwidth. As the number of processes increases, the performance of the heterogeneous PFS decreases because more processes lead to severe I/O contention in HServers and SServers, which degrades the overall system performance. However, in each case, HACIO is constantly better than ROMCIO and CACIO even when the number of process is 128. This is because HACIO considers the storage heterogeneity in the reordering the requests in each file domain, which can reduce I/O idle time of fast SServers. These results show that HACIO scales excellently with the number of I/O processes.

4.2.4 Different Buffer Sizes

Then we test the I/O performance with different buffer sizes. We change the collective I/O buffer size from 1 to 8 MB and run IOR with interleaved accesses. From Fig. 11, we observe that HACIO can improve I/O performance over the other two strategies. When the buffer size is small, the system has a low I/O performance because it needs more cycles to carry out the I/O request within a collective I/O operation, which means less hardware resources utilization. However, HACIO is still better than ROMCIO and CACIO because it reorders the requests based on the storage heterogeneity and aims to improve I/O concurrency on file servers. As the number of buffer size increases, the system performance increases because large requests enhance the server storage performance. We can also see that, as the buffer size increases, the improvement of HACIO is still better than CACIO strategy, which is mainly because HACIO can reduce the idle I/O time of SServers due to the better load balance it provides.

4.2.5 Different Server Configurations

Finally, we test the I/O performance with different server configurations. We varied the numbers of HServers and SServers with the ratios of 7:1 and 4:4. Fig. 12 shows the I/O bandwidth using the two collective I/O strategies. When the ratio of HServers to SServers is 7:1, we find that HACIO nearly has the same performance as CACIO strategy. HACIO can not bring large performance improvement because there is only one SServer so that the load imbalance among HServers and SServer is hard to alleviate. In this case, HACIO degrades to the CACIO strategy. As we can see, the performance gap is marginal, showing that the additional algorithm overhead of HACIO is small and negligible compared to CACIO. As the number of SServers increases, the system performance increases and HACIO has a large performance improvement. This is because there are more same type of servers to provide more chances for HACIO to get rid of the load imbalance among heterogeneous servers to optimize I/O performance.

4.3 HPIO Benchmark

HPIO is a benchmark designed by Northwestern University and Sandia National Laboratories to systematically evaluate I/O performance [19]. This benchmark can generate various data access patterns by changing three parameters: region count, region spacing, and region size, which indicates the number of requests, the distance between two requests, and the request size respectively. The region spacing is used to generate noncontiguous data access patterns. In our experiment, the number of process is set to 16

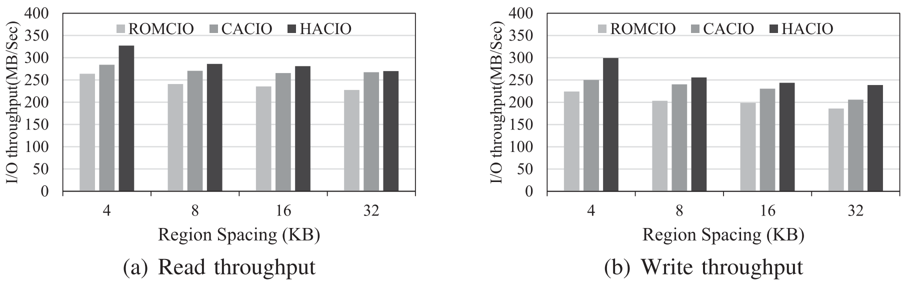

Fig. 13. Throughputs of HPIO with different region spacings.

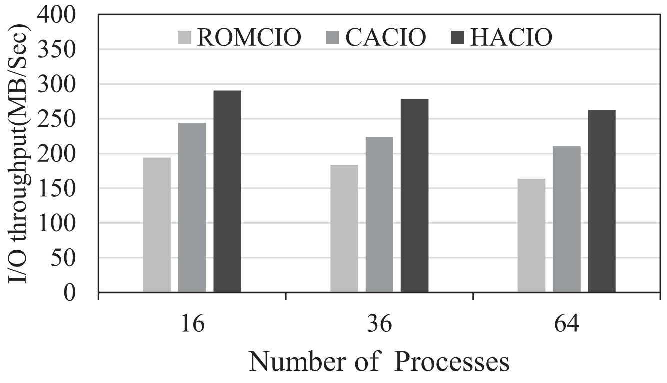

Fig. 14. Throughputs of BTIO with different number of processes.

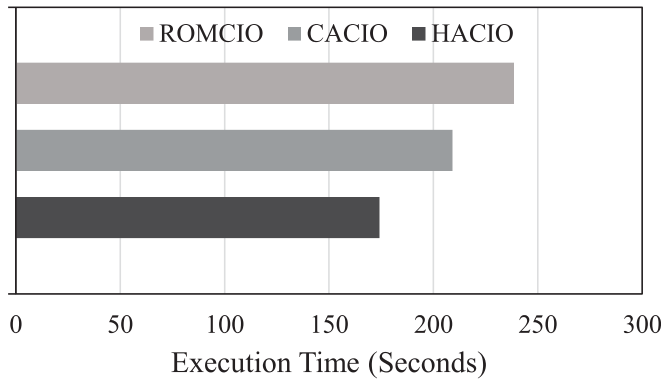

Fig. 15. Execution times of the application under two schemes.

processes; the region count is set to 8,192; the region size is set to 64 KB; and region spacing is varied from 4 to 32 KB.

As shown in Fig. 13a, HACIO can increase the read I/O throughput over the other two schemes by up to 23.9, 18.8, 19.3, and 17.5 percent respectively, which means that HACIO is effective with respect to HPIO benchmark. For write operations, the performance has similar trend as presented in Fig. 13b. This also confirms the adaptability of HACIO; when the application's I/O accesses have a poorer throughput (due to the poorer data sequential locality among consecutive accesses), more benefit is gained by using HACIO.

4.4 BTIO Benchmark

We also use the BTIO benchmark [20] from the NAS Parallel Benchmark (NPB3.3.1) suite to evaluate the proposed collective I/O strategy. BTIO represents a typical scientific application with interleaved intensive computation and I/O phases, and it uses a Block-Tridiagonal (BT) partitioning pattern to solve the three-dimensional compressible Navier-Stokes equations.

We configure BTIO benchmark with the Class C and full subtype, which means we write and read a total size of 6.64 GB data with collective I/O calls. We choose 16, 36, and 64 compute processes to conduct the experiments because BTIO requires a square number of processes. All the processes accesses a share parallel file on the heterogeneous OrangeFS file system. In the experiments, the collective buffer size is set to 4 MB, and the output data are distributed across six HServers and two SServers.

From Fig. 14, we observe that HACIO achieves better throughput and scalability compared to ROMCIO and CACIO. Compared to ROMCIO, HACIO achieves 49.6, 51.6, and 60.3 percent performance improvement with respect of 16, 36, 64 processes. In comparison with CACIO, HACIO also shows obvious performance improvements. The experimental results confirm that HACIO design can substantially improve the I/O performance of the heterogeneous file system.

4.5 Real Application

Finally, we evaluated HACIO with a real application, 'Anonymous LANL App 2' [21]. In these tests, the application ran with 512 processes, each issuing collective I/O requests in a non-uniform way at different parts of a shared file. The data accesses of this application were replayed according to the I/O trace to simulate the same data access scenario. We measured the performance of the application with HACIO strategy against ROMCIO and CACIO strategy. Fig. 15 reports the results of the application under the three collective-I/O strategies. Similar to the previous tests, HACIO can improve the performance by up to 26.7 percent compared to the other two policies.

All the experiment results have confirmed that the proposed HACIO strategy is a promising method to enhance the collective I/O technique for parallel I/O systems with heterogeneous servers. Hence it helps parallel I/O systems to provide efficient I/O service for data-intensive HPC applications.

5 Related Work

Collective I/O is the most widely used I/O middleware approach to boost I/O performance for multiple processes of a parallel program [8]. It merges small noncontiguous requests of multiple processes and performs large contiguous I/O operations on a subset of processes called aggregators. Traditional collective I/O strategies are based on request logical addresses, Resonant I/O [10] is a physical layout-aware collective I/O strategy, which ensures each disk to serve file requests in an order consist with the offsets of the requested data in the file domain. Chen et al. [11] propose another layout-aware collective I/O, named LACIO, which rearranges the partitions of file domains to make each disk to serve file requests from less aggregators, to reduce access contentions and increase data locality. Wang et al. [12] propose an iteration-based collective I/O strategy, which reorganizes I/O requests within each file domain instead of coordinating requests across file domains to reduce shuffle cost and disk I/O contention. To alleviate lock contentions, Liao et al. [13] propose a file partition method that allows file domains to be aligned to the lock boundary of file systems. Dickens et al. [22] propose to minimize the number of I/O nodes with which an aggregator communicates to increase the collective I/O performance. Chaarawi et al. [9] introduce an algorithm to select the number of aggregators based on consideration of process topology, file view, and data amount in a collective I/O operation.

Although various optimizing approaches exist, they are designed for homogeneous I/O systems. In contrast, HACIO aims to optimize collective I/O operations in heterogeneous parallel I/O systems with different types of servers.

6 Conclusion

We have proposed a heterogeneity-aware collective I/O strategy, HACIO, for heterogeneous I/O systems. The key idea of HACIO is to reorder I/O requests in each file domain with awareness of storage performance disparity among various file servers. In essence, HACIO provides a better request stream reorganization that matches both data access characteristics of applications and storage capabilities of underlying file servers. We have developed and presented the proposed HACIO collective I/O strategy in ROMIO. Experimental results show that HACIO outperforms the existing collective I/O strategies.

Acknowledgments

The authors are thankful to Adnan Haider from Illinois Institute of Technology for his constructive suggestions toward this work. This study is supported partly by NSFC No. 61572377, NSF of Hubei Province of China under Grant No. 2014CFB239, the Open-fund from HPCL under Grant No. 201512-02, and the Open Fund from SKLSE under Grant No. 2015-A-06. Yang Wang is the corresponding author.

References

[1] Y. Kim, S. Atchley, G. R. Vallée, and G. M. Shipman, "LADS: Optimizing data transfers using layout-aware data scheduling," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 67–80.
[2] S. He, X.-H. Sun, and B. Feng, "S4D-cache: Smart selective SSD cache for parallel I/O systems," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 514–523.
[3] J. Ou, J. Shu, Y. Lu, L. Yi, and W. Wang, "EDM: An endurance-aware data migration scheme for load balancing in SSD storage clusters," in *Proc. 28th IEEE Int. Parallel Distrib. Process. Symp.*, 2014, pp. 787–796.
[4] 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.
[5] 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.
[6] 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.
[7] 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.
[8] 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.
[9] M. Chaarawi and E. Gabriel, "Automatically selecting the number of aggregators for collective I/O operations," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2011, pp. 428–437.
[10] 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 *Proc. 23rd IEEE Int. Parallel Distrib. Process. Symp.*, 2009, pp. 1–12.
[11] Y. Chen, S. Xian-He, R. Thakur, P. C. Roth, and W. D. Gropp, "LACIO: A new collective I/O strategy for parallel I/O systems," in *Proc. 25th IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 794–804.
[12] Z. Wang, X. Shi, H. Jin, S. Wu, and Y. Chen, "Iteration based collective I/O strategy for parallel I/O systems," in *Proc. 14th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2014, pp. 287–294.
[13] W.-k. Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2008, pp. 1–12.
[14] R. Thakur and A. Choudhary, "An extended two-phase method for accessing sections of out-of-core arrays," *Sci. Program.*, vol. 5, no. 4, pp. 301–317, 1996.
[15] P. H. Carns, I. Walter, B. Ligon, R. B. Ross, and R. Thakur, "PVFS: A parallel virtual file system for linux clusters," in *Proc. 4th Annu. Linux Showcase Conf.*, 2000, pp. 317–327.
[16] S. Microsystems, "Lustre file system: High-performance storage architecture and scalable cluster file system," Lustre File System, White Paper, 2007.
[17] 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.
[18] Interleaved Or Random (IOR) Benchmarks, 2014. [Online]. Available: http://sourceforge.net/projects/ior-sio/
[19] 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, pp. 69–69.
[20] The NAS parallel benchmarks, 2014. [Online]. Available: www.nas.nasa.gov/publications/npb.html
[21] Application I/O Traces: Anonymous LANL App2, 2014. [Online]. Available: http://institutes.lanl.gov/plfs/maps/
[22] P. M. Dickens and J. Logan, "Y-lib: A user level library to increase the performance of MPI-IO in a lustre file system environment," in *Proc. 18th ACM Int. Symp. High Perform. Distrib. Comput.*, 2009, pp. 31–38.