Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols

Wei-keng Liao and Alok Choudhary
Electrical Engineering and Computer Science Department
Northwestern University
Evanston, Illinois 60208-3118
Email: {wkliao,choudhar}@ece.northwestern.edu

***Abstract*—Collective I/O, such as that provided in MPI-IO, enables process collaboration among a group of processes for greater I/O parallelism. Its implementation involves file domain partitioning, and having the right partitioning is a key to achieving high-performance I/O. As modern parallel file systems maintain data consistency by adopting a distributed file locking mechanism to avoid centralized lock management, different locking protocols can have significant impact to the degree of parallelism of a given file domain partitioning method. In this paper, we propose dynamic file partitioning methods that adapt according to the underlying locking protocols in the parallel file systems and evaluate the performance of four partitioning methods under two locking protocols. By running multiple I/O benchmarks, our experiments demonstrate that no single partitioning guarantees the best performance. Using MPI-IO as an implementation platform, we provide guidelines to select the most appropriate partitioning methods for various I/O patterns and file systems.**

I. Introduction

A majority of scientific parallel applications nowadays are programmed to access files in a one-file-per-process style [1]. This programming style is simple and often gives satisfactory performance when applications run on a small number of processes. One immediate drawback is that the application restart must use the same number of processes as the run that produced the checkpoint files. A more serious problem is that this method can create a management nightmare for file systems when applications run on a large number of processes. A single production run using thousands of processes can produce hundreds of thousands or millions of files. Simultaneous file creation in such a scale can cause the network traffic congestion at the metadata servers, as modern parallel file systems employ only one or a small number of metadata servers. Furthermore, accessing millions of newly created files becomes a daunting task for post-run data analysis. In order to reduce such file management workload, one solution is to adopt the shared-file I/O programming style.

Shared-file I/O provides a way to preserve the canonical order of structured data. Parallel programs often use global data structures, such as multi-dimensional arrays, to present the problem domain and partition them so that processes can concurrently operate on the assigned sub-domains. During a checkpoint, maintaining arrays' canonical order in files can ease up the task of post-run data analysis and visualization. However, shared-file I/O often performs poorly when the requests are not well coordinated. To address this concern, the message passing interface (MPI) standard defines a set of programming interfaces for parallel file access, commonly referred as MPI-IO [2]. There are two types of functions in MPI-IO: collective and independent. The collective functions require process synchronization which provides an MPI-IO implementation an opportunity to collaborate processes and rearrange the requests for better performance. Well-known examples of using such a collaboration are two-phase I/O [3] and disk directed I/O [4]. Process collaboration has demonstrated significant performance improvements over uncoordinated I/O. However, even with these improvements, the shared-file I/O performance is still far from the single-file-per-process approach. Part of the reason is that shared-file I/O incurs higher file system locking overhead from data consistency control, which can never happen if a file is only accessed by a unique process.

ROMIO is a popular MPI-IO implementation developed at Argonne National Laboratory [5]. It has been incorporated as part of several MPI implementations, including MPICH, LAM [6], HP MPI, SGI MPI, IBM MPI, and NEC MPI. To achieve the portability, ROMIO implements a layer of abstract-device interface named ADIO that contains a set of I/O drivers, one for a different file system [7]. This design allows ADIO to utilize the system dependent features for higher I/O performance. ROMIO's collective I/O implementation is based on the two-phase I/O strategy proposed in [3], which includes a data redistribution phase and an I/O phase. The two-phase strategy first calculates the aggregate access file region and then evenly partitioned it among the I/O aggregators into **file domains**. The I/O aggregators are a subset of the processes that act as I/O proxies for the rest of the processes. In the data redistribution phase, all processes exchange data with the aggregators based on the calculated file domains. In the I/O phase, aggregators access the shared file within the assigned file domains. Two-phase I/O can combine multiple small non-contiguous requests into large contiguous ones and

SC2008 November 2008, Austin, Texas, USA 978-1-4244-2835-9/08 $25.00 ©2008 IEEE

has demonstrated to be very successful, as modern file systems handle large contiguous requests more efficiently. However, the even partitioning method does not necessarily produce the best I/O performance on all file systems.

Modern parallel file systems employ multiple I/O servers, each managing a set of disks, in order to meet the requirement of high data throughput. Files stored on these systems can be striped across the I/O servers so that large requests can be concurrently served. For parallel I/O on striped files, it is not easy to enforce data consistency and provide high performance I/O at the same time. Two most important data consistency issues that file systems must enforce are I/O atomicity and cache coherence. Most file systems rely on a locking mechanism to provides a client an exclusive access to a file region and hence to implement the data consistency control. Due to the nature of file striping, lock granularity is usually the file block size or stripe size, instead of a byte. If two I/O requests simultaneously access the same file block and at least one of them is a write, they must be carried out serially, even if they do not overlap in bytes. On file systems that perform client-side file caching, this situation can also cause false sharing in which a block is cached by a process but flushed immediately, so the block can be accessed by the other process. Both I/O serialization and false sharing could happen in a collective I/O, if the partitioned file domains are not aligned with the lock boundaries.

In this paper, we investigate three file domain partitioning methods in addition to the even partitioning method used by ROMIO. The first method aligns the partitioning with the file system's lock boundaries. The second method, named static-cyclic method, partitions a file into fixed-size blocks based on the lock granularity and statically assigns the blocks in a round-robin fashion among the I/O aggregators. The third method, named group-cyclic method, divides the I/O aggregators into groups, each being of size equal to the number of I/O servers. Within each group, the static-cyclic partitioning method is used. This method is particularly designed for the situation that the number of I/O aggregators is much larger than the I/O servers. We evaluate these methods using four I/O benchmarks on two parallel file systems, Lustre and GPFS. Due to the different file locking protocols adopted in Lustre and GPFS, these partitioning methods result in significant performance differences on the two file systems. Our experiments conclude that the group-static and lock-boundary aligned methods give the best write performance on Lustre and GPFS, respectively. We analyze these behaviors and propose a strategy that dynamically chooses the partitioning method best fit to the underlying file system locking protocol.

The rest of the paper is organized as follows. Section II discusses background information and related work. Design and implementation of the file domain partitioning methods are described in Section III. Performance results are presented and analyzed in Section IV and the paper is concluded in Section V.

II. Background and Related Work

MPI-IO inherits two important MPI features: MPI communicators defining a set of processes for group operations and MPI derived data types describing complex memory layouts. A communicator specifies the processes that can participate in a collective operation for both inter-process communication and file I/O. When opening a file, the MPI communicator is a required argument to indicate the group of processes accessing the file. MPI collective I/O functions also require all processes in the communicator to participate. Such an explicit synchronization allows a collective I/O implementation to exchange access information among all processes and reorganize I/O requests for better performance. Independent I/O functions, in contrast, requiring no synchronization make any collaborative optimization very difficult. While the MPI-IO design goals are mainly for parallel I/O operations on shared files, one can still program in the one-file-per-process style using the `MPI_COMM_SELF` communicator, but it provides no benefit over using POSIX I/O directly.

A. Two-phase I/O Implementation in ROMIO

Two-phase I/O is a representative collaborative I/O technique that runs at user space. It assumes that file systems handle large contiguous requests much better than small non-contiguous ones. ROMIO implements the two-phase I/O for all the collective functions. It first calculates the aggregate access region, a contiguous file region starting from the minimal access offset among the requesting processes and ending at the maximal offset among the processes. The aggregate access region is then divided into non-overlapping, contiguous sub-regions denoted as file domains, and each file domain is assigned to a unique process. A process makes read/write calls on behalf of all processes for the requests located in its file domain. In ROMIO's current implementation, the file domain partitioning is done evenly at the byte range granularity in consideration of balancing the workload.

The two-phase method is generalized in ROMIO by taking two user-controllable parameters: the I/O aggregators and the collective buffer size [8]. Both parameters can be set through MPI info objects using hints `cb_nodes` and `cb_buffer_size`. The I/O aggregators are a subset of the processes that act as I/O proxies for the rest of the processes. On the parallel machines where each compute nodes contains a multi-core CPU or multiple processors, the ROMIO default picks one of the core/processor as the aggregator each node. Only aggregators make system calls, such as `open()`, `read()`, `write()` and `close()`. The collective buffer size indicates the space of temporary buffers that can be used for data redistribution. It is useful for memory-bound applications where spare memory space is limited. When a file domain is bigger than the collective buffer size, the collective I/O will be carried out in multiple steps of two-phase I/O and each two-phase I/O operates on a file sub-domain of size no larger than the collective buffer size. Figure 1 shows a two-dimensional array of size 10×15 partitioned among six processes in a block-block fashion and the array is written in the array's

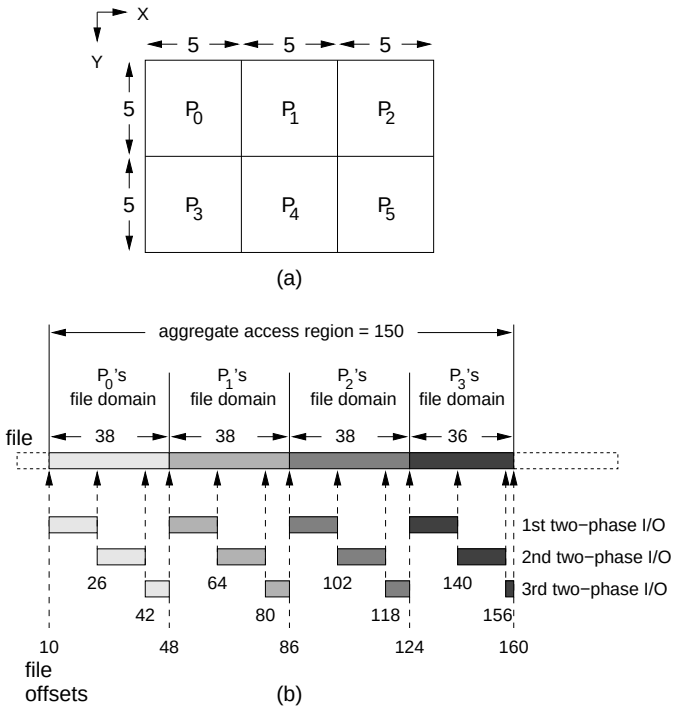

Fig. 1. (a) A 10×15 array stored in a file in a row major is partitioned among six processes. Each subarray is of size 5×5. (b) The aggregate access region is calculated and partitioned evenly among the I/O aggregators. Each aggregator is assigned a contiguous region as its file domain. In this example, the first four processes are picked as the I/O aggregators. Given the collective buffer size of 16 bytes, a collective I/O operation is carried out in 3 steps of two-phase I/O. Each step covers a file region equal or less than 16 bytes.

canonical order to a shared file starting from file offset 10. In this example, the I/O aggregators are set to be the first four processes. Under the even partitioning policy, file domains are non-overlapping contiguous file regions of size 38 bytes for processes P_0, P_1, and P_2 and 36 bytes for process P_3. If a collective buffer size of less than the file domain size, say 16 bytes, the collective I/O will be completed by running two-phase I/O three times. In each two-phase I/O, an aggregator only handles data redistribution for a file region equal or less than 16 bytes.

Nitzberg and Lo studied three file domain-partitioning methods for two-phase I/O, namely block, file layout, and cyclic target distributions [9]. The block method uses all processes as I/O aggregators and chooses an unlimited collective buffer size so that the two-phase I/O can be completed in a single step. The file layout method uses the number of I/O aggregators equal to the number of system I/O servers and the data redistribution matches the file striping layout. The two-phase I/O is carried out in rounds, each round processing the aggregate access region of size equal to the collective buffer size times the number of aggregators. In each round, an aggregator will make n read/write calls to the file system, where n is the number of stripe units in a collective buffer. The cyclic method is generalized from the file layout method, which allows file domain size to be set by the users to a multiple of file stripe size. Their experiments showed all three methods perform competitively and with selected collective buffer sizes the cyclic method can outperform others in some cases. They concluded that no single partitioning method provides the best performance and the performance varies depending on the I/O patterns. However, there is no analysis from the file system perspective on why these methods behave differently, but merely a performance observation. The file-layout and cyclic methods are similar to our static-cyclic partitioning method presented in this paper. The difference is that we consider the file system's lock granularity and our motivation came from the idea of how to minimize the lock contentions.

B. File Locking in Parallel File Systems

Many modern parallel file systems are POSIX compliant, which abide by the data consistency rules that were designed under the traditional non-parallel environment. It has been known that POSIX requirements on I/O consistency and atomicity are the two main factors causing significant performance degradation for parallel shared-file I/O [10], [11]. Data consistency requires the outcomes of concurrent I/O operations as if they were carried out in a certain linear order. It is relatively easy for a file system with one server to guarantee the sequential consistency, but much difficult for parallel file systems where files are striped across multiple servers. The atomicity requires that the results of an individual write call are either entirely visible or completely invisible to any read call [12]. Implementation for both requirements can become further sophisticate when client-side file caching is performed. Two well-known parallel file systems support file caching are the IBM's GPFS [13], [14] and Lustre [15].

Currently a popular solution for I/O atomicity and cache coherency uses a locking mechanism to provide a process the exclusive access privilege to the requested file region. However, exclusive access can potentially serialize concurrent operations. Especially, as the number of processors goes into the scale of thousands or millions, guaranteeing such consistency without degrading the parallel I/O performance is a great challenge. To avoid the obvious bottleneck from a centralized lock manager, various distributed file locking protocols have been proposed. For example, GPFS employs a distributed token-based locking mechanism to maintain coherent caches across compute nodes [16]. This protocol makes a token holder a local lock authority for granting further lock requests to its corresponding byte range. A token allows a node to cache data that cannot be modified elsewhere without first revoking the token. GPFS's file stripe size is set at the system boot time and not changeable by users. The Lustre file system uses a different distributed server-based locking protocol where each I/O server manages locks for the stripes of file data it stores. Unlike GPFS, users can customize striping parameters for a file on Lustre, such as stripe count, stripe size, and the starting I/O server. If a client requests a lock held by another client, a message is sent to the lock holder asking it to release the lock. Before a lock can be released, dirty cache data must be flushed to the servers. To guarantee atomicity, file locking is used in each read/write call to guarantee exclusive access to

the requested file region.

Both GPFS and Lustre adopt an extent-based locking protocol in which a lock manager tends to grant a request as the largest file region as possible. For example, the first requesting process to a file will be granted the lock for entire file. When the second write from a different process arrives, the first process will relinquish part of the file to the requesting process. If the starting offset of the second request is bigger then the first request's ending offset, the relinquished region will start from the first request's ending offset toward the end of file. Otherwise, the relinquished region will contain a region from file offset 0 to the first request's staring offset. The advantage of this protocol is if a process's successive requests are within the already granted region, then no lock request is needed. The extent-based protocol is carried out by the lock manager on both GPFS and Lustre. On GPFS, the lock token holder is the lock manager and hence the extent of a lock can virtually cover the entire file. On Lustre, since an I/O server is the lock manager for the file stripes stored in that server, the extent of a lock can only cover those file stripes.

III. Design and Implementation

There is no doubt that process collaboration is a key for high-performance I/O. In addition to the two-phase I/O and disk-directed I/O, many collaboration strategies have been proposed and demonstrated their success, including server-directed I/O [17], persistent file domain [18], [19], active buffering [20], collaborative caching [21], [11], etc. In this paper, we focus on the two-phase I/O method implementation in ROMIO. The primary idea of two-phase I/O assumes that file access cost is much higher than the inter-process communication. This assumption is still reasonable for the configuration of today's parallel machines where the I/O servers are much less than the compute nodes. In addition to the potential network congestion on the servers, the disk's slow latency and file system's overhead on data consistency and cache coherence controls also attribute to the higher I/O cost.

The significance of such file system control costs will become clear as we examine how a file system reacts differently to the one-file-per-process and shared-file I/O styles. Both styles deal with concurrent I/O requests to the file system, but only the shared-file method bears the consistency control cost. The one-file-per-process style does not introduce any lock conflicts and hence causes no I/O serialization. Its cost for acquiring locks is even smaller when the extent-based locking protocol is used. On the other end, multiple locks to the same file must be resolved by the file system in the shared-file I/O style and the more concurrent I/O requests, the higher cost of the lock acquisition. If two I/O requests overlap and at least one is a write request, the I/O will be serialized, which further worsens the I/O performance. Therefore, avoiding lock conflicts is very important and the first step for a two-phase I/O implementation to achieve better performance.

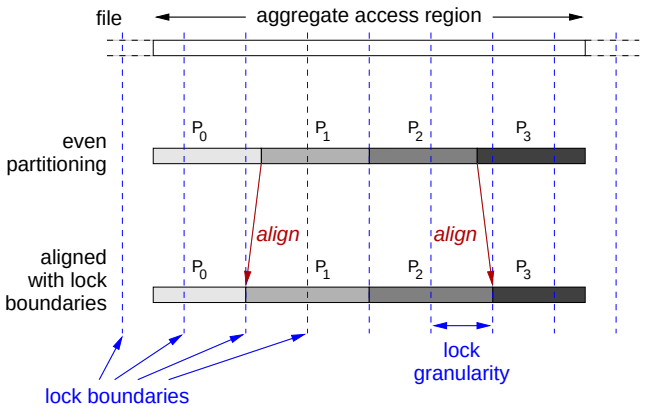

Fig. 2. File domains produced by the even and lock-boundary aligned partitioning methods.

A. Partitioning Aligned with Lock Boundaries

The lock granularity of a file system is the smallest size of file region a lock can protect. For single-disk file systems, it is the disk sector size. For file systems using a single RAID disk, it is the sector size times the number of redundant disks. For most of the parallel file systems, such as GPFS and Lustre, it is set to the file stripe size. Reasonably good parallel I/O performance has been seen from many parallel I/O benchmarks that used the I/O sizes being multiples of stripe sizes to avoid conflicts at lock granularity level. However, this perfect alignment does not always happen in real applications. In the two-phase I/O implementation, although the even partitioning method generates non-overlapping file domains at the byte level, it can still cause lock contentions at the lock granularity level. To avoid such contentions, the simplest method is to align each partitioning to a lock boundary. As depicted in Figure 2, our implementation aligns the partitioned boundary of two file domains to the nearest lock boundary. Similar approaches have been proposed and demonstrated performance enhancement on several benchmarks for both Lustre and GPFS file systems [22], [11].

B. Static-cyclic Partitioning

The static-cyclic partitioning method divides the entire file into equal-size blocks and assigns the blocks to the I/O aggregators in a round-robin fashion. The block size is set to the file system's lock granularity. The association of a block to an aggregator does not change from one collective I/O to another. For instance, given n I/O aggregators, blocks $i, i+n, i+2n, \cdots$ are assigned to aggregator rank i. We refer these blocks as process i's partitioning fileview, which is similar to the MPI fileview concept that defines the file regions visible to a process. Note that in the even and aligned partitioning methods, file domains only exist in the current collective I/O call and must be redefined in every collective I/O. In the static-cyclic method, the partitioning fileview of an aggregator does not change from one collective I/O to another. If the lock granularity is the same size as the file stripe and there is a common divisor between the number of I/O servers

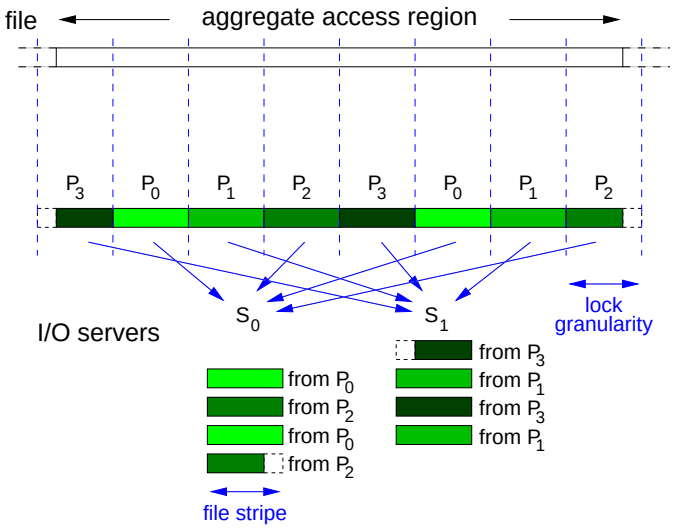

Fig. 3. Static-cyclic partitioning method. When the lock granularity is equal to the file stripe size, process P_0 and P_2 always communicate with I/O server S_0, and P_1 and P_3 always communicate with S_1.

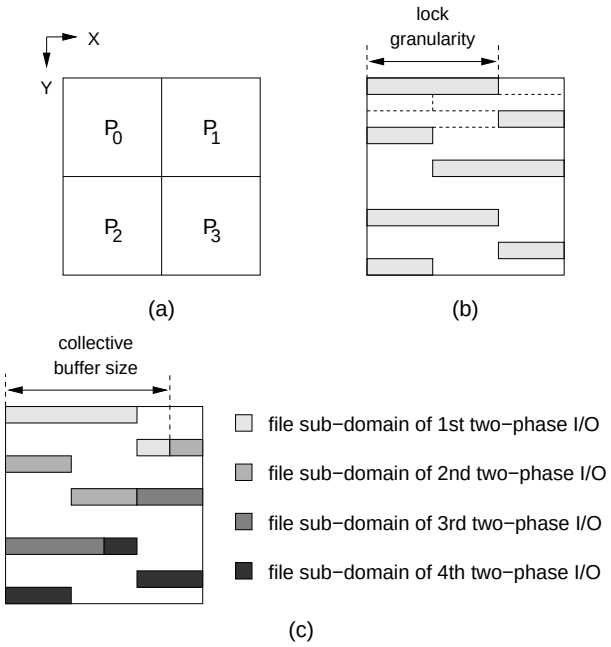

Fig. 4. (a) Data partitioning of a 2D array among four processes. It also represents the processes' MPI fileviews. The 2D array is stored the file in a row major. (b) The gray area is process 0's file domain generated by the static-cyclic partitioning method. (c) Process 0's file domain is further divided into four sub-domains, given the collective buffer size is one-forth of the file domain size. The collective I/O is carried out in four steps of two-phase I/O, each using a sub-domain.

and aggregators, each aggregator will always communicate with the same set of servers. Figure 3 depicts an example of the partitioning fileviews and file domains partitioned by the static-cyclic method. If persistent communication channels can be established between compute processor and I/O servers, this method can further reduce the network cost across multiple collective I/Os.

Compared with the even and aligned methods where each file domain is a contiguous region, the implementation of static-cyclic method is more complicate, especially when the collective buffer size is small. Given a collective I/O, what an aggregator will access is the intersection of its partitioning fileview and the collective I/O's aggregate access region. Although an aggregator's file domain still does not overlap with another aggregator like the other two partitioning methods, it is no longer a single contiguous file region. The size of a file domain is the sum of the coalesced strided blocks an aggregator is responsible within the aggregate access region. If the file domain size is larger than the collective buffer size, the collective I/O is decomposed into multiple steps of two-phase I/O. In each step, a file sub-domain is covering a subset of blocks whose coalesced size is equal or less than the collective buffer size. Figure 4 illustrates an example of an aggregator's file domain and sub-domains. For instance, after the redistribution phase of a collective write, the collective buffer contains non-contiguous data blocks spanning across the aggregate access region. There will be one write call for each of the blocks. Thus, the number of read/write calls made by each aggregator is more than the even and aligned partitioning methods. Apparently a performance trade-off exists, depending on how well a file system can handle such a request pattern.

C. Group-cyclic Partitioning

When the number of I/O aggregators is much larger than the number of I/O servers, the static-cyclic method may cause higher lock acquisition cost if the underlying file system uses the server-based locking protocol. In the example shown in Figure 3, there are four I/O aggregators and two I/O servers. In the static-cyclic method, although server S_0 only receives requests from processes P_0 and P_2, the file stripes accessed by the two processes are interleaved. Similarly, the file stripes accessed by processes P_1 and P_3 are also interleaved at server S_1. In this case, if the extent-based locking protocol is used, lock requests to each of the interleaved stripes must be resolved by remote processes. Such lock acquisition pattern can be harmful to the performance.

To avoid the interleaved access, the group-cyclic partitioned method divides the I/O aggregators into groups, each of size equal to the number of I/O servers. The aggregate access region of a collective I/O is then divided among the groups with the boundaries aligned to the file stripe size. Within each group, the static-cyclic method is used. Figure 5 illustrates an example of the group-cyclic partitioning method using eight I/O aggregators and four I/O servers. The first group, group 0, contains aggregators 7, 0, 1, and 2. Group 1 includes aggregators 3, 4, 5, and 6. The aggregator rank alignment is based on the starting file offset of the aggregate access region. The starting aggregator rank, 7 in this example, is calculated by the formula

$$\lfloor \frac{\text{starting offset}}{\text{stripe size}} \rfloor \ \textbf{mod} \ \text{np} \qquad (1)$$

where np is the number of aggregators. The grouping is made

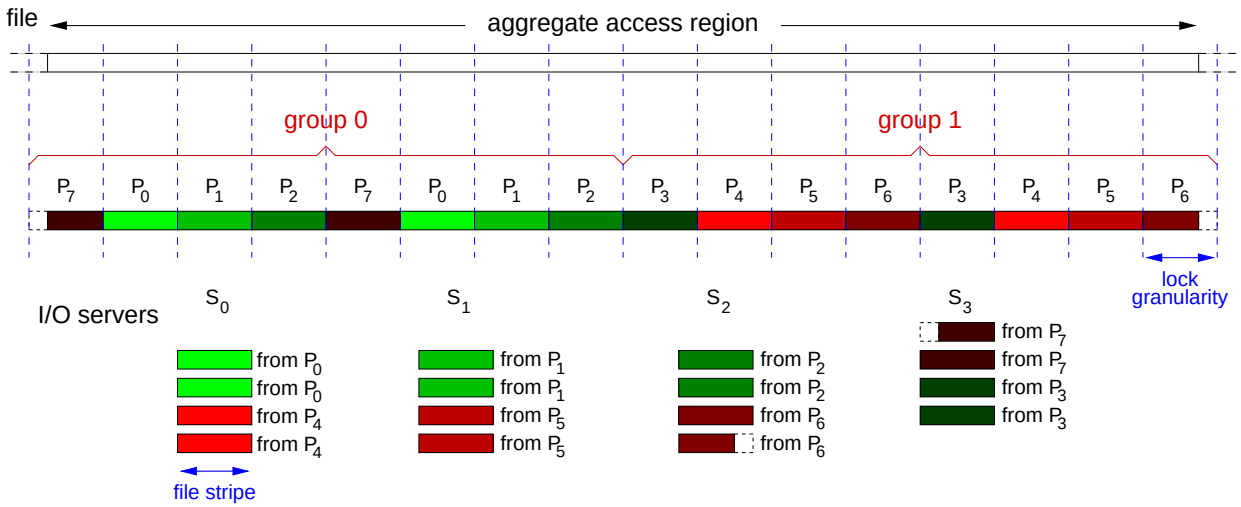

Fig. 5. Group-cyclic partitioning method. The I/O aggregators are divided into 2 groups, each of size equal to the number of servers. In this example, the file stripes accessed by P_0 have file offsets prior to the ones by P_4 at server S_0. Similarly, the stripes accessed by P_1 have offsets prior to the stripes by P_5 at server S_1, and so on.

in the continuous, round-robin aggregator rank order. Within each group, an aggregator will only make requests to one I/O server. Since group 0 covers the file region prior to group 1 and no aggregator is assigned to two groups, the interleaved access is eliminated. Under the server-based locking protocol, file stripes requested by a process are considered contiguous by the I/O server. Unlike the static-cyclic method, the association of file stripes to the I/O aggregators is no longer static across multiple collective I/O operations. However, the association of I/O servers to the aggregators is still static, if the numbers of the servers is a factor or multiple of the the aggregators. In other words, the group-cyclic method is static at the I/O server level while the static-cyclic method is static at the file stripe level. Note that the group-cyclic method only takes effective when the number of I/O aggregators is greater than the number of I/O servers. Otherwise, it operates exactly the same as the static-cyclic method.

IV. Experimental Results

Our performance evaluation was conducted on two parallel machines: Jaguar at the National Center for Computational Sciences and Mercury at the National Center for Supercomputing Applications. Jaguar is a 7832-node Cray XT4 cluster running Compute Node Linux operating system. Each of the compute nodes contains a quad-core 2.1 GHz AMD Opteron processor and 8 GB of memory. The communication network is a Cray SeaStar router through a bidirectional HyperTransport interface. The parallel file system is Lustre with a total of 144 object storage targets (I/O servers). Lustre allows users to customize the striping configuration of a directory and all new files created in that directory inherit the striping configuration. In our experiment, we configure a directory to store all output files with 512 KB stripe size, 64 stripe count (number of I/O servers), and the start server to be randomly picked by the file system. On Lustre, the lock granularity is the stripe size, 512 KB in our experiments. Mercury, a TeraGrid Cluster, is an 887-node IBM Linux cluster where each node contains two Intel 1.3/1.5 GHz Itanium II processors sharing 4 GB of memory. Running a SuSE Linux operating system, the compute nodes are inter-connected by both Myrinet and Gigabit Ethernet. Mercury runs an IBM GPFS parallel file system version 3.1.0 configured in the Network Shared Disk (NSD) server model with 54 I/O servers and 512 KB file block (stripe) size. Unlike Lustre, users cannot change the file striping parameters on GPFS. The lock granularity on GPFS is also the stripe size, 512 KB in our case. The MPI library installed on Mercury is MPICH version 1.2.7p1 configured with Myrinet.

We developed the proposed I/O methods in the ROMIO source codes from the MPICH package developed at Argonne National Laboratory. On Jaguar, we extracted the ROMIO package from the MPICH2 release of version 1.0.7 and on Mercury we used the ROMIO from MPICH version 1.2.7p1. We configured the ROMIO by enabling the ADIO Unix file system driver for both Lustre and GPFS and built the ROMIO as a stand-alone library separately from the MPICH. The library is then linked with the native MPI library on the two machines when generating application executable binaries. For performance evaluation, we use two artificial benchmarks, ROMIO collective I/O test and BTIO, and two I/O kernels from production applications, FLASH and S3D. The bandwidth numbers were obtained by dividing the aggregate I/O amount by the time measured from the beginning of file open until after file close.

A. ROMIO Collective I/O Test

ROMIO software package includes a set of test programs in which the collective I/O test, named `coll_perf`, writes and reads a three-dimensional integer array that is block partitioned along all three dimensions among processes. An example of its partitioning pattern on 64 processes is illustrated in

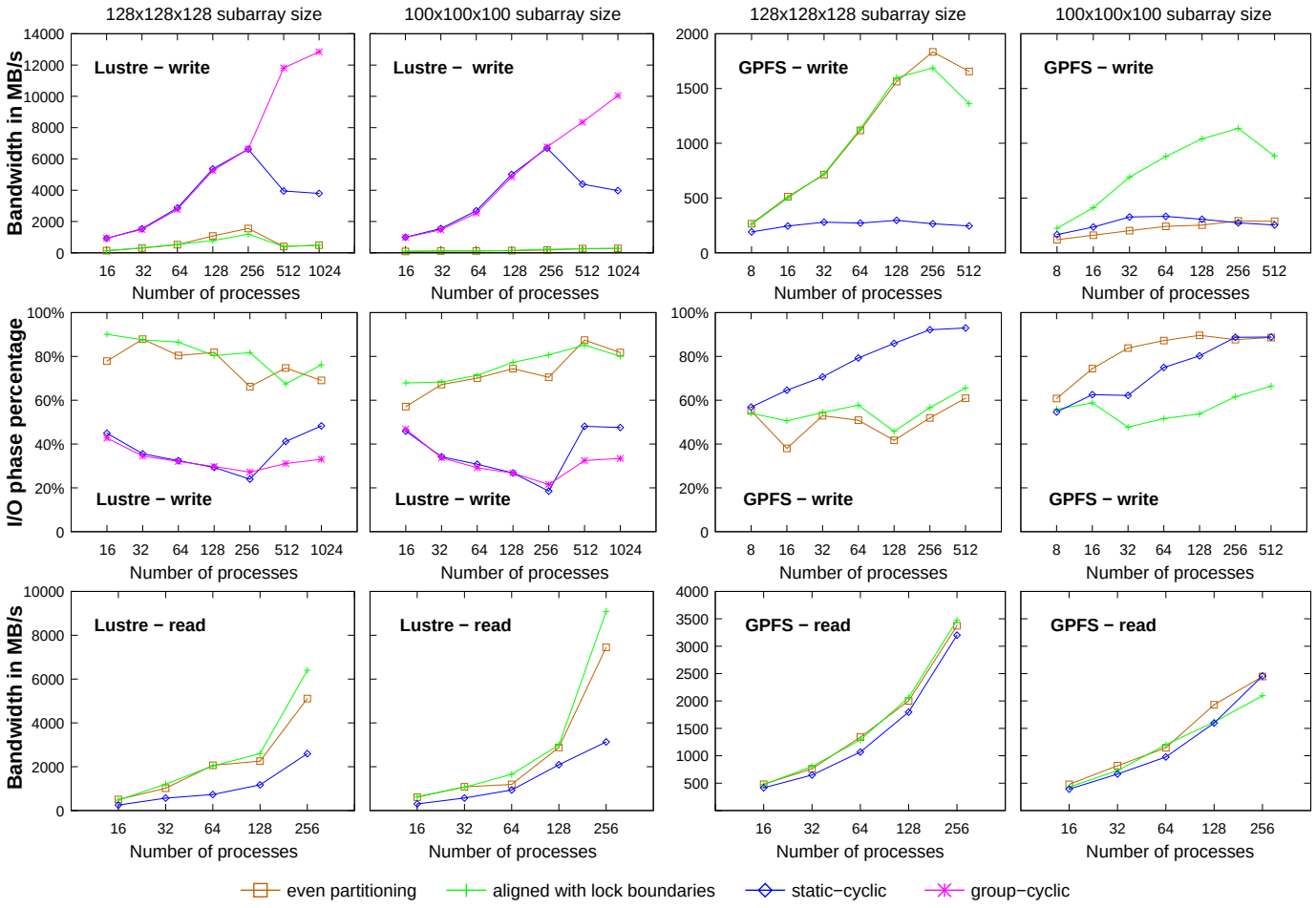

Fig. 6. Performance results of ROMIO collective I/O test.

Figure 11(a). In order to get stable performance numbers, we measured ten iterations of the collective operations. The subarray size in each process is kept constant, independent from the number of processes used, and hence the total I/O amount is proportional to the number of processes. We choose two sets of subarray size: $128 \times 128 \times 128$ and $100 \times 100 \times 100$. The $128 \times 128 \times 128$ size allows the even partitioning method to generate some file domains aligned with the file lock boundaries. The $100 \times 100 \times 100$ subarray size is chosen to make the unaligned case. The experimental results are shown in Figure 6.

On Lustre, the write bandwidths for both even and aligned partitioning methods are similar, but significantly lower than the cyclic methods. On GPFS, for $128 \times 128 \times 128$ subarray size, the even and aligned methods are close to each other and both are much better than the static-cyclic method. For the $100 \times 100 \times 100$ subarray size, bandwidths of the even method drop close to the static-cyclic method. This drop of the even method is because the file domains are no longer aligned with the lock boundaries like the $128 \times 128 \times 128$ case. The performance difference between Lustre and GPFS implies the important role of the system locking protocol to the I/O performance. On Lustre, every I/O server is the lock manager for the file stripes stored locally. If a process makes a write request of amount larger than a file stripe, it must acquire locks from those I/O servers responsible for the stripes that are part of the request. Lustre enforces I/O atomicity by having the process obtain all the locks to these stripes and hold the locks until the entire write data have been received by the servers. The cyclic methods best fit for this protocol, because they make the I/O aggregators access to the same set of I/O severs and hence minimize the cost of lock acquisition. An advantage of the static-cyclic method is that the client-side file system caches are always coherent across multiple collective I/O operations, since file stripes are statically assigned and no file stripe will be accessed by more than one aggregator. In other words, the file system's cache coherence control will ever be triggered and cached data are evicted only when the operating system is under memory space pressure or the cache pages are explicitly flushed. This property is not presented in the other methods, because their file domains may change from one collective I/O to another. However, the interleaved file stripe access starts to occur for the static-cyclic method when the number of processes is larger than 512. Since each compute node on Jaguar is a quad-core processor, the number of I/O aggregators in a

collective I/O is a quarter of the number of MPI processes. In the 512-process case, the number of aggregators is 128, twice the number of the I/O servers used in our experiments. Similarly for the 1024-process case, there are 4 aggregators requesting file stripes that are interleaved in each I/O server. Initially, the group-cyclic method behaves the same as the static-cyclic method till the 256-process case. It keeps scaling up beyond 256 processes. The scalable results are attributed to the goal of the group-cyclic method that is to rearrange the file domains by removing any possible interleaved file stripe access and hence minimizing the number of lock requests for each process. This phenomena demonstrates the importance of avoiding any conflicted lock acquisition to the parallel shared-file I/O performance.

On GPFS, the cyclic methods do not perform as well as on Lustre. We only show the results of the static-cyclic method as they are very similar to the group-cyclic method. Under GPFS's token-based locking protocol, any client process can become a lock manager for future lock requests to its already granted file range. Both even and align partitioning methods produce file domains as single contiguous file regions, one for each I/O aggregator. Since file domains are not overlapping, all write locks can be immediately granted if the align method is used. On the other hand, the cyclic methods produce file domains containing many non-contiguous file stripes. An I/O aggregator must make a write call for each stripe and thus there is a lock request for each write. Since the file stripes from one aggregator are interleaved with all other aggregators, multiple lock requests must be made and most likely will be served by remote token holders. From our experiments, it is the cost of waiting for lock requests to be served that slows down the write speed, not because of the conflicted locks, as file domains are not overlapping for all four partitioning methods. The aligned method is more suitable for the token-based locking protocol, because it results in each aggregator making only one large contiguous write request and thus there is only a lock request from a process in a collective I/O.

To understand the detailed impact of these file domain partitioning methods to the two-phase I/O, we measure the two phases separately. The percentage of the I/O phase to the total execution time, also shown in Figure 6, is a key indicator to the effectiveness of a partitioning method. The cyclic methods' I/O phase percentages are significantly lower than the other two methods on Lustre. In some cases, the I/O phase even takes less time than the data redistribution phase. Note that the aggregate access regions and hence the total write amounts are equal for all partitioning methods. On GPFS, although the difference in the I/O phase percentages is not as dramatic as on Lustre, we can see the aligned method has the lowest I/O phase percentage and hence the highest write bandwidth.

The read performance tells a different story, because read locks are sharable. The fact that collective read operations do not cause any lock conflict suppresses the significance of file domain partitioning methods. Although there is no dramatic difference among the three partitioning methods, the static-cyclic method performs slightly worse than the other two methods on both Lustre and GPFS. This is owing to the read-ahead operations performed by the underlying file system. File systems prefetch a certain amount of data immediately following the read request. In the static-cyclic method, the prefetched data by an aggregator in fact belong to the file domains statically assigned to different aggregators. All prefetched data will never be used and the more read requests, the more cost of prefetching. Compared to the even and aligned methods that make only one read request, the static-cyclic method makes many read requests, one per file stripe. Therefore, it is not recommended for collective read operations to use the cyclic methods.

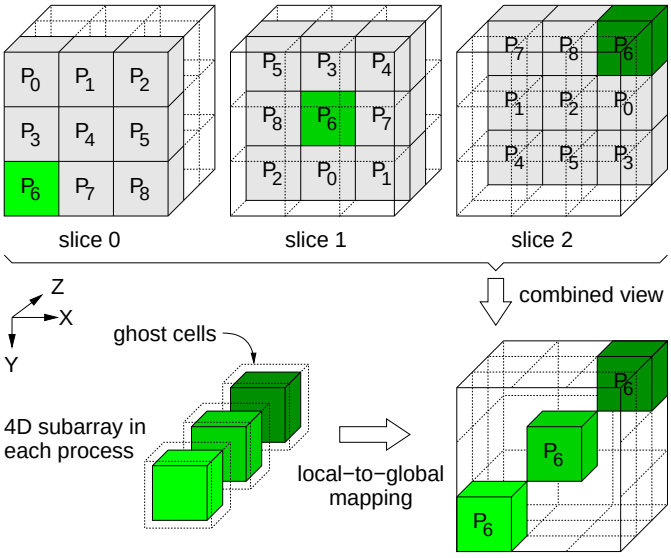

Fig. 7. BTIO data partitioning pattern. The 4D subarray in each process is mapped to the global array in a block-tridiagonal fashion. This example uses 9 processes and highlights the mapping for process P_6.

B. BTIO Benchmark

Developed by NASA Advanced Supercomputing Division, the parallel benchmark suite NPB-MPI version 2.4 I/O is formerly known as the BTIO benchmark [23]. BTIO presents a block-tridiagonal partitioning pattern on a three-dimensional array across a square number of processes. Each process is responsible for multiple Cartesian subsets of the entire data set, whose number increases with the square root of the number of processors participating in the computation. Figure 7 illustrates the BTIO partitioning pattern with an example of nine processes. In BTIO, forty arrays are consecutively written to a shared file by appending one after another. Each array must be written in a canonical, row-major format in the file. The forty arrays are then read back for verification using the same data partitioning. We evaluate the Class C data size which sets the global array size to $162 \times 162 \times 162$ and the total write amount for forty arrays is 6.34 GB. The global array size is fixed disregarding the number of MPI processes used. Hence, the I/O amount of individual processes decreases as the number of processes increases.

We measured BTIO write and read operations separately.

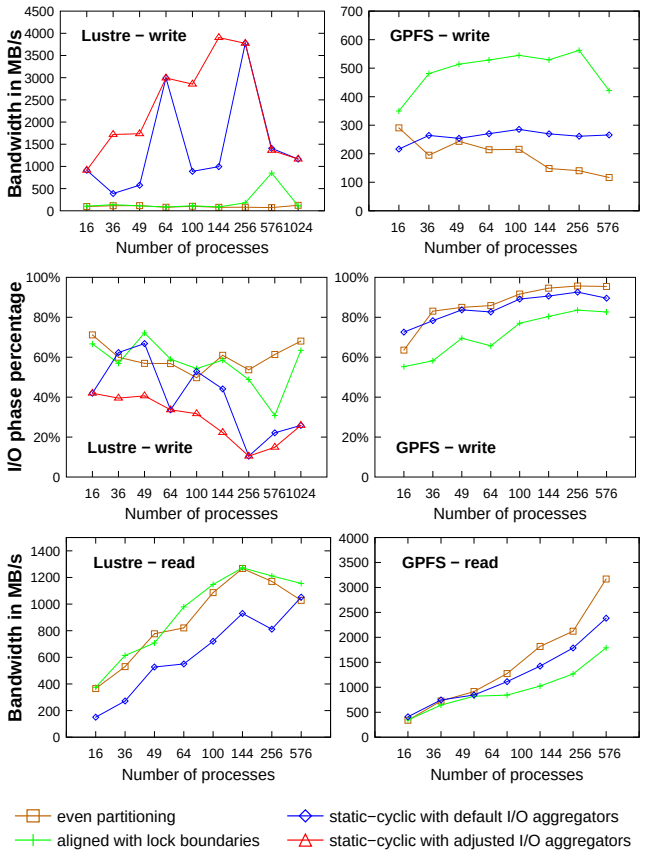

Fig. 8. Performance results for BTIO benchmark.

Figure 8 shows the write bandwidths, I/O phase percentages, and the read bandwidths. We use the static-cyclic method to represent both cyclic methods as their results are about the same on both Lustre and GPFS. On Lustre, the static-cyclic method outperforms the even and aligned methods for the write operation. The write bandwidth curve shows a few spikes in the cases of 16, 64, and 256 processes. In these cases, the number of default I/O aggregators are 4, 16, and 64, respectively. Since these numbers are factors of the number of I/O servers 64, the cyclic methods produce the file domains such that each aggregator is served by the same servers and no server receives write requests from more than one aggregator. Since BTIO runs only on square numbers of processes, other cases have no such advantage and their bandwidths are significantly lower.

To overcome this advantage, we also ran additional experiments for the cyclic methods by changing the default numbers of aggregators. The number of I/O aggregators can be set by the ROMIO collective buffering node hint, `cb_nodes`, and passed to ROMIO library as an MPI info object when opening the file. We set the number of aggregators to 8, 8, 16, 32, and 128 for the cases of 36, 49, 100, 144, and 576 processes, respectively. These numbers are the largest numbers that are factors of 64 and smaller than the default number of I/O aggregators. Adjusting the I/O aggregator numbers clearly further improves the write performance.

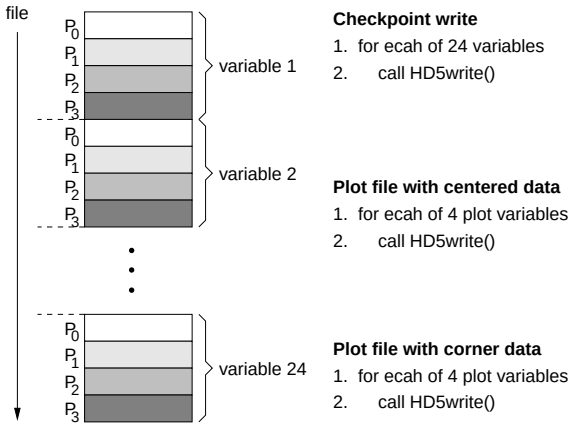

Fig. 9. I/O pattern of FLASH I/O benchmark.

As the number of processes reaches beyond 256, the write bandwidth of the static-cyclic method with 64 aggregators starts going down. This behavior is attributed to the smaller subarray size partitioned in each process, because the subarray size decreases as the number processes increases. When using 576 processes, each process only holds subarray size of 288.3 KB. When redistributing data from 576 processes to 64 aggregators, there are nine processes completing one aggregator during the data redistribution phase. With small write requests and large number of processes, the cost of data redistribution phase starts to grow and interfere the overall write performance. This can be observed from the Lustre's I/O phase percentage chart for those cases using adjusted numbers of I/O aggregators.

On GPFS, the aligned partitioning method outperforms the even and static-cyclic methods. The even method is never close to the aligned method because the $162 \times 162 \times 162$ array size only generates unaligned file domains for the even partitioning method. The even method is also slower than the static-cyclic method and the gap increases as the number of processes goes up. This implies that the cost of lock boundary conflict for a large number of small write requests is worse than the cost of communication contention at the lock token holders caused by the static-cyclic method.

Similar to the results of the ROMIO collective I/O test, the read bandwidths of the static-cyclic method are the worst. The same reason of data prefetching overhead slows down the static-cyclic method on Lustre. On GPFS, as the number of processes increase, both aligned and static-cyclic methods become worse. This behavior is caused by the smaller I/O amount from each process resulting uneven workload among the I/O aggregators. Of all three methods, the lock-boundary aligned partitioning generates the worst unbalanced workload.

C. FLASH I/O Benchmark

The FLASH I/O benchmark suite [24] is the I/O kernel of a block-structured adaptive mesh hydrodynamics code that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on neutron

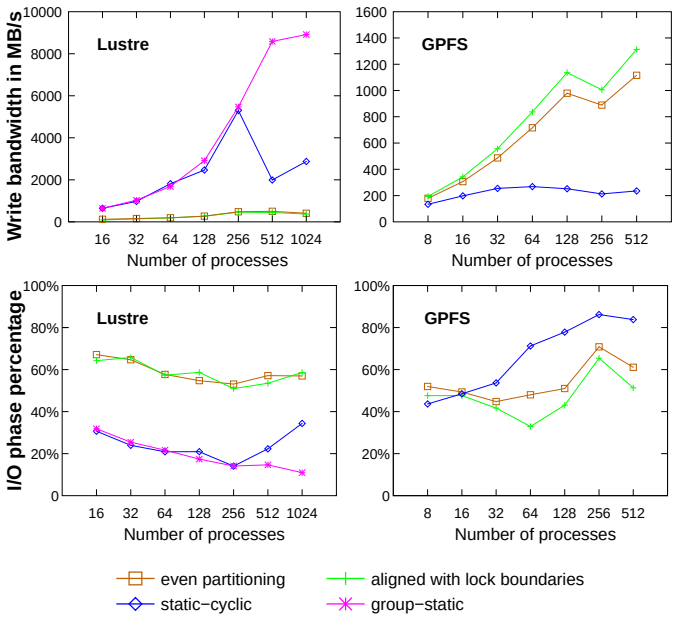

Fig. 10. Performance results for FLASH I/O benchmark.

stars and white dwarfs [25]. The computational domain is divided into blocks that are distributed across a number of MPI processes. A block is a three-dimensional array with an additional 4 elements as guard cells in each dimension on both sides to hold information from its neighbors. There are 24 variables per array element, and about 80 blocks on each MPI process. A variation in block numbers per MPI process is used to generate a slightly unbalanced I/O load. Since the number of blocks is fixed for each process, increase in the number of MPI processes linearly increases the aggregate I/O amount as well. FLASH I/O produces a checkpoint file and two visualization files containing centered and corner data. The largest file is the checkpoint, the I/O time of which dominates the entire benchmark. Figure 9 depicts the I/O pattern and extracts the program loops for the write operations. FLASH I/O uses the HDF5 I/O interface to save data along with metadata in the HDF5 file format. Since the implementation of HDF5 parallel I/O is built on top of MPI-IO [26], FLASH I/O performance reflects the use of different file domain partitioning methods. There are 24 collective write calls, one for each of the 24 variables. In each collective write, every MPI process writes a contiguous chunk of data, appended to the data written by the previous ranked MPI process. Therefore, a write request from one process does not overlap or interleave with the request from another. In ROMIO, this non-interleaved access pattern actually triggers the independent I/O subroutines, instead of collective subroutines, even if MPI collective writes are explicitly called. This behavior can be overridden by enabling the `romio_cb_write` hint. We use this hint so the four file domain partitioning methods can take effect in our experiments. In our experiments, we used a $32 \times 32 \times 32$ block size that produces about 20 MB of data per process in each collective write operation.

The performance results are shown in Figure 10. The write bandwidth curve on Lustre looks similar to the ROMIO collective write test. The even and aligned methods perform poorly and are much slower than the two cyclic methods. The static-cyclic method starts to slow down in the cases of 512 and 1024 process due to the interleaved file stripe access at the I/O servers. The group-cyclic method performs similar to the static-cyclic method for the cases of using 256 processes and less, but keeps scaling up beyond 256 processes. This difference is also reflected in the chart of I/O phase percentage, where the static-cyclic method increases significantly at 512 and 1024 cases. On GPFS, the aligned method has the best write bandwidth followed by the even method. The bandwidth curve of even method is closer to the aligned method than the static-cyclic method because we uses array size of $32 \times 32 \times 32$ which produces many evenly partitioned file domains aligned to the file stripe boundaries. In order to artificially generate a slightly unbalanced I/O load, FLASH I/O benchmark assigns process rank i with 80 + (i mod 3) data blocks and a process's write amount is either 20, 20.25, or 20.5 MB. With these amounts, the even partitioning method can create many file domains that are aligned with the 512 KB lock boundaries. The I/O phase percentage results also show the align method having the lowest percentages and the static-cyclic method the highest.

D. S3D I/O Benchmark

The S3D I/O benchmark is the I/O kernel of a parallel turbulent combustion application, named S3D, developed at Sandia National Laboratories [27]. S3D uses a direct numerical simulation solver to solve fully compressible Navier-Stokes, total energy, species and mass continuity equations coupled with detailed chemistry. The governing equations are solved on a conventional three-dimensional structured Cartesian mesh. A checkpoint is performed at regular intervals, and its data consists primarily of the solved variables in 8-byte three-dimensional arrays, corresponding to the values at the three-dimensional Cartesian mesh points. During the analysis phase the checkpoint data can be used to obtain several more derived physical quantities of interest; therefore, a majority of the checkpoint data is retained for later analysis. At each checkpoint, four global arrays, representing the variables of mass, velocity, pressure, and temperature, respectively, are written to files in their canonical order.

There are four collective writes in each checkpoint, one for a variable. Mass and velocity are four-dimensional arrays while pressure and temperature are three-dimensional arrays. All four arrays share the same size for the lowest three spatial dimensions X, Y, and Z, and they are all partitioned among MPI processes along X-Y-Z dimensions in the same block-block-block fashion. For the mass and velocity arrays, the length of the fourth dimension is 11 and 3, respectively. The fourth dimension, the most significant one, is not partitioned. Figure 11 shows the data partitioning pattern on a 3D array and the mapping of a 4D sub-array to the global array in file. In our evaluation, we keep the size of partitioned X-Y-Z

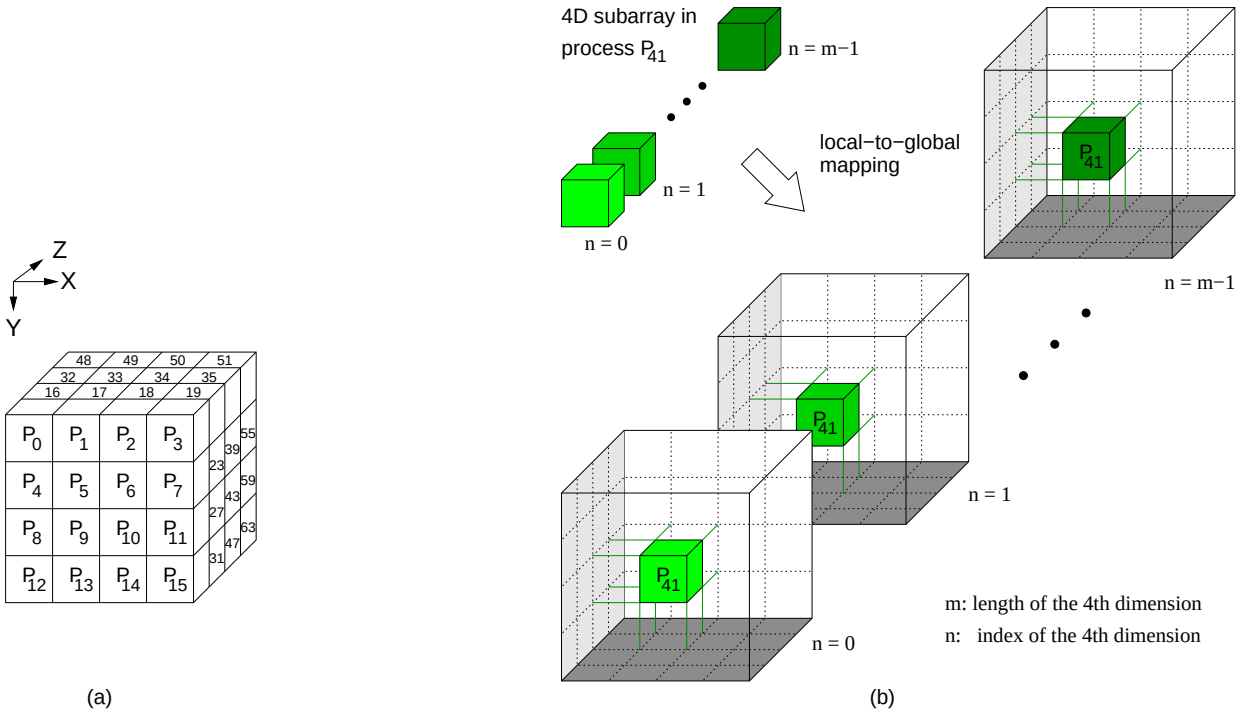

Fig. 11. S3D I/O data partitioning pattern. (a) For 3D arrays, the sub-array of each process is mapped to the global array in a fashion of block partitioning in all X-Y-Z dimensions.(b) For 4D arrays, the lowest X-Y-Z dimensions are partitioned the same as the 3D arrays while the fourth dimension is not partitioned. This example uses 64 processes and highlights the mapping of process P_{41}'s sub-array to the global array.

dimensions constant, $50 \times 50 \times 50$ in each process. These numbers were in fact used in real production runs. Each run produces about 15.26 MB of write data per process per checkpoint. As we increase the number of MPI processes, the aggregate I/O amount proportionally increases as well. We report the performance numbers by measuring ten checkpoints.

The performance results are given in Figure 12. The write bandwidth curve on Lustre is similar to the ROMIO collective write test and Flash I/O. For the static-cyclic method, the similar performance dips occur in the cases of 512 and 1024 processes. The group-cyclic method scales well beyond 512 processes. The I/O phase percentage also favors the cyclic methods over the even and aligned method. On GPFS, the aligned partitioning method performs the best, like all other benchmarks. With the array size used in the experiment, the even partitioning method does not generate any file domain that aligns to the lock boundaries and hence performs no closer to the aligned method. The I/O phase percentage charts are also similar to the previous I/O benchmarks. From all the I/O benchmark results presented in this paper, the impacts of the four partitioning methods to collective I/O performance are very consistent on both Lustre and GPFS.

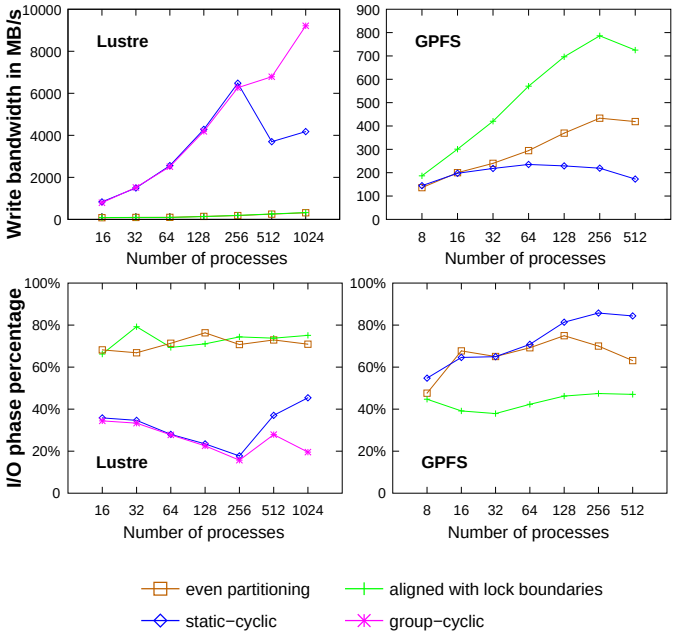

Fig. 12. Performance results for S3D I/O benchmark.

V. Conclusions

Through reorganizing file access regions among the I/O requesting processes, the two-phase I/O strategy can significantly improve the parallel I/O performance. However, it is rare to see a collective I/O performance near the system peak data bandwidth. The major obstacle lies on the file locking protocols used by the file systems and the parallel I/O libraries do not dynamically adjust their I/O methods for these protocols. The naive even partitioning method used by ROMIO in its two-phase I/O implementation produces well-balanced and large contiguous I/O requests, but may not best fit to the

underlying file system locking protocols. In fact, a collective I/O's performance depends on many factors, including the application access patterns, process collaboration strategies used in the MPI-IO library, and file system configurations. From our experiments, the way file domains are partitioned directly determines the I/O parallelism the underlying parallel file system's locking protocol can support. Among the four partitioning methods discussed in this paper, there is no single method that can outperform others on all file systems. A portable MPI-IO implementation must dynamically adapt a method that works best on the target file system.

The lessons learned from this work can be helpful for the MPI-IO implementation as well as application users to set the file hints. On file systems that implement server-based locking protocols, such as the Lustre, the group-cyclic file domain partitioning method is the best choice for collective write operations. Choosing the same number of aggregators as the number of I/O servers can avoid the interleaved file stripe access for static-cyclic method, as presented in the BTIO benchmark results. However, when the number of application processes become much larger than the servers, communication contention can easily formed at the aggregators during the data redistribution phase. Our future work will study the performance impact by varying the number of aggregators for large-scale runs. For token-based locking protocols, such as the one used by GPFS, the method that aligns the partitioning to the lock boundaries provides the best collective write performance. As for collective read operations, either even or aligned partitioning method is best to use. As new file systems with novel locking protocols are continuing to be developed in the future, it is important that a parallel I/O library dynamically adapts I/O strategies based on the file system configuration that can bring out the best performance.

ACKNOWLEDGMENTS

This work was supported in part by DOE SCIDAC-2: Scientific Data Management Center for Enabling Technologies (CET) grant DE-FC02-07ER25808, DOE SCiDAC award number DE-FC02-01ER25485, NSF HECURA CCF-0621443, NSF SDCI OCI-0724599, and NSF ST-HEC CCF-0444405. We acknowledge the use of the IBM IA-64 Linux Cluster at the National Center for Supercomputing Applications under TeraGrid Projects TG-CCR060017T, TG-CCR080019T, and TG-ASC080050N. This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] H. Shan and J. Shalf, "Using IOR to Analyze the I/O Performance of XT3," in *the Cray User Group Conference*, May 2007.

[2] Message Passing Interface Forum, *MPI-2: Extensions to the Message Passing Interface*, Jul. 1997, http://www.mpi-forum.org/docs/docs.html.

[3] J. del Rosario, R. Brodawekar, and A. Choudhary, "Improved Parallel I/O via a Two-Phase Run-time Access Strategy," in *the Workshop on I/O in Parallel Computer Systems at IPPS '93*, Apr. 1993, pp. 56–70.

[4] D. Kotz, "Disk-directed I/O for MIMD Multiprocessors," *ACM Transactions on Computer Systems*, vol. 15, no. 1, pp. 41–74, Feb. 1997.

[5] R. Thakur, W. Gropp, and E. Lusk, *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation*, Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Oct. 1997.

[6] J. Squyres, A. Lumsdaine, W. George, J. Hagedorn, and J. Devaney, "The interoperable message passing interface (IMPI) extensions to LAM/MPI," in *Proceedings, MPI Developers Conference (MPIDC)*, March 2000.

[7] R. Thakur, W. Gropp, and E. Lusk, "An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces," in *the 6th Symposium on the Frontiers of Massively Parallel Computation*, Oct. 1996.

[8] ——, "Data Sieving and Collective I/O in ROMIO," in *the 7th Symposium on the Frontiers of Massively Parallel Computation*, Feb. 1999.

[9] B. Nitzberg and V. Lo, "Collective Buffering: Improving Parallel I/O Performance," in *the Sixth IEEE International Symposium on High Performance Distributed Computing*, August 1997, pp. 148–157.

[10] R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen, "Implementing MPI-IO Atomic Mode Without File System Support," in *the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2005.

[11] W. Liao, A. Ching, K. Coloma, A. Choudhary, and L. Ward, "An Implementation and Evaluation of Client-Side File Caching for MPI-IO," in *the International Parallel and Distributed Processing Symposium*, Mar. 2007.

[12] IEEE/ANSI Std. 1003.1, *Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language]*, 1996.

[13] J. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges, "MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS," in *Supercomputing*, Nov. 2001.

[14] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *the Conference on File and Storage Technologies (FAST'02)*, Jan. 2002, pp. 231–244.

[15] Lustre: A Scalable, High-Performance File System, *Whitepaper*, Cluster File Systems, Inc., 2003.

[16] J. Prost, R. Treumann, R. Hedges, A. Koniges, and A. White, "Towards a High-Performance Implementation of MPI-IO on top of GPFS," in *the Sixth International Euro-Par Conference on Parallel Processing*, Aug. 2000.

[17] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, "Server-directed Collective I/O in Panda," in *Supercomputing*, Nov. 1995.

[18] K. Coloma, A. Choudhary, W. Liao, W. Lee, E. Russell, and N. Pundit, "Scalable High-level Caching for Parallel I/O," in *the International Parallel and Distributed Processing Symposium*, Apr. 2004.

[19] K. Coloma, A. Ching, A. Choudhary, W. Liao, R. Ross, R. Thakur, and L. Ward, "A new flexible MPI collective I/O implementation," in *the IEEE Conference on Cluster Computing*, Sep. 2006.

[20] X. Ma, M. Winslett, J. Lee, and S. Yu, "Improving MPI-IO Output Performance with Active Buffering Plus Threads," in *the International Parallel and Distributed Processing Symposium*, Apr. 2003.

[21] K. Coloma, A. Choudhary, W. Liao, W. Lee, and S. Tideman, "DAChe: Direct Access Cache System for Parallel I/O," in *the 20th International Supercomputer Conference*, Jun. 2005.

[22] H. Yu, R. Sahoo, C. Howson, G. Almasi, J. Castanos, M. Gupta, J. Moreira, J. Parker, T. Engelsiepen, R. Ross, R. Thakur, R. Latham, and W. D. Gropp, "High Performance File I/O for the BlueGene/L Supercomputer," in *the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, Feb. 2006.

[23] P. Wong and R. der Wijngaart, "NAS Parallel Benchmarks I/O Version 2.4," NASA Ames Research Center, Moffet Field, CA, Tech. Rep. NAS-03-002, Jan. 2003.

[24] M. Zingale, "FLASH I/O Benchmark Routine – Parallel HDF 5," Mar. 2001, http://flash.uchicago.edu/∼zingale/ flash_benchmark_io.

[25] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo, "FLASH: An Adaptive Mesh Hydrodynamics Code for Modelling Astrophysical Thermonuclear Flashes," *Astrophysical Journal Suppliment*, pp. 131–273, 2000.

[26] HDF Group, *Hierarchical Data Format, Version 5*, The National Center for Supercomputing Applications, http://hdf.ncsa.uiuc.edu/HDF5.

[27] R. Sankaran, E. Hawkes, J. Chen, T. Lu, and C. Law, "Direct Numerical Simulations of Turbulent Lean Premixed Combustion," *Journal of Physics: conference series*, vol. 46, pp. 38–42, 2006.