Scaling Parallel I/O Performance through I/O Delegate and Caching System

Arifa Nisar, Wei-keng Liao and Alok Choudhary

Electrical Engineering and Computer Science Department Northwestern University
Evanston, Illinois 60208-3118
Email: {ani662,wkliao,choudhar}@ece.northwestern.edu

***Abstract*—Increasingly complex scientific applications require massive parallelism to achieve the goals of fidelity and high computational performance. Such applications periodically offload checkpointing data to file system for post-processing and program resumption. As a side effect of high degree of parallelism, I/O contention at servers doesn't allow overall performance to scale with increasing number of processors. To bridge the gap between parallel computational and I/O performance, we propose a portable MPI-IO layer where certain tasks, such as file caching, consistency control, and collective I/O optimization are delegated to a small set of compute nodes, collectively termed as I/O Delegate nodes. A collective cache design is incorporated to resolve cache coherence and hence alleviates the lock contention at I/O servers. By using popular parallel I/O benchmark and application I/O kernels, our experimental evaluation indicates considerable performance improvement with a small percentage of compute resources reserved for I/O.**

I. Introduction

Modern scientific applications are relied on, to simulate physical and natural phenomenons, instead of carrying out real experiments in controlled environment. Physicists, astrophysicists and earth scientists require scientific applications to provide accurate and efficient modeling of natural phenomenons like nuclear explosion, molecular dynamics, climate modeling, ocean ice modeling etc. These applications require enormous computing power during the course of their execution, as well as huge storage space to store the checkpointing data generated for post-processing and program resumption. High Performance Computing is the answer to these crucial requirements posed by modern scientific applications. In modern era of scientific computations parallel scientific applications are being deployed on high performance computing systems, to achieve the goals of fidelity, high performance and scalability. Although, high performance computing resources provide massive parallelism and computing power to fulfil the crucial requirements of the scientific applications, but most of the high-end applications do not scale beyond few hundreds nodes [1].

As the number of processors grows to a large number, achieving high efficiency for computation becomes very difficult based on Amdahl's law. In other words, increased I/O overhead limits the incremental gain in efficiency with large number of processors. We posed the following question to several leading application scientists who, as described earlier, face significant I/O performance problems. "If you had an access to a large system, and if we asked you to leave 5 % of the nodes for the I/O software, and in turn if you are guaranteed significant improvement in performance, would you accept such a solution?" To our surprise, without hesitation, there was a unanimous answer of "yes". In addition to describing the state of I/O performance on production systems, this answer also shows a deep understanding of the efficiency curves on large-scale systems on the part of application scientists. They are willing to sacrifice a small fraction of resources (some nodes and their memory), in order to obtain much better performance on the I/O system.

Recently, we have been investigating I/O infrastructure on ultra-scale systems such as, the IBM BlueGene/L (BG/L) and BlueGene/P (BG/P) systems. Basic architecture of these systems implies that I/O bandwidth of the system is efficiently utilized by having separate nodes for performing I/O and computation, where one I/O node is often assigned to many compute nodes[2]. For example, in the case of BG/L, one I/O node performs I/O operations for 8 to 64 processors [3], [4]. This kind of architecture deceases the chances of possible disk contention, when thousands of nodes are concurrently performing I/O in a high performance system. Imagine, if all the compute nodes start performing I/O operations then there are enormously high chances of disk contention, and system will hardly use bandwidth provided by parallel file system. Most of the nodes will be waiting for acquiring locks and data disk will be overloaded with the huge number of requests.

We present an I/O Delegate Cache System (IODC) that organizes system cores into two categories (a) Application Nodes (b) I/O Delegate Nodes (IOD nodes). I/O Delegate Cache system is incorporated inside ROMIO [5] which is an MPI-IO implementation. IODC system intercepts all the I/O requests initiated by user application running on application nodes, and redirects them to IOD nodes. Figure 1 shows the level of our contribution in I/O stack hierarchy of the overall system. Parallel applications pass their I/O requests to MPI-IO layer directly or through an interface to MPI-IO e.g., HDF5 [6]. Our system sits in MPI-IO and provides an abstraction of the underlying parallel file system to the application nodes. Out of all the nodes, only IOD nodes can access underlying parallel file system. These IOD nodes optimize I/O accesses by caching the accessed data in their local memory. Also, they

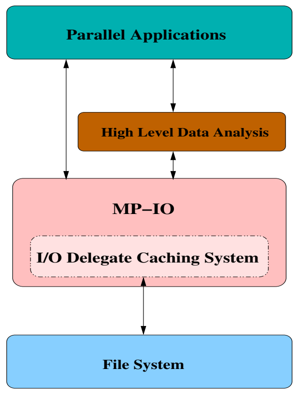


Fig. 1. Parallel I/O Stack

detect if multiple requests are accessing contiguous data in file. They combine multiple small I/O requests into fewer but larger data access requests. By exploiting processing power and memory of IOD nodes for performing I/O caching and data aggregation, we achieve considerably high percentage of I/O bandwidth improvement. We conducted our experiments using one I/O benchmark BTIO and two production applications' IO kernels, FLASH and S3D I/O on popular parallel file systems GPFS and Lustre. Experimental results show that I/O bandwidth is improved manyfold by setting aside less than 10% extra nodes to work as IOD nodes.

Rest of the paper is organized as follows; Section II discusses related research work. It also provides the guidelines for taking parallel system design decisions. Section III explains the architecture of IODC in detail. Section IV evaluates and analyzes the I/O performance of IODC for different I/O benchmarks on GPFS and Lustre. Section VI draws conclusions and discusses possible future improvements in the system.

II. Related Work

MPI-IO inherits two important MPI features: MPI communicators, which define a set of processes for group operations and MPI derived data types, which describe complex memory layouts. A communicator specifies the processes that can participate in a collective MPI operation for both, interprocess communication and I/O requests to a shared file. File open requires an MPI communicator to indicate the group of processes accessing the file. In general, MPI-IO data access operations can be split into two types: collective I/O and independent (non-collective) I/O. Collective operations require all processes in the communicator to participate. Because of this explicit synchronization, many collective I/O implementations may exchange access information amongst all the processes to generate a better overall I/O strategy. An example of this is the two-phase I/O technique [7]. Two-phase I/O is used in ROMIO, a popular MPI-IO implementation developed at Argonne National Laboratory [8]. To preserve the existing optimizations in ROMIO, like two-phase I/O, we incorporate our design in the Abstract Device I/O (ADIO) layer where ROMIO interfaces with underlying file systems [9].

Client-side file caching is supported in many parallel file systems; for instance, IBMs GPFS [10], [11] and Lustre [12]. By default, GPFS employs a distributed token-based locking mechanism to maintain cache coherency on nodes. Lock tokens must be granted before any I/O operation is performed [14]. Distributed locking avoids the obvious bottleneck of a centralized lock manager, by making the token holder act like local lock authority for granting further lock requests to the corresponding byte range. A token allows a node to cache data because this data can not be modified elsewhere without revoking the token. Lock granularity for GPFS is equal to file strips size. IBM's MPI-IO implementation over AIX operating system uses the data shipping mechanism; files are divided into equally sized blocks, each of which is bound to a single I/O agent, a thread in an MPI process. The file block assignment is done in a round-robin striping scheme. A given file block is only cached by the I/O agent responsible for all the accesses to this block. All I/O operations must go through the I/O agents which then "ship" the requested data to the appropriate process. Data shipping maintains cache coherency by allowing at most one cached copy of data amongst all the agents. The Lustre file system uses a slightly different distributed locking protocol where each I/O server manages locks for the stripes of file data it stores. The lock granularity for Lustre is the file stripe size. If a client requests a lock held by another client, a message is sent to the lock holder asking to release the lock. Before a lock can be released, dirty cache data must be flushed to the servers. Both Lustre and GPFS are POSIX compliant file systems and therefore respect POSIX I/O atomicity semantics. To guarantee atomicity, file locking is used in each read/write call to guarantee exclusive access to the requested file region. On parallel file systems like Lustre and GPFS where files are striped across multiple I/O servers, locks can span multiple stripes for large read/write requests. Lock contention due to atomicity enforcement can significantly degrade parallel I/O performance [13].

Shan et.al. conducted a study with principal investigators of 50 projects on computing platforms of National Energy Research Supercomputing Center (NERSC) to find out the I/O trends in real applications [14]. This study shows some interesting results like, I/O accesses are rarely random and parallel applications' write activity is dominated amongst the overall I/O activities of the system. It is important to understand these trends, as many of them can help determining the areas of improvements in order to scale I/O performance with higher number of nodes. In order to design an optimal parallel I/O system, we kept these I/O trends in mind while devising our system. Larkin et. al. discuss the guidelines for efficient I/O practices for Cray XT3/XT4 system on Lustre [15]. Some of these suggestions advocate accessing data in large I/O accesses and using a subset of nodes to perform I/O operations. In addition to Cray XT3/XT4 system, these guidelines also apply to other large scale systems as well, so, we have incorporated the applicable suggestions in our design. We provide the support of data aggregation by combining many small I/O requests into fewer but larger I/O request before initiating requests to file system. Also, we set aside a small number of nodes to perform I/O operation instead of letting all the nodes access the parallel file system. So, by keeping these trends and guidelines in mind, we designed a I/O subsystem that optimizes these requests, such that overall I/O performance is improved. By using IODC, I/O performance can be considerably improved even for the applications that do not follow good I/O practices.

Collective buffering approach [16] rearranges data in processors' memory, to initiate optimized I/O requests, thus reduces the time spent in performing I/O operations. This scheme requires a global knowledge of I/O pattern in order to perform optimization. Bennett et.al. present an I/O library Jovian[17], [18], which uses separate processors called 'coalescing nodes' to perform I/O optimization by coalescing small I/O operations. This approach requires application support to provide out-of-core data information in order to coalesce the contiguous data on disk. Our system consists of a portable MPI-IO plugin, which doesn't require any global knowledge of I/O accesses pattern or application support.

III. IO Delegate and Cache System

We propose an additional I/O layer termed as 'I/O Delegate Cache System' (IODC) for MPI applications, which is integrated into an MPI-IO implementation ROMIO [5] . This allows I/O performance to scale with processing power of massively parallel large-scale systems. Lock contention at disks occurs when two processes compete with each other to access the same file region. With the increase in number of processors, I/O request load at data disks increases. Also, lock acquiring becomes more difficult task, with more number of competitors trying to acquire lock. This also increases the number of processors waiting to receive I/O service at any given time. By introducing additional IOD nodes to delegate the I/O responsibility, we restrict the maximum number of processors accessing the file at one time equal to number of IOD nodes. Also, by deploying a caching system on these IOD nodes, we provide the benefit of caching and I/O request alignment with file system stripe size, which further decreases the chances of lock contention. In short, IODC improves the parallel I/O performance by reducing the possible disk contention, storing data in a collective cache system and aggregating small requests into bigger ones. This system organizes all of the system cores into two categories (a) Application Nodes (b) I/O Delegate Nodes. Our system sits in ROMIO and restricts the application nodes to access the underlying parallel file system. It rather redirects all the I/O operation to dedicated I/O Delegate (IOD) nodes. IOD nodes are the only nodes in the system, which can directly read or write a file to parallel file system. So, if a write request is generated at an application nodes, its request will be intercepted by our plugin inside ROMIO and it will be redirected to a IOD node which is responsible for performing I/O operation for that specific application nodes. Also data accessed by these nodes is stored in a collective caching system in IOD node. This cached data can be re-accessed, hence reducing number of I/O requests to the disks. These IOD nodes further optimize the I/O accesses by combining multiple I/O request in to smaller number of large requests for contiguous data on disk.

A. IODC Interprocess Communication Mechanism

Ratio of application nodes and IOD nodes is a flexible user controllable parameter. No application related computation is performed on IOD nodes, likewise, no I/O service task is carried out by application nodes. Application nodes have all of their processing power and memory reserved for their own computation, no memory or processing power is claimed by IODC. IODC requires a separate set of node, set aside to act like IOD nodes in addition to application nodes. Application nodes are mapped to IOD nodes in a sequential pattern. For example, if the number of application nodes are double of IOD nodes', then MPI rank 0 and 1 of application processes will be mapped to MPI rank 0 of IOD nodes, and so on. For the purpose of load balancing, application nodes are mapped to IOD nodes such that equal number of application nodes are assigned to each I/O Delegate nodes. If number of application nodes is not perfectly divisible by number of IOD nodes, then it is made sure that difference between number of application nodes assigned to any two IOD nodes in the system may not exceed 1.

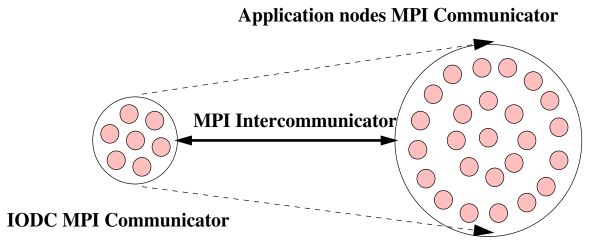


Fig. 2. MPI Intercommunicator

IODC uses a couple of important MPI features like MPI communicator, MPI dynamic processes management and MPI intercommunicator. MPI communicator enables a group of processes to communicate with each other. MPI dynamic processes management utility enables one MPI communicator to launch another application on dynamically created child MPI communicator. An MPI intercommunicator is established amongst these parent and child MPI communicators. Using MPI intercommunicator, processes can communicate across the application and communicators' boundary, similarly, they can communicate under the boundary of single communicator. So MPI intercommunicator provides a way to communicate amongst the two groups of processes running two distinct and independent applications. Figure 2 shows, how IODC scheme utilizes MPI intercommunicator and MPI dynamic processes management. At the start of system our IODC is started on IOD nodes. This system launches a parallel application on application nodes by dynamically allocating processes and creating a child MPI communicator. Application nodes and IOD nodes belong to two different MPI communicators which communicate with each other through MPI intercommunicator.

B. IODC System Design

IODC architecture can be understood by Figure 3. As shown in figure, the core system is divided in two main groups: application nodes and IOD nodes. All I/O requests generated from application nodes are transferred to IOD nodes. These I/O Delegate nodes, then access the parallel file system to perform the I/O operations requested by application nodes. Number of IOD node is kept considerably smaller than that of application nodes.

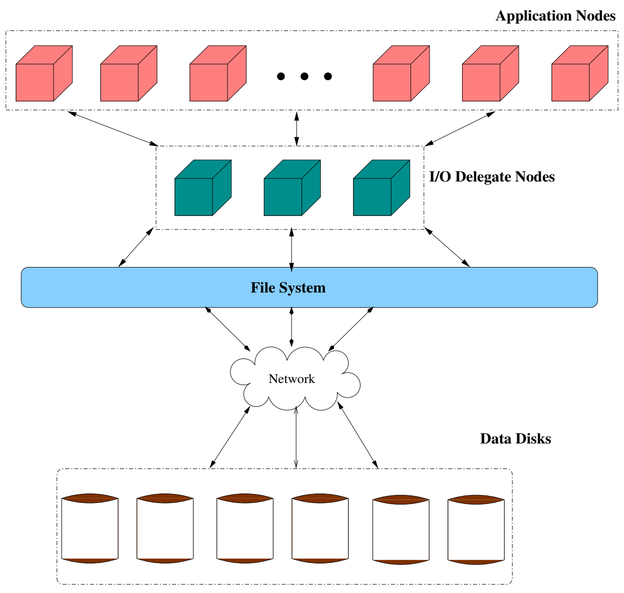


Fig. 3. System Architecture

Figure 4 explains the overall system design of IODC. Right side represents an application node and left side represents an IOD node mapped to that specific application node. Scientific computations are carried out on application nodes, which generate a number of I/O requests for underlying parallel file system. These I/O requests may include file create, open, read, write, close, truncate etc. On each application node, IODC plugin inside ROMIO triggers a redirection of these requests to IOD node. On IOD node, there is a waiting loop that detects the incoming requests sent by remote or local communicators. This loop keeps executing two probing routines one after the another. One probing routine polls on intercommunicator to detect incoming requests from any application node, and other keeps probing on local communicator to detect the requests from peer IOD nodes. A collective MPI caching system similar to [19][20][21] is deployed on IOD nodes to optimize the I/O requests for parallel file systems. Request detector calls corresponding MPI cache routines for performing I/O operations requested by remote application node. Collective MPI cache system completes the requested I/O operation by, caching the cacheable data (read,writes) in its local buffer or calling the corresponding system calls. Result of this I/O request is returned to application node and then computation at application node is resumed.

When a file is to be opened for the first time by an application, then this open request is transferred to the IOD node with MPI rank 0. After opening the file, a unique id is assigned to this file, which is mapped to system id of the file to correctly translate the future references to this file. This unique id is duplicated amongst all the IOD nodes for future reference. Also, this id is sent back to application node who originated the request. In future, when any IOD node receives an I/O request related to same file, IODC will use this unique id to manage cache data amongst IOD nodes. An application node sends its I/O requests to the IOD node mapped to it in static sequential fashion. In case, an IOD node receives a request to access a block in the file that is cached by another IOD node, then it will redirect the request to the corresponding IOD node.

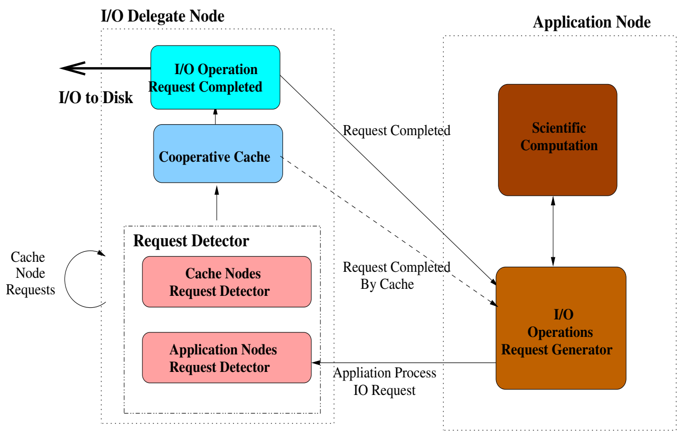

I/O Delegate Node
Application Node
I/O to Disk
I/O Operation Request Completed
Cooperative Cache
Request Completed
Scientific Computation
Request Detector
Cache Node Requests
Cache Nodes Request Detector
Request Completed By Cache
I/O Operations Request Generator
Application Nodes Request Detector
Appliation Process IO Request

Fig. 4. System Design

C. IODC System States

This section discusses the possible states for IOD nodes during the course of application execution. State diagram in figure 5 is shown for write only I/O access pattern. Each IOD node keeps waiting for incoming requests, let's call this as state 0. IOD node can receive two main categories of requests, (a) requests from application node (b) requests from a peer IOD node. When an IOD node receives a write request from an application node and if write data can fit in cache memory at IOD node then IOD node will change its state from 0 to 1 and stores the data in its cache. Note that during the process of caching some data communication amongst peer IOD nodes may also be required, which is not shown in this state diagram for the sake of simplicity. If data doesn't fit in the cache then it is directly written to the file on disk and state changed from 0 to 2. If this IOD node is not responsible for caching the file blocks to be written then it will relay the request to the correct IOD node. If such request is received by an IOD node then it changes its state from 0 to 3. Upon reception of a request from another IOD node, data is received and stored in local cache memory. After completing caching operation or write operation to disk IOD node goes back to state 0, which is the waiting state. If cache residing on a IOD node exhausts or file is closed all the data stored in cache memory is flushed to disk. In that case state changes from 1 or 3 to 2. when system is idle IOD node will keep waiting in state 0, until it receives a request to terminate the system.

Read requests are entertained the same way, except direction of data transfer is inverse. Other non-cacheable requests like open, close etc. are dealt the same way, except necessary information to carry out the I/O operation is transferred and not the write data. Corresponding IOD node performs the requested I/O operation and go back to the waiting state.

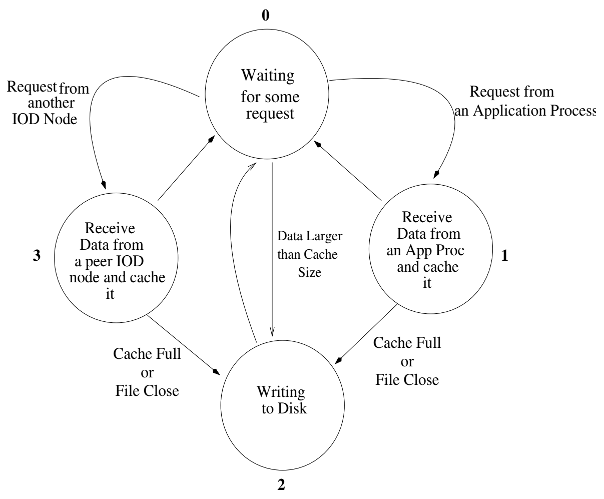

0
Waiting for some request
Request from another IOD Node
Request from an Application Process
3
Receive Data from a peer IOD node and cache it
Data Larger than Cache Size
Receive Data from an App Proc and cache it
1
Cache Full or File Close
Cache Full or File Close
Writing to Disk
2

Fig. 5. IOD Nodes States for write accesses

Traditional cache benefits include write behind and prefetch but in parallel system cache provides performance benefits such as alignment, intermediate buffering and ability to re-arrange the data before initiating I/O. Collective MPI caching system's policies and management details will be discussed in following section.

D. Cache Metadata Management

Caching scheme logically divides a file into equally sized pages, each of which can be cached. The default cache page size is set to the file system block size and is also adjustable through an MPI hint. A page size aligned with the file system lock granularity is recommended, since it prevents false sharing. Cache metadata describing the cache status of these pages are statically distributed in a round-robin fashion amongst the MPI processes, which collectively open the shared file. A modulus operation is required to find the MPI rank of the process storing a pages metadata. This distributed approach avoids centralization of metadata management. Cache metadata includes the MPI rank of the pages current location, lock mode, and the pages recent access history. A page's access history is used for making the decision of cache eviction and page migration. To ensure cache metadata integrity (atomic access to metadata), a distributed locking mechanism is implemented, where each MPI process manages the lock requests for its assigned metadata. There are two kind of possible lock modes, shared read locks and exclusive write locks. Metadata locks must be obtained before an MPI process can freely access the metadata, cache page, and the file range corresponding to the page. When a request to cache pages consecutive in file space are cached at different MPI processes, all pages must be locked prior to their access. Since deadlock may occur when more than two processes are simultaneously requesting locks for the same two pages, we employ the two-phase locking strategy proposed in [22]. Under this strategy, lock requests are issued in a strictly increasing page ID order and the prior page lock must be obtained before requesting the next lock.

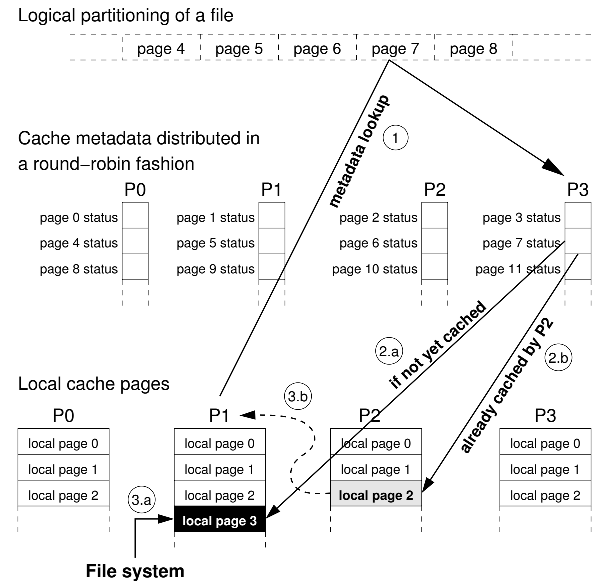


Fig. 6. Cache System

E. Cache Page Management

To simplify coherence control, we allow at most a single cached copy of file data among all MPI processes. When accessing a file page that is not being cached anywhere, the requesting process will try to cache the page locally, by reading the entire page from the file system if it is a read operation, or by reading necessary part of the page if it is a write operation. An upper bound, by default 1 GB, indicates the maximum memory size that can be used for caching. If the memory allocation utility, malloc() finds enough memory to accommodate the page and the total allocated cache size is below the upper bound, the page will be cached. Otherwise, under memory pressure, the page eviction routine is activated. Eviction is solely based on only local references and a least-recent-used policy. If the requested file pages are not cached and the request amount is larger than the upper bound, the read/write calls will go directly to the file system. If the requested page is already cached locally, a simple memory copy will satisfy the request. If the page is cached at a remote process, the request is forwarded to the page owner. An example I/O flow for a read operation is illustrated in Figure

6 with four MPI processes. In this example, process P1 reads data in file page 7. The first step is to lock and retrieve the metadata of page 7 from P3 (7 mod 4 = 3). If the page is not cached yet, P1 will cache it locally (into local page 3) by reading from the file system, as depicted by steps (2.a) and (3.a). If the metadata indicates that the page is currently cached on P2, then an MPI message is sent from P1 to P2 asking for data transfer. In step (3.b), assuming file page 7 is cached in local page 2, P2 sends the requested data to P1. When closing a file, all dirty cache pages are flushed to the file system. A high water mark is used in each cache page to indicate the range of dirty data, so that flushing needs not always be an entire page.

IV. Experiment Results

I/O Delegate Cache system implementation is evaluated on two machines: Tungsten and Mercury, at the National Center for Supercomputing Applications (NCSA). Tungsten is a 1280-node Dell Linux cluster, where each node contains two Intel 3.2 GHz Xeon processors sharing 3 GB of memory. The compute nodes run a Red Hat Linux operating system and are inter-connected by both Myrinet and Gigabit Ethernet communication networks. A Lustre parallel file system version 1.4.4.5 is installed on Tungsten. The lock granularity of Lustre is equal to file stripe size 512 KB on Tungsten. Output files are saved in a directory configured with a stripe count of 16 and a 512 KB stripe size. All files created in this directory share the same striping parameters. Mercury 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 size. The lock granularity on GPFS is equal to file stripe size, 512KB on Mercury. Note that because IBM's MPI library is not available on Mercury, we could not comparatively evaluate the performance of GPFS's data shipping mode. IODC is implemented in the ROMIO layer of MPICH version 2-1.0.6, the latest version of MPICH2 at the time our experiments were performed.

For experimentation, we use a 512 KB page size for IODC system. Setting the cache page size to the file system stripe size, aligns all the write requests to stripe boundaries and hence, to lock boundaries. For performance evaluation, we use one benchmark BTIO, and two I/O kernels of real applications namely FLASH I/O and S3D I/O. All the charts shown in this paper, report percentage improvement in I/O bandwidth for write-only access pattern. The I/O bandwidth numbers are obtained by dividing the aggregate write amount by the time measured from the beginning of file open until after file close. Percentage improvement is obtained by taking a ratio of net increment in I/O bandwidth and the I/O bandwidth achieved in native case. So, 100% means that I/O bandwidth with IODC is two times the native case. In this paper, speedup will refer to the ratio of I/O bandwidths achieved with IODC and native case.

As described earlier, by introducing IOD nodes into the system, we reduce I/O contention at data disks by reducing number of requests accessing the file system, aligning the accesses with lock boundary, and caching the accessed data. For all the experiments, we ran two processes per node for executing application and one process per node to execute IODC. We used very small number of IOD nodes as compared to the number of application processes. In our experimental evaluation, we kept number of IOD nodes no more than 10% of the number of application processes. For example, for 144 application processes we used 14 additional nodes as IOD nodes. In addition to reducing the degree of competition at data disk, there are number of other factors that play important role in this performance improvement. Caching at IOD nodes allows the first process who requests for a page, to buffer it locally; thereby, taking the advantage of data locality and increasing the likelihood of local cache hits. We expect performance improvement to increase with increase in number IOD nodes for a fixed number of application processes. This is because, higher number of IOD nodes translates into bigger cache memory size. With larger cache pool, system won't start thrashing soon. It is shown in the charts that in most cases, for a fixed number of application processes, I/O performance increases with the increased number of IOD nodes. Also, cache system detects the overlapping accesses to same lock granularity, and combines multiple small requests into fewer large requests. As most of the parallel file systems are optimized for larger requests, IODC decreases the I/O request load on disks and results in improved data access time. Data accesses initiated from IOD nodes are aligned on lock boundary of file system which further reduces the chance of two processes competing for the file region bounded by one locking stripe.

There are certain overheads associated with IODC. IODC adds an extra step of passing data through IOD nodes, which incurs extra data communication cost amongst application processes and IOD nodes. Generally, we expect percentage improvement to increase with number of application processes because we have higher potential to remove lock contention in case of large number of application processes. In some cases percentage improvement starts decreasing for further increase in the number of application processes. This can be explained by the fact that amount of data communicated amongst application processes and IOD nodes also increases, which can limit the performance improvement beyond certain number of application processes. This maximum optimal number of application processes depends on access pattern of application, underlying parallel file system and ethernet communication protocol. We conducted our experiment using TCP/IP, we expect to get even better performance results in case of Myrinet.

For one problem size, BTIO benchmark generates a checkpointing file of fixed size no matter how many processes take part in writing. So, with the increase in number of processes, size of data written by each process decreases. In contrast, for

FLASH and S3D I/O, all of the process writes a fixed size of data. So, data written to files is directly proportional to the number of processes executing the benchmark. We will keep this fact into account while analyzing the performance results. Note that although no explicit file synchronization is called in these benchmarks, closing files will flush all the dirty data.

A. 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. BTIO provides options for using either MPI collective or independent I/O. In BTIO, 40 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. We evaluate the array dimensionality $102 \times 102 \times 102$ and $162 \times 162 \times 162$ and an aggregate write amount for a complete run of 1.58 GB and 6.34 GB respectively. With this fixed aggregate write amount, we evaluate BTIO using different number of MPI processes; therefore, the amount written by each process decreases as the number of processes increases.

Figure 7 includes the percentage improvement in I/O bandwidth achieved for BTIO $162 \times 162 \times 162$ array size on GPFS and Lustre parallel file systems. In figures 7(a) and 7(d) percentage I/O bandwidth improvement is shown for 144, 256, 324 and 400 application processes, with no more than 10% additional processors as IOD nodes. Each line in the chart represents fixed number of application processes with varying number of IOD nodes. We achieve considerably higher percentage I/O bandwidth improvement by putting in less than or equal to 10% extra resources. In case of BTIO, data written by each process decreases as the number of processes increases. So as the number of processors increases, parallel I/O performance improvement is compromised. IODC enables aggregation that combines many small I/O requests into fewer but larger I/O accesses. By combining requests we reduce the number of requests sent to disk, as well as gain the benefit of accessing large contiguous data on file.

As described earlier, the distributed lock protocol used in parallel file systems maintains the cache metadata integrity, this protocol incurs a certain degree of communication overhead. It can be seen in the figure 7(d) that for 400 processes considerably higher bandwidth improvement has been achieved on Lustre as compared to GPFS. This can be explained by the difference in lock management protocol of these two parallel file systems. To enforce the atomicity, no two processes are allowed to access the same file block at one time. As we increase the number of processes, possibility of lock contention increases as more number of processes are competing with each other to access the same file. It is an established fact that lock contention can drastically degrade I/O performance[13]. So, for largest number of application processes, performance improvement achieved on Lustre is considerably higher than GPFS. This can be explained by the fact that without IODC, Lustre's lock contention resolving protocol takes more time as compared to the one in GPFS. As we increase the number of application processes, lock contention problem becomes more serious. IODC reduces degree of lock contention by not letting application processes access the file system, so, there is more potential of lock contention avoidance on Lustre.

As shown in the Figure 7(d), if number of application processes increases, percentage improvement also tends to increase. This is because of the fact that with large number of processors I/O disk contention becomes very critical performance limiting factor. So, for large number of application processes IODC gets more margin of enhancing performance. For each collective write operation in BTIO, the aggregate write amount is 162.18 MB. When partitioned evenly among all MPI processes, the file domains are not aligned with the file system lock boundaries. Thus, conflict locks due to false sharing occur and hence serialize the concurrent write requests. IODC further improves performance by aligning processor data distribution with file layout. By keeping lock boundary of file system aligned with cache page size, possible contention at the lock boundary is decreased. As BTIO generates the same size of file, so with the increase in application processes, data is divided in smaller chunks and data communication cost between application processes and IOD nodes increases. This effect can also be seen in the figure 7(a) but the case of Lustre dominates overall performance improvement as expected. For fixed number of application processes, we expect performance improvement to increase with increasing number of IOD nodes because of larger cache pool and increased degree of communication parallelism. But figure 7(a) shows that performance improvement doesn't scale with number of IOD nodes. This can be explained by the increased data communication overhead amongst peer IOD nodes with increasing number of IOD nodes. This overhead effect is not that evident in figure 7(d) as Lustre provides higher margin of lock contention avoidance than GPFS. So, lock contention avoidance seems to dominate in overall performance improvement. Figures 7(a) and 7(d) show that 3 to 4 times speedup has been achieved in some cases on GPFS and Lustre by just allocating IOD nodes no more than 10% of application processes. We are not giving detailed evaluation analysis for BTIO $102 \times 102 \times 102$ array size but a summary of those results is given in section V.

B. FLASH I/O Benchmark

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

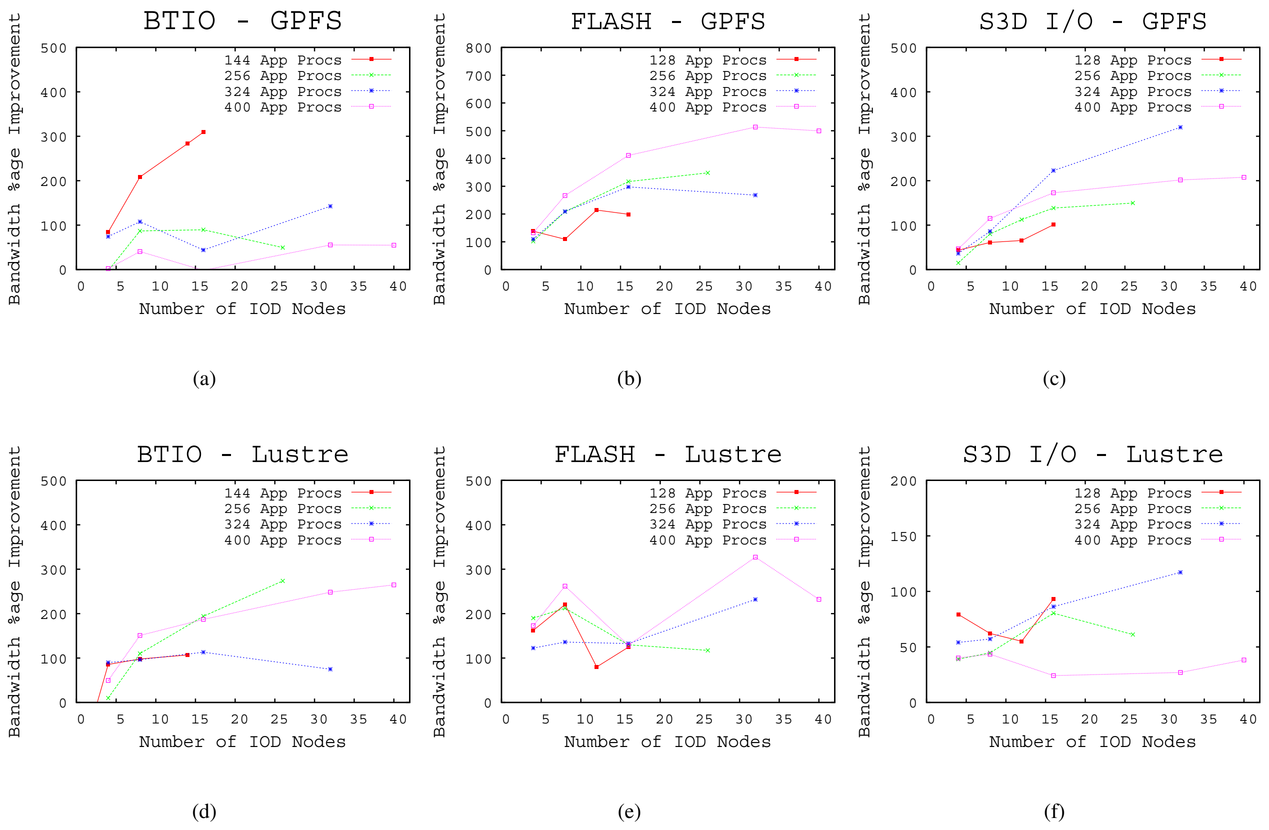


Fig. 7. Percentage Improvement in I/O Bandwidth for BTIO, FLASH and S3D I/O with GPFS and Lustre

in each dimension on both sides to hold information from its neighbors. In our experiments, we used $8 \times 8 \times 8$ and $16 \times 16 \times 16$ block size. There are 24 variables per array element, and about 80 blocks on each MPI process. So, total of 7.5 MB and 60 MB data is generated per process respectively. 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, increasing the number of MPI processes linearly increases the aggregate write amount. 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. FLASH I/O uses the HDF5 I/O interface to save data along with its metadata in the HDF5 file format. Since the implementation of HDF5 parallel I/O is built on top of MPI-IO [26], the performance effects of I/O delegate caching system can be observed in overall FLASH I/O performance. To eliminate the overhead of memory copying in the HDF5 hyper-slab selection, FLASH I/O extracts the interiors of the blocks via a direct memory copy into a buffer before calling the HDF5 functions. There are 24 I/O loops, one for each of the 24 variables. In each loop, every MPI process writes into a contiguous file space, appending its data to the previous ranked MPI process; therefore, a write request from one process doesn't 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. Note that FLASH I/O writes both array data and metadata through the HDF5 I/O interface to the same file. Metadata, usually stored at the file header, may cause unaligned write requests for array data when using native MPI-IO.

Figures 7(b) and 7(e) show percentage improvement in I/O bandwidth for $8 \times 8 \times 8$ array size for 128, 256, 324, 400 application processes on GPFS and Lustre. For the FLASH I/O pattern, forcing collective I/O creates a balanced workload, but not without some extra communication costs. The aggregate write bandwidth is calculated by dividing the data size written to all three files by the overall execution time. Similar to our analysis for BTIO, such alignment significantly eliminates the lock contention that can otherwise occur in the underlying GPFS and Lustre file systems from the use of native MPI-IO. As mentioned earlier, I/O performance improvement is expected to increase with increased number of application processes on Lustre. Figure 7(e) shows that maximum performance improvement achieved by increasing the number of application processes. For a fixed number of application processes, performance improvement doesn't consistently scale up. This phenomenon can be explained by FLASH I/O pattern where write data generated by each process is contiguous and not interleaved.

In order to generate a slightly unbalanced I/O load, FLASH

I/O assigns process rank i with 80 + (i mod 3) data blocks. Hence, the subarray size are 20, 20.25, or 20.5 MB, which generates many write requests that are aligned with lock boundaries. In this situation the improvement margin for IODC is slim. However, we still see considerably high improvement in all cases and these improvements shall be attributed to the I/O aggregation by the file caching. Figure 7(b) shows a relatively regular I/O performance improvement with increased number of IOD nodes. There is also a smooth increase in performance improvement with increasing number of application processes. There are some exceptions which can be explained by the effects of IODC overheads described earlier. Figures 7(b) and 7(e) show that for 400 processors maximum speedup of 4 and 6 is achieved on GPFS and Lustre respectively by just allocating IOD nodes no more than 10% of application processes. We are not giving detailed evaluation analysis for FLASH $16 \times 16 \times 16$ array size but a summary of those results is given in section V.

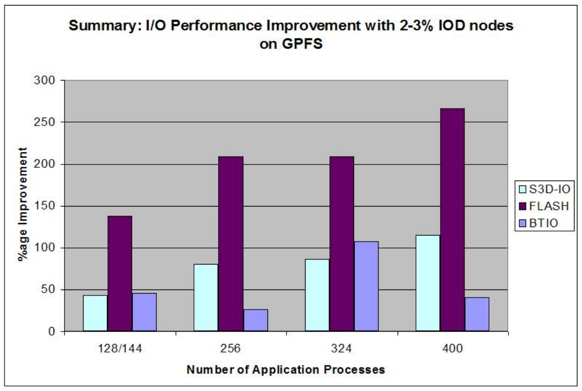
C. S3D I/O Benchmark

The S3D I/O benchmark is the I/O kernel of the S3D application, a parallel turbulent combustion application using a direct numerical simulation solver developed at Sandia National Laboratories [27]. S3D solves 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 are written to files and they represent the variables of mass, velocity, pressure, and temperature, respectively. 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 partitioning fashion. The length of the fourth dimension of mass and velocity arrays is 11 and 3, respectively, and not partitioned. In the original S3D, the I/O is programmed in Fortran I/O function and each process writes all its subarrays to a separate file at each checkpoint. We added the functionality of MPI-IO to write the arrays into a shared file in their canonical order. With this change, there is only one file created per checkpoint, regardless of the number of MPI processes used. For performance evaluation, we keep the size of partitioned X-Y-Z dimensions a constant $25 \times 25 \times 25$ and $35 \times 35 \times 35$. This produces about 1.9 MB and 5.2 MB of write data per process per checkpoint respectively. Similar to FLASH I/O, for fixed number of application processes we increase the number of IOD nodes and percentage bandwidth improvement increases with it.

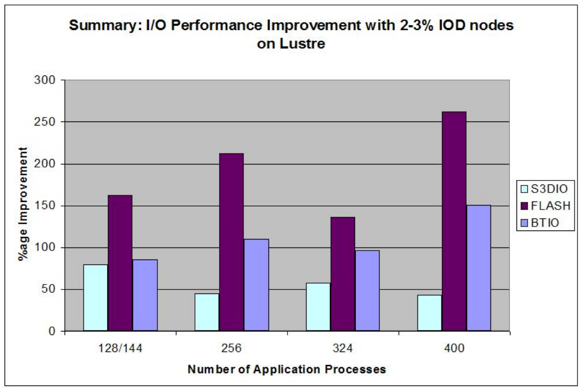
Figures 7(c) and 7(f) show percentage bandwidth improvement for S3D I/O benchmark with $25 \times 25 \times 25$ array size, on 128, 256, 324, 400 application processes. Figure 7(c) shows close to expected performance improvements pattern. Performance improvement achieved tends to increase with the increasing number of application processes because of higher margin of lock contention prevention. Also, for fixed number of application processes, performance improvement tend to increase with increasing number of IOD nodes because of better disk utilization, bigger cache pool and alignment. As for S3D I/O benchmark, total data written to disk increases with the number of application processes, we can see that for 400 application processes effect of communication overhead seems to reflect more than 324 case. Figure 7(f) also shows expected behavior up till 324 application processes but for 400 application processes, performance improvement decreases considerably. We are looking into details of S3D's I/O access pattern and underlying file system protocol to understand the behavior of such unexpected cases. S3D I/O improvement is not as significant as FLASH I/O and BTIO. This is owing to the number of collective I/O made for each file. In each checkpoint, S3D I/O makes 4 collective writes to save the 4 arrays into a shared file. At the time the file is closed, all cached data must be flushed. Therefore, there are not much I/O aggregation effect that can be expected in the S3D I/O pattern. We are not giving detailed evaluation analysis for S3D I/O $35 \times 35 \times 35$ array size but a summary of those results is given in section V.

V. Evaluation Summary & Additional Evaluation

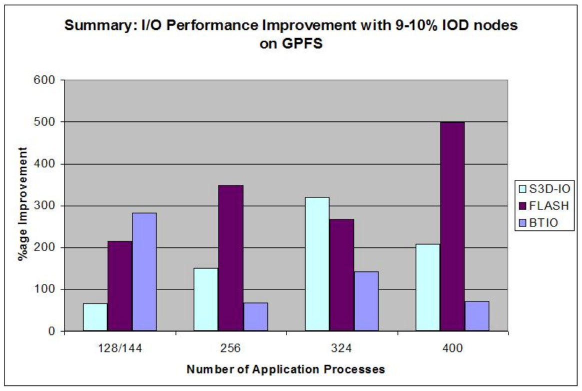
There must be an optimal number of IOD nodes for each specific case, which depends on access pattern of application, size of communicated data and underlying communication protocol. IODC will not show much of a performance improvement if: (a) I/O access generated by application are already aligned with file system lock boundary. (b) An application generates I/O accesses that are small enough to fit in the local cache. (c) If an application generates too large of the I/O accesses to overwhelm the memory of IOD nodes (d) A very small number of accesses are generated per file or (e) Network communication is very slow and overhead of IODC may become very significant. Some of the cases show unexpected behavior, for example, figure 7(e) shows that percentage performance improvement for all application processes converges for 16 IOD nodes. As chart shows percentage improvement and not absolute bandwidth, we can say that as caching and alignment benefit depends on number of IOD nodes, so approximately similar benefits are achieved for different number of application processes. Although for higher number of application processes we get more margin of lock contention reduction but it may be possible that for 16 IOD nodes I/O bandwidth provided by file system is under-utilized and becomes the restricting factor. As already discussed, figure 7(a) also shows some irregular performance improvement with varying number of IOD nodes. Figure 7(f) shows expected behavior up till 324 application processes



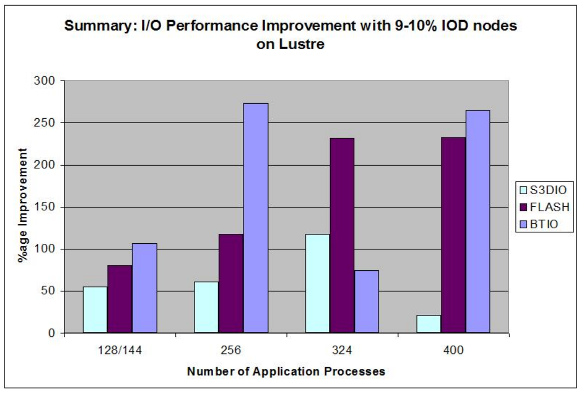
(a)



(b)



(c)



(d)

Fig. 8. Evaluation Summary of BTIO, FLASH and S3D I/O with GPFS and Lustre

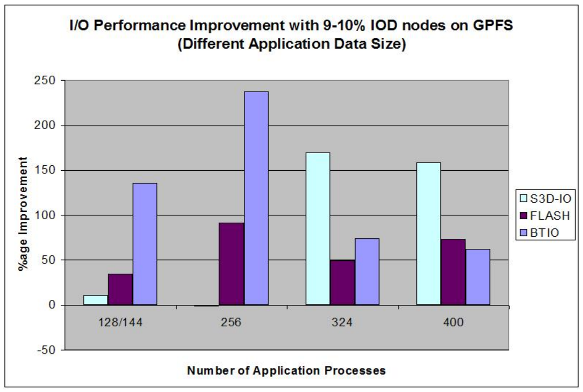
but for 400 application processes, performance improvement decreases considerably. We are looking into details of S3D I/O's I/O access pattern and underlying file system protocol to understand the behavior of such unexpected cases. But we are looking into more details of access patterns of different benchmarks and behavior of underlying parallel file systems to better understand and solve the problem.

By using IODC we observe I/O performance improvement in all the cases of BTIO, FLASH and S3D I/O shown above. Figure 8 sums up already presented results taken on Lustre and GPFS. These charts demonstrate the I/O bandwidth improvement observed with FLASH $8 \times 8 \times 8$, S3D I/O $25 \times 25 \times 25$ and BTIO $162 \times 162 \times 162$. Please note that for the first cluster of histograms, BTIO was executed for 144 applications nodes, whereas FLASH and S3D I/O were executed on 128 application processes. Figures 8(a),8(b),8(c) and 8(d) show that by using only 2-3% and 9-10% additional nodes as IOD nodes, we achieved considerable performance improvement for all the applications with GPFS and Lustre respectively.

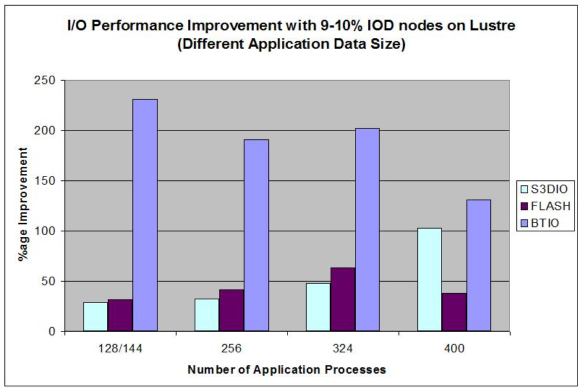
We also conducted our experiment with FLASH $16 \times 16 \times 16$, S3D I/O $35 \times 35 \times 35$ and BTIO $102 \times 102 \times 102$. Please note that array size of FLASH and S3D I/O is increased, while array size of BTIO is decreased in this second set of experiments. Figure 9 shows that for FLASH and S3D I/O, percentage improvement decreases by increasing the array size to $16 \times 16 \times 16$ and $35 \times 35 \times 35$ respectively. As already mentioned, extra cost of IODC includes extra communication time spent between application processes and IOD node. This communication is performed through high speed connection network so it is generally very small as compared to time spent in accessing file system. As array size increases, processors generates more data to be communicated to IOD nodes and communication time between application and IOD nodes becomes significant. So, I/O performance improvement is effected for larger array size. Please note that smaller array size has been used for BTIO case, that is why we don't see any less performance improvement than achieved in previous set of experiments. Although Graphs in Figure 9 show that average performance improvement is lesser as compared to the smaller array size, but still some performance improvement has been achieved as compared to the native case. Only exception is S3D I/O for 256 application processes on GPFS Figure9(a) case, which gives exactly the same performance as the native case.

VI. Conclusions and Future Work

We have proposed IODC, an MPI-IO layer for large-scale parallel applications, which enables I/O performance of the system to scale with computational parallelism. IODC intercepts the I/O requests generated by application and optimizes them before accessing the file system. IODC requires additional compute nodes to perform I/O optimizations before



(a)



(b)

Fig. 9. Evaluation Summary of BTIO, FLASH and S3D I/O benchmarks (with different application data sizes) with GPFS and Lustre

initiating I/O requests to underlying filesystem. Our system evaluation demonstrates that by using only less than or equal to 10% extra compute nodes, we achieve very high performance improvement in I/O bandwidth. Performance improvement depends on many factors, like I/O access pattern of application, parallel file system protocol, number of application processes, number of IOD nodes, data size and underlying network protocol etc. We conclude that for getting optimal performance improvement, it is important to deeply understand all of these factors to achieve correct combination of them. Currently, we are using synchronized writes to the file system, so an IOD node either receives data from some application process or writes to the file system at one specific time. As a part of our future research, we are planning to use asynchronous writes which will enable overlapping between data communication and file system access procedures. In addition, to enable asynchronous writes we plan to flush dirty data to the file system during the execution, instead of doing it at file close. This will also help overlapping I/O access time with data communication amongst application processes and IOD nodes, and amongst the peer IOD nodes.

VII. 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.

References

[1] C. W. McCurd, R. Stevens, H. Simon, W. Kramer, D. Bailey, W. Johnston, C. Catlett, R. Lusk, T. Morgan, J. Meza, M. Banda, J. Leighton, and J. Hules, "Creating Science-Driven Computer Architecture:A New Path to Scientific Leadership," National Energy Research Scientific Computing Center, Tech. Rep., October 2002. [Online]. Available: http://www.nersc.gov/news/reports/ArchDevProposal.5.01.pdf

[2] 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. Gropp, "High performance file I/O for the Blue Gene/L supercomputer," *hpca*, vol. 0, pp. 187–196, 2006.

[3] G. Almasi, C. Archer, J. G. Castanos, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, N. Smeds, and Burkhard, "Implementing MPI on the BlueGene/L Supercomputer." [Online]. Available: citeseer.ist.psu.edu/almasi04implementing.html

[4] R. D. Loft, "Blue Gene/L Experiences at NCAR," in *IBM System Scientific User Group meeting (SCICOMP11)*, 2005.

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

[6] V. . HDF Group. Hierarchical Data Format, "The National Center for Supercomputing Applications," http://hdf.ncsa.uiuc.edu/HDF5. [Online]. Available: http://hdf.ncsa.uiuc.edu/HDF5

[7] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel i/o via a two-phase run-time access strategy," *SIGARCH Comput. Archit. News*, vol. 21, no. 5, pp. 31–38, 1993.

[8] 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, October 1997.

[9] R. Thakur and E. Lusk, "An abstract-device interface for implementing portable parallel-i/o interfaces," in *in Proceedings of The 6th Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, 1996, pp. 180–187.

[10] "General Parallel File System," http://www-03.ibm.com/systems/clusters/software/gpfs/index.html. [Online]. Available: http://www-03.ibm.com/systems/clusters/software/gpfs/index.html

[11] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *FAST*, D. D. E. Long, Ed. USENIX, 2002, pp. 231–244.

[12] "Lustre: A Scalable, High-Performance File System. Whitepaper," 2003.

[13] R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen, "Implementing mpi-io atomic mode without file system support," in *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1135–1142.

[14] H. Shan and J. Shalf., "Using IOR to Analyze the I/O performance for HPC Platforms," in *Cray Users Group Meeting (CUG) 2007, Seattle, Washington*, may 7-10 2007.

[15] J. Larkin and M. Fahey, "Guidelines for Efficient Parallel I/O on the Cray XT3/XT4," in *Cray Users Group Meeting (CUG) 2007, Seattle, Washington*, may 7-10 2007.

[16] B. Nitzberg and V. Lo, "Collective buffering: Improving parallel I/O performance," in *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 1997, p. 148.

[17] R. Bennett, K. Bryant, J. Saltz, A. Sussman, and R. Das, "Framework for optimizing parallel i/o," Univ. of Maryland Institute for Advanced

Computer Studies Report No. UMIACS-TR-95-20, College Park, MD, USA, Tech. Rep., 1995.

[18] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz, "Jovian: A Framework for Optimizing Parallel I/O," in *Proceedings of the Scalable Parallel Libraries Conference*. Mississippi State, MS: IEEE Computer Society Press, 1994, pp. 10–20. [Online]. Available: citeseer.ist.psu.edu/bennett94jovian.html

[19] W. keng Liao, A. Ching, K. Coloma, A. N. Choudhary, and L. Ward, "An Implementation and Evaluation of Client-Side File Caching for MPI-IO," in *IPDPS*. IEEE, 2007, pp. 1–10.

[20] W. keng Liao, A. Ching, K. Coloma, A. Nisar, A. Choudhary, J. Chen, R. Sankaran, and S. Klasky, "Using MPI file caching to improve parallel write performance for large-scale scientific applications," in *SC*. The ACM/IEEE Conference on Supercomputing, November 2007.

[21] W. keng Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and S. Tideman, "Collective Caching: Application-aware Client-side File Caching," in *HPDC '05: Proceedings of the High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 81–90.

[22] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[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, January 2003.

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

[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," in *Astrophysical Journal Suppliment*, 2000, p. 131273.

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

[27] R. Sankaran, E. R. Hawkes, J. H. Chen, T. Lu, and C. K. Law, "Direct numerical simulations of turbulent lean premixed combustion," *Journal of Physics Conference Series*, vol. 46, pp. 38–42, Sep. 2006.