Delegation-based I/O Mechanism for High Performance Computing Systems

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*—Massively parallel applications often require periodic data checkpointing for program restart and post-run data analysis. Although high performance computing systems provide massive parallelism and computing power to fulfill the crucial requirements of the scientific applications, the I/O tasks of high-end applications do not scale. Strict data consistency semantics adopted from traditional file systems are inadequate for homogeneous parallel computing platforms. For high performance parallel applications independent I/O is critical, particularly if checkpointing data is dynamically created or irregularly partitioned. In particular, parallel programs generating a large number of unrelated I/O accesses on large scale systems often face serious I/O serializations introduced by lock contention and conflicts at file system layer. As these applications may not be able to utilize the I/O optimizations requiring process synchronization, they pose a great challenge for parallel I/O architecture and software designs. We propose an I/O mechanism to bridge the gap between scientific applications and parallel storage systems. A static file domain partitioning method is developed to align the I/O requests and produce a client-server mapping that minimizes the file lock acquisition costs and eliminates the lock contention. Our performance evaluations of production application I/O kernels demonstrate scalable performance and achieve high I/O bandwidths.**

***Index Terms*—Parallel I/O, I/O Delegation, MPI-IO, Non Collective I/O, Collaborative Caching, Parallel File Systems, File Locking**

I. Introduction

I/O architectures in modern high performance systems[1], [2], [3] have been contrived such that the compute nodes and storage servers are separated in groups and connected through high speed networking devices. Data generated by applications must pass through many abstraction layers of I/O stack before reaching the storage devices. Figure 1 shows a common perception of I/O stack. The best I/O throughput can only be guaranteed if all of these layers are utilized to the best of their capacities. Incidentally, most of these layers have been designed independently, and hence certain information that describes the I/O intention at one layer may not have adequate interfaces to pass to another.

Modern parallel file systems are configured with multiple I/O servers in order to provide high data throughput. Each server may contain one or more disk RAIDs (Redundant Array of Independent Disks) to further improve the data reliability and performance. A file stored on the parallel file systems can be partitioned across multiple servers so large requests can be served by multiple servers simultaneously. However, evolving from traditional distributed file systems, modern parallel file systems inherit certain I/O consistency semantics that were designed to protect data integrity from concurrent file accesses, a scenarios commonly occurred in a distributed environment. To achieve desired I/O semantics, file locking mechanism is used to guarantee the access permissions of individual I/O requests. Two important consistency requirements from POSIX standard known to restrict parallel I/O performance from scaling are atomicity and cache coherence [4], [5]. When multiple processes concurrently access a shared file, file locks may cause serialization of the I/O operations which adversely affects the I/O performance. While a large number of the application processes are waiting for acquiring locks on the same file regions, the I/O bandwidth sustainable by a parallel file system is underutilized. Details of file system locking issues is discussed in Section V-A.

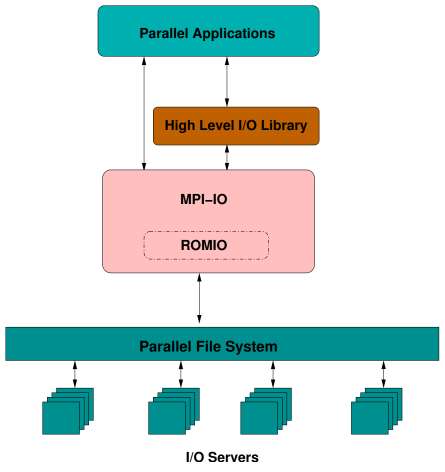

Fig. 1. A Common Parallel I/O Architecture Stack: This figure explains the way different I/O layers are commonly stacked one over another in large scale parallel environments. High end application layer leverages its parallel I/O related tasks directly or through a high level I/O library (PnetCDF, HDF etc.) to MPI-IO. ROMIO, an MPI-IO's implementation services these parallel file accesses by directly interacting with underlying parallel file systems.

In the traditional distributed environment, requests from different clients are seldom related, so the impact of performance degradation due to enforcing strict data consistency semantics is not a frequent problem. However, in the modern era of science and engineering, computational simulations like combustion, molecular dynamics, fusion, climate prediction, etc. are parallel programs that run on hundreds of thousands of cores to scale with the size of the problem. In contrast to the distributed computing, processes performing parallel computations are closely related I/O clients, which often partition global data objects and access shared files concurrently. For such parallel applications, treating each client process independently may restrict the I/O scalability.

Scientific community has started recognizing the problem of pessimist storage system protocols adopted by the file systems that are rarely required by the parallel applications but handicap their I/O parallelism. In recent years, various contributions have been made both on hardware and software to address this problems. A noteworthy example in hardware improvement is the IBM BlueGene systems that add a new I/O architecture layer sitting in between compute nodes and I/O servers, specially designed to reduce the scale of I/O contention. The I/O sub-system of BlueGene systems is discussed in Section V-D. MPI defines a set of programming interfaces for parallel file access, commonly referred as MPI-IO. With this framework, many optimizations such as two-phase I/O [6] and data sieving [7], have been successfully demonstrated significant performance improvement for the parallel I/O. One of the prominent software contributions is the collective I/O functionality proposed in the message passing interface (MPI) standard [8].

Designed for MPI collective I/O, the two-phase I/O rearranges small, non-contiguous requests amongst processes to form large, contiguous ones that can result in better I/O latency. Data sieving avoids small-sized I/O by first reading large file chunks into memory buffers, updating the buffers with the requests, and then writing the chunks back to the file. Despite of data sieving technique being available for MPI independent I/O functions, optimizations for independent I/O are generally considered to be a challenging task. High performance independent I/O is critical, particularly for the applications whose data is dynamically created or irregularly partitioned amongst processes. An example is the parallel programs based on Adaptive Mesh Refinement (AMR) algorithm [9]. For such data partitioning patterns, global process synchronization may not be practical and hence they must rely on independent I/O to complete the I/O task.

This paper presents an I/O delegation system that aims to minimize file lock conflicts and improve the MPI independent I/O performance. The I/O delegation work was initiated in [5] which provided an intermediate software layer between the application processes and parallel file systems to enable several I/O optimizations. I/O delegation system employs a set of additional compute processes to carry out the I/O requests for the application processes. These additional compute processes are alternatively referred to as I/O delegates or delegate processes. Application's I/O requests are forwarded to the delegate processes, where they are rearranged to best match the file locking characteristics, such as lock granularity, of the underlying file system.

In this paper, we present a new strategy for I/O delegate system, a static file domain mapping method that statically maps evenly partitioned file regions to the delegates in a round robin fashion. This essentially means that a unique I/O delegate can only access the assigned file regions, termed as the **file domain** of this delegate. The motivation is to minimize the number of I/O clients accessing an I/O server and hence potentially minimize the number of conflicted locks. We exercise this design in ROMIO, a popular MPI-IO implementation developed at Argonne National Laboratory [10]. With the static mapping of file domains, lock contentions that frequently occur in the parallel I/O operations can be mostly eliminated. A file caching mechanism[11] is implemented in delegate system that enables data aggregation across multiple requests aiming for improving MPI independent I/O performance. Implementation details and additional experimental analysis for caching system has been provided in supplementary sections VI-B and VII-C. This feature is also considered an optimization that spans multiple MPI-IO requests, collectives and/or independents, which have been ignored by existing MPI-IO optimizations. The I/O delegation system thins the performance gap between the collective and independent I/O, while latter's performance has long been considered much worse than that of former's. Most importantly, I/O delegate system achieves such performance improvement, while still fulfilling the MPI-IO data consistency semantics.

We conducted our experiments on two production parallel machines with real application I/O kernels. Franklin, a Cray XT4 system at National Energy Research Scientific Computing Center [3], and Abe, the TeraGrid Intel-64 Cluster at the National Center for Supercomputing Applications [12], were used to evaluate I/O delegate system. Two application I/O kernels FLASH[13], [14] and S3D[15], and an MPI-IO test program taken from ROMIO[10] are used in the evaluation. With only 4 to 6% of additional compute resources allocated as delegates, independent I/O achieves up to 2.5 times faster than the native collective I/O method on Franklin. On Abe, we achieved up to 15 times I/O bandwidth improvement over the collective I/O.

The paper is organized as follows. Section II explains the strategy of static file domain mapping to the delegates in detail. Section III presents our evaluations and analysis of the I/O performance for different I/O benchmarks. Section IV draws conclusions and discusses future work.

Additional sections have been added in supplementary file which are as follows. Section V discusses the research background and motivation from the perspective of existing I/O optimizations and characteristics of parallel file systems. This section also discusses a number of related works including Cray MPI-IO library [16] and I/O forwarding techniques [17]. Section VI discusses the intrinsic implementation details of basic I/O delegation operations. Section VII provides the details on experimental setup and additional evaluations of I/O delegation system.

II. Design and Development

This section discusses of file domain assignment strategy in I/O delegation system. Details about I/O delegation system architecture, and other I/O delegation functions, such as initialization, I/O request flow, and caching etc. may be found in Section VI.

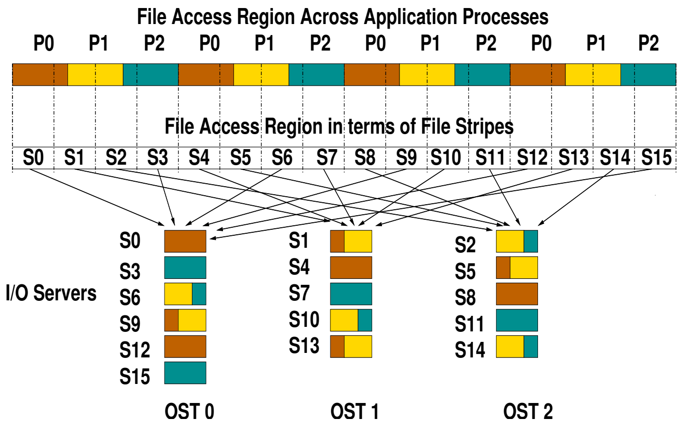
File Access Region Across Application Processes
P0 P1 P2 P0 P1 P2 P0 P1 P2 P0 P1 P2
File Access Region in terms of File Stripes
S0 S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14 S15
I/O Servers
S0 S3 S6 S9 S12 S15
S1 S4 S7 S10 S13
S2 S5 S8 S11 S14
OST 0
OST 1
OST 2

Fig. 2. File access region is partitioned among the application processes P_0, P_1, and P_2. Different colors represent data accessed by different application processes. From Lustre file system's perspective, the entire file is partitioned into 16 stripes S_0, S_1, $\cdots$, S_{15} which are distributed across the I/O servers, OST_0, OST_1, and OST_2. Even though file accesses are non-overlapping among the processes, when requests from two processes access the same stripe, lock conflict occurs.

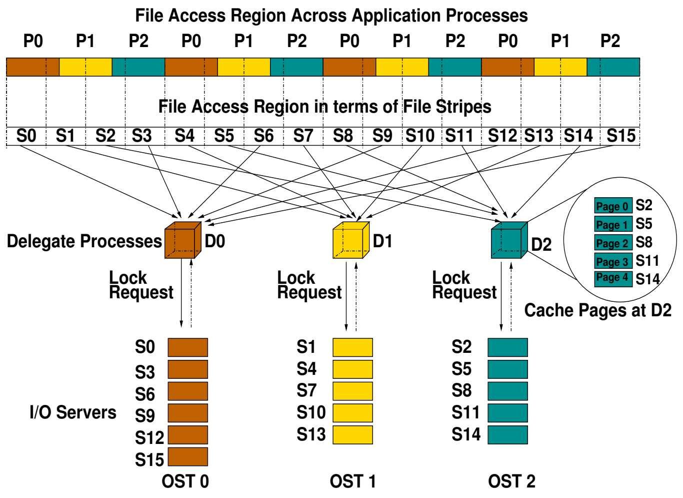

Fig. 3. Lock conflicts can be eliminated by identical partitioning of data across same number of delegate processes and I/O servers. Each of the delegate processes stores data in the form of cache pages directly mapped to the file stripes stored on a unique I/O server. Perfect cache page to file stripe mapping has been shown for the case of D_2-to-OST_2 mapping. This figure shows that with perfect mapping lock acquisition can be reduced to only 1.

A. Static File Domain Mapping

Lock conflict at the file system occurs when two processes compete with each other to acquire the lock to the same file region. We investigated Lustre [18] for exploring it's scalability issues, so that an adaptive solution for large scale systems and their underlying parallel file system can be developed. Modern parallel file systems, in order to meet high data throughput requirements, employ multiple I/O servers each managing a set of disks. Files stored on these systems can be striped across the I/O servers, so large requests can be served concurrently. Due to the nature of file striping, lock granularity is usually set to be the file block or stripe size instead of a byte. Details about locking mechanisms implemented in popular file systems, GPFS and Lustre may be found in Section V-A.

As described in Section V-A, Lustre's locking mechanism is an implementation of extent-based locking protocol. Extent-based locking protocol is implemented such that the I/O server tends to grant locks to as many stripes as possible. For example, on any given server, the first requesting process will be granted a lock over all the file stripes managed by that server. Future requests made by the same client process need not to acquire the lock for those stripes. Second lock acquisition to those stripes, will only be required if a different process has already held the locks to those stripes. Ideally, if we can arrange a one-to-one mapping between the I/O clients and servers, then lock conflicts can be entirely avoided.

Figure 2 illustrates a parallel I/O situation, where lock conflicts occur. In this example, three processes P_0, P_1, and P_2 concurrently write to a shared file, each covering multiple non-contiguous, non-overlapping file regions. The aggregate access region occupies 16 consecutive stripes, S_0, S_1, $\cdots$, S_{15}, which are stored on three I/O servers (Object storage Targets in Lustre), OST_0, OST_1 and OST_2 in a round robin fashion. Data written by different application processes is depicted in different colors. In this figure, each I/O server receives requests from all three processes, which essentially means that each process repeatedly acquires, relinquishes, and reacquires the lock in the midst of accesses from other processes. Considering OST_0, if the first request is made by process P_0 to write stripe S_0, then a lock covering all stripes S_0, S_3, S_6, $\cdots$ S_{15} is granted to P_0. However, if P_1's lock request to stripe S_6 arrives while P_0 is still writing S_0, then locks to stripe S_6 and onward will be relinquished from P_0 and granted to P_1. Later, P_0 must wait behind P_1 for acquiring the lock to S_9. Parallel I/O can cause lock permissions to oscillate from one process to another. In addition, partial accessing stripes S_6 and S_9 results in I/O serialization, given the lock granularity being of a file stripe size. Such conflicts are observed on all other I/O servers in this figure as well. Obviously, the lock conflicts can easily carry away when applications run on thousands of processes. With a large number of processes competing for locks to file stripes, I/O becomes a serious bottleneck for parallel applications [19].

I/O delegate system adopts a new static file domain mapping strategy that aims to minimize file lock conflicts. This strategy divides the whole file into blocks of size each equal to the file system stripe size and statically assigns the I/O responsibilities of the blocks to the delegate processes in a round robin fashion (identical to file system stripping configuration). All file blocks assigned to a delegate process are collectively termed as the **file domain** of this delegate. In order to achieve an optimal mapping between the delegates and servers, we specifically adjust the number of delegate processes to be a factor or multiple of the number of I/O servers. For the same number of delegate processes as I/O servers, each delegate process is uniquely mapped to a single server. When the number of delegate processes is a factor of the number of servers, each delegate is uniquely mapped to a group of servers which serve requests from that delegate only. When the number of delegates is a multiple of the number of servers, a group of delegates is mapped to a unique server which serves no requests other than this group of delegates. Since the mapping is static from one I/O request to another, most of the lock

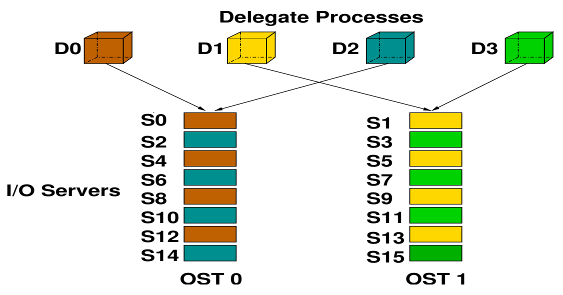

Fig. 4. Lock conflicts are completely eliminated if number of delegate processes are equal or a factor of the number of I/O servers. If number of delegates are more than I/O servers, lock conflicts can occur. To minimize lock conflicts, number of delegates should be kept a multiple of I/O servers. In this example delegate processes are double of I/O servers so each I/O server is shared by only two delegate processes. Lock conflicts can still occur between these two delegate processes but on a reduced level.

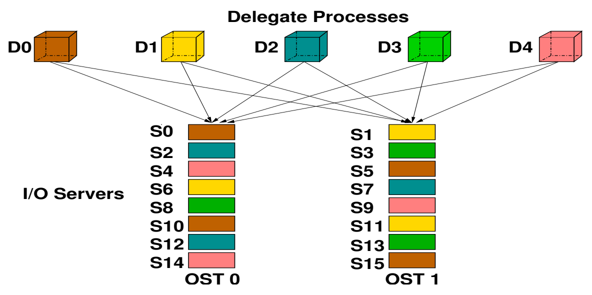

Fig. 5. If delegate processes are not a factor or multiple of I/O servers then all the delegate processes might be accessing all the I/O servers causing serious lock acquisition competition between the processes. In this example, each I/O server is contended by all the delegates which may deteriorate I/O performance.

conflicts can be avoided, given any arbitrary I/O pattern from the clients. Figure 3 shows an example of static file domain mapping on delegate processes D_0, D_1, and D_2 with the same number of I/O servers OST_0, OST_1, and OST_2. Static one-to-one delegate-to-server mapping enables only a unique delegate process requesting lock from a given I/O server. On the first I/O request a delegate process will be granted locks for all the stripes stored by the uniquely mapped I/O server. Therefore, despite the number of application processes and arbitrariness of application's I/O access pattern, there is only one lock acquisition necessary for writing all the stripes in a given I/O server. In this case where the number of delegate processes and number of I/O servers are the same, lock conflicts are completely eliminated.

B. Delegate-to-Server Mapping

Most of the high-performance computing systems, have only a few dozens to a few hundreds I/O servers, which is a small fraction of the total available compute nodes. One can expect that if the number of delegate processes is kept equal to the number of I/O servers, then the performance will not scale beyond thousands of nodes. For I/O delegate system design, the question becomes how we can still avoid lock conflicts or at least keep the conflicts minimal when the number of delegate processes is more than the I/O servers. This section discusses the strategy to minimize the lock conflicts if the number of delegate processes is more than I/O servers.

Figures 4 and 5 demonstrate how two different delegate-to-server mappings affect lock confliction. In both mappings, file domain is logically partitioned in to file stripe sized regions that are statically assigned to the delegates in a round robin fashion. Small write requests can be aggregated at the cache pages and later flushed to the file system. Collaborative caching mechanism enables aggregation of data across the multiple I/O calls, generates stripe sized I/O which matches the stripe boundary of underlying file system, avoids read-modify-write by flushing the cache pages which are already full, and reduces the network communication by keeping I/O size multiple of system page size. Section VI-B describes the details of caching mechanism implemented in the I/O delegation.

In Figure 4, the number of delegates is a multiple of the number of I/O servers. Each server is accessed by unique group of delegates. The potential lock conflicts happen only within the group of delegate processes that map to the single server. Such conflicts can be resolved by exchanging dirty cache pages within the same group of delegates, or coordinating the order of cache page flushing among different groups.

If number of delegates are not a multiple of I/O servers then, each I/O server may receive lock requests from all the delegate processes as shown in figure 5. In contrast to perfect delegate-to-server mapping case (figure 4), lock server needs to resolve the lock conflicts among all the delegates. Therefore, to minimize lock conflicts at the I/O servers, the number of delegate processes are adjusted such that they are always a factor or multiple of the number of I/O servers. Our experimentation conforms that performance is adversely affected if such delegate-to-server mapping is not enforced. Section VII-D evaluates I/O performance for mapped and unmapped delegate-to-server cases.

III. Experiment Results

I/O Delegate System is evaluated on two large production machines; Franklin, a Cray XT4 system at National Energy Research Scientific Computing Center [3] and the TeraGrid Intel-64 Cluster named Abe at the National Center for Supercomputing Applications [12]. Details about experimental setup is given in Section VII.

Performance evaluation consists of comparison of 1) independent MPI-IO with I/O delegation, 2) native MPI independent I/O, and 3) native MPI collective I/O. The latter two native methods use the default MPI library on the machines. We did not explicitly evaluate the MPI collective I/O over the I/O delegation method, because our delegation system treats collective I/O the same as independent I/O. Under the static file domain assignment strategy, data of an I/O request will be split and sent to delegates based on their file offsets. Hence, communication related to collective I/O optimizations will be redundant as delegate system will rearrange data according to the predefined file domain mapping. Therefore, if the collective I/O is changed to use independent I/O underneath, the advantage of I/O delegation can be fully utilized. We expect the significance of I/O delegation system is for independent I/O as independent I/O traditionally performs poorly.

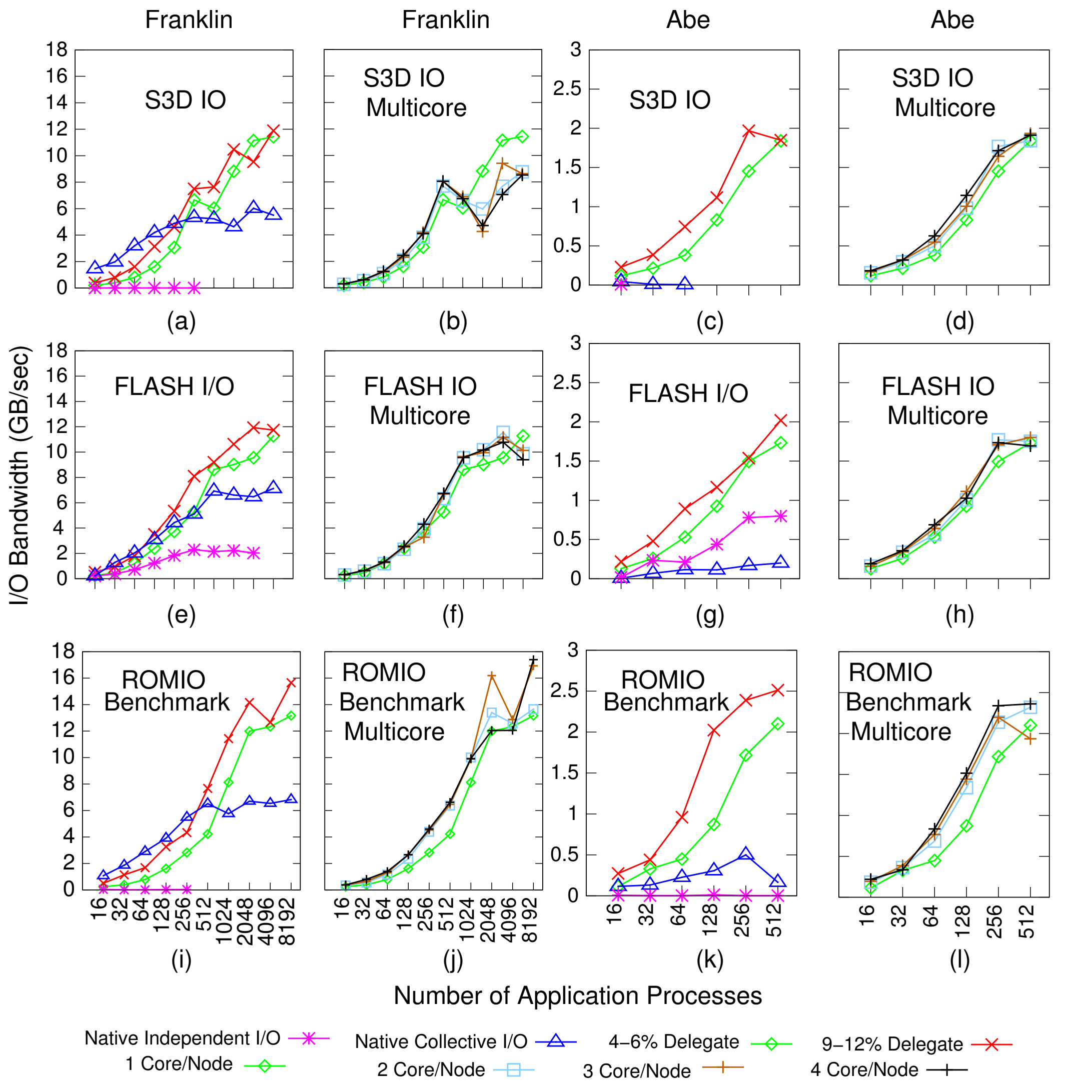

Fig. 6. I/O Performance Evaluation of S3D I/O Kernel, FLASH I/O Kernel and ROMIO Benchmark with Franklin and Abe machines. (a), (e), and (i) show the comparison of write bandwidths of three I/O methods: Independent I/O with I/O delegation, native independent MPI-IO, and native collective MPI-I/O on Franklin. These charts show the effect of changing the ratio of number of delegates to the application processes. Franklin's Theoretical peak I/O bandwidth is approximately 16 GB/sec [3]. I/O delegate provides independent I/O performance scaling up to the peak I/O bandwidth on Franklin. (c), (g), and (k) provide the similar comparison on Abe. (b), (f), and (j) report write bandwidths by utilizing more cores-per-delegate-node with 4-6% delegates allocation. These charts show that there is no advantage in terms of I/O performance by using more than one core-per-delegate. (d), (h), and (l) provide the similar comparison on Abe.

A. *S3D I/O Kernel*

The S3D I/O benchmark is the I/O kernel of S3D [15], a parallel turbulent combustion application using a direct numerical simulation solver developed at Sandia National Laboratories. Section VII-A provides further information about the S3D I/O kernel. For performance evaluation, we keep the sub-array size of globally block-partitioned array along X-Y-Z dimensions, a constant $50 \times 50 \times 50$. This produces about 15.26 MB of write data per process per checkpoint.

Evaluation shows that independent I/O with I/O delegation performs bout twice better than the default MPI collective I/O on Franklin and more than ten time better on Abe. Figure 6 show I/O performance evaluation of S3D I/O kernel on Franklin ((a), (b)) and Abe ((c),(d)) with the increasing number of application processes. Figure 6(a) shows the comparison of native independent I/O, native collective I/O, and independent I/O using 4-6% and 9-12% of additional computer resources as delegate processes. Keeping everything else constant, more delegates perform better because of the bigger cache pool and less communication contention for multiple application processes sending data to the same delegates. On Franklin, the native collective I/O performs better for the case of 256 application processes and less, but the bandwidths flatten thereafter. However, both I/O delegate methods keep scaling up beyond 256 processes. We provide our analysis of this observation in Section V-C. In the case of 8192 processes, we achieves up to two times performance improvement over the native collective I/O with just 4% to 12% of delegate processes. Figures (c) and (d) show the results of similar experimentation setups on Abe. Native independent and collective I/O perform so slow on Abe that their curves are almost coinciding with horizontal axis. On the other hand, the independent I/O using delegation system outperforms both native cases by a significant margin.

The bandwidth numbers obtained on these two machines show a significant difference for the native collective I/O method. The latest Cray MPI-IO adopts a strategy similar to the static file domain assignment that provides much better performance over the traditional collective I/O implementation. More discussion on this aspect is given in Section V-C.

1) I/O Delegation on Multi-core Platform: As modern computers moving toward multi-core architecture, it would be interesting to understand the performance impact of running I/O delegate processes on such systems. One of two possible implementations is to run delegate processes on a group of compute nodes separated from those running the application processes. The other is to run one delegate on one of the cores of each compute node and the rest of the cores for application processes. In this paper, we focus on the former scenario. We choose the 4-6% delegates cases on both machines and compare the I/O bandwidths by varying the number of cores as delegates in each compute node. The charts (b) and (d) show that if all other parameters are kept constant, having different number of cores per delegate node does not make any deterministic difference in I/O bandwidth. In theory, as the number of delegate processes increases, the requests arrived at the same I/O server from different delegates also increase which can potentially cause more the lock contentions. The similar bandwidths of our delegation system with more delegate cores can be explained by the fact that Lustre's distributed lock management scheme is implemented such that locks are held by nodes and not processes. In other words, lock requests originated by all processes belonging to the same compute node do not compete with each other for lock acquisition.

The adoption of static file domain mapping in delegate system aims to improve the costs of `read()/write()` calls made from the application side to the file system. In fact, this strategy reduces such costs so significantly that they no longer dominate the overall I/O performance. To understand the performance bottleneck, we profile the timing in the delegate system. We measured the time spent in the `read()/write()` calls and refer them as I/O time in this section. The rest of the time is referred as communication time, as the operations are mostly data transfer between application and delegate processes. We choose the case of S3D I/O on Franklin with 2048 application processes to investigate the I/O bandwidth trends with the changing number of delegates and number of cores per delegates. The profiling results are given in Figure 7.

Figure 7(a) shows the overall write bandwidth trend with the increasing number of delegates with different number of cores used per delegate. To maintain perfect mapping between delegate processes and I/O servers, the number of delegates are kept a multiple of 48, the number of I/O servers on Franklin. It can be observed that best I/O bandwidth is achieved when only 1 core per delegate node is used. Figures 7(b) and (d) report total time taken in the interprocess communication as a function of number of delegates and the number of cores per delegates respectively. As the total amount of data is kept constant, increasing the number of delegates results into smaller amount of data received by each delegate but more messages passing from application processes to delegates. Such changes of the communication patterns add complexity to the measured communication costs. We observed that in Figure 7(b) with the increase in the number of delegates, mostly communication time decrease except the cases of 48 to 96 delegates for 3 and 4 cores.

Figure 7(d) shows the effect of changing number of cores per delegate nodes while keeping everything else constant. As no other parameter is changed except the number of cores per delegate node, total data received by each delegate node does not change. As the number of cores per delegate node increases, number of messages received by each delegate node increases and size of individual message decreases. So, overall interprocess communication time does not improve as the number of cores per delegate node increases.

Exception is the 96-delegate case where the communication cost increases significantly. The S3D-IO kernel has the write amount proportional to the number of application processes and almost all of the individual write request sizes are not aligned to the file stripe size and hence the lock boundaries. Hence, using different number of delegates can results in different distributions of communication from the application processes to the delegates. We speculate the increasing communication time in the 96 delegates case might attributes to such distribution changes. However, the communication time is also affected by the hardware (hotspots) on the parallel machine and contention from other applications running at the same time (as the inter-process communication network is shared by all applications).

Nevertheless, observations from Figures 7(b) and 7(d) help us conclude that the best practice of I/O delegation configuration is to use only one core in a multi-core platform.

Figures 7(c) and (e) show the effect on I/O with varying number of delegates and number of cores-per-delegate respectively. Figure 7(e) shows that the number of cores per delegate node do not affect the I/O time much. We attribute this to the fact that locks are granted on the basis of nodes and not cores. So, as long as static file domain mapping is maintained on delegate node basis, no I/O time change should be observed. So, we conclude that to obtain a good overall I/O bandwidth (Figure 7(a)) using only one delegate process per node is the best option as additional cores do not provide further benefit.

Figure 7(c) shows very important fact about lock contention at file servers. As discussed in Section II-A, using more delegate nodes than I/O servers may introduce some lock conflict but this chart does not show any definite increase in I/O time when the number of I/O delegates is more than the number of I/O servers. We attributed this observation to the extent based locking mechanism of Lustre file system. As explained in Section V-A, each I/O server is the lock manager of the stripes stored on that server and it grants the locks growing downwards covering all the stripes to the largest uncontended extent. If a couple of requests from the same delegate node reach an I/O server, only first of them needs to acquire the lock and rest of the requests can proceed without any lock acquisition overhead.

For example, for 96 delegates only 2 will be accessing any

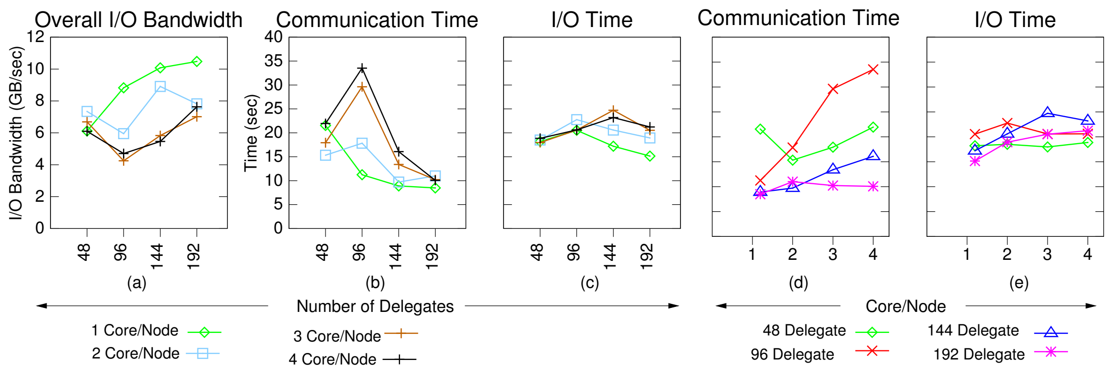

Fig. 7. S3D I/O Kernel, 2048 application processes, Franklin: Breakdown analysis of two major time consuming operations: (i) Data Communication between application and delegate processes. (ii) File system I/O. (a),(b) and (c). Overall I/O bandwidths, time spent in data communication amongst the application and delegate processes, and file system I/O time as a function of number of delegates respectively. (d),(e). Time spent in data communication amongst the application and delegate processes and time for file I/O with varying number of cores-per-delegate-node respectively.

given I/O server at a time writing to alternate file stripes. As shown in the figure 4 if a write requests for S_0 from D_0 arrives at OST_0 before any write request from D_2, then a downward grown lock for all the stripes on this server will be granted to D_0. In case write requests for S_4, S_8 and S_{12} from D_0 also arrive before any write request from D_2 then these additional 3 stripes will be written without any further lock acquisition. While D_0 is writing S_{12} and a request for S_2 arrives from D_2 then a downward grown lock from S_2 to S_{10} will be granted to D_2. So, requests for S_2, S_6, and S_{10} can be serviced with a single lock request. D_2 will have to send another write request to write D_{14} though. Section VII-D further explores the possible lock acquisition patterns to understand the figure 7(c) better. It also includes additional evaluation to compare mapped and unmapped delegate-to-server assignment strategies shown in figures 4 and 5.

An important observation from Figure 7 is that the I/O costs are about the same as the communication. Traditionally, in a parallel I/O operation, the I/O part dominates the entire performance. Optimizations such as two-phase I/O was proposed to addressed this problem by rearranging request data among processes to produce fastest I/O part, i.e. scarifying the interprocess communication for better I/O to the file system. I/O delegation system changes such scenario and raises the attention on the optimization for the communication part. The relative constant I/O cost also explains why increasing the number of delegate from 4-6% to 9-12% does not produce proportional performance improvement.

We conclude here that a substantial post of the maximum I/O bandwidth for an I/O server has been achieved with 4-6% of delegate processes. Any increase in number of delegates hence does not provide linear improvement to the overall performance. This observation also implies that I/O delegation system does not require many delegate processes in order to achieve a scalable performance.

B. *FLASH I/O Kernel*

The FLASH I/O benchmark suite [14] 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 [13]. 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. Further detail of FLASH I/O is given in Section VII-B. In our experiments, we used $16 \times 16 \times 16$ block size. There are 24 variables per array element, and about 80 blocks on each MPI process. So, total of 60 MB data is generated per process.

Figure 6 (e), (f), (g) and (h) show the similar performance trends as S3D-IO benchmark shown in the previous section. In Figures 6(e) and (g), independent I/O with I/O delegate system performs the best on both Franklin and Abe. An interesting observation from Figure 6(g) is that native independent I/O performs better than native collective I/O in Abe's case. This can be explained by the I/O access pattern from each process being already contiguous. As most part of the I/O accesses consists of large contiguous data, synchronization and data exchange has become not as critical as in the S3D-IO case. On the other hand, Figure 6(e) shows that although independent I/O does not perform as bad as in the case of other applications, collective I/O still performs better than independent I/O. We attribute this observation to the new collective buffering algorithm used for collective I/O on Franklin. Figures 6(f) and 6(h) show the effect of using multiple cores per delegate nodes on Franklin and Abe respectively. As discussed in Section III-A, using multiple cores per delegate node has no significant impact to the I/O performance.

C. *ROMIO Benchmark*

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. 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 set the subarray size to $100 \times 100 \times 100$. In order to get stable performance numbers, we measured ten iterations of the write operations. So, total of 38.15 MB data is generated per process.

Figure 6 (i), (j), (k), and (l) show the similar performance results as the S3D-IO and FLASH I/O cases. Figures 6(i) and (k) show that, independent I/O with the proposed I/O delegation performs best among the native collective I/O and native independent I/O on both Franklin and Abe. Native independent I/O performs very poorly, but when used with the I/O delegation architecture, its performance is improvement significantly on both machines. Figures 6 (j) and (l) show the effect of using multiple cores per delegate nodes on Franklin and Abe. As discussed in section III-A, using multiple cores per delegate node does not affect the I/O performance.

IV. Conclusions and Future Work

We have proposed an I/O software architecture, I/O delegation system with static file domain mapping for large-scale parallel applications and file systems. The proposed architecture bridges the gap between modern scientific applications' requirements and old fashioned parallel storage protocols. For many high performance scientific applications independent I/O is becoming critical, particularly for the applications whose data is dynamically created or irregularly partitioned amongst processes. For very large scale systems, global process synchronization may not be feasible for such data partitioning patterns.

Performance evaluation demonstrates very high I/O bandwidth for independent I/O which outperforms even optimized collective I/O. I/O delegate system can be used by parallel I/O library, such as MPI-IO, and enabled by automatically detecting the underlying system configurations like stripe count, stripe size, and stripe offset to choose the most optimal values of cache page size and number of cores per node and achieve optimal performance. The best practice for I/O delegation configuration is to produce perfect delegate-to-server mapping that requires choosing the number of delegates being either a factor or multiple of underlying stripe count.

We have observed that using multiple delegate processes per node does not provide any noticeable I/O benefit over single delegate process per node. This observation implies the extra compute cores can be used for computation best run closely to where data reside, such as data analytics, statistical operations, and subsetting operations. These extra compute cores can also be utilized for running application processes, thus reducing the overall resource allocation significantly.

We have demonstrated experimentation evaluation up to a few thousand application processes with 4-12% of delegates. For even larger application size, number of delegates may grow to a few thousands. For such a large number of delegates, lock contention between a larger number of delegate processes may also emerge. In order for I/O delegate system to scale for very large number of application processes, we plan to investigate new methods for lock conflicts avoidance.

We believe that scientific applications involving parallel reads can benefit from I/O delegate system. Collaborative caching on delegate processes can provide the benefits of read ahead as file system reads are performed on stripe basis and data is cached in memory. This prefetching mechanism can save many read requests from traveling across the network over to the parallel file system. We plan to study read performance of I/O delegation system with different applications.

V. Acknowledgements

This work was supported in part by NSF award numbers: CCF-0621443, SDCI OCI-0724599, CNS-0551639, IIS-0536994, and HECURA-0938000. This work was also partially supported by DOE grants DE-FC02-07ER25808, DE-FG02-08ER25848, DE-SC0005309, and DE-SC0005340.

References

[1] Teragrid Infrastructure. http://www.teragrid.org.

[2] Jaguar (Cray xt5). http://www.nccs.gov/computing-resources/jaguar/.

[3] Franklin (Cray xt4). http://www.nersc.gov/nusers/resources/franklin/.

[4] Rajeev Thakur, Robert B. Ross, and Robert Latham. Implementing byte-range locks using mpi one-sided communication. In Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *PVM/MPI*, volume 3666 of *Lecture Notes in Computer Science*, pages 119–128. Springer, 2005.

[5] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[6] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel i/o via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, 1993.

[7] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *In Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1998.

[8] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard, Version 1.1*, June 1995. http://www.mpi-forum.org/docs/docs.html.

[9] M. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics*, 53:484–512, Mar. 1984.

[10] 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.

[11] Wei keng Liao, Avery Ching, Kenin Coloma, Arifa Nisar, Alok Choudhary, Jackie Chen, Ramanan Sankaran, and Scott 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.

[12] Abe (teragrid intel-64 cluster) . http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64Cluster/.

[13] 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*, page 131273, 2000.

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

[15] 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*, 46:38–42, September 2006.

[16] Dick Oswald David Knaak. Optimizing MPI-IO for Applications on Cray XT Systems. White paper, Cray Inc, May 2009. Available online (20 pages).

[17] Kamil Iskra, John W. Romein, Kazutomo Yoshii, and Pete Beckman. Zoid: I/o-forwarding infrastructure for petascale architectures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 153–162, New York, NY, USA, 2008. ACM.

[18] High-Performance Storage Architecture and Scalable Cluster File System White Paper. White paper, Sun Microsystems, Inc., October 2008. Available online (20 pages).

[19] C. William McCurd, Rick Stevens, Horst Simon, William Kramer, David Bailey, William Johnston, Charlie Catlett, Rusty Lusk, Thomas Morgan, Juan Meza, Michael Banda, James Leighton, and John Hules. Creating Science-Driven Computer Architecture:A New Path to Scientific Leadership. Technical report, National Energy Research Scientific Computing Center, October 2002.

Arifa Nisar has received her BSc degree from Department of Electrical Engineering at University of Engineering and Technology Lahore, Pakistan in 2003. She received her Ph.D from Department of Electrical Engineering and Computer Science, Northwestern University in 2010. Arifa is currently a NSF/CRA Computing Innovation Fellow at Storage Systems Research Center, University of California Santa Cruz. Her main research interests include high performance I/O systems, parallel I/O, and file systems.

Wei-keng Liao is a Research Associate Professor in the Electrical Engineering and Computer Science Department at Northwestern University. His research interests are in the area of high-performance computing, parallel I/O, data mining, data management for scientific applications.

Alok Choudhary is a Professor in and the Chair of the EECS department and a Professor at the Kellogg School of Mgmt at Northwestern University. From 1989-1996, he was a faculty member in the ECE department at Syracuse University. Alok Choudhary received his Ph.D. from University of Illinois, Urbana-Champaign, in Electrical and Computer Engineering (1989), an M.S. from University of Massachusetts, Amherst, (1986) and B.E. (Hons.) from Birla Institute of Technology and Science, Pilani, India in 1982. Dr. Choudhary was a co-founder of Accelchip Inc. and was its Vice President for Research and Technology from 2000-2002. Dr. Choudhary has published more than 350 papers in various journals and conferences. Dr. Choudhary serves on the editorial boards of: IEEE Transactions on Parallel and Distributed Systems, Journal of Parallel and Distributed Systems and International Journal of High Performance Computing and Networking. His research interests include: High-performance computing and communication systems, power aware systems, computer architecture, high-performance I/O systems and software and their applications in many domains including information processing (e.g., data mining, CRM, BI) and scientific computing (e.g., scientific discoveries). Further interests lie in the design and evaluation of architectures and software systems (from system software such as runtime systems to compilers), high-performance servers, high-performance databases and input-output and software protection/security. He has also written a book and several book chapters on these research interests.

Supplementary Document: Delegation-based I/O Mechanism for High Performance Computing Systems

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

V. Background and Related Work

This work is inspired by the observation that file systems are unable to cater for a group of related clients accessing shared files. This hampers their ability to fine-tune the consistency control mechanism to meet I/O requirements optimally. We believe that lack of proper programming interfaces to the file systems prevents applications from passing down their I/O intents that could be very useful in optimizing I/O performance. For example, a group of processes writing a partitioned global array in parallel should be considered as a group of correlated I/O requests by the file system. In this case, data consistency control should only protect the file data from processes outside of the group, instead of the processes inside. Without a proper way to detect such cases, the file system is forced to protect individual request regardless of the client's group membership.

Under these circumstances, we believe that an I/O layer sitting in between application processes and file system is necessary to capture the missing information and potentially use it to enhance I/O performance. For understanding the characteristics of parallel file systems on supporting data consistency for concurrent requests to shared files, we investigated the file locking protocols and their implementations on existing file systems. Since different file systems may not use the same locking mechanism, it is important that this I/O layer adapts to their distinct features in order to produce the best I/O strategy.

A. Distributed Lock Management in Parallel File Systems

Modern parallel file systems, in order to meet high data throughput requirements, employ multiple I/O servers, each managing a set of disks. Files stored on these systems can be striped across the I/O servers, so large requests can be served concurrently. Due to the nature of file striping, lock granularity is usually set to be the file block or stripe size instead of a byte. If two I/O requests fall into the same lock granularity region and at least one of them is a write, they must be carried out serially even if they do not overlap in bytes. File systems rely on a locking mechanism to provide a client with an exclusive access to a file region and hence to implement the data consistency control. The implementation of a distributed file locking system aiming at reducing the lock acquisition frequency, varies among different file systems. Many parallel file systems, such as IBM's GPFS [1], [2] and Lustre [3], [4], adopt an extent-based locking protocol in which a lock manager tends to grant access to the largest possible file region. For example, the first requesting process to a file is granted the lock for an entire file. When the second write from a different process arrives, the first process will relinquish a part of the file to the requesting process. If the starting offset of the second request is ahead of the first request's ending offset, the relinquished region will start from the first request's ending offset toward the end of file. If not, the relinquished region will contain a segment from file offset 0 to the first request's starting offset. The advantage of this protocol is that a process's successive requests within the already granted region would require no lock request.

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 [1]. 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.

Lustre, a POSIX compliant file system, respects POSIX I/O atomicity semantics. To guarantee I/O atomicity, file locking is used for each read/write call, allowing only exclusive access to the requested file region. Lustre file system stripes a file in round robin fashion across the file servers. Lustre uses a distributed locking protocol where each I/O server manages locks for the file stripes it stores. Extent based locking is performed on the stipes stored at any individual I/O server. On an I/O request, I/O server grants the locks growing downwards covering all the stripes to the largest uncontended extent [5]. If a client requests a lock held by another client, a message is sent to the lock holder requesting to release the lock. Before a lock can be released, dirty cache data must be flushed to the servers. On parallel file systems like Lustre and GPFS, where files are striped across multiple I/O servers, conflicted locks can significantly degrade parallel I/O performance [6] and hence it is important that an I/O middleware recognizes the file system's locking behavior and minimizes lock conflicts. Our proposed work is motivated by such needs and designed to generate the I/O pattern which performs best with the underlying file system's locking mechanism. We use Lustre to demonstrate the impact of perfectly matched I/O access patterns with locking boundaries of underneath file systems.

B. MPI-IO

MPI defines a set of programming interfaces for parallel file access, commonly referred as MPI-IO. With this framework, many optimizations such as two-phase I/O [7] and data sieving [8], have been successfully demonstrated significant performance improvement for the parallel I/O. One of the prominent software contributions is the collective I/O functionality proposed in the message passing interface (MPI) standard [9]. In addition to two-phase I/O [7], many collaboration strategies have been proposed and demonstrated their success, including disk directed I/O [10], persistent file domain [11], [12], view based collective I/O [13], collaborative caching [14], [15], layout awareness [16] etc.

There are mainly two types of I/O access functions in MPI-IO: Collective I/O and Independent I/O. Collective functions require collaboration among processes to rearrange I/O requests for achieving better performance. This collaboration incurs the overhead of process synchronization but it provides significant performance improvements over uncoordinated I/O. ROMIO[17] implements collective I/O calls using the two-phase I/O method, which comprises of the request redistribution and I/O phases. Two phase I/O's implementation for collective functions is explained in figure 8. The implementation first calculates the aggregate access file region and then evenly partitions it among the I/O aggregators into **file domains**. The I/O aggregators are a subset of the processes which act as I/O proxies for all of the processes. In the redistribution phase, all processes exchange data with the aggregators based on the calculated file domains. If data to be distributed is larger than the maximum buffer size, collective I/O operation is decomposed into multiple steps of two-phase I/O. In the I/O phase, aggregators access the shared file within the assigned file domains. Two-phase I/O can combine multiple non-contiguous requests into large contiguous ones. This approach has been demonstrated to be very successful as modern file systems handle large contiguous requests more efficiently. On the parallel machines where each compute node contains a multi-core CPU or multiple processors, ROMIO, by default, picks one of the core/processor as the aggregator in every node.

Independent I/O calls, on the other hand, do not require process synchronization and hence lack the opportunity to exchange requests. Therefore, application users community is discouraged to use independent I/O citing its poor performance. However, not all scientific applications can afford process synchronization due to the irregularity of their data distribution and creation. For instance, when several global arrays are partitioned among the different groups of processes, synchronization I/O for a global array requires all processes to participate even for those groups that do not contain any data for this array. In such a situation, synchronization serializes I/O. Mostly, process synchronization may not even be possible as new data objects are created dynamically, and one process may not have any information about the data on a different process. Examples of such I/O pattern are the applications using Adaptive Mesh Refinement (AMR) algorithm. For these applications, independent I/O may be the only choice and it is important that the I/O systems provide performance similar to the collective I/O.

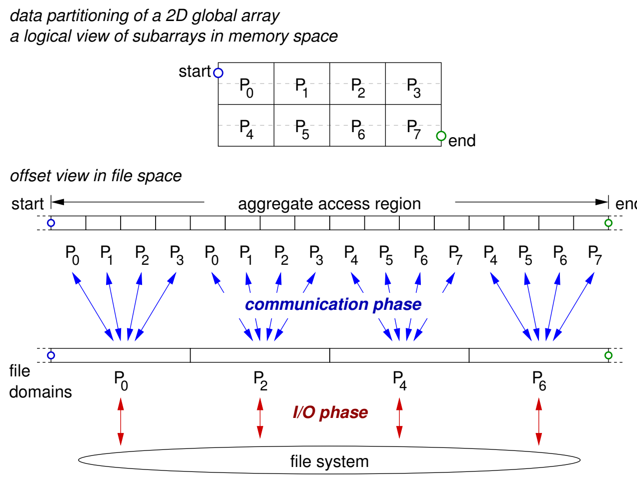
data partitioning of a 2D global array
a logical view of subarrays in memory space
start
P_0 P_1 P_2 P_3
P_4 P_5 P_6 P_7
end
offset view in file space
start
aggregate access region
end
P_0 P_1 P_2 P_3 P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_4 P_5 P_6 P_7
communication phase
file domains
P_0 P_2 P_4 P_6
I/O phase
file system

Fig. 8. Two Phase Implementation for MPI Collective I/O: In the first phase aggregate access region is evenly divided among the chosen processes termed as aggregators. In the second phase aggregators complete the file system I/O by performing the actual file system operations.

Traditional collective I/O does not have persistent file domain assigned to the aggregators. Every I/O access is treated individually and the access region specific to the I/O call is partitioned to the aggregators. Collective I/O is also limited by the maximum allowed size of the temporary buffer. If access region per process is larger than the maximum buffer size (16 MB is default) then data exchange is performed in multiple stages. Collective I/O generates large disjoint contiguous accesses to the file system and does not consider underlying file system locking strategy.

I/O delegation system allocates a small set of additional nodes to handle I/O responsibilities. I/O delegation system aims to minimize file lock conflicts and improve the MPI independent I/O performance. A file caching mechanism is implemented in the delegate system that enables data aggregation across multiple requests aiming for improving MPI independent I/O performance. This feature is also considered an optimization that spans multiple MPI-IO requests, collectives and/or independents, which have been ignored by existing MPI-IO optimizations. I/O delegate performs asynchronous data communication in a single step. I/O delegate system employs a static file domain mapping method that statically maps evenly partitioned file regions to the delegates such that the data layout is perfectly matched with underlying the file system.

C. Cray XT MPI Optimizations

In our experiments, we observed significant performance difference for the native MPI collective I/O method between Franklin and Abe. The two-phase I/O implementation in the MPI-IO library installed on Abe uses the traditional file domain partitioning method. This method divides the aggregate access file region of a collective I/O into contiguous, disjoint subregions, each assigned to an I/O aggregator. However, this strategy performs poorly on Lustre as investigated in [18].

Very recently, a new collective buffering algorithm used by the Cray MPI-IO library [19], [20] similar to the static file domain partitioning method proposed in [18] was made available on Franklin[21] and successfully demonstrated a dramatic performance enhancement. In traditional collective I/O, file domain is assigned at the time access. Access region is partitioned in disjoint regions and each region is assigned to an aggregator. In contrast to the traditional collective I/O a persistent file domain is assigned to the aggregators.

The idea of keeping a static and optimal mapping between the I/O processes and file servers is the key to scalable parallel I/O performance on Lustre file system. However, there are two limitations on the current design of the Cray's collective buffering algorithm. First, the default number of I/O aggregators is set to equal to the number of I/O servers no matter how large the number of application processes is. When the number of application processes is much larger than the I/O servers, the communication contention for rearranging request data to the I/O delegates can easily become the performance bottleneck. Second, the largest file access region that can be processed by a single two-phase I/O is equal to the file stripe size times the number of I/O servers.

For instance, when the file stripe size is set to 1MB on Franklin, the maximum file access region per two-phase I/O is only 48 MB. For requests with much larger aggregate file access region, this limitation will produce many two-phase I/O stages and each covers a file region no greater than 48 MB. Under such circumstance, this algorithm may incur higher overhead of process synchronization and communication.

From Figure 6(a) we observe that the native collective I/O on Franklin fails to scale beyond 512 application processes. On the contrary, the I/O delegate approach scales much better. The reasons behind can be the 48 MB file access limitation and the delegate system being able to aggregate small requests in the caches. Also, delegate can improve I/O performance across multiple access calls. Cray XT MPI does not allow more than 48 aggregators in the interest of removing lock contention. For very large number of application processes, only 48 aggregators may become communication bottleneck and hamper scalability.

File domain assignment for Cray MPI is similar to the file domain assignment in the delegate system. I/O delegate provides a way to minimize lock contention in case there are more delegates than servers. Other than the scalability problem, the new collective buffering only benefits the collective writes and is not applicable to collective reads or independent I/O. As explained in section VI-A, I/O delegate performs asynchronous data communication in a single step. Each application process sends data over to any delegate in a single MPI send only.

D. I/O Delegation and I/O Forwarding

I/O delegation with file caching framework (IODC) [22] provides an infrastructure where all the I/O accesses are pushed through a small number of additional compute processes (referred to as I/O Delegate processes). This infrastructure creates an intermediate layer between application and file system. IODC reduces lock contention by limiting the number of processes accessing the underlying parallel file system. It also employs a collaborative file caching subsystem to enable data aggregation, page migration, and request sequential consistency control. The delegation layer is implemented in ROMIO and hence transparent to the regular MPI-IO programs. Performance evaluation of this work demonstrated noticeable improvements for collective I/O on both Lustre and GPFS. In this paper, our proposed approach extends the I/O delegation concept and focuses on the file locking characteristics of the underlying file system. Moreover, in addition to collective I/O, our solution is also directed towards independent I/O.

In pursuit of avoiding the I/O bottleneck at storage systems, architectures like BlueGene, have brought I/O nodes closer to parallel storage layer. The IBM BlueGene systems adopt a new I/O architecture specially designed to reduce the scale of I/O contention. The new I/O sub-system consists of a group of additional I/O nodes physically situated in between the compute nodes and file system servers. Compute nodes on a BlueGene are organized into separate processing sets, each equipped with an I/O node. I/O requests from the compute nodes on the same processing set are accomplished via the I/O node [23], [24]. From file system's point of view, I/O nodes are the actual clients to the file system. Hence, data consistency semantics are enforced on the I/O nodes. Other existing contributions have also recognized the importance of using a middleware to coordinate parallel I/O requests by reducing potential conflicts before data reaches file system. Different system level solutions have been proposed to accomplish I/O forwarding between compute nodes and I/O nodes. CIOD (Control and I/O Daemon) [25], is a light weight kernel for I/O nodes developed at IBM. It receives I/O requests forwarded from the compute nodes over the collective network and invokes corresponding Linux system calls. ZOID (ZeptoOS I/O Daemon) [26], a function call-forwarding infrastructure developed at Argonne National Lab, is integrated into the ZeptoOS software stack [27]. Both of these I/O forwarding components allow communication between statically mapped compute nodes and I/O nodes only. They do not facilitate the intercommunication between I/O nodes, or flexibility between compute and I/O nodes interactions. Such inflexible I/O architectures may lose all the high level I/O information as well as the opportunity of any optimization at I/O nodes layer.

E. File Domain Partitioning in Collective I/O

Recent research has shown the importance of adjusting parallel I/O requests with the file system's locking boundaries [28], [18], [29]. Several file domain partitioning methods have been proposed and evaluated in [18]. The stripe-boundary alignment method appears to be the best choice for GPFS. This method aligns the partitioning of the aggregate access file region with the GPFS's file stripe boundaries which results in large contiguous requests to the file system. On Lustre, the static and group-static partitioning methods outperform other methods with significant margins. The static method assigns the file domains based on the stripes stored on the I/O servers by keeping the mapping of the I/O processes to the servers persistent. Since the client-server mapping does not change from one collective I/O call to another and the number of accessing clients per server is minimized, this method eliminates the possible lock conflicts. We adopt the static file domain strategy in our I/O delegation system, expecting that the lock conflicts from the delegate processes to the file system can be minimized.

Sanchez et. al [30] proposed an I/O proxy based I/O architecture, which uses local disks to implement an intermediate file system between application and parallel storage system. This architecture uses the local file system to perform some optimizations before data is flushed to parallel file system.

Panda [31] is a server-directed I/O strategy, in which one compute node and one I/O node act like master client and master server. Master client and master server exchange the layout of in memory and on disk data distribution to determine the optimized way of transferring data between clients and I/O nodes.

Collective buffering approach [32] rearranges requests in processors' memory, to initiate optimized I/O requests, thus reducing 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 [33], [34], which uses separate processors called 'coalescing nodes' to perform I/O optimization by joining small I/O operations. This approach requires application support to provide out-of-core data information in order to combine the contiguous data on disk.

VI. Implementation of I/O Delegate System

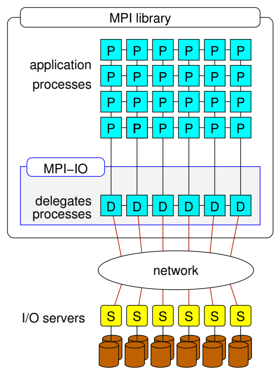

Fig. 9. I/O Delegate Architecture: It is a portable I/O middleware integrated inside the MPI-IO layer. A small percentage of application processes is allocated in addition to the required application processes (P). These additional resources termed as delegate processes (D) perform all the I/O operations like `open()`, `write()`, `read()`, `sync()` and `close()` on the behalf of application processes.

The I/O delegation system is implemented in ROMIO library, so it can be available to all the MPI-IO applications and is portable across different file systems. This system is activated by separating all the MPI processes allocated by a parallel job into two disjoint groups, one running the application and the other running the I/O delegate system. The delegate system emerges as an intermediate layer between application and parallel storage system. Compute processes running on this intermediate layer are called delegate processes. The number of delegate processes is kept no more than a small fraction of total compute processes executing the parallel application. Figure 9 illustrates the overall system architecture. All the I/O operations initiated by the application processes pass through the delegate processes which perform respective I/O operations on behalf of the application processes. Current implementation requires users to explicitly allocate additional processes for I/O delegation when submitting a parallel job. At the start of the application, the number of delegates is taken as an input parameter and automatically adjusted to match the number of I/O servers of underlying file system, so the number of delegates is either a factor or multiple of the number of servers, unless otherwise specified. Our current implementation does not change the number of delegates during the life time of the application. The entire MPI processes allocated by a single MPI job are split into two separate communicator groups, one for the delegate processes and the other for the parallel application, which exchange I/O and related information with each other using MPI inter-communicators.

To make this implementation generic, we use MPI dynamic process management functionality for initial communicator setups. For machines that have not yet supported the dynamic process management, such as Cray XT, we use the traditional communicator construction functions, such as MPI communicator split, to separate the two communicators.

During the MPI application's execution, all I/O requests from the application processes are redirected to the I/O delegate processes, limiting the file system interactions to delegate processes only. Reduction in the number of compute nodes accessing the storage system reduces the scale of overall I/O contention at storage system. Delegate processes continuously poll on incoming requests from application processes as well as from peer delegate processes. Application processes send requests, such as file open, write, read, and close, to delegate processes, and delegate processes collaborate with each other to perform I/O bookkeeping and optimizations. The lifetime of delegate processes is mapped to parallel applications execution time only. However, the idea of such I/O architecture can be extended to a set of physical compute nodes persistently serving requests from all the applications. As described earlier in Section V, the IBM BlueGene systems have already been configured with such I/O layer in hardware. To explore the maximal potential of such architecture, it's necessary to make the software layer aware of the MPI processes running a single program and treating them as an integrated I/O client. Following sections discuss various components of I/O delegation system in detail.

A. I/O Request Flow

All the delegate processes run an infinite loop that keeps polling incoming requests from both the application and delegate processes using respective inter- and intra-communicators. When a file is collectively opened by a group of application processes, only delegate process 0 creates the file and broadcasts the open request to the rest of delegates. On receiving the open request, all delegate processes open the file locally and initializes the data structures for I/O delegation. A unique global ID associated to local file ID at the delegates is returned to the clients, so it can be used for future references to this file. The metadata of a read/write request is packaged by each application process into an MPI message containing the information of file ID, request size, and an array of requesting file offset-length pairs if the request consists of multiple disjoint file regions. When a delegate process receives this message, it allocates proper memory space to receive the metadata, as well as the cache pages to accommodate the write/read data. For write request, metadata is sent to a delegate process followed by the write data. The write data is sent by using an MPI derived data type to pack noncontiguous data, so the communication can be completed in a single MPI send call. Delegate process separates the disjoint request segments based on the offset-length metadata and copies them to their respective location in the cache pages. The byte number of data received is sent back to the application process as the return value. I/O delegate system adds an extra step of passing data through delegate nodes, which incurs some extra data communication cost. But as our implementation does not use the optimizations implemented in ROMIO, we justify this communication overhead by saving two-phase I/O's synchronized communication overhead. We have also implemented an alternate approach that packs the write request metadata along with the actual data in a single message. Our experimentation shows that these two approaches perform about the same, so we selected two-messages approach and present its evaluation results. For read request, the operations are simply reversed. Data are fetched in units of file stripes and read data is also cached at the delegate processes. The file close operation is similar to file open, where delegate process 0 acts as a coordinator for all the delegate processes.

B. File Caching

We incorporate a file caching mechanism into the I/O delegation layer. Although caching is considered to be beneficial mainly for repeated data access, this caching mechanism is the essential component of I/O delegate layer aiming to improve both write and read performance.

With the feature of file caching, small write requests can be aggregated at the cache pages and later flushed to the file system. The size of I/O operations to the file system are in the units of cache page size, in our case also the file stripe size. Similarly, small read requests to a single file stripe will result in only multiple of stripe size read request at the delegate process. File domain is logically partitioned in to file stripe sized regions that are statically assigned to the delegates in a round robin fashion. Due to the use of static file domain strategy, local cache pages stored at an I/O delegate perfectly map to file stripes handled by a unique I/O server. Figure 3 illustrates an example of such mapping for delegate D_2 to server OST_2.

The caching policy used in our previous work [22], is a greedy algorithm that caches the first requested data on the delegates regardless the locking protocol implemented by the underneath file system. Our new delegation implementation incorporates the ideas of taking the file system locking behavior into concern. Static file domain assignment to the delegates ensures that there is only one copy of file data. Metadata information associated with a cache page is maintained by the same delegate that holds the cache page. The fact that only one delegate has access to the caching information of a file stripe, eliminates the need of distributed locking mechanism like the one proposed in [35], [36], [22]. In the absence of locking requirements, no data communication is required amongst the delegates for caching operations.

The caching mechanism keeps tracks of dirty segments in each file stripe in the form of offset-length pairs. Coalescing of two consecutive dirty ranges in the same stripe is performed when a new request accesses the cached stripe. Coalescing stops when a cache page is fully dirty. When flushing a cache page, if it contains more than one dirty segment, a read-modify-write will be performed. This approach allows one read and one write per file stripe in the worst case and helps avoiding unaligned I/O access by flushing partially filled cache pages.

In addition to avoiding lock conflicts by using static file domain mapping, we have achieved many performance benefits from our caching design. The caching mechanism enables aggregation of data across the multiple I/O calls, generates stripe sized I/O which matches the stripe boundary of underlying file system, reduces read-modify-write operations and hence the client-server communication cost.

C. Running I/O Delegates on Multi-core Compute Nodes

Modern high-performance computing systems are heading towards constructing multi-core compute nodes architectures. It would be interesting to explore the performance impact of running I/O delegates, each on a single core of a multi-core compute node. For file system perspective, all processes running on a single compute node are handled by the sole copy of client-side file system on that node, so lock requests coming from different processes on the same node do not cause any conflicts. We enable the I/O delegate system in such a way that when more than one core per nodes are used as delegate processes, the file domain assignment conforms the mapping of the I/O servers to the delegate nodes, instead of delegate cores. For example, in Figure 3, multiple cores working as delegate processes in delegate node D_0 are still assigned stripes S_0, S_3, S_6, $\cdots$. No matter how many cores per nodes are used, file domain assignment remains same at the delegate node level. This assignment guarantees no new lock conflicts that would occur among the processes within the same delegate node, while the I/O workload is shared by more delegate processes.

D. MPI-IO Semantics

MPI-IO data consistency requirements differ from that of POSIX's [37]. POSIX's semantics require that by the time a write operation is returned, all other processes should maintain sequential consistency and atomicity. On the other hand, MPI-IO semantics require that by the time a write is returned, only the processes in the same communicator group are guaranteed to maintain semantics. Since I/O delegation system is integrated in ROMIO, it is important that the MPI-IO data consistency is not broken. The data cached at the I/O delegate processes is available to all the application processes that collectively open the shared file.

VII. Experimentation

I/O Delegate System is evaluated on two large production machines; Franklin, a Cray XT4 system at National Energy Research Scientific Computing Center [21] and the TeraGrid Intel-64 Cluster named Abe at the National Center for Supercomputing Applications [38]. Table VII describes the technical summary of Franklin and Abe, as well as the file system configurations used in evaluation. For performance evaluation, we used one artificial benchmark from ROMIO test programs, and two I/O kernels from production applications FLASH and S3D. The I/O bandwidth numbers were calculated by dividing the aggregate I/O amount by the time measured from the beginning of file open until after file close.

For all three I/O applications, each process writes a fixed size of data to the shared file(s). Thus, the total data size to be written increases proportionally as the number of processes. Although no explicit file synchronization is called in these benchmarks, closing files flushes all the dirty cache data. We have collected results up to 8192 application processes on Franklin and 512 application processes on Abe. I/O delegation system was evaluated with 4-6% and 9-12% additional compute resources allocated as delegate processes. For evaluation purposes we have used 4 cores per compute node for application processes, and 1 to 4 cores per node for delegate processes, while the number of delegate nodes are kept either a factor or multiple of the number of file system I/O servers. Section VII-D from supplementary document demonstrates the change in performance when number of delegate nodes are co-prime to the number file servers.

A. S3D I/O

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.

B. FLASH I/O

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 [39], 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 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.

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.

C. Cache Eviction

This section provides evaluation and analysis of cache pages eviction in I/O delegate. A full-dirty page is marked with a high priority for flushing and are first ones to be evicted under memory usage pressure. The metadata of each cache page contains a time variable to record the last access time. For page eviction, a least-recently-used (LRU) policy is used amongst the fully dirty pages. At the file close, all the dirty cache pages, fully dirty by now, are flushed to the file system. If a page is to be evicted before it is fully dirty then only the dirty segment is flushed. When flushing a cache page, if it contains more than one dirty segment, a read-modify-write will be performed. This approach allows one read and one write per file stripe in the worst case and helps avoiding unaligned I/O access by flushing partially filled cache pages.

We have performed additional experimentation to observe the effect of varying memory pressures on overall I/O performance. S3D I/O kernel is used with the sub-array size of globally block-partitioned array along X-Y-Z dimensions, a constant $50 \times 50 \times 50$. This produces approximately 15.26 MB of write data per process per checkpoint. Keeping all other parameters constant, we reduce the cache pool size to trigger eviction.

TABLE I
COMPARISON OF TECHNICAL SPECIFICATIONS BETWEEN FRANKLIN AND ABE

Specification	Franklin	Abe
Number of Compute Cores	38,288	9,600
Processor Cores per Node	4	8 (Dual socket quad core)
Number of Compute Nodes	9,572	1,200
Processor	Intel 64, 2.33 GHz	Opteron 2.3 GHz Quad Core
Memory	8 GB/node (2 GB/core)	8 GB/node (1 GB/core)
Network Interconnect	SeaStar2	InfiniBand
Compute Node Operating System	Compute Node Linux (CNL)	Red Hat Enterprise Linux 4 (Linux 2.6.18)
Parallel Programming Models	Cray MPICH2 MPI	MVAPICH 2 (v. 2-1.2)
File System	Lustre v. 1.6.5-2 (Two /scratch file systems; 436 TB)	Lustre (100 TB) v. 1.6
Number of OSTs	48 for each scratch (used 48)	180 (used 128)
Theoretical IO Bandwidth	350 MB/sec x 48 = **16.4 GB/sec**	-
File Stripe Size (Smallest Lock granularity)	1 MB	1 MB

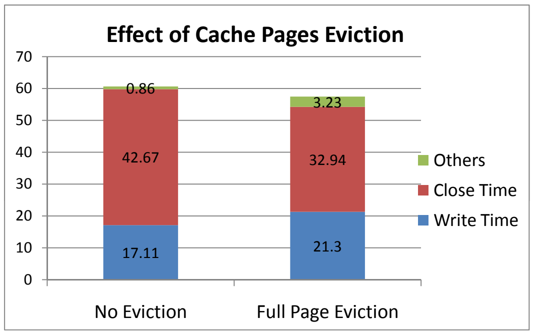

Fig. 10. Time distribution of I/O operations of S3D I/O kernel on Franklin under varying cache pool sizes. Number of Checkpointing Files = 10, Number of Application Processes = 4096, Number of Delegates = 192, Data Generated = 610.35 GB, Data/Delegate/File 325.12 MB (No Eviction). Cache pool size is 2GB/delegate so data fits in the cache and flushed at `MPI_file_close()` only. `MPI_file_write()` time mainly encompasses communication, cache management, memory copy etc. (Full Page Eviction). Cache pool size is 256MB/delegate, so cache page eviction occurs during `MPI_file_write()` calls. Eviction of Fully dirty pages alleviates some flushing cost at the time of `MPI_file_close()`. Overall I/O time remains the same.

If the total data received by a delegate do not fit in the cache pool then some of the pages are flushed to file system during `MPI_file_write()`. If data fits in the delegate cache pool then cache pages are flushed during `MPI_file_close()`. Figure 10 shows time spent in `MPI_file_write()` and `MPI_file_close()` with and without cache pages eviction. In this evaluation, data received by each delegate is approximately 325.12 MB. Cache pool size per delegate is varied from 2GB to 256 MB.

For first case, cache pool size is big enough to not to cause any eviction before file close. We can see that only a small fraction of the total time is spent in performing `MPI_file_write()` calls. File close takes most of the time as in the absence of eviction, all the I/O is performed at file close. In this case I/O accesses at close time are file stripe aligned which is the lock boundary for Lustre.

In second case, cache pool size per delegate is limited to 256 MB. In this case eviction will be triggered during `MPI_file_write()` operation to accommodate incoming accesses. Delegate cache may be able to hold approximately maximum of 78% of the total data in the memory at a given time. Eviction policy is such that the fully dirty cache pages are chosen to evict. In this example an average of 69.5 pages per delegate per checkpointing file were evicted. Overall time taken by these two cases is almost the same but there is shift of timing from `MPI_file_close()` to `MPI_file_write()`. Now `MPI_file_write()` takes more time to complete because of eviction while `MPI_file_close()` time is much less than first case. That essentially means that there are less cache pages to flushed at close time because significant portion of evicted data consisted of fully dirty pages.

Time taken by other components of application include file open operation, cost of cache management, and memory copy etc.

From this evaluation we know that eviction does not hurt the performance in I/O delegation if fully dirty pages are chosen for eviction. It just shifts time taken in this flushing from `MPI_file_close()` to `MPI_file_write()`.

D. *Exploring Extent Based Locking Algorithm*

Lustre's internal extent based lock implementation is adaptable to the I/O load and access patterns. Lock heuristic changes with changing number of clients contending for the locks. It is possible that locks may be granted according to different heuristics, depending on the arrival time of the requests to a shared file.

Each I/O server is the lock manager of the stripes stored on that server and it grants the locks growing downwards covering all the stripes to the largest uncontended extent. If the number of processes contending for locks is more than an internally specified threshold, locks will grow only upward[40]. If some of the locks are already held by a set of clients before upward lock extension is triggered, then the clients do not have to give up the locks they previously held. New requests may be granted locks grown upward until the uncontended extent. As I/O delegate flushes stripe by stripe in ascending order of their offsets, alternated by other stripes from other clients, mix of downward and upward grown locks may prevent unnecessary lock relinquishing. This scenario may prevent degradation of I/O performance.

In figure 7(c), it is difficult to identify the effect of continuous changing of lock acquisition heuristics during the execu-

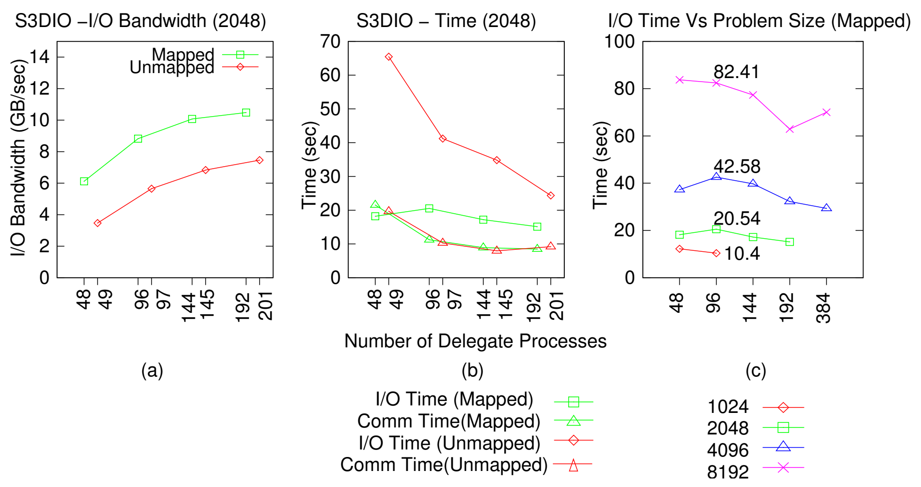

Fig. 11. S3D IO Kernel Analysis on Franklin: (a) I/O Performance comparison of perfectly mapped and unmapped I/O access patterns generated by I/O delegate system for a fixed problem size (b) Breakdown analysis of total time spent in communication and system I/O. Unmapped delegates-to-server case provides much slower I/O as compared to mapped case, while communication time is unchanged (c) I/O time for perfectly mapped I/O access patterns with increasing problem size.

tion of an application. We conducted an additional set of experiments with relatively larger number of delegates accessing an I/O server. We have conducted some additional experiments to compare performance of mapped and unmapped delegate-to-server assignment strategies shown in figures 4 and 5.

Figure 11(a) shows I/O bandwidths achieved by both mapped and unmapped delegates-to-server cases. By keeping everything else constant, we vary the number of I/O delegates to observe any change in I/O performance. If the number of delegates is a co-prime to the number of I/O servers then all the delegates access all the servers, and such I/O pattern limits the advantages of extent based locking protocol. Given the number of I/O servers 48 on Franklin, we chose the number of delegates 49, 97, 145, and 201 to violate delegates-to-server mapping. It is evident from figure 11 that I/O bandwidth decreases significantly for unmapped delegates-to-server case. On the other hand, mapped case provides much higher I/O bandwidth than unmapped case.

Figure 11(b) provides a break down analysis of total time spent in performing I/O operations. Total time spent in completing the I/O operations can be divided in to two main components 1) I/O time, and 2) Communication time. We measured the time spent in the `write()` calls and refer them as I/O time. The rest of the time is referred as communication time, as the operations are mostly data transfer between application and delegate processes.

Figure 11(b) shows the communication and I/O time for both mappings from figure 11(a). This chart shows that communication time does not deviate much by changing the number of delegate processes from a ‘multiple of servers’ to the closest ‘co-prime’ but I/O time increases drastically. Also Figure 11(b) confirms that dramatic decrease in I/O bandwidth in unmapped case is (figure 11 (a)) triggered by the slow I/O only. In case of mapped delegate-to-server case, the number of delegate processes accessing one I/O server is limited to 1, 2, 3, and 4 only. By the adaptive nature of lock granting heuristic, we expect some benefits from extent based locking protocol even though multiple delegates accesses one server. Changing the number of delegates from 48 to 49 violates the perfect mapping between delegates-and-I/O servers. As each I/O server is contended by all the delegates, extensive lock confliction at I/O servers may be introduced. Lustre lock acquisition heuristic may adapt to this development by limiting or suspending extent based locking. For mapped case, no I/O performance degradation is seen in figure 7(c) and 11(b) with the increase in number of delegates. We attribute this observation to dynamic adaptation of extent based locking heuristics during he life time of an application. We believe that in changing the direction of lock growth may only benefit the clients which are holding the downward grown locks. We conclude that by keeping the number minimal we may avoid serious performance degradation.

Another lock acquisition heuristic comes into the effect when even larger number (32 and above for Franklin) of clients access an I/O server. For more than 32 clients accessing an I/O server, extension of lock is limited to 32 MB range only. When a large number of clients are accessing one server, reducing the range of lock extension may help reducing the overheard of lock acquiring-relinquishing-reacquiring phase. This will essentially allow all the process to compete in the range of 32 MB only[40].

Unmapped case performs slower than mapped case but as the number of delegates are increased, unmapped case improves gradually. In fact, the I/O time of unmapped delegate-to-server case decreases with the increase in number of delegates accessing the shared file. This may also be attributed to the adapted lock heuristic. When lock is highly contended, Lustre switches from extent-based mode to as-requested mode and hence avoids further lock conflicts.

For perfect mapping between delegate-to-servers, chart 11 (c) demonstrates the effect of increased problem size on I/O component to the total time. Each curve in this chart represents the I/O time for a specific problem size with varying number of I/O delegate processes. Problem size is doubled by doubling the number of application processes from 1024 to 2048 and so on. We observe that as the problem size is doubled, I/O time is also doubled and does not deteriorate further. This shows that if a perfectly mapped I/O access pattern is chosen then the I/O cost may longer be a bottleneck with growing problem size.

These results advocate that in order to utilize extent based protocol to the full of its potential we need to minimize the number of clients per I/O server.

REFERENCES

[1] General Parallel File System. http://www-03.ibm.com/systems/clusters/software/gpfs/index.html.

[2] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In Darrell D. E. Long, editor, *FAST*, pages 231–244. USENIX, 2002.

[3] Peter J. Braam et al. The Lustre Storage Architecture. www.lustre.org.

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

[5] Feiyi Wang, Sarp Oral, Galen Shipman, Oleg Drokin, Tom Wang, and Isaac Huang. Understanding lustre filesystem internals. White paper, Oak Ridge National Laboratory. http://wiki.lustre.org/images/d/da/Understanding_Lustre_Filesystem_Internals.pdf, April 2009. Available online (76 pages).

[6] 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*, pages 1135–1142, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel i/o via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, 1993.

[8] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *In Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1998.

[9] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard, Version 1.1*, June 1995. http://www.mpi-forum.org/docs/docs.html.

[10] David Kotz. Disk-directed I/O for MIMD Multiprocessors. In *OSDI*, pages 61–74, 1994.

[11] Wei keng Liao, Kenin Coloma, Alok Choudhary, Lee Ward, Eric Russell, and Neil Pundit. Scalable design and implementations for mpi parallel overlapping i/o. *IEEE Transactions on Parallel and Distributed Systems*, 17(11):1264–1276, 2006.

[12] Kenin Coloma, Avery Ching, Alok N. Choudhary, Wei keng Liao, Robert B. Ross, Rajeev Thakur, and Lee Ward. A new flexible MPI collective I/O implementation. In *CLUSTER*. IEEE, 2006.

[13] Nawab Ali, Philip H. Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert B. Ross, Lee Ward, and P. Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *CLUSTER*, pages 1–10. IEEE, 2009.

[14] Javier García Blas, Florin Isaila, Jesús Carretero, Robert Latham, and Robert Ross. Multiple-level MPI file write-back and prefetching for Blue Gene systems. In *Proc. of the 16th European PVM/MPI User's Group Meeting (Euro PVM/MPI 2009)*, September 2009.

[15] Seetharami Seelam, I-Hsin Chung, John Bauer, Hao Yu, and Hui-Fang Wen. Application level i/o caching on blue gene/p systems. In *IPDPS*, pages 1–8. IEEE, 2009.

[16] Yong Chen, Xian-He Sun, Rajeev Thakur, Huaiming Song, and Hui Jing. Improving parallel i/o performance with data layout awareness. In *CLUSTER*, 2009.

[17] 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.

[18] Wei-keng Liao and Alok Choudhary. Dynamically adapting file domain partitioning methods for collective i/o based on underlying parallel file system locking protocols. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[19] Dick Oswald David Knaak. Optimizing MPI-IO for Applications on Cray XT Systems. White paper, Cray Inc, May 2009. Available online (20 pages).

[20] Mark Pagel, Kim McMahon, and David Knaak. Scaling the MPT software on the cray XT5 system and other new features. In *Cray XT Cray Users' Group Meeting, May 4-7, 2009, Atlanta, GA.*, May 2009.

[21] Franklin (Cray xt4). http://www.nersc.gov/nusers/resources/franklin/.

[22] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[23] George Almasi, Charles Archer, Jose G. Castanos, C. Chris Erway, Philip Heidelberger, Xavier Martorell, Jose E. Moreira, Kurt Pinnow, Joe Ratterman, Nils Smeds, and Burkhard. Implementing MPI on the BlueGene/L Supercomputer.

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

[25] José E. Moreira, Michael Brutman, José G. Castaños, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger L. Haskin, Todd Inglett, Derek Lieber, Patrick McCarthy, Michael Mundy, Jeff Parker, and Brian P. Wallenfelt. Blue gene system software - designing a highly-scalable operating system: the blue gene/l story. In *SC*, page 118. ACM Press, 2006.

[26] Kamil Iskra, John W. Romein, Kazutomo Yoshii, and Pete Beckman. Zoid: I/o-forwarding infrastructure for petascale architectures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 153–162, New York, NY, USA, 2008. ACM.

[27] The zeptoos project. http://www.zeptoos.org/.

[28] Hao Yu, R. K. Sahoo, C. Howson, George. Almasi, J. G. Castanos, M. Gupta, Jose. E. Moreira, J. J. Parker, T. E. Engelsiepen, Robert Ross, Rajeev Thakur, Robert Latham, and W. D. Gropp. High performance file I/O for the BlueGene/L supercomputer. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, February 2006.

[29] Phillip M. Dickens and Jeremy Logan. Towards a high performance implementation of MPI-IO on the Lustre file system. In *Proceedings of GADA'08: Grid computing, high-performAnce and Distributed Applications. Monterrey, Mexico*, November 2008.

[30] L. M. Sánchez García, Florin Isaila, Félix García Carballeira, Jesús Carretero Pérez, Rolf Rabenseifner, and Panagiotis A. Adamidis. A new i/o architecture for improving the performance in large scale clusters. In Marina L. Gavrilova, Osvaldo Gervasi, Vipin Kumar, Chih Jeng Kenneth Tan, David Taniar, Antonio Laganà, Youngsong Mun, and Hyunseung Choo, editors, *ICCSA (5)*, volume 3984 of *Lecture Notes in Computer Science*, pages 108–117. Springer, 2006.

[31] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective i/o in panda. In *In Proceedings of Supercomputing &#039;95*, 1995.

[32] Bill Nitzberg and Virginia Lo. Collective buffering: Improving parallel I/O performance. In *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, page 148, Washington, DC, USA, 1997. IEEE Computer Society.

[33] Robert Bennett, Kelvin Bryant, Joel Saltz, Alan Sussman, and Raja Das. Framework for optimizing parallel i/o. Technical report, Univ. of Maryland Institute for Advanced Computer Studies Report No. UMIACS-TR-95-20, College Park, MD, USA, 1995.

[34] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A Framework for Optimizing Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20, Mississippi State, MS, 1994. IEEE Computer Society Press.

[35] Wei keng Liao, Avery Ching, Kenin Coloma, Alok N. Choudhary, and Lee Ward. An Implementation and Evaluation of Client-Side File Caching for MPI-IO. In *IPDPS*, pages 1–10. IEEE, 2007.

[36] Wei keng Liao, Avery Ching, Kenin Coloma, Arifa Nisar, Alok Choudhary, Jackie Chen, Ramanan Sankaran, and Scott 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.

[37] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 23–32, May 1999.

[38] Abe (teragrid intel-64 cluster)
. http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64Cluster/.

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

[40] Lustre mailing list. http://www.mail-archive.com/lustre-discuss@lists.lustre.org/msg05640.html.