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**

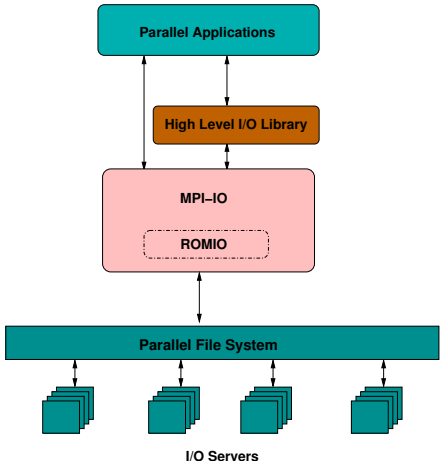

Fig. 1. A Common Parallel I/O Architecture Stack: This figure explains the way different I/O layers are commonly stacked one over another in large scale parallel environments. High end application layer leverages its parallel I/O related tasks directly or through a high level I/O library (PnetCDF, HDF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

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.

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.

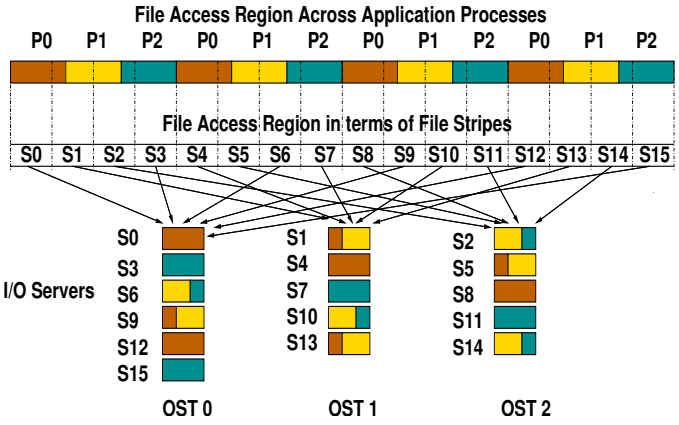

Fig. 2. File access region is partitioned among the application processes P_0, P_1, and P_2. Different colors represent data accessed by different application processes. From Lustre file system's perspective, the entire file is partitioned into 16 stripes S_0, S_1, $\cdots$, S_{15} which are distributed 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.

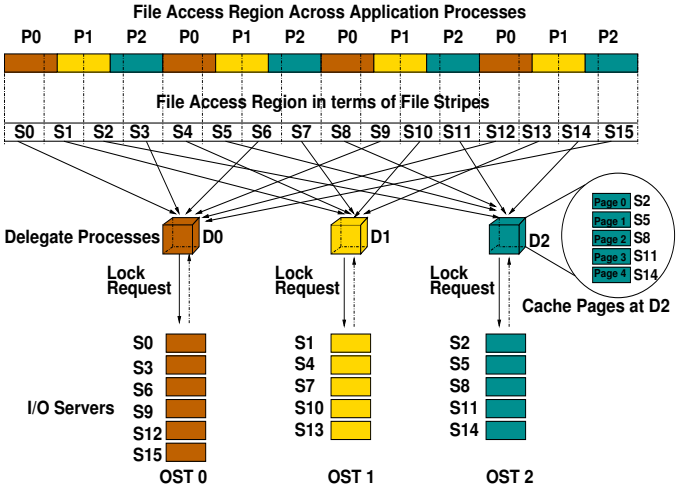

Fig. 3. Lock conflicts can be eliminated by identical partitioning of data across same number of delegate processes and I/O servers. Each of the delegate processes stores data in the form of cache pages directly mapped to the file stripes stored on a unique I/O server. Perfect cache page to file stripe 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

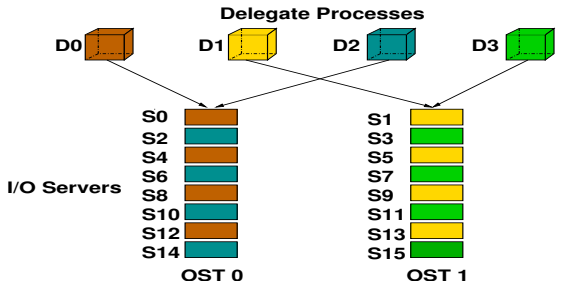

Fig. 4. Lock conflicts are completely eliminated if number of delegate processes are equal or a factor of the number of I/O servers. If number of delegates are more than I/O servers, lock conflicts can occur. To minimize lock conflicts, number of delegates should be kept a multiple of I/O servers. In 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.

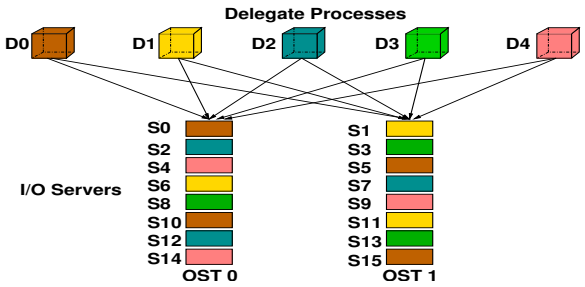

Fig. 5. If delegate processes are not a factor or multiple of I/O servers then all the delegate processes might be accessing all the I/O servers causing serious lock acquisition competition between the processes. In this example, each I/O server is contended by all the delegates which may deteriorate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

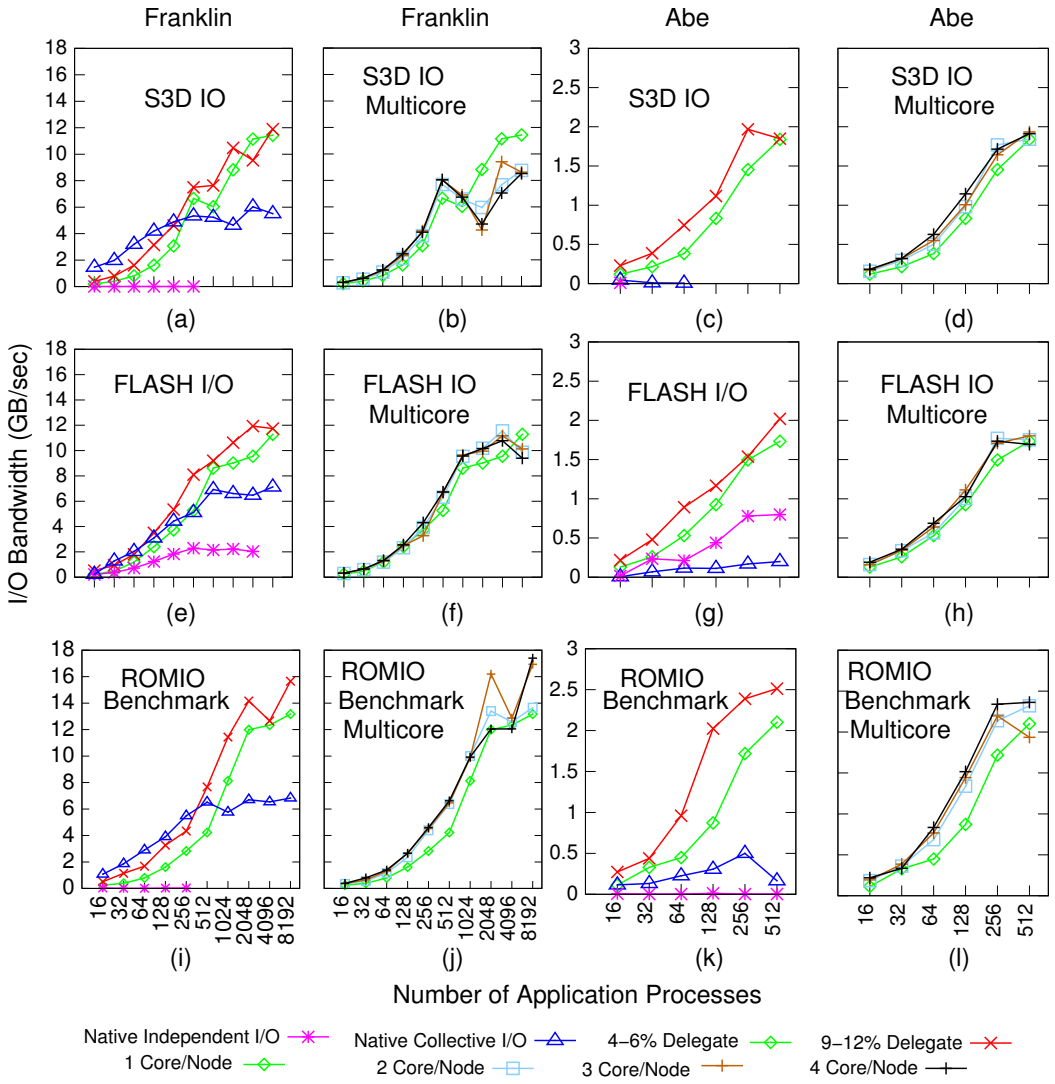

Fig. 6. I/O Performance Evaluation of S3D I/O Kernel, FLASH I/O Kernel and ROMIO Benchmark with Franklin and Abe machines. (a), (e), and (i) show the comparison of write bandwidths of three I/O methods: Independent I/O with I/O delegation, native independent MPI-IO, and native collective MPI-I/O on Franklin. 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

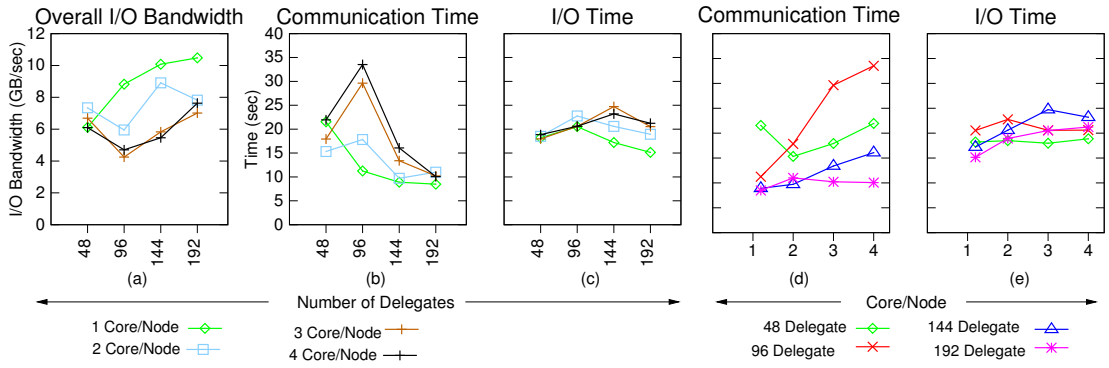

Fig. 7. S3D I/O Kernel, 2048 application processes, Franklin: Breakdown analysis of two major time consuming operations: (i) Data Communication between application and delegate processes. (ii) File system I/O. (a),(b) and (c). Overall I/O bandwidths, time spent in data communication amongst the application 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.