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

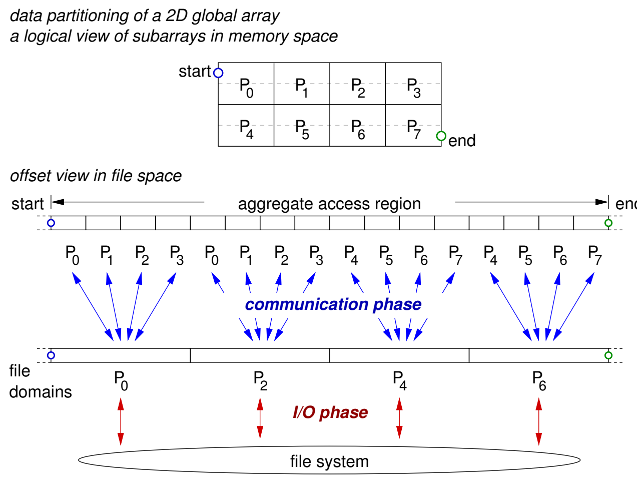
data partitioning of a 2D global array
a logical view of subarrays in memory space
start
P_0
P_1
P_2
P_3
P_4
P_5
P_6
P_7
end
offset view in file space
start
aggregate access region
end
P_0 P_1 P_2 P_3 P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_4 P_5 P_6 P_7
communication phase
file domains
P_0
P_2
P_4
P_6
I/O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

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.

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

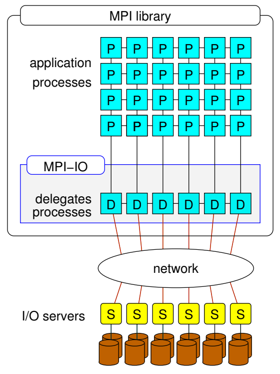

Fig. 9. I/O Delegate Architecture: It is a portable I/O middleware integrated inside the MPI-IO layer. A small percentage of application processes is allocated in addition to the required application processes (P). These additional resources termed as delegate processes (D) perform all the I/O operations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

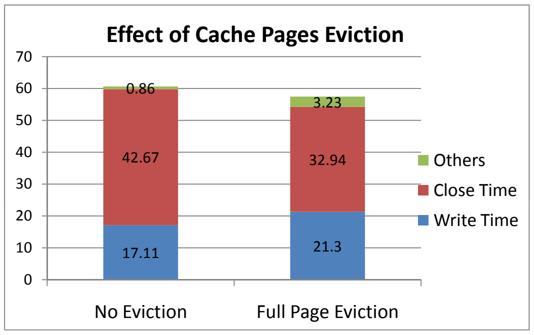

Fig. 10. Time distribution of I/O operations of S3D I/O kernel on Franklin under varying cache pool sizes. Number of Checkpointing Files = 10, Number of Application Processes = 4096, Number of Delegates = 192, Data Generated = 610.35 GB, Data/Delegate/File 325.12 MB (No Eviction). Cache pool size is 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-

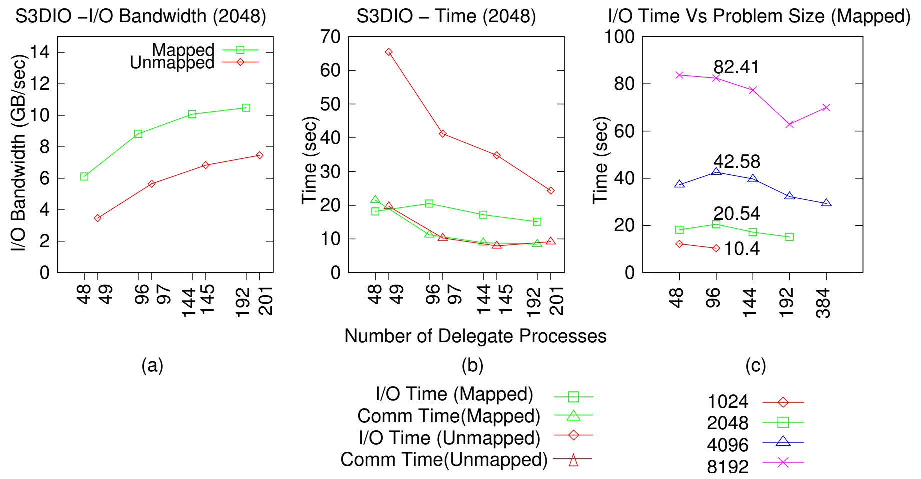

Fig. 11. S3D IO Kernel Analysis on Franklin: (a) I/O Performance comparison of perfectly mapped and unmapped I/O access patterns generated by I/O delegate system for a fixed problem size (b) Breakdown analysis of total time spent in communication and system I/O. Unmapped delegates-to-server case provides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