Memory-Conscious Collective I/O for Extreme Scale HPC Systems

Yin Lu
Computer Science Department
Texas Tech University
Lubbock, TX 79409, USA
yin.lu@ttu.edu

Yong Chen
Computer Science Department
Texas Tech University
Lubbock, TX 79409, USA
yong.chen@ttu.edu

Yu Zhuang
Computer Science Department
Texas Tech University
Lubbock, TX 79409, USA
yu.zhuang@ttu.edu

Rajeev Thakur
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
thakur@mcs.anl.gov

ABSTRACT
Upcoming extreme scale platforms are expected to have millions of nodes with hundreds to thousands of small cores for each node. The continuing decrease in memory capacity per core and the increasing disparity between core count and off-chip memory bandwidth can lead to significant challenges for I/O operations in extreme scale systems. Collective I/O is a critical I/O optimization technique, and the extreme scale challenges require rethinking this strategy for the effective exploitation of the correlation among I/O accesses. In this study, considering the constraint of the memory capacity and bandwidth, we introduce a *Memory-Conscious Collective I/O*. The new collective I/O strategy restricts aggregation data traffic within disjointed subgroups, coordinates I/O accesses in intra-node and inter-node layer, and determines I/O aggregators at run time considering memory consumption and variance among processes. The preliminary results have demonstrated that this strategy holds promise in mitigating the memory pressure, alleviating the contention for memory bandwidth, and improving the I/O performance for projected extreme scale HPC systems.

Categories and Subject Descriptors
B.4.3 [**Hardware**]: Interconnections(Subsystems) – *Parallel I/O*; C.1.4 [**Computer Systems Organization**]: Parallel Architectures

General Terms
Algorithms, Design.

Keywords
Extreme scale system, many-core architecture, parallel I/O, collective I/O, high performance computing

1. INTRODUCTION
High performance computing (HPC) applications, simulations, and visualizations extend across a wide range of science and engineering disciplines such as astrophysics, climate sciences, material sciences, biology, and high-energy physics. Many applications become increasingly data intensive. These applications contain a large number of I/O accesses, where large amounts of data are stored to and retrieved from storage systems [1][2][3][4]. For example, several representative INCITE applications run at Argonne Leadership Computing Facility (ALCF) of Argonne National Laboratory (ANL) generated datasets in the terabyte range and store them on-line [5]. Application teams are projected to process hundreds of terabytes or even petabytes of data in a single simulation run by the end of this decade. Meanwhile, the next generation extreme scale HPC system is near the horizon. The extreme scale system is projected to have millions of nodes, with thousands of cores in each node [6][7]. The rapid advance of computing capability and the phenomenal increase in datasets on extreme scale systems bring critical challenges than ever to the I/O system. The inadequate I/O system capability could substantially lower the performance of extreme scale systems.

The current parallel I/O system often performs inadequately in dealing with a large number of small and noncontiguous requests, which is a common access pattern for scientific applications [8]. Collective I/O is a technique developed to address this problem by merging small and noncontiguous I/O requests into large ones for better performance and has been widely used [9]. Whereas researchers have contributed through years with a number of collective I/O optimizations in the petascale system, comparatively little efforts have been devoted to investigating challenges of data intensive extreme scale computing and designing collective I/O scalable to enter an exaflop era yet. Table 1 compares a potential extreme scale HPC system design with current HPC designs [10]. From the table, it is important to note that *neither available memory capacity nor memory bandwidth will scale by the same factor as the total concurrency (the scale of number of cores)*. The

Table 1. POTENTIAL EXASCALE COMPUTER DESIGN AND ITS RELATIONSHIP TO CURRENT HPC DESIGNS [10]

	2010	2018	Factor Change
System Peak	2 Pf/s	1 Ef/s	500
Power	6 MW	20 MW	3
System Memory	0.3 PB	10 PB	33
Node Performance	0.125 Tf/s	10 Tf/s	80
Node Memory BW	25 GB/s	400 GB/s	16
Node Concurrency	12 CPUs	1000 CPUs	83
Interconnect BW	1.5 GB/s	50 GB/s	33
System Size (nodes)	20 K nodes	1 M nodes	50
Total concurrency	225 K	1 B	4444
Storage	15 PB	300 PB	20
I/O Bandwidth	0.2 TB/s	20 TB/s	100

factor of *memory per core* expected to scale can be expressed with a simple formula - the quotient of the factor change of system memory and system size, divided by the factor change of node concurrency, i.e. $\frac{33}{50*83}$. This projection indicates that the average memory per core even drops to megabytes in extreme scale systems. In addition, due to the memory capacity shared by projected *O(1K)* cores, the available memory per node can vary significantly among nodes. Similarly, the enlarging gap between node concurrency and node memory bandwidth leads to a continuing per-core off-chip bandwidth reduction. Such limited memory per core, significant variance of available memory among nodes, and off-chip bandwidth contention put even more pressure on storage and I/O system. Even though the I/O bandwidth is projected to increase with 100x folds, this improvement does not help the memory pressure issue. In fact, the limited available memory per node and an improper I/O solution can also underutilize the I/O bandwidth. There is an emerging research need for advanced strategies to coordinate small and non-contiguous I/O requests *with memory consciousness* to meet data intensive applications' demand on extreme scale systems.

In this paper we present a new collective I/O strategy namely *memory-conscious collective I/O* to address the above discussed issue. While we focus on the data movement performance and scalability of collective I/O in extreme scale systems, special consideration is given to the memory and off-chip bandwidth. Collective I/O uses I/O aggregators to gather I/O requests and perform reads/writes on behalf of the entire group. The global data shuffling traffic aggravates the memory pressure on aggregators and leads to off-chip memory bandwidth contention. Given I/O aggregator as one of the decisive factors in optimizing collective I/O performance, the newly proposed collective I/O strategy divides the I/O workloads into separated subgroups, restricts the I/O requests aggregation traffic within each subgroup and determines I/O aggregators dynamically by taking data distribution and memory consumption into consideration. Through these mechanisms, memory-conscious collective I/O improves the performance of original two-phase collective I/O, reduces aggregator memory consumption and variance, and conserves off-chip memory bandwidth. In summary, we make the following contributions in this paper:

- First, we identify the performance bottleneck and the scalability constraint imposed by memory and off-chip bandwidth for collective I/O at projected extreme scale systems;
- Second, we propose a new collective I/O strategy that determines the I/O aggregator distribution dynamically on the fly with memory-aware data partition and aggregation mechanisms. The proposed strategy is significant given the importance of improving noncontiguous I/O accesses, reducing the memory pressure and alleviating off-chip bandwidth contention of any collective I/O optimization strategy;
- Third, we demonstrate our approach using synthetic and application benchmarks to yield significant improvement.

The rest of this paper is organized as follows. Section 2 briefly reviews essential concepts of parallel I/O and collective I/O as the background of this study. The design and implementation of memory-conscious collective I/O strategy are presented in Section 3, and the experimental results with analysis are given in Section 4. Section 5 discusses related work, latest advancements in this field, and compares them with this study. We conclude this study in Section 6.

2. MPI-IO, COLLECTIVE I/O AND IMPLEMENTATION

In this section we first briefly review the MPI-IO and its popular implementation ROMIO [9][14]. Then we review the most critical performance optimization strategy, collective I/O, and the widely-used two-phase protocol implementing collective I/O.

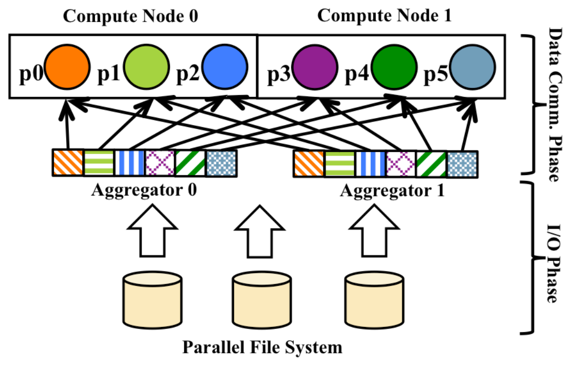


Figure 2: Collective I/O and Two-phase Implementation

MPI-IO is a part of the MPI-2/MPI-3 specification [11][12][13]. It defines an I/O access interface that supports many parallel I/O operations and optimizations. The purpose of MPI-IO is to achieve much higher performance than the Unix API can deliver. In general, the implementation of MPI-IO is a middleware connecting parallel applications and underlying various parallel file systems, providing the code-level portability across many different machine architectures and operating systems. ROMIO is a popular MPI-IO implementation developed at Argonne National Laboratory [9][14]. ROMIO implements an internal layer named ADIO [15](Abstract Device Interface for MPI-IO) to achieve the portability and higher I/O performance. It performs various optimizations, including collective I/O and data sieving, for common access patterns of parallel applications [9].

Collective I/O is one of the most important I/O access optimizations for parallel applications. It differs from independent I/O, in which each process of a parallel application issues I/O requests independently of all other processes. Although independent I/O is a straightforward form of I/O and is widely used in many applications, this form of I/O is not recommended for parallel applications because it does not capture the complete data access information of a parallel application. This shortcoming offers MPI-IO middleware the opportunity to optimize I/O performance given the knowledge of parallel processes.

Collective I/O allows the middleware and the parallel file system to have a comprehensive view of data movements from all processes (involved in the collective I/O) of a parallel application. The motivation to utilize collective operations is several-fold. First, collective I/O can filter overlapping I/O requests from multiple processes and reduce the amount of data accesses to the parallel file system. Second, the requests of multiple processes are often interleaved and may constitute a large contiguous portion of a file together. The performance of handling a large and contiguous request is generally better than handling many noncontiguous and small requests to a storage system. Third, the number of I/O calls is reduced by combining small and noncontiguous requests into large and contiguous ones, thus the overhead involved is reduced too. Note that the collective I/O is a general idea that takes advantage of collective operations among accesses from multiple processes of a parallel application and optimizes its I/O accesses. It can be applied at the disk level (disk-directed I/O [16]), at the server level (server-directed I/O [17]), or at the client level [9]. In this study, we focus on parallel I/O middleware level. If the entire I/O access pattern of a group processes is known to the MPI-IO middleware, the MPI-IO implementation can improve the I/O performance remarkably by merging the requests of different processes and servicing the merged requests by performing collective I/O.

Two-phase protocol is the most popular method of implementing collective I/O. This strategy performs two steps as illustrated in the Figure 2 for the collective read case: I/O access phase and data communication phase. Six processes shown in the upper part of this figure read the file from the parallel file system depicted on the lower part of the figure. In I/O access phase, the data is gathered in contiguous chunks at a part of the compute nodes that act as *aggregators*. In the implementation of two-phase collective I/O, each process first analyzes its own I/O request respectively and let the aggregators know the entire aggregated I/O requests from all processes. In the I/O phase, aggregated I/O requests are divided into *file domains* and each file domain is assigned to one aggregator. After the file domains are determined, each aggregator will access data only from the file domain assigned to it. In this example, we assume there are two aggregators carrying out I/O requests for their own file domains. In the data communication phase, each aggregator sends data to the requesting processes, and each process receive the data from corresponding aggregators that carry the data for it.

3. MEMORY-CONSCIOUS COLLECTIVE I/O

This section describes the design and implementation of the proposed memory-conscious collective I/O strategy. We first introduce the software architecture and then describe the core components in detail.

The main purpose of the memory-conscious collective I/O is to enhance the two-phase collective I/O with new mechanisms for alleviating the memory pressure and mitigating the external bandwidth bottleneck in extreme scale systems. Figure 3 illustrates the high-level view of the proposed memory-

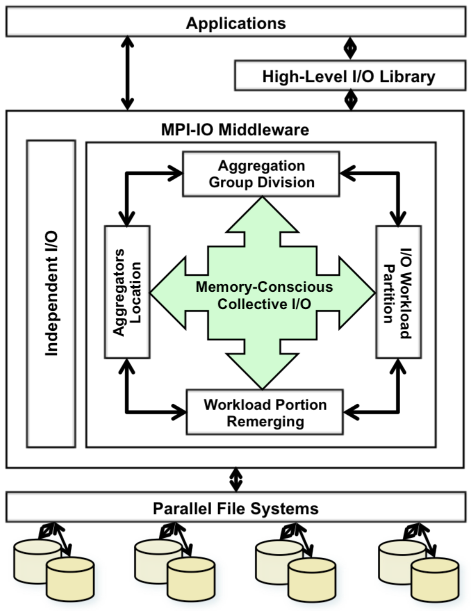


Figure 3: Memory-Conscious Collective I/O Software Architecture

conscious collective I/O software architecture. Four new/revised components are introduced. The *Aggregation Group Division* component divides the I/O requests into separated groups. In each group, the *I/O Workload Partition* component further calculates the aggregate access file region and partitions it into contiguous file domains. The *Workload Portion Remerging* component is designed to rearrange the file domains considering the memory usage of physical nodes. The *Aggregators Location* component determines the placement of aggregators for each file domain.

The ROMIO implementation picks exactly one process per node as I/O aggregator by default. Using a default number of I/O aggregators will inevitably lead to suboptimal performance in the many-core architecture for projected extreme scale systems. In the proposed memory-conscious collective I/O prototype, the corresponding parameters are measured for optimizing the performance of collective I/O. First we determine the optimal number of aggregators N_{ah} and message size Msg_{ind} per aggregator that can fully utilize the I/O bandwidth in one physical compute node, which acts as a *host* to the aggregators. Next we identify the minimum memory consumption Mem_{min} for one physical node. Each node uses N_{ah} I/O aggregators with Msg_{ind} message size to achieve the best performance. Finally, we consider the aggregation I/O traffic contention on system level by increasing the number of aggregators across the system network. The throughput from aggregators to the parallel file system is measured and performance variation is considered to find the optimal group message size Msg_{group} for an aggregation group.

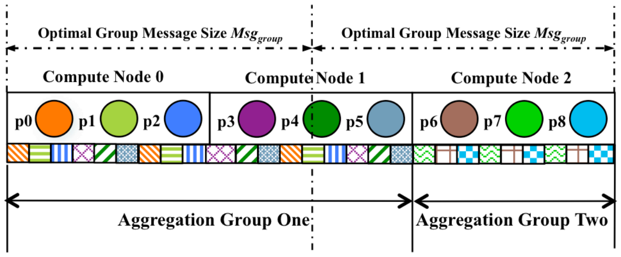


Figure 4: Aggregation group division example

3.1 Aggregation Group Division

The global data shuffling traffic in two-phase collective I/O increases the memory pressure on aggregators and leads to off-chip memory bandwidth contention. The memory-conscious collective I/O first divides the I/O workloads into groups. These groups in turn perform their own aggregation in a disjointed manner, thus restricting the data shuffling traffic within each group. The goal of the aggregation group division is to maximize the data movement speed during the data shuffle phase and reduce the amount of memory needed and variance for each aggregator with a balanced memory consumption design, which is critical for scaling the collective I/O to an extreme scale or beyond.

To divide the I/O workloads among aggregation groups, analytics are applied to the entire file domain to detect the data access pattern of an application. A large number of applications use explicit offset operations in the I/O calls, or the data segments are serially distributed among processes. In this case, an offset calculation guided by the optimal group message size Msg_{group} will divide the I/O workloads into non-overlapping chunks. For example as shown in Figure 4, a linearization of data distribution across 9 processes in three compute nodes, the size of aggregation group one is extended to the ending offset of the data accessed by the last process in compute node one. In this way, we avoid that processes from the same physical node become I/O aggregators for different groups. Scientific applications with complex structured datatype exhibit more complicated access patterns, where the beginning and ending offsets are interwoven with each other. In these situations, the aggregation group division can be determined by analyzing the MPI file view across processes.

3.2 I/O Workload Partition & Portions Remerging

Within each aggregation group, the *I/O Workload Partition* component analyzes the offsets and lengths of all I/O requests. Data are gathered in a contiguous chunk and dynamically partitioned into distinct domains where each aggregator can achieve the optimal performance at the given workload. To obtain such a partitioning, a dynamical workload partition algorithm is applied to the file region by generating a binary partition tree. Each vertex in the tree represents a non-overlapping portion of the whole file region requested by all processes in one aggregation group. The internal vertices in the tree stand for the portions that no longer exist, but were split at some previous time. Each divided file domain is represented as a leaf of the binary partition tree. Essentially the core of the algorithm is a recursive bisection method to divide the file region into two sets until the termination criterion Msg_{ind} is met. As a result of the algorithm presented above, different number of file domains will be generated in each group depending on the amount and distribution of data during one collective I/O operation. Also, the I/O message size in each file domain results in the I/O saturation for one aggregator.

Although the I/O data size Msg_{ind} can lead one aggregator to achieve the best performance with the optimal file domain size, the aggregator may perform less well than expected because of other resource constraints, especially the amount of memory available for the aggregation buffer. When the processes associated with one file domain are short of memory resources, this file domain will be merged with the domain nearby to expand the search area until find the aggregator host that satisfies the memory requirement.

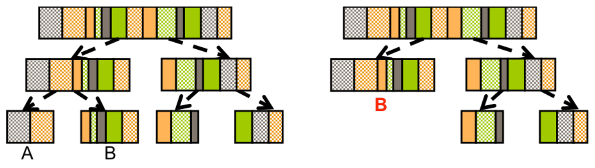


Figure 5a: File Domain Remerge with the Neighbor Case 1

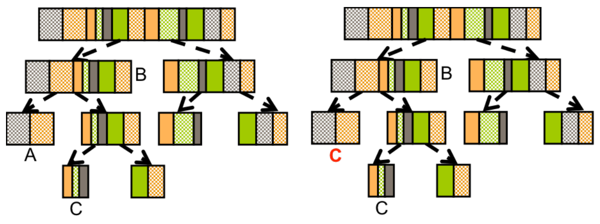


Figure 5b: File Domain Remerge with the Neighbor Case 2

When a file domain is remerged with the neighbor, the correlated vertex leaves the partition tree. The file domain the departed vertex occupied is taken over by the remaining vertices. This concept can be easily defined using the partition tree. Suppose a leaf vertex "A" leaves the tree and the file domain it owns need to be remerged with other file portions. There are two cases when a leaf is leaving: if the sibling of this leaf is also a leaf (call it "B"), then B will take over A directly. We simply merge leaves A and B, making their former parent vertex a leaf and assign vertex B to that leaf. Thus the regions correlated to A and B merge into a single region that is owned by vertex B. Figure 5a shows an example of such a takeover. If A's sibling B is not a leaf because the sibling file portion has been further split, then it is necessary to perform a depth first search (DFS) in the sub-tree rooted at B until a leaf vertex is found. In particular, in order to remerge with the neighbor region nearby and if A is the left sibling of B, the DFS must visit left siblings before right ones. Otherwise if A is the right sibling of B, the traversal will first visit right siblings. This leaf, call it "C", acts as A's sibling and takes over the region owned by A. Figure 5b shows an example of such a takeover. Note that the remerge procedures are limited within each aggregation group.

3.3 Aggregators Location

The number of file domains produced by the *I/O Workload Partition* algorithm can determine the number of aggregators during one collective I/O operation. The *Workload Portions Remerging* component reorganizes the file domains considering the memory consumption for the aggregation. These two components prompt the system to locate the aggregators for compute processes.

The strategy to locate the aggregators within one file domain is to first obtain all processes of which I/O requests are located in this file domain; then it compares the processes related hosts (utilizing the IP address of each process) while each candidate host should have less than N_{ah} aggregators. The host with maximum system memory Mem_{avl} available is identified. If Mem_{avl} is larger than the memory Mem_{min}, the corresponding process will be selected as the aggregator in this file domain. Otherwise, it indicates setting any compute nodes related to this file domain as the aggregator host may underperform because there is not enough aggregation memory to guarantee the best I/O performance. In this case, the file domain will be integrated with the domain nearby. Processes related hosts are repeatedly inspected as in the above process until the one that satisfies the memory requirement is identified.

4. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we present the experimental results of the proposed memory-conscious collective I/O. We also compare it with the existing two-phase collective I/O approach.

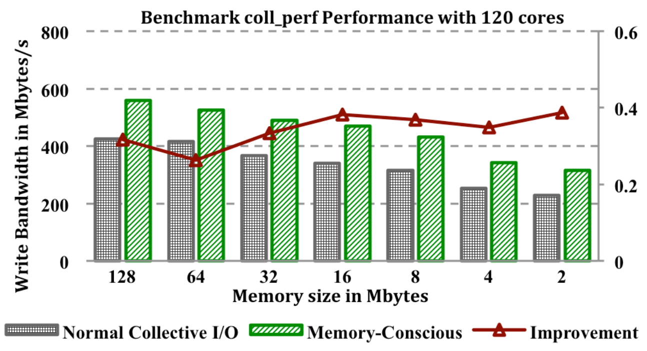
The experiments were conducted on a 640-node Linux-based cluster test bed with DataDirect Network storage systems. Each node contains two Intel Xeon 2.8 GHz 6-core processors with 24 GB main memory. All nodes are connected with double-data-rate Infiniband networking that provides full cross-section bandwidth among the parallel nodes. A 600TB Lustre file system and MPICH2-1.0.5p3 library manage the storage system and runtime environment. Files were striped over all I/O servers with the round robin default striping strategy (with 1 MB unit size in the experiments).

The performance of memory-conscious collective I/O introduced in section III was evaluated and compared with the normal two-phase collective I/O strategy. In this paper, we empirically determined the number of aggregators N_{ah}, message size Msg_{ind} per aggregator and the group message size Msg_{group}. We leave the examination of these optimal values to a future study as it is correlated with the I/O pattern of a particular application. The normal two-phase I/O uses the default number of I/O aggregators for data access, which is exactly one process per node. The assignment of aggregators in normal collective I/O is independent of the distribution of the data over the process.

While we study the performance of the proposed strategy, our focus is on the scalability analysis as we varied the buffer size for collective I/O and the number of processes. In each collective call, the memory buffer used by each aggregator was fixed for normal collective I/O. For the memory-conscious collective I/O, the memory buffer sizes for processes were set up as random variables following a normal distribution. The arithmetic mean of this normal distribution was equal to the aggregator buffer size of the normal collective I/O in each run. The standard deviation was set as 50 in our experiments.

We choose two well-known MPI-IO benchmarks for evaluation: *coll_perf* from ROMIO software package [14] developed at Argonne National Laboratory and IOR [18] from the ASCI Purple benchmark suite developed at Lawrence Livermore National Laboratory.

4.1 Experimental Results of coll_perf Benchmark



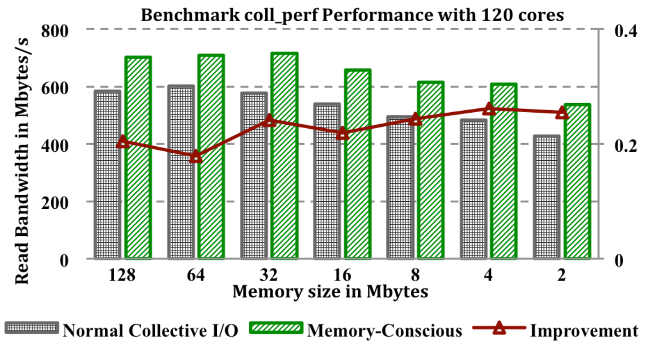
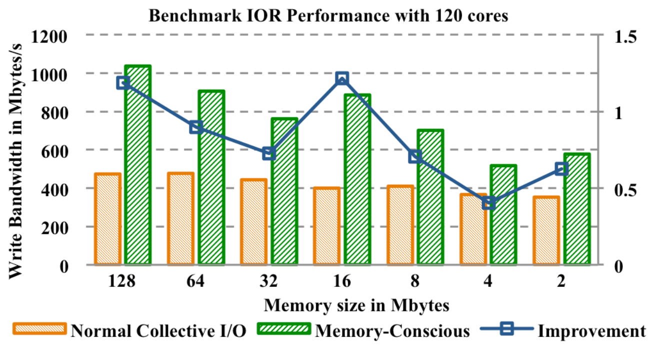


Figure 6: Performance comparison with coll_perf benchmark with various memory sizes at 120 cores

coll_perf is one of test programs from ROMIO software package. This benchmark writes and reads a 3D block-distributed array to a file corresponding to the global array in row-major order using collective I/O. We ran the benchmark with 2048^3 as the array size to measure the I/O bandwidth. 120 MPI processes were used to write and read a 32 GB file that resides on Lustre file system. We modified the original implementation and evicted cached data with memory flushing after write phase.

Figure 6 shows the write and read bandwidth for both normal two-phase collective I/O and the proposed memory-conscious collective I/O. As expected we can observe that both collective I/O strategies showed a drop in performance as the available memory buffer size reduced. However, the memory conscious collective I/O always performed better than two- phase collective I/O especially memory per aggregator at smaller sizes. By utilizing the new strategy, the average performance for write and read tests were 34.2% and 22.9% respectively.

4.2 Experimental Results of IOR Benchmark



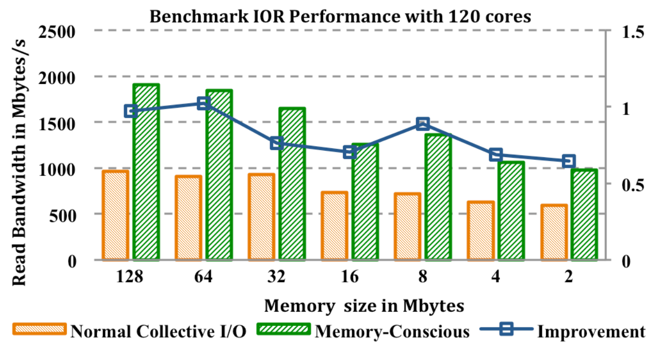
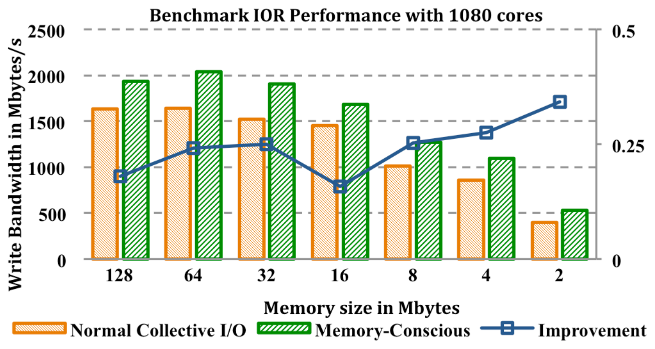


Figure 7: Performance comparison with IOR benchmark with various memory sizes at 120 cores

Interleaved Or Random (IOR) benchmark measures the performance of parallel I/O through different I/O interfaces, including MPI-IO, POSIX as well as higher-level libraries. In this study, we performed interleaved read and write operations to a file as we varied the buffer size for collective I/O, the message size transferred per process and the number of processes. The tests were carried out with 120 and 1080 processes respectively.

Figure 7 compares the write and read bandwidth with both the normal two-phase collective I/O and the new memory-conscious collective I/O at 120 cores as we varied memory buffer used on each aggregator. The tests were conducted with 32 MB I/O data message per MPI process. As shown in Figure 7, the new strategy can affect the IOR benchmark testing performance considerably. The best write performance improvement was achieved with the memory size at 16 MB. The memory-conscious collective I/O was observed with an improvement of 1.2 times compared to that of normal two-phase collective I/O. For the memory size at 8 MB, the performance improvement for read was 89.1%. The



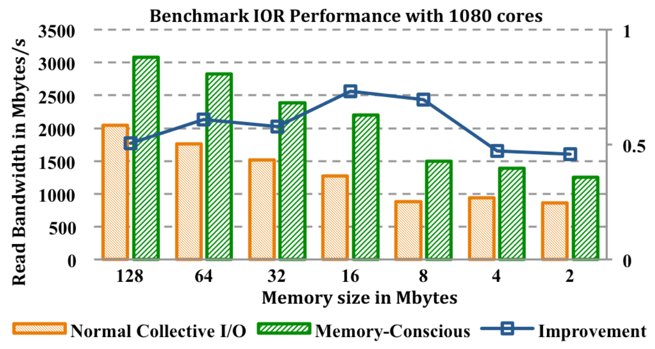


Figure 8: Performance comparison with IOR benchmark with various memory sizes at 1080 cores

performance improvements of write tests are more sensitive for the new strategy, as it varied from 40.3% improvement to 121.7% improvement. The performance improvements of read tests are less sensitive. The performance speedup varied from 64.6% to 97.4%. The average performance improvements for read and write tests were 82.4% and 81.2% respectively.

We have also measured and compared the performance with varying the number of processes. Figure 8 compares the performance of the normal two-phase collective I/O and the memory-conscious collective I/O at 1080 cores while we decreased the aggregation buffer size over different runs. The experimental results showed that the write bandwidth of normal collective I/O dropped from 1631.91MB/s to 396.36MB/s as the aggregation memory size decreased from 128MB to 2MB. Meantime, the read bandwidth dropped from 2047.05 MB/s to 861.62 MB/s. We observed that the proposed memory-conscious collective I/O improved both read and write bandwidth over the existing strategy constantly with 1080 cores. The average improvement of the memory-conscious collective I/O with different memory sizes was 24.3% and 57.8% for interleaved writes and reads respectively. It demonstrates that the memory-conscious collective I/O is very beneficial to collective I/O by dynamically determining a better I/O aggregator placement within an environment that has limited memory resources.

5. RELATED WORK

Many research efforts have been devoted to optimize parallel I/O performance in the past, such as collective I/O [9], data sieving [9], server-direct I/O [17], disk-directed I/O [20], and ADIOS library [21]. Recently, Zhang et. al. proposed to find and match the I/O request pattern with the striping pattern to have an efficient resonant I/O [22]. Iskra et. al. introduced an

I/O forwarding component in the ZeptoOS for petascale architectures such as IBM Blue Gene systems [23]. Ali et. al. proposed a new I/O forwarding software layer sitting between the MPI-IO and parallel file systems to ship the I/O calls to dedicated I/O nodes and improves the scalability of parallel I/O systems [24]. A layout-aware collective I/O strategy was recently introduced in [25] to provide a better integration of parallel I/O middleware and parallel file system and improve the overall performance. These strategies collect and merge small and noncontiguous requests into a large and contiguous one for aggregators to carry out more efficiently, or ship I/O calls to dedicated processes. This study further improves the collective I/O strategy and proposes a memory conscious method that dynamically makes the decision for I/O workload segmentation and aggregators determination. It improves the performance of the existing collective I/O approach and reduces the memory consumption and bandwidth contention.

Parallel file systems, such as Lustre [26], GPFS [27], PanFS [28] and PVFS/PVFS2 [29], enable concurrent I/O accesses from multiple clients to files. Numerous optimizations also exist to improve the file system performance, such as data staging services [30], coordinated access interface [31], performance bridging [32], a log-structured interposition layer and latent asynchrony I/O [33]. While parallel file systems perform well for large and well-formed data streams, they often perform inadequately when dealing with many small and noncontiguous data requests. The collective I/O and memory-conscious collective I/O proposed in this study address these issues well. The memory-conscious approach dynamically determines I/O aggregators at run time considering data distribution and memory consumption among processes and is more beneficial than the existing collective I/O approach. This research can have an impact for extreme scale high performance parallel I/O system.

6. CONCLUSION

Extreme scale high performance computing systems are near the horizon. The projected substantially increased total concurrency in extreme scale systems and the decreased memory capacity per core, increased available memory variance per node, and decreased bandwidth per core as analyzed in this study can be critical challenges for collective I/O to work effectively at an extreme scale. In this study, we identify the performance bottleneck and the scalability constraint imposed by memory and off-chip bandwidth to collective I/O. Motivated by these observations, we propose a new *memory-conscious collective I/O* strategy that determines the I/O aggregator distribution dynamically on the fly with memory-aware data partition and aggregation mechanisms. The proposed strategy was evaluated on MPICH2 and Lustre file systems with simulating the limited memory capacity, increased memory variance, and limited off-chip bandwidth. The evaluation results confirmed the proposed memory-conscious collective I/O strategy outperformed existing strategies given the memory pressure and bandwidth constraints. This study could be significant given the importance of improving noncontiguous I/O accesses, reducing the memory pressure, and alleviating off-chip bandwidth contention of collective I/O on projected extreme scale systems.

7. ACKNOWLEDGMENTS

The authors acknowledge the High Performance Computing Center (HPCC) at Texas Tech University at Lubbock for providing HPC resources that have contributed to the research results reported in this paper.

8. REFERENCES

[1] J.Dongarra, P. H. Beckman, et. al. The International Exascale Software Project roadmap. IJHPCA 25(1): 3-60 (2011)

[2] Isaila, J. G. Blas, J. Carretero, R. Latham, R. B. Ross. Design and Evaluation of Multiple-Level Data Staging for Blue Gene Systems. IEEE Trans. Parallel Distrib. Syst. 22(6): 946-959, 2011.

[3] D. Donofrio, L. Oliker, J. Shalf, M. F. Wehner, C. Rowen, J. Krueger, S. Kamil and M. Mohiyuddin. Energy-Efficient Computing for Extreme-Scale Science. IEEE Computer 42(11): 62-71 (2009)

[4] R. E. Bryant. Data-intensive supercomputing: The case for DISC. In Tech Report CMU-CS-07-128, Carnegie Mellon University School of Computer Science, 2007.

[5] DOE Innovative and Novel Computational Impact on Theory and Experiment program, http://hpc.science.doe.gov/

[6] J. Shalf, S. S. Dosanjh and J. Morrison. Exascale Computing Technology Challenges. VECPAR 2010: 1-25

[7] S. Borkar. Thousand core chips: a technology perspective. In Proceedings of the 44th annual Design Automation Conference (DAC '07). ACM, New York, NY, USA, 746-749. 2007.

[8] P. M. Dickens and R. Thakur. Improving Collective I/O Performance Using Threads. In Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing: IEEE Computer Society, 1999.

[9] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation, 1999.

[10] J. S. Vetter, V. Tipparaju, W. Yu and P. C. Roth. HPC Interconnection Networks: The Key to Exascale Computing. High Performance Computing Workshop 2008: 95- 106

[11] W. D. Gropp, E. Lusk, and R. Thakur. Using MPI-2. MIT Press, 1999.

[12] J. May. Parallel I/O for High Performance Computing. Morgan Kaufmann Publishing, 2001.

[13] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. http://www.mpiforum. org/docs/docs.html, 1996.

[14] ROMIO website. http://www-unix.mcs.anl.gov/romio/.

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

[16] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. ACM Transactions on Computer Systems, 15(1):41-74, 1997.

[17] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-Directed Collective I/O in Panda. In Proc. of Supercomputing Conference, 1995.Ding, W. and Marchionini, G. 1997. *A Study on Video Browsing Strategies*. Technical Report. University of Maryland at College Park.

[18] IOR benchmark. URL https://computing.llnl.gov/

[19] MPI-IO Test (fs_test) benchmark. URL http://institutes.lanl.gov/data/software/#mpi-io

[20] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. ACM Transactions on Computer Systems, 15(1):41-74, 1997

[21] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki and C. Jin. Flexible I/O and Integration for Scientific Codes Through the Adaptable I/O System (ADIOS). In Proc. of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, 2008.

[22] X. Zhang, S. Jiang, and K. Davis. Making Resonance a Common Case: A High-performance Implementation of Collective I/O on Parallel File Systems. In Proc. of the 23rd IEEE International Symposium on Parallel and Distributed Processing, 2009.

[23] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman. ZOID: I/O Forwarding Infrastructure for Petascale Architectures. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 153 - 162, 2008.

[24] N. Ali, P. H. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. B. Ross, L. Ward, P. Sadayappan. Scalable I/O Forwarding Framework for High-performance Computing Systems. Proceedings of the 2009 IEEE International Conference on Cluster Computing, 2009.

[25] Y. Chen, X.-H. Sun, R. Thakur, P. C. Roth and W. Gropp. LACIO: A New Layout-Aware Collective I/O Strategy for Parallel I/O Systems. In the Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), 2011.

[26] Cluster File Systems Inc., Lustre: A scalable, high performance file system, Whitepaper, http://www.lustre.org/docs/whitepaper.pdf

[27] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In Proceedings of the First USENIX Conference on File and Storage Technologies, pp. 231-244, USENIX, January 2002

[28] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B.Mueller, J.Small, J. Zelenka and B. Zhou. Scalable Performance of the Panasas Parallel File System. In Proc. of the 6th USENIX Conference on File and Storage Technologies, 2008.

[29] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS:A parallel file system for linux clusters. In Proceedings of the 4th Annual Linux Showcase and Conference

[30] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan and S.Klasky. Just In Time: Adding Value to the I/O Pipelines Of High Performance Applications with JITStaging. In Proc. of Intl. Symp. on High Performance Distributed Computing (HPDC), 27-36, 2011.

[31] S. Lang, R. Latham, R. B. Ross and D. Kimpe. Interfaces for Coordinated Access in the File System. CLUSTER, pp. 1-9, 2009.

[32] P. Gu, J. Wang and R. Ross. Bridging the Gap between Parallel File Systems and Local File Systems: A Case Study with PVFS. The 37th International Conference on Parallel processing 2008 (ICPP'08), Pages 554-561, September 2008.

[33] P. Widener, M.Wolf, H. Abbasi, S. McManus, M. Payne, M. J. Barrick, J. Pulikottil, P. G. Bridges and K. Schwan. "Exploiting Latent I/O Asynchrony in Petascale Science Applications." IJHPCA 25(2): 161-179, 2011.