Improving Collective I/O Performance by Pipelining Request Aggregation and File Access

Saba Sehrish
Northwestern University
USA
ssehrish@eecs.northwestern.edu

Seung Woo Son
Northwestern University
USA
sson@eecs.northwestern.edu

Wei-keng Liao
Northwestern University
USA
wkliao@eecs.northwestern.edu

Alok Choudhary
Northwestern University
USA
choudhar@eecs.northwestern.edu

Karen Schuchardt
PNNL
USA
Karen.Schuchardt@pnnl.gov

ABSTRACT

In this paper, we propose a multi-buffer pipelining approach to improve collective I/O performance by overlapping the dominant request aggregation phases with the I/O phase in the two-phase I/O implementation. Our pipelining method first divides the collective buffer into a group of small size buffers for an individual collective I/O call and then pipelines the asynchronous communication to exchange the I/O requests with the I/O requests sent to the file system. Our performance evaluation of a representative I/O benchmark and a production application shows 20% improvement in the I/O time, given theoretical upper bound of 50% when both phases completely overlap.

Keywords

Parallel I/O, Collective I/O, Two-phase I/O performance

1. INTRODUCTION

I/O has always been considered as one of the limiting factors in achieving scalability and high performance in large-scale computer systems, such as those from the Top500 list [8]. Thus, many parallel I/O libraries have been developed to work with the underlying file systems to achieve better performance. One of the most common file access patterns used in parallel applications running on these large-scale systems is shared-file I/O, in which multiple processes simultaneously read or write from/to different locations of the same file. While the shared-file I/O reduces the number of files created from a single application run, it can cause file lock contentions at the file system level and thus hamper the achievable performance. To tackle such problem, collective I/O was proposed to reduce the lock contentions by reorganizing the parallel I/O requests among all the requesting processes. Well-known examples of collective I/O implementations are *two-phase I/O* [5] and *disk directed I/O* [9]. The two-phase I/O method uses a subset of processes to aggregate the I/O requests from all processes into fewer larger contiguous requests, so they can be serviced more efficiently by the underlying file systems. This strategy has demonstrated significant performance improvement over the uncoordinated approaches.

The two-phase I/O is adopted by the ROMIO library to implement the collective MPI-IO functions. ROMIO is the most commonly used implementation of MPI-IO functions and has been incorporated by almost all MPI libraries [19]. The two-phase I/O strategy conceptually consists of a *request aggregation phase* (or referred as the communication phase) and a *file access phase* (or simply the I/O phase). In the request aggregation phase, a subset of MPI processes is picked as I/O aggregators that act as I/O proxies for the rest of the processes. The aggregate file access region requested by all processes is divided among the aggregators into non-overlapping sections, called file domains. During writes, the non-aggregator processes send their requests to the aggregators based on the file domain assignment. In the file access phase, each aggregator commits the aggregated requests to the file system. When the aggregated request amount is large, the collective I/O is carried out in multiple rounds of two-phase I/O to conserve the memory usage.

Recent studies on collective I/O performance have shown that the cost of request aggregation phase starts to exceed the file access phase for large runs [12, 10]. TAU profiling result from one of our experiments using 10,240 compute cores on Hopper shows write phase contributes to only 32.3% of the collective write time. The reason is that there are only 156 Lustre file object server targets (OSTs) available on Hopper and hence 156 aggregators were selected by the Cray's MPI-IO library. Such a small number of aggregators causes inter-process communication contention and impacts the performance. Hence, as the number of compute cores on a parallel computer increases and the I/O-to-compute node ratio remains small, it becomes important to study different approaches to minimize inter-process communication contention and balance the cost of two phases to achieve best end-to-end performance.

In this paper, we propose an optimization to balance the cost of request aggregation and file access phase. Our approach is designed specifically for improving the performance of a single collective I/O call by reducing the impact of

EuroMPI '13 September 15 - 18 2013, Madrid, Spain

communication time. Our main contribution in this paper is *design and implementation of a multi-buffer pipelining method that allows overlapping between the request aggregation and file access phases within a single collective I/O call*. The pipelining method divides the collective buffer, an internal temporary buffer, which accommodates the aggregated requests into several sub-buffers. It then pipelines the two-phase I/O on the individual sub-buffers by overlapping the MPI asynchronous send-receive calls and the I/O phase alternatively. This strategy can significantly improve the performance if both the request aggregation and file access phases spend equal amount of time in a collective I/O call.

We evaluate the proposed method on the Cray XE6 parallel computer named Hopper at the National Energy Research Scientific Computing Center (NERSC) using an I/O benchmark and application I/O kernel for up to 40,000 MPI processes. The ideal scenario happens when both phases take equal amount of time and the pipelining method can completely overlap these two phases. Therefore, the theoretical upper bound of performance improvement is 50%. In reality, it is rare that the two phases take an equal amount of time and the upper bound is in fact determined by the phase with smaller percentage to the overall time. For example, if the aggregation and file access phases take 30% and 70% respectively, the upper bound of the overlapping is 30%, i.e. the maximum amount of work that can be overlapped between the two phases. In our experiments, we obtain up to 20% improvement for most of the cases, which we consider a decent performance improvement.

The rest of this paper is organized as follows: Background and related work is discussed in Section 2. Design and implementation of proposed pipeline approach is described in Section 3. Performance evaluation is presented and analyzed in Section 4 followed by conclusion in Section 5.

2. BACKGROUND AND RELATED WORK

The I/O functions defined in MPI are classified into two categories: collective and independent. Collective I/O requires all processes that belong to the communicator group used in `MPI_File_open()` to open a file. Such synchronization provides a collective I/O implementation an abundant opportunity for process collaboration, for instance, by exchanging access information and reorganizing the I/O requests. Many collaboration strategies besides the two-phase I/O and disk directed I/O have been proposed and demonstrated their successes, including server-directed I/O [18], persistent file domain [3, 2], active buffering [13], and collaborative caching [4, 11]. Independent I/O, in contrast, requiring no synchronization makes any collaborative optimization difficult.

Two-phase I/O is a representative collaborative I/O technique that runs at user space. It is motivated by the fact that much slower I/O phase than the aggregation phase is observed in parallel machines at the time it was proposed. It leverages much faster inter-process network to redistribute the requests among processes into large and contiguous chunks, so that the I/O cost to the underlying file system is significantly improved. The request redistribution is referred as the aggregation phase and file accessing the I/O phase. The aggregation phase first identifies the I/O aggregators and calculates the aggregate access region of the entire collective I/O. I/O aggregators are a subset of processes that can make `read()` and `write()` calls to the file system. The aggregate access region is a contiguous file region starting from the minimal access offset to the maximal offset among all the requesting processes. The region is partitioned into non-overlapping sub-regions denoted as **file domains**, and each is assigned to a unique I/O aggregator. For non-aggregator processes, their file domains are null.

File domain partitioning methods were first studied by Nitzberg and Lo who examined three file domain-partitioning methods, namely block, file layout, and cyclic target distributions [14]. The three methods perform competitively, and with carefully selected collective buffer sizes the cyclic method can outperform others in some cases. File domain partitioning methods were also studied in [21], which focuses on the request arrival orders at the disk level. By rearranging the file blocks at the I/O aggregators into sequential accesses in disk blocks, collective I/O performance can be effectively improved. File domain partitioning methods for Lustre and the IBM GPFS file systems were proposed in [12, 10] to minimize the file lock contentions.

Many strategies have been proposed to improve collective I/O performance [6, 20, 1]. None of the existing approaches, however, try to overlap the data aggregation and I/O phases. Partitioned collective I/O improves collective I/O performance by partitioning file area, I/O aggregator distribution and intermediate file views. View-based collective I/O uses MPI file views across multiple calls, so that the request redistribution does not need to calculate and exchange the request information after the first call [1]. However, the requirement is that the file view must keep unchanged. Patrick et al. [15] overlap the MPI communication, computation, and I/O at the application level and none is designed for improving the performance of a single collective I/O call.

3. DESIGN AND IMPLEMENTATION

As the request aggregation phase for larger runs starts to dominate the overall collective I/O cost, we propose a *multi-buffer pipelining* approach to overlap the aggregation with the file access phases. In ROMIO's implementation for the collective I/O, a *collective buffer* is an internal temporary buffer used for the request aggregation. When the file domain size assigned to an I/O aggregator is larger than the size of the collective buffer, the collective I/O operation is carried out in multiple rounds of two-phase I/O. Each round handles a file sub-domain with size equal to or less than the collective buffer size. More than one round of two-phase I/O happens very often especially on parallel computers that have large number of CPU cores on each compute node and only one core is chosen to serve as an aggregator. However, these rounds of two-phase I/O are currently implemented in a blocking manner. In other words, one round of two-phase I/O must complete before the start of the other round of two-phase. In fact, we found that the sequence of two phases in multiple rounds can be overlapped through a pipeline mechanism to reduce the effect of longer request aggregation phase on overall collective I/O cost.

There are potentially two design options to enable the overlapping: 1) using asynchronous I/O, and 2) using asynchronous communication. The former divides the collective buffer into two halves and runs the two-phase I/O on the two sub-buffers alternatively. By replacing the blocking read/write calls by the non-blocking calls, the I/O phase of one round can be overlapped with the aggregation phase of the next round. POSIX standard defines a set of asyn-

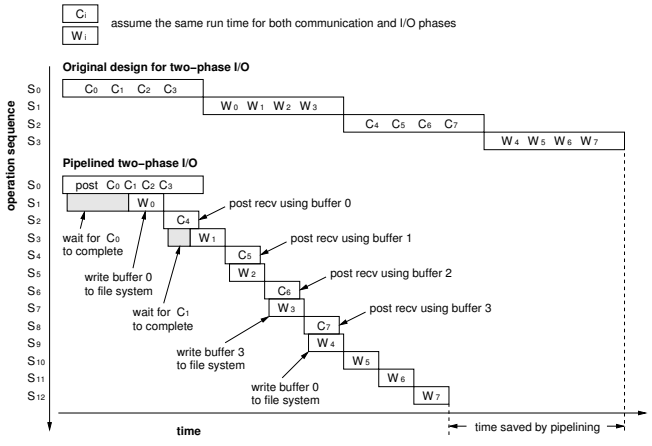

Figure 1: Comparing the conventional non-pipelined method with the pipelined method. In the pipelined method, the collective buffer is divided into multiple sub-buffers used to enable the overlapping of the request aggregation and file access phases.

chronous I/O functions, generally referred as aio, including `aio_write`, `aio_read`, and `aio_suspend`. A wait call for the asynchronous read/write must be used to enforce the exclusive use of the buffers to prevent one phase outrunning the other. Since two consecutive rounds of two-phase I/O uses different buffers, overlapping of the two phases can be achieved. This approach can be extended to use multiple sub-buffers to increase the overlapping the two phases across multiple rounds. However, the drawback is that the overlapping depends on the support of asynchronous read/write from the file systems. Many modern parallel file systems do not support asynchronous I/O, e.g. aio functions on the Lustre file system are actually blocked, which commit the requests before returning the calls. For this reason, we choose the second option that relies on the *MPI asynchronous communication to overlap the two phases*.

In ROMIO, the default size of collective buffer is 16 MB. A customized value for the buffer size can be set through the MPI-IO hint `cb_buffer_size` . In our pipelining design, the collective buffer is divided into smaller sub-buffers of equal size. Each sub-buffer is used to receive requests from the remote processes for data aggregation in a successive but independent round of two-phase I/O. We set the size of sub-buffer buffer equal to the file striping size to minimize the file lock contention. The range of the file domains of all aggregators are made known to all processes, hence each process can calculate the number of rounds of two-phase I/O based on the local requests. Before starting the two-phase I/O, the offset-length pairs of the requests by all processes are made available to the relevant I/O aggregators through an all-to-all personalized MPI communication. Therefore, no global synchronization is required during the multiple rounds of two-phase I/O because all processes know their requests, non-aggregators know which aggregators will serve their requests, and aggregators know which non-aggregators to serve. In each round, requests are received in the sub-buffer independent of each other and committed to file system in a pipelined manner.

A collective write example shown in Figure 1 illustrates our pipelining design. In this example, there are 2 rounds of two-phase I/O when the traditional approach is used. The collective buffer size is equivalent to the size of 4 sub-buffers if our pipelining approach is used. Without loss of generality, we assume the time spend on writing the sub-buffer data to the file system is the same as the communication cost of receiving remote requests in the sub-buffer. The communication time for sub-buffer i is denoted as C_i and the write time W_i. In the Figure 1, each block of C_i represents the time duration starting from the posts of MPI asynchronous receive calls from the non-aggregators to the completion of the receive calls. Note that the time for each W_i is more deterministic than C_i, as the MPI asynchronous communication can truly overlap with other operations, even other asynchronous communication. In this figure, the operation sequence is indicated by S_i and there are 12 steps in this example. In S_0, an aggregator posts all the asynchronous receive calls for all 4 sub-buffers and since the asynchronous calls return immediately, the call to wait for C_0 to complete in S_1 can be initiated with a very little delay after S_0. Once C_0 is complete, the write phase starts to write sub-buffer 0 to the file system. Only when W_0 finishes, sub-buffer 0 can be free for the next request. This is depicted by S_1 and S_2, where C_4 must be run after W_0. As can be seen from S_2 to S_9, the communication for request aggregation phases fully overlap with the write phases. Obviously, the larger number of two-phase I/O rounds means longer overlapping pipeline and better performance improvement. At the end of the pipeline, when all aggregation phases have been posted, the write phases must be carried out one after another.

The same example using the traditional, non-pipelined design is also depicted in the top portion of the Figure 1. An `MPI_Waitall` is called at the end of each communication block to ensure that the write data is in place, and the write phase can use the buffer to write to the file system. Since the aggregation and I/O phases run one after another, there would be no overlapping. Note that the write phase is a blocking operation for both pipelined and non-pipelined method. The end-to-end time saved by our pipelined method is shown at the bottom of the figure.

The pseudo code for the I/O aggregators of our pipeline method is shown in Figure 2. Two important values determine the depth of pipeline, the number of sub-buffers n_sub_bufs and the number of two-phase I/O rounds $ntimes$. In the conventional non-pipelined method, there is only one loop of $ntimes$ as the aggregation phase run after the I/O phase in each round. In the pipelined method, this loop is broken into two loops, the first loop starting from line 4 to 10 and the second loop from line 11 to line 14. The inner loop on lines 5 and 6 describes the posting of the asynchronous receive calls. The if condition from line 8 to 10 indicates that the I/O phase starts only when the pipeline is full. The wait and write calls for the same sub-buffer ID shown in lines 8, 9, 13, and 14 are adjacent to each other to guarantee that the aggregated data is ready to be written to the file. The maximum depth of overlapping can be seen from the if condition which is $n\_sub\_bufs - 1$. When $ntimes > n\_sub\_bufs$, our proposed method can achieve the maximum overlap. Otherwise, the pipelined method should perform about the same as the non-pipelined approach.

The efficacy of our approach depends upon the following factors: 1) cost of each phase in the two-phase I/O, and 2) the length of the pipeline. The ideal scenario happens when both phases spend equal amount of time in collective I/O

```
nprocs : total number of processes
file_domain_size : size of file domain of this aggregator
coll_buf_size : size of temporary buffer used by collective I/O
sub_buf_size : size of each sub-buffer
n_sub_bufs : number of sub-buffers
ntimes : number of two-phase I/O rounds to complete the collective I/O
-------------------------------------------------------------------
1.  n_sub_bufs ← coll_buf_size / sub_buf_size
2.  ntimes ← file_domain_size / sub_buf_size
3.  buf_id ← 0
4.  for i ← 0 . . . ( ntimes −1)
5.      for j ← 0 . . . ( nprocs −1)
6.          post an irecv call to receive from rank j to buffer buf_id
7.      buf_id ← (buf_id +1) % n_sub_bufs
8.      if i ≥ (n_sub_bufs −1)
9.          wait for the irecv calls for buffer buf_id to complete
10.         write buffer buf_id
11. for i ← 0 . . . ( n_sub_bufs −1)
12.     buf_id ← (buf_id +1) % n_sub_bufs
13.     wait for the irecv calls for buffer buf_id to complete
14.     write buffer buf_id
-------------------------------------------------------------------
```

Figure 2: Pseudocode of pipelining algorithm to overlap the request aggregation and write phases in collective I/O.

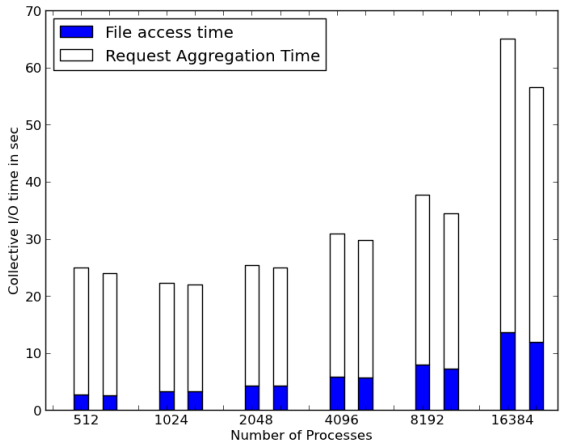

Figure 3: Timing breakdown of the aggregation and I/O phase in coll_perf benchmark. The first bar in each pair corresponds to the file domain, and second to the pipelining optimization.

call, and then our approach can completely overlap these two phases. Therefore, it can be stated that the theoretical upper bound of performance improvement is 50%. However, it is rare that the two phases take equal amount of time in real applications. In fact, the upper bound is determined by the phase with smaller percentage contribution to the overall time. For example, if the aggregation and file access phases each take 30% and 70% of the I/O time respectively, the maximum performance improvement is 30%. Hence, the cost of each phase determines the performance improvement that can be achieved with the new approach. The large number of rounds of two-phase I/O leads to a longer pipeline, more overlap and better performance. If there is only one round of two-phase I/O, then the I/O request may be too small to benefit from the pipelining design. In that case, our approach would perform same as non-pipelining method.

4. PERFORMANCE EVALUATION

Our performance evaluations were performed on Hopper, a Cray XE6 parallel computer at the National Energy Research Scientific Computing Center (NERSC). Hopper has a peak performance of 1.28 Petaflops with 153,216 processor cores, 212 TB of memory, and 2 Peta bytes of disk storage. Each compute node contains two 12-core AMD CPUs. The parallel file system available on Hopper is Lustre with a total of 156 OSTs and a peak I/O bandwidth of 35 GB per second. In our experiments, we use all 156 OSTs and a striping size of 1 MB for all different cases of benchmark runs. Our implementation is based on the ROMIO source tree released from the MPICH 2-1.4.1p1 version. We kept all the default MPI-IO hints. In particular, the default collective buffer size `coll_buf_size` 16MB is used, the sub-buffer size equal to the file striping size, which gave us 16 sub-buffers. Our evaluations used one artificial benchmark, the collective performance test from ROMIO distribution, and one application I/O kernel from GCRM. We compare the bandwidth obtained using system default ROMIO, an existing optimization based on file domain partitioning [12, 10], our pipelining approach and the ideal improvement that can be achieved. The timing breakdown results only compare file domain partitioning and pipelining method because we only intend to show that file access time remains the same. It should be noted that the performance improvement we claim is compared with the file domain optimization instead of system default ROMIO.

ROMIO collective I/O test: coll_perf is a synthetic benchmark from the ROMIO test suite. It uses a block partitioning method to write and read 3 dimensional arrays in parallel. In order to obtain stable timing and bandwidth results, we modified the benchmark to run ten iterations of the collective I/O call and kept the size of block partition constant across processes. We used a local array size of 128×128×128 and hence the I/O amount is proportional to the number of processes. Each process wrote the same amount of data independent of number of processes, but the amount of data aggregated by aggregators varied with the number of processes (up to 128 GB total data per iteration and 850MB per aggregator).

Figure 3 shows the timing breakdown of collective writes into the request aggregation phase and I/O phase. It can be seen from the figure that file access phase contributes only 11-21% to the overall I/O time for the original, non-pipelined method for 512-16,384 processes. Note that this percentage indicates the maximum amount of time that can be overlapped with the aggregation phase time. Hence, the maximum performance improvement for this set of test is 21% and ideal case would have been if there were 50% contribution by each phase. We observed an average of 19% improvement in overall I/O time, under the 50% upper bound. The figure also shows that the I/O phase time is about the same for both pipelined and non-pipelined methods. This is because the write amounts per process do not change for both methods and our proposed approach targets the request aggregation phase. However, the request aggregation time is reduced by up to 19% in the pipelined case. Only the overall collective write time and I/O phase time are re-

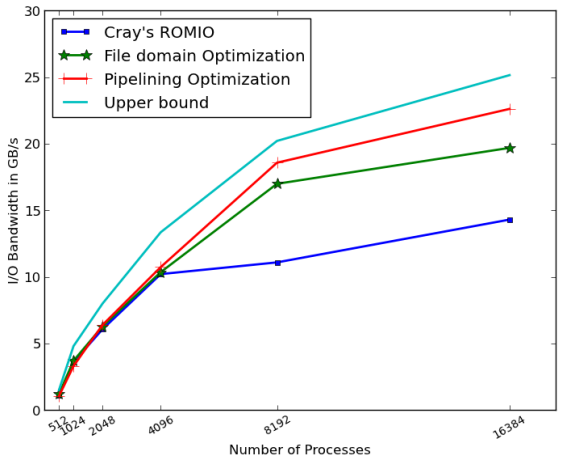

Figure 4: Write bandwidth comparison of pipelined method with other approaches for coll_perf benchmark.

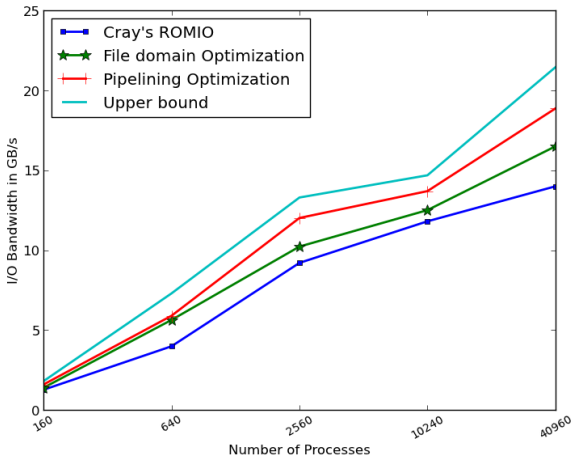

Figure 6: Write bandwidth comparison of pipelined method with other approaches for GCRM.

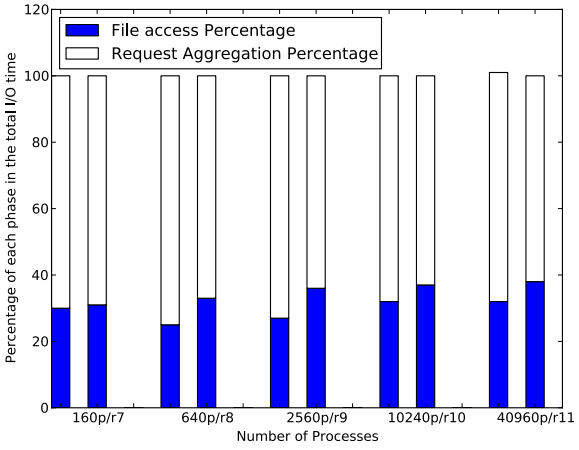

Figure 5: Timing breakdown of the aggregation and I/O phase in GCRM. The first bar in each pair corresponds to the file domain, and second to the pipelining optimization.

ported, because the aggregation phase I/O is hard to measure due to the overlapping.

Figure 4 shows the improvement in write bandwidth using the pipelined approach. For the largest run on 16K processes, we observe 19.6% improvement in the write bandwidths as compared with the File domain optimization method.

Global Cloud Resolving Model (GCRM): is a climate simulation application framework. It is designed to simulate the circulations associated with large convective clouds [16]. In our experimental setup, there are 256 horizontal layers and 38 grid variables. Each variable is approximately evenly partitioned among all the processes. We collected results for 5 cases: 160 processes with resolution level 7, 640 processes with resolution level 8, 2,560 processes with level 9, 10,240 processes with level 10 and 40,960 with level 11. Resolution levels 9, 10, and 11 correspond to the geodesic grid refinement at about 15.6, 7.8, and 3.9 km, respectively. GCRM I/O uses Geodesic Parallel I/O (GIO) library [17], which interfaces parallel netCDF (PnetCDF) [7].

Our TAU profiling results for GCRM show that on average 30% of collective I/O time is spent on the I/O phase. This allows a maximum of 30% overlapping between the two phases and hence up to 30% performance improvement. Figure 5 shows the timing breakdown of the collective I/O time in request aggregation time and file access time for GCRM as percentage of collective I/O time. We present the percentages in this figure instead of time in seconds because the execution times for the smaller run are too small to see when putting them in the same chart as the larger runs. It should be noted that file access time remains constant across the methods presented here, but due to overlapping file access and communication, the over all I/O time decreases. The constant file access time and smaller overall I/O time results in the higher percentage of file access time for pipelining in Figure 5.

From the measurement, we observe up to 14% performance improvement in the bandwidth, as shown in Figure 6. As the number of processes increases the ratio of the number of I/O aggregators to non-aggregators decreases, implying the larger file domain size per aggregator. In our pipelined design, the larger file domain size produces the longer pipeline which explains why the performance improvement is more obvious for large runs. Similar to all previous benchmarks, this figure also shows a weak scaling evaluation. For 640 and larger runs, we observe that longer pipelining improves the request aggregation time, hence improving overall I/O time. However, the file access time remains constant in each case.

5. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a multi-buffer pipelining approach to improve the performance of the request aggregation phase of the two-phase I/O. Through our pipelining design, we show that the cost of the dominant request aggregation phase can be minimized by overlapping it with the file access phase, thereby reducing the total collective I/O time. By using multiple sub-buffers of smaller size instead of a larger collective buffer, we allow asynchronous receive oper-

ation to continue using different sub-buffers while a different sub-buffer is used during the file access. The best performance can only be achieved if both request aggregation and file access contribute 50% to the collective I/O time, otherwise the amount of overlap is determined by the smaller percentage contribution of the both phases. However, in our tests we observe only 20% improvement as in reality 50% overlap does not exist. Our future work will include the analysis of the effect of different number of spread out aggregators on pipelining approach and the communication cost in other collective I/O operations i.e. open call and investigate the implications of the proposed optimizations on I/O benchmarks and production applications.

6. REFERENCES

[1] Javier García Blas, Florin Isaila, David E. Singh, and J. Carretero. View-based collective i/o for mpi-io. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '08, pages 409–416, Washington, DC, USA, 2008. IEEE Computer Society.

[2] K. Coloma, A. Ching, A. Choudhary, W. Liao, R. Ross, R. Thakur, and Lee Ward. A new flexible MPI collective I/O implementation. In *the IEEE Conference on Cluster Computing*, September 2006.

[3] K. Coloma, A. Choudhary, W. Liao, W. Lee, E. Russell, and N. Pundit. Scalable High-level Caching for Parallel I/O. In *the International Parallel and Distributed Processing Symposium*, April 2004.

[4] K. Coloma, A. Choudhary, W. Liao, W. Lee, and S. Tideman. DACche: Direct Access Cache System for Parallel I/O. In *the 20th International Supercomputer Conference*, June 2005.

[5] J. del Rosario, R. Brodawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In *the Workshop on I/O in Parallel Computer Systems at IPPS*, pages 56–70, April 1993.

[6] Phillip M. Dickens. Improving collective i/o performance using threads. In *In Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 38–45, 1999.

[7] J. Li et al. Parallel netCDF: A High-Performance Scientific I/O Interface. In *SuperComputing Conference*, 2003.

[8] http://www.top500.org/.

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

[10] W. Liao. Design and Evaluation of MPI File Domain Partitioning Methods under Extent-Based File Locking Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 22(2):260–272, February 2011.

[11] W. Liao, A. Ching, K. Coloma, A. Choudhary, and L. Ward. An Implementation and Evaluation of Client-Side File Caching for MPI-IO. In *the International Parallel and Distributed Processing Symposium*, March 2007.

[12] W. Liao and A. Choudhary. Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2008.

[13] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPI-IO Output Performance with Active Buffering Plus Threads. In *the International Parallel and Distributed Processing Symposium*, April 2003.

[14] B. Nitzberg and V. Lo. Collective Buffering: Improving Parallel I/O Performance. In *the Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 148–157, August 1997.

[15] Christina M. Patrick, Seung Woo Son, and Mahmut Taylan Kandemir. Enhancing the performance of mpi-io applications by overlapping i/o, computation and communication. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 277–278, New York, NY, USA, 2008. ACM.

[16] D. Randall, M. Khairoutdinov, A. Arakawa, and W. Grabowski. Breaking the Cloud Parameterization Deadlock. *Bull. Amer. Meteor. Soc.*, 84:1547–1564, 2003.

[17] K. Schuchardt, B. Palmer, J. Daily, T. Elsethagen, and A. Koontz. IO Strategies and Data Services for Petascale Data Sets from a Global Cloud Resolving Model. *Journal of Physics: Conference Series*, 78, 2007.

[18] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed Collective I/O in Panda. In *Supercomputing*, November 1995.

[19] 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.

[20] Weikuan Yu and Jeffrey Vetter. Parcoll: Partitioned collective i/o on the cray xt. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 562–569, Washington, DC, USA, 2008. IEEE Computer Society.

[21] 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 *the International Parallel and Distributed Processing Symposium*, March 2009.