Cooperative Write-Behind Data Buffering for MPI I/O

Wei-keng Liao[1], Kenin Coloma[1], Alok Choudhary[1], and Lee Ward[2]

[1] Electrical and Computer Engineering Department, Northwestern University
[2] Scalable Computing Systems Department, Sandia National Laboratories

Abstract. Many large-scale production parallel programs often run for a very long time and require data checkpoint periodically to save the state of the computation for program restart and/or tracing the progress. Such a write-only pattern has become a dominant part of an application's I/O workload and implies the importance of its optimization. Existing approaches for write-behind data buffering at both file system and MPI I/O levels have been proposed, but challenges still exist for efficient design to maintain data consistency among distributed buffers. To address this problem, we propose a buffering scheme that coordinates the compute processes to achieve the consistency control. Different from other earlier work, our design can be applied to files opened in read-write mode and handle the patterns with mixed MPI collective and independent I/O calls. Performance evaluation using BTIO and FLASH IO benchmarks is presented, which shows a significant improvement over the method without buffering.

Keywords: Write behind, MPI I/O, file consistency, data buffering, I/O thread

1 Introduction

Periodical checkpoint write operations are commonly seen in today's long-running production applications. Checkpoint data typically are snapshot of the current computation status to be used for progress tracking and/or program restart. Once written, files created by checkpointing are usually not touched for the rest of the run. In many large-scale applications, such write-once-never-read patterns are observed to dominate the overall I/O workload and, hence, designing efficient techniques for such operations becomes very important. Write-behind data buffering has been known to operating system designers as a way to speed up sequential writes [1]. Write-behind buffering accumulates multiple writes into large contiguous file requests in order to better utilize the I/O bandwidth. However, implementing the write-behind strategy requires the support of client-side caching which often complicates the file system design due to the cache coherence issues. System-level implementations for client-side caching often hold dedicated servers or agents to be responsible for maintaining the coherence. In the parallel environment, this problem gets even obvious since processes running the same parallel application tends to operate their I/O on shared files concurrently. User-level implementation of write-behind data buffering has been proposed in [2] which demonstrated a significant performance improvement when the buffering scheme is embedded in ROMIO [3], an I/O library implementation for Message Passing Interface [4]. However, due to the possible file consistency problem, it is limited to MPI collective write operations with the file opened in write-only mode.

We propose *cooperative write-behind buffering*, a scheme that can benefit both MPI collective and independent I/O operations. To handle the consistency issue, we keep tracking the buffering status at file block level and at most one copy of the file data can be buffered among processes. The status metadata is cyclically distributed among the application processes such that processes cooperate with each other to maintain the data consistency. To handle MPI independent I/O, each process must respond to the queries, remote and local, to the status assigned without explicitly stopping the program's main thread. Thus, we create an I/O thread in each process to handle the requests to the status as well as the buffered data. Our implementation requires every process first look up the buffering status for file blocks covered by the read/write request to determine whether the request should create a new buffer or overwrite the existing buffers (locally or remotely). To ensure I/O atomicity, a status-locking facility is implemented and locks must be granted prior to the read/write calls. We evaluate the cooperative buffering on the IBM SP at San Diego supercomputing center using its GPFS file system. Two sets of I/O benchmarks are presented: BTIO, and FLASH I/O. Compared with the native I/O method without data buffering, cooperative buffering shows a significant performance enhancement for both benchmarks.

The rest of the paper is organized as follows. Section 2 discusses the background information and related works. The design and implementation for cooperative buffering is presented in section 3. Performance results are given in section 4 and the paper is concluded in section 5.

2 Background

Message passing interface (MPI) standard defines two types of I/O functions: collective and independent calls [4]. Collective I/O must be called by all processes that together opened the file. Many collective I/O optimizations take advantage of this synchronization requirement to exchange information among processes such that I/O requests can be analyzed and reconstructed for better performance. However, independent I/O does not require process synchronization, which makes existing optimizations difficult to apply.

2.1 Active Buffering and ENWRICH Write Caching

Active buffering is considered an optimization for MPI collective write operations [2]. It buffers output data locally and uses an I/O thread to perform write requests at background. Using I/O threads allows to dynamically adjust the size of local buffer based on available memory space. Active buffering creates only one thread for the entire run of a program, which alleviates the overhead of spawning a new thread every time a collective I/O call is made. For each write request, the main thread allocates a buffer, copies the data over, and appends this buffer into a queue. The I/O thread, running at background, later retrieves the buffers from the head of the queue, issues write calls to the file system, and releases the buffer space. Although write behind enhances parallel write performance, active buffering is applicable if the I/O patterns only consist of write operations. Lacking of consistency control, active buffering could not handle the operations mixed with reads and writes as well as independent and collective calls.

Similar limitations are observed in ENWRICH write caching scheme which is a system-level optimization [5]. The client-side write caching proposed in ENWRICH can only handle the files that are opened in write-only mode. For I/O patterns mixed with reads and writes, caching is performed at I/O servers due to the consistency concern. Similar to active buffering, ENWRICH also appends each write into a write queue and all files share the same cache space. During the flushing phase, the I/O threads on all processes coordinate the actual file operations.

2.2 I/O Thread in GPFS

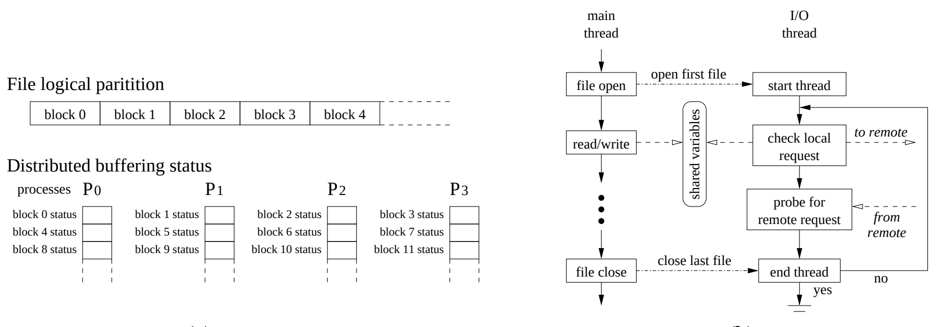
IBM GPFS parallel file system performs the client-side file caching and adopts a strategy called data shipping for file consistency control [6, 7]. Data shipping binds each GPFS file block to a unique I/O agent which is responsible for all the accesses to this block. The file block assignment is made in round-robin striping scheme. Any I/O operations on GPFS must go through the I/O agents which will ship the requested data to appropriate processes. To avoid incoherent cache data, a distributed file locking is used to minimize the possibility of I/O serialization that can be caused by lock contention. I/O agents are multi-threaded residing in each process and are responsible for combining I/O requests in collective operations. I/O thread also performs advanced caching strategies at background, such as read ahead and write behind.

3 Design and Implementation

The idea of cooperative buffering is to let application processes cooperate with each other to manage a consistent buffering scheme. Our goals are first to design a write-behind data buffering as an MPI I/O optimization at client side without adding overhead to I/O servers. Secondly, we would like to support read-write operations in arbitrary orders, which implies the incorporation of consistency control. Thirdly, the buffering scheme would benefit both MPI collective and independent I/O.

3.1 Buffering Status Management and Consistency Control

We logically divide a file into blocks of the same size and buffering status of these blocks is assigned in a round-robin fashion across the MPI processes that together open the file. Our consistency control is achieved by tracking the buffering status of each block and keeps at most one copy of file data in the buffers globally. As illustrated in Figure 1(a), the status for block *i* is held by the process of rank (*i* mod *nproc*), where *nproc* is the number of processes in the MPI communicator supplied at file open. The status indicates if the block is buffered, its file offset, current owner process id, a dirty flag, byte range of the dirty data, and the locking mode. Note that buffering is performed for all opened files, but buffering status is unique to each file. An I/O request begins with checking the status of the blocks covered by the request. If the requested blocks have not been buffered by any process, the requesting process will buffer them locally



(a) **(b)**

Fig. 1. (a) The buffering status is statically distributed among processes in a round-robin fashion. (b) Design of the I/O thread and its interactions with the main thread and remote requests.

and update the status. Otherwise, the request will be forwarded to the owner(s) of the blocks and appropriate reads or over-writes from/to the remote buffer are performed. Unlike active buffering and ENWRICH appending all writes into a queue, cooperative buffering writes to existing buffers whenever possible. Since at most one copy of the file data can be buffered in the process memory at any time, data consistency is maintained. In addition, a status locking facility is implemented, in which locks to the file blocks covered by an I/O request must be all granted before proceeding with any operation on the blocks. To enforce MPI sequential consistency and atomicity, the locks to multiple blocks are granted in an increasing order. For example, if an I/O request covers file blocks from i to j, where $i \leq j$, lock request for block k, $i \leq k \leq j$, will not be issued until lock to block $(k-1)$ is granted. This design is similar to the two-phase locking method [8] used to serialize multiple overlapping I/O requests to guarantee the I/O atomicity.

3.2 I/O Thread

Since buffered data and buffering status are distributed among processes, each process must be able to respond to remote requests for accessing to the status and buffered data stored locally. For MPI collective I/O, remote queries can be fulfilled through inter-process communication during the process synchronization. However, the fact that MPI independent I/O is asynchronous makes it difficult for one process to explicitly receive remote requests. Our design employs an I/O thread in each process to handle remote requests without interrupting the execution of the main thread. To increase the portability, our implementation uses the POSIX standard thread library [9]. Figure 1(b) illustrates the I/O thread design from the viewpoint of a single process. Details of the I/O thread design are described as follows.

– The I/O thread is created when the application opens the first file and destroyed when the last file is closed. Each process can have multiple files opened, but only one thread is created.
– The I/O thread performs an infinite loop to serve the local and remote I/O requests.

– All I/O and communication operations are carried out by the I/O thread only.
– A conditional variable protected by a mutual exclusion lock is used to communicate the two threads.
– To serve remote requests, the I/O thread keeps probing for incoming I/O requests from any process in the MPI communicator group. Since each opened file is associated with a communicator, the probe will check for all the opened files.
– The I/O thread manages the local memory space allocation which includes creating/releasing buffers and resizing the status data.

3.3 Flushing Policy

Since our file consistency control ensures that only one copy of the file data can be buffered globally among processes and any read/write operations must check the status first, it is not necessary to flush buffered data prior to end of the run. In our implementation, buffered data can be explicitly flushed when file is closed or the file flushing call is made. Otherwise, implicit data flushing is needed only when the application runs out of memory, in which the memory space management facility handles the space overflow. Our design principle for data flushing includes: 1) declining buffering for overly large requests exceeding one-forth of entire memory size (direct read/write calls are made for such requests); 2) least-recent-used buffered data is flushed first (an accessing time stamp is associated with each data buffer); and 3) when flushing, all buffers are examined if any two buffers can be coalesced to reduce the number of write calls. For file systems that do not provide consistency automatically, we mimic the approach used in ROMIO that wraps byte-range file locking around each read/write call to disable client-side caching [10].

3.4 Incorporate into ROMIO

We place cooperative buffering at the ADIO layer of ROMIO to catch every read/write system call and determines whether the request should access the existing buffers or create a new buffer. ADIO is an abstract-device interface providing uniform and portable I/O interfaces for parallel I/O libraries [11]. This design preserves the existing optimizations used by ROMIO, such as two-phase I/O and data sieving, both implemented above ADIO [3, 12]. In fact, cooperative buffering need not know if the I/O operation is collective or independent, since it only deals with system read/write calls.

4 Experimental results

The evaluation of cooperative buffering implementation was performed using BTIO benchmark and FLASH I/O benchmark on the IBM SP machine at San Diego Supercomputing Center. The IBM SP contains 144 Symmetric Multiprocessing (SMP) compute nodes and each node is an eight-processor shared-memory machine. We use the IBM GPFS file system to store the files. The peak performance of the GPFS is 2.1 GBytes per second for reads and 1 GBytes per second for writes. The I/O will approximately max out at about 20 compute nodes. In order to simulate a distributed-memory environment, we ran the tests using one processor per compute node.

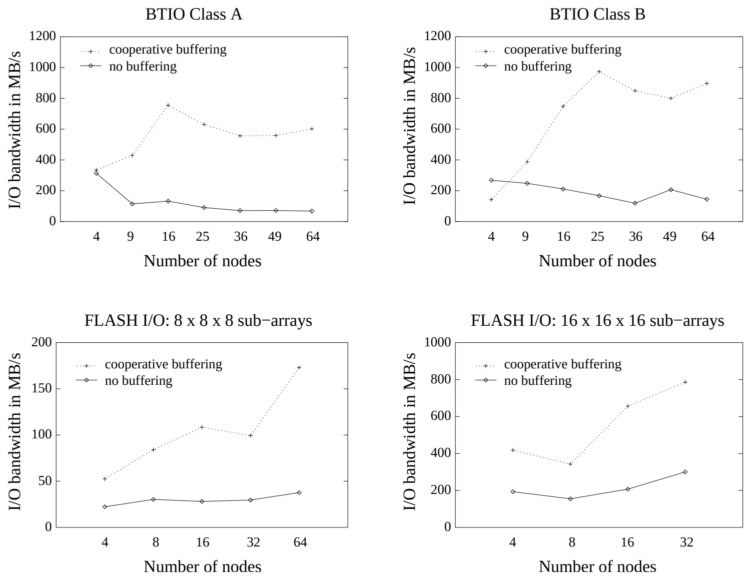


Fig. 2. I/O bandwidth results for BTIO and FLASH I/O benchmarks.

4.1 BTIO benchmark

BTIO is the I/O benchmark from NASA Advanced Supercomputing (NAS) parallel benchmark suite (NPB 2.4) [13]. BTIO uses a block-tridiagonal (BT) partitioning pattern on a three-dimensional array across a square number of compute nodes. Each processor is responsible for multiple Cartesian subsets of the entire data set, whose number increases as the square root of the number of processors participating in the computation. BTIO provides four types of evaluations, each with different I/O implementations, including MPI collective I/O, MPI independent I/O, Fortran I/O, and separate-file I/O. In this paper, we only present the performance results for the type of using MPI collective I/O, since collective I/O generally results in the best performance [14]. The benchmark performs 40 collective MPI writes followed by 40 collective reads. We evaluated two I/O sizes: classes A and B, which generate I/O amount of 800 MBytes and 3.16 GBytes, respectively. Figure 2 compares the bandwidth results of using cooperative buffering with the native approach (without buffering.) We observe that cooperative buffering out-performs the native approach in most of the cases. Especially, when the number of compute nodes becomes large, cooperative buffering can achieve bandwidth near the system peak performance. The main contribution to this performance improvement is due to the effect of write behind and read from buffered data.

4.2 FLASH I/O benchmark

FLASH is an AMR application that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on neutron stars and white dwarfs [15]. The FLASH I/O benchmark [16] uses HDF5 for writing checkpoints, but underneath is using MPI I/O for performing parallel reads and writes. The in-memory data structures are 3D sub-arrays of size $8 \times 8 \times 8$ or $16 \times 16 \times 16$ with a perimeter of four guard cells that are left out of the data written to files. In the simulation, 80 of these blocks are held by each processor. Each of these data elements has 24 variables associated with it. Within each file, the data for the same variable must stored contiguously. The access pattern is non-contiguous both in memory and in file, making it a challenging application for parallel I/O systems. Since every processor writes 80 FLASH blocks to file, as we increase the number of clients, the dataset size increases linearly as well.

Figure 2 compares the bandwidth results between the I/O implementation with and without cooperative buffering. The I/O amount is proportional to the number of compute nodes, ranging from 72.94 MBytes to 1.14 GBytes for the case of $8 \times 8 \times 8$ arrays and from 573.45 MBytes to 4.49 GBytes for the case of $16 \times 16 \times 16$ arrays. In the case of using $8 \times 8 \times 8$ array size, we can see that the I/O bandwidth for both implementations is far from the system peak performance. This is because FLASH I/O generates many non-contiguous and small I/O requests and the system peak performance can only be achieved by large contiguous I/O requests. The bandwidth improves significantly when we increase the array size to $16 \times 16 \times 16$. Similar to the BTIO benchmark, the I/O performance improvement demonstrates the effect of write behind.

4.3 Sensitivity Analysis

Due to the consistency control, cooperative buffering bears the cost of remotely and.or locally buffering status inquiry each time an I/O request is made. For the environment with a relative slow communication network or the I/O patterns with many small requests, this overhead may become significant. Another parameter that may affect the I/O performance is the file block size. The granularity of file block size determines the number of local/remote accesses generated from an I/O request. For different file access patterns, one file block size cannot always deliver the same performance enhancement. For the patterns with frequent and small amounts of I/O, using a large file block can cause contention when multiple requests access to the same buffers. On the other hand, if small file block size is used when the access pattern is less frequent and with large amount of I/O, a single large request can result in many remote data accesses. In most cases, such parameters can only be fine-tuned by the application users. In MPI, such user inputs usually are implemented through `MPI_Info` objects which, in our case, can also be used to activate or disable cooperative buffering.

5 Conclusions

Write-behind data buffering is known to be able to improve I/O performance by combining multiple small writes into large writes to be executed later. However, the overhead for write behind is the cost of maintaining file consistency. The cooperative buffering proposed in this paper addresses the consistency issue by coordinating application processes to manage buffering status data at file block level. This buffering scheme can benefit both MPI collective and independent I/O while the file open mode is no longer limited to write-only. The experimental results have shown a great improvement for two

I/O benchmarks. In the future, we plan to investigate in depth the effect of the file block size and study irregular access patterns from scientific applications.

6 Acknowledgments

This work was supported in part by Sandia National Laboratories and DOE under Contract number 28264, DOE's SCiDAC program (Scientific Data Management Center), award number DE-FC02-01ER25485, NSF's NGS program under grant CNS-0406341, NSF/DARPA ST-HEC program under grant CCF-0444405, and NSF through the SDSC under grant ASC980038 using IBM DataStar.

References

1. Callaghan, B.: NFS Illustrated. Addison-Wesley (2000)
2. Ma, X., Winslett, M., Lee, J., Yu, S.: Improving MPI-IO Output Performance with Active Buffering Plus Threads. In: the International Parallel and Distributed Processing Symposium (IPDPS). (2003)
3. Thakur, R., Gropp, W., Lusk, E.: 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. (1997)
4. Message Passing Interface Forum: MPI-2: Extensions to the Message Passing Interface. (1997) http :// www.mpi-forum.org / docs / docs.html.
5. Purakayastha, A., Ellis, C.S., Kotz, D.: ENWRICH: A Compute-Processor Write Caching Scheme for Parallel File Systems. In: the Fourth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS). (1996)
6. Prost, J., Treumann, R., Hedges, R., Jia, B., Koniges, A.: MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS. In: Supercomputing. (2001)
7. Schmuck, F., Haskin, R.: GPFS: A Shared-Disk File System for Large Computing Clusters. In: the Conference on File and Storage Technologies (FAST'02). (2002) 231–244
8. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
9. IEEE/ANSI Std. 1003.1: Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language]. (1996)
10. Thakur, R., Gropp, W., Lusk, E.: On Implementing MPI-IO Portably and with High Performance. In: the Sixth Workshop on I/O in Parallel and Distributed Systems. (1999) 23–32
11. Thakur, R., Gropp, W., Lusk, E.: An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In: the 6th Symposium on the Frontiers of Massively Parallel Computation. (1996)
12. Thakur, R., Gropp, W., Lusk, E.: Data Sieving and Collective I/O in ROMIO. In: the 7th Symposium on the Frontiers of Massively Parallel Computation. (1999)
13. Wong, P., der Wijngaart, R.: NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, NASA Ames Research Center, Moffet Field, CA (2003)
14. Fineberg, S., Wong, P., Nitzberg, B., Kuszmaul, C.: PMPIO - A Portable Implementation of MPI-IO. In: the 6th Symposium on the Frontiers of Massively Parallel Computation. (1996)
15. Fryxell, B., Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Tufo, H.: FLASH: An Adaptive Mesh Hydrodynamics Code for Modelling Astrophysical Thermonuclear Flashes. Astrophysical Journal Suppliment (2000) 131–273
16. Zingale, M.: FLASH I/O Benchmark Routine – Parallel HDF 5 (2001) http://flash.uchicago.edu/~zingale/flash_benchmark_io.