2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Multithreaded Two-Phase I/O: Improving Collective MPI-IO Performance on a Lustre File System

Yuichi Tsujita*§, Kazumi Yoshinaga*§, Atsushi Hori*§, Mikiko Sato†§, Mitaro Namiki†§ and Yutaka Ishikawa‡*
*RIKEN AICS, Hyogo, Japan
Email: {yuichi.tsujita, kazumi.yoshinaga, ahori}@riken.jp
†Tokyo University of Agriculture and Technology, Tokyo, Japan
Email: {mikiko@namikilab, namiki@cc}.tuat.ac.jp
‡The University of Tokyo, Tokyo, Japan
Email: ishikawa@is.s.u-tokyo.ac.jp
§JST CREST

***Abstract*—ROMIO, a representative MPI-IO implementation, has been widely used in recent large-scale parallel computations. The two-phase I/O optimization scheme of ROMIO improves I/O performance for non-contiguous access patterns; however, this scheme still has room to improve performance to make it suitable for recent data-intensive computing. We propose overlapping data exchange operations with file I/O operations by using a multithreaded scheme to achieve further I/O throughput improvement. We show up to 60% improvement by the multithreaded two-phase I/O relative to the original two-phase I/O in performance evaluation of collective write operations on a Lustre file system of a Linux PC cluster.**

I. Introduction

ROMIO [1] is one such widely used MPI-IO [2] implementation. ROMIO's two-phase I/O optimization (hereinafter, TP-IO) [3] improves collective I/O performance for non-contiguous access patterns. The TP-IO consists of a series of data exchange and file I/O operations using a temporary buffer referred to as a collective buffer (hereinafter, CB). I/O performance improvement is realized by contiguous I/O accesses which consist of data regions of other MPI processes together with own data regions, followed by data exchanges between MPI processes to collect own data which were on different processes.

However the scheme still has room to overlap file I/O with data exchanges for further I/O throughput improvement. In this paper, we propose a multithreaded TP-IO implementation by using Pthreads [4] to overlap file I/O with data exchange in a pipelining manner. We report performance results of the multithreaded implementation on a PC cluster system which had a Lustre file system [5]. Remarkable performance improvements up to 60% are observed compared with the original one. Furthermore, multiple slots in the queues also had a positive impact in performance improvement. In the remainder of this paper, we first explain the mechanism of TP-IO briefly and then explain our multithreaded implementation, in Sec. II. Performance evaluation results are discussed in Sec. III. Related work is mentioned in Sec. IV, and finally we give our conclusions in Sec. V.

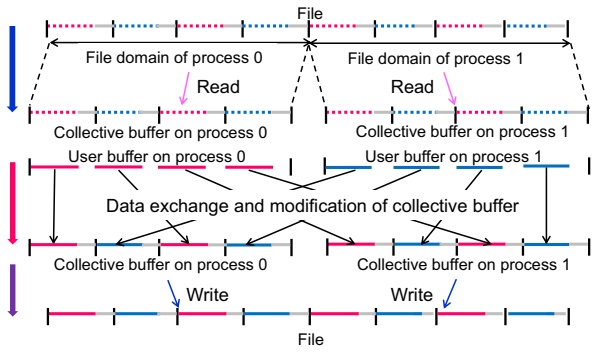


Figure 1. Typical collective write using TP-IO with two MPI processes

II. Performance Improvements in Two-Phase I/O

A. Two-Phase I/O implementation

TP-IO is designed to improve collective I/O performance for non-contiguous access patterns. When trying to access non-contiguous data regions, each process should have many file accesses in a conventional way. However, this scheme leads to poor performance because of the large amount of I/O overhead inclusions. Therefore, TP-IO organizes contiguous accesses including data gaps in the file accesses by splitting the total data region evenly between the processes. Figure 1 shows a typical TP-IO operation in a collective write with two MPI processes. In the figure, the red and blue dotted lines represent the destination of data regions by processes 0 and 1, respectively, and gray lines denote data gaps. Solid red and blue lines indicate that data is filled there.

In order to mitigate the I/O operation overhead inclusions and improve throughput, the total data region including data gaps of a target file is divided into data chunks aligned to a CB. Once every process has read its assigned data region and stored the content, including the data gaps, in its CB, every process performs data exchange operations to get its

1066-6192/14 $31.00 © 2014 IEEE
DOI 10.1109/PDP.2014.46

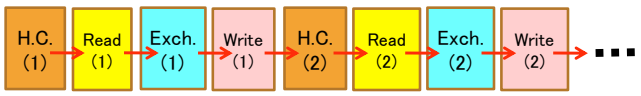


Figure 2. Typical TP-IO flow in collective write operations, where *H.C.*, *Read*, *Exch.*, and *Write* stand for hole check, read, data exchange, and write operations, respectively

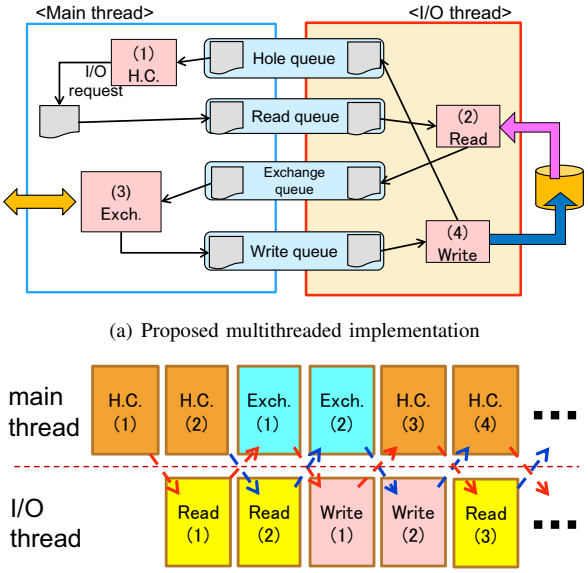
target data on its CB. Finally, a modified CB including data gaps is written back to a target file. The operation sequence is repeated until the entire data region has been accessed.

Figure 2 depicts the typical TP-IO operation flow in collective write operations. As depicted in this figure, we have four operations, hole check (*H.C.*), read (*Read*), data exchange (*Exch.*), and write (*Write*) operations, where every operation manages a chunk of a data region aligned to a CB size. Every operation is sequentially carried out and repeated until all the data chunks have been accessed. In this paper, a sequence from a hole check operation to a write operation is referred to as one TP-IO cycle.

B. Multithreaded Two-Phase I/O

We focused on the possibility of overlapping data exchange operations with I/O operations, as we explained above. After careful examinations to find a favorable solution for a pipelining implementation, we adopted a multithreaded I/O scheme by using Pthreads, although we found another option using asynchronous I/O functions. To the best of our knowledge, availability and I/O performance of the asynchronous I/O functions depend on the operating system. Because almost all operating systems including Linux support Pthreads, we chose Pthreads for our implementation.

We have already implemented a multithreaded TP-IO for collective read operations only [6], and collective write operations have been supported in the multithreaded TP-IO by extended implementations of the collective read mechanism. Figure 3(a) shows a proposed multithreaded scheme for collective write operations. A main thread invokes another thread, referred to as the I/O thread, in order to achieve multithreaded TP-IO. The main thread manages the hole check and data exchange operations, while the I/O thread manages the read and write operations. Both threads share four queues named the hole queue, read queue, exchange queue, and write queue. An I/O request issued by the main thread is enqueued into the read queue after a hole check step ((1) in Fig. 3(a)). Then the I/O thread dequeues the request and carries out read operations according to the request, followed by enqueueing an I/O request into an exchange queue ((2) in Fig. 3(a)). The main thread then dequeues the I/O request and starts its data exchange step, after which the main thread passes the request to a write queue ((3) in Fig. 3(a)). Finally, the I/O thread starts write operations once it dequeues the request from a write queue ((4) in Fig. 3(a)). An I/O thread also enqueues a dummy request into a hole



(a) Proposed multithreaded implementation

(b) Multithreaded operation scheme (2 slots case)

Figure 3. Proposed multithreaded TP-IO for collective write operations

Table I
SPECIFICATIONS OF A PC NODE OF THE T2K-TODAI

CPU	AMD Opteron 8356 Barcelona (2.3 GHz, 4 cores) × 4
Memory	32 GiB (8 GiB × 4)
Interconnect	Myrinet 10 Gbps × 2, 1 Gbps Ethernet × 2
OS	Linux kernel 2.6.18-53
glibc	version 2.5
Parallel file system	Lustre version 1.8.9

queue to initiate the next operation. The above sequence is repeated until all the data chunks have been processed.

Fig. 3(b) shows TP-IO operation scheme with 2 slots in queues. If we have multiple slots, each thread can go forward without waiting for completion of the running task in the next operation phase. As a result, good overlap of main thread operations with I/O thread operations is expected. In our case, the total number of slots in queues can be managed by MPI_Info_set with a key-value pair.

III. PERFORMANCE EVALUATION

A. Evaluation platform

I/O performance is evaluated on a PC cluster system at the Information Technology Center of The University of Tokyo. For simplicity, we refer to this system as T2K-Todai in the remainder of this paper. The specifications of a PC node of T2K-Todai are summarized in Table I. We utilized 32 PC nodes, which were in a different network segment from commonly used PC nodes, in order to exclude interference from other users' computing tasks. Each node had two Myrinet 10 Gbps links [7]. File I/O operations were done on a Lustre file system (version 1.8.9) consisting of 1 Meta

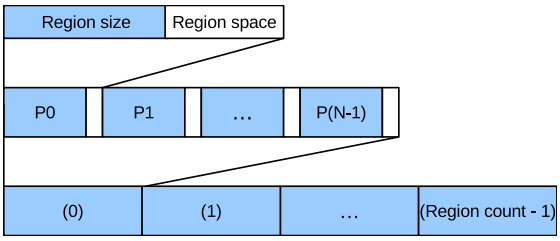


Figure 4. Non-contiguous access pattern generated by HPIO benchmark

Data Server (MDS) and 30 Object Storage Targets (OSTs) via a single Myrinet 10 Gbps link. During the evaluation, all the OSTs were utilized with several striping sizes. The number of processes which performed I/O operations, which is referred as I/O aggregators, was set to be 30, which was same with the number of OSTs. In order to examine the impact of the number of TP-IO cycles on I/O performance, we changed the striping size of the Lustre file system. We remounted the Lustre file system prior to every I/O evaluation to avoid file cache effects.

Although the T2K-Todai had its own MPI library named MPICH-MX provided by Myricom [7], we utilized an MPICH2 library [8] version 1.4.1p1 with our enhanced ROMIO. The library was built to utilize a Myricom's MX library for MPI communications.

B. HPIO benchmark

We chose the HPIO benchmark program [9] from among the many kinds of MPI-IO benchmarks because the HPIO benchmark code provides flexible pattern combination for derived data type generation. We evaluated collective write performance for some derived data types including data gaps as shown in Figure 4. The HPIO benchmark has three parameters, region size, region space, and region count, for generating a non-contiguous access pattern by using a derived data type. Region size refers to the data region assigned for one of the MPI processes. One data group is organized by connecting all the data regions and the following data gaps which size is described by region space from every MPI process. Region count denotes the number of groups. In this evaluation, we measured I/O throughput for a non-contiguous access pattern by incorporating a 256 B data gap for every 488 B region size. Region count was configured to generate about 1 GiB per process. Note that the total data region, including data gaps, was read or written in TP-IO cycles due to its read-modify-write manner. The HPIO benchmark calculated I/O throughput values by dividing the data region size without data gaps (thus only the target data region size from a user program view) by the I/O time. In this evaluation, we calculated mean values obtained from 20 iterations of MPI-IO function call.

C. I/O throughput evaluation

By using the HPIO benchmark code, I/O throughput of collective MPI-IO write operations was measured. In order to show performance improvements in the multithreaded scheme relative to the original one, we show relative I/O throughput values obtained by dividing the throughput value of a multithreaded case by that of the original one.

Figure 5 shows the relative I/O throughput values in file accesses with data gaps with respect to the number of TP-IO cycles. Here "ior-2," "ior-4," and "ior-8" denote multithreaded TP-IO with 2, 4, and 8 slots in queues, respectively. Performance improvements up to about 60% relative to the original performance were achieved. Also, performance improved as the number of slots in queues increased at almost all the TP-IO cycles. It is considered that a larger number of slots is suitable for I/O performance improvement.

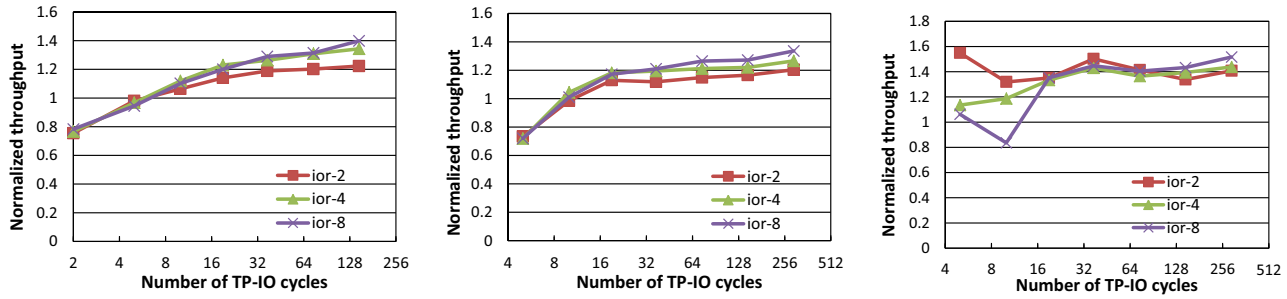
A smaller number of TP-IO cycles had an especially large impact on performance degradation with larger file size. For example, we had performance drops at 5 and 10 TP-IO cycles in the case shown in Fig. 5(c). In the 5 TP-IO cycles case, every 30 I/O aggregator process had 2 GiB CB, and thus every aggregator participated in file I/O four times, and from 17 to 18 aggregators participated in file I/O in the last cycle. In contrast, in the 10 TP-IO cycles case, all 30 aggregator processes participated in file I/O 9 times and 4 or 5 aggregators participated in the last cycle. From the point of view of I/O throughput, the latter case had disadvantages in performance due to the small number of aggregator processes involved in the last cycle. However, for larger numbers of TP-IO cycles, such disadvantages become negligible.

IV. Related Work

MPI-IO performance improvements have been realized in many research studies. Blas et al. [10] proposed view-based collective I/O to improve TP-IO operations by eliminating communications to have access information on every I/O aggregators prior to every TP-IO cycle unlike the original TP-IO. Instead their scheme informs file access information of all the MPI processes to every I/O aggregators prior to I/O operations.

Blas et al. also reported an improved collective I/O performance by letting I/O operations run in the background and having read-ahead techniques on a GPFS [11]. They adopted a multithreaded method to overlapped I/O operations.

Sehrish et al. [12] proposed an overlapping scheme similar to our proposal, however they realized the scheme by using asynchronous MPI communication functions inside the TP-IO. Performance of their scheme strictly depends on performance of the asynchronous MPI communication functions, and there may be small possibilities to have further I/O performance improvement. On the other hand, our implementation has adopted multithreaded way using



(a) np=64, 68.5 GiB (b) np=128, 137 GiB (c) np=256, 274 GiB

Figure 5. Relative collective write throughput to the original ROMIO for (a) 68.5 GiB with 64 processes, (b) 137 GiB with 128 processes, and (c) 274 GiB with 256 processes using the HPIO benchmark with data gaps

Pthreads. Since there are more flexibility in implementation in Pthreads programming and its implemented layer is closer to underling system software than their implementation, our scheme has larger possibilities to tune I/O performance.

V. Concluding Remarks

We have proposed a multithreaded TP-IO for further I/O performance improvement to overcome the I/O performance wall. We have observed that the multithreaded implementation has outperformed the original one up to 60% in I/O throughput evaluation of MPI-IO's collective write API for non-contiguous access patterns by using the HPIO benchmark code. One of the reasons for the improvement in the multithreaded case was the higher overlap of I/O operations by an invoked I/O thread with communications by a main thread. Furthermore, multiple slots in shared queues led to further improvement. For further I/O performance improvement, well-tuned overlapping is essential. For the purpose of such overlapping, improved Pthreads programming in the multithreaded TP-IO implementation is our future work. Proposing of an estimation model for optimal number of slots in shared queues and appropriate TP-IO cycles is also our future work.

Acknowledgment

This research has been partially supported by JST CREST. The authors also would like to thank the Information Technology Center, the University of Tokyo for assistance in using the T2K-Todai.

References

[1] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, 1999, pp. 23–32.

[2] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[3] R. Thakur, W. Gropp, and E. Lusk, "Optimizing noncontiguous accesses in MPI-IO," *Parallel Computing*, vol. 28, no. 1, pp. 83–105, 2002.

[4] Institute of Electrical and Electronic Engineers, "Information Technology – Portable Operating Systems Interface – Part 1: System Application Program Interface (API) – Amendment 2: Threads Extensions [C Languages]," 1995.

[5] Lustre. [Online]. Available: http://wiki.lustre.org/index.php/Main_Page

[6] Y. Tsujita, H. Muguruma, K. Yoshinaga, A. Hori, M. Namiki, and Y. Ishikawa, "Improving collective I/O performance using pipelined two-phase I/O," in *Proceedings of the 2012 Symposium on High Performance Computing*, ser. HPC '12. Society for Modeling and Simulation International, March 2012, pp. 7:1–7:8, (CD-ROM).

[7] Myricom, Inc. [Online]. Available: http://www.myricom.com/

[8] MPICH. [Online]. Available: http://www.mpich.org/

[9] A. Ching, A. Choudhary, W. keng Liao, L. Ward, and N. Pundit, "Evaluating I/O characteristics and methods for storing structured scientific data," in *Proceedings 20th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, April 2006, p. 49.

[10] J. G. Blas, F. Isaila, D. E. Singh, and J. Carretero, "View-based collective I/O for MPI-IO," in *CCGRID*, 2008, pp. 409–416.

[11] J. G. Blas, F. Isaila, J. Carretero, D. Singh, and F. Garcia-Carballeira, "Implementation and evaluation of file write-back and prefetching for MPI-IO over GPFS," *International Journal of High Performance Computing Applications*, vol. 24, pp. 78–92, 2010.

[12] S. Sehrish, S. W. Son, W. keng Liao, A. Choudhary, and K. Schuchardt, "Improving collective I/O performance by pipelining request aggregation and file access," in *20th European MPI User's Group Meeting, EuroMPI'13, Madrid, Spain*, J. G. Blas, J. Carretero, and J. Dongarra, Eds. ACM, September 2013, pp. 37–42.