Improving MPI Independent Write Performance Using A Two-Stage Write-Behind Buffering Method

Wei-keng Liao[1], Avery Ching[1], Kenin Coloma[1], Alok Choudhary[1], and Mahmut Kandemir[2]

[1]Northwestern University
Dept. of Electrical Engineering and Computer Science
Evanston, IL 60208, USA

[2]Pennsylvania State University
Dept. of Computer Science and Engineering
University Park, PA 16802, USA

Abstract

Many large-scale production applications often have very long executions times and require periodic data checkpoints in order to save the state of the computation for program restart and/or tracing application progress. These write-only operations often dominate the overall application runtime, which makes them a good optimization target. Existing approaches for write-behind data buffering at the MPI I/O level have been proposed, but challenges still exist for addressing system-level I/O issues. We propose a two-stage write-behind buffering scheme for handing checkpoint operations. The first-stage of buffering accumulates write data for better network utilization and the second-stage of buffering enables the alignment for the write requests to the file stripe boundaries. Aligned I/O requests avoid file lock contention that can seriously degrade I/O performance. We present our performance evaluation using BTIO benchmarks on both GPFS and Lustre file systems. With the two-stage buffering, the performance of BTIO through MPI independent I/O is significantly improved and even surpasses that of collective I/O.

1. Introduction

Modern scientific applications often run for a long time and perform data checkpointing to store snapshots of the current computational state. Checkpointing data is used for post-simulation data analysis, such as visualization, and also for restarting the execution in case of failure. Once written, checkpointing data is never accessed again during the application run. Such write-once-never-read patterns have been reported to dominate the overall I/O performance in many large-scale applications. Therefore, designing efficient techniques for such I/O patterns is very important.

Write-behind strategies are a well-known technique by operating system designers as a way to speed up sequential writes [10]. The basic algorithm accumulates multiple small writes into large contiguous file requests in order to better utilize the network bandwidth. The implementation of write-behind is often a part of the client-side caching system, which must consider complicated issues such as cache coherency. Design for coherence control commonly involves the bookkeeping of cache status at the servers and invoking client callbacks when flushing dirty cache is necessary. Such mechanisms require that file locking is part of each read/write request to ensure atomic access to the cache data. Since file locking is usually implemented in a centralized manner, it can easily limit the degree of I/O parallelism for concurrent file operations. However, we believe the write-behind optimization, as an independent component from the caching system, can be used more effectively if the write-only nature is known beforehand.

The Message Passing Interface (MPI) standard [5] defines an application programming interface for developing parallel programs that explicitly uses message passing to perform inter-process communication. MPI version 2 [6] extends the interface to include, among other things, file I/O operations. MPI I/O focuses on the functions for concurrent accessing shared files. Realistically, today there are very few large-scale scientific applications that actually use MPI I/O functions as their primary I/O methods. Instead, the majority of the MPI applications simply uses the POSIX I/O functions (e.g. open, read, write, close) and let each process to access a unique file independently. This "file-per-process" model in a single production run may generate thousands or millions of files which can easily overwhelm users for management. The two possible reasons for choosing the POSIX I/O API over MPI I/O are: 1) accessing shared files concurrently often results in a significant file system overhead; and 2) programming in MPI I/O, especially in collective I/O, is difficult for programmers. For the former reason, the overhead mainly comes from the conflict file locks when the file system is maintaining coherence cache data and enforcing the I/O atomicity. As for the latter, collective I/O usually requires the use of derived data types to define file views for every process, which is not trivial.

We propose a two-stage write-behind buffering scheme for enhancing MPI independent I/O performance. Designed to fully exploit the advantages of write behind, this scheme has two requirements: 1) the I/O patterns must consist of write operations only; and 2) atomic I/O mode is disabled. Data checkpointing in today's scientific applications fulfills these requirements, since there are only non-overlapping write operations. The two stages consist of local and global buffering mechanisms. The first stage accumulates write data into local buffers which are flushed to the global buffer if they are full. The second stage uses global buffering which is based on a static cyclic file domain assignment among the MPI processes, such that data for the same domains are flushed to the same processes. Once the global buffers are full, they are written to the file system. The global buffering enables write alignment with the file system stripe size, which minimizes file lock contention that otherwise can seriously degrade I/O performance. We evaluate the proposed method on two parallel machines running Lustre and IBM GPFS file systems. We used the BTIO benchmark and compare performance between MPI collective I/O and independent I/O functions. With the two-stage write behind, independent I/O can even outperform collective I/O which has been reported in [2] do much better than independent I/O.

2. Background and Related Work

MPI I/O inherits two important MPI features: the ability to define a set of processes for group operations using an MPI communicator and the ability to describe complex memory layouts using MPI derived data types. A communicator specifies the processes that participate in an MPI operation, whether for inter-process communication or I/O requests to a shared file. For file operations, an MPI communicator is required when opening a file to indicate which processes will access the file. In general, there are two types of MPI I/O data access operations: collective I/O and independent (non-collective) I/O. Collective operations require all processes which opened the file to participate. Thanks to the explicit synchronization, many collective I/O implementations take this opportunity to exchange access information among all processes to generate a better overall I/O strategy. An example of this is the two-phase I/O technique proposed in [1]. In contrast, independent I/O does not require synchronization, making any cooperative optimizations very difficult.

Many production MPI applications do not use MPI I/O functions for file access. Instead, I/O is programmed using POSIX interfaces using one file per process. Two possible reasons can be the poor performance for shared-file I/O and difficulty in using MPI I/O for programming. As most modern file systems adhere to the POSIX standard, several POSIX requirements for shared-file I/O are known to degrade the performance. I/O consistency and atomicity are the top two performance problems. The issue of I/O consistency requires cache coherence control for file systems that provide client-side file caching. POSIX atomicity requires that all bytes written by a single write call are either completely visible or completely invisible to any read call [3]. A common solution for the two requirements is file locking. File locking will guarantee exclusive access for the requested file regions. Since file locking is usually implemented in a centralized manner, it severely limits the degree of I/O parallelism for concurrent file operations.

Regarding problems when MPI I/O programming, defining file views for each process using MPI derived data types is often the way to describe the data partitioning of a global data structure among processes. The creation of a data type for setting file view requires information such as the starting file offsets of the sub-array relative to the global array, the accessed file ranges along all dimensions, and the stride size for repeating non-contiguous segments. The construction of a data type can also be nested. Defining a file view is particularly essential for collective I/O operations because it can be used for collaborating processes to improve I/O performance. For independent I/O, the most common programming style is to use explicit file offset for accessing a contiguous file space at a specific location. Although programming in MPI independent I/O is closer to using POSIX I/O, its performance has compared poorly to collective I/O in several studies [1, 11, 15]. We suspect that MPI I/O use in production applications would dramatically increase if performance and programming issues were address.

2.1. Active Buffering

Active buffering is considered an optimization for MPI collective write operations [4]. It accumulates write data into a local buffer and uses an I/O thread to perform write requests in the background. I/O threads can dynamically adjust the size of local buffers based on the available memory space. For each write request, the main thread allocates a buffer, copies the data over, and appends this buffer to a queue. The background I/O thread later retrieves the buffers from the head of the queue, writes the buffered data to the file system, and then releases the buffer space. Active buffering demonstrated a significant performance improvement when it is embedded in ROMIO [13], an I/O library implementation for Message Passing Interface.

2.2. Data Shipping in IBM's MPI I/O

IBM's MPI I/O implementation for GPFS parallel file system adopts a strategy called data shipping that binds each GPFS file block to a unique I/O agent responsible for all the accesses to this block [7, 9]. The file block assignment is done in a round-robin striping scheme. I/O operations must go through the I/O agents which ship the requested data to the appropriate processes. I/O agents are threads residing in each MPI process and are responsible for combining I/O requests in collective operations. In the background, I/O threads also assist in advanced caching techniques such as read-ahead and write-behind. Although data can be cached at the I/O agents, the write-behind strategy is not used to accumulate data locally. Once the I/O request returns, all data must have been shipped to the designated I/O agents.

3. Design and Implementation

The two-stage write-behind buffering proposed in this paper has two requirements: 1) the I/O patterns must only consist of write operations; and 2) the MPI atomic mode must be set to false. The first requirement can be detected at file open by checking the file access mode argument for `MPI_MODE_WRONLY`. The atomic mode can be set through calling `MPI_File_set_atomicity()`. Our implementation is placed at the ADIO layer of ROMIO to catch every write system call. ADIO is an abstract-device interface providing uniform and portable I/O interfaces for parallel I/O libraries [12]. This design preserves the existing optimizations used by ROMIO, such as two-phase I/O and data sieving, which are both implemented above ADIO [13, 14].

3.1. Two-Stage Write Behind

The write-behind scheme consists of two stages. The first stage uses local buffering and the second stage uses global buffering. The write data is first accumulated in the local buffers along with the corresponding file offset and write length. Once the local buffer is full, the data is flushed to the global buffer. A double buffering method is implemented so that one buffer can keep accumulating the write data while the other is used for asynchronous communication. The global buffering is the distributed file pages among the MPI processes that collectively open a shared file. Similar to IBM's data shipping approach, a file is logically divided into equal sized pages, each bound to a single MPI process in a round-robin striping scheme. Thus, page i resides on the process with rank (i mod *nproc*), where *nproc* is the number of processes in the MPI communicator. The

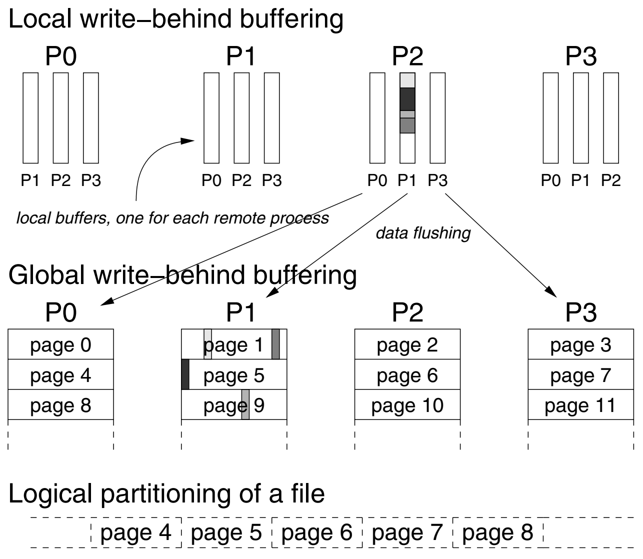


Figure 1. Design of two-stage write-behind buffering.

page size is set by default as the file system stripe size but is changeable through an MPI hint.

The local buffer is further separated into (*nproc* - 1) sub-buffers, each which corresponds to a remote MPI process for data flushing. When accumulated in the local buffer, write data is appended to the sub-buffers based on its destination MPI process. Therefore, data from a single write may be copied to different sub-buffers. For example, when a write spans two pages in the global buffer, it will be separated and appended to two local buffers. The default size for each local sub-buffer is 64 KB, which is also changeable through an MPI hint. When flushing the local buffers to the global buffers, the offset-length information is flushed along with the data. Since the local data accumulation is based on the destination processes, a sub-buffer may contain data spanning across multiple pages on a destination process. The offset-length information helps the receiving process distribute the received data to its proper file pages. The file pages will reside in memory until their eviction is necessary. Figure 1 illustrates the operations of the two-stage write-behind buffering.

3.2. I/O Thread

Since the global buffering is distributed among multiple processes, each process must be able to respond to remote requests for flushing the write-behind data. Because collective I/O is inherently synchronous, remote queries can be fulfilled easily with inter-process communication. The fact that independent I/O is asynchronous makes it difficult for any one process to explicitly handle arbitrary remote requests. Hence, we choose to create an I/O thread that runs concurrently with the main program thread for buffering. To improve the portability, our implementation uses the POSIX thread library [3]. In our design, each process can have multiple files opened, but only one thread is created. It is important to note that every process will create at most one I/O thread even if a process opens multiple files. Our algorithm will create its I/O thread when it opens the its first file. It destroys the I/O thread when the all files are closed. Once the I/O thread is created, it enters an infinite loop to serve both local and remote write requests until it is signaled by the main thread for its termination. All operations related to data buffering are carried out by the I/O thread only. A shared conditional variable protected by a mutual exclusion lock is used to indicate if a write request has been issued by the main thread or if the I/O thread has completed the request. The communication between the two threads is done through a few shared variables which contain information such as the file handler, write offset, write buffer, etc. To serve remote requests, the I/O thread probes for incoming I/O requests from all processes in the MPI communicator group. Since each opened file is associated with a communicator, the probe will scan all the opened files.

3.3. Flushing Policy

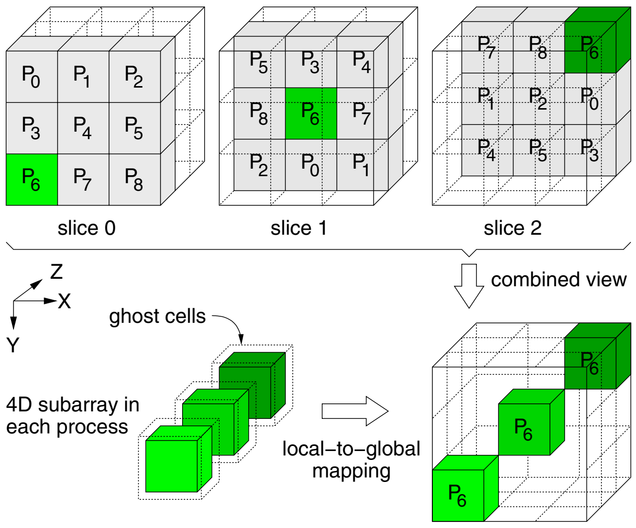
During the first stage, data is flushed when the local sub-buffers are full. During the second stage, file pages are flushed when under memory pressure or at file close. An upper bound, by default 64 MB, is used to indicate the maximum memory size that can be used for the second-stage buffering. For a new page, if the memory allocation utility, `malloc()` finds enough memory to accommodate the page and the total allocated buffer size is below the upper bound, the page will be created. Otherwise, i.e. under memory pressure, page eviction is activated. Eviction is based on the local references and a least-recent-used policy. We keep the dirty data ranges in a linked list for each file page and during the eviction, only dirty data will be written to the file system. When closing a file, because file pages are cyclically distributed across all MPI processes, a two-phase flushing function is devised to shuffle pages such that neighboring pages are moved to the same processes before the flush. Although shuffling requires extra communication cost, this approach enables each process to make consecutive, contiguous I/O accesses and further improves performance.

4. Experimental Results

Our evaluation was done on two machines, Tungsten and Mercury, at the National Center for Supercomputing Applications. Tungsten is a 1280-node Dell Linux cluster where each node contains two Intel 3.2 GHz Xeon processors with a shared 3 GB memory. A Lustre parallel file system version 1.4.4.5 is installed on Tungsten. We created a directory for output files with the configuration of 64 KB stripe size and 8 I/O servers. Mercury is a 887-node IBM Linux cluster where each node contains two Intel 1.3/1.5 GHz Itanium II processors with a shared 4 GB memory. Mercury runs an IBM GPFS parallel file system version 3.1.0 configured in the Network Shared Disk (NSD) server model with 54 I/O servers and 512 KB file block size. Regarding thread-safety, our MPI-IO caching was implemented in the ROMIO layer of the MPICH version 2-1.0.3, the thread-safe and latest version of MPICH2, at the time our experiments were performed. Thread-safety is only supported for the default sock channel of MPICH2. Therefore, although both Tungsten and Mercury have both Myrinet and Gigabit Ethernet installed, for thread-safety reason we can only use Gigabit Ethernet, which is relatively slower than the Myrinet on the same machines.

4.1. BTIO Benchmark

Developed by NASA Advanced Supercomputing Division, the parallel benchmark suite NPB-MPI version 2.4 I/O is formerly known as the BTIO benchmark [16]. BTIO presents a block-tridiagonal partitioning pattern on a three-dimensional array across a square number of processes. Each process is responsible for multiple Cartesian subsets of the entire data set, whose number increases with the square root of the number of processors participating in the computation. Figure 2 illustrates the BTIO partitioning pattern with an example of nine processes. BTIO provides options for four I/O methods: MPI collective I/O, MPI independent I/O, Fortran I/O, and separate-file I/O. According to our experiments and the performance evaluation reported in [2], the independent I/O option results in much worse execution time than collective I/O. Therefore, the case of independent I/O running current ROMIO implementation is not provided. In BTIO, forty arrays are consecutively written to a shared file by appending one after another. We evaluated two I/O sizes: classes B and C with array dimensions of $102 \times 102 \times 102$ and $162 \times 162 \times 162$, respectively. The aggregate write amount for class B is 1.62 GB and 6.49 GB for class C. Table 1 shows the number of `write()` calls generated from the BTIO. Since ROMIO's implementation for MPI collective I/O uses the two-phase I/O strategy that redistributes I/O requests in order to make large contiguous file accesses, the number of write requests for collective I/O is significantly less than the independents. We report the aggregate write bandwidth, because we only use MPI asynchronous functions in our implementation and it is very hard to separate the costs for computation, communication, and



P0 P1 P2 P3 P4 P5 P6 P7 P8
slice 0
P5 P3 P4 P8 P6 P7 P2 P0 P1
slice 1
P7 P8 P6 P1 P2 P0 P4 P5 P3
slice 2
combined view
Z
X
Y
ghost cells
4D subarray in each process
local–to–global mapping
P6
P6
P6

Figure 2. BTIO data partitioning pattern. The 4D subarray in each process is mapped to the global array in a block-tridiagonal fashion. This example uses 9 processes and highlights the mapping for process P_6.

file I/O. The I/O bandwidth numbers were obtained by dividing the aggregate write amount by the time measured from the beginning of file open until after file close. Note that although no explicit `MPI_File_sync()` call is made in BTIO benchmark, closing files will flush all dirty data and is, thus, also included in our performance results.

4.2. Performance Analysis

The performance results are given in Figure 3. Since running independent I/O gives very low I/O bandwidth, we consider it as the worse case. To see how much the two-stage method can improve the independent BTIO, instead of using the naive independent BTIO, we compare our ideas against the performance of naive collective BTIO. As shown in Table 1, collective I/O only generates 40 write requests per process and hence its I/O bandwidth can be considered as a baseline for any new I/O strategy that wishes to further improve the performance. The one-stage write behind is an implementation based on the IBM's data shipping, since the IBM's MPI is not available on both machines. This method is actually the global write-behind buffering part of the two-stage method. As shown in Figure 3, the two-stage write behind outperforms both one-stage buffering and the naive collective I/O in most of the cases.

When looking into the two-phase I/O strategy, one can see that write data is redistributed based on the contiguous file domains assigned among the processes. This strategy is

Table 1. Number of write calls per process generated from BTIO benchmark using collective (coll.) and independent I/O (indep.).

Number of	Class B		Class C	
processes	coll.	indep.	coll.	indep.
16	40	104040	40	262440
25	40	83240	40	209910
36	40	69360	40	174960
49	40	59400	40	150000
64	40	52000	40	131240

similar to the first stage of our two-stage method. One difference in comparison with two-phase I/O is that we use the static file domain assignment in our method. Additionally, the two-phase I/O implementation flushes out data before the function returns. The better network utilization from the write-behind strategy used in our method makes a strong performance improvement.

The cyclic distribution of file pages used in the second stage avoids the communication bottleneck during the data flushing in the first stage. Obviously, the file page size can impact the communication cost of the first-stage data flushing. In fact, choosing a proper page size also affects the flushing cost for the second stage, because it heavily depends on the write performance of the underlying file system. Both Lustre and GPFS are POSIX compliant file systems and therefore respect POSIX I/O atomicity semantics. To guarantee atomicity, file systems often enforce file locking in each read/write call to gain exclusive access to the requesting file region. On parallel file systems like Lustre and GPFS where files are striped across multiple I/O servers, locks can span multiple stripes for large read/write requests. It is known that lock contention due to enforcing atomicity can significantly degrade parallel I/O performance [8]. In some cases, even though the requests do not have overlapping byte ranges, lock contentions can still exist when the lock granularity is the file block and there are overlapping file blocks. This scenario is also known as false sharing. By choosing the file system stripe size as the page size in our two-stage method, all write requests are aligned with the stripe boundaries and hence conflict locks can be avoided entirely in the second-stage flushing. This alignment effect can also be observed by the better write bandwidth from the one-stage write behind over the naive collective I/O.

5. Conclusions

Write-behind data buffering is known to be able to improve I/O performance by combining multiple small writes

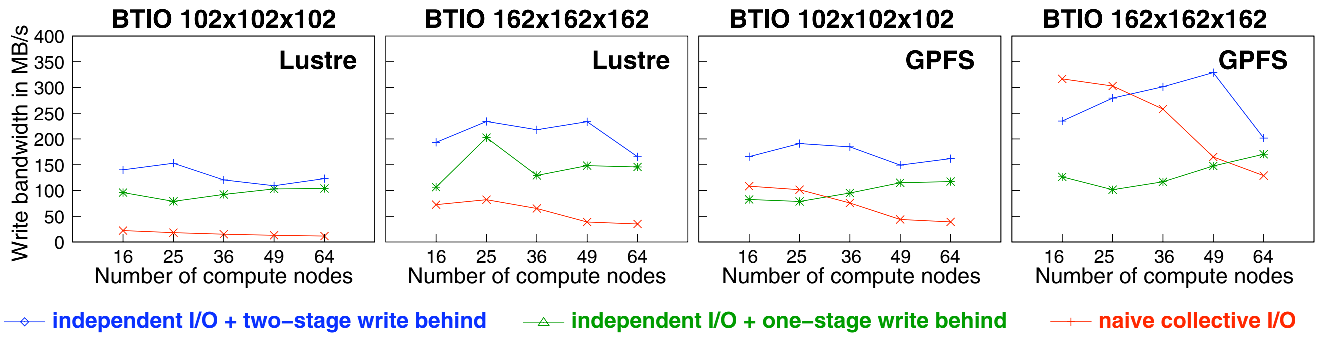


Figure 3. I/O bandwidth results for BTIO benchmark.

into large writes to be executed later. We apply this concept to generate aligned I/O that reduces the lock contention in the file system. For write-only I/O patterns, the performance of MPI independent I/O is significantly improved and even better than collective I/O. In BTIO, the independent I/O functions are called with explicit file offset. There is no derived data type created for setting the file view in comparison to the collective mode of BTIO. Therefore, the two-stage write-behind buffering allows programmers to use the simpler, independent MPI I/O functions, while simultaneously providing performance that meets or exceeds the more difficult collective I/O programming model.

6. Acknowledgments

This work was supported in part by NSF's NGS program under grant CNS-0406341, NSF/DARPA ST-HEC program under grant CCF-0444405, NSF HECURA CCF-0621443 and the National Center for Supercomputing Applications under DDM050006 and CCR060023, and utilized the Dell Xeon Cluster and IBM IA-64 Linux Cluster, and DOE's SciDAC Scientific Data Management Center.

References

[1] 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 '93*, pages 56–70, Apr. 1993.

[2] S. Fineberg, P. Wong, B. Nitzberg, and C. Kuszmaul. PMPIO - A Portable Implementation of MPI-IO. In *the 6th Symposium on the Frontiers of Massively Parallel Computation*, Oct. 1996.

[3] IEEE/ANSI Std. 1003.1. *Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language]*, 1996.

[4] 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 (IPDPS)*, Apr. 2003.

[5] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard, Version 1.1*, June 1995. http://www.mpi-forum.org/docs/docs.html.

[6] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. http://www.mpi-forum.org/docs/docs.html.

[7] J. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS. In *Supercomputing*, Nov. 2001.

[8] R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen. Implementing MPI-IO Atomic Mode Without File System Support. In *the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, May 2005.

[9] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *the Conference on File and Storage Technologies (FAST'02)*, pages 231–244, Jan. 2002.

[10] A. Tanenbaum and M. van Steen. *Distributed Systems - Principles and Paradigms*. Prentice Hall, 2002.

[11] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, 1996.

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

[13] 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, Oct. 1997.

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

[15] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 23–32, May 1999.

[16] P. Wong and R. der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, NASA Ames Research Center, Moffet Field, CA, Jan. 2003.