

# Improving MPI-IO Output Performance with Active Buffering Plus Threads

Xiaosong Ma

Marianne Winslett

Jonghyun Lee

Shengke Yu

Department of Computer Science, University of Illinois at Urbana-Champaign  
{xma1, winslett, jlee17, syu3}@uiuc.edu

## Abstract

*Efficient collective output of intermediate results to secondary storage becomes more and more important for scientific simulations as the gap between processing power/interconnection bandwidth and the I/O system bandwidth enlarges. Dedicated servers can offload I/O from compute processors and shorten the execution time, but it is not always possible or easy for an application to use them. We propose the use of active buffering with threads (ABT) for overlapping I/O with computation efficiently and flexibly without dedicated I/O servers. We show that the implementation of ABT in ROMIO, a popular implementation of MPI-IO, greatly reduces the application-visible cost of ROMIO's collective write calls, and improves an application's overall performance by hiding I/O cost and saving implicit synchronization overhead from collective write operations. Further, ABT is high-level, platform-independent, and transparent to users, giving users the benefit of overlapping I/O with other processing tasks even when the file system or parallel I/O library does not support asynchronous I/O.*

## 1 Introduction

Most scientific simulation codes need to write out large data sets periodically. Typically, these data sets are multi-dimensional arrays holding snapshot data for visualization or checkpoint data for restart. Usually, processors running a simulation enter *computation phases* and *I/O phases* alternately, performing computation in timesteps and writing out intermediate results once a certain number of timesteps has been completed. Efficient transfer of this output from memory to local or remote disks is very important to achieve high performance for such applications.

Often the output data are disjoint subsets of a logically shared data set, encouraging the use of *collective I/O* [1, 3, 7, 9, 11]. In this approach, all the processors cooperate to transfer data between disk and memory. Information about the on-disk and in-memory layouts of the data set is used to plan efficient file operations, and reorganize the data across the memory of the processors if necessary. Most collective I/O techniques block collective I/O participants until file system I/O calls are made, i.e., processors participating in a collective write call do not return from the call until the output data are at least buffered by the file system.

Compared to having each participating processor making its own file system I/O requests, all the collective I/O techniques cited above improve performance by combining smaller, more random file accesses into long, sequential read/write operations, and by overlapping inter-processor communication for data reorganization with I/O. However, these two optimizations alone do not take advantage of some of the common characteristics of modern simulation applications that led us to propose *active buffering*, an aggressive buffering scheme for high performance collective output [5].

In this scheme, extra processors called I/O servers are used to carry out the output tasks for the compute processors on which the application is running. The I/O servers and their clients actively use their idle memory to buffer the output data. After the clients return to computation, the servers write the data out from the buffers. In contrast to previous research on buffering or caching for collective I/O [10], active buffering tries to *hide the cost of writing* by increasing the overlap between I/O and computation, instead of trying to optimize the actual writes. In addition, active buffering has no minimum buffer space requirement and handles overflows gracefully [4, 5].

In our previous research, we used dedicated I/O servers to offload I/O tasks from compute processors and perform write-behind while the compute processors

are computing, which maximizes the performance gain of active buffering. However, dedicated I/O servers are not always available or convenient to use. For example, the global MPI communicator (`MPI_COMM_WORLD`) needs to be split into a computation communicator and an I/O communicator, for the compute processors and I/O processors to communicate within their groups. For a simulation code that uses `MPI_COMM_WORLD`, using an I/O library with dedicated I/O servers requires replacing all the instances of `MPI_COMM_WORLD` with a new sub-communicator, which brings extra labor for the application developers. For another example, scientists who put a great deal of effort into developing a  $2^n$ -processor mesh for a problem will be reluctant to dedicate additional processors to I/O, as doing so will typically require them to remesh their problem or else run in a slower and more expensive  $2^{n+1}$ -processor job queue. Even when additional processors are available and the user application is adjusted to use I/O servers, for accessing data buffered at the compute processors the servers need to use one-sided communication, an MPI-2 feature not yet supported on all platforms.

In this paper, we investigate using *active buffering with threads (ABT)* to hide collective output cost, instead of using dedicated I/O servers. The main contributions of the paper are:

1. We propose ABT, a method of combining background I/O using threads with aggressive and flexible buffering to maximize the overlap between computation/communication and collective I/O in scientific simulation runs. This combination allows parallel simulations to benefit from active buffering without dedicating extra processors as I/O servers or using one-sided communication.
2. Compared to previous studies on performing I/O using background threads, our proposed approach makes no assumptions regarding buffer space availability, with a buffering scheme automatically adjusting to available memory space and applications' output patterns.
3. We implemented ABT in a widely used and supported parallel MPI-IO library, ROMIO [14], in such a way that the buffering and background I/O are transparent to the users and the collective output interfaces remain unchanged. Further, ABT is implemented in the file-system-independent layer of ROMIO, allowing it to be easily ported and to provide extra performance gain on top of optimizations for specific platforms.
4. We evaluated ABT on two popular parallel platforms with micro-benchmarks and realistic appli-

cations. Our results show that ABT can significantly improve both the apparent collective write throughput and the applications' overall performance. As expected, overlapping I/O with other computing tasks can slow down the main thread's processing. However, we found that the total run time is reduced significantly with this overlap. We also found through our experiments that ABT can help ROMIO to *reduce* the actual I/O cost, by reducing the synchronization overhead between the main threads during the data exchange process of collective writes.

The rest of this paper is organized as follows: section 2 discusses background information on active buffering and ROMIO, as well as some related work on using threads in collective I/O. Section 3 presents our approach of combining active buffering and threads. Section 4 shows performance results and analysis. Section 5 compares ABT with asynchronous I/O and section 6 concludes the paper.

## 2 Background

### 2.1 Active buffering

In the active buffering scheme [5], processors actively organize their idle memory into a hierarchy of buffers for periodic output data. The basic idea of active buffering is that output data should be buffered whenever possible. Our previous work on active buffering used dedicated I/O servers to run the server program of the Panda parallel I/O library [11], and compute processors to run the simulation code. In a collective write operation using active buffering, if the compute processors have some idle memory, they will buffer as much data as possible, and only send the overflow to the servers using MPI over the interconnection networks. The servers will do the same thing, and only write data out when there is server-side overflow. The compute processors can return to computation when all the output data are buffered or sent to the servers. While they are computing, the servers use one-sided communication to fetch data from compute processors' buffers and performs actual writing in the background.

Note that this scheme has no hard buffer space requirement and can be used by any applications. With the help of a pair of state machines, it gracefully adapts as buffers at different level of the hierarchy fill and empty, and as new collective I/O requests arrive. Experimental results with synthetic benchmarks and a real rocket simulation code on an SGI Origin 2000, an IBM SP, and a Linux cluster show that active buffering improves the apparent collective write throughput to be within 70% of

the local memory copy bandwidth when sufficient space is available on the compute processors to buffer all the output, and within 90% of the MPI bandwidth for any portion of the output that overflows client buffers and fits into server buffers [5].

## 2.2 The ROMIO library

ROMIO [14], developed at the Argonne National Laboratory, is a well-known implementation of the MPI-IO specification in the MPI-2 standard [6]. As part of the popular MPI implementation MPICH, it supports a wide range of file systems, and has an intermediate interface called the ADIO (Abstract Device Interface for I/O) layer, which is built to hide the implementation details on different underlying file systems for better portability.

MPI-IO supports collective I/O on multi-processor systems by providing an abstract presentation of shared files called the MPI file view, which defines the current set of data visible and accessible from an open file. During collective I/O operations, each processor participating in disk I/O is responsible for accessing a portion of the data in the disk file, and redistributing/gathering data to/from all processors whose I/O requests overlap with this portion of the data. For asynchronous collective I/O, MPI-IO defined *split I/O*, which has a pair of “begin” and “end” calls for collective operations. In the current ROMIO implementation, the *split I/O* interface is supported. However, the implementation does not really enable asynchronous I/O, since the “begin” operation simply invokes the blocking collective I/O calls, and the “end” operation does nothing.

## 2.3 Related work on using threads in collective I/O

Dickens et al. studied the use of threads to overlap I/O and other activities to improve collective I/O performance on four parallel platforms [2]. In this work, a thread is spawned to perform the entire or partial collective write operation in the background as the main thread continues with computation. Their results suggest that performing the whole collective I/O operation in the background, including a communication phase to reorganize data among the processors and a disk I/O phase to perform actual writes, often brings performance degradation, while only overlapping the actual write phase with foreground computation and communication can efficiently hide the write cost on most platforms. The main limitation of the authors’ proposed method of using threads to perform collective writes in the background is that it assumes that there is enough memory space to buffer all the output data for each write request. The authors also reported high thread

switching/scheduling cost. Therefore, creating a separate thread for each write request may cause higher thread overhead for I/O intensive applications where the background I/O threads overlap with each other.

More et al. also studied a multi-threaded implementation of collective I/O in their parallel I/O library called MTIO [8]. The authors’ approach used one single background thread to take over the processing of I/O requests on each compute processor. This thread performs both file I/O and communication, and is created and terminated by a pair of MTIO initialization/shutdown interfaces. Their results on an IBM SP2 show that the MTIO approach can overlap a large part of I/O with computation, and improve the test program’s overall performance. However, the MTIO multi-threaded collective I/O algorithm is quite sensitive to the size of buffer space. It has a requirement on the minimum amount of extra memory available, and uses different I/O strategies according to different buffer sizes (buffer size affects the number of processors that participate in actual file I/O). Their collective I/O results show that performance will degrade seriously when the buffer size grows past the optimal point.

Some asynchronous I/O implementations are based on threads. However, we are not aware of any parallel I/O library supporting non-blocking collective I/O through file system asynchronous I/O interfaces. The difficulty in doing this and a more general comparison between our approach and asynchronous I/O will be presented in section 5.

In addition, IBM’s GPFS parallel file system uses separate threads to perform write-behind from file system buffer pool to disks [12]. This file-system level optimization is platform-specific and uses a preconfigured buffer size.

## 3 Efficient and flexible background I/O with threads and active buffering

### 3.1 Overlapping I/O and other activities with threads

In ABT, we overlap only the file system writes in the collective output operation with the main thread’s computation and communication. As mentioned above, previous work by other researchers has shown that performing data exchange in the background has little or no performance advantage, because data exchange is processor-intensive and the contention between the foreground and background threads may incur high overhead. Further, many current MPI implementations are not thread-safe. The communication between the I/O threads may interfere with the communication between

the main threads, causing undesired results or even crashing the program run.

The intuition behind using I/O threads to enhance an application's overall performance is that it allows the other processing tasks to continue in parallel with file I/O, while in single-threaded blocking I/O the processor is mostly idle waiting for the I/O operation to complete. To hide the I/O cost, one does not want to use *user-level* threads, in which case the whole process will be blocked if one thread gets blocked. In our implementation we used the portable POSIX **pthread**, which can be scheduled at the kernel level on all popular parallel platforms.

Both the previous projects discussed in section 2.3 reported reduced overall run time when background threads are used to perform I/O. Before we set out to implement and test ABT in ROMIO, we verified this by checking the overlap between I/O and other activities using pthreads on our test-beds: the SGI Origin 2000 and a Linux cluster. In these simple experiments, we measured the total latency of a sequence of computation or communication operations, and a sequence of file output operations, executed in serial or in parallel with threads. For computation, we performed iterations of floating point computation on a 32MB 1-D array. For communication, we performed iterations of broadcasting 1MB messages among all the processors. For I/O, we repeatedly wrote to the file system with request sizes ranging from 128KB to 4MB. We varied the number of iterations of these operations, and in all configurations we found the total time of execution in parallel with threads significantly shorter than that of execution in serial, on both platforms. Also, on both of the platforms the multi-threaded execution time appeared to be insensitive to the change of write request size.

### 3.2 Thread synchronization with active buffering

Knowing that I/O can overlap well with computation and communication, we designed an efficient background I/O scheme and applied it to ROMIO. In ABT, we intercept the write requests ROMIO makes to the file system and delay the actual writes with active buffering. The main thread and the I/O thread share a *buffer queue* and act as the producer and consumer respectively. For each write request, the main thread allocates some buffer space, copies the data over, and appends this buffer into the buffer queue, along with other information about the write request, such as its file descriptor (or file name) and write size. The background thread retrieves the buffers and write request information from the head of the queue, issues the requests to the file system, and releases the buffer space.

Since active buffering uses whatever memory that

is available in the system, this becomes the classic *bounded buffer* problem. Both the main thread and the background I/O thread share a variable called ABS (available buffer size), whose initial value can be set by the application code or a default value from the I/O library. The main thread decreases the ABS by the amount of buffer space it allocates, and the background thread increases this number by the amount of buffer space it releases. If there is enough space to buffer all the output data, the main thread can return to computation after all the output data is buffered. If the current ABS value is lower than the amount of memory needed to buffer the next write request, the main thread will be blocked on this condition, until signaled by the background thread when a buffer is written out and freed up. In this case, part of the write cost will become visible to the main thread. On the other hand, the background thread will be blocked if the buffer queue is empty, until signaled by the main thread when a new buffer is appended to the buffer queue.

An appropriate write size is important in ABT for efficient buffering, high buffer space utilization, and optimal parallelism between the main thread and the I/O thread. If write sizes are too small, appending and retrieving tiny buffers to and from the queue will bring high overhead and synchronization cost. If write sizes are too large, output data may not fit into the active buffer space and the main thread can be blocked for a long time, wasting both buffer space and processor cycles. ROMIO combines non-contiguous small requests using the *data sieving* technique [13], issuing larger read/write requests than actually needed by the user code. For reads, it will read in a data sieving buffer and pick out pieces of data requested by the user code in memory. For writes, it will read in the whole data sieving buffer, update it with pieces of user output data, and write the whole buffer out. The current ROMIO data sieving buffer size is 512KB, which is the lower limit of write sizes issued in its collective I/O operations. Therefore we are not worried about small requests. However, ROMIO can issue very large requests. We adopt the default ROMIO buffer size for data exchange, 4MB, as the active buffering write size. ROMIO write requests larger than that will be partitioned and appended to the buffer queue individually. Since our scheme allocates buffer space for each ROMIO write request instead of reusing pre-allocated buffers, there is no internal fragmentation.

Instead of spawning a separate thread to handle each collective I/O operation, we use only one background thread throughout the entire run, as in MTIO [8]. This has several advantages. First, it minimizes the cost of thread scheduling and switching. Second, it minimizes the synchronization cost between the threads over shared objects such as the buffers and the ABS. Third, it ben-

efits the actual I/O performance by maintaining the order of the write requests, already optimized by ROMIO to promote sequential access, while overlapping threads working for different collective write operations and accessing different files may destroy such an I/O pattern, resulting in inferior disk I/O performance.

Figure 1 gives an example of how a single thread handles multiple collective requests. The background I/O thread on each processor is created at the first collective write operation and can start writing once some data arrive in the buffer queue. The I/O thread continues with writing while the main thread goes on with computation. When there is a buffer space shortage (as the collective output size exceeds the total active buffer space, or the computation phase is not long enough for the I/O threads to write out enough data as demonstrated by computation phase 3 in figure 1, or both), the main threads will be blocked and wait for the I/O threads to return enough buffer space. In this case, the main thread is forced to wait during part of the actual writes, as shown in the figure by a prolonged I/O phase 3. Though buffer overflows may occur, we can see that active buffering already maximizes the overlap between I/O and computation automatically, as allowed by the memory space it can use and the application’s I/O pattern. In contrast to MTIO, more buffer space will never hurt ABT’s performance.

In our experiments, we found that the modified ROMIO with ABT may take a shorter time to finish its output than the original ROMIO implementation. This means ABT can reduce I/O cost, in addition to hiding it. This is because ROMIO does data reorganization incrementally, repeatedly cycling between message passing and writing. Since file system writes are often the bottleneck in the system, and there can be high variances between the write latency on different processors, performing writes between message passing steps can incur a higher synchronization overhead. With active buffering, the background I/O operations need not be synchronized between processors.

### 3.3 Incorporating ABT into ROMIO

All of our modifications to ROMIO reside in the ADIO layer of ROMIO, making them file-system-independent and hence available to benefit ROMIO’s performance on all platforms. Further, ABT is orthogonal to other optimizations for ROMIO’s file I/O performance, whether native or external, general or platform-specific. Our implementation approach allows ABT to further enhance ROMIO’s overall performance in addition to such optimizations. In turn, these optimizations can help ABT to reduce the communication cost incurred in collective output calls, or to write the buffered data faster and reduce the amount of overflow I/O.

We mentioned earlier that ABT is transparent to users and there is no need to add user interfaces for ABT. In more detail:

1. Although ABT performs write-behind, it provides users with a blocking I/O semantics: the user output buffer can be safely reused when the main threads return from the collective write call.
2. The user does not need to initiate the background I/O thread. ROMIO can create this thread during the first collective write operation.
3. The MPI file close operation can be modified to work with ABT. In the presence of background I/O threads, the closing of a file should be delayed to when these threads finish accessing this file. In our implementation, we appended empty buffers with special tags to the buffer queue to prompt the I/O thread for this special file operation.
4. The maximum active buffering space can be specified through the MPI-IO collective buffering space hint in the MPI Info object. This hint was originally designed to enable larger and more contiguous file accesses in collective I/O. When ABT is used, ROMIO can not benefit from a large value specified by this hint, since larger requests issued from the original collective write process will be partitioned when put into the buffer queue anyway. Through this hint, the user can also control the active buffering space at run time. Our implementation makes it easy to adapt to changes in the MPI buffer space hint: simply adjust the ABS value at the beginning of each collective write operation, and the active buffer space automatically expands or shrinks to the new size. In particular, if the user specifies the buffer size to be zero, meaning that no extra memory space is available, collective output with ABT will be carried out exactly as in the original ROMIO implementation.
5. The tricky problem is the termination of the I/O thread. The I/O library at runtime knows which collective I/O operation is the first one, where it initiates the background threads. But it cannot tell which one is the last, and the background thread will not terminate itself. It is not safe for the main thread to simply quit the program, as the background thread may still have data in the buffers not delivered to the file system. In our current implementation, the main thread needs to make a special function call to add a special buffer with a “quit” tag to the buffer queue, and wait for the I/O thread to terminate. One better alternative, as pointed out by ROMIO authors at Argonne National Lab, is

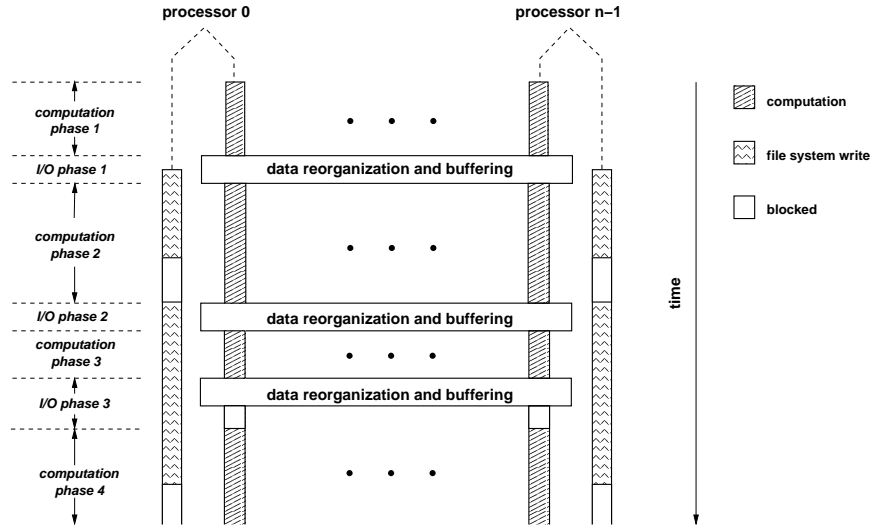


Figure 1. Sample execution timeline of an application run using background I/O threads.

to include ABT finalization into a pre-registered ROMIO call-back function, which will be executed during MPI finalization.

Although our target applications perform many rounds of write-only file access during their computation, we can not exclude the possibility that a simulation reads a file that it wrote collectively. For ROMIO to work correctly with ABT, its collective read operations must also be adapted, which is relatively easy to implement: when ABT has been used for output, ROMIO can check whether the file to read from has data in the buffer queue. If so, ROMIO forces the background threads to write those data out immediately (with some additional communication protocol between the main threads and the background I/O threads), before starting the read operation. Of course this solution loses some of ABT’s performance gain, and a simulation that frequently reads recently-written data may want to turn off ABT by specifying a zero buffer size. The same thing applies to operations such as `MPI_File_sync`.

The execution and interfaces of the remaining MPI-IO operations are not affected by the modifications in ROMIO for ABT. All these operations can be supported by ROMIO’s original implementation while ABT enhances the collective output performance.

## 4 Performance results and analysis

### 4.1 Results from an SGI Origin 2000

The Origin 2000 we used is a distributed-shared memory machine at NCSA with 256 250MHz MIPS

R10000 processors running IRIX 6.5, and 128GB memory in total. The nodes are connected with NUMA-link. Our I/O operations write to the XFS file system on shared RAIDs.

For initial performance evaluation, we used a collective I/O micro-benchmark that comes with the ROMIO distribution, which writes a 3-D array distributed on all processors into one file in row-major order. We modified it to test two common HPF BLOCK style distributions. The first one is a row-major distribution, i.e., `[BLOCK, *, *]`, where the number of partitions in the first dimension is the number of processors. In this case, the distribution of the global array conforms with its distribution in the file, and there is no need for data exchange in collective I/O. The second one is a `[BLOCK, BLOCK, BLOCK]` distribution generated automatically by `MPI_Dims_create`, with the number of partitions in each dimension being `[2, 1, 1]`, `[2, 2, 1]`, `[2, 2, 2]`, `[4, 2, 2]`, `[4, 4, 2]`, and `[8,4,2]` respectively, for 2, 4, 8, 16, 32, and 64 processors. This distribution requires data reorganization in collective I/O. We call the two test cases “conforming” and “non-conforming” respectively. We show the data point with two processors in the “conforming” chart only, because in this case the conforming and non-conforming distributions are same.

To study the impact of active buffering threads on individual collective write operations’ performance, we fix the data size on each processor at 32MB, and increase the size of the global array as the number of processors grows. We enlarge the global array in the way the number of partitions on the dimensions increases. For example, the global array is a  $1024 \times 128 \times 128$  integer array with 2 processors, and a  $1024 \times 256 \times 128$  ar-

ray with 4 processors in the non-conforming test, and a  $2048 \times 128 \times 128$  array with 4 processors in the conforming test. This way, the file size scales with the number of processors, reaching 2GB with 64 processors. We measured the aggregate collective write throughput by dividing the total file size by the maximum latency of the ROMIO `MPI_File_write_all` function call among all the processors. We show the results from our modified version of ROMIO using active buffering threads (denoted as “ABT”), and the original library installed in the system (denoted as “original”). To test the effect of buffer overflow, we used two total active buffer sizes: 64MB and 16MB per processor. The collective write calls are placed with enough “think time” in between for the background I/O threads to finish their writing before receiving the next wave of data. Therefore, buffer overflow will not happen with the 64MB buffer size, while with the 16MB buffer space, half of the data (16MB out of 32MB per processor) will overflow in each collective write call.

Figure 2 shows that for both distributions, original ROMIO’s performance does not scale up well for larger output sizes. Although the output data size remains the same on each processor, it takes longer to write them out when more processors are used. In the non-conforming case, with 8 or fewer processors, data reorganization cost dominates the collective write cost. The fluctuation of throughput in all three lines reflects the change in data exchange cost when the array distribution varies. Particularly, with two processors, ABT’s performance with 16MB buffer space is about 10% lower than that of original ROMIO, because ABT’s buffering and overflow writing cost offsets the gain of hiding the other half of the writing. As the number of processors grows, ABT’s aggregate write throughput scales up very well with 64MB buffer space, reaching about 600MB/s with 64 processors, an improvement of about 500% over the original ROMIO performance and well beyond the peak aggregate file system write throughput we measured on the same system. With 16MB buffer space, ABT’s performance also picks up as the number of processors grows, achieving an improvement of about 35% over the original ROMIO throughput. All the data points include an error bar showing the 95% confidence interval calculated from three or more experiments, which in many cases is not big enough to be visible.

The conforming case incurs no data reorganization costs, except for messages exchanging information about the collective write request. In this case, the throughput of ABT almost doubles as the number of processors doubles, reaching over 2GB/s with 64 processors. Its throughput with buffer overflows also approaches twice the original ROMIO’s throughput, the theoretical bound for throughput when only half of the

data can be buffered.

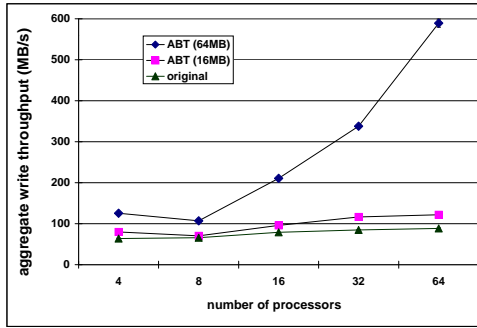
To evaluate the impact of using ABT on a real simulation’s overall performance, we used Jacobi relaxation as the computation module between I/O phases to investigate a real combination of computation and communication<sup>1</sup>. Jacobi relaxation is an iterative algorithm which, in each iteration, updates the values of cells in a grid with the average value of their neighbors from the previous iteration. Using parallel computers, each processor computes on its local distribution of the grid and exchanges boundary values in its local grid with neighboring processors between the computation iterations. Such interleaved floating point matrix computation and boundary value communication is very typical among simulation codes in general. We repeated a subset of the tests described below, using a parallel matrix multiplication code, and obtained similar results.

In our test, we used 16 processors to compute on a  $8192 \times 8192$  double array, distributed in BLOCK style with a  $4 \times 4$  grid to the 16 processors. Again each processor has 32MB of local data and 64MB active buffer space. We ran the program for 256 iterations, taking snapshots at selected frequencies during the computation, using the two versions of ROMIO to write the global array into one 512MB file. We measure the total time spent on the computation phases and show this time as the “computation time” in figure 3(a). The “visible I/O time” is calculated by deducting this computation time from the total execution time.

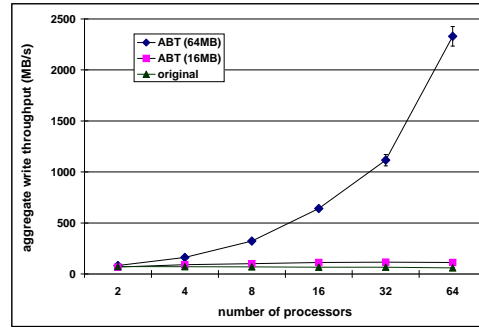
Figure 3(a) shows that ABT reduces the overall run time by as much as 40% in this test. For the first four snapshot frequencies, the I/O time using ABT constantly remains at approximately 23% of that of the native ROMIO. When 64 snapshots are taken (one every four computation iterations), collective I/O is performed so often that the background thread cannot keep up with the request pace, and buffers overflow after a number of snapshots. However, the difference in total execution time between the “ABT” and “original” bars exceeds the total computation time. This indicates that active buffering threads finish writing sooner than the processes using original ROMIO, by avoiding the implicit synchronization between processors, as described in section 3.2. Our Jacobi array has a non-conforming distribution, and we expect the performance gain using ABT to be larger with the conforming distribution, as the visible I/O cost will be even smaller without data reorganization.

As can be seen from figure 3(a), the main thread’s processing was indeed slowed down by the background I/O activities. When snapshots are sparse, most of the time the active buffering thread is blocked, and there is

<sup>1</sup>The rocket simulation mentioned previously currently uses HDF4 format, which, unlike HDF5, does not have parallel interfaces on top of ROMIO.

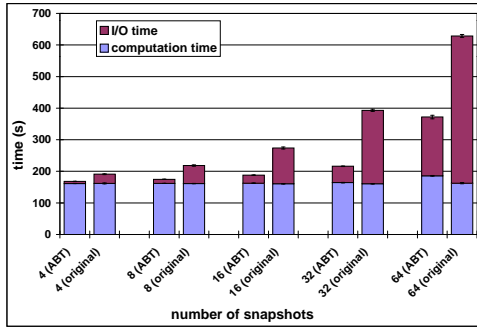


(a) Non-conforming distribution

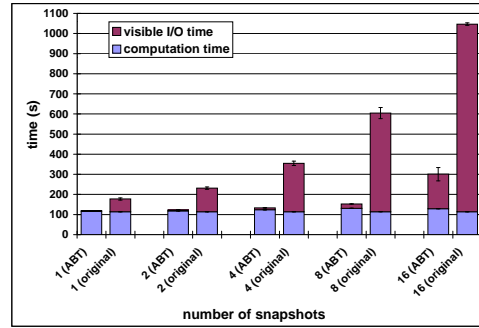


(b) Conforming distribution

**Figure 2. ROMIO collective write performance and scalability on the NCSA Origin 2000.**



(a) NCSA Origin 2000



(b) NCSA Linux cluster

**Figure 3. Execution time of the Jacobi test.**

no obvious slowdown. In the case of 64 snapshots, however, the original total computation time of 160 seconds is slowed down to about 185 seconds.

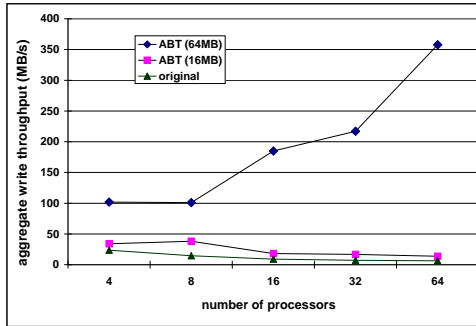
## 4.2 Results from a Linux cluster

The Linux cluster we used is the IA-64 cluster at NCSA named Titan. It has 128 compute nodes running Linux 2.4.16, each with dual Intel 800MHz Itanium processors and 2GB memory. The cluster has 4 NFS mounted shared filesystems through 4 storage nodes, where our experiments wrote the output files. The nodes are connected with Myrinet.

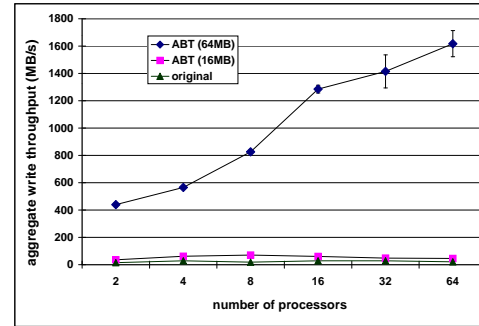
We repeated the experiments described in the previous section on Titan. Compared to the SGI Origin 2000, Titan has faster processors and higher memory bandwidth, but its I/O bandwidth accessing the shared file system is lower and scales poorly. On the Origin 2000,

we found that the aggregate file write throughput levels off after the number of concurrent writers grows past 8. On Titan, this throughput actually decreases visibly after the number of concurrent writers grows past 4. Further, the concurrent writers had a larger variance in write latency on Titan. These facts caused different characteristics in both ABT's and the original ROMIO's performance, shown in figure 4 and figure 3(b).

In the aggregate throughput experiments (figure 4), the original ROMIO's performance slips when the number of processors becomes more than 4, and ABT's performance with 16MB buffer space shows the same trend with 8 or more processors, both due to the degradation of file write performance mentioned above. However, ABT's apparent write throughput is in most cases approximately twice of that of the original ROMIO, even with buffer overflows. With 64MB buffer space, ABT's performance increases with the number of processors.



(a) Non-conforming distribution



(b) Conforming distribution

**Figure 4. ROMIO collective write performance and scalability on the NCSA Linux cluster.**

As in the Origin 2000 test, the non-conforming case with 8 processors is an exception, where the performance gain from additional concurrency in buffering is offset by higher communication cost to reorganize the data. The aggregate collective write throughput of ABT with 64MB buffer space reaches over 350MB/s in the non-conforming test, and 1.6GB/s in the conforming test, in both cases achieving a 2000% plus improvement over the original ROMIO’s performance.

Figure 3(b) shows the results from the Jacobi relaxation test. Because Titan has higher processing power and lower I/O bandwidth than the Origin 2000, we reduced the frequency of output by a factor of four, taking 1, 2, 4, 8 and 16 snapshots respectively during the 256 computation iterations. The results from Titan look similar to those from the Origin 2000, only the performance gain is even larger here. This is because ABT’s ability to avoid synchronization between data exchange steps can make a more significant difference on this platform. As we mentioned before, the Titan processors show a larger variance in file write latency to the shared file system, which is confirmed by measuring the overflow volume on different processors when ABT is used. Such large variance causes long waiting times in the original ROMIO’s collective I/O operation. Like on the Origin 2000, ABT delivers the highest performance gain in the second-to-last test (taking 8 snapshots in figure 3(b)), where the background I/O threads are kept busy without overflow. In this case, ABT reduces the total run time of the simulation by almost 75%.

## 5 ABT vs. asynchronous I/O

ABT and asynchronous I/O can both be used to overlap I/O and computation to increase CPU utilization and

enhance I/O-intensive applications’ performance. However, there are a few key differences between the two.

Unlike asynchronous I/O, ABT imposes a simple blocking I/O interface, and therefore is transparent to application developers. Users can reuse their output buffers provided to a collective I/O call immediately after the call, without checking whether the background I/O has completed.

Second, unlike asynchronous I/O, ABT is platform-independent and implemented at a relatively high level in the system, making it easier to implement, port and use. It provides the performance advantage of overlapping I/O and computation when the underlying file system does not support asynchronous I/O. Also, ABT can be used to write files in scientific data formats that have no or limited asynchronous I/O support (such as HDF/HDF5).

In addition, active buffering is more suitable than asynchronous I/O for optimizing collective I/O performance. Collective I/O operations often involve file I/O operations punctuated by closely synchronized communication to reorganize the data in the processors’ memory to the target layout on disk. Thus it is pointless to simply use asynchronous I/O to perform each disk write, because the output buffer will be immediately needed to perform the next round of data exchange. The output buffer is a user buffer from the file system’s point of view, and can not be reused until the asynchronous I/O request issued on it is completed. Even when double buffering is used, this buffer needs to be refilled quite soon and the issuer of the write request needs to wait for the asynchronous I/O to finish. Therefore straightforward asynchronous I/O can not overlap I/O with computation *outside* the collective I/O call effectively. To get around this problem, a flexible scheme like active buffer-

ing is required.

## 6 Conclusion

In this paper, we proposed ABT: using threads to perform active buffering for background I/O. For simulations that require periodic output and do not require the output data to be forced to disk at the end of each I/O phase, ABT provides a flexible way to utilize idle memory for hiding the I/O cost and overlapping I/O with computation. It provides the performance benefit of asynchronous I/O, but with unchanged MPI-IO blocking collective interfaces, and is platform-independent. We implemented and evaluated ABT using a portable thread library on ROMIO, a widely used implementation of MPI-IO. Our experimental results on two popular platforms with very different underlying shared file system performance show that ABT can significantly reduce the total run time of an application with typical array output requests, by reducing both the visible I/O cost and implicit synchronization cost, while requiring no extra processors or MPI-IO interface changes.

## 7 Acknowledgments

This research is funded by the U.S. Department of Energy under award number DOE DEFC02-01ER25508 and subcontract number B341494<sup>2</sup>, and by a Computational Science and Engineering Fellowship from UIUC. We thank William Gropp, Rajeev Thakur, and Robert Ross at Argonne National Lab for helpful discussions and helping us in building and modifying the ROMIO source library. We also gratefully acknowledge use of the advanced computing resources at NCSA.

## References

- [1] R. Bordawekar, J. Rosario, and A. Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, 1993.
- [2] Phillip Dickens and Rajeev Thakur. Improving collective I/O performance using threads. In *Proceedings of the Joint International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing*, April 1999.
- [3] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, November 1994.
- [4] J. Lee, X. Ma, M. Winslett, and S. Yu. Active buffering plus compressed migration: an integrated solution to parallel simulations' data transport needs. In *Proceedings of the 16th ACM International Conference on Supercomputing*, June 2002.
- [5] X. Ma, M. Winslett, J. Lee, and S. Yu. Faster collective output through active buffering. In *Proceedings of the 2002 International Parallel and Distributed Processing Symposium*, april 2002.
- [6] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Standard*, July 1997.
- [7] J. Moore and M. J. Quinn. Enhancing disk-directed I/O for fine-grained redistribution of file data. *Parallel Computing*, 23(4-5):447-499, June 1997.
- [8] Sachin More, Alok Choudhary, Ian Foster, and Ming Q. Xu. MTIO: a multi-threaded parallel I/O system. In *Proceedings of the Eleventh International Parallel Processing Symposium*, April 1997.
- [9] J. No, S. Park, J. Carretero, A. Choudhary, and P. Chen. Design and implementation of a parallel I/O runtime system for irregular applications. In *Proceedings of the Joint International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing*, March 1998.
- [10] A. Purakayastha, C. S. Ellis, and D. Kotz. ENWRICH: a compute-processor write caching scheme for parallel file systems. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, May 1996.
- [11] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, November 1995.
- [12] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of FAST '02*, January 2002.
- [13] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999.
- [14] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, May 1999.

---

<sup>2</sup>Through the Center for Simulation of Advanced Rockets (CSAR) at UIUC.