

# An Efficient, Nonintrusive, Log-Based I/O Mechanism for Scientific Simulations on Clusters

Soumyadeb Mitra

Rishi Rakesh Sinha

Marianne Winslett

Department of Computer Science, UIUC  
{mitra1, rsinha, winslett}@cs.uiuc.edu

Xiangmin Jiao

Center for Simulation of Advanced Rockets, UIUC  
jjiao@csar.uiuc.edu

## Abstract

*Scientific simulations are often very I/O intensive, requiring high I/O bandwidth to store the data generated by the simulation. Traditional supercomputers have specialized I/O systems with multiple I/O nodes and specialized interconnects to handle such high I/O loads. However, with the increased availability of inexpensive clusters of workstations, more and more simulations are now run on clusters. Unfortunately, cluster supercomputers are usually not very well equipped for I/O, making I/O a serious bottleneck for such applications. To address this problem, we propose Log-Based I/O (LBIO), an approach that can substantially increase the I/O performance of simulations on clusters by utilizing free space on the cluster's local disks to stage data on its way to remote storage. LBIO uses local disks to create a log of all I/O calls, and uses a background thread to replay the log at the rate that best utilizes the server and network resources. LBIO is implemented as an easy-to-use, non-intrusive library—a user can turn on LBIO by adding a single initialization call to the simulation code. LBIO also works with existing scientific I/O libraries like HDF, as well as collective libraries like ROMIO. Our performance studies on microbenchmarks and a real-world scientific simulation code show that LBIO can provide upto 35% improvement in I/O performance for raw I/O and over 50% for I/O through libraries like ROMIO or HDF.*

## 1 Introduction

In the last decade there has been a pronounced increase in the number of high-performance clusters of commodity computers, which offer aggregate processing power comparable to traditional supercomputers at a much lower price.

Although such clusters compete neck and neck with traditional supercomputers in terms of processing capabilities, they usually lag far behind when it comes to I/O. Traditional supercomputers have specialized I/O optimizations involving multiple I/O nodes, special interconnects, and custom built file systems such as GPFS [16]. Clusters, on the other hand, are typically configured with a single network file server, accessible to the rest of the computing nodes over a remote file system such as NFS. Such an I/O configuration becomes a bottleneck for I/O intensive applications like scientific simulations. To make things worse, the throughput of I/O systems is not increasing at the same rate as processor speeds.

Typical simulation codes cycle between computation and I/O phases, computing for a number of time steps in a *computation phase* and then writing out a snapshot of the simulation's current data in an *I/O phase*. The output snapshots are saved on disk for later analysis, visualization, and/or restart. Transfer of snapshots from memory to stable storage is a very frequent and important operation in any simulation and hence needs to be very efficient. Unfortunately most clusters lack the I/O setup to handle this very efficiently. For example, up to 30% of the running time for the Rocstar rocket simulation codes [10] at the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois ([www.csar.uiuc.edu](http://www.csar.uiuc.edu)) is spent waiting for disk I/O to complete, when running on a cluster. In this paper, we propose a novel method that can substantially improve the I/O performance of scientific simulations on clusters by exploiting the special I/O characteristics of these simulation applications. The key contributions of this paper are:

1. We propose *Log Based I/O* (LBIO), a novel way of doing I/O for scientific simulations in which all the I/O calls are logged on the local disks of the compute nodes, and executed afterwards in a separate thread.

Creating a log involves sequential writes to the local disk and is much faster than writing to a remote file system. As a result the I/O time *visible* to the simulation application, i.e., the time that the application spends waiting for I/O to complete, is substantially reduced. The log replaying is carried out in the background, thereby overlapping the I/O with computation. The simulation's execution is not significantly slowed by the background replaying.

2. LBIO has several advantages over other methods of improving I/O performance of simulations. LBIO has minimal memory requirements. LBIO is non-intrusive, in the sense that incorporating it into existing simulations requires virtually no modification to the application code. LBIO works equally well with applications that perform I/O through external libraries, such as ROMIO [18] and HDF [13].
3. Systematic comparisons of LBIO with conventional I/O approaches on microbenchmarks and real simulation applications show that LBIO can significantly improve aggregate I/O throughput and application response time. Further, LBIO's *token-based* log replay approach can be adopted by other I/O libraries to improve their aggregate write throughput.

The remainder of the paper is organized as follows. In section 2.1, we discuss I/O characteristics of scientific simulations. Section 2.2 describe a typical cluster setup, the motivation behind our design, and certain previously proposed I/O optimization techniques. In section 3, we present our design and summarize its advantages over previous methods. We present the experimental setup and results in section 4, and conclude and discuss future work in section 5.

## 2 Background

### 2.1 I/O for Scientific Simulations

The I/O characteristics of simulation applications are quite specific, and are summarized below.

- i) Simulation I/O is write-intensive : each snapshot operation involves writing out in-memory data structures such as arrays and meshes and their associated physics variables, and these write operations are the main component of simulation I/O. Reads are confined to either the start of the simulation run, where the initial state is loaded from data files, or to the metadata reads required during snapshot operations, as explained below<sup>1</sup>.

---

<sup>1</sup>There are out-of-core applications which have substantial read components. Our research does not target those applications.

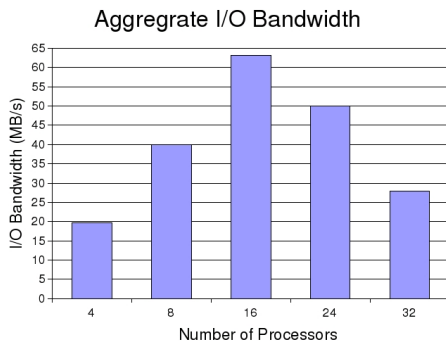
Scientific simulations often write out output through external I/O libraries in formats like HDF. HDF is a portable external file format used for efficient storage of scientific data. A HDF file stores metadata like array dimensions and sizes along with the actual data. Thus during a file write operation (like adding a new array), this metadata is read in, updated and then written back. Since the amount of metadata present is negligible compared to the amount of data, the read is usually negligible compared to the writes.

- ii) Simulation snapshots have two forms. The first option is for each process to dump its output to a separate file. The second option is for all processes to write to (different portions of) the same file. The Rocstar simulation mentioned previously uses the first option, with each process writing out its portion of the mesh structure, along with all the physics variables such as temperature and pressure, into a separate file during each snapshot operation. Codes that employ a large array distributed across all processors typically use the second option to restore the global array to canonical order in a single file.
- iii) Writes do not conflict, i.e, processes do not overwrite the data values written by other processes. Thus writes are idempotent, i.e., the order of carrying them out does not matter.
- iv) I/O is roughly synchronized. Processes enter each I/O phase at approximately the same time. Once they finish their I/O, they typically must exchange some of their current data values with their computational neighbors; thus they cannot proceed with computation until their neighbors have also finished the current I/O phase. As a result, the processes are roughly synchronized. However, the synchronization is not nearly so rigid as would be produced by placing a global synchronization barrier before and after each I/O call. If we can reduce the natural variance in the time that I/O calls take to complete, we may reduce the time spent waiting for other processors to complete I/O., which may translate into a reduction in total run time.

Scientific codes often use collective I/O libraries like ROMIO for efficiently writing output into a shared file. These libraries also introduce implicit synchronization between processes. For example, ROMIO implements a data sieving [17] stage in which processes coordinate to reorganize data across their memory before writing to the disk, if their in-memory data layout is different from the intended on-disk layout. This coordination between processes also causes all the processes to be roughly synchronized.

## 2.2 I/O on Cluster Computers

A typical cluster configuration includes a powerful file-server exporting a file system to compute clients over a remote file system protocol like NFS. The file server either uses its local disks or is connected to an enterprise class storage server such as Apple's Xserve RAID [1] or IBM's DS8000 [2]. The storage disks are typically configured in a RAID architecture for high data reliability and performance. The connection to the remote file system is usually an Ethernet-like network, with a more robust Myrinet interconnect reserved for interprocess communication.



**Figure 1.** The graph shows the aggregate bandwidth on the storage server

Studies have shown that the single NFS fileserver configuration found in typical clusters performs very poorly and attains only a fraction of its peak bandwidth in the face of concurrent writes from multiple clients [9, 12]. We corroborated these previous results with experiments on the CSE-owned cluster described earlier, which has 640 Apple dual processor machines running at 2GHz, with a Myrinet interconnect for MPI and a 100 Mbps ethernet for other network traffic, including the filesystem traffic. We wrote a parallel application involving multiple processes writing simultaneously to the shared NFS file system, with each process writing to a separate file. The total aggregate data written by all the processes was kept constant at 1.5G, which is larger than the cache size on the fileserver. The graph in Figure 1 shows the aggregate I/O bandwidth achieved on the filesystem by the application, as the number of processes varies. The aggregate bandwidth is initially low, achieves its peak at 16 processes and then drops beyond that. This poor scalability is due to poor file system/disk performance for noncontiguous I/O and interconnect/network bottlenecks in the face of multiple connections. We briefly expand on these issues in the following paragraph.

Typical remote file systems like NFS break up large write requests into chunks of smaller units, controlled by

the *wsize* parameter in NFS. In the face of a few hundred clients making simultaneous I/O requests, the storage server is likely to receive and process these requests in an interleaved fashion. Even if each client is making contiguous I/O requests (as usually is the case with simulation writes), the interleaving of requests of multiple clients can make the server perceive the requests as random I/O. Typical disk drives perform very poorly with random I/O because of the large seek and latency costs incurred. The caching implemented in storage/file servers alleviates this problem by buffering and reordering the writes and issuing batch requests to disk. However, the effectiveness of caching very much depends on the amount of data written. Typical big simulations write huge amounts of data per snapshot, often more than the cache sizes in typical storage servers, thereby severely restricting the cache's effectiveness in improving I/O performance by reordering requests. The poor scalability of Ethernet-like networks is another reason for the low I/O throughput during concurrent I/O. Studies have shown that the CSMA/CD protocol used in Ethernet-like networks utilizes only a fraction of the total potential bandwidth when many senders try to connect to a server [5, 6]. As discussed above, implicit and explicit synchronizations between processes may also increase wait times, which can translate into longer total run times.

## 2.3 Related Work

Much research has sought to improve the I/O performance of simulation applications on clusters, from better parallel file system designs like PVFS [8] and Lustre [3] to parallel I/O libraries like ROMIO [18]. One optimization technique that can be employed in almost all parallel file systems and libraries is the use of **background** or **asynchronous** I/O [11, 4, 15]. The underlying principle is to *hide* the I/O time behind computation time. For example, the blocking POSIX **write** call can be replaced by its asynchronous counterpart, **aio.write**. If implemented appropriately, the asynchronous call returns immediately without waiting for the file I/O to complete, while the actual file system write is carried out in the background by the OS. Similarly, in **background** I/O, the foreground simulation code only queues the I/O request, and a separate application thread carries out the file I/O in the background. Since the foreground I/O calls usually return almost immediately, both of these approaches can potentially reduce the I/O time *visible* to the simulation application.

There are, however, limitations to these approaches. Their effectiveness depends on the level at which the facilities are provided and the conditions at run time. For example, the memory buffer on which an asynchronous write has been called may be overwritten only after the I/O completes. Hence the benefit of asynchronous I/O is restricted

by the duration available between a write call and the time the buffers need to be overwritten. Since simulations typically overwrite their variables shortly after each snapshot operation, high-level asynchronous I/O facilities will be of little benefit to them, and lower-level asynchronous I/O facilities will only be helpful if there is free memory to buffer snapshot output. Since the bulk of simulation I/O is performed through external libraries such as netCDF, HDF, and ROMIO, asynchronous file-system-level calls will be of little help to simulations unless their underlying libraries make use of them. Unfortunately, the conversion of blocking I/O calls to their asynchronous counterparts is a significant undertaking [7]. Compiler support can help to automate the process [4], but the performance results are unlikely to be satisfactory in the typical carefully tuned parallel I/O library; a thorough rethinking of the library internals is called for. Background I/O can be provided at the application level, but since its implementation is a significant undertaking [14], all but the most highly motivated simulation writers need background I/O to be provided at the underlying I/O library level.

Finally, asynchronous or background I/O *hides* the I/O cost, without trying to *reduce* it. If we can reduce I/O costs, then there will be less to hide, and the hiding techniques will be more effective. We introduce a tokenizing approach that can increase the effectiveness of LBIO, background I/O, and asynchronous I/O for file systems whose performance does not scale well with the number of concurrent writers.

### 3 Log based I/O

LBIO hides I/O behind computation, but uses a different technique than asynchronous or background I/O. Figure 2 illustrates the overall LBIO system. The user activates LBIO by calling the LBIO initialization routine, passing in a regular expression that LBIO uses to identify the file names used for simulation output. LBIO treats the simulation executable as a black box and externally traps its I/O calls to simulation output files, redirecting them to LBIO's implementation of those calls the exact mechanism used is explained in Appendix A.

Specifically we trap and redirect the following C library I/O calls: *open*, *close*, *write*, *read*, *lseek*, *fcntl*, and their fstream counterparts *fopen*, *fclose*, *fwrite*, *fread*, *fseek* and *ftell*. Within our implementation of those functions, instead of executing the I/O calls on the remote file system, we create a *log* of those calls on the local disk of the compute nodes.

The resulting log contains all the information needed to create the intended snapshot output files. We also maintain an additional data structure for each file being written, which lets us service read requests directly from the log. The log is replayed technically, the log is being played

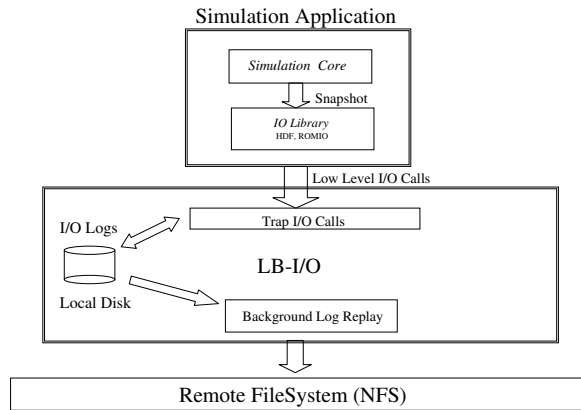


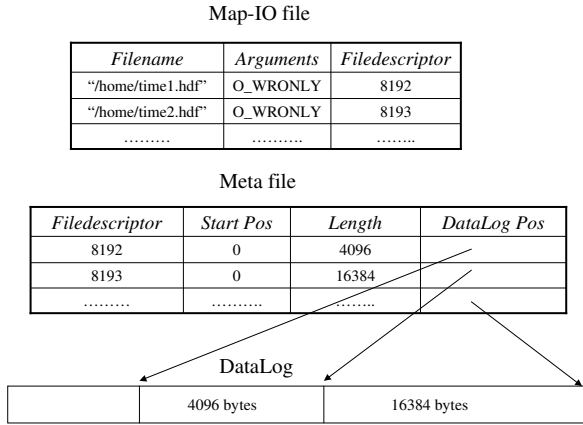
Figure 2. LBIO system architecture

for the first time, but the use of the word "replay" is traditional when discussing logs. In a separate background thread within the LBIO library to create the intended output files.

Our LBIO implementation maintains three separate log files: *map-log*, for maintaining the mapping between file names and integer descriptors; *meta-log*, for storing *write* metadata; and *data-log*, for the actual output data. An example file organization is shown in Figure 3.

- On an *open* or *fopen* call, a unique fileID is generated and appended to the *map-log* log, along with the file name, flags, and file mode arguments to the *open* call. The fileID is returned to the application. LBIO-generated fileIDs are greater than the number of simultaneously open files allowed by the underlying file system, so that they do not conflict with the descriptors for its other open files. For new files, an additional auxiliary data structure is also created, as described below. For each opened file, LBIO maintains additional metadata, such as the current file position and size.
- On a *write* or *fwrite* call, the metadata (including the file descriptor, the length of the data being written, the position within the file where the data is to be written, and the current position in the *data-log*) is appended to the *meta-log*. The actual data are then appended to the *data-log*.

The LBIO implementation also maintains an interval tree data structure for each snapshot output file, to record the mapping between intervals within a file and the corresponding data positions in the *data-log*. The interval tree is a B+ tree on start positions of the intervals (the length of each interval is also stored with the interval start positions). On each write call, the interval



**Figure 3. An example of what the various log files look like.**

tree is updated to include a mapping between the output file interval being written and the current position of the data for that interval in the data-log.

- For output data that has not yet been sent to the remote file system, *read* and *fread* calls are directly answered from the log. The interval tree is traversed to find all the intervals overlapping with the read, and the corresponding data is read from the *data-log*. If part of the requested data has already been sent to the remote file system, or has never been written by this process, then the LBIO implementation requests that data from the remote file system and combines it with whatever relevant data was still resident in the log. A read from the logs runs in time  $O((\log(n)) + k)$  worst case, where  $n$  is the number of intervals in the tree for that file, and  $k$  is the number of intervals spanned by the *read*.

Typical simulations execute large writes, so we expect  $n$  to be reasonably small. Furthermore, we expect reads to be confined to metadata that have been written by previous write calls, as is done in scientific I/O libraries. Hence we expect  $k$  to be 1 or 2.

The LBIO implementation uses a log cleaner process to remove entries from the log files that have been replayed. Cleaning up the log also requires appropriately modifying the file interval trees, so that any subsequent read requests are answered from the remote file system rather than from the log. We have implemented a *lazy* reclaim policy, in which log space is freed only if the log becomes larger than a threshold value that the user can adjust during in the LBIO initialization call.

The size of LBIO’s map-log file depends on the num-

ber of files opened during each simulation snapshot. Most simulations create a small number of files per process per snapshot, e.g. one file per scientific variable to be saved, or one file for all the variables. Hence the map-log file is usually very small. The meta-log file stores 16 bytes of meta-data for every write call. Simulations usually execute large write calls for dumping their arrays or meshes, making the overhead of these log files very small in comparison to the data-log that stores the actual write data.

LBIO stores each file’s interval tree data structure in main memory. The memory requirement of the interval tree structure is  $O(n)$ , where  $n$  is the number of intervals in the tree. The size of  $n$  depends on the number of disjoint file intervals on which *write* has been called. Most simulations execute a relatively small number of write calls, each of which writes out a relatively large amount of data. Consequently, the memory requirements for interval trees are very small.

### 3.1 Log Replay

Log replay involves reading metadata records from the meta-log file and executing the corresponding write calls on the remote file system. The frequency of the replaying is controlled via a parameter that the user can adjust at initialization time. The background thread waits until the total amount of logged data crosses that threshold, and then initiates replay.

Since the actual file system I/O is carried out by LBIO’s I/O thread, LBIO has flexibility to implement additional semantics-preserving optimizations as it carries out the I/O. The most important optimization that we implemented is *token-based* replaying. As observed above, the single file system/storage server configuration found in typical clusters may perform very poorly in the face of concurrent I/O requests from many clients. We have implemented a token based approach where each replay thread waits to receive a token before it replays a portion of its log. The number of tokens given out at any time is limited to the number of simultaneous I/O requests that maximizes the aggregate throughput of the underlying file system (for example, 16 tokens in Figure 1). This number is predetermined by running a parallel script that performs concurrent writes to the file system with varying numbers of concurrent requesters. Upon receiving a token, each processor replays its log, up to a predetermined threshold number of bytes an alternative possibility is to let each process complete all its logged I/O before passing on the token.. We implemented the token scheme by having a separate token server that runs on a front-end machine of the cluster. The background log threads communicate with the token server over the network socket interface. A decentralized implementation of tokens would also be possible.

A related optimization called *grouped I/O* was proposed for early parallel file systems, such as CFS. Grouped I/O also addresses the problem of file system performance that does not scale well with increasing numbers of concurrent writers. Under grouped I/O, predetermined groups of processors take turns in writing their data to disk. Grouped I/O explicitly synchronizes the beginning and end of each group's activities, which causes a cascading impact if one processor in each group is delayed in completing I/O, as commonly occurs in clusters. Grouped I/O also implicitly assumes that each processor is writing the same amount of data, which is no longer necessarily the case in today's simulations. Token-based I/O is designed to provide the benefits of grouped I/O without these drawbacks.

With LBIO, one can also implement optimizations that look at the log of I/O calls requested by the application and apply semantics-preserving transformations to reduce I/O costs. One optimization that we implemented in the log replayer was to merge multiple small, but contiguous, write I/O requests into larger requests. This is particularly effective for some of the scientific I/O libraries like HDF which do a number of small metadata writes. Our experiments show that such small metadata writes perform very poorly on remote filesystems. By merging these small write calls into larger calls, one can substantially improve the performance of I/O.

The other optimization that we implemented was to get rid of multiple open-close calls that applications like Rocstar perform on the same snapshot file. In Rocstar, each module of the simulation (e.g., solids, fluids, crack propagation) separately writes out its data by opening the snapshot HDF file, creating a separate dataset, and then closing the file. These multiple open-close operations are very costly on NFS, because the NFS file-close cache synchronization policy flushes all client cache updates to the server's filesystem before returning. With LBIO, we get rid of these intermediate open-close calls, thereby letting users maintain tidy modular of their simulation I/O, without suffering the performance penalty of multiple open-closes.

### 3.2 LBIO Correctness Guarantees

For semantic correctness of the result of replaying log entries, LBIO relies on the properties of simulation I/O that were described earlier.

- **Write-write conflicts.** LBIO guarantees that the effect of replaying an individual process's log will be the same as if the process's log entries are replayed in their original FIFO order. LBIO does *not* guarantee a particular replay order across multiple processes. Here LBIO is relying on the characteristic of in-core simulations that no two processes will write different data

to the same location of a snapshot file. A single process can overwrite its own data, due to LBIO's FIFO correctness guarantee.

It is fairly obvious that simulation codes do not have conflicting writes on non metadata data from snapshot output operations. (today's application writers seem to prefer to have their snapshot output data double as application checkpoints. However, if an application writer wants to create checkpoint files and periodically overwrite them, then that is fine; those files should not be included in the LBIO initialization parameters.) However, processes often overwrite their own metadata. For example, the HDF library overwrites the old file metadata with new metadata as datasets are added to a file. If processes dump their snapshots to separate files, then each process overwrites separate metadata, so the overwriting is not a problem. If multiple processes write to the same file, the metadata reads and writes are normally done by a single process (often the process with rank 0), because all the processes must agree on the metadata values in the file. Single-process metadata management is compatible with LBIO, as explained in the next paragraph.

- **Read-write conflicts.** The data written by a process is not available to be read by another process until LBIO has replayed the relevant portion of the log. Here LBIO is relying on the characteristic that in-core simulations do not reread the data that they write out. In-core simulations do reread the metadata that they write out, but as explained in the previous paragraph, the metadata management facilities used in modern scientific I/O libraries localize metadata management to a single process. Within a single process, LBIO guarantees that a read of a previously written file region will see the most recent values written to that file region. The most recent (meta)data values are obtained from the local log, or fetched from remote disk if they are not available locally. For metadata reads, all processes rely on the designated metadata management process for that file; since the metadata management process is guaranteed to see correct metadata values, all other processors will also see correct metadata values.
- Our current LBIO implementation does not support advisory locking calls (`fcntl` and `flock`) or file attribute calls (`fstat`). We also do not support file modification through `mmap/munmap`. The LBIO implementation can easily be extended to handle `fstat`, `mmap`, and `munmap`. Support for `fcntl` and `flock` would require LBIO design changes, and is an interesting topic for future work.

## 4 Performance Evaluation

We evaluated LBIO performance on a microbenchmark and on a real simulation application, the Rocstar simulation code described below. The experiments with Rocstar provide a real-world reality check for a particular application, and the microbenchmarks allow us to twiddle application parameters to observe the effect of LBIO on a wider range of applications than are represented by Rocstar alone. All the experiments were conducted on the CSE cluster described earlier. The cluster has a single Apple XRAID Storage server with 7TB of storage space, connected to a Linux NFS server over a 2Gbps Fibre Channel Interface. The Linux server exports the volume over NFS to the rest of the cluster (consisting of 640 dual processes Apple Xserver machines) over a 100 Mbps ethernet link.

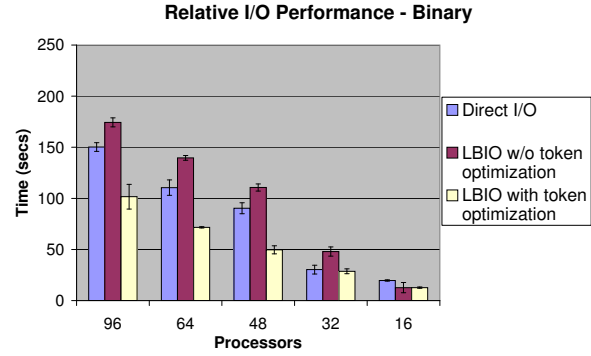
All the performance numbers were obtained by taking 5 runs and taking the average. The error bars show 95% confidence interval for the mean.

### 4.1 Microbenchmarks

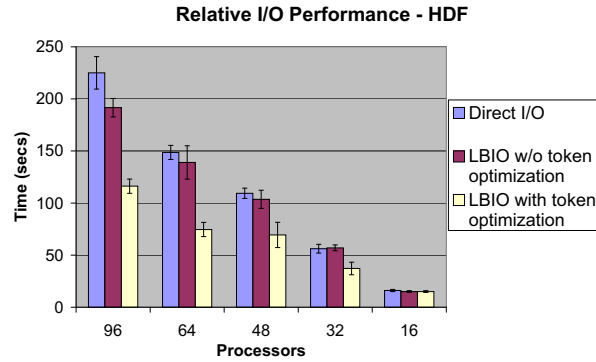
The first set of experiments demonstrate the I/O performance improvement achieved by LBIO. The microbenchmark for this evaluation is an MPI-based parallel application with each process executing a series of I/O phases. Since the focus of this study was to measure I/O throughput, we omitted the computation phase, thereby simulating the extreme case of an I/O-bound simulation. In each I/O phase, each process wrote out 10, 256Kbytes arrays to snapshot files, with each process writing to a separate file. The number of I/O phases was 10 for all the experiments. In order to simulate the dependency between processes, we also added a dummy data exchange step in which each process exchanged an integer value with its four neighbors.

Figure 4(a) shows the total I/O time measured at process 0, as the time between MPI.Barrier's at the start and at the end of the application/ for the microbenchmark, comparing direct I/O (raw I/O in which each process wrote directly to the remote file system); LBIO, without the token based replaying optimization; and LBIO, with token based replaying. In this experiment, there was no compute phase and hence nothing to hide I/O behind. Moreover individual write calls were fairly large and contiguous. As a result, LBIO without the token based optimization does not offer any advantage over direct I/O. Hence, expectedly, it performs slower than direct I/O because of logging and replaying overhead. However the slowdown was only about 20%.

On the other hand, LBIO with the token based group replaying performs substantially better than direct I/O. Replaying with a limited number of concurrent writers reduces the I/O contention on the file server, thereby improving its



(a) Binary Files



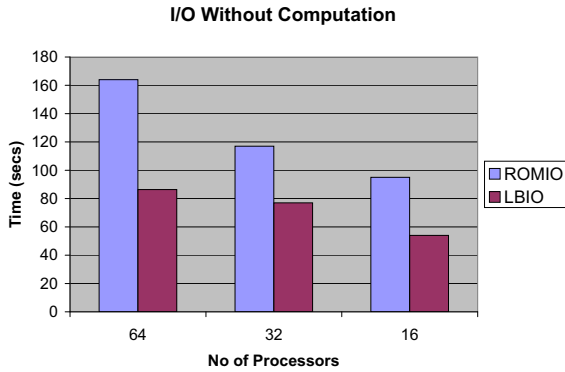
(b) HDF Files

**Figure 4. Comparison between direct I/O, LBIO without token optimization, and LBIO with 16 tokens**

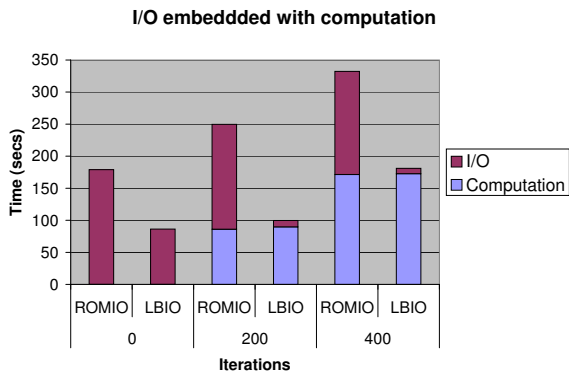
throughput. As expected, the speedup is higher for higher numbers of concurrent writers. The group size in these experiments was set to 16, the value for which the filesystem achieved peak bandwidth (Figure 1).

We repeated the same experiment with each process writing HDF files using the HDF4 I/O library, instead of raw I/O. In this case (as shown in Figure 4(b)), LBIO without token based replaying performs slightly better than direct I/O. LBIO with token based replaying gives almost 2 times speedup. HDF executes some small writes for metadata. LBIO merges small writes into a big contiguous writes, which usually gives better performance. As a result, LBIO outperforms direct I/O, even without the token based optimization.

The next set of experiments uses LBIO for all subsequent



**Figure 5. Comparison of ROMIO and LBIO without computation**



**Figure 6. Comparison of ROMIO and LBIO with emedded computation**

experiments, we use LBIO with token based replaying with the default configuration of the ROMIO implementation of the MPI-IO standard for parallel I/O. The application has a 64M two dimensional array, distributed in a (BLOCK, BLOCK) manner across the processors. The microbenchmark executed a sequence of 40 consecutive I/O phases, with no intervening computational phases. In each I/O phase, the processors used collective ROMIO I/O calls to reorganize the array into canonical order and write it to the remote file system. LBIO trapped and logged the file system calls executed by ROMIO. We measured the total execution time at process 0, as the time between MPI\_Barrier's at the start and at the end of the application

As illustrated in Figure 4.1, ROMIO can perform substantially better when LBIO is used. This is because ROMIO's internal facilities for array reorganization introduce implicit synchronizations between pairs of processes as the processes work to reorganize a portion of the array, output that portion, and then start to reorganize the next por-

tion. If one processor takes longer than the others for a particular file system call to finish, then the implicit synchronizations have a ripple effect that can slow down the whole computation [15]. LBIO overcomes this problem because the foreground I/O cost visible to ROMIO is only the data-logging time, which usually has a much lower variance than for writes to a remote file system. Processors do not need to communicate among themselves while replaying their logs, so delays in replaying the log do not have a ripple effect.

The microbenchmarks described so far have no compute phases, hence no way to hide I/O costs. The next set of microbenchmark experiments uses the same ROMIO benchmark as above, but with a computation phase between consecutive I/O phases. The computation phase involves iterating through loop that accesses the data in a local array. Figure 4.1 compares the I/O and compute times for ROMIO with and without LBIO, for different length compute phases (controlled by the number of iterations of the loop). At 200 iterations, the compute phase duration is about of the same length as the I/O time for LBIO. As evident from the figure, LBIO almost overlaps the entire I/O behind computation. At 400 iterations, computation becomes the dominating component. Again, LBIO hides almost the entire I/O behind computation. Also the computation time for the main application remains almost unaffected by the background I/O thread of LBIO. This experiment shows that LBIO can be very effective in hiding I/O behind computation.

## 4.2 Large Scale Simulations

The next set of experiments demonstrates the applicability and utility of LBIO in a real simulation application, the Rocstar multicomponent rocket simulation developed as a part of an ongoing ten year project at the Center for Simulation of Advanced Rockets at the University of Illinois. The objective of the Center is to construct a detailed, whole-system simulation of solid rocket motors, such as the Space Shuttle Reusable Solid Rocket Motor (RSRM), operating under both normal and abnormal operating conditions [10]. In Rocstar, the simulation object (rocket) is pre-partitioned into a large number of mesh blocks and each processor is assigned a number of such blocks. Along with the mesh structure, each block also has a number of additional physics variables, such as temperature and pressure. Rocstar performs periodic snapshots, where each processor separately writes its mesh structure and its variables to separate HDF files. The user controls the frequency of snapshots through a parameter setting. The snapshots are used as input to the ROCKETEER visualization program; frequent snapshots result in a high quality video. Rocstar runs that use the native HDF output interface spend a substantial fraction of their total run time waiting for I/O to complete.

Table 1 shows the performance results from test runs of

	I/O Time	Computation Time	Total Time
Normal	226.48	606.93	833.41
With LBIO	15.73	609.19	624.92

**Table 1. Rocstar simulation performance on 60 processors, with and without LBIO**

Rocstar with and without LBIO, for a simulation run of the 60 processor Inhibitor rocket dataset incorporating LBIO just required adding one initialization call to the simulation code. The simulation was carried out for 10 timesteps, with a snapshot being output after timesteps 2, 4, 6 and 8. The total data written per snapshot operation (and hence the output time) was almost constant across different timesteps, at 800M and 57 seconds.

As evident from the table, LBIO hides more than 90% of the I/O time for Rocstar, thereby substantially improving its total running time. Furthermore, the background log replayer thread had almost negligible impact on the foreground simulation (compute time went up less than 1%). This experiment demonstrates that LBIO can be a very effective tool for reducing I/O time for real simulation application.

## 5 Discussion, Conclusions, and Future Work

Other researchers have observed that background I/O and asynchronous I/O have the potential to provide significant improvements in the total run time of simulations, by reducing the I/O costs that are visible to applications. LBIO takes advantage of special characteristics of in-core simulations to reduce visible I/O costs while preserving I/O correctness guarantees. LBIO treats the simulation code as a black box, externally traps all its I/O calls, writes them to a log, and then replays the log in the background once the application’s computation has resumed. LBIO uses small auxiliary index structures to handle read requests on recently written data.

LBIO’s execution of *write* calls only involves appending the call arguments to the log file on local disk, using large sequential writes. Our experiments show that this operation can be very fast, thereby substantially reducing the I/O time visible to the application. Our experiments show that LBIO has a very low computational overhead and performs as well as asynchronous or background I/O, both in microbenchmarks and with a real-world rocket simulation code.

LBIO is non-intrusive and transparent to users. Incorporating LBIO into existing simulation codes only requires the addition of a call in the simulation code to initialize the LBIO library. The same initialization call works whether

the simulation directly accesses the remote file system or uses I/O via libraries such as HDF or ROMIO.

Previous work on background I/O relies on the presence of idle memory to buffer snapshot output; its effectiveness is reduced if little memory is available. LBIO provides an additional level for staging data as it moves between memory and remote disk, and can be coupled with techniques that take advantage of idle memory when it is available, to provide even greater reductions in visible I/O time than would be possible with either technique alone. LBIO’s logs rely on the presence of unused local disk space, which is usually in plentiful supply. A separate log cleaner incrementally frees the replayed portion of the log. The presence of a log also provides a natural attachment point for I/O optimizations aimed at reducing the total I/O time.

Instead of having each background thread replay the log independently, we have devised a token-based mechanism in which the log replayer waits to receive a token from a token server before replaying a portion of its log. By controlling the number of tokens, we limit the number of threads simultaneously executing I/O calls on the remote file system. Our experiments show that the token approach also offers significant performance benefits for file systems that do not scale well as the number of concurrent writers increases. The token approach can be incorporated into I/O libraries that do not use LBIO.

For future work, we intend to explore ways to apply AI learning techniques to self-tune the number of tokens during the course of the simulation, based on current file system performance. We will also examine additional optimization opportunities that take advantage of the log to provide a broader perspective on I/O activities. We plan to extend the LBIO implementation to support flock and fcntl calls, which are helpful for storage libraries like HDF and netCDF. Our solution approach may make use of a separate metadata manager process to keep track of all the metadata for the logged I/O. The central metadata manager can be used by processes to look up and read file contents directly from other processes’ logs, and can be used to implement file locking. Careful optimization (and elimination) of calls to the metadata manager can be used to ensure that the manager does not become a bottleneck for I/O.

## References

- [1] Apple Xserve RAID, [www.apple.com/xserve/raid/](http://www.apple.com/xserve/raid/).
- [2] IBM DS8000, [www-1.ibm.com/servers/storage/disk/ds8000/index.html](http://www-1.ibm.com/servers/storage/disk/ds8000/index.html).
- [3] Lustre filesystem, [www.lustre.org](http://www.lustre.org).
- [4] G. Agrawal, A. Acharya, and J. Saltz. An interprocedural framework for placement of asynchronous I/O

operations. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 358–365, Philadelphia, PA, May 1996. ACM Press.

- [5] G. T. Almes and E. D. Lazowska. The behavior of ethernet-like computer communications networks. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 66–81, 1979.
- [6] Andr B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the second international workshop on Software and performance*, pages 195–203, 2000.
- [7] Bradley Broom, Rob Fowler, and Ken Kennedy. KelpIO: A telescope-ready domain-specific I/O library for irregular block-structured applications. In *Proceedings of the 2001 IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, 2001.
- [8] P. Carns, W. Ligon III, R. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.
- [9] Dell Report [www.dell.com/downloads/global/power/ps4q04-20040145-Jancic.pdf](http://www.dell.com/downloads/global/power/ps4q04-20040145-Jancic.pdf). *Maximizing NFS Scalability*.
- [10] W. Dick and M. Heath. Whole system simulation of solid propellant rockets. In *Proceedings of the 38th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit*, Indianapolis, IN, July 2002.
- [11] 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*, pages 38–45, April 1999.
- [12] Hal L Stern and Brian L. Wong. NFS performance and Network Loading. In *LISA VI*, Long Beach CA, 1992.
- [13] <http://hdf.ncsa.uiuc.edu/UG41r3.html/>. *HDF 4.1r3 User's Guide*.
- [14] X. Ma, J. Jiao, M. Campbell, and M. Winslett. Flexible and efficient parallel i/o for large-scale multi-component simulations. In *Proceedings of The 4th Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications, in conjunction with the 2003 International Parallel and Distributed Processing Symposium*, April 2003.
- [15] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPI-IO Output Performance with Active Buffering

Plus Threads. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

- [16] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies*, Monterey, CA, January 2002.
- [17] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, February 1999.
- [18] Rajeev Thakur, William Gropp, and Ewing Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 23–32, May 1999.

## A Trapping I/O

There are several possible ways to trap C library I/O calls and redirect them. (We focus on C calls, because I/O libraries usually issue C I/O calls.) The simplest option is to redefine the function calls in a separate library and put the library in the link path. Linkers normally resolve a function reference so that it refers to the first implementation of the function that the linker finds in the link path. For dynamic linking, one can use the LD.PRELOAD.FLAG (or its equivalent on a particular platform) to ensure that the LBIO library is loaded before the C stdio library. If relinking the application is impractical (e.g., only the code's executable is available to users), code injection can also be used to redirect I/O calls. A good overview of these and other redirection techniques can be found at [http://rentzsch.com/mach\\_inject](http://rentzsch.com/mach_inject).