IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications
6-8 September 2007, Dortmund, Germany

Using Object Based Files for High Performance Parallel I/O*

Jeremy Logan [1], Phillip M. Dickens [2]
1) Department of Computer Science, University of Maine, jeremy.logan@maine.edu
2) Department of Computer Science, University of Maine, dickens@umcs.maine.edu

Abstract – *We contend that the scalable I/O problem in high performance computing is largely due to the legacy view of a file as a linear sequence of bytes. In this paper we introduce an alternative to the traditional "flat file" that uses the information contained in file views to partition a file into an optimal set of objects, minimizing locking contention and simplifying the lock management strategy. We illustrate the use of an object based cache added to ROMIO to efficiently and transparently add object-based file capabilities to MPI-IO. We analyze the performance of our system using the FLASH-IO benchmark, and demonstrate a substantial performance improvement over the standard ROMIO implementation.*

Keywords – *parallel I/O, MPI-IO, object-based files, object-based cache, I/O Performance, file view*

I. INTRODUCTION

Large-scale computing clusters are increasingly being used to execute large-scale, data-intensive applications in several disciplines including, for example, high-resolution simulation of natural phenomenon, large-scale climate modeling, earthquake modeling, visualization/animation of scientific data, and distributed collaboration. The execution of such applications is supported by state-of-the-art file systems (e.g., Lustre [1], GPFS [2], Panasas [3]) that provide tremendous aggregate storage capacity, and by parallel I/O interfaces that can interact with such file systems to optimize access to the underlying store. The most widely used parallel I/O interface is MPI-IO [4], which provides to the application a rich API that can be used to express complex I/O access patterns, and which provides to the underlying implementation many opportunities for important I/O optimizations. The problem, however, is that even with all of this hardware and software support, the I/O requirements of data-intensive applications are still straining the I/O capabilities of even the largest, most powerful file systems in use today. Thus new approaches are needed to support the execution of current and next-generation data-intensive applications.

There are many factors that make this problem, generally termed the *scalable I/O problem*, so challenging. The most often cited difficulties include the I/O access patterns exhibited by scientific applications (e.g., non-contiguous I/O [5-7]) poor file system support for parallel I/O optimizations [8], strict file consistency semantics [9], and the latency of accessing I/O devices across a network. However, we believe that a more fundamental problem, whose solution would help alleviate all of these challenges, is the legacy view of a file as a linear sequence of bytes. The problem is that application processes rarely access data in a way that matches this file model, and a large component of the scalability problem is the cost of translating between the process data model and the file data model. In fact, the data model used by applications is more accurately defined as an *object model*, where each process maintains a collection of (perhaps) unrelated objects. We believe that aligning these different data models will significantly enhance the performance of parallel I/O for large-scale, data-intensive applications.

This research is attacking the scalable I/O problem by developing the infrastructure to merge the power and flexibility of the MPI-IO parallel I/O interface with a more powerful *object-based* file model. Toward this end, we are developing an *object-based caching system* that serves as an interface between MPI applications and object-based files. The object-based cache is based on MPI file views [10], or, more precisely, the intersections of such views. These intersections, which we term *objects*, identify all of the file regions within which conflicting accesses are possible and (by extension) those regions for which there can be no conflicts (termed *shared-objects* and *private-objects* respectively). This information can be used by the runtime system to significantly increase the parallelism of file accesses and decrease the cost of enforcing strict file consistency semantics and global cache coherence.

We demonstrate the benefits of this research using FLASH-IO: a widely-used benchmark that specifically addresses the issue of checkpointing in large, data-intensive scientific codes. Checkpointing is becoming an increasingly significant cost for data-intensive simulations as the size of the computing systems upon which they execute, and the time period for which they are executing, continues to increase. An important consideration for implementing checkpointing is that while checkpoint files are written frequently, they tend to be read infrequently. This is a very significant observation, since it suggests that the checkpoint write operation should be highly optimized, even if the optimization results in a more expensive checkpoint read operation.

The primary contribution of this paper is the introduction of a promising new approach that can provide significantly enhanced I/O performance to data-intensive applications in the general case, and to applications that must continually save state in particular. The importance of the work is further enhanced by it seamless integration

* This material is based upon work supported by the National Science Foundation under Grant No. 0702748.

into ROMIO [7]: a widely-used high-performance implementation of the MPI-IO standard developed at Argonne National Laboratory. This paper should be of interest to a large segment of the high-performance computing community given the importance of scalable I/O to the successful execution of HPC applications.

The remainder of the paper is organized as follows. Section II discusses the most significant related research. In section III, we cover file views and their significance with respect to object-based I/O. We introduce our Object-Based Caching System in section IV and our closely related Object-Based Files in section V. Section VI details our work with checkpointing in general, and specifically with the FLASH-IO benchmark. Section VII covers our experimental results. Finally, section VIII contains conclusions and future research directions.

II. Related Research

Our research draws on a large body of previous work in parallel I/O. In this section, we describe the most closely related efforts.

Thakur, *et al.* [11] introduced the abstract device interface for parallel I/O (ADIO) on which our system is based. ADIO abstracts away details about any particular filesystem, and presents a portable interface that can be used in implementing a wide variety of parallel I/O libraries including ROMIO [7].

A variety of work has focused on enhancing parallel I/O with caching or buffering. DAChe [12] is a user-space client side cache for high performance parallel I/O. Our work also employs caching, but we are caching objects while DAChe caches disk blocks. Active Buffering [13] aims to improve the performance of I/O write operations by using local buffering and performing the I/O in the background. We do not currently use threads, but we intend to take advantage of multithreading in the future.

Two-phase I/O [14] has been shown to increase parallel I/O performance by coordinating communication between I/O processors and reducing the number of small, fragmented disk accesses, instead sharing file data among processors and performing large contiguous disk accesses which are significantly faster. We have taken advantage of this idea in implementing our cache prefetch and flush operations.

III. File Views

An important feature of MPI-IO is the file view, which maps the relationship between the regions of a file that a process will access and the way those regions are laid out on disk. A process cannot "see" or access any file regions that are not in its file view, and the file view thus essentially maps a contiguous window onto the (perhaps) non-contiguous file regions in which the process will operate. If its data is stored on disk as it is defined in the file view, only a single I/O operation is required to move the data to and from the disk. However, if the data is stored non-contiguously on disk, multiple I/O operations are required.

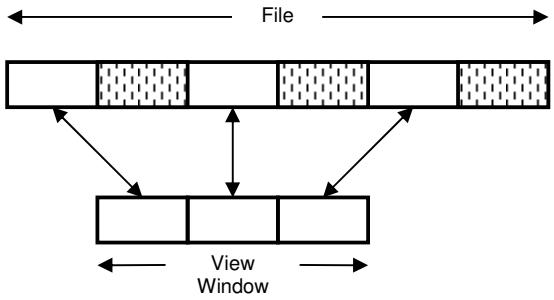

Figure 1. File view example. The shaded areas in the file are not visible to the application unless the file view is altered.

Figure 1 illustrates the effect of a file view. Regions of the file are mapped to a view window on a processor. Subsequent file accesses are performed with respect to the view window. File regions outside of the processor's view window are inaccessible to that processor unless the file views are changed (via a collective operation).

The file views contain valuable information regarding file access patterns, showing the file regions within which contention can occur, and by extension, those regions in which contention is not possible. In the next section we show how such information can be used to significantly improve parallel I/O performance.

IV. Object-Based Caching System

All of the processes that share a given file participate in the object cache for that file. The cache buffer consists of memory from the participating processes plus any available local disk space. The cache objects are created when a shared file is opened, and the objects and cache for that file are torn down when the file is closed. There is a local object manager for each process participating in the cache. Once the objects are created, they are distributed among the managers based on a cost model of assigning a given object to a given process. The local manager controls the meta-data for its objects and performs any object locking necessary to maintain global cache coherence. Once the objects are created, all subsequent I/O operations are carried out in the cache (except in the case of a sync() or close operation).

The way in which MPI objects are derived from the file views of the participating processes is shown in Figure 2. First, a collective operation sets the file view associated with each processor (a). Next the views are shared among all processors (b). The boundary offsets of each process's file view are collected into a global boundary list, which is sorted and duplicate boundaries are removed (c). Finally, an object is created for each pair of consecutive boundaries (d).

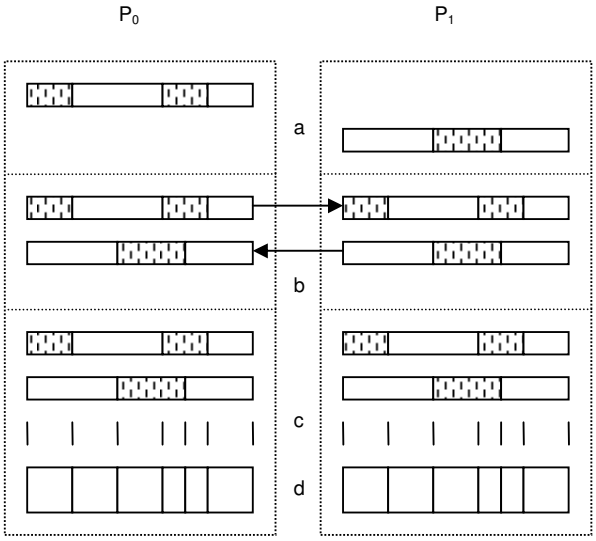

Figure 2. The object derivation process.

Once the objects are created, an additional step is taken to calculate the *reverse access set* for each object, which is simply a list of every processor that has access to that object according to the current file views. This is done efficiently using a *p*-bit bitmap, where *p* is the number of processors participating in the file access. If the bitmaps are arranged contiguously in memory, and each process sets its bit for every object it can access, then a single `MPI_Allreduce` operation is sufficient to share the entire collection of reverse access sets among all processors. The reverse access sets have two important uses. First, they determine to which processor a particular object should be assigned. Secondly, they show exactly which objects are shared between processes and which are private to a process (known as *shared-objects* and *private-objects* respectively).

This distinction between shared objects and private objects has two very important ramifications: First, only shared objects must be locked, and such objects represent the minimum overlap of shared data. This completely eliminates any false sharing and provides the maximum possible concurrency for data access. Second, such information can simplify the locking mechanism and significantly increase its performance. This is because each object manager knows exactly which processes can access its objects, and acts as a centralized lock manager for those processes. Thus contention for write locks, which can significantly reduce performance, is limited to the subset of processes that can access the objects being controlled by a given manager. In essence, this creates a set of centralized lock managers that are operating in parallel.

V. OBJECT-BASED FILES

Objects are created when a file is opened, and all subsequent I/O operations are carried out in the cache (except for sync() and close() which require the data to be written to disk). Performing all possible I/O operations in the cache can, in and of itself, provide tremendous performance gains, especially when the locking mechanism is simple and fast. However, even further gains are possible when the file data is stored on disk as objects because this allows each process to read/write its objects from/to the disk in a single I/O operation.

Because objects do not necessarily correspond to the traditional file model as a linear sequence of bytes, metadata is needed to be able to map between the object model and the traditional model. The structure of an object-based file is shown in Figure 3. The file contains a header section containing file metadata and a subsection for each participating process containing the data written by that process. The file metadata contains (at a minimum) the number of process data blocks in the file and the file offset to each of the blocks. It may also be useful to include file view information, though this is not essential. Each of the process data blocks includes a section of object metadata, followed by the data from the objects themselves. The object metadata includes the offset and size of each object in this file, as well as the object's offset in the flat file.

This functionality is implemented transparently to the user by treating object-based files as simply another file system upon which ROMIO is implemented. Thus the application can open an object-based file, manipulate the data, and then write it to disk as a flat file. Alternatively, it can open a flat file and then store it on disk as an object-based file. This flexibility is a result of the cache design. Cached data is stored as objects, but the metadata present in the cache allows the flat file to be reconstructed efficiently. Much like two-phase I/O, a subset of the processors are used as aggregators to arrange object data in file order, allowing disk writes to be efficiently performed on large contiguous blocks.

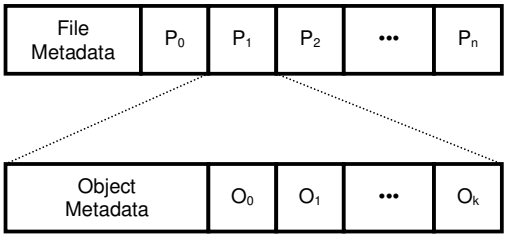

Figure 3. The structure of an object-based file. The upper rectangle represents the entire file, while the lower rectangle expands one of the processor blocks to show more detail.

VI. FLASH IO AND CHECKPOINTING

The FLASH [15] simulation allows the solution of fully-compressible, reactive hydrodynamic equations. It was developed to study nuclear flashes on the surfaces of neutron stars and white dwarfs. We designed our experiment to show the effectiveness of using an object-based cache in the checkpoint operations done by FLASH. As FLASH is implemented in Fortran using the HDF5 Library for I/O, we wrote a MPI-IO version in C using the same file layout and memory layout. Our FLASH/MPI benchmark uses MPI-IO operations to write checkpoint data from memory. Since our goal was to analyze I/O performance, we did not include any computation step or attempt to simulate one.

The principal data stored by FLASH consists of 80 three-dimensional blocks for each processor involved in the simulation. Each block, in turn, consists of 512 smaller sub-blocks, and the data contained in each sub-block consists of 24 variables of type `double`. A simplified version of the memory and file arrangements used by FLASH is shown in Figure 4. In memory, variables for each sub-block are stored together. The 512 sub-blocks comprising a block are also adjacent. In the file, however, the primary arrangement is by variable, so all of the variables V_0 from every block on every process are stored contiguously, followed by all of the V_1's, and so forth.

The objects created when the file is opened are shown in Figure 4 as dark rectangles. Each object contains all of the variables for a particular block on a particular process. Each object is 4096 bytes, and the file will contain 1920 objects for each processor involved in the run.

Note that while we can use MPI's strided datatypes to perform the writes from memory, it will require at least 24 individual MPI-IO operations to write the checkpoint file once. Our use of an object-based cache mitigates the need for most of these filesystem accesses, requiring only an initial filesystem read, and a single write operation each time the checkpoint is written to disk.

Our object-based system also provides an additional benefit over traditional block caching schemes since each object is categorized as shared or private based on which processes may access it. Access to shared objects in the cache requires that the objects be locked, while access to private objects requires no locking. In the case of FLASH-IO, all objects are private, thus the MPI-IO write operations are completely exempt from cache level locking.

Further benefit is achieved by writing the FLASH checkpoint file to disk in our object based file format. Instead of using two-phase I/O to reorder the data and write it to disk as a linear sequence of bytes, the cache objects and associated metadata are written to disk in a single contiguous write operation. This is advantageous not only during writing, but as will be seen, also when the checkpoint file is needed to restore program state.

VII. EXPERIMENTAL RESULTS

As noted above, the checkpointing of application state is becoming an increasingly important cost in large-scale, scientific applications. However, in the vast majority of cases the checkpointed data is never accessed. In such cases, the checkpointed data can be written to disk as objects thus significantly reducing the cost of creating such files. If it turns out that the data is subsequently needed, the object meta-data stored with the objects can be used to recreate the objects within an application or to convert the object-based file back to a flat file.

All experiments were performed using the Mercury cluster, at the National Center for Supercomputing Applications (NCSA). Mercury consists of 1,774 Itanium 2 processors connected with Myrinet and running SuSE Linux SLES 8. The filesystem used was the General Parallel File System (GPFS) developed by IBM. This filesystem is organized in a Network Shared Disk Server (NSD) configuration using 58 dedicated dual-processor 1.3 GHz Intel Itanium nodes. The GPFS Storage Area Network (SAN) also available on the Mercury cluster was

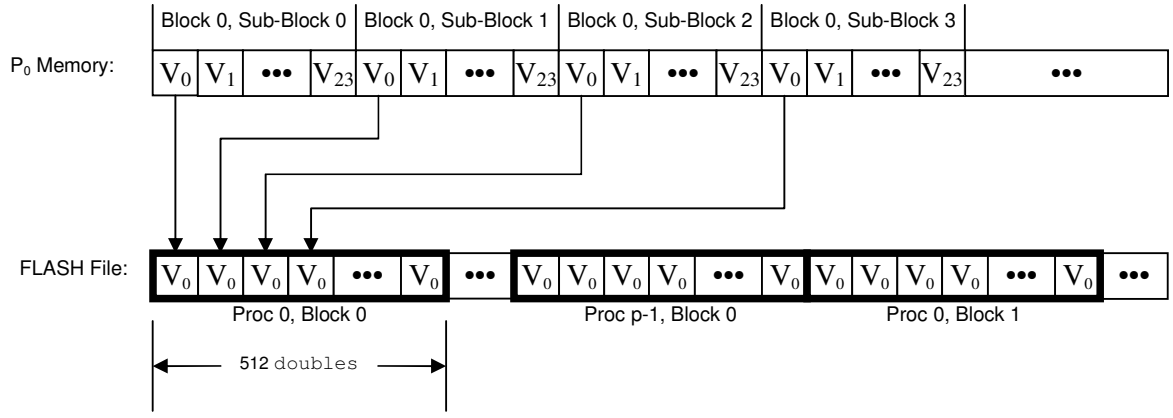

Figure 4. File and memory layout for FLASH I/O.

not used for these experiments.

We began with MPICH2 (version 1.0.3), and fitted its I/O subsystem (ROMIO) with our object-based cache implementation. Memory for a cache is allocated at file open, and the structure of the cache is calculated during the collective operation that sets file views. Low level filesystem reads and writes are replaced with corresponding cache read and write operations. Finally, the file synchronization function is modified to flush the cache contents to the filesystem

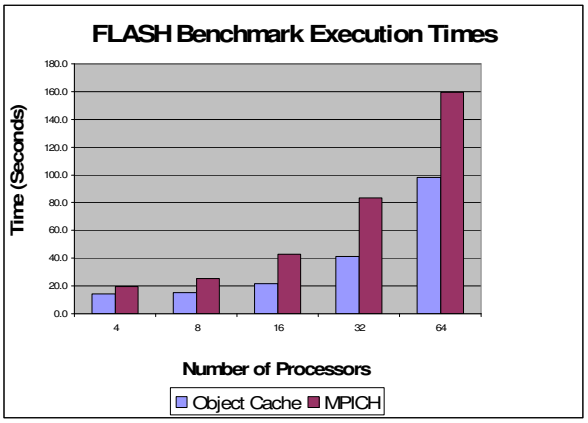

Figure 5. This figure shows the time required to produce checkpoint files as a function of the number of processes.

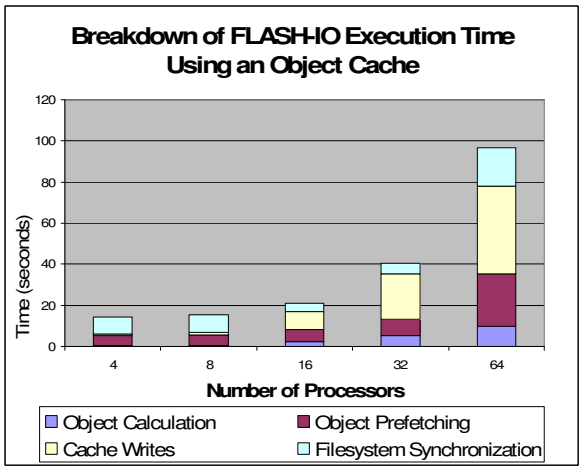

Figure 6. FLASH-IO performance breakdown.

A `checkpoint write` option was added, allowing the contents of the cache to be written by each processor independently to a special checkpoint file. A corresponding `checkpoint read` operation was also added to allow the cache to prefetch data from existing checkpoint files.

To determine cache write performance, we ran our FLASH/MPI benchmark first using the unmodified version of MPICH, and again using our cache enhanced version of MPICH with the new "checkpoint write" mode. Figure 5 shows the average time required to create checkpoint files using our object-based approach compared with unmodified MPI-IO, on between 4 and 64 processes. For 4, 8 and 16 processors, each run was repeated ten times, and the results averaged to produce the data shown. Since the resulting times showed very little variability, and since the runs become significantly more costly at each doubling of processors, tests for 32 and 64 processors were each run twice and the results averaged.

As can be seen, the object-based approach significantly enhanced performance, resulting in a 38% reduction in execution time with 64 processors. These are very encouraging results, and provide strong support to the notion that treating data as objects rather than as a flat file can offer tremendous performance improvements.

In Figure 6, we show the breakdown of creating the objects and writing them to disk. Also, since object calculation and prefetching are only required for the first checkpoint operation, cache performance for subsequent checkpoint operations would be further increased.

Reading of the checkpoint files was not a major concern since it is assumed to be a relatively rare event. However, to demonstrate feasibility, we implemented a routine to perform object prefetching directly from the checkpoint files. When executed on 64 processors, the prefetching time was *reduced* from 25.6 seconds using the FLASH file format to 5.5 seconds using individual checkpoint files. This improvement is largely due to the ability of each process to read its part of the checkpoint file independently as a contiguous block without inter-processor communication.

VIII. Conclusion and Future Research

In this paper we have introduced our object-based caching system for parallel I/O. We have shown that using the information in file views results in the efficient partitioning of file data into objects. We then showed that writing checkpoint files as objects can significantly improve performance when compared to MPI-IO.

Our future work will include optimizing the implementation of our current system. We also intend to extend the set of benchmarks and applications used to evaluate our approach. Work on our lock management system is ongoing, and we will report on its implementation and performance in future publications.

References

[1] Lustre: scalable, secure, robust, highly-available cluster file system. An offshoot of AFS, CODA, and Ext2., www.lustre.org

[2] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters. ," presented at Conference on File and Storage Technologies, IBM Almaden Research Center, San Jose, California.

[3] Panasas, http://www.panasas.com

[4] MPI-2: Extensions to the Message-Passing Interface, http://www.mpi-forum.org/docs/mpi2-report.pdf

[5] A. Ching, A. Choudhary, K. Coloma, W.-k. Liao, R. Ross, and W. Gropp, "Noncontiguous I/O Access Through MPI-IO," presented at the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03), 2003.

[6] R. Thakur, W. Gropp, and E. Lusk, "Optimizing Noncontiguous Accesses in MPI-IO," *Parallel Computing*, vol. 28, pp. 83-105, January 2002.

[7] R. Thakur, R. Ross, and W. Gropp, "Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation," Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Revised May 2004.

[8] R. Latham, R. Ross, and R. Thakur, "The impact of file systems on MPI-IO scalability," presented at the 11th European Parallel Virtural Machine and Message Passing Interface Users Group Meeting, 2004.

[9] P. Aarestad, A. Ching, G. Thiruvathukal, and A. Choudhary, "Scalable Approaches for Supporting MPI-IO Atomicity.," presented at 6th International Symposium on Cluster Computing and the Grid (CCGrid).

[10] MPI File Views. http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0/node184.htm

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

[12] K. Coloma, A. Choudhary, W.-k. Liao, L. Ward, and S. Tideman, "DACche: Direct Access Cache System for Parallel I/O," presented at International Supercomputer Conference, 2005.

[13] X. Ma, M. Winslett, J. Lee, and S. Yu, "Faster Collective Output through Active Buffering," presented at IDPDS 2002, 2002.

[14] R. Thakur and A. Choudhary, "An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays," *Scientific Programming*, vol. 5, pp. 301-317, 1996.

[15] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo, "FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes," *The Astrophysical Journal Supplement Series*, pp. 273-334, 2000.